GENDER AND PROGRAMMING: A DIFFERENCE IN STYLE?

Thesis submitted in accordance with the requirements of the University of Liverpool for the degree of Doctor in Philosophy by

Peter McKenna

March 2002

TEXT BOUND INTO THE SPINE

Preface

The work reported in this thesis is original and carried out solely by its author. Its conceptualization, design and implementation have been effected by the author, with the direction and assistance gratefully received from his supervisor, colleagues, and students. The analysis, synthesis, judgment, and critical ideas which contribute to the new knowledge presented by the thesis, are those of the author, based on critical reading, observation and experience. The third-party information contained within the thesis is derived either from published literature and data, or from data collection conducted by the author within the specified institutions of higher education. The American Psychological Association (APA) style for reference citations and reference lists has been used to acknowledge and clarify the use of all such information.

Signed:

_____

Peter McKenna

March 2002

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I wish to acknowledge the help and support of my partner Natalie Seeve, supervisor Janet Strivens, my sons Declan and Leo (for their computer games expertise, and enlightening me about Sonja Blade), the students and staff of Liverpool Hope University College and the Manchester Metropolitan University; and to dedicate the work to my daughter Simone.

0. Introduction

The deeper motivations for the research lie in a passion for equality, and a knowledge and experience of the under-representation of women, and over-representation of men, in the ever-important discipline of computer programming. As a parent and educator whose work teaching programming to undergraduates has coincided exactly with the trials of working parenthood - and of part-time PhD study - gender has been more present (or immanent) than ever.

A more particular trigger for the research has come from the background reading engaged in by a supposed computer 'scientist' from an Arts tradition - notably literature that crosses over from social science into computer science by examining the sociology of computing. At a general level, this research makes that crossover in reverse, but at the same time returns to familiar territory.

The idea which is central to the research, and which is brought into question by it, is that women and men program in particular ways that are different but of equal epistemological value. It was first advocated by science sociologist and psychoanalyst Sherry Turkle (1984), then developed in collaboration with Seymour Papert, the co-developer of LOGO at Massachussetts Institute of Technology (MIT), and LEGO Professor of Learning. Their analysis, along with the corresponding dichotomy between the 'soft' or 'concrete' and the 'hard' or 'abstract' styles of programming, has been subsequently accepted without serious question (Sutherland and Hoyles, 1988; Kvande and Rasmussen, 1989; Frenkel, 1990; Peltu, 1993; Grundy, 1996; Stepulevage and Plumeridge, 1998).

The ready acceptance of such a stylistic dichotomy may be due to the automatic resonance it has in traditional cultural perceptions of women and men: where the 'hard' or 'abstract' programmer uses abstraction, decomposition, structured programming and algorithmic reasoning, the 'soft' or 'concrete' stylist engages and communicates with the computer (and 'computer objects') in an artistic and holistic manner. Soft stylists, it is alleged, are naturally given to this more

human style, and, Turkle and Papert claim, are repelled by the coldness of the 'abstract' approach.

Turkle and Papert have sought to validate both the concrete style, and a polarised dichotomy between it and the 'canonical' abstract style. This gender mapping links back into wider theories: most notably to feminist epistemology, and by means of boldly generalised conflations with object relations theory and the ideas of Carol Gilligan (1982) and Evelyn Fox Keller (1983, 1985). The validity and consistency of such interdisciplinary linkages require particular examination in the light of a fuller understanding both of the concrete and the abstract in programming, and of the modulations of gender in computing culture. Initial doubts have also informed this direction of enquiry. In the wider gender perspective, there is an immediate concern that assumptions of inherent gender difference can be too easily made - that what is being explained has already been assumed; that such gendered styles may propagate stereotypes, linking women's capabilities once again to their 'nature', and valorising a hacker style that actually invokes detachment from the real world as a corollary of attachment to the computer. Concerns at a more specific level include the cross-relation of programming concepts - such as abstraction and objects - to their philosophical and psychoanalytical homonyms; and the possible confusion of learning strategies with programming strategies.

The question of 'styles' concerns the way in which programs are actually coded by individuals. While Turkle and Papert's emphasis is on an approach to coding rather than on the software product (a fact which is itself problematic, particularly because the notion of equal validity is so central), it is at least implied that the software produced by both styles will be equally good. The question is a serious one in that this hypothesis of gendered styles is widely accepted - and if it really were true, then equal access to programming-related education and employment would depend on teachers and trainers accepting the equal validity of the 'soft' approach. Turkle would be correct to suggest that giving the 'abstract' approach a privileged position results in the "exclusion" of women from computing (1992, p.3); and commentators such as Peltu (1993)

would have a reasonable case when they propose that structured programming be removed from the computing curriculum because it is inimical to women.

It is therefore also of the utmost practical importance that these claims be scrutinised rigorously and, ideally, subjected to more practical testing. The research is set in the context of the unequal representation of women and men in computing. The need for action to remedy this gender inequality in the defining technology of computer programming is particularly urgent: but if proposed action is not grounded in sound theory, then it runs the risk of actually compounding rather than redressing problems. Assumptions about gender and programming can be counterproductive if they are actually based on misconceptions and stereotypes. To present what appear to be experimental and concrete approaches to learning as a fully-fledged style of programming, is problematic enough: to identify such a style with women, runs the risk of marginalizing half of the population in a world where real programming projects simply cannot succeed without the tools of abstraction.

This is the spirit in which the research has been engaged. Such a thorough theoretical examination has not previously been attempted, and would therefore - regardless of the conclusions - result in the development of new knowledge. Insofar as the conclusion of the theory critique - and of the supporting survey research - confirms the preliminary concerns about the existing theory, then this knowledge would be more than incidental. Early findings have therefore been disseminated, by means of peer-reviewed journal publication, both in relation to the analysis (McKenna, 2000) and the data collection (McKenna, 2001); and variations on the analysis have also been presented to a computing audience - the primary audience - at a range of international conferences (McKenna, 1996; McKenna and Waraich, 2000a; McKenna and Waraich, 2000b).

This question has largely been ignored within the computing 'fraternity' on the one hand, and Turkle's analysis uncritically accepted by most of those within the field of gender studies on the other. It has been caught between disciplines - the superficial programming knowledge of sociologists, psychologists, and

gender theorists, matched by computer scientists' lack of understanding – where there is concern at all for the underlying issue of gender inequality - of gender discourse. The thesis intends to allocate equal emphasis and depth to each disciplinary component, in order to unravel the rationale of this hypothesis concerning gender and programming style.

The literature review therefore begins by providing a general background to the understanding of feminism and gender theory, difference and equality, along with a more particular focus on aspects which have a bearing on the psychological theory that underpins the concept of gendered programming styles. It will frame Turkle and Papert's programming style hypothesis as a product of a broader ideology, and provide the basis for a critique of that hypothesis. The second chapter deals with the literature on gender and computing, examining inequality in computing alongside explanations other than and complementary to those concerning programming styles, and how gender theory has influenced this literature. The importance of the gender and programming is also placed in the wider context of computing.

The third chapter sets out the literature on programming styles, focusing on the most important details and illustrations of Turkle and Papert's theory of gendered programming styles. Prior to the main critique of this theory, but with that theory now in mind, the relevant programming concepts of abstraction and black boxing are elucidated and explored in the fourth chapter. These concepts can then be compared in detail to the gender theory concepts that Turkle and Papert apply to programming, and the fifth chapter provides the major critique of their gendered programming styles theory, with the sixth chapter following the trajectory of Turkle and Papert's theory into the nature of object-oriented and visual programming. The seventh chapter discusses methodology issues and thesis structure, and in particular examines how to proceed as a result of the serious objections to Turkle and Papert's theory. The issues involved in devising an empirical test of their hypothesis are discussed, and a practical method developed to acquire data. The collection and analysis of data at two

Higher Education institutions is described in the eighth chapter. The ninth and final chapter summarises and concludes the research.

1. Sex, Gender: what's the difference? - A Literature Review

1.1. Introduction

Gender and feminist theory provides a context that is essential to an understanding of the research issue. In particular, the concepts of difference and equality lie at the heart of the research: do extrinsic differences in the representation of females and males in computing have intrinsic correlatives? Are there differences in the way women and men understand and relate to the world? What does actual 'equality' mean, and is it desirable? Most specifically, are such differences reflected in the programming styles and philosophies of females and males, and to an extent that fundamental changes in the approaches to the learning and teaching of programming are needed? Turkle's hypothesis is based on feminist theory in general, and feminist object relations theory in particular. This chapter will critically examine the fundamental concepts on which the questions above are premised, in order that the meaning and cultural resonance of those questions may first be clarified.

Feminist theory represents the only body of knowledge and literature that explicitly addresses gender. It is this that most clearly differentiates it from other discourses. While definitions of feminism are constraining, and it is as diverse as cultures and individuals, it is not without boundaries. In different forms, it questions assumptions across and at the core of a wide, interdisciplinary range of disciplines and contexts. Feminism is also, at least in principle, concerned with praxis and change, with a defining emphasis on advancing the position of women. It is a reflection of existing inequities that serious study of gender has been conducted largely by and about 'women'. This does not have to imply that masculinity is given and not in need of deconstruction: that men are agents, and women objects of study; that men are people, and women, women. The masculine bias embedded in dominant ideas distorts the masculine as well as the feminine. Men have 'gender', and so are no

more coherently defined than women. An examination of masculinities must also be important in any investigation into inequities, the more so when the context is traditionally 'masculine'.

The epistemologies and methodologies of feminism are the obvious starting-point for any analysis of gender differences. The notion of difference is central to feminist theory, and we will examine two intersecting debates centred around difference: the 'difference versus equality', and 'gender versus sex' debates. The extent to which equality can accommodate dichotomised differences – whether essential or extrinsic – constitutes important parameters for the consideration of inequality and change. The difference between socially determined 'gender' and biological 'sex' – and the extent of any uncertainty in between the two - is clearly important in qualifying overall differences. Gender is the benchmark of feminism and gender theory: the process of its 'discovery', along with its interpretative nuances, will be briefly outlined.

1.2. A brief timeline

The notions of sequence and category are useful to any summary, and in the case of feminist thought, can also provide a context for development and theoretical nuance – in this case, a context for the key themes that underpin the research question.

The efforts of women to reflect on and/or change their status and roles in society, are likely to embrace all of recorded history. For most of that history, however, the recorders and interpreters have been men, women have been largely excluded, and their interests often actively opposed. Discourse concerning women's condition, however, precedes explicit reference to gender, but nonetheless addresses implicit gender inequality. Miles (1993) refers to the (male) author of a 1505 book titled *Of the Nobility and Superiority of the Female Sex*, as a "proto-feminist". Goreau (1983) identifies Aphra Behn, a seventeenth century English novelist, as a "feminist" who expresses anger at discrimination. In the eighteenth century, Mary Wollstonecraft's *Vindication of the Rights of Woman* (1792) demanded citizenship at the time of the French

Revolution. Margaret Fuller has been described as the nineteenth century inspiration for American feminism (Urbanski, 1983). In broad terms, the European Enlightenment of the 18th century established the possibilities of destinies being both shaped and changed by human endeavour, and exposed to reasoned enquiry that which had previously been accepted. As a generalised intellectual 'grand narrative', the Enlightenment set the context for other metanarratives - such as feminism, the study of power and wealth, and the rationalist concepts of citizenship and equality.

The idea of a universal human nature and human rights is a foundation of feminism: that this idea found intellectual expression in the Enlightenment, has convinced subsequent western intellectuals (postmodern and liberal alike) that human rights are uniquely derived from that stage of their own culture. The endeavour of feminism may be said to be rooted in the struggles of ordinary people – across the world and throughout history - at least as much as in the academic taxonomies and analyses of western intellectuals.

The coming-out of modern western 'feminism' as an explicit 'theory', however, is often dated to the 1960s, and a period of 'latent' feminist theory is identified with the years between the changed social roles during the Second World War, and the 1960s (Delphy, 1993). However, the feminism of Simone de Beauvoir's *The Second Sex* (published in 1949) – while it is more analysis than activism - is far from latent. *The Second Sex* examined the position of woman in Western culture from an existentialist perspective. It revealed that position to be secondary in relation to men, and explained it in terms of social environment and control rather than in terms of nature.

> One is not born, but rather becomes, a woman. No biological, psychological, or economic fate determines the face [translation amended] that the human female presents in society; it is civilization as a whole that produces this creature, intermediate between male and eunuch, which is described as feminine. (1949/1954, p.295)

Inequality thereby produces profound psychosocial 'difference'. As we shall see, Beauvoir's existentialist philosophy essentially dismisses nature in favour of nurture, and contradicts much that subsequent feminists have had to say about differentiation in infancy.

1.2.1. 'First Wave'

Betty Friedan was a focus for the 'first wave' in the USA, with the publication of *The Feminine Mystique* in 1963, and the founding of a National Organization of Women in 1966. Friedan, sometimes referred to as the "mother" of feminism, is credited – in spite of Beauvoir's seminal work two decades previously - with exploding the myth of the happy housewife, and with identifying "the problem without a name". The focus of the 'first wave' was generally on identifying this problem and the enormity of it, and on the invisibility and omission of women. Landmark publications by Germaine Greer, Shulamith Firestone, and Kate Millett followed in 1970, which concentrated on the visible – the distorted representation of women - and identified the nameless problem as male power and malice.

1.2.2. 'Second Wave'

The concept of the personal as political led into the 'second wave', and a preoccupation with difference, as manifested and experienced in lived identity and psychology. A more studied 'gynocentrism' was said to be needed, in order to counter women's deeply-internalised image of themselves as lesser beings. One such focus has been on sexual relations – sexuality is theorised as something that is not just the neutral territory of pleasure and affection, but as characterised and defined by dominance and subservience (Rich, 1981; Dworkin, 1987). Power is exposed as the dynamic of sexual desire, and this domination is seen to define gender. The ambiguity contained within the word 'sex' is confirmed: gender and sexuality are intrinsically linked.

Within the context of all-defining male-supremacy, identity is "constituted by women's position as victim" (Nicholson 1997, p.148). Such a status can imply

struggle rather than passivity. A wide range of theorists explored positive aspects of identity which appeared to diminish the victim position, in a spirit of resistance. Part of this positive gynocentrism appeared to accept positions denied by materialist theorists: Chodorow's positive aspects of difference included the psychology of motherhood (1978), and an object relations theory that credits girls with natural relational competence because they do not have to repress their bond and identification with the mother. With the second wave's emphasis on the subjective and personal, psychoanalysis presented as a useful theoretical tool:

> The centrality of sex and gender in the categories of psychoanalysis, coupled with the tenacity, emotional centrality and sweeping power in our lives of our sense of gendered self, made psychoanalysis a particularly apposite source of feminist theorizing. (Chodorow, 1989)

Object relations theory - essentially a development of Freudian psychoanalytic theory - holds that relationships, beginning with the mother-infant dyad in the first year of infant life, are primary to an individual's identity and sense of personal boundaries in relation to others. The notion of an 'object' represents anything that is internalised as a psychological structure, particularly those formed through early experiences. (This concept will be particularly important to an understanding of the research question). According to Klein (1946), this process of "introjection" leads to the splitting off, and projection into another person, of parts of the self. The classic example of this process – and the one on which gender difference (and hence feminist object relations theory) is often premised – is that of the infant needing to feed: the feeder (and first significant object) is assumed to be the mother (the primacy of the mother-infant dyad is implicitly interpreted as natural rather than socially constructed). The hungry infant splits off and projects its longing and rage into the mother. In order to make itself whole again, the infant must identify itself with the mother; as the nature of identification is assumed to be profoundly gendered, only female infants can do this. Males – and females whose mothers have not consistently accepted their projected feelings - will continue to be repressed in adulthood.

The feminist perspective on this theory focuses on relations as crucial to an understanding of sex, gender, and male dominance. Instead of the traditional Freudian father-son relationship, however, it is the pre-oedipal mother-child relationship that is posited as definitive. Benjamin (1995) sees acceptance of the primacy of this relationship, and of the mother's subjectivity within it, as a prerequisite for seeing the world "as inhabited by equal subjects". However, the theory's feminist properties are frequently ambivalent - an ambivalence perhaps epitomised in its apparently essentialist reassertion of the primary role of the mother. Chodorow and others may examine the consequences of mothering as difference, but not the premise of it.

Object relations theory lends itself to the idea – one that permeates much of second-wave feminism, and is important to the research question - that there *is* a true female nature, and that women possess innate characteristics that are devalued by men. Object relations theory ensures that this 'nature' cleaves to traditional characteristics: a key characteristic is "a capacity to nurture and care for others" (Evans 1983, p.356). This 'capacity' is celebrated as distinctively feminine (counterposed by an equivalent masculine incapacity), and is frequently conflated with biological rather than social determinants. It might conceivably be considered part of the traditional femininity that Beauvoir criticised as a socially-determined revelling in "immanence". Delphy criticises Segal's (1999) desire to "hold on to what being a woman means, while contesting the cultural and social meanings given to 'femininity'", as a contradiction in terms (Delphy 2000, p.160). The revaluing of existing or observed behaviours does not have to imply acceptance of them as the inherent characteristics of women and men – they can constitute a gender resource rather than a destiny, and a desideratum for males. However, as long as 'caring' is interpreted as natural rather than learned behaviour for females and not for males, there would appear to be little prospect of equality in this respect. Where traditional feminine behaviours are explicitly identified and celebrated as natural and intrinsic, they serve to determine (rather than be determined by) and to reinforce traditional social and economic roles. Where they are seen as natural, any vision for change and equality will inevitably be constrained.

1.2.2.1. Standpoints

'Standpoints' feminism represents a deepening of the emphasis on difference and personal, gynocentric experience, presenting extremity of difference as an epistemology. As such – given the emphasis placed on gendered differences in epistemological and programming styles - it deserves special attention.

In advancing the notion of a "feminist standpoint" as "an important epistemological tool", Hartsock (1983/1997, p.216) adapts the Marxist idea of a unique proletarian 'standpoint' or 'correct' class position. Viewing waged labour as "activity more characteristic of males in capitalism", she argues that "the position of women is structurally different from that of men, and that the lived realities of women's lives are profoundly different from those of men" (p.217). It is "women's activity as contributors to subsistence and as mothers", as producers of people and "producers of goods in the home", that determines the sexual division of labour. Standpoint epistemology is structured in terms of "a duality of levels of reality" (p.218), and a woman's reality represents "the deeper reality". A woman's 'standpoint' is not just an interested position – it is an "engaged" position. Women's lives offer a "privileged vantage point on male supremacy" (p.217)

In making this analysis, Hartsock pursues at least an element of universalism – a belief in common characteristics shared by all women. She uses a quotation from Marilyn French's *The Women's Room* (1977) to assert that the unique feminist standpoint is available "to even non-working-class women" – that "women's work in housekeeping involves a repetitious cleaning" such as the cleaning of toilet bowls used by males (p.224) – whom Hartsock designates as the "ruling gender" (p.218). While women have "constant contact with material necessity", the male lives a life "at the furthest distance from contact with concrete material life" (p.224). This appears to differ only in interpretation from Beauvoir's earlier perception that women were taught "patience and passivity" by the waiting on and obedience to the material elements of domestic duty.

There are undertones of essentialism – the idea that there is a fundamental feminine nature or essence - in the emphatic and even inextricable linking of housework and childcare to women. There is also an implication that the sacrifice of domestic inequity may redeem women by endowing them with special insights. This adds an element that resembles religious fervour – encompassing duty, sacrifice, and spiritual reward. The division of labour facilitates "an intensification of class consciousness" that is analogous to the perspective afforded the proletariat in Marxist theory (Hartsock, 1983/1997). The revolutionary, however, emphasises even small acts of revolution at the same time as examining actual conditions. The correctness or truth of the proletarian's perspective is only a means towards an end. With Hartsock's standpoint, in the absence of any consequent ideology (analogous to the Marxist dictatorship of the proletariat) or manifesto for action, it is the standpoint itself that becomes its own manifesto. As such, it can appear to reclaim rather than challenge the division of labour.

The key to women's experience being more valuable than men's is repeatedly expressed in terms of what may be perceived as social role rather than intrinsic female characteristics:

> ..the vantage point available to women on the basis of their contribution to subsistence represents a deepening of the materialist world view and consciousness... (Hartsock 1983/1997, p.165)

Hartsock distinguishes between the invariant and the changeable, suggesting that the "fact" that "women and not men rear children ...is clearly a societal choice"; however, she goes on to conjoin childbearing and childrearing as part of a production process:

> Women's activity as institutionalised has a double aspect – their contribution to subsistence, and their contribution to childrearing. Whether or not all of us do both, women as a sex are institutionally responsible for producing both goods and human beings and all women are forced to become the kinds of people who can do both. (p.222)

Although qualified by reference to its institutionalisation, this conflation of women's activity with their biology is an important determinant of the feminist

standpoint. The greater depth of a woman's perspective is explained not just in terms of institutionalised lived reality, but also in terms of "nature" in general, and childbirth in particular: "The female experience in reproduction represents a unity with nature which goes beyond the proletarian experience of interchange with nature" (p.225). Childbirth appears to merge into childcare in changing the consciousness of the woman, and enhancing her vision.

Hartsock calls upon object relations theory to identify women's experience as a unique and truer reflection of reality. While this unique achievement of non-conflictual individuation is general to the entire 'class' of women, Standpoints feminism further ascribes special value to individual experience. Certain lived-experiences are valued above others: those outside the perceived western mainstream - often related to experience of racism and/or sexual orientation - are of particular value in describing and revealing reality. Their authenticity depends entirely on the subjective accounting of experience. Writers such as Andrea Dworkin have emphasised the primacy of their own personal experience, and their right to assert reality. (It is therefore perhaps paradoxical that Dworkin should be accused of insisting that all women will or should experience certain things in given ways (Grant, 1992)). This represents a privileging of distinctive subjectivity, not just a philosophical reliance on subjectivity: as such, it may imply solipsism, and certainly raises questions as to what possibilities exist for communicating knowledge if it is unique and hidden, and premised on individual experience.

The instantiation of the individual's own experience - or own class or group – for universal experience is not, however, unique to second wave feminism and its emphasis on the personal. hooks (1984, p.2), for example, criticises Friedan's identification of "the problem with no name" as a description of "the plight of a select group of college-educated, middle and upper class, married white women-housewives" rather than of women. Even in a postmodern age, theorists can often fall back on universalised identities that are in fact derived from their own experience within narrow western academic circles: Irigaray

asks (in rhetorical good faith), "What woman has not read *The Second Sex*?" (Rodgers 1998, p.74).

Whatever its unconscious presumptions about the universal, the women's movement in the 1970s did aspire towards a universalising of a common women's experience. Clearly this concealed the many differences between different women's experiences: as hooks (1984, p.2) says, Friedan "did not speak of the needs of women without men, without children, without homes. She ignored the existence of all non-white women and poor white women. She did not tell readers whether it was more fulfilling to be a maid, a babysitter, a factory worker, a clerk, or a prostitute, than to be a leisure class housewife." Yet an implication of Standpoints is that some women - if, for example, they are not mothers (or even not traditional mothers) - can only have a less valid understanding of the 'essential' position of women, and of reality itself. Barrett and Phillips (1992) identify an alternative focus on "the active creation and recreations of women's needs or concerns" (p.6), but wonder whether "such developments leave feminists with nothing general to say" (p.7). If there are no universals, there may well be nothing general to say: just individual and personal experience and testimony. If difference is such that access to another's position is denied, the possibilities of enlightenment and therefore progress would appear to be restricted.

1.2.2.2. Gilligan and difference

Difference informs the work of Carol Gilligan: she embraces difference between two universalised gender experiences, and roots it in object relations theory. Her themes of female attachment and male attachment are defining influences on Turkle's work. The idea that women see the world in a categorically different way from men, is elaborated by the fact that the male view is regarded as the orthodox default, and universalised as human rather than male. Gilligan (1982) exposes male bias in academic work and ethical theory in terms of object relations theory: that which is accepted as normative actually reflects the male side of object relations. The male model of individuation and distance from the object, is privileged at the expense of

women's relating and caring. Gilligan accepts the underlying dichotomy, but simply revaluates the female model. The basic premise of Freudian theory is accepted –with negative characterisations of female psychology (such as weaker ego boundaries) eclipsed by positive ones (such as stronger empathy).

Gilligan concurs with Chodorow's attribution of "certain general and nearly universal differences that characterize masculine and feminine personality and roles" to "the fact that women, universally, are largely responsible for early childcare" (cited in Gilligan 1982, p.200). Her agenda is not to question or change these characteristic differences, but primarily to present a more gynocentric (or at least non-masculine) perspective on them. In doing so, it may be that she is at least partly guided by prior assumptions about characteristics and subjective experience, into self-fulfilling observations of objective phenomena. The fact that, in game observation, boys were able to resolve disputes more effectively than girls, is not explained in terms of negotiational skills, but rather in terms of the boys' obsession with "legal elaboration of rules", against the girls' "more tolerant" and "pragmatic" attitude to rules. The girls are said to have "subordinated the continuation of the game to the continuation of relationships" (p.202).

Gilligan even accepts the notion of "moral weakness" ascribed to women by the androcentric perspectives of Freud and Piaget – but again revaluates it as positive. This stereotypical "deference" by women to others, is said to be

> rooted not only in their social subordination but also in the substance of their moral concern. Sensitivity to the needs of others and the assumption of responsibility for taking care lead women to attend to voices other than their own and to include in their judgment other points of view. Women's moral weakness, manifest in an apparent diffusion and confusion of judgment, is thus inseparable from women's moral strength, an overriding concern with relationships and responsibilities. (Gilligan, pp.206-207)

These revaluations represent a corrective to androcentric perspectives. On the other hand, in celebrating the value of feminine 'characteristics', many of the assumptions that underpin those perspectives are still accepted. Gilligan is

concerned with "women's place in man's life cycle" – a place which she says "has been that of nurturer, caretaker, and helpmate, the weaver of those networks of relationships on which she in turn relies" (p.207). Women have "taken care of men". The importance of care, she argues, has been diminished because it has been regarded as "intuitive" to women, "a function of anatomy coupled with destiny". However, it is only the degree of importance, not the natural intuition, that is in dispute. And when she says that women define themselves "in a context of human relationships", and judge themselves "in terms of their ability to care" (p.207), it is seen as fundamental to fulfilling the essential role of woman in the human lifecycle.

Gilligan's work is intended as a corrective to the apparently androcentric assumptions made by Freud, Piaget, and Kohlberg, concerning moral development. The traditional concept of moral maturity – as taken for granted by society as well as these theorists – reflects the individuation that is said to characterise male development, rather than the cooperation and care that is also said to characterise female development. Masculine morality is said to be oriented around rules for legalistic individual rights – which emphasise "separation rather than connection". Women's moral decisions, she claims, are driven by "caring" considerations rather than rules. Their "conflicting responsibilities" and relationships, and moral problems manifest "a mode of thinking that is contextual and narrative rather than formal and abstract" (p.208).

While this critique accepts many of the assumptions made by Freud and Piaget, the difference – which characterises Gilligan's work as 'feminist' – is that she disputes their designation of 'feminine' moral development as something that is appropriate to the home, but deficient in public life. She agrees that women's morality "appears inconclusive and diffuse", but values the "insistent contextual relativism" that underlies it. Gilligan makes no claims for true universality – just for universality within the respective gender of the separate masculine and feminine perspectives; her claim appears to be for the valorisation of traditional feminine perspectives (like caring) as different but equal. This is crucial – at

least by analogy - in the context of the current research, as is her conclusion that the canonical "morality of rights and non-interference may appear frightening to women in its potential justification of indifference and unconcern" (p.210).

This intra-gender universality appears to be premised on prior assumptions – and material from Freud, Piaget, and Kohlberg is selected to illustrate those assumptions. From ancient Greek myth to Freud, ideas and symbols are drawn to, and coalesce around, this seemingly inevitable dichotomy of gender. In particular, Gilligan (1982) appears to extrapolate generality from one male and one female interviewee of Kohlberg: yet the questions asked of each appear to be different – those asked of the man inviting an 'objective' response, and those asked of the woman inviting an equally 'subjective' response. The universal masculine perspective is characterised as one of "non-interference" on the basis of the man's reference to "not interfering" with the rights of others. Ultimately she references an *a priori* female psychology "that has consistently been described as distinctive in its greater orientation toward relationships and interdependence" (p.210). There is also no consideration of variables other than gender – the responses to the questions about morality clearly differ in terms of their political perspective, and there could conceivably be variations in the expressive discourse that are interpreted as more substantial variations in meaning.

Valorisation of the traditionally feminine does not necessarily remove subordination; indeed, some of the characteristics are said to be actually rooted in social subordination. Gilligan does not take into account the existence of group-based ideologies, cultures and societies which value cooperation, but which are also male-dominated. The assumed perspective is western-bourgeois – perhaps consequent upon the emphasis on subjectivity and concrete context.

While 'difference' is explained as a product of the given that "the primary caretaker in the first three years of life is typically female" (Gilligan, p.200), there are – as is often the case - no indicators of change. Few theorists speculate about alternative object relations, where children are brought up either by men,

or by both sexes equally. Shared is frequently assumed to be maternally-delegated parenting. Pruett (1987) indicates that children raised by fathers thrive, but accounts for this in terms of the input of two parents; and as Lamb (1987) points out, such studies are conducted in situations where the father chooses his role. Such studies do not address the impact on traditional object relations theory, and few point to a future where heterosexual partners might actually discuss which of them will care for their children during these years. Indeed, some explicitly wish to retain the assumptions of such responsibility – to exclude men from "child care and home life", and thereby to implicitly accept and maintain at least one half of the traditional division of labour (Cockburn, 1985). And Gilligan declares that "the continuation of the mother-daughter relationship" is "in some mysterious way" tied to the very "fertility of the earth", viewing the human lifecycle as arising from "an alternation between the world of women and that of men" (Gilligan 1982, p.211). This would seem to be an essentialist perspective, recognising and valorising characteristics that have been hitherto unrecognised and devalued, while offering the benefits of enhanced connectedness instead of an ideology of empirical change.

1.3. Difference and Equality

The concept of sexual 'difference' is therefore of pivotal importance to feminist theory. Indeed, Irigaray (1984/1993) has claimed that it is *the* question of our time – one which would be our salvation if we thought it through. It is also central to the research question, which is concerned with difference and equality.

We have seen the distinction made between those who emphasise difference (and, usually, reposition perceived or given female cultures in terms of relative value), and those who emphasise equality (with a broad predisposition to a 'sameness' that is sometimes criticised as masculine) and rights. While the so-called "first wave" of feminism was primarily concerned with equality, the concerns of the "second wave" were more with difference (Humm, p.11) – a difference more fundamental than, and autonomous from, other differences.

Where 'first wave' feminism was concerned with the object, 'second wave' feminists gave expression to the subject: the first was preoccupied with materialism, the second with materiality. Second-wave feminism expresses identities and standpoints, and to do so frequently adopts and adapts concepts and techniques (exemplified by object relations theory) from psychoanalysis. This accentuates essential (rather than material) differences – between women and men, and between different women:

> "Currently feminism attacks universalism and describes and celebrates the experiences and identities of many 'different women'" (Humm, p.193).

Second-wave feminists frequently position themselves as inheritors of the first-wave, building upon the material achievements of the former. While the value of perception and experience is undoubtedly compelling and of the utmost importance, these subjectivities were frequently more than important: they defined theory. More recent materialists such as Delphy suspect that reliance on psychoanalysis implies belief in "some trans-historical, unchangeable bedrock of human nature, whose mechanisms of 'identification' and 'desire' are impermeable to social construction" (Delphy, 2000).

Butler (1993) criticises the general use of psychoanalysis in feminist theory to theorise sexual difference "as a distinct and fundamental set of linguistic and cultural relations" (p.167), relations that assume and embody a heterosexualised symbolic and "racial injunctions". She suggests that gender difference is most often a deterministic framework.

When difference is foregrounded, therefore, the reality of 'equality' can be hard to confirm: continued subordination can sometimes be read into the acceptance of difference – as with Gilligan's female protector and modifier of man's lifecycle. The corrective and protective role of smoothing over hard masculine edges is recognisable in traditional and stereotyped gender relations. Moreover, Gilligan's identification of rights with a masculine perspective on morality, implicitly undercuts the pursuit of equality: instead, women and men have their own "worlds" – worlds which alternate but do not meet, save for "woman's

place" as a corrective to man's lifecycle. This role is itself described in traditional terms – woman will continue to "protect" the female recognition of the importance of attachment, while orthodoxy continues to intone "the celebration of separation, autonomy, individuation, and natural rights" (p.211). As such it provides an essentialist echo of what might have been assumed to be just a description of an historical or actually-existing role of nurturer and helpmate.

1.3.1. Equality versus Difference

The embracing of subjectivity can be interpreted as a move away from rationality and objectivity - abstractions which the new emphasis on 'difference' often characterised as 'masculine'. Although, as Meijer (1991) points out, difference and equality may be based on each other, theorists such as Kristeva and Irigaray reject equality in favour of difference. Kristeva – who appears to have repudiated feminism as such but is regarded as feminist by Anglo-American feminists, and frequently cited and included in 'second-wave' feminist readers – explicitly rejects the pursuit of material equality in what they see as 'masculine' territory (Oliver, 1993). For Kristeva, the 'power' of public equality represents "phallic power", and its pursuit a distortion of women's true nature (Kristeva & Oliver, 1997). Indeed, she conjoins the epithet "phallic" to the term "equality". Difference is desirable, and equality denies difference. The 'first wave', to her, was attuned to the 'masculine'. Second-wave theorists can, for example, equate Beauvoir's perspective as "masculine" because she studiously refused to appropriate the "feminine" on the basis that it connotes subordination. Evans (1983) sees "rationality, personal autonomy and independence" as masculine characteristics, and Beauvoir's entire argument as one "for women to become rather more like men".

Materialist feminists on the other hand, set about deconstructing the notion of sex difference, and in so doing expose the inadequacy of accepting it. In the words of the Editorial Collective of *Questions Féministes* (1977), women need "access to the neuter, the general"; to "reclaim for ourselves all human potentials, including those unduly established as masculine".

The feminist perspective on equality is therefore often filtered through a wariness of identifying 'sameness' with a dominant masculine default. Yet this domination of the masculine can also play a part in actually defining the 'difference' that some feminists accept and celebrate. Resolving the coexistence of difference and equality is difficult; yet the difference is often ascribed to women, and also seen to actually define 'equality': in Peggy Seeger's words, 'different therefore equal' (1979). A relationship between supposed gender differences, and their (equally supposed) social and political manifestations as inequality, implies that inequality is determined by the characteristics of women and men. The position represented by Seeger – in common with the conservative position of established religion – not only accepts these differences, but moreover actually celebrates them in order that they might be regarded more equally.

Humm (1992) uncritically poses a second-wave question that also contains some of these assumptions: "do we need a room of our own as well as more space in the boy's locker room?". The reclamation of what had seemed like socially imposed and entrenched gender stereotypes during the first wave, led to what some would regard as a deterministic ceding of the material, public arena to the masculine. The view of public life and the workplace as "the boy's locker room" is often accompanied by a retreat into 'the women's room'. Traditional feminine practices are frequently valorised and celebrated (and perpetuated) as distinctively female experience and culture.

Kristeva's rejection of equality as "phallic power" (and of any human potentials characterised by Delphy as "unduly established as masculine"), casts difference and equality as competing claims. Claims for difference frequently do not challenge roles and stereotypes that manifest the difference. Many feminist academics (in common with popular contemporary women's publications) seek out difference in order to celebrate it.

The ideological drive for reassuring difference can therefore mean that the empirical case for there being a difference is left unexamined. For example, Herrmann's study of 'garage sales' in the USA (1996) declares that women are "empowered" by garage sales because they "valorise" their "housekeeping and shopping skills". The study identifies garage sales as an area for feminist research on the grounds that 66% of participants are women, and it does not seek to make any comparisons with male participants. Nor is there any notion that role-based traditional 'abilities' could be counterbalanced by traditional 'inabilities'. Other feminists seek to link given social roles to women's biology: Rossi's argument (1970) that female nurturance is innate (rather than a circumstance of male hegemony), is based, like Herrmann's study, on studies only of women (and in this case only mothers).

Kristeva in any case attacks (as masculine) the whole notion of building arguments on the basis of empirical evidence. Phenomena such as language itself are thereby rendered into 'objects', in accordance with the masculine agenda of distance from, and control of, the 'other' (Sellers 1991, p.48). She identifies the force of motherhood, and the web of female interrelations – expressed through women's bodies – with *chora*, a metaphysical term from Plato's Timaeus. The *chora* has no substance, but is permanent; associated with nourishment, it is primordially female in nature – a positive version, perhaps, of Beauvoir's negative 'immanence'. This primal 'space' has prior existence, has no form, and cannot be perceived by reason. The essentialist argument for extreme difference thereby proposes different epistemologies based on sex, while at the same time closing itself off from debate.

Within the framework of difference, technology is frequently examined as a means of enhancing the performance of 'women's work', and proposals for sometimes radical change often focus on revolutionising technological organisation so that it facilitates traditional roles, rather than on subverting the underlying social division of labour. Kramarae (1988), for example, thinks that public transport should be planned in a way that enables women to fulfil the tasks 'expected' of them: "obtaining medical and dental care for family

members, transporting children to school and social activities, carting groceries" (p.8).

Such analyses imply an acceptance and even promotion of predetermined gender difference. Scott (1988) provides a practical illustration of the equality versus difference debate, as enacted in a sex discrimination complaint brought in the USA by the Equal Employment Opportunities Commission against Sears. Scott recognises that although 'difference' and 'equality' do not mean the same thing, there can be a symbiotic relationship between the two ideas – and that 'difference' can be reframed as constant 'differences' which are "the very meaning of equality itself". Traditional bipolar 'difference', which naturalises the traditional roles of women and men, obscures these 'differences' at the same time as it fixes rather than challenges gender identity. In rejecting singular difference, she suggests that the antithesis of feminine and masculine hides an interdependence, and that in that interdependence one term *must* be dominant or prior, and its opposite subordinate and secondary. The terms 'woman' and 'man' should be open to scrutiny, rather than allowed to define each other.

'Equality' is concerned with challenging the gendering of power. Bem (1993) qualifies the common reference to *male power* as signifying "the power historically held by rich, white, heterosexual men" (p.3), while other feminists – commonly associated with the epithet 'radical' - would identify 'men' as a ruling class. While the concept of 'power' broadly signifies ownership and overall control, the concept of feminine 'empowerment' (as typically used in the garage-sales study) often denotes control within traditionally feminine aspects of interpersonal interactions and domestic decision-making - often within an overall environment of material and traditionally 'public' disempowerment.

One might interpret such a use of 'empowerment' as a further manifestation of a movement towards localised and personal notions of power – which itself is interpretable as a 'feminised' inclination. Feminist academic adaptations of post-structuralism and post-modernism have also led to a movement away from

larger theory – or grand narratives that attempt to explain everything - and towards local studies.

The emphasis on difference is likely to be premised on an underlying acceptance of a primary and profound dichotomy between female and male. The traditional first-wave feminist analysis of the male default – "man represents both the positive and the neutral...whereas woman represents only the negative, defined by limiting criteria" (Beauvoir, 1949/1954) – was taken and transformed by virtue of the celebration of the 'difference' as positive rather than neutral or negative. The challenge of the underlying dichotomy is marginalised. It may be the 'becoming' of Beauvoir's famous axiom, that is celebrated, and augmented by appeals to nature and biology. The 'women's room' may be a place – different from, but in the likeness of the locker room – where strategy can be decided and morale lifted, rather than a locus of biological determinism. The main shift in emphasis, however, was away from the notion that the feminine is shaped by society - "it is civilisation as a whole that produces this creature" (Beauvoir 1949/1954, p.295) – and towards subjective identity. To identify differences between men and women as products of gender identities rather than of biologies, however, validates the primary dichotomy, and at the same time leaves open the question of how those identities are produced.

1.3.3. Poststructuralism

Notwithstanding the androcentrism and misogyny within Freud and Lacan, post-structuralism's emphasis on external structures such as gender, combined with an emphasis on psychoanalysis, has presented as a theoretical framework attractive to feminism. Lacan's (1966) view of women as essentially driven by a desire to be wanted (rather than loved or fully known) – of women existing for men - may translate into an immutable reduction in personhood and self-determination, a location of being in others rather than in the self, and of power in an image rather than in actions. Yet versions of these characteristics fit with female identities that have been celebrated within feminism; and Lacan's emphasis on the subject - distinct from the ego and linked to the symbolic - and

his elaboration of object relations theory, appear to appeal to a localised and essentialist interpretation of feminism.

Lacan (1966) and Levi-Strauss (1966, 1969) hold sex difference to be fundamental to psychoanalysis and cultural deconstruction respectively. Lacan repositions the phallus as the cultural symbol of masculinity which forms the individual and rules language. Such structuralist and poststructuralist notions appear to echo object relations theory in terms of separation from the mother and the experience of absence as a child's first object (the mother, of course, experiences the child as 'missing' phallus) (Lechte, 1994, p.69). Again, many feminists have simply revalued rather than challenged these factors. The positive-negative polarity of the cultural phallus and the biological penis in relation to the feminine-maternal is simply reversed. The conflation of the feminine and the maternal is thereby not assumed to be problematic. While Lacan's conceptualisation of woman as some sort of not-man is fundamentally masculine, feminist critics such as Irigaray (1974/1985), point out this masculine character, but reverse the polarities and take the feminine side. Difference, albeit positive, might thereby appear to be defined in terms of a masculine default. Within this influential Lacanian perspective, woman is seen as sign rather than person: the generality of the masculine and the particularity of the feminine is inherent in Lacanian terminology. Irigaray identifies the Freudian masculinist assumptions (as exemplified by Lacan's phallus-centred world) that have been incorporated into post-Freudian psychoanalytical theory, and extends the terminology, adding the labia as a signifier of plurality to the penis as signifier of unity (Irigaray, in Cameron, 1992, p.171; and in Sellers, 1991). The dominant way of expressing meanings is seen as imbued with masculinity. The implicit alternative is that of female communality versus a male individuality that is often dysfunctional. Proximity and difference in relation to the 'other' is a recurring theme in Irigaray's discussion of gender 'difference'.

The French materialist feminist Cixous (1994) has identified the extent to which Freudian and post-Freudian discourse relies on its own assumptions

out sex. Cixous sees the discourse as reductive, and determined by cultural models of biology. The terms 'masculinity' and 'femininity' are, to her, politically and culturally determined markers, between which people of both sexes fluctuate. Patriarchal law defines, and appropriates, gender difference. As Delphy (2000, p.162) observes, "biology is the contemporary name of nature".

Butler (1987, 1993), critical of the essentialism implicit in the distinction and emphasis given to sexual difference within psychoanalysis, sees gender as performance, and convention as repeated performance. Gender is something that people do, rather than something that is. Butler refers to Derrida's focus on "those ostensibly 'structural' features of the performative that persist quite apart from any and all social contexts" (1993, p.188), but suggests that, as performance, gender is not rigid, and is rather constantly open to new signification.

Wearing (1984) argues that motherhood is an ideology, an allocation of responsibility that is crucial in determining the division of both domestic and social labour – and, thereby, gender relations. She cites Oakley's identification of women's instinct and propensity for childcare as a myth, and elaborates on this myth as a construct and an ideology that is generated and communicated by official and semi-official 'caretakers'.

Post-structuralist analyses based on the work of Derrida as well as Foucault, claim to identify a common base for both the minimisers and the maximisers of 'difference'. In psychology the two positions are referred to as alpha and beta bias (over- and under-estimation of gender difference), and Hare-Mustin and Maracek (1988) conclude that:

> alpha and beta bias have similar assumptive frameworks despite their diverse emphases. Both take the male as the standard of comparison. Both construct gender as attributes of individuals, not as the ongoing relations of men and women. Neither effectively challenges the gender hierarchy, and ultimately neither transcends the status quo (p.69).

"Paradoxes arise", Hare-Mustin and Maracek maintain, "because every representation conceals at the same time it reveals". A focus on gender differences "marginalizes and obscures the interrelatedness of women and men, as well as the restricted opportunities of both", and can also obscure institutional and structural factors. Beta-bias, on the other hand, merely affirms the male default (1988, p.462).

The multiplication of differences within gender identities appears to reach beyond simple alpha-bias. However, while Derrida has explicitly criticised the 'essentialism' of 1970s feminism, this position may in turn be criticised for an overemphasis on local differences that has actually slowed down or impeded broader progress (Gallop, 1997, p.17). It may also be argued that the shift from binary dichotomy to degrees of granularity of difference, and intra-gender difference, has indeed – as Barrett and Phillips (1992) speculated - left nothing general to say. The 'question of our time' appears to have locked gender discourse into a double-bind: the promise of defining the identity of 'woman' is as compelling as its failure is inevitable. This, and the dependence of identity on that which must be repudiated - what Lacan (1966) calls 'foreclosure' - has perhaps inevitably led to rancour, division, and marginalisation.

1.4. Gender and Sex

We have seen how feminist accentuations of difference may have compromised the sex-gender distinction, embraced aspects of essentialism, and become caught up in paradox. Even the conceptual clarity of the distinction between gender and sex may appear somewhat modernist in a post-modern ambience. Some theorists use the affirmation of 'differences' as being in any case deep-seated parts of a core identity, to apparently bypass the biology versus society issue. However, the concept of gender as distinct from sex cannot be dismissed without argument, bound inextricably as it is into the very notion of difference.

In the social sciences, the differentiation between biological 'sex' and socially constructed 'gender' (Oakley, 1972) has been almost axiomatic. Oakley was not the first to draw the distinction, and herself acknowledges the earlier (non-

feminist) work of, *inter alia*, the psychoanalyst Stoller (1968) and the sex research of Money (1965). Oakley says that "'Sex' is a biological term: 'gender' a psychological and cultural one" (p.158); and concludes that "gender has no biological origin, that the connections between sex and gender are not really 'natural' at all" (p.188). This idea is important to feminist theory, at least insofar as biology has traditionally been used to justify the position of women in society. However, the perceived negativity of early feminist theory towards women's 'biology', has produced a reaction that may in turn be perceived as a reclamation of biology.

In 1970 Shulamith Firestone, for example, identified the biology of reproduction as the basis of women's oppression, and the goal of feminism as not just the elimination of male privilege, but also of "the sex *distinction* itself". Genital differences should no longer matter culturally (Firestone, 1970). Before her, Beauvoir appeared to depict women's biology as troublesome, and to represent relative physical strength within an assumed competitive framework. She referred critically to woman's identification as "a womb, an ovary". This perspective also led her to point to biological similarities rather than differences in the early years of childhood: "it is through the eyes, the hands, that children apprehend the universe, and not through the sexual parts". It is only with external intervention that children think of themselves as "sexually differentiated" (Beauvoir, 1949/1954).

Biological studies suggest that hormonally there are at least five broadly 'different' categories of people, with differing balances of 'female' and 'male' hormones cutting across the traditional sex dichotomy (Bleier, 1986). This suggests that 'sex' may be rather less binary and more analogue than culture and convention determines. Hood-Williams (1996) indicates that studies of chromosomes are ambiguous, and culturally filtered. There are also indications that these subtle gradations may be largely unspoken features of people's experiences of discrimination. Woolston (2001) quotes a professor of chemistry at the State University of New York, as identifying discrimination centred on more subtle nuances of 'gender': she accounts for the lack of respect she often

receives in terms of gradations of the 'feminine' and 'masculine' - to her "wispy voice and youngish face" – and indicates that "soft-spoken, young-looking men also seem to have trouble getting ahead in science". Studies of househusbands show that both traditional women and traditional men question the masculinity of men who take on a caring role (Grbich, 1992).

Biologically, sex may appear to be more significant than other more obviously pseudo-biological social taxonomies, such as 'race' - although it may be argued that the significance of, say, reproductive 'differences' is culturally given rather than biologically determined. Yet studies of embryonic development, and of hermaphrodites, can suggest that even these physical 'differences' are not so absolute. Even then the familiar 'gender' debates do not go away: does such evidence point to 'sameness' or diversity, and must sameness default to the culturally dominant gender? The ancient one-sex model (implicit in Genesis, and explicit in Greek and Roman texts) held that women were an imperfect version of men. (This is not, perhaps, all that far from Freudian and post-structuralist models!) In the second century AD, Galen, as cited by Hood-Williams (1996), believed the vagina to be an internal penis, the labia a foreskin. This could well be a model of sameness, were it not for the latent androcentrism. Modern biology, identifying the common embryonic roots of female and male, indicates that, in fact, the prostate is a residual vagina, and the scrotum parallels the labia, just as much as the clitoris is a residual penis.

When comparison is made with other life-forms, 'sex' differences can certainly seem marginal; and differences among females, and among males, can be more significant than differences between male and female. The dichotomous categorisation on the basis of genitalia is obviously a feature of a society which attaches considerable social importance to gender, and as such appears to be a defining feature of patriarchy. While biological difference is significant in terms of reproduction, it does not absolutely follow that it must therefore be socially significant. Even the politics of sexual relations does not inevitably follow, if sexual orientation is not perceived as biologically determined. Different roles in reproduction *become* socially significant. It may be argued

that it is the largely social construction of different experiences of parenthood which forms the bedrock for not only the obvious social differences of inequities, but also for a wide range of deeper gendered cultural tendencies. Stemming from this, power in the public sphere is therefore exercised in the name of biological difference, and the actual 'difference' which this exercise of power represents, becomes its own, self-fulfilling, rationale.

In spelling out what he sees as a 'post-mortem' for the sex-gender distinction, Hood-Williams (1996) draws on previous studies of hermaphrodites and chromosomes to rehearse the arguments that biology is not fixed, there is no clear biological definition of female and male, and biology too is part of the cultural domain in which 'gender' only has traditionally been located:

> Gender is always already implicated within the attempts to define sex - whether as difference or similarity. (p.13)

The contribution of this apparently significant article to the debate is not clear. In examining the sex/gender distinction, Hood-Williams assumes it implies that gender must be based on sex, and the biology of reproduction. This does not necessarily follow, and in many ways 'sex' has not been so much an objective correlative within the debate as a loose comparative metaphor for all those things which, for practical purposes, are not subject to change. The fact that biology may not be fixed does not alter the theoretical or ideological distinction between that which may be changed and that which may not be, or the fact that people will disagree as to what may or may not be changed. The denial of the distinction can therefore still serve as a basis for the full range of opinion - from those who believe that gender roles are biologically determined, to those who regard everything as socially determined. The whole point of much of the gender discussion is that ties to biology *are* artificial. Hood-Williams sees the Freudianism of 80s feminism and, like Barrett and Phillips (1992), 'postmodernism', as "extremely corrosive" of the sex/gender distinction, without clearly demonstrating how this is so. Wilmott (1996), in a response to Hood-Williams, believes that rejecting the distinction renders sociology impossible. Cultural phenomena are not self-explanatory, and biology, whether it be dichotomous or not, fixed or otherwise, largely precedes culture, and is

one basis on which, for instance, the prevalence of patriarchy might be understood.

The scope and manner of the correlation between gender and sex, and the bias of 'explanation' of gender, is the heart of contention - and this is where the 'mutability', or cultural and economic interpretation, of biology as if it were gender, may be conceptually useful. Places in society are not permanent by virtue of being historical, traditional – or even 'biological'. Similarly, the positions of women and men in society may well be traced back to reasons of biology without implying a biologically deterministic perspective. In seeing biology as the origin of gender, many make a leap of reason to argue that gender roles are therefore biologically determined - including, as we have seen, theorists whose declared perspective is feminist. Implicit in this 'leap' is faith in 'nature'. Feminists who assert the social primacy of mothering, both as definitive of women, and as the superior and natural form of parenting - often appeal to the same models as male advocates of a woman's place - primates, and 'primitive' human societies. Yet as Burgess (1997) indicates, non-human primate parenting behaviour is varied and not fixed, and the notion that so-called primitive human societies are 'natural' and not culturally mediated, is a Eurocentric prejudice. Indeed, in many cases these societies are more directly shaped by harsh economic forces, which vary greatly from culture to culture. In the case of the Aka Pygmies of the African Congo, both parents share the nurturing role due, it would seem, to the fact that they share the co-operative, family-based hunting activity on which their subsistence depends (Hewlett, 1991, as cited in Burgess, pp.87-88).

Delphy (1993) suggests that it is essential to challenge the assumption that sex precedes gender: if one wishes to change reality, then it is necessary to abandon assumptions about it. She suggests that "when historical 'gender' is taken seriously, there is no more room for ahistorical 'sexual difference'", a concept that is "part of the problem, not of the solution" (2000). The framework of some second-wave feminist discourse is that of assumed roles - even when it recognises the cultural determination of those roles. Delphy traces the

acceptance of gender-differentiated characteristics back to Mead's 1935 work on sex and temperament in "primitive" societies (1996) – and finds this valuing of gender and the 'feminine' still reproduced in recent feminist writing (Delphy 2000). Mead's wish to take advantage of apparent benefits of the division was accompanied by a belief that there were no traits other than feminine and masculine ones. The division of labour consequent upon acceptance of a division of characteristics was therefore allowed to be seen as 'natural' - with the naturalness being premised on the twin differences in reproductive functions and physical strength. It has been observed, however, that the less 'primitive' a society is, the more elaborate is the male domination and consolidation of inequality, and that science as well as religion serve as elaborate instruments of injustice (Miles, 1993).

1.5. Gender and Nature

The 'rationalisation' of inequities in terms of the 'nature' of perceived groups of people has been manifested in a wide variety of theories. The phenomenon of 'scientific racism', for instance, found 'natural' explanations for inequities a useful distraction from environmental and economic reasons. In the 19th century, human skulls were measured, and jaws profiled, in the pursuit of genetic 'differences' whose 'natural' quality would place the status quo beyond challenge (Stanton 1960, p.25). As far as gender is concerned, bone measurements, and robust versus gracile skulls, may well serve as a forensic means of identifying male and female remains – although even in this context sex is, according to Henderson, "a continuous variable with a clear bimodal distribution" (1989, p.78), with broad overlapping between both sexes. It is the attachment of ordered values to 'differences' that results in – and is the product of - racism and sexism. Scientific racism was often conflated with a scientific sexism. In the words of Carl Vogt, a professor of natural history at the University of Geneva and a contemporary of Darwin: "The grown up Negro partakes, as regards his intellectual faculties, of the nature of the child, the female, and the senile white." (Gould, 1977, p.217).

The brain-related 'reasons' for male 'superiority' have varied historically: brain size, frontal and then parietal lobe positions, hemispherical lateralisation, have all been related to intelligence for as long as comparisons appeared to be favourable to Anglo-Saxon males. Bleier (1986, p.7) argues that these theories are groundless, including the more recent: that the hemispherical variability between individuals of the same sex is greater than any variability between the sexes. Prenatal hormone theory (e.g. Imperato-McGinley, Peterson, Gautier, and Sturla, 1979) points to a differential development of male and female brains in utero. According to Bem (1993), however, there is no evidence that prenatal hormones serve as a primary influence on the functioning of the primate brain; even rat behaviour is influenced by social interaction. Yet many of these ideas have gained widespread cultural acceptance - presumably because they reflect and legitimise assumptions that are already culturally rooted.

Bem points out and exemplifies that throughout history, biological theory has been used "to naturalise, and thereby perpetuate, social inequality" (1993, p.6). She examines biological determinism at its crude roots in the deterministic evolution theory of Herbert Spencer - whereby existing social structures were biologically ordained and optimised by evolution. While Darwin did not relate evolution to social organisation, he did presume that males of each species are subject to more selection (and hence more highly evolved). This state of affairs may have been modifiable, but it presented a case for *de facto* male superiority and dominance. Bem indicates the extent to which inequalities were naturalised in the late 19th/early 20th century by citing the fact that feminists of the time resorted to racial difference.

Persistent social conditions are sometimes designated as virtually natural. The social organisation of gender, and the differentiation of the public and domestic spheres hinge, according to Chodorow (1978), on the social construction of mothering. The traditional family reflects a division of labour that is prehistoric – a division so long-standing that its 'construction' is often treated as if it were 'natural'. Bioevolutionary accounts see existing patterns as a product of this, though the idea "that women have greater mothering capacities than men apart

from lactation" (p.17) is at least disputed by many first-wave and materialist feminists. As we have seen, it is commonly argued that masculine identity is harder for males to obtain because they are brought up by women; and object relations theory has not been tempered by investigations into children nurtured by men. Chodorow (1978) argues for shared parenting as beneficial for both genders and for boys, and 'people's sexual choices might become more flexible, less desperate" (p.218). Mothering "creates a psychology of male dominance, and fear of women in men" (p.219). The "ideology of women as mothers extends to women's responsibilities as maternal wives for emotional reconstitution and support of their working husbands" (p.219). Assumptions that child care is indistinguishable from child bearing continue to "serve as grounds for arguments against most changes in the social organisation of gender"; and "resistance to changes in the sex-gender system is often strongest around women's maternal functions". Chodorow sees "conscious organisation and activity" leading to shared parenting, but does not indicate *what* organisation or activity.

A considerable body of feminist writing moves beyond the identification of women with their 'natural' reproductive capabilities, to identify women even more essentially with Nature. Feminists such as Susan Griffin (1992) use this essentialist identification to value rather than devalue women. In *Woman and Nature: The Roaring Inside Her*, she extends the identification to communication with trees and the dead. Man is "set on this world as a stranger", while "woman speaks with nature" (Griffin, p.76).

Such an essentialist identification – far from advancing the position of women - may actually conform to the ethos and tradition of patriarchy. Sherry Ortner (1972) has argued that such assertions represent a thinly veiled extension of the identification of women with their procreative functions (something that Kristeva does explicitly). Ortner concludes that woman is not in reality any closer to nature than man (p.506). Griffiths (1988) is also critical of the essentialist ramifications of this perspective, and presents one negative interpretation: women are seen as perceiving the world in a fundamentally

different way; they are "part of natural things; men are in control of them" (p.132). She sees the stereotypes of feminine/female connectedness, intuition, and natural communicativeness as basic expressions of this essentialism.

Lowe and Hubbard (1983) present a series of essays to demonstrate that there is a vast range (across classes and societies) of stereotypes of woman's nature, rather than one single myth. While the effects are seen to be varied, those in power often use ideas of biologically inherent limitations to limit women and resist change. The 'universal' myth of the passive, nurturing woman that focuses on motherhood and domesticity, is seen in particular as a white middle class myth.

Lowe and Hubbard go on to criticise the dangers of feminist approaches that rely on notions of woman's nature - be they assuming the ineluctable oppression of women by men, or the innate superiority of women "because of traits they trace to female reproductive capabilities". Yet they also appear to conflate biological motherhood and socially-constructed childcare and related duties in validating this equal 'women's work', rather than challenging its allocation.

Difference, and the sex/gender dichotomy, have figured prominently, albeit in different ways, in every strand of feminism. The fact that both issues are problematic - and in particular that the two sides of the standard dichotomy between sex as biology, and gender as social construct, are themselves occasionally thrown into doubt – may appear to undermine the foundations of traditional 1970s feminism. Barrett and Phillips (1992) state that the need to identify a **cause** of women's oppression is common to every traditional tendency – whether it be personal attitudes, the capitalist system, or men. Barrett and Phillips go on to argue that the assumption of cause - and, therefore, the debate about difference - has been rendered irrelevant by the problemising of the sex/gender distinction, as well as by anti-racist perspectives that introduce possibly more significant variables.

> Sexual difference therefore came to be viewed as more intransigent, but also more positive, than most 1970s feminists had allowed. This shift was variously signalled in the growing interest in psychoanalytic analyses of sexual difference and identity; in the analysis of women's experience of mothering as forming the basis for alternative (and more generous) conceptions of morality and care; and in its most "essentialist" moments, the celebration of Woman and her Womanly role. The impulse towards denying sexual difference came to be viewed as capitulation to a masculine mould.

Yet these tendencies coincide with central aspects of patriarchy and sexism when they identify women and their social roles with and through nature. In asserting 'difference', feminists such as Hartsock (1983/1997, p.162) define women in terms of biological reproduction, and social caring roles. It is a popular truism in the late 90's that "women" are a) good communicators, and b) 'jugglers'. In this regard, the academic and the popular frequently reflect each other. Comment on the 1997 intake of female Members of the United Kingdom Parliament regularly refers to these characteristics, irrespective of the personal circumstances of the individual women. Glenda Jackson, for instance – echoing Hartsock's quotation from Marilyn French - asserts that women make better MPs because "all women" do a 16 hour day by definition, and "women are used to having to do several things at once" (Liverpool Echo, 22-7-97). In talking about the right to work flexibly and the right to parental leave, Government ministers consistently target women rather than men.

In feminist theory, Hartsock also defines women, uniquely and universally, in terms of their "double-day" (1983/1997, p.165), an experience that is apparently assumed to be inherent in the condition of being female. There appear to be at least two tensions to resolve: firstly, the tension between acknowledging that domestic responsibilities are gendered but not universal to all women, and the desirability of using inclusive language as part of a will to change; and secondly, the tension between combating subordination and celebrating the skills (often as intrinsic female skills) that arise from that subordination. The latter is perhaps a defensive mechanism, as Wajcman points out:

> Retaining overall managerial responsibilities for the family appears to be one way in which women exercise power within the home and retain their gender identity (Wajcman 1991, pp.625/6)

This is rarely problemised, and there is little sense that 'domestic' change may be a precondition to equality in the more traditionally public domain. The universal assumption – and the territorial reluctance to use proactively inclusive language - implies an acceptance that domestic work is actually a feminine duty. In Wajcman's own study of senior managers' domestic lives, she qualifies the need of female managers to adapt to long working hours by hiring domestic help, but leaves unquestioned the underlying gendering of domestic responsibility:

> Given that women managers have had to conform to the male model of work, they have little choice but to transfer these tasks to others. (Wajcman, p.622)

The arguments of 'biology' perhaps inevitably extend into the territory of social construction. Hartsock (1983/1997) asserts the unique "unity of mind and body" involved in women's experience of bearing *and raising* children. This notion of "connectedness" as the defining characteristic of "women" (which makes them good with people rather than 'things') is always based on definitions of women in terms of their bodies' reproductive characteristics. In Hartsock's version, it is the "challenges to bodily boundaries" represented by women's experience of "menstruation, coitus, pregnancy, childbirth, lactation". This "connectedness" is therefore premised on biology, and is also dependent on the continued conflation of childcare with childbirth, insofar as it allegedly privileges a woman with attributes that are suited to the care of children.

We can therefore perhaps better understand how feminist object relations theory could arise naturally from traditional object relations theory, and some of the problems inherent to it: both assume the universal dominance of mothering as an inevitable corollary of women's reproductive role, and the focus for individuation; and both tend to define women in terms of their biology. Males deny the body, females accept and become it; females successfully individuate, males do not. The feminist emphasis ultimately rests upon the different

experiences of girls and boys in this respect, the mutual identification of mother and daughter, and of boys with a distant father. That this may appear to be an entirely recursive description - males are distant because they model themselves on distant males - of something that is in fact social convention, is frustrating, not least because it is usually left impervious to the prospect of change, and ultimately may justify existing sexual divisions of labour in terms of innate differentiation between female and male skills. Any revaluation of the feminine versus the masculine does not detract from the intimations of more traditional scientific sexism.

To characterise the female as communal and plural, while associating the male with individuation, may not only validate gender, but also deny women individuation. Such a characterisation may have much in common with the failure to regard women as people – and with the reason why one has to become a woman. Standpoints, however, embraces essentialism through a filter of personal individualism, and reacts to the notion of individuation: "Women's failure to separate then becomes by definition a failure to develop" (Gilligan 1982, p.201). The positive response, that individuation is not a goal because it is seen as masculine, can be problematic. The individualism of standpoints, and the rejection of universalism, is compatible with categorical rejection of this concept of 'individuation' because it is masculine, and characterised by detachment rather than connectedness. By the same (self-defeating, ironic?) logic, the aspiration that women should be regarded as people may be compromised if 'people' default as masculine. Beauvoir's concept of women as "existents" (rather than "a womb, an ovary") becomes a masculinist aspiration, relinquished in favour of the very revelling in "immanence" that Beauvoir deplored as the existential burden of women's oppression (Beauvoir, p.208). Evans (1983) complains that Beauvoir operates "in a male world" when she notes that "wives" at parties congregate and talk to each other. She suggests that some feminists might even want to call her an "Aunt Tom" (p.352), because of her denial that women have ever constituted "a closed and independent society" in favour of terms of reference whereby women "form an integral part of the

group, which is governed by males and in which they have a subordinate place".

On the other hand, Delphy makes a convincing case that it is those who value gender difference – and femininity – and regard it as if it were natural, who are the reformists, who want to live with things fundamentally as they are, but with more emphasis and value placed on women's traditional roles and abilities. Segal (1999), like many other North American theorists, "does not want sexual difference to stop being socially significant, just to stop being so hierarchical" (Delphy 2000, p.161). These issues, concerning subjectivity, stereotypes, and the interpretation of gender, therefore need to be carefully navigated when studying both the social definition of computing 'culture', and the subjective identification of programming 'styles' in the context of that culture. Gender as a social arrangement can be rendered predetermined and unalterable, and styles as subjective interpretations can be misinterpreted in the light of essentialist assumptions.

1.6. Conclusion

We have critically surveyed a range of perspectives on gender. This has enabled us to explore the reasons that may be given to account for inequality, and the thinking behind those reasons. Gender theory is a crucial background for key gender and computing concepts, and a range of feminist theory has been covered, with a trajectory that has broadly led from material politics to psychological individualism, tracing a growing role for 'reclaimed' versions of biological essentialism. This trend is reflected in Turkle and Papert's ideas, and this chapter has therefore given particular emphasis to those wider theoretical ideas. For example, Gilligan's contextual and narrative mode of thinking has been examined as an important theoretical basis for Turkle and Papert's 'concrete' programming style, just as her identification of a canonical policy of non-interference is analogous to their 'abstract' programming style. Object relations theory is also important to their conceptualisation of black boxing in programming.

In addition, we have identified broad tendencies in gender theory that we believe are also reflected in Turkle and Papert: the promotion of change as just a change of emphasis - a revaluation of perceived existing roles and abilities which does not represent fundamental change; and an emphasis on difference at the expense of material equality. The tradition of 'connectedness', premised on (reproductive) biology and nature, and on gendered childcare, as a positive and distinctively female attribute, is implicit in Turkle and Papert's theory. So too is a denial of the broad categorical sameness of sensory apprehension of the world, in favour of fundamental and biologically based difference.

The position of the researcher is critical of these aspects, and therein generally aligns with the perspective of Beauvoir and contemporary materialist feminists such as Delphy. While there may be different gradations of gender - as socially exemplified by Woolston's (2001) wispy professor, or as biologically postulated by Bleir (1986) - the gender categories of the empirical study inevitably mirror the terms of reference of the official statistical data, where gender is mapped onto biological sex. In order to challenge the binary division, it is necessary first to accept it as a statistical category, as the premise of that division. Yet no evidence is found for significant intrinsic categorical differences between females and males. Indeed, the researcher's position is to challenge difference itself, as representing inequality rather than plurality. Within a materialist perspective, variations of 'gender' across cultures still present as the behaviours of female subordination and male dominance (Delphy, 1993). Material conditions produce gender, and gendered consciousness. Those conditions are largely amenable to change: an equality which is not premised on difference is fully achievable. Masculine (and feminine) conquered territory is reclaimable, and women and men can be open to a full range of potentials.

This perspective will be brought to bear in critiquing the more specific literature on gender and computing, and gender and programming. It is also intended, in the empirical part of the thesis, to avoid any emphasis on difference that might actually reproduce difference, and to minimise the explicitness of gender difference in the format and conduct of the empirical study.

We have therefore critiqued ideas – in their original context - that are to underpin much of the literature on gender and programming, along with the deeper assumptions which inform that literature. In particular, we shall see that the notions of object relations theory, and gendered styles, have been interpreted, and applied by means of analogy, to the discourse around gender and computing, and gender and programming in particular. Before examining the meaning of this discourse as informed by this wider context, we will need to situate our analysis within the more immediate context of difference and inequality in the domain of computing.

2. Gender and Computing

2.1. Introduction

Where gender and computing is concerned, 'difference' manifests in two main senses: a) the difference between female and male rates of participation in computing work, study, and leisure; and b) attempts to account for this difference in terms of underlying differences between females and males in computer-related approaches, attitudes, aptitudes, and abilities. Difference in subject-category, or in what is meant by 'computing', is a further factor that would affect both of these concepts.

The purpose of this chapter is to examine – within the context of the feminist theory studied and analysed in the previous chapter - the anatomy of the social construction of gender in computing as a general domain (first placed in the context of gender dichotomies in science); to study statistical data to identify or measure difference between female and male participation in computing; and to review critically the most salient trends in the gender and computing literature concerning 'reasons' for the difference in participation. This should enable new ideas to arise from the analysis of gender in the previous chapter, and provide a stronger context for the more specific study of gender difference and programming.

2.2. Gender dichotomies in Science and Technology

2.2.1. Sex and Gender

We have already seen that inequities are frequently rationalised in terms of alleged cognitive differences, and that these may in turn be justified by biology – be it brain-sizes and sides, or hormones. Additionally, women's roles and abilities have been frequently viewed as defined by (and subsidiary to) their reproductive role – the 'biological' quality of which is frequently extended to encompass childcare as well as childbirth. The exclusive assumption of such a role inevitably produces differences and inequalities in 'public' life – which complete the equation between biology and inequality, and reinforce the

contrast between males as existential agents and females as passive elements of nature. Science, technology, and computing are no exception: indeed, within these domains women's roles and abilities have if anything been more diligently set out in terms of women's 'nature'.

2.2.2. Gender and Science

This emphasis on what Beauvoir called immanence in women, may have a 'feminist' permutation: nurture, connection to earth and nature, with the consequent social implication that one is more acted upon than acting. In such a view, science and technology are masculine, nature feminine; and science acts upon nature, penetrating its mysteries -at best the dissection and abstraction of nature, at worst the rape of nature (Turkle and Papert, 1990).

The debate arising from feminism that is most fundamental to the research question, therefore concerns whether or not feminism and science are mutually compatible or exclusive. This debate largely concerns the relationship between objective truth and science as a social product: is science purely relative to its social shaping, or does it allow for intrinsic objectivity? Keller (1983) attempts to distinguish between the masculine character of science, and its objective character; between that which is "parochial", and that which is "universal": she believes objectivity to be the goal of science, albeit science has been masculinised. For Keller, however, a purely relativist analysis "dooms women to residing outside of real politik modern culture", negates the emancipatory potential of science, and deprives science of enhancing discourses (Harding, p.233). Given a relativist perspective, it is possible to abandon science and rationality as part of the male domain, and instead embrace feminine subjectivity and intuition (Harding, p.237). Keller's intention was to reclaim science through 'feminine' elements and discourses denied by canonical male science (Harding, p.238). (We will later see this template of feminine reclamation applied in the defining studies of gender and programming.) Keller also uses object relations theory to link personality development to the assumed maternal environment of infanthood, and to associate being female with "merging", and being male with " separateness" (Harding, p.239). Objectivity,

associated with the separation of subject from object, is thereby associated with masculinity, and objective competence with alienation.

Keller's 1983 biography of Barbara McClintock, *A Feeling for the Organism*, provides a gender critique of science, analysing its discourses, contexts, and social construction. She draws attention to differences in McClintock's technique – differences that she reads as 'feminine'. In studying the maize plant, McClintock dedicated herself to achieving comprehensive understanding of the organism, rather than applying to it experimentation designed to produce a result.

As has been observed in the opening chapter, traditional object relations theory can be deterministic of gender roles and identities: women are seen as more likely to sustain a primal impulse to attach themselves to objects or people, and are capable of being whole through identification with their first internalised object and repository of separated feelings. Men on the other hand are seen as repressed – and possibly obsessive - in adulthood, because they are unable to identify with a female primary object.

Israelsson (1993) goes beyond identifying science as masculine, to a position where even "analysis" is seen as masculine. She asserts that science or technology is "generally analytical and thus does not fit into the female way of conceiving reality as a whole." She refers to "the female holistic approach to reality". Pohl (1997) alludes to traditional programming approaches as "based on the analytical style of 'Western' philosophers" (p.192). The very act of dissection is framed as inherently masculine, a violation of the whole that is inimical to women's nature.

It has been noted in the natural sciences that it was patriarchy and male bias which propagated "stereotypes of 'the feminine', and thereby made it seem self-evident that women were totally unsuited for 'penetrating' nature's mysteries" (Bleier, 1986). In gender and computing, the stereotypes which are used to rationalise inequity, are sometimes also deployed as part of the 'solution'.

Writers on gender and computing may not always appeal directly to the supposed biology of testosterone and brain lateralisation, but it is not uncommon, in the literature, to refer to female biology as a basis for different attitudes to computers (Turkle, 1988). The use of object relations theory is commonplace (based on the unique biological experience of pregnancy and childbirth - and an assumed unique social experience of childcare) as a basis for the received stereotype that females are 'connected' and good with people, while males are disconnected and good with things – hence explaining a perceived male aptitude for traditional computing, which in turn explains empirically observed inequity.

Studies on gender and computing - as elsewhere in women's studies - frequently accept the socially constructed conflation of childbirth with childcare, and identify all women with mothers or potential mothers who have a unique responsibility to care for their children (Etzkowitz, as cited in Frenkel, 1990). In a study which otherwise radically challenges the masculine construction of technology study, Cockburn (1985) explicitly supports the exclusion of men from "child care and home life". Within such a framework, 'solutions' will frequently work from or around, rather than challenge, fundamental social stereotypes and basic divisions of labour.

The vehicle for the metaphor of 'stereotype' is to be found in print technology. The defining characteristic of the metal printing plate cast is that it is permanently fixed. We use this term therefore when gender characteristics or roles are cast as essential or immutable, and where this casting is linked with differentiation and discrimination.

2.2.3. Stereotypes

Gendered dichotomies profoundly influence computer systems, computer studies, and computer culture. In many studies, stereotypes are reclaimed – to embrace and strengthen the stereotypically 'female' paradigms - men like machines and obsession, women like people and communication. Ideologically,

this may produce a relatively new division of labour, with women in PR/Service type jobs, and men where they've always been.

Some areas of computing are frequently cited as more suitable for women than others. Artificial Intelligence (AI) is seen as a "natural" area for women because its cognitive content is related to the stereotype, asserted as an axiom, that "women are generally more introspective, attuned to psychology, and more verbal than men" (Strok, 1992). However, the UCAS data show that women are considerably less likely than men to study Artificial Intelligence as a degree subject – and that the disparity is even greater (15:85) than that which holds for Computer Science generally.

Various other areas, such as Human Computer Interaction, and Systems Analysis, are sometimes claimed in the pursuit of subdivisions (or ghettos) that are distinctively matched to supposedly intrinsic feminine capacities. The implicit inverse of this is that software and hardware production – and other areas that are not primarily concerned with presentation - is suited to men's nature.

Even in post-industrial economies, technology-based companies' recently-discovered need for women, as we shall see, is frequently based on perceptions of woman's 'nature', and on the acceptance of larger traditional roles. It is characterised by 'positive' stereotypical (apparently innate) abilities and characteristics that are often based on the traditional domestic servicing role of women. Kramarae (p.217) refers to women's responsibility "for the household and family"); and Morris (p.63; p.71) sees childcare as being necessitated by mothers working.

Eden and Hulbert (1995), Wajcman (1991), and Cockburn (1985), see the computer as intrinsically a 'male machine'. Cockburn, however, deploys feminine stereotypes in proposing, as a solution, "to domesticate technology". This leads to, at the extreme end of role stereotyping, computers being seen in affluent societies as helping women in their traditional roles - allowing women

to shop, exchange recipes, and liase with their children's schools (Gurton, 1995). A 'smart' multimedia system called MOM, provides daytime TV, shopping, digital aerobics videos, a multimedia kitchen (the adverts for which are not dissimilar to the old ads for a new-fangled kitchen: 'the rest is push-button magic'). In the economies that service consumer societies, the labour of computer assembly and data entry has been 'domesticated' - feminised - on the basis of women's 'natural' aptitudes.

The tradition of binary dichotomies would present only one alternative for women – to adopt the male stereotypical traits to succeed - "Be good, hungry, driven, and aggressive" - and try to ignore gender: "Never think in terms of gender where work and ability are concerned" (Strok, 1992).

The ramifications of stereotyping are considerable, and they may lie deeply embedded or hidden in feminist critiques as well as in normative ideas. In considering the research question, it is therefore vital that latent stereotypes be elicited and identified.

2.3. Gender and Computing: issues of representation

The central gender issue in computing is generally assumed to be the lower participation rates or achievements of females when compared to males. Some writers suggest, however, that women *do* in fact achieve in computing, and commonly cite people such as Augusta Ada Byron Lovelace and Grace Murray Hopper as examples of women who played a vital role in the development of computing (Gürer, 1995). Reference to such historic figures does not necessarily contradict the thesis of underachievement: Lovelace and Hopper are frequently used as positive role-models who demonstrate the equal capacity of women to make significant contributions to computing. It is interesting that the contribution of both Lovelace and Hopper was specifically in the field of programming: Lovelace is known as the first conceptual computer programmer, Hopper as "the grandmother of COBOL" (Shashaani, 1993, p.171). In addition, many of the first programmers, in wartime, and in programming ENIAC (the post-war Electronic Numerical Integrator And Computer) were women, Adele

Goldstine being only the best-known. Sr. Mary Kenneth Keller contributed significantly to the development of BASIC; and Jean Sammet and Mary Hawes helped develop a common business language (CBL), with Sammet also responsible for the development of FORMAC, the first popular symbolic maths language. Her 1969 work, *Programming Languages: History and Fundamentals*, is a seminal work on programming languages.

The basic variations in general feminist approaches suggest themselves also in relation to computer science: add women theorists and practitioners (and stir); reject the body of knowledge and start again; or deconstruct the body of knowledge and amend it as appropriate. Two nuanced positions also arise: one does indeed maintain that the participation and performance of females *is* equal, and that to see this we need to redefine what computing is; the second accepts that female participation in what is traditionally recognised as computing is *not* equal, but that it does not matter because such participation is not desirable, and the subject of computing is of no interest to the 'feminine' standpoint. This would echo, at least in part, feminist adaptations of Kristeva's assertion that traditional materialist feminism represents the pursuit of "phallic power".

Some therefore suggest that equal representation in this area is something that women can do without: that either differences in representation do not matter, or that indeed such differences may be positively desirable (Siann, 1997). Whether differences in participation are significant is therefore not just a statistical question. The traditional case, however, is that computing technologies constitute the defining technology of post-industrial and globalist economics, and that women's under-representation within them is disempowering. Powerful technology that can be used to control, can also be used to empower: computer literacy is, in this view, analogous with literacy (Smith and Balka, 1988) .

Various reasons are sought for the under-representation. We will examine the main reasons given in the literature, and attempt to ascertain the relevance of differences in attitude and psychology to levels of participation. First of all, it is

necessary to establish the premise of difference in representation by examining the statistical evidence.

2.4. Statistical Difference

2.4.1. North America

In North America, the Taulbee Survey, the Computer Research Association's (CRA) annual survey of U.S. and Canadian Ph.D.-granting Computer Science and Computer Engineering departments, indicates that for 1999 85% of Computer Science Ph.D. recipients were male, and that 82% of Bachelor's and Master's recipients were male. These figures reflect those of previous years. In 1999, the female percentage of new Ph.D. enrolments (which had risen steadily from 16.2% in 1996, 17.0% in 1997, to 18.8% in 1998) had declined to 17%. 13% of new tenure-track faculty staff were women; while the proportion of female professors remained stable at 16% assistants, 12% associates, and 8% full professors. CRA authors Irwin and Friedman comment: "At this rate, it's going to take a very, very, very long time to attain gender equity" (Irwin and Friedman, 2000).

Other specifically US-based statistics can appear to vary from this data – indicating perhaps different definitions of Computer Science as much as discrepancies between Canada and the USA. Spertus (1991), for example, reports that in 1991 women "received a third of the bachelor's degrees in computer science" in the USA. In 1998, girls represented 17 percent of students who took high school computer science advanced placement tests. Women received less than 28 percent of computer science bachelor's degrees, down from a high of 37 percent in 1984. The decline in female participation from virtual parity at high-school level, through figures around 20% at first degree level, to single-figure percentages for full professors, is traced by Camp (1977), and identified as the "pipeline shrinkage problem". Camp also traces a steady latitudinal decline in the percentage of US women acquiring first degrees in Computer Science from 1984 (37.1%) to 1994 (28.4%). The Taulbee North American figure for female bachelor recipients in 1996 was 16% (Andrews

1997). The Taulbee data, however, conflates Computer Engineering data with Computer Science. As Camp observes, "the percentage of degrees awarded in CE to women is dramatically lower than the percentage of degrees awarded in CS to women". However, from 1997 the Taulbee figures break the bachelor statistics down, and the number of CS degrees greatly outweighs CE, and the CS statistic (16% in 96-97) is identical to the overall figure. Camp acquires her data from the National Center for Education Statistics at the U.S. Department of Education, which classifies Computer Science departments within the category of "Computer and Information Sciences" (CIS). She therefore uses the acronym 'CS' "to represent all the fields of study in CIS". While most of the degrees within this category are still traditional 'computer science', it also covers information systems and information science. The Taulbee survey covers the USA and Canada, and collects data directly from departments of computer science and computer engineering: the number of respondents varies from year to year.

Mayfield (2000) indicates that the declining numbers of women obtaining computer science degrees is in contrast to an increase in the numbers acquiring science and engineering degrees.

2.4.2. Great Britain

The under-representation of women in British University Computing courses is also well-documented (Hoyles, 1988; DfE, 1994; UCAS, 1996-2001). Figure 1 shows the applications received by the Universities and Colleges Admissions Service (UCAS) in the year 2000 for Computer Science degree courses, along with other subjects which they categorise as "Mathematical sciences and informatics". Software Engineering has also been highlighted – as a subject directly relevant to the research question. No finer distinctions are made within the subject-category of Computer Science. The final two columns show the percentage of males and females actually accepted onto degree course places.

Subject	Applied					Accepted	
	Men	Women	Total	M:	F:	M	F
G6 Computer systems engineering	639	73	712	90%	10%	88%	12%
G8 Artificial intelligence	80	12	92	87%	13%	85%	15%
G7 **Software engineering**	**1,590**	**302**	**1,892**	**84%**	**16%**	**84%**	**16%**
G5 **Computer science**	**19,663**	**4,488**	**24,151**	**81%**	**19%**	**81%**	**19%**
G1 Mathematics	2,421	1,504	3,925	62%	38%	61%	39%
G4 Statistics	60	42	102	59%	41%	57%	43%

Figure 1

Women represent 18.58% of applicants to Computer Science degrees, and only 15.96% of Software Engineering places. The idea that the under-representation of women in Computer Science is due to mathematical content, would appear to be undermined by the significantly more balanced figures for Mathematics as a subject in its own right.

It is interesting that none of the Physical Sciences show as low a ratio of women to men in the same year. Of the most popular physical sciences, Geography was 45:55 (Geography as a Social Studies discipline was 50:50), Chemistry 42:58. Even the ratio for Physics show higher female applications, at 21:79. Only in the Engineering disciplines do comparable figures occur: 14:86 for general Engineering, 10:90 for Aeronautical Engineering and for Electronic Engineering, with Mechanical Engineering at 8:92. The ratio for Chemical Engineering, however, was higher (30:70), and a majority of applicants for Polymers and Textiles was female. The number of applicants for Computer Science, however, was greater than the total number of applicants across all Engineering disciplines.

One area that presents as 50:50 overall is that of Business and Administrative Studies. Within that category, however – while the major subject of Business Management (the only subject with a larger number of applicants than Computer Science) reflects the overall equity, there is a finer breakdown of subject, with men more highly represented in applications for Financial

management, and for Land and Property Management, and women more highly represented in applications for Industrial Relations, and for Marketing/Market Research. One problem is that there is no similar breakdown for the subject designated as Computer Science.

The overall gender balance in HE applications has shifted gradually from virtually 50:50 in 1994, to a female majority of 53:47 in 1999 and 2000. Figures for those accepted have been slightly more pronounced. This may be reflected in the fact that the percentage of Computing applications from women in 2000 was slightly higher than in the previous year, when only 18.04% of Computer Science, and 14.71% of Software Engineering, applicants were women. In 1998, the figures were 17.87% and 12.25% respectively; in 1997, 17.28% and 10.23% respectively.

Prior to the creation of the new Universities in the UK, the old UCCA and PCAS figures showed female participation varying over an extended period of time. Henwood (1993) compares old UCCA "computer studies" entrance figures of 24% women in 1980, to 10% in 1987; while O'Dubchair and Hunter (1995) compare UCCA female acceptances for "Computing courses" of 13% in 1985 to 15% in 1993, and PCAS figures of 12% and 20% respectively. The latter represented a more rapid growth than the acceptance of females on all courses – but the current figure of 19% reflects the wider fluctuating pattern rather than this particular trajectory. Department for Education data for 1994, however, give a lower figure, indicating that for the academic year 1993-94, places on computing courses for UK domicile women and men in Great Britain, were 1004 females and 7388 males (i.e. 13.5% female). This discrepancy would suggest that the UCCA/PCAS categorisation differed from the Department for Education's. In any case, the UCAS figures do *not* include students doing Computer Studies or Information Technology as part of a Combined degree: these will fall into either Combined Sciences or "Science combined with social studies or arts". These figures are not finely distinguished in terms of component subjects, but are in any case relatively small when compared to Computer Science. Computer systems engineering, software engineering and

artificial intelligence (as with the Taulbee figures) were included within "Computer Studies" category until 1996.

In relation to teaching staff at British Universities, Grundy's relatively informal survey of academic computing staff in 28 UK universities in the early 90s showed women representing 6% of senior staff and 10.7% of all staff (Grundy, 1996, p.152). In the USA at the same time, Spertus (1991) reported that only 7.8% of faculty staff in computing were female.

Gender segregation in computer-related jobs is also well documented (Strober and Arnold, 1987). While women are under-represented in systems analysis and programming, they are over-represented in the low-paid areas of data entry and computer assembly. Women are better represented in software than in hardware.

Frenkel (1990) cites statistics from 1989 that show women starting out with comparable pay in the USA, falling 25% behind their male counterparts after ten years. Spertus (1991) reports that the salary gap widens after about six years.

2.4.3. Hard and Soft Difference

Researchers from the Washington Research Institute and the School of Computer Science at Carnegie Mellon University gathered 250 Advanced Placement Computer Science teachers across the country in summer sessions to train them in two tracks: C++ and gender equity (Margolis, Fisher and Miller, 2000). They found that the contextualisation of computing was characteristic of women rather than men: in other words, they were interested in putting the technology to a use, rather than in the technology for its own sake. The fact that many courses often emphasise the latter rather than the former, is perhaps definitive of the divide between 'soft' "Information Technology" (IT) and 'hard' Computer Science. And by definition, contextualised IT skills are seldom discretely valuable in themselves, deriving relative value from the context: it is therefore very difficult to ascertain whether they are remunerated as well as

Computer Science skills. There is also insufficient statistical particularity to determine whether there is a gender division between IT and Computer Science.

IT is associated with office and business courses; most courses devoted specifically to IT are not at degree level: those that are, generally form a part of a combined degree, and/or are taught at non-PhD granting institutions of Higher Education.

Many early studies – typically from the early 1980's – were conducted at a time when programming and computer literacy were treated as being virtually synonymous. This accounts in part for the very wide gender discrepancies that existed then in terms of computer literacy (e.g. Hess and Miura, 1985, Hawkins, 1985).

2.5. Explaining the gap

The statistics therefore show that women are significantly under-represented on Computer Science degree courses, and that the under-representation occurs at the application rather than the selection stage.

Traditionally the question concerning the reasons for this has been posed in terms of 'what puts women off?'. Rather than enquiring into larger structural and systemic reasons, there has been an assumption that there must be something, either about the courses in particular or about computing in general, that intrinsically 'puts women off' - and that if that something can only be identified, changing it can at least help remedy the 'problem'. In searching for that something, stereotyped views of gender can sometimes offer convenient terms of reference. In line with the tendencies within feminist theory at the time of the 'information revolution', subjective attitudes and experiences have been most frequently pursued.

2.5.1. Attitudes

Given that the under-representation occurs at the application stage, attitudes would appear to be a potentially productive area to explore – much more, certainly, than any biological concept. The elicitation and exploration of attitudes requires careful consideration if it is not to reproduce preconceived assumptions.

Various scales have been developed that purport to measure attitudes to computers. These include CAGE, a scale that is specifically intended to measure attitudinal gender differences. While studies with children can show significant differences in attitude between girls and boys (Wilder, Mackie and Cooper, 1985), when educated adults take similar questionnaires (e.g. Francis 1994 using Gressard and Loyd's CAS), there is seldom any significant difference. This perhaps points to the inadequacy of these scales in measuring affective attitude, and/or the irrelevance of attitude as a differentiating factor. Hoyles (1988) reports on a different technique used by Siann et al (1986) where half of the otherwise identical questionnaires related to female computer scientists, and half to male; no difference was found.

Liao's (2000) meta-analysis of gender differences in attitudes towards computers suggests that the methodology used does not significantly affect the outcome. Liao's study integrated data from numerous journal articles and theses: differences were more observable the more recent the study had been conducted – and were more pronounced in studies with smaller sample sizes. He suggests that "gender differences on computer attitudes" are observed only in small to medium sample size samples, and are not evident in larger sample studies. The larger sample studies will of course carry greater statistical power. He concludes that minor gender differences do exist – female attitudes are slightly less positive than male attitudes. While this gap increases slightly with age, the differences are still so small as to be statistically insignificant.

Females and males who do not undertake computing study or work usually share the same reasons. The "we can, I can't" paradox mentioned by Francis

(1994) - women allegedly believe that women can, but they personally can't - can have positive nuances. In a recent survey concerning Internet awareness, 100% of men aged 18-35 said they knew what the Internet is; yet when asked to define it, more than 50% got it "totally wrong". Only 50% of women interviewed claimed to know what it is, but only 8% were wrong (Personal Computer Magazine, March 1996). While this may imply a greater openness to learning, Linn (1985) found males more optimistic and females more pessimistic in the use of a computer-based strategy game, leading males beyond the information given.

Arch and Cummins (1989) found that there was little or no difference in female and male students' attitudes towards computers where they had received a structured introduction to computers; and that only among those whose use was unstructured and voluntary, males had more positive attitudes. This would suggest a difference in attitude only the hobbyist end of the spectrum.

Levin and Gordon (1989) concluded that home ownership (or other prior exposure) among students in Tel Aviv had more influence than gender on students' attitudes toward computer - though boys were more likely to have home computers.

Research suggests that, while males may often have more computer experience, there is also the likelihood that they inflate how much they know, and underestimate their abilities (Sax 1995; Lundeberg, Fox and Puncochar 1994).

The extent of the statistical difference between female and male representation on Computer Science courses is therefore not reflected in any remotely similar level of difference between female and male attitudes to computers. The questions concerning female under-representation remain unanswered: it is necessary to bring gender analysis to bear on the question, and to look more deeply into the cultural and cognitive background of computing.

2.5.2. Language

The use of vocabulary such as "execute" or "abort" in computer interfaces has, according to Sherry Turkle, "kept women fearful and far away from the machine" (Turkle, 1988). Turkle, however, provides no evidence. The "hard methods" of science generally allegedly involve violent language: the "language of male domination and female submission". This language reflects an aggressive "distance" from nature, and is associated with the whole notion of "objectivity in science": "If science is first a rape, it is then a duel." (Turkle and Papert, 1990).

> "Programs and operating systems are 'crashed' and 'killed.' We write this chapter on a computer whose operating system asks if it should "abort" an instruction it cannot "execute." This is a style of discourse that few women fail to note. (Turkle and Papert, 1990)

The 'execution' of a computer program, however, refers to the performance of a sequence of instructions. Its use is therefore based on its original meaning, and any association with capital punishment is a homonymic accident or contrivance.

Other comment on language focuses on the use of 'metaphors' such as engineering and even science. Grundy (1996) states that one powerful reason for using these two terms to describe a subject she identifies as "the use of computers", is to "put women off and help to mark out computing as predominantly male territory" (89). The very designation of computing in these terms is said to be using a name that will consistently "put off half the potential market". Tellingly, Grundy compares it to marketing a car with a large boot to men as a 'Ford Shopper'! This reflects a rather deterministic view of gender roles and traits, with the intention of challenging only the most superficial manifestations of those roles.

The 'actual' meaning of words, however, is not necessarily relevant to how those words are perceived: a word's meaning is mutable, and it may well be that it is the culture that redefines these words as puerile rather than the words that define the culture. Even if the words did not represent an affected combat

language that is reminiscent of boys' games, the words would possibly still serve as tokens that demarcate territory that is already the boys' room.

2.5.3. Mathematics

Mathematical content, along with the historical association of computing with mathematics, is frequently cited as something that 'puts women off' (Frenkel, 1990; Collis, 1987). Both the fundamental theory and the original application of computers are mathematical in nature – yet mathematics is often unnecessarily linked to computing as a subject, both in terms of academic organisation and curriculum content. However, the data above show clearly that more women study mathematics than computing degrees. Kramer and Lehman (1990) also report that mathematics grades do not significantly predict success in higher education computer courses. In any case, the transferral of the problem to another domain is of limited use. Reducing or eliminating unnecessary mathematical content may make educational sense, but there is no evidence to suggest that it would encourage more women to do computing degrees.

2.5.4. School

Although school has been identified as the wide end of a pipeline that shrinks through higher education, school has frequently been identified as the source of gender inequality in computer participation. Hess and Miura (1985) surveyed over 5,000 students in summer camps and classes that offered training in programming for microcomputers, and found that three times as many boys as girls were enrolled, and that the ratio in favour of boys increased with grade and level of difficulty.

Newton (1991) has suggested that the use of personal computers in secondary schools may be responsible for the decline in female entrance to computing courses in UK universities. This is commonly accounted for in terms of the association of computers in schools with mathematics and science – which are in turn associated with males. Henwood (1993) questions whether these associations really explain the production of a masculine computer culture. In

any case, the daisychaining of 'masculine' subjects in this way, at best defers any fundamental explanation.

In school, girls are commonly supposed to be less likely than boys to have a home computer, and less likely to be involved in extra-curricular computer activities. Studies such as Culley's (1986) find substantial discrepancies in home computer ownership, and project the inequities into examination results and job opportunities - yet the discrepancies are not explained. Many studies show that, even in schools, boys effectively elbow the girls out from the computers (Carmichael et al, 1986). Chivers (1987) suggests girls-only computer clubs run by female teachers and technologists as a way forward.

Kohl and Harman (1987) found that institutional and economic barriers to computer literacy were more important than gender; and Miura (1987) also found that gender differences were mediated by the socioeconomic status of students.

2.5.5. Curriculum delivery issues

A purportedly 'new' approach to teaching computing is reported in Frenkel (1990). Danielle Bernstein (1992) suggests starting computer science courses with software packages. These, she says, "are less tied to mathematics, and allow students to do something functional quickly". As well as avoiding mathematics, Bernstein also takes as given the stereotype that "women use computers as tools", and that women prefer group-work. All these things, she claims, are served by introducing students to computing via software packages.

This approach is commonplace in 'softer' computing 'studies' or informatics courses, such as those delivered in the UK at Colleges and Institutes of Higher Education. While it may have a lot to recommend itself in general, Frenkel does not provide any empirical evidence of its inherent and particular suitability for women, and is more concerned with establishing whether or not it is really computer science. Such courses may well attract more female applicants, and

Frenkel's concern is pertinent, given that the traditional association of office technology courses with women parallels that of computer science with men.

2.5.6. 'Culture' and Male cultures

The adoption of total dedication to work as a benchmark for professional worth is an obstacle, in any profession, to anyone with childcare commitments. The traditional social model places responsibility for childcare on women, and accepts that senior positions will be held largely by men who are either single or domestically serviced. These men in turn reinforce the culture to which they owe their positions. The phenomenon is perhaps particularly acute in higher education, where research is an open-ended pursuit that can 'require' a virtually unlimited amount of personal time; and also in computing, where the need to update is constant, and software is never, ever, ready on time. Etzkowitz, Etzkowitz, Kemelgor, Neuschat, and Uzzi (1990) report that the main barriers to female chemistry, physics, computer science, and electrical engineering faculty members in a study based in New York State University, are what they term "structural and cultural" obstacles. What this means is the "male model of total devotion to the worksite", and women faculty members who are pursuing careers "in tandem with family responsibilities". Such studies will usually identify the work-culture, but not the allocation of domestic responsibilities, as the cause of disadvantage. Inequity in 'family responsibilities' is perhaps inevitably reproduced in the workplace – yet studies into gender inequity in computing tend to accept this underlying inequity as given.

There can be no doubt that computing has a particularly masculine culture: the statistics alone evidence this. Much of the literature refers very generally to this masculine image of computing, and positions it, in general terms, somewhere along the male-autistic spectrum. Margolis et al (2000) cite the obsessiveness of male behaviour on computing courses – but as something that is necessary for success, that is definitive of the culture, and which puts women off: "women students who enter with high enthusiasm and interest in computing quickly lose faith in their ability and their interest in the subject."

Consumer computer buyers (and hence private computer users – although personal computers are increasingly bought for family use) are traditionally men. Hoyles (1988) reports that in 1982 96% of computer buyers were men. The readership of computer magazines are also traditionally men. Advertising is therefore commonly targeted at men, including father/son combinations. Hughes et al (1985) equate advertisements for computers with those for cars and motorbikes, as "likely to show men sitting at the keyboard in full control, while women are more likely to be draped provocatively around the visual display unit". They also cite computer book covers as appealing to boys rather than girls because they allegedly "often use space fantasy images". Hoyles quotes Ware and Stuck's findings, also from 1985, that men appeared in computer magazine illustrations almost twice as often as women, and that women were over-represented as clerical workers and sex objects.

Advertisements, book illustrations, and publicity material frequently show women and men together at a computer. While much is often made (e.g. Hughes, above) of the body-language implied in many of these images, it is not clear how much of this is in the eye of the socially-conditioned beholder rather than intrinsic to the image: when a male is standing, he is 'standing over', telling the sitting female what to do; when a male is sitting, is the seatless female watching while he shows her what to do? It is, perhaps, analogous to the perception of the few men in advertisements for domestic products, as helpers for the women.

2.5.6.1. Games

Computer games culture is overwhelmingly masculine. Computer games act out traditional masculine fantasies: Jenkins (1998) suggests that one of the functions of playing games "is to rehearse and explore what it means to have a gender". Yet until recently popular games have contained virtually no strong female characters (Huff and Cooper, 1987; Provenzo, 1991). Psychologists have also indicated that children learn significant cognitive skills when they play computer games, and suggest that these abilities will differ between boys

and girls because boys play more than girls (Subrahmanyam and Greenfield 1994). Both this and the identification issue inevitably extend into edutainment and educational software, and the whole computer culture.

There are now increasing numbers of apparently 'strong' female characters in computer games: central games characters such as Lara Croft - and, contrary to Cassell and Jenkins' (1998) erroneous assertion that "none of the warriors is female....no women play active roles" in Mortal Kombat, Sonja Blade - led the way for a raft of TV characters (Xena, Buffy, Alias, Dark Angel). However, it may reasonably be argued that both the mould and the main audience for these characters is traditionally male; indeed, such characters are referred to in the popular media as "action babes" (Sky, 2002).

Studies have shown that typical computer games make boys more aggressive, but have no effect on girls. However, when non-gendered protagonists were used in games, they made girls slightly *more* aggressive (Huff and Cooper, 1987). This identification issue extends into educational software, and the whole computer culture.

The realisation of violent machismo in popular games culture presents a high-profile image, and the symbiotic relationship between unreal games, and real smart weapons and warfare, cannot be ignored in considering computer culture. Military Simulations Inc's Web page blurbs its latest game, "Back to Baghdad": "You are going Back To Baghdad to finish the war that George Bush stopped prematurely. You're the flight leader of an F-16C Block 50 with all the armament needed to get the job done". Users are invited to 'bomb' a real city, within the context of an extreme-right political agenda. Military contractors and games companies cooperate mutually - Holderness (1996) cites Sega and Lockheed.

In a 1996 press release, Psygnosis Ltd present their new CD game product, *Deadline*, and describe the target audience as "15-25 Males". The game is described as an "anti-terrorist" siege and raid game that offers "fully authenticated (by ex SAS personnel) weapons and missions". The catch-phrase

is - "don't forget the Deadline and protect your men". Interestingly, POS and store support information is to be obtained by telephoning a named new member of their sales team, a woman who, we are told, "is going to be a great asset".

The difficulties concerning games mirror those that present themselves generally in considerations of gender and computing, and gender and programming: whether to incorporate females, and if so, how to do so without "colluding in stereotypical understandings of femininity" (Brunner, Bennett and Honey,1998). When 'better' games are produced for girls, they frequently reflect the broader marketing strategy of the toy companies to polarise gender stereotypes, and prepare girls for their traditional roles in life. Games companies seek out easy marketing hooks: their interest is in promulgating, not understanding or modifying, a masculine desire to conquer the world and score points, and feminine preoccupations with living and appearing for others. It is perhaps also the case that computers are less capable of subtlety – of representing, for instance, persuasion rather than conquest.

When asked to describe a "perfect instrument", girls typically invented "human-like household helpers" (such as "the Season Chore Doer"), while boys invented instant transport devices with "elaborate model numbers" (Brunner et al, 1998). In this study, women's inventions are characterised by the researchers as representing a view of technology as "a fellow creature on the earth...needing care and guidance to grow to its best potential within the balance of things surrounding it, within the social and natural network in which things live". However, the actual detail of a typical female invention appears simply to reflect traditional women's cultures and roles: "a beautiful piece of platinum sculptured jewelry, worn around one's neck...would operate all day-to-day necessities to communicate and transport people". It is perhaps not surprising that such research would reproduce dominant cultures: however, these cultures are characterised by the researchers as female-positive and male-negative. They speculate that if the Season Chore Doer had been designed by a boy, it would "pulverize" rather than tidy up things, and affirm that girls "are

interested in thinking about the issues that adult women must face these days, including how to juggle career and family". Stereotypes are thereby implicitly accepted – at the same time as their more obvious manifestations are rejected, both in Mortal Kombat, and in McKenzie & Company (girl scenario software that centres around make-up, how to dress, and how to handle boyfriend problems). Brunner et al advance the idea that games about "being chosen" might be "extremely interesting to young women" – with the proviso that they are interested only in the results of being chosen, not in having any say about it.

Computer-controlled games combine software with more traditional toys. Lego bricks have been linked to the Logo programming language in LEGO Mindstorms. A commercial development of Seymour Papert's *MindStorms* and the MIT "programmable brick", Lego MindStorms had its first prototype in the Lego 'Dacta' schools system. Incorporating Papert's 'learn by doing' constructionist philosophy, it invites children to construct programmable robots and vehicles; to create scenarios and personal devices, and LEGO creatures that can explore their environment. Papert – who has collaborated with Sherry Turkle in analysing the gendered nature of programming styles - presents it as "the culmination of over a decade of both theoretical and technical development in educational information Technology". Yet it presents culturally as a deeply masculine toy for the construction of zappy control machines. Inevitably, perhaps, battlebots and Lego weapons arise among the community represented on the Mindstorms website. Mindstorms.lego.com offers a drop-down menu item entitled 'girls', which jumps to a page on Lego's main site, featuring (non-mechanised) games for girls, and illustrating fairy tea party games, puppies, and various pink items. This is particularly interesting, given Papert's contribution to the literature on gender and programming.

2.5.6.2. Intelligent Agents

The 'problem' of the game developer, who gains easy marketing and recognition through gender stereotyping, is replicated in the developing area of so-called 'intelligent' agents. Agents are virtual characters who help the user at the interface: as such, they have the potential to transform the apparently

gender-neutral way in which most people interact with computers, into a highly-gendered zone. They are particularly deserving of consideration given predictions that they will provide the next major metaphor in computing, transforming computers from tools into assistants (Ball et al, 1997).

The key to the virtuality of agents is whether the user is able to suspend disbelief in them as computational objects, and recognise them instead as psychological companions. They need therefore to be responsive to the user: the user will suspend disbelief only if interactions with the agent occur within the bounds of the user's existing cultural and social norms. Virtual believability is clearly not synonymous with realism: characters are shaped to meet the user's predilections, and not to be independent in any genuine way. In other words, their vicarious quality is not necessarily diminished.

Stereotypes present a clear attraction: ready-made templates that are both easily recognisable and easily constructed. Reilly (1997) states that "believable agents are defined to be interactive versions of quality characters in traditional artistic media like film". Agents may therefore be shaped by social assumptions: as with games, these hazards are compounded by the limitations of digital representation. Laurel observes that "[w]hen we anthropomorphize a machine or animal, we do not impute human personality in all its subtle complexity; we paint with bold strokes, thinking only of those traits that are useful to us in a particular context" (Laurel, 1997).

Human personality is 'analogue' in that its 'values' lie on a continuum and cannot be fully quantized, or represented as a series of discrete values. In digitising personality, the problem of 'aliasing' is inevitable. Where a picture rendered into a finite set of pixels, or a soundwave sampled at discrete intervals, may exhibit jaggy staircasing or noise, the 'jaggies' in a digital personality may manifest as stereotypes.

Categorization is therefore an attractive method for digital analysis and design. Pisanich and Prevost (n.d.) suggest a personality taxonomy, based on five

'types' (introverted, conscientious, interpersonal, emotional, and intellectual). Petta (1998) seeks to define and systemise even "emotional processing", building it into an agent architecture framework.

Agents frequently 'participate' in on-line Multi-User Domain (MUD) communities. Participants also include a minority of males who identify and pursue characters bearing female names; male characters who log on under female names and behave in the sexually aggressive way that they would like women to behave; and by male characters who assume that a woman will need their help and their company. (Bruckman, 1996): the level of unwanted attention and harassment in virtual worlds is an exaggeration of - although a fulfilment of intentions that undoubtedly exist in - the real world. Some of the best-known agents – like Julia - are built to respond to this social environment. Agents that are simplistically responsive to the user are frequently shaped by such minorities. A female agent may not have to be young and photogenic like Lara Croft or a Hollywood star: an agent that is even ready to flirt or look pretty builds a particular social environment that is not likely to encourage or motivate women and girls. MUDs of course are also used as environments for teaching and learning – to teach, for example, writing and communication – and for educators and scholars to share experiences and information. While such educational environments could be expected to be more wholesome, educators frequently look to ready-made TinyMUD robots as retrainable and adaptable agents, and it is still necessary to pay attention to the general pitfalls of constructing stereotyped and gendered agents.

There is no particular shortage of (usually male-authored) female agents. They are often defined as female - whether descriptively or graphically - by details concerning hair and other bodily features. They will have an extensive range of one-liners for sexualised exchanges with males (which appears to pass as 'emotional intelligence'). It may be argued that this reflects the reality of life in a sexist society, and in particular that the defensive strategies are similar to those frequently adopted by women when in the company of predatory males. Yet it may also be argued that there is no need for such strategies on the part of

an AI, and that such characterizations shape expectations as much as it reflects reality - that it is a facile way of attaining the reality of the male 'user'.

85% of the source code of one 'female' agent program, "Julia", is the same as its programmer's other prototype robot, Colin (Foner, 1993). Foner comments that "a large percentage of Julia's code deals with detecting and deflecting passes". One might therefore deduce that this code has been added on to a male-default in order to represent the quality of femininity.

The growth in female agents can perhaps be ascribed in part to the discovery of "emotional intelligence" as a key to the simulation of humanity: a feminine stereotype in itself. Flirting often appears to go with the territory of feminine psychological attunement: Erin O'Malley, a prototype character described by Hayes-Roth (1998), is a bartender, who needs "to flirt a little", even if "she won't let those conversations go too far". Traditional gender traits may be subtly remodulated and repositioned, yet still ultimately reinforced. Clearly even the newest technologies have at least as much capacity to be shaped by, as they have to shape, social attitudes.

2.5.6.3. Internet

The Internet – originally a network for military intelligence - has historically been dominated by men. The Internet has in the past been an overwhelmingly masculine environment. Even in (unmoderated) feminist newsgroups, (alt.feminism, soc.women) around 80% of messages were reportedly posted by men (Shade, 1993). Morahan-Martin (1998) views the metaphors used to describe it (cyberspace, superhighway, the electronic frontier) as 'masculine', and indicative of dominant attitudes on the Internet. On the other hand, as Stein (1999) suggests, interactive communication metaphor is central both to the Internet and to modern computing in general. Indeed, much appears to have changed rapidly towards the end of the 1990s, leading to reinterpretations of the Internet as a communicative medium. Plant (1996) refers to the (post-serial) computer itself as "'connectionist' machine", and presents cyberfeminism as an inevitable development from, and analogy to, the technological development of

the distributed and communicative technology that coalesced into the Internet. Plant is pleased that *matrix* is Latin for womb, and (somewhat vaguely) associates cyberfeminism with subversive hacking.

More recent data suggests that women actually now constitute the majority of Internet users in countries with the highest usage: according to Nielsen NetRatings, as of June 2001, 51.7 percent of active US Internet users were women – an exact reflection of their population ratio. However, males still use the Internet more, in that they do it more often, spend more time online, and view more pages. Nielsen attribute this to Internet pornography. Inevitably – given the dramatic rise in the female figures in recent years – women are less likely to have years of experience; though, according to Content Intelligence [http://www.contentintelligence.com/ci_lite/] women are "three to six times more likely than men to become frequent Internet users within two years" from 2001. Content Intelligence monitors women's online behaviour closely, and advises e-commerce content providers "to focus on capturing the attention of online women", because "women control the majority of the consumer purchasing dollars in the United States".

Nielsen Netratings further indicate that Internet use by women is also increasing in the Asia-Pacific region - in south Korea (45%), Hong Kong (44%), Singapore (42%), and Taiwan (41%) (Nielsen NetRatings, as cited in South China Morning Post, July 4, 2001). Hafkin and Taggart (2001) indicate that "women constitute 22 percent of all Internet users in Asia, 38 percent of those in Latin America, and six percent of Middle Eastern users". They indicate that no figures by sex are available for Africa, and that economic and educational imbalances constrain women's access.

Herring (1994) claims that the Internet does not neutralise distinctions, but rather that women and men demonstrate different "communicative ethics", and have "recognizably different styles in posting to the Internet". Herring's "different characteristic online styles", by her own admission, do not characterise "the majority of users of each sex", but are said to be

"recognizably—even stereotypically—gendered". This somewhat paradoxical admission perhaps 'characterises' the ambivalence of much of the literature with regard to gender stereotyping. Herring categorically describes women and men as "different discourse communities in cyberspace", and cyberspace as "inhospitable to women" because of "masculine net culture". She advocates "women- centered lists", and the reformulation of netiquette rules on women's own terms.

Pohl (1997) reports that feminist researchers are divided "as to whether the Internet is a more 'feminine' technology than more traditional computer technology". She cites Turkle's regard for the Internet as "a female technology which supports special female abilities"; even Dale Spender's more critical analysis still regards the Internet as supporting "the communicative skills of women" (Pohl, p.191). Pohl goes on, however, to argue against the definitive categorisation of feminine and masculine cognitive characteristics, as well as the homogenisation of 'the Internet'. Harcourt's collection of essays (1999) dwells less on essentialist considerations, but looks at the Internet in a more active, concrete way, and seeks to build new cultures rather than reproduce old ones. She concludes that "the Internet is a tool for creating a communicative space" (p.219), within which "the cyborg woman" can develop an identity broken free from her body and from domestic space (p.224).

Even in the USA – where Content Intelligence identify women as critical to the successful commercialisation of the Internet – women's participation is coloured by traditional roles and characteristics: "Women in the U.S. generally use the Internet to find information that will make their lives easier as well as improve the quality of life for themselves and their families" (Hafkin and Taggart, 2001, p.21).

On the other hand, DeLoach (1996) refers to "grrrls on the Web" as women who "make no apologies for holding and disseminating feminist views", but who reclaim individualism for women. RosieX, founder of the cyberzine "Geekgirl", seeks also to reclaim 'geekiness' (grrrls enjoy playing action games and winning), and rejects the idea of a movement as

> based on an older style feminist rhetoric which tended to homogenize women with the same wants/needs/desires.... Heh,

> a bunch of us girls really like each other but we certainly don't pizz in each others pockets for ideas and strengths. Oh well, I can't speak for everyone...

2.6. Industrial divisions of labour: women into computing?

The idea that there is men's work and women's work, and that the differences may be explained by biological differences between men and women, is commonplace: men do strenuous tasks that require skill (such as hunting), and women "have a nurturant character, and in their pursuit a responsible carrying out of established routines is likely to be more important than the development of an especially high order of skill." (Barry, Bacon, and Child, 1957). Such commonplace ideas are frequently asserted or reformulated in more modern discourse about women and computing, both within post-industrial societies and in the offshore electronics industries established by Western companies in less developed economies.

The pursuit of cheap labour has led western industrialists to workers who are 'offshore' and female. The pattern of feminising labour is often accompanied by fostering of belief in the 'natural' suitability of women - in the case of data entry or computer assembly, it is the "routine, yet careful repetition" that women are said to be innately suited to (Enloe, 1992). Where men might present with or acquire individual skills, women, even in the employment market-place, are identified as women: women are attributed with non-physical characteristics that are intrinsic to their womanhood and make them suited to particular work. They are also seen as docile and reluctant to unionise. The belief that their wage must by definition be supplementary, is premised on the assumption that childcare is a woman's primary responsibility (and breadwinning a man's). Nor is it restricted to 'underdeveloped' economies: this assumption is fundamental to UK governmental initiatives - childcare is needed because mothers are at work; women are needed to work in an emergency – such as a severe shortage in computing staff – where their womanly skills can be put to good use. Grossman (1985) quotes Intel's Personnel Officer in Penang, Malaysia as saying that the company hires "girls because they have less energy, are more disciplined, and are easier to control". It is also reported that the Penang plant's practices of holding beauty contests, and make-up

classes, are not unusual. Grossman comments that the workers are allowed "to feel they are part of a global culture which includes the choice between Avon and Mary Quant products, posters of John Travolta and Farah Fawcett-Majors by their beds...."

Henwood (1993) explicitly addresses equality and difference, and draws a distinction between a 'women *in* technology', and a 'woman *and* technology' approach. The former is concerned with superficial 'equality' and mechanisms for getting women into technology – usually to serve the declared needs of business; whereas the latter emphasises the division of labour and changing gender relations, but can assume that 'feminine' and 'masculine' associations and attributes are predetermined. Henwood is concerned that thinking about gender and technology is "caught between the twin dangers of technological determinism and essentialism" (p 43), and that this strategically hampers the prospects of change. Gender discourse around technology can too often serve to reproduce and reinforce difference and inequality rather than to understand its source and change it.

Computerisation has frequently been viewed as a threat to traditionally female occupations in offices and banks: the repetitive, routine tasks that women are stereotypically supposed to be suited to, are also those most suited to computerised automation. However, the economic argument for increasing women's participation is ubiquitous, and premised on the notion of a critical labour shortage (Camp, 1997; Pearl, Pollack, Riskin, Thomas, Wolf, and Wu, 1990). The argument is usually promulgated by government and industry, who are all too willing to reclaim stereotypes, and promote PR and service-related jobs as women's work.

This might be viewed as airline hostess syndrome - where communicative women service mainly male customers. While computer buyers are mainly male, sales teams and support are often fronted by young women. Advert readers are frequently encouraged to "give Dawn a call", usually accompanied by a superfluous photograph of the smiling woman in question.

Women in any case have, for well over a decade now, been sought out by employers and Government, "as if they were an endangered species", because of their "instinct to look after not only themselves, but their families too" (Morris, 1989). This is not dissimilar from the advancement of women's 'natural' aptitude for the "routine, yet careful repetition" of cheap, offshore data entry (Enloe, 1992): in both cases, it is woman's nature rather than women's individual skills that identifies them as suitable for computer work. While nature is clearly one basis for skills, the point here is that the mapping between skills and nature has been strictly and artificially gendered, and is at least in part premised on the continued acceptance of traditional social roles.

The rhetoric surrounding this 'women into computing' movement is reminiscent of the wartime demand for women to fill in the gaps left by male combatants. In the Eighties, Morris referred to the "drastic shortage of skilled personnel" (1989, p.60) in computing – one that could only be filled by women. Most recently, the deputy chair of the UK Equal Opportunities Commission, referring to an alleged five-year demand for a million extra computing professionals, asks: "Where are these people going to come from? Girls and women are going to have to fill in those gaps". (Haughton, 2002). To solve this always "urgent" problem (Morris, p.88), women "are now being pursued and wooed as if they were an endangered species" (Morris, p.94). Women are said to intrinsically have skills of "management, logic, creativity, organisation"...and their "instinct to look after not only themselves, but their families too, makes them superior to men as managers" (Morris, p.95). The Director of UK government business at IBM asserts that women "like collaborating and working in teams, and like taking data and looking for creative connections" (Haughton, 2002).

"Family responsibilities" are often officially identified with women (Department of Trade and Industry, 1995); this is seen both as a natural corollary of a set of suitable skills innately unique to women, and as the key

(rather than the obstacle) to getting women to work in IT - hence the childcare portfolio approach targeted at women.

Women into Technology (WIT) is one of several women's organisations that have taken an approach which, as Henwood (1993) comments, is deterministic both in terms of technology and gender. They quote, with enthusiasm, an employer as saying that "women are indeed more loyal if you treat them properly" (cited in Henwood 1993, p. 36). This approach tends to accept and serve the stated needs of industry - what Henwood refers to as the "perceived skills shortage" (p. 35) – and might be even more harshly criticised as part of an overall corporate strategy for reducing costs. Mayfield (2000) refers to women as "a valuable talent pool in the employee-starved computer science world", and writes about patching the "leaks" that occur through school and university. Kozen and Zweben (1998) also indicate that bringing women into programming is critical to a labour shortage:

> With fewer women programmers and designers entering the field, fewer women workers will help fill the increasing IT job shortage, advocates fear.

Assertions about labour shortages are made in the face of a constant stream of job-losses and redundancies in IT, and high levels of graduate unemployment. In the USA, Matloff, testifying to the U.S. House Judiciary Committee Subcommittee on Immigration in 1998, refuted the notion that there was a desperate shortage of software developers.

2.7. Conclusion

We have seen that there is a significant numerical difference between the representation of women and men in Computing degree courses. The dominant culture is masculine – a construct that in part reflects a wider cultural dominance, but which is also distinctive to computing. We have seen larger, socially gendered patterns, inequalities, and stereotypes replicated within the discipline of computing, and how carefully one has to proceed in extrapolating and categorising reasons that are specific either to computing, or to 'different'

cognitive and psychological characteristics. There has been a readiness in some of the women and computing literature to use existing roles in seizing on aspects that present themselves superficially as culprits. This strand appears to arise out of the deterministic and essentialist trends within feminist thought.

While Internet usage has dramatically levelled out within a few years, it has been suggested that this is due to differentiation in motivations and patterns of usage – a differentiation that is not reflected within Computer Science curricula. Gender data along such lines - on the distinction between application-driven IT and more technical Computer Science - is not available: however, it would be of limited relevance, given that the subject under scrutiny **is** Computer Science. The question as to whether differentiation (in the form of different or more varied content) should be actively introduced to the Computing curriculum, is therefore limited. The core of Computer Science – the sub-discipline that most differentiates it from application-oriented and other curricula - is the creation of software by means of computer programming. It is the activity and body of knowledge that is most identifiable with both power and empowerment in computing; and yet, in the words of one University admissions tutor, "it is the programming that puts people off" (Krechowiecka, 2002).

3. Gender and Programming: an Introduction

3.1. Introduction

While there is a recognised body of literature concerning programming styles and paradigms (e.g. Weinberg, 1971; McConnell, 1998; Halpern,1990; Gray and Boehm-Davis, 1996), little has been written specifically about gender and computer programming. Programming at least gives the appearance of being a highly engaging mental activity. It is therefore perhaps unsurprising that what gender-driven interpretation there has been of programming, should gravitate toward Chodorow's (1989) "apposite" source of feminist theory, psychoanalysis. Personal 'styles' denote how one engages with the process of programming and learning to program. A 'style' reflects one's beliefs about knowledge and objectivity, as well as cognitive tendencies and learning preferences. In general terms, the dimensions of styles are bipolar, often allocated to particular population groups, and – echoing the essentialist and deterministic implications previously identified within gendered psychoanalytic study - can coincide with biologically deterministic notions such as analytical/holistic brain lateralisation (Freedman, 2001).

Within the discipline of computer science itself, even literature on the psychology of programming avoids the issue of gender. Mendelsohn et al (1990), for example, in a study of factors that might make programming easier to learn, recognise that "computing remains inaccessible to many people", and provide a list of exemplary categories that includes "the old, the very young, the disabled, and those with low educational achievements", but not girls or women. These exclusions are in spite of the best efforts of "the computing fraternity" (p.176). Others distinguish between different programming styles, mapping them onto different cognitive models (Green, 1980), but do not attempt to attach any gender significance to those styles.

The Kristevan argument ('Kristevan' in the sense that the argument accepts established boundaries of gender difference, rejecting the pursuit of equality or phallic 'power' in 'masculine' territory) which commentators such as Siann

advance in relation to computing generally, may be attached more specifically to programming. Programming can be viewed as an inherently intense occupation that promotes (or even requires) obsessiveness, a character trait, it has been said, which is more common and more accepted in men than it is in women (Huang et al,1998). It might not, therefore, necessarily be desirable to either increase the representation of women in programming, or to dress up the cultures of programming to be more acceptable to women.

On the other hand, the argument that valorises the historical and existing role of women in computing, points to the fact that the major female pioneers were – as we have seen - involved specifically with programming. Lovelace, the first programmer, after whom the programming language Ada is named; Hopper, who developed some of the first high-level programming languages, recognised how important such languages would become, and coined the terms 'bug' and 'debug' that are central to programming cultures; and Keller, who conducted pioneering work on algebraic languages, and participated in the development of the BASIC language. The earliest programmers, during the Second World War, were almost all women (Gürer, 1995), then designated as "computers" rather than "programmers". Indeed, Gürer cites early career literature to indicate that women have previously been stereotyped as being naturally suited to the task of programming: "Programming requires lots of patience, persistence and a capacity for detail and those are traits that many girls have".

These two perspectives are not entirely incompatible. A view frequently expressed by many of the pioneering female programmers, is that the gender roles imposed by society in the period from the end of the Second World War, generally restricted women to jobs that were compatible with domestic and childcare responsibilities, while pushing men into a more dedicated, concentrated commitment to providing financially for the family. These roles were temporarily disrupted and redefined in many countries by war – hence the prominence of women in early computer programming. Programming may be viewed as an occupation – whether the activity is deemed intrinsically obsessive or not, and whether this characteristic is socially determined or in the

nature of programming - that requires levels of time commitment that are not compatible with persistent levels of domestic commitment. Part of the specific cultural determination of the time-intensive culture may well be inherited from the frontline urgency with which wartime work was conducted: Judy Clapp, one of the programmers on Whirlwind, the first real-time, timesharing computer, refers to feeling like she was "on the forefront, working day and night, inventing as we went" (Gürer, 1995).

3.2. Turkle and Papert

The only original attempt to psychoanalyse different styles of programming has been that of Sherry Turkle who, along with Seymour Papert, has developed the notion of 'hard' and 'soft' styles of computer programming. Gender values are attached to these different styles (unsurprisingly, 'hard' is masculine, and 'soft' is feminine), on the basis of analogies with key concepts from psychology and psychoanalysis, in combination with observations of selected young children at U.S. schools and interviews with programming students at Harvard and the Massachusetts Institute of Technology (MIT). Turkle and Papert's research, and the issues that arise from it, is hugely influential, and lies at the heart of the research question.

The influence that this gendered dichotomy of hard and soft has had is reflected in its frequent reiteration: commentary on gender and computing regularly emphasises, on the basis of Turkle and Papert's work, that a structured approach to programming, and the use of abstraction, alienates women (Sutherland and Hoyles, 1988; Kvande and Rasmussen 1989; Frenkel 1990; Mahony and Van Toen 1990; Grundy 1996; Stepulevage and Plumeridge 1998). Miller et al cite Turkle in stating that, while every computer user is different, "Nowhere are the style differences more dramatic than between the male and female approaches to use of computer technology and the programming for this computer technology" (2001, p.126). Where commentators and analysts have an interest in education, this has inevitably given rise to proposals for reform. Turkle is concerned that teachers of programming are trained to recognise hard mastery as the only real way to

program, whereas it is only 'male mastery'. This, it is claimed, amounts to "discrimination in the computer culture" (Turkle and Papert, 1992, p.8). To bring women into computing, teachers must recognise that this canonical orthodoxy is inimical to the psychological make-up and preferred learning styles of women, and instead "encourage students to develop" soft mastery (1984b, p.49). Others go further, insisting that structured programming should be dropped from the computing curriculum in order to encourage more women to study computing (Peltu 1993).

The response, however, from what Mendelsohn et al (1990) call the 'computing fraternity', has generally been to ignore Turkle's work. (Out of nineteen contributors to this collection published as 'The Psychology of Programming', eighteen are identified only by initials: the nineteenth is called 'Diane'.) Yeshno and Ben-Ari's paper, "Salvation for Bricoleurs" (2001), deals with Turkle's distinctive concept of programming styles as its subject, but contains no reference to gender, women, men, girls or boys (other than in the citation of Turkle's article in the Journal of Women in Culture and Society, and of the International Journal of Man-Machine Studies).

3.3. Programming psychoanalysed: patients and prognoses

The development and nuances of Turkle's, and Turkle and Papert's, analyses, will be traced in more detail before it is critically scrutinised.

3.3.1. hard and soft mastery

In her first publication about programming, Turkle describes two different programming styles. The first is exemplified by "programmer X" - a man who equates his style with "a kind of wizardry", and whose style is explicitly identified with the 'hacker' subculture. The second is exemplified by "programmer Y" – again, a man, but one who likes to work on precisely defined and specified projects; he also "enjoys documentation" (Turkle 1980, p.20). Documentation is the means whereby the meaning and use of code is clarified and communicated to people other than the programmer, in order to facilitate both team development and subsequent maintenance by third parties.

It incorporates both external paper-based documentation, and explanatory comments within the code of programs.

It is Programmer X's hacker approach as a "soft mastery" that subsequently comes to be identified as a communicative and feminine "mastery of the artist" (1984b, p.49). Programmer Y's "hard mastery" conversely became "masculine"- "the mastery of the engineer".

The seeds of paradox are, however, planted early. In apparent contradiction to the identification of Programmer X with a female-friendly style, Turkle in the same year describes the hacker's world as a "male world...peculiarly unfriendly to women" (1984a, p.216). By 1990, Turkle and Papert reformulate soft and hard mastery as "concrete" and "abstract" styles of programming respectively. They come to explicitly acknowledge the similarity between the supposedly feminine soft/concrete style and the "culture of programming virtuosos, the hacker culture", but valorise hacking as "countercultural" (1990, p.141; 1992, p.16). The masculine characteristics of the hacker "counterculture" are therefore set aside, and it is structured programming that is labelled a "Western male gender norm". Hard masters treat "computational objects" as "abstraction", soft masters treat them "as dabs of paint" (1992).

> For the hard master the keynotes of programming are abstraction, imposition of will, and clarity. For the soft master they are negotiation and identification with the object (1984a, p.133).

It is observed at an early stage that "girls tend to be soft masters, while the hard masters are overwhelmingly male" (1984b, p.49). While both types are "masters", the soft-master female is characterised by sensitivity and intuitive artistry: she will "try this, wait for a response, try something else, let the overall shape emerge from an interaction with the medium" (1984b, p.49). She will try things out at the keyboard instead of planning them first on paper. Indeed, she will not like any sort of documentation. This specific sense of 'hacking' is part of a culture that, as Turkle herself illustrates in *The Second Self* (1984a), is intensely masculine. The hacker - as Mahony and Van Toen, citing *The Second Self* as support, argue (1990, p.326) - is the epitome of the competitive techie

machismo that is indulged in mainstream computing culture. Where a masculine obsession with inanimate objects is disconnected and weird, its feminised equivalent is connected and fusional.

3.3.2. Style and Realities: Examples

Turkle and Papert's exemplification focuses on the individual: examples of actual programming tasks are rare. University students and school children are most frequently interviewed about their attitudes to programming and to learning to program. With the undergraduate subjects, there is no reference either to a language or to specific examples, and the interview discourse is very generalised. They are MIT undergraduates - and the references that Turkle and Papert use to directly support their central idea of gendered concrete versus abstract approaches to programming, are either by the author(s) or by MIT PhD students. Where reference is made to a programming language, it is to the children using Logo. Although BASIC and PILOT are cited as other languages that the children use (Turkle, 1984a, p.93), they are not subsequently referred to. Logo is a graphically oriented language that is strongly associated with children's education, in particular with a philosophy of constructivism whereby children learn, not just through 'constructing' their own mental models as in mainstream constructivism, but through concrete experimentation and actual extraneous construction. Turkle's partner and colleague at MIT, Seymour Papert (whose long-standing association with Logo - and Lego - has already been noted in chapter 2) indicates that an acceptance of "negatives" is characteristic of "the Logo spirit". "Logoists", he asserts, "reject School's preoccupation with getting right or wrong answers as nothing short of educational malpractice." In other words, Logo encourages the use of 'errors' as opportunities to explore and learn, as settings for the practice of bricolage. This suggests that the programming environment studied is weighted towards the 'concrete' style and 'soft' mastery in a way that more typical environments may not be.

The children are pupils at a private school which Turkle calls Austen, and which was the site of a computer research project. Fifty children were selected for a particularly immersive experience, fifteen of which were further chosen

(on the basis of their "backgrounds, interests, and talents") for intensive study. Of these fifteen, Turkle concentrates on those who responded enthusiastically to the computer-rich environment provided by the project (1984a, p.96). The children provide the few illustrative programming examples: they use LOGO to draw shapes and scenarios. These scenarios, along with some of the more important individuals, will therefore serve as the best means of expressing Turkle and Papert's ideas.

3.3.2.1. Anne [soft]

One of the children, a nine year old called Anne, 'relates' strongly to the computer, and has "strong views about the machine's psychology": she anthropomorphizes it even more than the other children, believing that it 'thinks' and has preferences just "as people do", and "insists on calling the computer 'he'" (the male-default identification goes unremarked). When Anne "programs the computer she treats it as a person" (1984a, p.110).

Anne is introduced as "artistic" (1984a, p.98), and subsequently described as "an expert at writing programs to produce visual effects of appearance and disappearance" (1992, p.30). She has, it is implied, transferred her artistic skills to the domain of computer programming. In order to make birds of various colours disappear, she rejects the "algebraic" method of assigning the colour of each bird to a different variable. Instead, Anne masks each bird with a sky-coloured sprite, makes each sprite travel with its assigned bird, and makes the sprite appear to render the bird invisible, disappear to render it visible. This, Turkle indicates, enables Anne to "feel" and relate to the screen objects rather than use "distant and untouchable things that need designation by variables": "she is up there among her birds". Although part of Anne's "woven" philosophy believes that the computer is "close to being alive because he does what you are saying" (1984a, p.110), Turkle – in one of many echoes of object relations theory - says that each sprite in her program is "not to be commanded as an object apart from herself". Anne "is programming a computer, but she is thinking like a painter" (p.113).

Anne's "programming style is characteristic of many of the girls in her class": her method allows her to feel that her programming objects "are close". The girls respond to computer sprites as physical, and their work with them is "intimate"; while the boys respond to them as abstract, as "something radically split from the self", and their work with them is characterised by distance. "Most of the boys seem driven by the pleasures of mastering and manipulating a formal system" (p.114).

3.3.2.2. Jeff [hard] and Kevin [soft]

Like Anne, Kevin is described as "artistic", and is also judged to be warm, "easygoing", and to have "interest in others". He is contrasted with Jeff, a meticulous boy whose dialogue is more harshly perceived by Turkle as "tending to monologue", and who is said to be recognisable as "someone who conforms to our stereotype of a 'computer person' or an engineer" (1984a, p.99). Moreover, the lacklustre Jeff "doesn't draw, or paint, or play an instrument" (127). These characterisations are important to Turkle because she believes that "programming style is an expression of personality style" (103).

Jeff's approach to developing a space-shuttle program is characterised by the fact that he makes a plan, "conceives the program globally", then "breaks it up into manageable pieces". This is said to represent the top-down divide-and-conquer strategy that is officially approved 'good programming style'. Kevin on the other hand talks about how he feels, and about "the aesthetics of the graphics". He doodles, and "works without plan, experimenting, throwing different shapes onto the screen". His work is not "systematic", and his programs "emerge – he is not concerned with imposing his will on the machine". He is "like a painter who stands back between brushstrokes" (102). This is perceived to be an "open, interactive" approach (103). Jeff is characterised as inherently 'technical': when he makes a mistake, he calls himself stupid and "rushes to correct his technical error"; when Kevin makes a mistake, his response is a little more effusive, he is said to be frustrated but not

resentful, and the mistake "leads him to a new idea". A model of constructionism in action, he experiences mistakes as learning opportunities.

Like Anne, Kevin's perception of the program is not 'abstract' in that he does not see it as something "apart from his everyday life"; he places himself imaginatively in his program: "Kevin says that, as he works, 'I think of myself as the man inside the rocket ship.'". On the other hand, however, Jeff also projects himself into the program, only in a different way: "When Jeff programs he puts himself in the place of the sprite; he thinks of himself as an abstract computational object." The difference between this and Anne's being 'up there among her birds' is not immediately obvious, but it is implied that Jeff either separates himself from, or even supplants, the sprites. Although the distinction may appear to be somewhat subtle, Jeff and Kevin are therefore said to "represent cultural extremes" (102). And although Kevin is a boy, "girls tend to be soft masters, while the hard masters are overwhelmingly male" (p.107). In addition, compared to Kevin, "Anne does something more", moving further in the direction which Turkle explicitly calls "feminine", of "seeing the sprite as sensuous rather than abstract" (p.117).

3.3.2.3. Henry [hacker]

Henry, an Austen pupil, is described as having "a hybrid style" (1988, p.133). While he "takes pleasure in imposing his will over the machine", and "revels in technical detail", for him the "keynote of programming is not clarity but magic". Although Turkle does not make it explicit, Henry appears to be the bridge between hard and soft mastery that is the hacker. His identification with the computer is said to go "far beyond anything that we have seen soft masters do", in that he "actually identifies with the computer". The distinction with the soft master is not entirely clear: it may be that the soft master identifies with computational objects, and anthropomorphises the computer as another person. Henry sees the computer as "a person he could control", and wants people to be more like machines. Henry's style is psychoanalysed as a "schizoid style", having its roots of course "in infancy and early childhood", and centred on the mother. The schizoid style betrays a crisis of "basic trust" which interferes with

"the process of differentiation of the self from the mother", and precipitates a paradoxical "terror of intimacy and a terror of being alone". This is said to be "Henry's paradox", and his strategies involve both not feeling, and seeking admiration. Magic is said to be his means of implementing these strategies. Although the magician of the machine – the computer wizard with special powers – may appear to be a traditionally male posture, Turkle and Papert appear to conflate this hacker style with the soft style. Its only ambiguity lies in the perspective that the computer "offers a unique mixture of being alone and yet not feeling alone" (147).

3.3.2.4. Deborah [poor] and Bruce [rich]

Deborah is a thirteen year old pupil at a school which Turkle calls the Jefferson Middle School (1988, p.140). As part of a government-funded programme involving MIT researchers as well as the local school committee, the children were provided with the facilities to program in Logo. Sixteen children were subject to closer study, and Turkle returned to the school two years after their computing experience began. Turkle explains how 'turtles' can be programmed to form paths and shapes on the computer screen, and in particular gives a clear explanation of how more complex patterns can be constructed from functionality packaged up as subroutines. So houses, for example, can be constructed by using subroutines for drawing rectangles. Unfortunately Turkle does not relate any of her case-studies to this example, or indeed to the hard and soft mastery dichotomy. The older pupils at this school are psychoanalysed, but are not explicitly put into either camp. Her later work with the schoolboy Alex (see below) gives some indication, however, of how a soft master might relate to such an example of structured abstraction.

Like practically all of Turkle's subjects, Deborah likes computers. Every day she can't wait for computer classes to start. She likes the computer because she puts her feelings in it: "the computer can be just like you if you program it to be, your thoughts, your pictures, your feelings, your ideas" (145). Turkle writes about the time when Deborah first "met" the computer. It is through programming that she is able to identify so strongly with the computer.

However, although she has the soft master's identification with the computer, Deborah uses the computer to gain control through rules and careful planning. She does so, Turkle thinks, because her family is poor, and her reality involves "drink, drugs, and sex".

Bruce, on the other hand, is "well off". He programs in a way that is unpredictable and ungoverned by rules and plans. Although he is also emotionally and artistically oriented, he wants the computer to be predictable, and the idea of a feeling computer makes him cringe.

3.3.2.5. Lisa [soft]

Lisa is an 18 year old first-year Harvard student on an introductory programming course, who considers herself to be a poet, good with words and bad with numbers. Having rejected mathematics in high school because she "wanted to work in worlds where languages had moods and connected you with people", she finds it "scary" that she has become, according to Turkle, "an excellent computer programmer" (1988, p.44/42). Lisa tells of young men at high school "turning to mathematics as a way to avoid people", and some "turning to computers as 'imaginary friends'": she decided to avoid such people, and reject the computer "as a partner in a 'close encounter'". Turkle feels that "computational reticence" is felt by women because they want to keep their distance from the formal systems and regimentation that the computer embodies. Her solution is that women should be "encouraged toward a more personal appropriation" of computer technology (1988, p.59).

Turkle relates how it was therefore a surprise to Lisa that she initially found the undergraduate programming course easy. However, as the course progressed, she was forced to "think in ways that were not her own". Although previously irritated by the identification of mathematics as a language (1988, p.55), when presented with a programming 'language' (unidentified by Turkle), she had wanted to "manipulate computer language the way she works with words as she writes a poem", to

> "feel her way from one word to another," sculpting the whole. When she writes poetry, Lisa experiences language as

> transparent; she knows where all the elements are at every point in the development of her ideas. She wants her relationship to computer language to be similarly transparent. When she builds large programs she prefers to write her own smaller "building block" procedures even though she could use prepackaged ones from a program library; she resents the latter's opacity. Her teachers chide her, insisting that her demand for transparency is making her work more difficult; Lisa perseveres, insisting that this is what it takes for her to feel comfortable with computers.

As the course progressed, Lisa was told

> that the "right way" to do things was to control a program through planning and black-boxing, the technique that lets you exploit opacity to plan something large without knowing in advance how the details will be managed. Lisa recognized the value of these techniques—for someone else.

There is no concrete example of how Lisa "feel(s) her way from one word to another" in a programming language, or how this experience reveals the computer as "a partner in a great diversity of relationships". The programming terminology neither contextualised or explained: it is left to be broadly understood in general terms. A proper explanation of it will be necessary to inform a critique of this case study - which we will return to after we have met Lisa's Harvard colleague, Robin.

3.3.2.6. Robin [soft]

Where Lisa was described as a poet, Robin – a second year student at Harvard - is said to be a pianist. Robin has "gone through much of her life practicing the piano eight hours a day", but "rebels against the idea of a relationship with the computer" because she "doesn't want to belong to a world where things are more important than people" (1988, p.46). She too has observed compulsive young men forming relationships with the computer. Having some empathy with their compulsive intensity, Robin has more to do with these people than Lisa – although she finds them gross because they relate to machines. They advise her that she is going about programming in the wrong way, and tell her that she is "not establishing a relationship with the computer". To Robin this is

"gross", and completely different from her own acknowledged "relationship with her piano" (1992, p.27).

Just as Lisa feels her way from one word to another, Robin "masters" music

> by perfecting the smallest "little bits of pieces" and then building up. She cannot progress until she understands the details of each small part. Robin is happiest when she uses this tried and true method with the computer, playing with small computational elements as though they were notes or musical phrases. (Turkle and Papert, 1990)

Robin is also "frustrated with black-boxing or using prepackaged programs", and is in conflict with her teachers:

> "I told my teaching fellow I wanted to take it all apart, and he laughed at me. He said it was a waste of time, that you should just black box, that you shouldn't confuse yourself with what was going on at that low level." (1992, p.7)

3.3.2.7. Alex [soft]

Alex is a nine year old pupil at the Hennigan Elementary School in Boston, USA. Turkle and Papert describe him as "a classic bricoleur". The concept of the *bricoleur* is adapted from Levi Strauss (1966), with the general meaning of someone who tinkers around with objects, and uses them - in perhaps unexpected ways - to improvise solutions or makeshift repairs. The literal meaning of the original French (masculine) word is 'handyman'.

The programming Alex is involved with is, again, "Logo programming and computer controlled Lego construction materials" (1992). When dealing with Lego wheels and motors, Alex "looks at the objects more concretely; that is, without the filter of abstractions". He uses the wheels as robot shoes by placing them on their sides, and using "one of the motor's most concrete features: the fact that it vibrates" to move the robot.

"When Alex programs", Turkle and Papert say, "he likes to keep things similarly concrete". Using the Logo turtle, Alex wants to draw a skeleton:

> Structured programming views a computer program as a hierarchical sequence. Thus, a structured program TO DRAW

> SKELETON might be made up of four subprocedures: TO HEAD, TO BODY, TO ARMS, TO LEGS, just as TO SQUARE could be built up from repetitions of a subprocedure TO SIDE. But Alex rebels against dividing his skeleton program into subprocedures; his program inserts bones one by one, marking the place for insertion with repetitions of instructions. One of the reasons often given for using subprocedures is economy in the number of instructions. (1992, p.13)

For Alex, the packaging of functionality into subprocedures inhibits his "sense of where I am in the pattern". He prefers a non-structured approach because "it has rhythm", and apparently thinks that

> using subprocedures for parts of the skeleton is too arbitrary and preemptive, one might say abstract. 'It makes you decide how to divide up the body, and perhaps you would change your mind about what goes together with what. Like, I would rather think about the two hands together instead of each hand with the arms. (1992, p.13)

He is said to have "resisted the pressure to believe the general superior to the specific or the abstract superior to the concrete". The prescribed packaging is assumed to be exemplified by "hands as a subset of arms", and for Alex this is too far away from the "reality of real hands". Rather than starting, like a structured programmer, "with a clear plan defined in abstract terms", Alex "lets the product emerge through a negotiation between himself and his material". Turkle and Papert compare this to "the style of chefs who don't follow recipes but a series of decisions made as a function of how things taste." Alex shapes the whole program gradually, rather than build it out of components.

3.3.2.8. Alex [hacker]

Another Alex is a straightforward hacker. He spends fifteen hours a day on his computer, three eating, and six sleeping. When he's programming, it feels like "one of those Vulcan mind melds" from Star Trek (1988, p.217). He does not think of the computer as a person, but does feel as if it is one – "someone who knows just how I like things done". If the computer is invested with a psychology by this Alex, it is one that more explicitly services and reflects his own.

3.3.2.9. other women

The other women that Turkle speaks to about programming all appear to have a standard feel for insides and components, for personalised computers, and a dominant interest in arts and crafts. Lorraine has dreams "about what the program feels like inside", imagines "what the components feel like", and says that programming is "like doing my pottery". (1984a, p.116). Shelley thinks of "moving pieces of language around", not like a poem, but "more like making a piece of language sculpture". (1984a, p.116). Tanya "has a passionate interest in words and the music of speech", calls her computer "Peter", has a "personal relationship with Peter" that "showed the intensity of the most driven programmers", and comes, through the mediation of the computer as "demiperson", to "define herself as a writer", mostly of poetry (1984a, p.124).

Doris, a professor of history, does not program, but is said to use her word-processing software in the same tactile, 'soft' way that Anne does: she is interested "in transparent understanding". This focus on transparency is said to be characteristic – in a somewhat unclear distinction - of the philosophy of the computer hobbyist, rather than the more mysterious culture of the hacker wizard, which focuses on 'magic'.

How this aligns with the soft-hard dichotomy is not immediately clear. A plurality of styles is being presented. However, soft masters want black boxes and computational objects to be transparent, but Turkle says that both Anne, an advanced 'soft' programmer, and Henry, a young hacker, "put the highest value on magic", and try "to keep the computer mysterious. They do not try to understand it completely." In other words, soft masters want transparency in computer objects, but not in computer hardware. Turkle explicitly aligns the hobbyist with the hard master: the hard style of programming "is an element in a coherent system of thinking about the computer as transparent and under control" (1988, p.198).

3.3.2.10. other men

Many of the men in Turkle's study of personal computer owners are not explicitly categorised as hard or soft masters. Indeed, they appear to draw characteristics from both camps – although the ambiguity is not commented on by Turkle. Howard, for instance, has some of the purported hallmarks of a soft master: he takes risks and has magician's fantasies when he programs, and he "finds documentation a burdensome and unwelcome constraint" (p.180). Carl, on the other hand, enjoys documentation: "he likes to have a clear, unambiguous record of what he has done", and derives a "sense of power over the program" from mastering "its precise specifications". We may recall that programmer Y, the prototype for the hard master, enjoyed documentation, where his concrete counterpart found it a burdensome "constraint" (Turkle, 1980, p.21): it is part of the "pursuit of clarity" that characterises hard mastery.

However, the soft/hard dichotomy that this suggests, appears to be contradicted by more significant information. In terms of programming language preference, it is Howard who prefers high-level languages, and Carl who prefers low-level assembly language. It is Carl who is said to enjoy "contact with the bare machine and its logic", for the "feeling of having direct contact with what is 'really going on' in his computer". He "wants to feel in close contact with machine logic", and to have "a direct relationship with the CPU"; but he also "wants the reassurance of step-by-step mastery". Carl, significantly, dislikes black boxes: he wants to work "in an environment where there were no black boxes" (184). Howard, on the other hand, is said to have lost the "transparent relationship between the steps written by the programmer and events taking place in the machine". In terms of the characteristics that appear to be at the core of Turkle's dichotomy, it must be Howard who is largely a hard master, and Carl the soft master. Indeed, Carl is later listed along with Doris and Anne, as someone with "a tactile, 'soft' access to a world of hard rules" (196).

Many of the male computer hobbyists whom Turkle studies, are frustrated by hardware 'black boxes', and "labor to make the computer transparent" (1988, p.194). Another adult male, Arthur, teaches himself high-level languages, then

wants to expand his knowledge vertically: "I wanted to get down there and play with the machine. I liked getting inside, changing things around, seeing that I really understood them" (1988, p.193). Like the soft master, he wants to get inside the machine, but with an apparently different psychological motivation. He becomes "a master of 'peeks' and 'pokes'", whereby memory locations and CPU registers are directly manipulated, and describes this sort of programming as "a very sexual thing".

3.4. Summary

Turkle's rejection of black boxes appears to originate in the perception that, in the words of another MIT student, Jessie, "playing around with things that you don't understand requires a certain amount of self-confidence". The implication is drawn that women, who lack that self-confidence, need to understand those things before they can use them. And to do so by 'relating' to them and how they work inside.

We have seen that there are several subtle nuances here, presenting considerable potential for criticism: because women are identified as being willing to accept "risks in relationships" (1998, pp.48-49), the hacker's relationship with the computer appears to suggest itself as a means whereby women might overcome their 'computational reticence'. On the other hand, an important part of the hacker's risk-taking is that they 'play around' with the hidden internals, and, in Jessie's words, solve problems "by means other than the 'right procedure'". These are the elements that appear to lead Turkle into a seemingly paradoxical position of reclaiming for women aspects of the 'countercultural' male hacker culture. Women, she says, "see themselves as cut off from a valued learning style" when they look at the male "programming virtuosos around them" who resent black boxes and take risks, recognising the by-rights feminine virtue of "computational 'intuition'" leading to close relationships (1988, p.49).

Turkle acknowledges that the "characteristics of soft mastery" are culturally "taught" (1984a, p.107), and a model of socially constructed "correct behavior".

Girls nurture and relate to human dolls, boys erect and control inanimate objects. Nonetheless, she presents traditional soft mastery positively as a feminine style. The roots of this lie in her conviction that these models of correct gender behaviour are validated by object relations theory, and that object relations – and parenting roles - are natural and beyond social construction. In identifying the territory of object relations theory as "far earlier" than the giving of dolls and blocks, she appears to be locating the defining processes at a foetal and primal stage. The unconventional aspect of her apparently essentialist analysis, is that she reinterprets the computer as a suitable object, not just for masculine obsessive infatuation, but for feminine bonding skills as well.

In order to analyse this dichotomy between concrete and abstract styles in programming, it is necessary first to establish a fuller understanding of the discourse which is particular to programming, but which Turkle and Papert have generalised and conflated with the discourses of sociology, psychology, and gender studies. To construct a meaningful critique, the most relevant aspects of programming will require clarification and exemplification: what abstraction, packaged procedures, and black boxes mean and look like; the meaning of 'high' and 'low' levels in programming; and the job of a programmer, as well as different ways in which its art or science may be experienced.

4. Programming discourse: abstraction and black boxes

The key literature on gender and programming has been set out. In order to critique that theory, it is necessary first to place it in the particular context of programming – especially so because the literature itself is written for a non-specialist audience, and works on assumptions of a generalised understanding based on the ready transference of concepts from other disciplines. The purpose of this chapter is to explain the particular programming concepts that have been identified in the literature review as central to the characterisation of gendered programming styles; to unpack the 'canonical' approach to programming; and to elaborate and contextualise abstraction and black boxing. This critical understanding is essential not only to a critique of the literature, but also to any practical testing of the key propositions in that literature. The literature has been examined first, before any explanation of the programming terminology, because it is the primary concern of the research, and any technical explanation should be informed first by the relevant theory (rather than vice-versa); because the literature does not offer any such explanation; and because a careful deconstruction of its terminological generalisation will therefore be a prerequisite and central part of any serious critique.

An examination of the nature of programming, the discourses, metaphors, and paradigms within it, and the applied meaning of the abstract and the concrete, will therefore start the critique of Turkle and Papert's key ideas concerning programming style and gender. It will moreover begin the process of identifying concrete contexts and examples, preparatory to testing the validity of those ideas.

4.1. What a Programmer does

People who produce software are of crucial importance in computing. In post-industrial economies, the number of software development jobs is overwhelmingly greater than the number of hardware development jobs.

Matloff (2001) indicates, for example, that in the USA there are currently 11 times as many. Software development is an activity that is carried out, not just by and for software publishers such as Microsoft, but also for a full range of employers who need dedicated 'bespoke' software for their own particular needs.

4.1.1. Job Titles

The job title of a professional programmer has various interchangeable designations, including Software Engineer, Analyst/Programmer, Software Developer, and Computer Engineer, as well as the by now rather old-fashioned Programmer. Title differences mainly reflect differences between countries or business sectors, with software houses leading the way in the introduction of grander-sounding designations. However, they also reflect changes over time in programming technology. Matloff (2001) notes that, while in the past, being a Programmer rather than an Analyst "was considered low skill work", now coding is performed by practically everyone involved in software development. Networked Personal Computers (PCs), and readily available tools that instantly compile and run code, collapsed the old hierarchical division of labour, from the Analyst who produced diagrams, to the Programmer who translated these into code, to the people in Data Entry who converted the code to punched cards, and the Operators who fed batches of cards into the computer. It may also have helped facilitate a movement, within the actual programming process, away from hierarchical team structures and toward communicative 'egoless' programming.

Although different job titles may carry different connotations, they do not necessarily explain what the 'programmer' does, nor how her/his activity may be conceived or characterised. The term 'engineering' suggests a rigorously systematic science for the development of machinery – a metaphor which software development academia and industry have for some time now attempted to convert into a reality. However, even the 'engineering' interpretation can be ambiguous. In his 1998 manifesto to promote the aesthetics of computing, David Gelernter denies that programmers are either writers or mathematicians: he defines programmers as designers of virtual

machines – the most complex 'machines' in the world - but sees aesthetics as a driving principle of their design. So while the building of a virtual machine might be described as an engineering endeavour, it need not be a mathematically-based engineering, at least insofar as most software does not model rules of physics. Nor can such complex 'machines' be constructed by means of technical 'writing'.

4.1.2. The nature of a program

The virtuality of the software product immediately raises issues concerning the concrete and the abstract. Within a simplistic interpretation, *everything* in programming may be conceived of as 'abstract', in that it is the logic rather than the physical medium of a program that constitutes its essence. Software *per se* is 'abstract': it is, invisible and untouchable, primarily available to the intellect rather than to the senses. It does not occupy space, other than the medium on which it is stored, and the computer memory within which it is run. What the programmer designs is in effect a virtual machine – one that is quickened by another, physical, machine. Together, software's virtual machine and hardware's physical machine constitute a multi-purpose machine that can handle and process almost any kind of data.

However versatile, useful or powerful this combination may be, it does not generate meaning or sensation. The relevance of the artificial intelligence debate, and Turkle's conceptualisation of the computer as a psychological machine (on which her psychoanalysis of programming is based), is reviewed in more depth later.

Software is 'abstract' in a different way, not just in terms of its virtuality: as a final product - and within its development - the computer-specific detail of how it works has been removed (abstracted) and replaced by detail that is understandable by people.

In computer terms, it is the software that is abstract, the hardware that is concrete. As Brooks (1986, p.363) says:

> An abstract program consists of conceptual constructs: operations, data types, sequences, and communications. The concrete machine program is concerned with bits, registers, conditions, branches, channels, disks, and such

The 'concrete' in this sense is closer to the machine, the 'abstract' closer to the human user.

It is because of its intangibility, as well as its complexity, that programmers – and users - frequently have to use metaphor in order to talk more clearly about software and its development. The most basic metaphor - and the one that possibly encourages a view of programming as analogous to an artistic activity - is that of 'writing' in a 'language'. The mathematical notion of a 'formal language', however, serves as the root for the language metaphor, rather than expressive, communicative language. And there are of course many kinds of 'writing': the vehicle for the metaphor is closer to the writing of (extremely complex) recipes or knitting patterns rather than short stories or poems. Furthermore, the 'text' of the program represents a process – the sole point of which is to generate a product (in a way that recipes and other written instructions do not). This fact in part explains the more recent drive towards an engineering paradigm. Developers now talk, not of writing, but of "building" software. The 'building block' is part of this metaphor, which extends into engineering, as in 'software engineering': 'specifications' are followed, and 'components' assembled.

Gelernter argues that it is the complexity of software that makes the principle of 'beauty' so important: "Beauty is our most reliable guide, also, to achieving software's ultimate goal: to *break free of the computer,* to break free *conceptually.*" Gelernter is an advocate of intuition rather than mathematics: he bemoans the dominant perception that "mathematics is serious, aesthetics not", and the fact that computer scientists "would far rather pursue mathematical solutions, so-called formal methods, than teach programmers about beauty". He approvingly quotes Feynman's claim that real programming successes "come to those who start from a *physical* point of view, people who have a rough idea

where they are going and then begin by making the right kind of approximations" (p.25). In the debate and competition between the IBM command-language interface and the Apple Macintosh graphical interface, he notes that the Apple designers saw themselves as artists, and cites as the heart of the matter a technology columnist's comment that the IBM PC is "a man's computer designed by *men* for men" (p.36). Technology "is a man's world", he notes, and "elegance is sissy".

For Gelernter, the attainment of artistic beauty is fundamentally intertwined with the attainment of transparency – breaking free from the computer. He does not conceive of the computer as a partner: it is "the machine's transparency and willingness" that can amplify the thought of the programmer (p.26). Indeed, one of the virtues of the transparent machine is instant accommodation of the programmer's idea: "no backtalk, no bargaining" (p.26). The virtual machine is presented as a concept that "frees us to think of software design as machine design". The talent and training required for programmers is "of the sort that makes for structural engineers, automobile designers, or (in a general way) architects – not for writers or mathematicians".

This drive – however qualified - towards making software development at least a direct analogy of engineering, is fundamental to the question of programming 'styles', and to the idea of a 'canonical' way of programming. We will now examine both the inspiration and implications of this paradigm.

4.2. The Software Crisis and the impulse to formality

As a prelude to examining the real significance of 'abstraction' in computer programming, this section examines the discourse of the more formal software development culture, and the extent to which it relates to how Turkle and Papert conceive of the abstract style.

4.2.1. The software crisis

The idea that software development was or is in a state of crisis first arose as early as the mid-1960s. Most major software projects would not be delivered on

time; many that were delivered, would not be used; and some would either be significantly reworked, or abandoned entirely (Brooks, 1975). The issues that were central to the software crisis have become increasingly important over the years, as cheaper and smaller computer technology has been integrated into virtually every part of life in post-industrial societies, and dependence on software has increased dramatically. The so-called millennium bug - and the huge investment in averting a potential disaster - has been only one prominent symbol of this dependence.

A particular difficulty has been that the complexity of software needs has also increased dramatically. Software's role in safety-critical applications, and in war technology, has grown, and such programs are immensely large and complex. In the US space flight programme, the number of instructions contained within software increased hugely – from a few million for Gemini, and even Apollo, in the 1960s, to around 80 million for a space station in the 1990s (Gibbs, 1994). The earliest pressure to transform software development into an engineering discipline – and the identification of the software crisis - came from military sources such as NATO (Naur and Randell, 1968). The exponential increase in the power of computers has meant a similar increase in the problem of programming them. As an intangible entity, there is no real physical constraint to the level of complexity that software can attain to.

The identification of the "software crisis" was therefore accompanied by calls to bring greater consistency and rigour to the process of developing software - to turn software development into a systematic and repeatable process, with all the methods and tools appropriate to an engineering discipline. Various factors were blamed for software failure. In addition to errors in logic, what is often classified as 'human error', is in many instances attributable to poor design of the software user interface, or to limitations and assumptions that have been embedded into the construction of the software. Traditionally piecemeal and individual approaches to development were sharply criticised, along with the neglect of interface and interaction design. Analysis and design were frequently said to be centred around either the technology or the analyst. All of these

factors led to two major themes: pressure for more communication with the user (and understanding of the user-domain and requirements); and pressure for more methodical and 'egoless' approaches to software development.

The urgent demands that were made at the time, are still periodically repeated, if anything with increased urgency. The US National Science Foundation's report (1998) on a Software Research Program for the 21st Century refers to the engineering of "the large, unprecedented systems of the next century", and indicates that software developers need to "develop the empirical science underlying software as rapidly as possible". Dramatic software failures occur with increasing frequency, and it is specifically programming style factors such as individuality, intuition, and solipsistic artistry that are blamed:

> A quarter of a century later software engineering remains a term of aspiration. The vast majority of computer code is still handcrafted from raw programming languages by artisans using techniques they neither measure nor are able to repeat consistently. (Gibbs, 1994).

"Intuition", Gibbs comments optimistically, "is slowly yielding to analysis".

In June 1978, the United States Department of Defence specified a five-set stage of requirements for high order programming languages. These were – rather tellingly – entitled Strawman, Woodenman, Tinman, Ironman, and Steelman. The IEEE/ACM use the similar titles Strawman, Stoneman, and Ironman to describe cyclical stages in the development of a Software Engineering Body of Knowledge (SWEBOK). The idea of robustness and dependability appears to be associated both with masculinity and with physical building materials.

The proposition that software development should be based on the application of scientific principles, is frequently referred to as 'software engineering'. Such engineering principles would be characterised by rigour and repeatable reliability.

Although there is still no recognised standard body of 'software engineering' knowledge (and no consistent professional certification), the 'software crisis' gave rise to the creation of numerous systematic and methodical approaches to software development. Methodologies for analysing and developing systems - for example, the Structured Systems Analysis and Design Methodology (SSADM) - constitute a part of that movement that is broadly located at the design stage. These are generally referred to as 'structured methodologies', and involve systematic and structured conventions for analysing the information content and flow of existing systems, as well as designing structures for new systems. Probably the key component of the subsequent claims of software development practice to being an engineering discipline, however, is a sub-discipline that is strongly associated with implementation and coding, and is known as "formal methods".

4.2.2. Formal Methods

Turkle and Papert appear to confuse formal methods with a logical approach to computing in general, and structured programming and abstraction in particular:

> What does concern us is that the new trends -- icons, object-oriented programming, actor languages, society of mind, emergent AI -- all create an intellectual climate in the computational world that undermines the idea that formal methods are the only methods. (1992, p.32)

The phrase 'formal methods', however, denotes something very specific in computer science and programming. They will be examined briefly in order to clarify this misunderstanding and its possible link to the misunderstanding of abstraction.

Formal methods use mathematics to formulate a problem, specify its solution, and prove that the solution is correct and meets the original specification. They are seen as important in the development of military software, but have proven too time-consuming (and difficult in themselves) for the great majority of commercial software development. Proofs are obtained by means of deductive reasoning, which is modelled by symbolic logic, independent from natural

language and its multiple ambiguities. Formal methods are founded on mathematics that are known as "discrete". Their discreteness reflects its concern with discrete elements (such as integers) rather than a continuous number line (real numbers), and with finite sets and boolean (false/true) values in particular. Where classical physics and applied mathematics are concerned with the continuous values that are found in the real world, computation is discrete and programs are finite. Both discrete and non-discrete mathematics may of course be abstract, but whether it is through the particular nature of discrete mathematics or the general nature of mathematics, formal methods are associated with the mainstream meanings of "abstraction". As Denvir (1986, p.ix) says, "mathematics is perhaps the most abstract form of thought the human mind can contrive". Sets, relations, and functions are used to specify data types, and algebras to specify procedures, along with propositional and predicate logic. The formalism and complex mathematics of formal methods, may indeed be legitimately regarded as 'abstract' in the sense of being separated from the concrete. These methods are not, however, a concomitant feature either of structured programming, or of 'abstraction' as a programming tool.

4.2.3. Language

Although formal methods undoubtedly represent an extremely formal and mathematical approach to programming, they have nonetheless also been represented in artistic terms:

> Computer programs are built of abstractions at all levels. They are like poems whose language is pure thought, whose form is of science, and whose power, if controlled by an engineering discipline, can be put to extending ourselves and our environment or destroying them. (Denvir, 1986, p.ix)

This appears to resonate with the comparisons which Turkle and Papert make between writing code in a programming language, and natural language composition. Such a comparison is explicitly made in the cases of the Harvard student Lisa who wants to "manipulate computer language the way she works with words as she writes a poem", and Shelley, who thinks of programming as

"moving pieces of language around"; and it is implicit in the general depiction of the creative soft stylist. This section considers whether the association is appropriate, and whether it may be a false analogy - as with the sense of 'abstract' - that distorts their analysis.

The concept of 'language' is undoubtedly an ingrained part of the discourse of programming. Programmers 'write' programs in a programming 'language' that has a syntax and semantics. Much of the theory of programming language syntax has indeed been developed from Chomsky's work on natural languages. Work on programming language semantics has its basis in mathematics and logic. The abstract roots of programming 'languages' are, as we have seen, mathematical, and the semantics of first order languages are based on set theory, functions, and propositional logic.

Mathematics, logic, and programming, may all be viewed as language in the semiotic sense of symbol manipulation that generates meaning. Bolter refers to Galileo's claim that "the book of nature was written in the language of mathematics", and describes mathematics as "a special kind of writing", mathematical theories as "symbolic text of the highest order". He sees "formal language" as "the natural language of computers" (Bolter 1991, p.9-10), and uses the example of the building of data structures through pointers to suggest that "programming itself may be defined as the art of building symbolic structures in the space that the computer provides - a definition that makes programming a species of writing" (p.19). When the programmer writes, s/he writes, in a language that is intermediate, a 'text' that is known as source code: in this primary act of writing, the manipulation of the written signs is deferred: when its instructions are actually carried out, the program itself "creates writing of the second order" (p.185).

However, there are those within the programming community who dissent, and see the language concept only as a metaphor, one that programmers should move away from. Holmes (2000) suggests that programmers' accuracy is compromised by a need to make their code resemble readable prose: without

the language paradigm, programmers would feel more comfortable with, for example, systematic rather than natural spatial alignment of symbols and punctuation. Semi-colons, for instance, are used as statement separators in most high-level languages: because they are psychologically understood as punctuation marks, they are traditionally placed at the end of each line of code. As the source code of programs by definition has extreme ragged-right margins - in other words, lines of code are not at all of any consistent length - it is harder to spot when a semi-colon is missing. The absence of a semi-colon will often mean that a program will not compile, but can also result in a program that runs incorrectly or crashes at a crucial point. Holmes refers to a suggestion from the early 1960s, that, rather than placing semicolons at the end of statements, they should all be aligned vertically at the beginning of each statement. He suggests that, while this would make debugging easier, it did not fit with the average programmer's psychological conception of programming as "a literary practice". Holmes gives an account of the hostility with which this suggestion was met - including, interestingly, his being told that this was "not structured programming". He concludes that the computing profession "would be wise to promote a more considered view of programming that contrasts coding and literary endeavor", and suggests that a good start would be "to abandon expressions like 'programming languages' and 'writing programs' in favor of 'coding schemes' and 'program coding'".

Holmes's suggestions also contain recommendations for abbreviating identifiers - including removal of all vowels. This is supposedly to avoid dangerous spelling errors by systemisation. However, it is not explained how shortening words will improve spelling accuracy (especially with the added cognitive burden of following a secondary set of rules for abbreviation); and in any case modern programming languages normally require the prior declaration of all identifiers, and compilers will therefore in any case detect misspellings and refuse to compile such code. The objection on the grounds of "the coding effort required" (in other words, typing) seems equally specious. Bolter makes the more profound point that "we cannot understand 'higher' or cognitive mental

activity except by analogy with writing" (186), and describes computer programming as the manipulation of signs - as "applied semiotics" (195).

While there is a tendency within the programming community to overly dehumanise the process of coding, the similarities between natural language and writing on the one hand, and programming languages and coding on the other, are not sufficiently meaningful to justify Turkle and Papert's assumptions in this respect. This does not mean that coding is not an art – simply that it is not the art of creative writing. Such analogies are likely to create false expectations of programming, and undermine the concept of the 'soft' style.

4.3. Low and High Levels

The 'signs' of programming languages are understood at different levels – by the programmer and by the computer; and by the user of the program's product, the software. The packaging of signs from one level, into signs at another, is crucial to the communication of meaning. Low-level essentially describes the characteristic of nearness to the computer's own electronic binary language, and high-level signifies greater closeness to people than computers. As Turkle herself says, "The logic of high-level languages is adapted to how people think, no to how the machine 'thinks'." And "when you program in Logo or BASIC, the internal structure of the machine need never cross your mind" (1988, p.183). Turkle does not categorise programmer preference for high and low level languages, but does comment that high level languages are "less satisfying" for programmers who have a "desire for transparency" (1988, p.183). We will consider what is meant by 'transparency' in the next chapter.

The high-to-low scale, mapped onto traditional gender stereotypes in relation to other technological objects, is traditionally translated as female-to-male: women using, say, the engine and controls of a motor car, as effective black-boxes without any concern for the workings of the internal combustion engine, and men wanting to know their inner workings. The turning of the ignition key is an abstraction of the process of ignition, the turning of the steering wheel an abstraction of the implementation of the steering: it does not matter to the driver how it has been implemented, as long as they know how to use it. Black-boxes

use functionality but do not reveal it. Different levels of detail are available on a need-to-know basis: the mechanic does not need to know how the spark-plug is assembled in order to place it in the engine. In programming and program design terms, the range of this high-low scale may be exemplified as in figure 2:

High	natural language	**Human**
↑	Structured English	↓
↑	graphical 'direct manipulation' interface	↓
↑	4th Generation languages	↓
↑	scripting languages	↓
↑	third generation languages	↓
↑	assembly language	↓
Low	binary code	**Machine**

Figure 2

Each level up the ladder represents a more "abstract" level than the level below. The higher level 'language' allows the programmer to use functionalities or data structures without knowing the details of how they are implemented at a lower level. Abstraction is the removal, or 'abstraction', of unnecessary detail - the temporary disappearing of a whole level of complexity. A key feature of procedural abstraction is that a routine may be handled by the programmer in terms only of its functionality - what it does, and how it is used. The programmer does not need to know how it works. It may be treated as a "black box".

By "prepackaged program" Turkle and Papert mean a module, procedure or routine within a programming environment. The case of Alex the soft master introduced the notion of the procedure by inference, as something inimical to his style; yet while Deborah also identifies with the computer, her turtle programming appears to involve the packaging of more complex functionality, such as a rectangle-drawing subprocedure that makes it possible to draw houses and other substantial everyday objects. The latter actually provides a simple but

concise example of the use of abstraction in computer programming. A procedure encapsulates a number of operations that work together to achieve a single purpose. It can be called by name when needed, without the necessity to repeat the full sequence of its constituent operations. In design, a procedure usually corresponds to one of the divide-and-conquer sub-task statements of what the program needs to do. Procedures may be 'prepackaged' by the language designer, the software house that developed the language compiler, the writer of a commercial library of routines, or by another member of one's programming team.

Every programming language has 'prepackaged' routines. The *Write* procedure in Pascal, for instance, enables Pascal programmers to output text and numbers to a screen, printer, or disk file. They do not need to know *how* it does this - they know that it will do it with data of any elementary type, and that they can use it. Such prepackaged routines are coded using lower-level language: a lower-level 'out' procedure would write a given byte to a given Input/Output port. While the items which *Write* deals with will be of a humanly recognisable type - strings of letters, whole numbers - the code that implements it will deal with variables of computer-understandable types such as WORD and BYTE, and will include hexadecimal numbers, memory addresses, processor registers, and assembly-language commands. Normally the programmer does not see this, and certainly is unable to change it. At some point even the most prying, technically-acquisitive, mind will give up before it reaches all the zeros and ones.

4.4. Abstraction in practice: a demonstration

A 'concrete style of reasoning' is perhaps nowhere more essential than in discourse about programming! Concrete exemplification is needed to illuminate the range of arguments concerning 'abstract' and concrete styles. A practical illustration of the uses of 'abstraction' in programming will provide insight into what abstraction actually means in practice, and what difficulties are entailed in rejecting it. In order to facilitate easier understanding, we will express solutions

in English-like pseudocode rather than in an actual programming language, and use a constructive graphical example.

We will start then with a problem that, like Turkle and Papert's examples, is limited and visual in nature. We want to draw a block of stars, or asterisks, on the computer screen – initially one that is twelve stars wide and three deep. Using a 'concrete' approach (and having first found a precise width and depth for the block) we could output each star one by one - rather like Anne does with her Logo birds, or Lisa 'feeling her way from one word to another' in Turkle and Papert. We would have to tolerate some sort of prepackaged routine (such as FORWARD in Logo, or Write in Pascal) to do this. (A feature of the 'concrete' approach appears to be going straight to a chosen computer language, without any 'abstract' planning.) We could produce one line by writing out a sequence of asterisks, followed by a carriage return for a new line, then repeat the same steps two more times, and the problem is solved:

```
write out "************"
write out a carriage-return
write out "************"
write out a carriage-return
write out "************"
write out a carriage-return
```

The output from these steps – a 12 by 3 block of stars - will be the same as that produced by an 'abstract'-style process, so it appears to be of equal 'quality'. However, not only has the 'solution' involved a lot of typing (depending on how large the block is to be), but, more significantly, it is entirely static: if it is for a block which is ten stars wide and four high, it will need recoding if a block which is 12 wide and 6 high is required.

A 'loop', or repetition, is a key feature of 'structured programming'; it is also, essentially, an 'abstraction' of a lower level of detail wherein the repetition of an action is literally represented (do x - do x - do x) rather than simply 'do x 3

times'). In a non-structured approach, it is represented less intuitively by creating a label at the start of the action, then putting a GOTO direction at the end of it, that will go back to the label if a given condition has not been fulfilled. This has the effect of repetition, but a 'control structure' for iteration, such as WHILE, FOR, or REPEAT, is more linguistically intuitive, and packages up the repetition with a single entrance and a single exit point. The FOR loop is used to perform the action a set number of times. If we know that the block of stars is 12 wide and 3 high, then we can draw one line of it by repeating the star 12 times; in turn, we can draw 3 lines by packaging up this single-line routine and doing that 3 times. This involves 'abstraction', or 'black-boxing': if we can work out how to draw one line, then instead of writing out the detail of that 2 more times, we can 'black-box' it into a reusable 'draw one line' routine.

We can do this either top-down by starting with the overall block:

```
draw a block of stars
        FOR line count from 1 to 3
                draw one line of stars
                write a carriage-return
        END FOR
```

- where the detail of 'draw one line of stars' is postponed, or 'abstracted' until the overall logic is settled. Alternatively, we can do it in a more bottom-up fashion by first working out how to draw a line of stars:

```
        draw one line of stars
                FOR star count from 1 to 12
                        write out '*'
                END FOR
```

Either way, we are using functional decomposition and abstraction to enable us to understand the problem a bit at a time rather than all at once. For a novice programmer, it would be tricky to work out the required logic of the nested

loops without using these techniques - in particular, the location of the carriage-return in relation to the loops would be difficult to understand with any certainty. When the 'black-boxes' are slotted together, however, it is relatively straightforward. In the top-down version,

```
FOR line count from 1 to 3
        draw one line of stars
        write a carriage-return
END FOR
```

we substitute the detail of 'draw one line of stars', to get the solution:

```
FOR line count from 1 to 3
        FOR star count from 1 to 12
                write out '*'
        END FOR
        write a carriage-return
END FOR
```

The 'bottom-up' model is not strictly either bottom-up or top-down, but a combination of both. Its advantage is that it allows the programmer to test out the 'draw one line of stars' routine first. This incremental approach to development is essentially a 'concrete' (and often bottom-up) approach that still depends on abstraction and stepwise refinement. It is indicative of the plurality and diversity that is in fact inherent in the practical use of abstraction and stepwise refinement - top-down, bottom-up, middle-out.

This may all seem harder (and more 'abstract') than the initial solution of writing out three sequences of asterisks and carriage returns. However, this is only true for extremely small variations of the problem. The direct approach loses all practicality with a very large number of stars and/or lines – or indeed if the number of stars and lines needed to vary in response to input from the user. If the problem were a little more meaningful - say, to write out given characters in blocks of asterisks - then we have to move to a higher level of abstraction again (I am choosing to present this problem in a "bottom-up" manner). At that

level of abstraction - printing out a capital L, for instance - we can in turn use our 'draw a block of stars' as a black-box building-block. The only problem so far would be that the dimensions of the vertical part of the 'L' differ from those of its horizontal part. We cannot, therefore, package the routine above, and call it several times. We would need to spell it all out two times, each time stepping both the line count and the star count to different specified numbers.

Call the routine once and we get:

```
**********
**********
**********
```

So call it four times and we get the main body of the 'L':

```
**********
**********
**********
**********
**********
**********
**********
**********
**********
**********
**********
**********
```

In order to go in a horizontal direction now, to finish off the 'L', we will need to draw a further block that is wider than it is long. This means writing a new routine. And this is just for one letter - other characters will require a large number of other new routines.

It would be easier if we write one routine that enabled us to vary the dimensions. Instead of drawing 3 rows, draw however many lines are required in a particular situation; and instead of 12 columns, however many columns are needed - so a DrawBlock routine could be called as DrawBlock (3, 6), or DrawBlock (8, 12), or whatever:

```
DrawBlock (height, width)
    FOR line count from 1 to height
        FOR star count from 1 to width
            write out '*'
        END FOR
        write a carriage-return
    END FOR
```

This routine could then be used flexibly to build up letters of a given size (with appropriate adaptations to write indents). Building the letters star-by-star, or even line-by-line (bone-by-bone, as Turkle and Papert's 9 year old Alex wants to do with a skeleton), would be an immense task, with the constituent elements atomised and divorced from their meaning. It is easier to handle it initially at the highest level, in segments that are human-recognisable and that match the terms of the problem – in other words, as letters. Even drawing them block-by-block makes less intuitive sense than designing letter-drawing procedures for each letter. This is the highest level of abstraction: drawing blocks is the next level down; and drawing stars is the lowest level. It is often true that novice programmers cannot see the wood for the trees, but whether a flexible top-down approach is used or not, abstraction is a fundamental tool for reducing the complexity of the problem. Like any tool, its usefulness is clearer in concrete situations.

4.5. Building Blocks and Versatility

Black boxes, at varying levels of abstraction, increasingly mean that the programmer can become a user, and the user a programmer. From a different perspective, it is likely that there will be two types of programmers in future:

those who develop the reusable components, and those who use them. This convergence is not new: Myers, Hudson and Pausch (2000/2002, p. 226) cite spreadsheet users as the first end-user programmers, and refer to "productivity applications" (such as the Microsoft Office suite) that are "becoming increasingly programmable". They predict that "end-user programming will be increasingly important in the future". It is likely that software in the future will consist of reusable software components which end-users can plug together to meet their particular needs. Such components present the prospect of "making every computer user a programmer". The new 'programmer' will only know how to use the available components and tools in order "to solve the problem at hand" (Cox 1990/1995, p.385): they will not be distracted by the hidden complexity . Such a situation would represent the use of "prepackaged routines" to facilitate user-friendly 'programming' - two things which Turkle sees as antithetical.

At the highest practical level, the principle of the 'prepackaged program' defines any commercial software package: a word processor is essentially a tool presented to users at a very high level of abstraction. And in using any feature of a language above binary code, programmers are using a prepackaged routine, an abstraction, a 'black box'. A simple integer calculation - such as 4 x 5 - makes use of an abstract data type that is usually built into a programming language compiler: we do not need to (or usually want to) think about the definition of the data type and that of the multiplication operation. Processing text - a string of characters - usually involves using predefined types (such as STRING) and operations (such as Write or Read): again, we do not need (or have the time) to know how the STRING abstract data type is implemented.

In many languages, routines are defined and implemented separately. This provides a means for hiding the internal workings of modules from any program (or programmer) that uses them. The program, or programmer, therefore only knows what they need to know in order to make use of the routine. This sealproofs the server module, as a client program cannot inadvertently change some aspect of the server module. In this way, black-

boxing dramatically reduces the possibilities and complexities of errors and side-effects, and greatly simplifies maintenance, modification, and the isolation and correction of bugs. At the same time, the black-box module is also protected from any deliberate interference on the part of a hacker-programmer.

4.5.1. Sub-procedures

The concept of the sub-procedure, or local procedure, utilises the packaging of code in order to control the visibility of names and items within a piece of code. The solution for a sub-problem is packaged into a sub-procedure, which is nested within the larger procedure that calls it. Variable data items that are only needed within the sub-problem (or sub-sub-problem), are only visible and usable within that sub-procedure: they are invisible to the rest of the procedure outside. The dominant principle is that the more code a variable is visible to, the more chance there is of it being incorrectly set, and, in addition, the less chance there is of the programmer understanding all the possible permutations. It is therefore more likely that the program will be both accurate in its functionality and clear in its readability to other programmers; and that the program will consequently be easier to maintain in terms of correction, extension, adaptation, and other modifications.

Nested subprograms can, however, become unduly complicated as the level of nesting deepens. The less sharing there is of data, the more independent a subprogram will be. However, procedures can normally access any data that is used in sub-procedures that lie inside (rather than outside) it. Unless it is a compelling part of the logic and structure of the problem, it is often more desirable to separate out sub-programs, and place them at the same level. They can still call each other, but cannot access or change each other's internal state. Their independence will be stronger, and they can be separately tested, enabling the programmer to develop the program in an incremental way – making sure one part works before moving on to the next. The 'canonical' view of programming practice is that the procedure (or other packaged code) should be a self-contained unit: its internal state cannot be changed from without, and it in turn should not be dependent on any global data. Its interactions with the rest of the program should be controlled through *parameters*.

4.5.2. Parameters

The concept of parameters has been exemplified in the block-drawing routines above – where they allowed blocks of different widths and heights to be drawn. Parameters generalise a procedure, allowing it to vary the detail of its functionality by using specific data supplied as its argument or arguments when it is called. So the DrawBlock procedure can be called to draw, not just a set block, but a six-by-three block, or a four-by-two block, or a block of any practical dimension. When the procedure is called, it is supplied with these arguments, or actual parameters: their values are input into the procedure. In this way information can be passed between procedures where it is necessary for the accomplishment of a procedure's task, rather than having it declared globally to all procedures.

A formal parameter is declared in the procedure's declaration, in the parentheses that follow the procedure name, and is effectively implemented as a local variable; however, it is changed only when the argument is input. If the input value is to be processed and a result output, it is necessary to use either a function or, if the language allows, a variable parameter, to get the information out of the procedure.

4.5.3. Modularity and Encapsulation

Some procedural languages take the idea of the prepackaged procedure, and the restriction of access to data, a step further. Modular languages such as Modula-2 and Ada are themselves constructed out of modules. Usually they need a bank of library modules in order to function at all. In this respect they provide a clear illustration of the rationale for modules. For example, in Modula-2, basic input and output routines (reading from the keyboard, writing to the screen) are provided by a service module. Typically this is the InOut service module published by Niklaus Wirth, the author of both Modula-2 and Pascal – although alternative modules can be provided by third parties and as a part of commercial compilers. Without such a module, the programmer would have to

write input and output operations within the body of their own program. Such embedded code would – apart from its intrinsic complexity – obscure the logic of the actual problem that the program itself solves. The InOut module serves to abstract that unnecessary complexity from the logic of the specific task at hand.

This principle can also be used by the programmer to reduce complexity by abstracting coherent types of detail into separate modules. When the programmer needs particular functionality, it can be imported. Detailed operations are referred to in an abstract way, and the code that the programmer is writing can focus on solving one particular aspect of the overall problem.

In addition to encouraging modular design, modular languages also require the programmer to keep the interface and implementation separate. An interface module will simply define what a procedure does, and everything else the programmer needs to know in order to use it: its name, its purpose, and the order and type of any arguments that should be supplied to it when it is called. The detail of how it is actually implemented, must be kept separately. In some procedural languages, this will mean in a separate section, in others – such as Modula-2 – the detail must be kept in an implementation module that is a physically separate file. All modules – including definition and implementation modules - are compiled separately in Modula-2, and a definition module must exist before a corresponding implementation module can be compiled. When changes are made to an implementation module, only that module needs to be recompiled, but the integrity of its links to other modules are still checked. As we shall see, the theme of modularity with abstraction is developed further in object-oriented programming.

4.5.4. Typing of data

The difficulties of coding on a large scale are legion, and are not necessarily resolvable by formal methods. The example of a Venus probe that strayed off course is instructive. The spacecraft was sent permanently off course by an error in one out of many hundreds of thousands of lines written in FORTRAN:

```
DO 3I=1.3
```

had been encoded, instead of the correct

```
DO 3I=1,3
```

Instead of executing a repetition that would have stepped the variable I from 1 to 3, the code instead assigned the value 1.3 to a new variable `DO 31`. The error at its source is more likely to have been a typing error than a logical error. In addition, aspects of the programming language and compiler facilitated the error: a more strongly-typed language would have required all variables to be pre-declared, and to be of a given type. Such a language would not have allowed the false assignment to have occurred, because the new variable had not been declared: the program would never have compiled, and the error would have been identified.

4.5.5. Abstract data types.

Real-world problems usually involve non-trivial data structures. Programmers need to develop such structures along with the operations needed to manage the data. Modular languages allow the programmer to do this separately from actually producing the code. This means that the programmer can focus on reflecting the real-world structure rather than be guided by the peculiarities of the programming language.

Some structures occur so frequently in problem solving that their names – and the names of their access operations - are standard, and implementations are often provided with a compiler: stacks, queues, and linked lists are among the most common examples. The nature of the structure determines the particular operations needed: a stack of trays, for instance, needs an operation to take a tray off the top of the stack (pop), and one to place a tray on the top (push); in the case of a queue of people – where it is important to preserve order - these adding and removing operations would take place at the bottom and top respectively.

4.5.6. Pointers and Opaque types

An abstract data type need only consist of a structured data type and a group of operations to access data items of that type. Opaque types, supported by some languages, represent greater levels of hiding, and make abstract data types even more abstract. In Modula-2, implementing a data structure by using pointers means that its declaration need not be visible to clients. If implemented as a static array, a data type must be declared in full within the definition module.

If the structure of the data is declared in the definition module, any programmer could use this knowledge to bypass the access routines provided and alter the elements of the array directly. The access routines supplied with the data type would be robust – they would, for example, check that a structure was not empty before attempting to take an item from it – and would therefore in theory ensure that no attempt was ever made to remove an item from an empty structure. This safeguard is removed if the internal structure of the data item is visible: the programmer can, whether deliberately or accidentally, bypass the standard access procedures. As a result, illegal operations would become possible. Such bugs would be more difficult to locate, because the means of accessing the structure is not standardised. The service module itself would have also been interfered with, and would be unlikely to recover from any such misuse.

The dimensions of arrays have to be declared within the type definition. This is a consequence of the way the language physically separates definition from implementation: client programs are compiled separately, and need to know how much memory to allocate. If the declaration is not included in the definition module, this information is not available and the program will not compile. Doing so exposes the location of each individual data item. Pointers remove this problem by allocating memory dynamically: the dimensions of a structure can be dynamically altered at runtime. It is therefore not necessary to declare dimensions within the definition module. Pointers store the memory locations of data, rather than (as with static variables) the data itself. This has the added benefits that unnecessary storage need not be allocated, and that no

size limit need be observed. In declaring such a structure, it is necessary only to declare its identifier in the definition module: its full declaration, not just the implementation of its access procedures, can be hidden in the implementation module, in which case it is said to be an opaque type. The only operations that can be used on items of that type – apart from simple equality tests and same-type assignments – are those supplied with the procedures. The opaque type hides implementation detail more thoroughly, and decouples the implementation module from client modules - and even from its own definition module.

4.6. An example of complexity and error: Windows DLLs

This section explains the centrality of prepackaged functionality and black boxes within Graphical User Interfaces, and further illustrates the way in which software complexity necessitates a modular approach, highlighting this by means of the complexity to be found even in an ostensibly straightforward software package such as a word processor.

Modular programming in the Windows graphical user interface – as well as the Sun and Acorn Archimedes operating systems - is characterised by the use of *dynamic linking*. In Windows, this is implemented through the use of dynamic link libraries - better known as DLLs. Most features of the user environment are common to all applications that run within that environment. Rather than have all Windows software include these code elements within the final compiled programs, libraries can be linked to each and every application program as and when that program is loaded or run. Windows programs – and the Windows interface itself – access common compiled functions located within these library files. DLL files are normally Microsoft files that are shared by several programs – although programmers can compile their own DLL files to serve as libraries for their own software. In either case, DLLs break code down into reusable chunks, and serve to make the interface appearance and functionality consistent across different applications, as well as to ensure that special functionality is loaded only when it is needed.

It should be noted therefore that the entire endeavour of the direct manipulation graphical user interface is commonly based on the use of libraries of pre-packaged functionality. Given the complexity of representing graphical interfaces such as the Windows environment, it is hard to conceive of it being otherwise.

However, both graphical interfaces and the modularity on which they are founded, can also have their disadvantages. The Windows environment is particularly vulnerable to problems with the integrity and consistency of DLL files: for example, one software program might need to access a function within a DLL, only to find that the particular routine has been updated in the more recent version installed and required by another piece of software. Or the DLL could have been removed when another program was uninstalled. Newly-installed software can replace more recent versions with older versions. It is seldom clear which program installed (or removed) the original file. Moreover, DLLs can frequently depend on code within each other, and there is therefore considerable potential for different versions to be out of step with one DLL and still be in step with another.

The software will invariably crash (or refuse to start) as a result of any such lack of compatibility, and the friendly graphical interface will present the user with a decidedly unfriendly message telling them that the software has caused an invalid page fault, in a specified DLL module, at a specified memory location. At this point, the friendly, high-level graphical environment takes refuge in long and impenetrable lists of memory registers and locations of file bytes that are both meaningless and useless to virtually every user.

Alternative failure scenarios are linked to the complexity of programming at the relatively low level usually required in the development of DLLs. The point of a black box is that it cannot be corrupted by external routines, and that any errors that emerge can be confined to the black box. However, although the DLL functions are black boxes that are to be used with reference to what they do and not to how they do it, it is possible for a Windows application developer to write code which, when it calls a Microsoft function, will produce an error. Such problems are, by definition, bugs in the DLL file. One of the more frequently encountered bugs in Windows applications at the time of writing

(2001) is the generation of an "invalid page fault in Msvcrt.dll". While Microsoft advise that application developers "need to ensure that their applications are using the C run-time small-block heap correctly" (Microsoft Product Support Services, 2001), the problem fundamentally arises out of heap management changes implemented in a newer version of this DLL, which can result in the referencing of bad pointers. In other words, errors in the programming both of application packages, and of the user interface environment itself – and in the interfaces between both – are not uncommon, and cause software and computers to fail, or 'crash'.

Pointers are a particular source of problems. Pointers allocate memory dynamically, and depend on the validity of the links between particular memory locations. It is sufficient to understand that programming with pointers is usually complex; that they are part of most high-level languages; and intrinsic to object orientation. The corruption of Microsoft Word documents is a useful illustration, in that it has been experienced by large numbers of computer users! Word is object-oriented: a Word document consists, internally, of a linked list of the objects it contains – where an 'object' is anything in a Word document, and will have several properties that define how it appears and behaves. Objects are connected to these properties by pointers to byte locations in the Word file. Most objects will have many such pointers, and some pointers may in turn point to collections of standard properties for common types of objects. As Word has developed, the number of potential objects and properties has increased hugely: the number of potential combinations of objects and pointers is exponentially vast – and not all potential combinations will necessarily be valid.

The multiple tables containing the complex network of pointers that constitutes all the properties of a Word document – and that makes sense of the binary content of the rest of the file - are stored within the document. Section Breaks are used as containers for this purpose. Every Word document has a default Section Break, which is effectively located within the last paragraph mark – a mark that is contained within every document, and which cannot be deleted because it contains binary information that is fundamental to the structure and content of the document.

Master Documents in Word represent a concrete example of the strong potential for corruption, of a faultiness that is in practice inherent, due to the complexity of software. A Master Document incorporates a number of related subdocuments: a typical use (if the facility worked) would be to organise and edit the chapters of a book or thesis. All of the subdocuments will have their own default Section Breaks, (along with any number of user Section Breaks). The structure – and the exponential potential for conflicts – is simply too complex to be sustained: Master Documents consistently become corrupt, and text consequently disappears irrevocably from the file.

The modularity represented by object-based programming, and by dynamic link library files, is necessitated by the sheer complexity of representing software interfaces and interactions in a way that is visual and responsive. The further layer of application functionality obviously represents another layer of complexity that also intricately interacts with the complexity of the environment. The complexity even of individual modules – and the daily failure of these, that has so characterised environments such as Windows – is often overwhelming.

4.7. Conclusion

In examining the nature and difficulties of programming, we have discussed different discourses and metaphors at different levels, and what different styles can mean in practice. The meaning and purpose of abstraction has also been illustrated in order to facilitate a more focused critique of Turkle and Papert's reading of gendered programming styles. The subsidiary theme of top-down decomposition as part of the programming 'canon' has also been exemplified. The analysis suggests that Turkle and Papert have conflated top-down approaches with abstraction, and have missed the full meaning and significance of 'abstraction' or black boxing in programming. We are now in a position – in the light of gender theory, of the general position of gender in computing, and of a more particular understanding of programming - to develop a focused critique of their specific propositions concerning the gendering of programming styles.

5. Turkle's abstract and concrete programmers: an analysis

5.1. Existing criticism of Turkle and Papert

As indicated in chapter 3, Turkle's analysis has rarely been questioned: Wajcman (1991) does express reservations about Turkle's underlying acceptance of "sexual difference in cognitive skills" (p.157), and is generally sceptical of the evidence for such difference. She also touches upon the implicit contradiction between Turkle's idea that males follow rules, and research in mathematics that shows girls tend to follow rules diligently. She comments:

> This would lead one to suppose that girls would be the hard masters in computing. Are we now to believe that in computing boys follow the rules and girls are practising an alternative style? (Wajcman, p.157)

However, it may be argued that Turkle is dealing with a level of psychology beneath the level of outward conformity that rule-based systems demand, and seeks to uncover what lies beneath the mask of what she calls 'computational reticence'. In any case, as a sociologist, Wajcman does not examine Turkle's particular evidence concerning programming, and accepts her less fundamental proposition that there are indeed gendered styles of programming that are unfairly valued. She sees Turkle's feminised concrete style as an instance of "getting the right results by the wrong method" – a concise summary of Turkle's own argument - but does not give any insight into the significance of process and method in an activity where 'right results' may not be what they seem, and need constant care and maintenance.

Wajcman explains Turkle's deterministic tendencies in terms of the influence of the work primarily of Gilligan (1982), but also of Keller (1983, 1985), and Chodorow (1978). As she points out, binary oppositions (between what is female and what is male) often lend themselves to an essentialism or

romanticism that depends on biological sex rather than socially constructed gender.

5.2. Other perspectives

There is, therefore, no direct critique of the essence of Turkle and Papert's influential work on computer programming. Some studies outside of the literature on gender, do touch upon the same territory in relation to programming – but without necessarily relating it to gender.

At a system level, Goguen (1992) identifies a "dry" culture, which sees programming as a branch of mathematics and computer programs as mathematical objects; and a "wet" culture, which regards social factors as the crucial determinant of success for software. Green (1980) makes a similar distinction among computer programmers between "neats and scruffies", as well as between 'neat' and 'scruffy' programming languages. The characteristics of "scruffies" include a predilection for languages, such as C, that allow the programmer "to get at the internals of the machine" (p.22), and a desire to take risks and not to be inhibited by strict rules. The 'scruffy' programmer therefore shares significant characteristics with the soft/concrete programmer, and the 'neat' programmer similarly with the hard/abstract programmer.

Neat languages, such as Pascal, are well defined, and prevent the programmer "from doing things that might be 'dangerous'". Green comments that 'scruffies' "regard that as a paternalistic, even authoritarian, attitude, and programmers in scruffy languages are expected to look after themselves". The dichotomy is summarised in terms of style and preference: "Some people enjoy writing cryptic code, while others enjoy writing self-evident code" (p.24). Turkle's concrete stylist will write code in 'negotiation' with the machine: while the driving style may be creative, the resulting code will inevitably be more cryptic. The thought processes are individual, shared with the machine, but hidden from others. The feminine and masculine stereotypes of 'mystique' and individuality converge to produce equally difficult code.

Green et al (1993) point out that texts on neat and scruffy programming languages tend to reflect the corresponding culture: C identifiers tend to be short and cryptic, while Pascal identifiers tend to be long and more meaningful. In discussing the different 'pedagogical traditions' associated with each language, Green says that 'neat' language students are "exhorted not to start coding until they have fully analysed the problem, nor to approach the computer until the coding details are fully worked out". Scruffy books often set tasks that are

> domain-specific problems, such as interacting with the operating system, or algorithms for natural-language parsing, rather than to analytical reasoning. Students are expected to think in code, and to get the feel of hands-on experience as soon as possible...understanding is gained by making 'fruitful mistakes'.

Such a culture tends to look for solutions that are strongly tied into a particular programming language, rather than design software as an abstract, conceptual construct that would yield multiple representations and still remain the same.

5.3. The Psychological Machine

5.3.1. Machine Bugs and Personality Quirks

Turkle's categorisation of the computer as "a psychological being" (Turkle 1984a, p.54; 1984b, p.50), or "psychological machine" (Turkle 1988, p.50) with an "intellectual personality" (Turkle and Papert, 1992, p.3), raises ontological problems: if the computer really is "a psychological being", at least "on the border between mind and not mind", it follows that a sensitive person would not tell it what to do, and would certainly not expect it to be perfect. Soft masters therefore do not expect their programs to be 'perfect'. Just as the Austin pupil Anne would tolerate and negotiate around imperfections in people, as a programmer she "makes no demand that her programs be perfect". In Turkle and Papert's model of programming psychology, only proponents of "male mastery" are obsessed with debugging - removing "the small errors". Soft masters are tolerant of such "small errors". In this respect the soft master's demand for total understanding appears to be waived: "People can be understood only incompletely: because of their complexity, you can expect to

understand them only enough to get along, as well as possible for maintaining the kind of relationship you want". Boys are criticised for believing that programs are "either right or wrong" – wanting to have "small errors" removed is seen as petty and unforgiving (1984a, p.110). The Venus probe incident referred to in chapter 4 is just one real-world example of a "small error" – many more such incidents have resulted in loss of life.

Turkle indeed reclaims the bug as a personality trait – somewhat like a flaw. An example she gives concerns a student, Mary. Where painter Anne's soft mastery is graphically expressed , "Mary differs strikingly from Anne in having a soft style that is verbal" (1984a, p.110). Mary's programs are "marked by her interest in language". This appears to reflect – or be reflected in – the fact that her programs use text dialogue. Turkle gives an example of this verbal soft style: when the user has finished playing Mary's game program, the program wants to know whether they want to play another game, or want to exit. She writes lines of code in Logo, which Turkle, describing them as a 'program', renders into pseudocode:

> If what-the-user-types is 'Yes,' start a new game.
> If what-the-user-types is 'No,' print score and stop.

When these lines of code run, the result is "not quite as Mary originally planned". If the user types in 'No', the program asks again whether they wish to exit, and finally exits when it is typed in a second time.

The softness of Mary's approach lies in the fact that she likes this error, and decides not to fix it, seeing it as a "humanlike quirk" that gives the program "more of a personality". The personality trait imputed is that "He will not take no for an answer" (Mary, like Anne, insists on using the male pronoun to refer to the computer). It is not clear how this attitude is ascribed to the interaction; how it represents 'personality'; nor how it helps Mary relate to the computer.

Although consistency is not something that can be expected of people, it is a dominant theme in the literature concerning the construction of user-friendly human-computer interaction. Users (rather than programmers) tend to need

consistency in behaviour, and to resent unpredictable 'quirks'. The behaviour imputed by the bug is not backed up by any variation in the response (for example, 'are you really sure?'): the response is the same one, and the obvious danger is that the user, confronted with a prompt that has not changed, will think either that their response has been ignored, or that the program has just stopped working.

The bug/quirk is, as Turkle says, easily fixed (though she does not explain how). Turkle accounts for it by explaining that the computer "is a serial machine; it executes each instruction independently". This is not an entirely precise representation either of how a program executes, or of the reasons for the bug: a computer program will normally store needed information in memory 'variables', and the execution of instructions will in fact depend on the contents of those variables. In this case, the typed response of the user would have to be stored in a variable (in spite of Turkle's identification, in Anne's case, of variable use with a hard 'algebraic style of thinking"), and there would be no need to read the response twice. Mary's problem would appear to be that her program reads the answer once for each condition, within two separate *if* statements. She has the computer wait for "what-the-user-types" twice rather than once - within an *if* statement used for 'Yes' responses, then within a second *if* statement used for 'No' responses. If the answer is 'No', the computer still waits for a 'Yes' response first. It would be more natural to ask once, and then decide (with a single *if-else*), a course of action on the basis of this one-off response. Mary's 'error' is therefore more logical than verbal – it is 'verbal' in the superficial sense that the output and input are textual; and perhaps insofar as one would not normally ask two questions where one would do just as well.

The soft master negotiates with the computer rather than the user "about just what should be an acceptable program" (1984a, p.111), because the framework of an individual artist-and-canvas metaphor does not have any significant place for a user. Such an approach would be potentially disastrous in the world of software development.

5.3.2. Real World or DisneyLand?

Computer programming as a profession is located in what Turkle casually refers to as "real organisations" (1995, p.51). To avoid accident or social and economic damage in this real world, the "small errors" are critical: programs have to be not just debugged, but also debuggable. This means that, for maintenance purposes, people other than the individual programmer have to be able to understand how the program has been written, and ideally to pinpoint a clearly discrete sub-section that contains the bug. Furthermore, if it is to be of use, the program itself needs to translate to the *users'* experience of their real world environment and tasks. In Turkle's evaluation, programming is a means of "working through personal issues relating to control and mastery" (1980, p.15). It is the user who appears to be most significantly left out of this personal and tactile relationship between programmer and computer. And within a programming team, the individualistic intimacy is at odds with the communicative practice essential to holding large projects together. Focusing on "negotiation, relationship, and attachment" (1992, p.9) with machines at the expense of real people is *not* something that is traditionally perceived as a feminine trait.

Indeed, Margolis et al (1999) quote a female programmer who associates programming with the real world rather than the computer, and who explains "how this people-oriented purpose for computers is what resonated with her desire to connect computer science to real world problems" . They also refer to the drive to "figure out how it works" as something that was "prevalent in the male interviews and very rare in the female interviews", positioning the male students "in a very active relationship with the machine". This appears to contradict Turkle's account of soft female programmers wanting to take things apart – programmers like Anne, who does not want to "forget about 'how the bird works'" – and her positioning of relationship and connection as feminine cognitive virtues. In addition, women in this survey "talk about the pleasure in 'systematic thinking'" – something which Turkle and Papert describe as a predilection of "the hards" (1992, p.9).

In her later popular work, Turkle is concerned with MUDs, cybersex, and other fantasies, rather than the real world of programming. It is apparent in this work that she confuses the act of programming a computer with the usable software product. And it is consequently programming as well as software that is confused with DisneyLand, where "the representation exists in the absence of the real thing" (1995, p.47). Post-modern theorists may well "write of simulacra, copies of things that no longer have originals" - but however intangible software is, it **does** represent something - and the fidelity of the representation of real-world needs and environments is vitally important. Relating to computer objects, or "lines of computer code" (1995, p.59) in the way that Turkle suggests, may be at the expense of the real world people and things that those objects and lines represent: real-world environments are **not** like Disneyland's Main Street (1995, p.47; p.236).

5.3.3. Art, design, and the abstract

The metaphors and applications of programming suggest the possibility of artistry. As we have seen in Chapter 3, we 'write' a program in a programming 'language'; and Logo, the language that Turkle and Papert focus on, is commonly used to draw shapes. Turkle and Papert exemplify soft mastery through people who are poets, musicians, potters, and artists. The real artist does not normally communicate the process of composition - nor is it usually desirable or even possible to do so: it is the final work of art that the artist 'communicates', multivariously, ambiguously, and indirectly. They ignore, however, the implications of transferring this pattern to software development. Software produced in this way is unlikely to be maintainable: only the 'artist' would be able to fix or update it.

The "soft mastery" school of thought not only denies the need to document software design, but the very need for a design stage at all. In general, Turkle tends to ignore or overlook the distinction between program design and programming – or to deny the necessity of this stage of software development, perhaps as part of an idealised analogy with the artist, and because the distinction separates, or 'abstracts', design from the act of implementation. Because designs are made regardless of the language of implementation, this

may be perceived as indicative of the 'abstract' style. Indeed, design is normally interpreted as "laying out at some level of abstraction, the pieces of the solution and their interrelations" (Pennington and Grabowski, 1990). Grundy identifies even "advance planning" as something that soft masters do not get involved in (Grundy, 1996, p.142). This is a valid extension of Turkle's position, in that 'some level of abstraction' is intrinsic to design. However, design in anything other than trivial computer programming is ineluctable - whether it is done meticulously and methodically, or directly between brain and keyboard, some advance planning takes place.

Edwards (1990, p.105) sees computers as "a medium for thought", and compares it as such with language: "In order to think with a computer, one has to learn its language". Design, however, also mediates between human thought and the programming languages 'understood' by the computer. The informal language of design - sometimes referred to as 'structured English', or 'pseudocode' - is intermediate between natural language and high-level programming language, between a user's requirements and a computer solution. It is a medium for thought that is positioned between the human and the computer programming language.

Brooks, in his *No Silver Bullet* paper, noted that the "essence of a software entity...is abstract in that such a conceptual construct is the same under many different representations" (1987). The software developer who thinks through a problem in terms of the domain of that problem rather than in the language of implementation, can represent a solution in any suitable language. Such a solution may be distanced from the computer, but it is closer to the problem domain in the real world.

5.3.4. Documentation and egoless programming

In order to communicate the purpose and functioning of program elements, programmers write not just the program code, but additional 'documentation'. Knuth (1984), in his concept of "literate programming", extends both the writing metaphor element of programming, and the bridging between the real-world problem domain and code. A "literate programmer" will write programs

explicitly as if they were essays for humans: the program code is incorporated into a comprehensive design document, incorporating a table of contents, conceptual and physical design, and full code documentation. In a study by Bertholf and Scholtz (1993), programmers set the task of completing an incomplete program were considerably more likely to find a solution if they were given such a 'literate' program.

Whatever its extent, location or manner of presentation, documentation is written in natural language, and may be separate from or embedded into the program code. It is written, not for the computer (which skips over embedded comments), but for people. Documentation might therefore be reasonably generalised as a human-communicative practice.

Turkle, however, characterises documentation as asocial, part of a culture of "individualism": to the concrete-style hacker, it is "a burdensome and unwelcome constraint" (Turkle, 1980, p.21). The magician of the machine wishes to share the secrets of their wizardry only with the computer. For Turkle, one aspect of soft mastery is that "the keynote of programming is not clarity but magic". The pursuit of clarity is seen as an obsession of hard masters, like the luckless Austen pupil Jeff, who "wants his programs to be clear so that he can share them and be famous". The "creation of a private world" is inevitably part of forming a relationship with the computer, and the development of "labyrinthine code" is one means of facilitating that privacy and mystery (1988, p.133).

Egoless programming is based on the need for open, shared and understandable code within a team-working context. It lies at the heart of the dominant 'formal' culture of programming, and is essential to teamwork on large projects (Weinberg 1971, p.72). The key to the decentralised, egoless team is "that no single individual feels private ownership of any piece of the program" – it is "a shared work product and decisions concerning it are reached by consensus" (Curtis and Walz, 1990, p.255). Abstraction provides communicative clarity in this free flow of information; and by expressing structure, it allows the parts of

a large programming project to be shared out. The sense of a development 'lifecycle' - where the whole is larger than the individual parts - may reasonably be characterised as more holistic than the solipsism of the programmer who sustains an intimate relationship with the computer.

The concept of "programming in the large" is a central concern. Even advocates of independent styles analogous to soft mastery, who believe that good programmers "leap intuitively ahead, from stepping stone to stepping stone, following a vision of the final program" (Green 1980), recognise this:

> Development laboratories need freedom to experiment; but in commercial software production of large systems that are intended to have a long life, each program has to interact with other programs, written and maintained by other people, over long periods of time, and the impetus must be towards standardization and simplification – similar solutions to similar problems, standard coding and documentation styles, and formalization of change procedures. The non-conformist...would experience very strong pressure to change. (p.306)

Curtis and Walz (1990, p.256) identify creativity as well as complexity as two of the conditions favourable for an egoless team structure. They also note that "most of the empirical research on software development has been performed on individual programming activities" because most psychologists engaged in study have not been social or organisational psychologists, and it is more difficult and expensive to conduct team and organisational studies. While the psychosocial study of group and organisational dynamics – whether, for instance, a programming team functions better on egalitarian cooperation or authoritarian leadership (Weinberg, 1971) – is both interesting and diverse, we are concerned only with the impact of group and organisational factors on individual programming style.

In the 'formal' culture that Turkle eschews, 'problem' programmers are ego programmers, they are "territorial", and resist peer review (McConnell, 1998). Her 'concrete' programming culture, defined by "negotiation and experimentation with the machine" and by an antipathy toward documentation and design (Turkle and Papert, 1990), is inevitably in tension with the 'egoless'

aspect of this culture. The account Turkle and Papert give of children (Anne and Alex) and college students (Robin and Lisa) is of learner programmers with a narrow focus on self-satisfaction: while this is probably inevitable with learners assigned to small problems that do not require collaboration, Turkle and Papert are emphatic that this approach is a complete "different way of knowing", an equal style rather than an early evolutionary stage of knowing.

Indeed, the differences between the formal and concrete cultures are not always clearly exemplified: how Anne's mask sprites are to be rendered visible and invisible without "designation" and "command" is not, for example, explained - nor how this differs significantly from showing and hiding the birds directly. Such a solution does not reflect reality, or treat birds as birds: birds 'disappear' and appear on the horizon, rather than fly around with shields; they are also liable to pass or land on objects that are not sky-coloured, or to fly across skies of irregular appearance. And on the other hand, the alternative 'structured' solution uses elementary rather than structured components - though a structured programmer would use a single composite data structure (such as an array of records) rather than a multitude of individual variables; and structured programming would designate variables by meaningful identifiers rather than by the letters Turkle suggests. Anne's feeling that she is in the company of the birds may be a subjective feeling unconnected either to feminine 'connectedness' or to the fact that she makes mask sprites disappear rather than bird sprites.

Software concerns the real world, and other people. Fundamental problems arise if people - and women in particular - are led by Turkle and Papert's analysis into adopting an 'ego' approach. Software is developed by teams of people within real-life scenarios involving real people. From benefit clerks and claimants to airline pilots and passengers, they have real needs. However unfortunate it may be, those needs are unlikely to be met by an approach that treats programming as a private, self-satisfying relationship with the computer.

5.4. Psychology, context, and tactility

Turkle transfers theoretical concepts from disparate fields into the study of computer programming, without any suggestion that such a transfer might be anything other than entirely unproblematic. Keller's focus on "a feeling for the organism" or object of study, and Gilligan's "ethics of care", are applied very generally, without any justification for the application, or much clarity as to what it is they are applied to. While traditional studies of the psychology of programming are largely concerned with cognitive psychology, the most fundamental lens of analysis which Turkle uses is that of psychoanalysis. Just as cognitive psychology has not necessarily transferred well into computer based learning in the form of instructional design (Laurillard, 1993, pp.74-75), so the imposition of psychoanalytical theory rests uneasily on top of the intricacies and subtleties of programming discourse and practice. A key problem is that concepts and terminology that are mapped, by means of analogy, into other analytical fields, have contexts within programming that are specific and significantly different.

5.4.1. What context?

Turkle and Papert conflate their concrete, hacker style with Carol Gilligan's description of a countercultural, concretely contextualised (as opposed to abstract) style of moral reasoning. The hacker style is thereby associated with a feminine, connected style. Also adapted is the notion, from object relations theory, that females are more given to attachment because they do not have to separate their identity from the mother. Differentiation, objectivity, and precision are said to be inherently masculine (Turkle 1984a, p.109), their gendered status being derived from the natural experience of the foetus and the infant. To Turkle, the 'images' of object relations theory "suggest" a relation between programming style and gender (1984a, p.108). As women are assumed to be good at forming relationships, they are further assumed to want "a personally meaningful relationship with a computational object", to instinctively need to "treat the computer as much like a person as they can" (Turkle and Papert, 1990:145; 149). If females are strongly linked to a concrete style of reasoning, and are less 'abstracted' from interior space, then, it is

asserted, they will be naturally predisposed to a 'concrete' rather than an 'abstract' style of programming.

Gilligan, however, refers to concrete reasoning in terms of real-world contexts. Turkle translates Gilligan's sense of the "importance of attachment in human life", into women's preference for "attachment and relationship with computers" (Turkle and Papert, 1990, p.157;150); Gilligan's need to "stay in touch with the inner workings" of arguments, into the need to stay in touch with the inner workings of the machine. The concreteness of the hands-on, experimenting programming style resides therefore in the computer rather than in the real world of the client or user. It may conversely be argued that it is actually abstraction which allows for Gilligan's "contextual and narrative" mode of thinking. The expressiveness of abstraction allows reasoning within the context of the real world, and provides the 'language' to bridge between human understanding in the real world and its representation on computer.

Without this bridge to the real world, it is difficult to make sense of a program. In an empirical study of forty professional programmers' strategies for comprehending program code, Pennington (1987) refers to "a cross-referencing strategy", whereby programmers "think about both the program world and the domain world to which the program applies while studying the program". Such programmers are compared to those who adopt strategies "in which programmers focus on program objects and events" or on domain objects and events, but not both. The results showed that the programmers who attained high levels of comprehension were those who adopted the cross-referencing strategy. This supports Brooks' concept (1983) of program comprehension as one of reconstructing and cross-relating knowledge about the problem domain with other domains such as the algorithm and language domains. While Brooks insists that the process of understanding is necessarily a top-down process, the essential process of refining an initially vague hypothesis by reading increasing levels of detail of the code could easily be reframed to allow more flexible and mixed-level strategies.

The computer-centric culture of programmers is often blamed for the frequent failure of computer software: they have communicated with the technology rather than with the people who need and will use their software. Indeed, the "millennium bug" would not have arisen if programmers had been more connected to the real world than to the immediate burden on their computer's 'memory'.

5.4.2. Bricolage

We have already encountered the 'classic bricoleur' in Alex, the nine year old Hennigan pupil. Turkle's use of this concept dates to 1984, when she first associated hard and soft mastery with Lévi-Strauss's "discussion of the scientist and the *bricoleur*" (103). Western science and hard mastery are identified as "a science of the abstract", and 'preliterate' science and soft mastery as "a science of the concrete". The term "bricolage" later again denotes the "concrete science" of what Turkle and Papert call non-Western "primitive societies" (1990, p.135). The hard master "thinks in terms of global abstractions", where the soft master tinkers: "arranging and re-arranging" concrete elements of the problem. The hacker's penchant for "experimentation" invokes a comparison with tactile experimentation in the sciences: the 'soft master' with a 'concrete' style is designated a 'bricoleur'. Notwithstanding that scientific experimentation is an integral part of mainstream 'canonical' science (as is implicitly acknowledged by references to scientific discovery and Nobel laureates (1990, p.130)), it is not in any meaningful way analogous to programming by trying things out on the computer. There is also perhaps an implicit primitivism whereby women as well as "primitive" societies are automatically associated with what is regarded as 'natural'.

Yeshno and Ben-Ari (2001) pick up on Turkle and Papert's adaptation of the term, but they "reserve the term bricolage (and use it more or less pejoratively) for *aimless* trial-and-error". Aimlessness in this context means that the trial-and-error does not lead to "refinement of concepts", and ultimately concentrates "on task performance rather than on understanding". Their study explores the importance of teaching "an explicit conceptual model". They cite the

observation of Smith et al (1993), that knowledge can often be constructed on top of misconceptions, where misconceptions are defined "as prior knowledge that is used outside of the context in which it is viable". Turkle would presumably question the notion of what is or is not viable, but this observation is perhaps equally applicable to her own use of knowledge from different contexts in the study of programming psychology.

Bricolage represents a "style of organisation", and is one aspect of 'soft mastery'. A second aspect of the soft style is filled out at the same time: the psychoanalytical concept of "closeness to the object" is linked to object-oriented programming (Turkle and Papert, 1992). This is a potentially complex issue, and will be dealt with separately.

5.4.3. "Abstract": what's in a word?

Turkle frequently interprets the meaning of abstraction in programming as broadly synonymous with the everyday use of the word. The term 'abstract' can indeed be used in this general way when talking about programming. When Kahn (1999) writes about "concretisations" (in relation to his logic programming environment for children) as "mappings between programming language abstractions and tangible objects", for instance, he is referring to concepts that are 'abstract' in the normal sense – such as clauses and constants in a programming language.

When Turkle and Papert present a 'concrete' style of programming, it is as a hands-on style that is intimately closer to reality, lends itself to a communicative, 'soft' style, and is thereby a 'feminine' approach. 'Abstract' is taken to be the opposite of the 'concrete', and is interpreted simplistically as an antonym to concrete - as in "an abstract idea" (1990, p.131). Jeff, the young hard master from the Austen school, was seen to take an approach that was said to be 'abstract' because he saw the Logo sprites as "something apart from his everyday life" (1984a, p.105).

Abstraction in programming, however, is not so simple. Abstract does not mean, as it does in fine art, something that has no representational qualities. In program design, abstraction signifies the removal of complex computer-specific detail and its replacement by human-specific actions and things. In a digital world, it is abstraction that facilitates a human-recognisable representation of reality. In Turkle's own words (referring to a word processor):

> The art of writing a program....that must be accessible to the computationally unsophisticated, consists of designing an abstract, artificial world so that it feels like a familiar physical one. (Turkle 1988, p.194).

Detail is concealed within 'abstractions' that are more 'concrete' and less complex. The sense of the abstract in programming discourse may therefore be argued to be directly antithetical to that of 'abstract' art: while in abstract art it is figurative representational details of reality that have been removed, in program design and development abstraction translates detail of the computer's reality into detail of the user's reality.

Where Turkle and Papert associate the concrete style of reasoning with a "closeness to objects" (1990, p.147), however, it may be argued that the concrete 'objects' of the machine are either real microchips, bits, registers, buses, disks; or virtual interface 'objects'. Consequently, abstraction can actually be viewed as the means by which the 'concrete' objects close to the machine are used to represent the real world: it is only by abstracting low-level detail that software can move closer to the human user.

In breaking down a problem, abstraction also provides scope for creativity and expression. There is seldom one orthodox 'reality': the 'ideal' representation will inevitably vary according to the individual (Cox and Brna, 1995). Indeed, the stronger the machine's role in the solving of a problem, the less expressive and communicative the process is likely to be for the programmer(s).

A further confusion with abstraction is the frequent failure to distinguish between teaching programming in an 'abstract' way, and 'abstraction' as a tool in problem solving (Turkle 1990; Stepulevage and Plumeridge 1998).

Discovery-based learning and reinforcement through practice does not represent an approach to programming, concrete or otherwise, but rather a pedagogical approach to the *teaching* of programming. One of the advantages of such an approach is that it can serve to build up a genuine understanding of abstraction: abstraction can be used and taught in practical ways. While the amount of exemplification and practice may frequently be inadequate on programming courses, there is also often insufficient emphasis on making sense of software development through abstraction and its practical benefits. Indeed, Hoadley et al (1996, p.109) advocate "instruction that emphasises abstract understanding", and can go on to indicate, without inconsistency, that learners need "practice and concrete examples" in order to gain an abstract understanding of patterns and reusable templates for problem-solving .

5.5. Transparent and opaque boxes

In the same way as an assessment of 'abstraction' must depend on what it is that is being abstracted, and from what, so the nature of transparency is defined by what it is that can be seen through the transparency: in the case of the black-boxing of routines in programming, whether it is the machine or the task that is visible or invisible. Turkle briefly glimpses this ambiguity in *Life on the Screen*, where she refers to the Macintosh's transparency as "somewhat paradoxically, a kind of transparency enabled by complexity and opacity" (1995, p.42). The confusion, however, runs through her analysis.

On the one hand, the soft-mastery of Turkle's Harvard students is defined by their desire to experience programming as transparent - hence their resentment of opaque black boxes: Lisa, a poet, "resents the opacity of prepackaged programs" and wants to take them apart or write her own (1990, p.128). She is "frustrated with black-boxing or using prepackaged programs" (Turkle and Papert, 1992, p.7). Oddly, however, George, a hard-master physicist, in language remarkably reminiscent of that used to describe the resentment the soft master feels for black boxes, also "feels threatened by opaque objects that are not of his own devising" (1995, p.39). In terms of computer operating systems, Maury, a sociology student, prefers the "old-time modernist

transparency" of MS-DOS and Windows over the opaque Macintosh, because "Windows is written in C" and his ability to program in C means he can see into it (1995, p.38). It is also a hard master, Henry, who "revels in technical detail" (1988, p.133). Soft-master and Apple aficionado Joel, however, unlike George and Maury, loved "to create programs that were opaque" so that one had no idea how it worked and one "could forget that there was something mechanical beneath" (1995, p.40).

The notion of transparency is therefore deeply ambiguous in Turkle's analysis: soft masters insist on transparency in computational objects, but they also do not want to understand the computer completely. Hard masters (and hobbyists), want the computer to be transparent so that it can be under control: and 'transparency' in code is part of their pursuit of clarity and precision. However, it is extremely difficult to insist that the software be transparent, but not the hardware – given that, as a virtual machine, software ultimately works by relating to the physical machine. The use of structured programming is regarded as a quest for clarity, "using nested subprocedures to give programs a transparent, hierarchical structure". Hard masters work to "make transparent" the logic of their processes. The source of a soft master's "personal, magical power", however, must be "buried deep and hidden within the machine" (1988, p.133).Turkle therefore hints at a distinction between software and hardware transparency, but does not explicitly address the issue.

Robin is told by her teachers not to take apart the prepackaged programs, and not to concern herself with "what was going on at that low level" (Turkle, 1988, p.59; Turkle and Papert, 1990, p.134). In this way, teachers are said to impose the orthodoxy of black-boxing on unwilling women. Yet in Life on the Screen, Turkle favours the 'opaque' Macintosh interface because it "hid the bare machine from its user" (1995, p.23), and is antithetical to "the traditional modernist expectation that one could take a technology, open the hood, and see inside" (1995, p.35). She goes on to dismiss the "reductive understanding" of those who want to "open the box" (p.43), and

approvingly observes that we "have grown less likely to neutralize the computers around us by demanding, 'What makes this work?' 'What's really happening in there?'" (1995, p.42). Turkle herself was uncomfortable with the old command language interface of her Apple II because it embodies the hard master theory "that it was possible to understand by discovering the hidden mechanisms that made things work" (p.33). It is said to have been designed "to make it easy for people who wanted to get at its 'innards'" (1988, p.192).

However, where software development is concerned, black-boxing makes structured programmers "exultant": they "feel a sense of power when they use black-boxed programs" because it is not to be changed by others (1992, p.16). Yet Lisa herself, while she may resent programs packaged by others, and does not appear to consider letting other people use her programs, makes her own black boxes. She does, after all, prefer "to write her own, smaller, building block procedures" (1990, p.133).

It is puzzling that this is taken to exemplify a non-structured approach: it appears in fact to be a clear example of a novice programmer carrying out structured programming and 'divide-and-conquer' techniques. These "building block procedures" may not be prepackaged, but they serve as 'black boxes' nonetheless. Within the same article, Turkle and Papert maintain that it is the soft master who makes the "demand for transparency" (1990, p.133), and when confronted with a prepackaged program written by someone else "wanted to take it all apart" (p.134). Confusingly, they go on to indicate that it is **hard** masters who demand "transparent understanding" of programs *made by others*, and want to know "how the program works" (p.140).

To write "her own, smaller, building-block procedures", Lisa would have to make use, at some level in her programming language, of procedures that have been prepackaged by others. Each level up the ladder represents a more "abstract" level than the level below. The higher level 'language' allows the programmer to use functionalities or data structures without knowing the details

of how they are implemented at a lower level. At each level, the unnecessary detail below has been abstracted. In a very general sense, the entire enterprise of computer language is to move towards people and away from computers, by means of abstraction.

In using any feature of a language above binary code, programmers are using a "prepackaged routine", an abstraction, a 'black box'. A simple integer calculation - such as 2 + 2 - makes use of an abstract data type that is usually built into a programming language: we do not think about the definition of the data type and that of the addition operation. As Sebesta points out, "all built-in types....are abstract data types" (Sebesta 1993, p.375). The *Write* procedure in Pascal, for example, enables Pascal programmers to write text and numbers to a screen, printer, or disk. They do not need to know *how* it does this - only how to use it. As we saw in Chapter 4, such prepackaged routines are coded using lower-level language: a lower-level 'out' procedure would write a given byte to a given computer port. While the items which *Write* deals with will be of a humanly recognisable type - words and numbers - the code that implements it will deal with variables of computer-understandable types such as WORD and BYTE, and will include hexadecimal numbers, memory addresses, processor registers, and assembly-language commands.

Abstraction and prepackaged black boxes are everywhere in user-friendly computer programming: from the use of a meaningful variable identifier (rather than a raw memory address), to the use of a control tool for drawing screen objects in Visual Basic (rather than the low-level plotting and drawing of that object) - or third-party VBX custom controls that may be plugged in and used. Within a team working on a large project, where nobody is able or needs to understand everything at a low level of detail, a divide-and-conquer breakdown of the problem is fundamental. Each sub-group or individual will produce routines that are black-boxed so that others understand how to use it, rather than how it works inside. The future of programming may well rely on reusable components which end-users can plug together to meet their needs.

Procedures may be 'prepackaged' as part of the language, by the software house that developed the language compiler, the writer of a commercial library of routines, or by another member of one's programming team. A procedure encapsulates a number of operations that work together to achieve a single purpose. It can be called by name when needed, without the necessity to repeat the full sequence of its constituent operations, and combined with other procedures to perform more complex functions. In design, a procedure usually corresponds to one of the stepwise refinement (divide-and-conquer) sub-task statements of what the program needs to do.

In many programming languages, routines are defined and implemented separately. This provides a means for hiding the internal workings of modules (the implementation) from any program or programmer that uses them – while separately specifying precisely how each procedure should be used (the definition). The program, or programmer, therefore only knows what they need to know in order to make use of the routine. This sealproofs the server module, as a client program cannot inadvertently change some aspect of the server module. In this way, black-boxing dramatically reduces the possibilities and complexities of errors and side-effects, and greatly simplifies maintenance, modification, and the isolation and correction of bugs.

5.6. Conclusion

Turkle and Papert misquote Gilligan as speaking of the "importance of attachment in human life" (Turkle and Papert, 1990, p.157): like the male hackers they model the 'concrete' style on, they appear to curtail and lose sight of the larger "life cycle" perspective (Gilligan, p.23), and choose a narrow individual attachment to the computer rather than a holistic and social understanding of the real world and its human contexts. In so misconceiving the nature of computer programming, they may unwittingly consign women to an amateur computer role that is wholly inappropriate and more in keeping with the original literal French meaning of *bricoleur*: one who tinkers around at odd-jobs (as in the French D-I-Y chain, Monsieur Bricoleur!). The idea that structured programming, and abstraction in particular, is inimical to women or anyone with a holistic style of learning, is potentially quite damaging. The

aesthetic and elegance of abstraction, its power to invest complex problems with context as well as understanding and resolution, should mean that it is considered in more depth, beyond superficial analogy. If properly understood and used, it may even have the potential to make programming, and computing, available to everyone.

We have seen that the concept of an 'object' – and one's closeness to it - has been important to wider considerations of gender difference, and it is perhaps not surprising therefore that the object-oriented programming paradigm has been drawn into the discussion concerning programming style and gender. Such a discussion can bring to its sharpest focus our analysis of the sense in which abstraction is in fact strongly linked to real concreteness.

6. Aspects of the concrete: objects, and visual programming

6.1. Introduction

Having critically scrutinised the process whereby "computational objects" have been instantiated for the 'objects' of object relations theory within Turkle and Papert's gender analysis of programming, we will now examine the nature of 'objects' in object-oriented programming (OOP). This is a distinct area, which Turkle has identified (1992, 1996) as specifically suited to, and illustrative of, the concrete style. Along with other cognate aspects – such as multimedia and web authoring – it also represents the cutting-edge, up-to-date face of computer programming, and is for that reason also essential to the development of our analysis.

A key characteristic of 'soft masters' and their "concrete style of reasoning" is closeness to, and identification with, what they call "computational objects". The proximal use of objects is counterposed, as a way of thinking and knowing, with "the rules of logic" – the "abstract formulae that maintain reason at a distance from its objects". This is then related to object relations theory: children develop by forming either "a proximal or distant relationship" to objects, and their use of either "the abstract and analytic or concrete and negotiational style of thinking follows." The analysis is said to be based on Evelyn Fox Keller's (1983) identification of canonical science and objectivity as a separation of scientific objects from everyday life. This separation is said to reflect the "earliest experiences" of men, whereby they are left "with a sense of the fusional as taboo", and an investment "in objective relationships with the world" (Turkle and Papert 1984a, p.115). Keller (1983) describes Barbara McClintock, a Nobel laureate in biology, relating to chromosomes to such an extent that she "wasn't outside", but became "part of the system". Although Keller comments that this fusional experience is also experienced by male scientists, Turkle hypothesises that McClintock could more fully exploit such an experience "because she is a woman" (1984a, p.117), and states that this is "surely the case for the girls in the Austen classrooms". There is a will to

transpose object relations theory as gendered, onto computer programming: 'computational objects' in general can be related to "in a way that is not separate from the experience of the self". They are likened to "transitional objects" in psychoanalytic theory – objects such as baby blankets or teddy bears that "mediate between the child's closely bonded relationship with the mother and his or her capacity to develop with other people who will be experienced as separate, autonomous beings" (p.118). The existence of object-oriented programming appears to be at least a terminological gift to such an analysis.

6.2. From object-relations to object-oriented?

Turkle and Papert herald object-oriented programming as the fulfilment and "revaluation" of the 'concrete' approach (1992, p.29): it is, they say, "more congenial to those who favor concrete approaches", and "puts an intellectual value on a way of thinking that is resonant with their own" (p.155). Turkle even indicates that the object oriented programming paradigm "associated computation with the object-relations tradition in psychoanalytic thought" (1995, p.296). Grundy too equates object-oriented programming with a concrete (as opposed to abstract) style of programming, interpreting the growing popularity of the object-oriented paradigm as "a surge of interest in a concrete style of programming" (1996, p.142).

Objects are called 'objects' because they frequently model real-world objects. Yet texts on object oriented programming repeatedly emphasise that an object is also a black box – indeed, objects are the ultimate in black boxes. The programmer knows and works with an object in the way that a customer uses a vending machine: the customer need not know how the machine works, only its state (whether specific drinks and change are available, how much money has been entered), and the actions to be performed by the customer and the machine. The details of the machinery are kept hidden and secure from the customer, to avoid misuse, but also because the customer simply does not need to have access to it.

6.3. Object orientation

6.3.1 How people think?

The respective proponents both of logic-based programming (as exemplified by languages such as Prolog), and of object-oriented programming, have advanced both paradigms as more 'natural'. Mendelsohn et al (1990), however, suggest empirical research shows that logic "is not how people think" (p.25). In respect of object-oriented programming, Green (1980) correctly points out that is far from being 'artless': programming 'objects' are very complicated yet at the same time do not reflect the intricacies of real world objects. Object oriented programming is not, therefore, 'natural'. Nor is it necessarily easy. Indeed, various aspects of object oriented programming render it more difficult to learn than traditional structured programming: in particular, identifying object boundaries, and getting objects to communicate with each other, are highly challenging aspects of OOP.

In her personal account of life as a software engineer, *Close to the Machine*, Ellen Ullman (1997) tells how object orientation was a test of whether she was "technical". As a project manager, she says that "there is really only one thing programmers want to know: are you technical or not?". Her particular 'test' was to locate a problem within some object-oriented C++ code:

> Object-oriented software: small hunks of code, understandable only if you know a whole hierarchy of logic. Tiny window, fifty-line viewport, to see little blocks in an elaborate pyramid. Murk and confusion. (Ullman, p.112).

6.3.2. History of Objects

Objects are fundamentally based on the earlier code-packaging concept of the procedure. The technique and philosophy of object oriented programming can be traced back to the design of the Simula 67 programming language in 1967. Simula was inspired by an earlier language, Algol 60, and in particular by the use of reusable packages of code in the form of procedures or sub-routines. The fact that Algol's procedures created their own mini-environments, ones that were both individual and self-contained, led to the innovative concept of

allowing such 'environments' to persist after they had 'finished', so that they could communicate with other such entities. It is probably significant that Simula was a language designed to build simulators: it was intended to develop software that ideally reflected real-world structure. The concept was developed by researchers at Xerox, who developed both Smalltalk, and the first graphical user interfaces.

6.3.3. Encapsulation: abstraction's child

In traditional 'imperative' or 'procedural' programming, the "unit of thought" is said by Turkle and Papert to be "an instruction to the computer to do something", but "in object-oriented programming the unit of thought is creating and modifying interactive agents within a program for which the natural metaphors are biological and social rather than algebraic" (1992, p.31). The language of object oriented programming does indeed use such metaphors: child, parent, inheritance, message-passing, properties. However, the essence of objects lies in encapsulation, and in the separation of what an object does from how it does it.

Abstraction is fundamental to object oriented programming: as Sebesta puts it, "object-oriented programming....is an outgrowth of the use of data abstraction" (1993, p.375). Objects represent the guaranteed level of information hiding that was provided by abstract data types that use opaque data types: the access procedures are encapsulated along with the data, so that access to the data is strictly and intrinsically limited to the access procedures. While the philosophy is different from that of functional decomposition, object oriented programming is nonetheless a fulfilment of the promise of the 'abstract' approach, not, as Turkle and Papert would have it, of the 'concrete' approach. The features of object oriented programming are clearly indebted to the characteristics of what they call hard mastery.

Turkle and Papert observe that "hierarchy and abstraction are valued by the structured programmers' planner's aesthetic", and that the formal approach "decrees" the design of "a set of modular solutions" (1990, p.136). On the other hand, soft mastery is characterised by a "non-hierarchical style" (1992, p.9).

Hierarchy, abstraction, and modularity, however, are defining characteristics of object orientation. Gorlen et al (1990, p.1) refer to data abstraction as the "necessary foundation" of object oriented programming. Encapsulation in the object oriented programming paradigm is synonymous with "information hiding" or black-boxing: it provides a "cover", in Satzinger and Ørvik's words, "that hides the internal structure of the object from the environment outside" (Satzinger and Ørvik, 1996, p.40). The object-oriented paradigm abstracts data as well as processes, and binds the processes to the data - in a way not dissimilar to the 'hard master' who would create her or his own abstract data types (ADTs) in separate modules with a public functional interface and a private implementation. An 'object', normally an abstraction of an entity in the real world, is therefore roughly analogous to a module containing ADTs: it is the ultimate 'black box'. Its defining characteristic is that no other object needs to be aware of its insides: its internal data structure, and the methods for manipulating instances of it, are hidden. It is the behaviour, rather than the internal implementation, of an object that matter. An object prevents client programs from directly accessing its internal elements; it makes data elements and their operations

> behave analogously to the built-in or *fundamental* data types like integers and floating-point numbers. We can then use them simply as black boxes which provide a transformation between input and output. We need not understand or even be aware of their inner working...(Gorlen et al, 1990, p.1).

What Turkle and Papert disapprovingly present as the formal, hard master's alternative to Anne's "dazzling" bricolage with the masked birds (1992, p.15), is therefore in fact a basic object-oriented strategy:

> From their point of view, Anne should design a computational object (e.g., her bird) with all the required qualities built into it. She should specify, in advance, what signals will cause her bird to change color, disappear, reappear, and fly. One could then forget about "how the bird works"; it would be a black box.....Structured programmers usually do not feel comfortable with a construct until it is thoroughly black-boxed, with both its inner workings and all traces of the perhaps messy process of its construction hidden from view. (Turkle and Papert, 1990, pp.139-140)

In object-oriented programming, the class is the major unit of programming. It "represents the abstraction of a number of similar (or identical) objects" (Bell and Parr, 1998, p.624). The more general the class is, the more 'abstract' it is: for example, the class of mammals is an abstract class because, while specific instances of mammals exist, mammals as such do not exist. In Java, a truly 'abstract class' exists only to serve as a modifiable template, and cannot be instantiated. In order for this to be useful – and, indeed, for classes to be reusable - the characteristics of classes in the hierarchy must be inheritable.

6.3.4. inheritance: connected hierarchy

Hierarchy is clearly a key organising principle for any large body of information – taxonomic classification helps to manage complexity and information access. It is analogous to decomposition in structured programming, in that it requires a breakdown of elements according to a structure that represents different levels of abstraction. As with abstraction generally, the higher levels focus on the essentials, and leave the unnecessary detail till later. In object-oriented programming, the filling in of those details – along with any changes or variations - is neatly handled by inheritance. The same methods, inherited from the most abstract class (the parent, or superclass), can be used to manipulate a range of objects of different classes (the child, or subclass); and anything unique to a particular subclass, can be kept within that particular subclass.

Satzinger interestingly borrows the example of a (non-gendered) baby from object relations theory in order to illustrate the concept of object hierarchy and at the same time show that hierarchy is a natural way of learning and organising information.

The new-born baby's model of the world initially consists of mother (or primary carer) and other things. This gradually extends to mother-people-other things, refining to parent-people-living things-other things, to parent-big people-people-living things-things. The characteristics and behaviours of instances of each category are inherited down the classification hierarchy: a

person, for example, inherits the properties of a living thing, and adds some extra just for itself. This process, Satzinger says, "allows the baby to infer information about newly encountered objects" (11), both in terms of new classes (e.g. small people) and instances of classes (e.g. a new aunt). This notion of a connected 'hierarchy', based as it is on set membership and linkage rather than on value ranking (as conceptualised by Gilligan), can be seen as more analogous to Gilligan's sense of context and association. Learning is premised on a facility to link new things to that which is already known:

> Learning something new often means associating a new concept with a previously known concept while the new concept "inherits" everything known about the previous concept. (Satzinger, p. 42).

This is made possible by the combination of abstraction with inheritance, and is antithetical to the "soft master" desire to break apart rather than reuse that which has already been accomplished and packaged. Although they laud object-oriented programming as concrete and in tune with the soft-feminine bricoleur style, Turkle and Papert's characterisation of the canonical style actually reflects these characteristics of the object-oriented paradigm:

> The bricoleur scientist does not move abstractly and hierarchically from axiom to theorem to corollary...[the canonical approach] decrees that the "right way" to solve a programming problem is to dissect it into separate parts and design a set of modular solutions that will fit the parts into an intended whole. Some programmers work this way because their teachers or employers insist that they do. But for others, it is a preferred approach; to them, it seems natural to make a plan, divide the task, use modules and subprocedures. (Turkle and Papert 1992, p.12)

6.3.5. messages and polymorphism

Much of the language of object-oriented programming is based on communicative metaphor. Interactions with and between objects involve sending "messages" back and forth (rather than 'input and output', or 'commands') in order to perform one of the actions associated with the object's data. These 'object relationships' whereby objects may be associated with each other are often defined through inheritance. The terminology of communication

theory is used to describe message-passing: a 'sender' sends a 'message' to a 'receiver'. Where a soft master such as Turkle and Papert's Alex insists on "repetitions of instructions" (1990, p.137) to get a feel for the program, messages obviate the need for duplication of data, and help to keep the detail packaged inside objects discrete.

Meaning is a dynamic process in object as in human communication: the same 'message' may be interpreted in different ways by a variety of different receivers. This principle is described in the object oriented programming paradigm as 'polymorphism'. A corollary of inheritance-based messaging, polymorphism allows for the same method (message) to be reused and interpreted differently by different types of black-boxed objects. The same command can be sent to different objects, and it is the object that decides what action is appropriate for the command. This cornerstone of authentic object-orientation - message-passing and polymorphism - therefore also falls, as Graham notes (1994, p.14), "under the general heading of abstraction".

6.4. Java: different kinds of 'black boxes'?

Java is the most popular object-oriented programming language in use today. We will examine two aspects of it that can elicit significant nuances in the meaning of black boxing and abstraction. Firstly, that there is a distinction between black boxes and abstraction, and the deferral of detail in the learning process. There are features of some programming languages that make it educationally difficult if not impossible to explain or understand everything at once: this is a significant educational issue, but it is categorically different from the deliberate deferral of detail that is used in programming abstraction. Secondly, however, many programming educationalists attempt to overcome some of these difficulties by using their own black boxes.

6.4.1. 'Hello World'

The '`Hello World`' program - or variations thereon – is frequently presented in introductory programming texts to illustrate how a simple program is constituted in a particular language. Before the shift to popular object-oriented

languages, Universities typically used programming languages such as Pascal and Modula-2 to teach programming. All aspects of a `Hello World` program written in these languages were immediately explainable. The program would typically read as follows:

```
PROGRAM HelloWorld;
BEGIN
      Write ("Hello World!")
END.
```

In Java, however, the most basic program immediately presents complexities. `Hello World` typically reads as follows:

```
public class HelloWorld
{
      public static void main ( String args[ ] )
      {
            System.out.println( "Hello World!");
      }
}
```

Deitel and Deitel (1999) 'explain' the first word, the *public* keyword, thus:

> Several times early in this text, we ask you to simply mimic certain Java features we introduce as you write your own Java programs. We specifically do this when it is not yet important to know all the details of a feature to use that feature in Java. All programmers initially learn how to program by mimicking what other programmers have done before them. (p.38)

This is typical of the way in which Java is introduced in textbooks. The idea that it is not necessary to "know all the details of a feature" in order to use it, appears to echo the concept of the black box. However, there is an important and fundamental difference, and it is expressed above in the word "yet". The programmer uses black boxes as a programming tool, and does not at any point need to know how it works. On the other hand, the programmer *does* need to understand how every part of the 'Hello World' code works. The fact that some basic features of Java may not be immediately explainable, is a learning and teaching problem, one that may be intrinsic to the design of the Java language. This deferral of detail in teaching programming is therefore completely distinct from the deliberate use of opaqueness as a tool in program design. Full understanding is being deferred. In some ways this pedagogical problem might

be identifiable with the problems that Turkle and Papert associate with the programming tool of abstraction.

A holistic learning approach might require that the learner understand all that they need to know from the outset. Instead, the Java texts consistently defer this understanding. In Pascal, the program will begin simply with the word "`begin`"; in Java, it begins with the header `public static void main ( String args[ ] )`. Bailey and Bailey explain that the *public* prefix makes the class or method accessible, but that the keywords *static* and *void* "are technical terms we will get to later" (p.3). Parsons bravely attempts to explain *static* as well as *public* and *void*, but indicates that "[t]he meaning of the square brackets ([]) will also be explored later" (p.22). Bell and Parr indicate (1998, p.18) that the only piece the reader needs to understand "for some time to come" is the output line - g.drawString("Hello", 50, 50); the line of code `public void paint(Graphics g)` is just 'explained' as "a heading specifying that the statements that follow are the statements that paint ...the window". Rebelsky (2000), referring again to the main method header, says that "For now, we won't worry about why it looks like this; it just does." (p.3). Savitch (2001, p.28) advises readers "for now" to "ignore the following few lines that come at the start of the program", explaining that they "set up a context for the program, but we need not worry about them yet". Savitch goes further and passes over the `println` statements too, explaining that "for now" the reader can consider these lines to be "a funny way of saying 'output what is shown in parentheses'".

Skansholm (2000) comes close to explaining every part of his `Hello World` method header. When he gets to the parameters of `main`, however, he falters: "We will not go further into this now but will merely state that this is the form it must have" (p.12).

The typical approach in programming texts is to provide the 'Hello World' code and then examine the pieces of the program. Bishop (2001) takes a slightly different approach. She does not attempt to explain anything about the `hello world` program code beyond what it does: "At this stage we shall note only a

few points about the program" (2001, p.17). Her emphasis, however, is on the student running the program, and changing the message. Through concrete experimentation, the student gains familiarity with the 'physical' programming environment.

6.4.2. Java packages and classes: prepackaged routines

The difficulty of the main method header is not the only way in which Java is front-loaded with complexities that are inappropriate to the level of the beginner. Another major difficulty which arises early on concerns user input. In Pascal and Modula-2, keyboard input is read from the user by the direct use of simple built-in procedures, such as `Read`. In Java, reading and processing input is much more complicated: input objects need to be created, and the initial input will then usually need to be converted to an appropriate type, using further object-oriented notation, before it can be processed. To display the result, the processed data would then often have to be converted back into a type compatible with the output object used to display it on screen!

Rather than subject beginners to this sort of detail, textbook writers normally provide their own simple methods for input. This means that the authors write their own 'packages', which provide simplified routines that the beginner can use with ease. A problem with such packages is that they are non-standard; however, one of the major features of Java is the package, and the possibilities it provides for customisation and reusable libraries of routines.

6.5. Visual programming

Turkle and Papert appear to conflate graphical interfaces with object oriented programming: using computer software at the interface, with authoring that software as a programmer. They state that the graphical user interface typified by the Macintosh and its icons, reflects a deeper "philosophy of 'object-oriented programming'". The Macintosh's replacement of command language with "concrete icons", they declare, "has theoretical roots in a style of programming usually called 'object oriented'".

The debate concerning differences in user interface dialogue-styles has been widely described (Booth, 1989; Schneiderman, 1998), and relates to the underlying concept of a "psychological machine". However, visual programming is quite distinct from the pervasive graphical user interface. The SWEBOK guide (Bourque et al, 1999) breaks software construction down into three major styles of construction interfaces, one of which is visual (the others being linguistic and mathematical). While visual programming environments (such as Visual Basic) and customisable software components appear to blur the distinction between programmer and user, using software to process data is categorically different from designing and implementing the set of data structures and instructions that facilitate the performance of these specific content-based tasks. The purpose of human-computer 'dialogue' is usually conceived of in terms of "control" of an "information exchange". Sheehy (1987) suggests that we can best acquire this information by suspending disbelief at the interface and imagining that we are communicating with another person (cited in Booth, 1989). Turkle appears to take this suspension of disbelief somewhat more earnestly, and as more than a means to an end.

Using graphical user interfaces such as the Macintosh operating system or Microsoft Windows does not, of course, involve the user in object oriented programming. What is probably being suggested is that such interfaces, as well as presenting a concrete representation of data and actions that is more directly manipulable, follow an object oriented design and implementation philosophy. It may be further deduced that perhaps, instead of using programming environments analogous to the user's command language dialogue, the programmer could use a programming environment analogous to the graphical user interface. Rather than type "proposition-like commands", the programmer could – like the Macintosh user - manipulate "concrete icons".

Object oriented programming languages, however, are not necessarily or intrinsically linked to graphical environments. Many compilers are text-based, and ultimately all programming languages are text-based languages. Visual environments are now often superimposed on languages – usually, but not

always, object-oriented or object-based – to make programming easier and more powerful. Such graphical development environments (or 'Integrated Development Environments') provide packaged functionality and powerful tools – for instance, the developer can simply drag a button onto a form, rather than write all the code themselves. Such a button is a concrete 'object' in the immediate sense that it is a potentially manipulable and functional user interface object. However, it is also an 'object' in a deeper sense: it is a programming 'object', implemented usually as an instance of a class.

In Java, for example, an on-screen button may be implemented by instantiating an object of the class `Button` (or `JButton`, depending on which library is being used). Usually this is a constructor which takes the button's visible label ('OK', 'Next' or whatever) as a parameter. A visual integrated development environment will allow the programmer to do this by dragging and dropping rather than by typing in the appropriate code, and by simply typing in the caption. This will generate the necessary code (which the programmer would otherwise write).

```
myButton = new Button("Welcome");
add (myButton)
```

This code is relatively simple because the Button class is a library class that has already been written: all that is left to do is create an object that is an instance of that class, using the `new` constructor. The real concealment is that of the class code that underlies the button – whether it is created by dragging and dropping at the user interface, or by keying in the actual program code at the keyboard.

This distinction between an on-screen interface 'object', and the underlying software 'object', is fundamental to Turkle and Papert's misunderstanding. On-screen interface objects are normally implemented as 'objects', and there will be a close correlation between the two senses of the word. An interface push-button, for instance, is experienced in terms of its properties (such as its colour, or its caption) and its actions (such as what happens when the mouse pointer moves over it, or when the user clicks on it). It is also typically implemented as an instance of a button class, which will contain methods for setting its colour and other attributes, and for handling specific events.

However, such objects are merely the means by which the user interfaces with the main functionality of a program: the objects involved in that functionality are drawn from the real world rather than from the world of computer interface conventions. A major shortcoming of the only detailed study of programming that Turkle provides, is that the only objects which the artistic Austen children program are screen graphics: the objects they are manipulating have intrinsic graphical representation on the computer screen. An object in real-world functionality will be any 'thing' relevant to the domain of the software being developed: if it were a payroll program, an object could be an employee, the data associated with it might include name, address, and age instance variables inherited from a parent class (such as Person), along with its own data (such as job title and salary). An object could also perform computer-specific functionality: a specialised data structure for compressed encoding of a text or music file, or another one for handling a print queue. While the Austen children may imaginatively project themselves into spaceships and flocks of birds, the data objects that actually arise in modern software development are rather more difficult to represent and identify with on the computer.

6.6. Authoring Software and Scripting

The development of media-rich software raises new questions about approaches to software design, and about levels of abstraction, that do not naturally arise with the development of data-oriented software. Authoring software provides a higher-level software development environment than programming languages such as Java and C. Their key features are the level of pre-programmed elements for developing interactive multimedia, and scripting languages that are integrated with these components. They allow for significantly easier and faster software development than traditional programming languages: their emphasis is on rapid development rather than efficiency. To a considerable extent, interfaces and even functionality can be built, without any programming, but simply by using commands and tools that are available from the interface - in the same way as they are in a word processor. However, the ability to write

scripts and attach them to the interface-created elements, provides the developer with considerably more power and versatility.

The general metaphor of authoring software is that of an "author" – perhaps a little grander than that of a writer. This aspect is reflected in the fact that 'authors' do not always have to program; and when they do, it is 'scripting' rather than 'programming'. More specific artistic metaphors include, as we shall see, 'pages' in a 'book' – although these pages are more 'constructed' than they are written.

There are two taxonomies commonly used to categorise authoring software. The first relates to the extent to which programming or scripting is required, as opposed to interface-driven development; and the second is based on the metaphor used for the construction of software. Probably the most intuitive metaphor is the page-based metaphor. Apple's Hypercard was the first popular example of this: 'cards' containing different media assets – text, graphics, audio, video, animation – were organised into 'stacks', and contained hyperlinks between each other. ToolBook requires authors to place media on 'pages' which together form one or more 'books': links can be made between pages, and more intricate functionality and interactivity can be scripted into any object on any page.

Macromedia Director, and its scripting language, Lingo, is used to produce stand-alone interactive multimedia software, and highly functional Shockwave material within web browsers. Its metaphor is reflected in its name: the software developer is a 'director' of a 'movie', and needs to create a 'cast', members of which are moved onto a stage, and their sequencing and interaction arranged and scripted in a 'score'.

From the perspective of teaching people to program, authoring software has the potential to allow people to 'program' as users initially – using interface-driven commands to build screens, place objects, and set up basic interactive functionality. When they are confident with this environment, they can move on

to 'script' the same objects, within a familiar environment. This moves beyond what Bernstein (1992) and Frenkel (1990) saw as a new curriculum paradigm for computer science - (female-friendly) package-based problem solving. It is more convincingly computer science: it does not eschew programming languages, but still allows students to achieve something functional quickly. Such rapid prototyping provides pedagogical scaffolding, encourages exploration, and boosts confidence by allowing the development of significant software products.

The scripting languages of authoring software are broadly similar to programming languages, but they operate at an even higher level. This is reflected in their syntax, and in their use. The syntax is more like English or pseudocode, and is accordingly much less rigid than third generation programming languages. They are generally interpreted languages. The metaphor of interpretation versus translation suggests the distinction between interpreted and compiled languages. Unlike compiled languages – which translate an entire piece of code into a self-contained executable program – lines of code written in an interpreted language are performed immediately, within the interpreter and without separate translation. There is also no need (and often no way) to declare type information for variables. The primary function of scripting languages within authoring software is to control media assets, the behaviour of media and interface elements, navigation (including hypermedia); however, they have the full range of control structures and operations, and can also be used for normal algorithmic and data-oriented development. It follows that it is often considerably 'easier' to script in a scripting language than in a third-generation programming language. Its looser syntax, its function as a 'glue' that can dynamically combine separately written and packaged components, and the fact that it is generally interpreted rather than compiled, tends to lead to a more experimental and spontaneous style. This is not – as we shall see – necessarily synonymous with Turkle and Papert's 'concrete' style.

Authoring software is usually event-driven and object-based. Event languages are oriented around the actions of users at the graphical user interface – such as

mouse clicks - and as such are commonly regarded as user-oriented, "since the user is in charge of generating events that the application handles" (Myers et al 2000/2002, p. 217). While proper, data-oriented object-oriented scripting is possible in some environments, 'object-based' broadly refers to screen 'objects', the hierarchy they must be placed in, and the messages that they pass to each other in accordance with this object hierarchy. A further aspect of this model is that of 'properties'.

Properties are analogous to variables, but they are attached to an object rather than defined separately. As such, they can usually be either built-in (as part of all instances of that type of object) or user-defined (for a particular object). They are manipulable and definable both through the user interface, and by direct scripting. In terms of the task that Turkle and Papert's bricoleur programmer, Anne, performs with the flock of birds, both the current and the original colour of each bird would be stored in properties that were attached to the birds, rather than in 'distant' variables. Each bird would carry with it, as a user property, its original colour, along with – usually as a built-in system property – its current colour. These properties – along with visibility, size, and position properties - would be a more intuitive and integral representation of a bird, enabling it to (in Turkle and Papert's words) "exhibit a greater complexity of behavior" than if it carried a camouflage shield around with it (which, for Anne and for Turkle and Papert, is "how the bird works"). For example, a bird could change, within itself, its colour or visibility in relation to other internal properties such as its position. Further properties (or sometimes functions) can typically also be used to handle relative group behaviour, such as intersection or touching. Whatever the behaviour, it would be triggered by a signal – a message that is sent, perhaps 'spontaneously' during the course of animation, when the user initiates an event, or when the developer explicitly sends the message. This is made possible by the fact that the construct is (again in Turkle and Papert's words) "thoroughly black-boxed" (1992, p.16).

It is clearly deeply problematic to characterise such a process as "formal" and "abstract", and antithetical to Anne's 'intuitive' and 'concrete' approach. The

root of the problem lies again in a misapprehension of the nature of black boxes, and in superficial practical assumptions about 'transparency' and 'concreteness'. Authoring software provides opportunities to develop objects, each of which is both highly concrete, and has "all the required qualities built into it" (1992, 16-17). It is the technical, computer-oriented detail of "how the bird works" that is hidden in a 'black box' and forgotten about. Its "inner workings and all traces of the perhaps messy process of its construction" are "hidden from view" – to enable it as a model of a bird, rather than of a computational object.

6.6.1.Prepackaging in Authoring Software: behaviours and widgets

Authoring software packages are defined by their high-level 'packaged' functionality: interfaces, structure, navigation, and basic interactions, can be built without the need to program. They enable a much wider range of people – for example, graphic designers, linguists, educationalists - to develop software. More of the developer's focus can be concentrated on the purpose of the software - on its external rather than its internal functionality. In this respect, authoring software represents the ultimate in black-boxing, in high-level prepackaging, and thereby liberates programming from its more exclusive and specialised technical and mathematical roots.

However, the black-box versus glass-box issues that these authoring packages raise, can be a little more complicated. Part of the black-box/glass-box question under scrutiny is the distinction between just looking inside a glass box, and altering or creating its contents oneself. With regard to the core functionality of authoring software packages, at the highest level it is generally no more possible to view the 'prepackaged' code than it would be to inspect the code behind the functionality of a word processor. While some software developers may well resent the lack of power and fine control offered by such packages in comparison with third generation programming languages, for the confirmed user, inspection of underlying code is not an issue. Where the possibility of inspection does arise, however, is in the use of 'additional' functionality that is provided in the form of library palettes or catalogues of drag-and-drop objects.

Such functionality is not built into the core interface of the package, although it can be achieved by scripting. The drag-and-drop library objects again bypass the need to script for oneself - but because they are external library objects, they generate script code and place that code where the developer would normally write their own.

For example, in a Macromedia Director 'movie' the default state is for the movie to play straight through to the end. One of the most basic tasks is therefore to keep it still at particular points, so that static 'screens' can be displayed, awaiting the user's actions. This can be done from the user interface by using a prepackaged 'behaviour' from a Library Palette. The appropriate behaviour, titled "Hold on current frame", is dragged from a Library Palette, and dropped onto the frame in question. This process is very quick, and provides the required functionality (as long as the behaviour is dropped at the correct location and on the correct object). There are similar facilities (known as "widgets") made available in Asymetrix Toolbook – which we will look at shortly.

Dropping the 'Hold on Current Frame' behaviour onto an object generates the following Lingo script, which is immediately available for inspection:

```
-- DESCRIPTION --
on getBehaviorDescription me
  return "¬
HOLD ON CURRENT FRAME"&RETURN&RETURN&"¬
Drop this behavior into the Script Channel of the Score
or onto the Stage ¬
in order to keep the playback head in the current
frame."&RETURN&RETURN&"¬
PARAMETERS: None"
end getBehaviorDescription

on getBehaviorTooltip me
  return "¬
Frame behavior."&RETURN&RETURN&"¬
Holds the playback head still."
end getBehaviorTooltip

-- HISTORY --
-- 3 November, written for the D7 Behaviors Palette by
James Newton

on exitFrame me
  go the frame
end exitFrame
```

If, on the other hand, the developer had chosen instead to write their own code, it would have a decidedly less intimidating appearance:

```
on exitFrame
      go the frame
end
```

The verbosity of the behaviour relates primarily to documentation that is deemed to be necessary for the 'user' of the authoring software, in the form of descriptions that appear in the Behaviours Palette – but it is all copied into the user's own code. In this case, seeing inside the box is considerably more daunting than composing it for oneself.

The core handler code that is generated, is almost the same, but is more complicated again because it operates at one remove: 'me' is the current instance of the behaviour – a parameter that identifies the object which the behaviour is currently attached to, and which is receiving the event. Again, a conceptual meta-level has been introduced. In the case of this particular behaviour, it is generally unnecessary: the practical purpose of the parameter is to enable access to the properties of the individual object that the behaviour is attached to. In order to hold on a frame, however, it is not necessary to know anything about any properties of that frame.

Widgets perform a similar function in ToolBook. Simply by dragging a widget from the widget catalogue, dropping it on the page, then using a combination of menu commands and dialogue box completion, one can display a video clip in a page. When this entirely interface-driven process is completed, the developer can view the script that has thereby been generated and attached to the frame in which the video plays:

```
to handle buttonclick
      send ASYM_Trigger to self
end
```

This appears relatively simple at first sight – except that the handler sends another message to its object. The `ASYM_Trigger` message is in turn defined at a higher level of the object hierarchy. The task of this message, however, is not restricted to displaying the video clip in the frame: because it is a widget, it

needs to translate from the general to the specific: it needs to handle all inputs and eventualities that might occur during the interface-driven process of using the widget. This level of meta-functionality includes checking if various options were mistakenly missed out – if the developer neglected, for instance, to specify a video clip to play in the frame. It also sends several further messages - for instance, to set the video controls. This large script (about twelve pages of A4) also contains further functions and messages that it needs – to get the specification of the particular clip to be played, or to set the controls to those selected.

By way of contrast, if the developer chose *not* to use a widget, a minimalist script could be attached to a play button or an enterPage handler could be coded as follows:

```
mmPlay clip "myVid" in stage "Video Stage"
```

The more intricate the automated task is, the more complex the generated script becomes. ToolBook, for instance, provides widgets for generating quizzes. Altering – or even understanding – such generated scripts is again usually a hugely difficult task, because of the meta-layer that is needed to generate them from author-input at the interface.

6.6.2. Web 'languages'

The page metaphor is of course now commonplace in the form of the World Wide Web, where it is associated more with 'mark-up language'. Mark-up languages (such as HTML – HyperText Markup Language) are more analogous to desktop publishing than to coding: they are used to determine layout and appearance. The intention of mark-up languages is that the presentation of material should be separated from its actual structure. They are not 'languages' in any sense that programming 'languages' might be: they have no logical constructs (repetitions and selections), do not deal with any time-based process, with 'tags' that identify spatial elements of a document and render them in a particular way. While they are not procedural, the process of composing pages

in these 'languages' is frequently referred to as 'programming', and the composition as "source code".

The elegance and power of markup languages is that they separate the presentation of a document from its structure. In the case of the Cascading Style Sheets extension – which also uses the 'property' concept - content is also separated from style. The principle is one of modularity and decoupling, or 'abstraction', for the purposes of reduced complexity, flexibility, and maintainability. Increasingly complex and powerful extensions to the markup 'languages' that generally originate in SGML (Standard Generalised Markup Language) appear to bring the process of web page development somewhat closer to that of traditional 'programming'.

Web-based plug-in technologies frequently come with their own scripting languages. Animation software such as Macromedia Flash, utilises very high-level interface-driven pseudo-programming, as well as the additional facility of its own scripting language, ActionScript. In addition, Javascript code can be embedded into HTML with a <SCRIPT> tag, and interpreted, on the client machine, by JavaScript-aware browsers to perform data validation functions and change the appearance of the web page.

6.7. SELF: objects without classes

Classes are at the heart of the mainstream object-oriented paradigm. A class "represents the abstraction of a number of similar (or identical) objects" (Bell and Parr, 1998, p.624). Experimental research at Sun Laboratories, on a language known as SELF, attempts to dispense with these meta-objects altogether.

In visual programming environments, the programmer can inspect and alter the properties or state of objects – whether they are visual screen elements like the button, or conceptual objects such as employee records. Packaged functionality is clearly an example of 'prepackaged procedures': however, the tools used to view and edit both interface and conceptual objects, present a layer that

mediates between the object in the program and the 'object' on the screen. Chang et al (1995) argue that, while transparency in tools can lead to direct manipulation of objects, the programmer by default interacts with this layer of tools and views, rather than with the objects directly. This is a conceptual meta-layer that is in addition to – and on top of – the existing conceptual meta-layer (of objects as instances of classes) that dominates object-oriented programming.

ToonTalk (Kahn 1999) is described by its developers as a "game world...in which the objects and actions map directly onto programming language constructs". Developed for children, to help in learning the principles of logic programming, it provides "concretizations" for programming language abstractions in the form of "tangible objects". So, for instance, a clause may be represented by a robot, and putting robots into a truck may express a procedure call.

At first sight, SELF appears to do something similar. SELF is an object oriented language that contains neither classes nor variables, but instead uses prototypes as its means of creating objects. State information can only be obtained by means of such objects sending messages to 'self'. The developers of the language characterise it as a language for "exploratory programming", and its metaphor as one "whose elements are as concrete as possible" (Ungar and Smith, 1991).

Whereas traditional object-oriented programming languages use a plan or template based metaphor in creating objects via the instantiation of classes, SELF does so by directly cloning, or copying, a prototype. A prototype is more concrete in that it deals with examples of the object, rather than plans for it – plans that require interpretation. There is no meta-object in the form of a class. Without instantiation, relationships are simpler, insofar as there is no need to conceive of objects as both instances of a class, and subclasses of another class. Instead of two relationships, there is only the one inheritance relationship. And the emphasis on behaviour and message-passing further presents an object model that is intended to be more intuitively active than the traditionally

passive object. Objects are defined by what they do: "The only way to know an object is by its actions" (Ungar and Smith, 1991).

As has already been remarked, the meaning of abstraction in the ToonTalk context is a general reference to conceptual ideas that need to be learned. The typical object, too, functions as "a concrete analog" – a representation, a metaphorical mapping based on an environment that is familiar to the user. These objects facilitate conceptual learning, and also enable the user to generate rather than write code. Interface 'objects' in visual programming languages such as Visual Basic also generate code based on direct manipulation of tools and menus (though these objects do not function on the basis of analogy, but are what they appear to be – interface objects).

However, the 'object' in a language such as SELF is an actual programming object, rather than an analogue of such an object. The object is therefore represented directly as itself.

6.8. Conclusion

We have studied objects in programming because of claims that they represent an affirmation of the concrete style, and because they are such a fundamental aspect of modern programming. In relation to objects and style, we have reached the opposite conclusion, and found in them a valorisation of abstraction. As a concept, the 'object' has little direct relevance to object relations theory. Objects do manifest in the programming product as user-friendly direct manipulation visual interfaces, as software that more closely reflects the real world, and as modular, plug-in software, where common components are reused across packages, and may be significantly adapted to the individual user's needs. Objects therefore present a product that can reasonably be said to be highly 'relational', providing 'concrete' and more representative on-screen 'objects'. The same can also be said in the very different but parallel context of the programming process itself – but it is because of, and not contrary to, the techniques of abstraction.

The theoretical analysis of this dichotomy – between a soft-feminine, concrete style, and a hard-masculine, abstract style – has thus proceeded to a 'logical' conclusion at the object-oriented paradigm and beyond. The analysis has raised questions that have produced a substantial argument against the dominant hypothesis that identifies and genders these styles. The concept of abstraction and black-boxing has proven to be both the most crucial, and the most misunderstood, aspect of the argument. We will now consider how to exemplify this concept and thereby test the validity of the dominant hypothesis.

7. Methodology

This chapter justifies the overall research process, indicating the ways in which all of the major parts of the research work together to address the research question. It explains how the central research question was formulated, and discusses the issues that arise in making the question precise and testable. The importance of the preceding theoretical analyses as research is emphasised, and its contextual relationship to decisions made in collecting evidence is clarified.

7.1. The Literature Review

7.1.1. Multiple disciplines

A gender analysis of programming must inevitably be interdisciplinary. The research question concerns several different domains of knowledge – gender, how people learn, computer cultures, and programming among them. This interdisciplinary aspect, however, requires care: the input of different disciplines calls for fine integration and distinction, not wholesale juxtaposition and transference; studied translation rather than impressionistic transliteration.

Gender is clearly at the heart of the question: gender and feminist theory is a body of knowledge that interrogates inequities, deconstructs underlying 'differences', and seeks to explain attitudes, cultures and contexts. Perhaps most crucially, it can illuminate how and when other bodies of knowledge - and ways of knowing such 'bodies' – are socially and psychologically constructed, and how assumptions can define defaults and received wisdoms. In itself it is multidisciplinary, with considerable pertinent input in particular from psychoanalysis.

While particular premises have been drawn from gender analysis – such as the promotion of the position of women – and taken as essential to the research, it cannot serve as the only (or even the most prominent) body of knowledge in this study. The literature on gender and programming in particular has, to date, presented such an imbalance, relying on gender analysis at the expense of programming knowledge and theory.

The main context of the research is that of students learning to program - but the focus is firmly on the act of programming rather than the learning of programming. Because programmers never write the same program twice, the way in which people learn is part of the context. Yet the process of learning, and the process of programming, should not be conflated: distinctions between learning (or teaching) methods and styles, and programming methods and styles, are too easily blurred. A grounded understanding of programming is necessary in order to make these distinctions – along with some understanding of how and why learners and others do not understand.

These are diverse disciplines, but their discourses appear to intersect in the form of similar (or at least analogous) vocabulary. Issues concerning the concrete and the abstract, opacity and transparency, and relationships to objects, arise in gender studies and in programming; and pedagogy is concerned with moving from the concrete to the abstract. Without a proper depth of analysis across the different disciplines, this might present by default as a shared discourse. Yet even within the area of computer culture and programming, the same terminology can be used in different senses. A study of literature in each of the disciplines has therefore been essential to clarify the terms and expression of the argument. This review has been structured and ordered to move from the general, to the specific as informed by the general – though this does not imply any relative precedence or depth of study. However, given the multidisciplinary scale, the less specific layers must inevitably be representative rather than necessarily exhaustive in terms of the literature reviewed. The most specific literature on gender and programming has been scrutinised very closely, and the research relies heavily on this analysis.

7.1.2. Sources

7.1.2.1. Data

The statistical data concerning inequity have primarily concerned degree level Computer Science courses in the United Kingdom and in North America. Data from English-speaking countries is more readily available and accessible to the researcher. However, there is a more substantive contextual rationale: the research being critiqued was largely conducted among degree level students in

the USA, and the situated research in this study is to be conducted to reflect the context of those students, but to be located in the UK. There is also the argument that, in terms of programming theory and practice, the USA and UK are currently most definitive of computing culture and the programming 'canon'. Additionally, the statistical identification of Computer Science (as distinct from applications software and other courses) is clearer in the USA and UK than it is, for example, in many European countries. There is a clear relationship between the deeper nature of a society and representation in computing, and this will obviously vary to some extent between different societies; however, the review refers to surveys and literature which find that attitudes do not. It is also very generally true that, however different its manifestations, all societies may at present be characterised as patriarchal.

Programming, as a discipline that is important to careers in computing and programming, is a crucial part of Computer Science, and neither applications software courses, nor the question of whether Computer Science courses should be applications software courses, is substantively material to the question. Programming is simply the discipline under examination: there is no point in deferring or displacing this emphasis.

7.1.2.2. Literature

The theoretical material reviewed is again characterised by the predominance of US and UK sources. Again, this is the context of the study: it does not in any way suggest that either feminism or computing is only a US/UK concern – although some aspects of western feminism may be products of specifically north American culture.

The literature concerning gender and programming specifically, is limited. The analysis – and the research question – focuses on the research and articles of Sherry Turkle, along with Seymour Papert, firstly because this work represents the substantial part of the dedicated literature, and secondly because their analysis has been so widely and uncritically accepted. While it is likely that their analysis would not be so readily accepted within computing circles, this is

to be expected, and has not been articulated in detail or within the original terms of discourse and context of gender theory. The central critique is therefore focused on these authors: the critique is original, and represents the creation and interpretation of new knowledge.

7.2. The Question

The central research question that arises from the fact of unequal representation in Higher Education, and the analysis of the literature, is whether women have a negative attitude to black boxes in computer programming: whether abstraction in programming is 'inimical' to women. The question is of considerable significance: the techniques associated with abstraction are fundamental to the current teaching and practice of programming. Such techniques include the decomposition of problems into sub-problems, and the packaging of common functionality into discrete chunks of code that can be reused - and used by others - on the basis of knowing what these 'black boxes' do, rather than how they do it. It has been proposed - by Sherry Turkle and Seymour Papert principally, but also by others - that allowing the individual to reject these techniques is a prerequisite if "equal access" to computation is to be achieved; and that structured programming should be removed from programming courses in order to open up the field to women. The opening up of computing is thereby linked to the opening up of the black box, via broad associations with 'interior space' and holistic understanding. The validity of these associations - and therefore of the linkage between black boxes and gender – has been closely interrogated in the analysis of the literature, and in the light of the critical study of gender theory. In order to formulate the question more clearly, Turkle and Papert's analysis is expressed as a testable hypothesis: that abstraction techniques are more likely to 'put women off' computer programming than men.

A less 'grounded' expression of Turkle and Papert's hypothesis might be formulated, but would not readily meet the criterion of testability: the more theoretical and 'objective' idea that such techniques are 'inimical' to an epistemological style common in women, for instance, is not easily demonstrable by means of attitudinal survey or observation. Turkle and Papert's

research emphasises the subjective styles of programmers, and repeatedly illustrates these intellectual styles by citing attitudes to particular programming techniques.

Turkle and Papert's hypothesis concerns attitude and 'epistemology'. They clearly assume that students, of both sexes, hold attitudes to black boxes, and that these attitudes can be elicited by means of interview. Attitudes are notoriously difficult to measure - seldom expressible (if at all) in precise terms, and easily influenced by the research process itself. While other techniques – for example, the observation of actions - might or might not be more meaningful than the elicitation of attitudes, my research is in response to Turkle and Papert's theory. As such, it makes the same ontological assumption, that attitudes exist about black boxes and are knowable – and entails many of the same problems that are inherent to attitudinal research methods.

These problems are evident in Turkle and Papert's work, and it is essential that they should as far as possible be minimised in my own research. The vocabulary they use to describe attitudes is vague: abstraction is 'inimical' to women, women are 'frustrated with', 'resent', and do not 'want to use', prepackaged programs. The most specific common expressions involve 'wanting'. The alleged stylistic dichotomy is presented as a matter of 'preference' and 'tendency' as well as a 'stance'. The epistemological notion of a 'mode of thinking' is expressed in terms of 'openness' or otherwise to "a close connection with the object of study" as gendered ways of knowing the essence of programming. While it would be difficult to test the degree of distance or closeness in a person's "relationship with the object of study", the harm that abstraction might do to women, or to decouple any abstraction-related frustration from other causes, it should be possible to test the extent to which people want to use black boxes. This in turn would allow for the testing of the notion that there is a natural gender pattern in the extent to which women and men want or do not want to use, construct, and see inside black boxes.

The second aspect of terminology and meaning concerns what is meant by abstraction (and other programming-related terms), and what the 'object of study' is and should be.

The analysis in the previous chapters raised these questions, and considered, as particular issues, the identification of what abstraction means in computer programming and in other fields, and what proponents of concrete and glass box approaches mean when they would prefer not to use it. It was observed that 'abstraction' had been assumed to have a single meaning - to have the same meaning in computer programming as, for example, abstraction in philosophy and mathematics; that concepts valid within the study of science, psychology and ethical reasoning were loosely transferred into computer programming; and that in particular styles from music and poetry were also uncritically transposed into computer programming. The analysis concluded that it is the 'mathematical' implementation detail of a problem that abstraction techniques remove, so that it may be readily understood within the human domain: in other words, that the function and nature of 'abstraction' in computer programming was, in most respects, the opposite of what Turkle and Papert assumed it to be. This confusion was seen to be founded on the identification of the computer – rather than the problem domain and user context – as the object of study in programming for the concrete-style advocates, who therefore perceived abstraction or distancing from the computer as a characteristically masculine epistemological style.

Abstraction in programming is misinterpreted as the ascription of meaning without context, reflecting the sense of 'abstract' in pure mathematics or philosophy. More specifically, it is conceived of as a distancing from the object of study – which, in the case of computer programming, is assumed by Turkle and Papert to be the computer.

This meaning is ascribed at the level of interpretation: at the broad level of exemplification given by Turkle and Papert, they correctly identify black boxing, and the use of prepackaged procedures, as applications of 'abstraction'

in computer programming. While the theoretical meaning of the term 'abstraction' is disputed, the question can therefore use a practical meaning that accords with the real meaning of the term in computer programming, and at the same time reflects the substance of their field research and analysis.

7.2.1. The influence of the researcher

A key problem with attitudinal research is the extent to which the investigator's attitudes may influence the research. The analysis of the literature specifically indicates not only that Turkle and Papert's interpretation of the abstract and concrete in computer programming was confused – but also that their respondents often shared the misunderstandings of the researchers.

The observation that Turkle and Papert's respondents appeared to share some of their misconceptions, has significant implications in terms of methodology. As has been observed, the notion of objectivity has frequently been rejected within feminist methodology in favour of subjective experience. Maher and Tetreault (1993, p.31) liken the feminist researcher's relationship to their informants to that of teachers with their students. Pohl (1997) observes that Turkle's main source of data is interviews, "rather than the concrete practices they adopt", and cites Halpern's 1992 study as evidence that "the actual differences in cognitive style between women and men are less obvious than the images people form about them" (192). This additionally raises a problem that is perhaps more significant than the researcher's apparent closeness to her subject: the power relationship between researchers and informants where the relationship is in actuality a teacher-student relationship. There is some indication that Turkle's students reflect her ideas, and share her misinterpretations of fundamental programming concepts. There is a strong possibility that their observations and responses reflect Turkle's own psychosocial theories, and that these theories are reshaped on the basis of her presumptions concerning programming. (In addition, Turkle often uses descriptive vocabulary that subtly appears to lend weight to her thesis: Anne's "way of seeing", for instance, is said to be "woven" (1984a, p.110)).

There was clearly an equal danger in my own research that student respondents could be unduly influenced by what they felt their teacher expected of them. While care could be taken to ensure that there was no explicit set of expectations, the most ingrained dangers would concern the extent to which the curriculum and its delivery implied expectations of how students should approach programming. Turkle's students presumably overcame the alleged epistemological orthodoxy of their course at least in part through her assistance: if there is a 'canonical' style, then perhaps students might not otherwise overcome the pressure to conform. Some of these concerns could be allayed by a curriculum that did not explicitly address abstraction, combined with delivery methods that used concrete learning environments – and, of course, by respondent anonymity.

The decision in favour of anonymity has implications for follow-up interviews and descriptive accounts of the ways in which respondents experience programming. Respondents could only be asked to volunteer for interview. The limited extent to which this was achieved placed severe constraints on any possible follow-up interviewing: in the case of the first research site, a trivial number of males volunteered for interview; and in the case of the second, no females volunteered.

It was decided to use questionnaires rather than interviews as the primary source of information in order to minimise the researcher's influence. In interviews, the researcher participates in and shapes the conversation – and indeed needs to establish a rapport with the subjects in order to elicit responses. These responses then have to be interpreted – by the researcher - as attitudes. While survey questions may invite particular answers in the same way as interview questions frequently do, the researcher has less direct influence. The potential benefits of interviews - richer data, and clarity of understanding – can be added in by means of supplementary interviews after the completion of the survey. On the other hand, there is a danger that a questionnaire could impose clarity rather than explores attitudes, and could even furnishes attitudes that the respondents would not otherwise wish to express.

7.2.2. Environmental factors

Anthropologists and community sociologists claim that ethnographic studies enable them to obtain information that it is not possible to get otherwise. According to Wilson (1977), ethnography is based on "two sets of hypotheses about human behaviour: (1) the naturalistic-ecological hypothesis and (2) the qualitative-phenomenological hypothesis..." (1977, pp.247-249).

The naturalistic-ecological hypothesis holds that setting is a crucial influence on research, and should therefore be studied as an integral part of the research. As laboratory and field results can be very different, there is a need to conduct research "in settings similar to those the researchers hope to generalise about" (Wilson 1977, p.248). Particularly where the research concerns attitudes, these can be strongly influenced by "internalised notions of what is expected or allowed" (Wilson 1977, p.249). For this and other reasons it has been argued that responses collected by means such as questionnaires do not necessarily fully reflect the information that can be obtained from observing actions.

Although Turkle and Papert's main focus ins on explicit attitude, their study of programming styles among children involved field research: children were observed, over a period of time, undertaking programming tasks; observations were followed up with interviews. It might therefore be deduced that observation was a practical option in terms of my own research. However, it is significant that Turkle and Papert's observation work was restricted to children: their study of programming styles among adults appears to have involved interviews, but no observation. The programming that was performed by the children involved the simple manipulation of graphical sprites on screen: its processes were almost entirely visible as direct output, and could therefore be easily observed. The same is not true of the kind of programming that is taught in Higher Education, and which is commonly practised by professional

programmers in commerce and industry. The task of making significant and comprehensive observation of programming behaviour would be immensely complex even with a single individual. It would be very difficult to devise practical situations that were likely to provide a realistic opportunity for eliciting the extent of predilections for the use of black boxes. The mental processes of non-trivial algorithm development are very difficult to externalise in a demonstrable way, and it would also be extremely difficult to identify the reasons for particular actions or decisions – for example, between those taken because of a lack of knowledge or understanding, and those taken because of an attitude or inclination.

A key problem identified by ethnographers is – as observed earlier - the extent to which the subjects of research might have a "sense of the behaviour that is either appropriate or expected", and "desire to be evaluated positively" (Wilson 1977, p.248). The major issue in this respect for the current research, would concern the extent to which students might have been taught that using black boxes was a good thing. In the case of both studies – one at Liverpool Hope University College, and one at Manchester Metropolitan University – the level of the course was chosen to ensure firstly that the tasks represented would be understood, but also secondly because the concepts of 'good practice' in relation to black boxes had not yet been made explicit. However, the purpose and implementation of procedures (or methods) had been explained. On the other hand, many students found the concept of procedures with parameters difficult. And any sense of conformity that might have been transmitted by virtue of the teaching of procedures, may well have been balanced by the possible perception that they might also be expected to choose the more difficult options – which would involve rejecting black boxes. It is this problem that led to the decision to ensure that the questionnaires could be returned anonymously. It also contributed to the decision not to conduct observations.

7.2.3. Quantitative or Qualitative methodology

A quantitative strategy may be seen to imply a more impersonal and 'abstract' approach. Particularly within the context of the debate at hand, quantitative methods may alternately be seen as 'abstracting' experience, or as making abstract concepts concrete and definite. However, in the light of the literature analysis and the nuances and inconsistencies observed above, several compromising aspects of qualitative factors were carefully considered: the influence of subjective stereotyped perceptions on discourse; the influence of the researcher; the self-selecting nature of any volunteers; the likelihood that discourse and vocabulary could reproduce the misunderstandings of transferred concepts - such as what abstraction means in the context of programming – particularly within the context of the teacher-student relationship. Qualitative methodology would have the capacity to develop, on the part of both the researcher and the respondents, a dependence on gender, and a self-conscious knowledge of gender as an issue. Turkle and Papert's research with first-year programming students is characterised by the absence of clear exemplification. Quantitative methods had not been used before, and had the capacity to minimise side-effects; also, one of the weaknesses of previous research was the lack of attention to the specifics of programming in regular 3rd generation systems programming languages. Ironically, there is virtually no concrete illustration of 'concrete' and 'abstract' styles at work. Children are observed using Logo in the classroom, but the study of university students provides no practical exemplification. In the general epistemological terms apparently espoused by Turkle and Papert, we can best understand what it is to program in a concrete or abstract way, first by providing concrete exemplification, then theorising on that basis. To test whether these styles are gendered, a specific and representative aspect of typical introductory programming would need to be unambiguously and fairly exemplified.

The identification of a testable aspect, and the choice of examples - and questions - would therefore be crucial, and require very careful consideration. The concept of black boxes is fundamental to the concrete/abstract stylistic dichotomy, and the assertion that females "shun prepackaged procedures" and

"dislike the opacity of the black boxes" (Grundy, 1996) is central to its gendered interpretation.

It was therefore decided to use quantitative methods in order to minimise side-effects that might prejudice results, and to clarify meaning by the use of specific exemplification. The scope for subjective interpretation, misunderstanding, and prejudice, could be curtailed by making concepts concrete and presenting a statistical analysis of responses. To quantify attitudes, it would be necessary to formulate questions carefully, focus on the specifics of programming in regular 3rd generation systems programming languages, and ensure that gender acted as an independent variable.

7.3. The Surveys

A more detailed explanation of the process of sampling– and the context of the surveys - is given in the following chapter. This section sets out and discusses the salient methodological issues within the larger context.

To examine the validity of Turkle and Papert's central thesis, it was decided to ask questions of a sample from a broadly similar population to that used in their research. In this respect it was decided to focus on the student rather than the school population. As has been observed (in chapter 3), Turkle and Papert's children were highly atypical in that they were the product of an ultra-selective process that isolated a handful of the most talented and enthusiastic children from a larger body that had itself been selected for a uniquely immersive computer experience. This part of their study ignores the less responsive and talented individuals even within a highly selective larger population. Additionally, the programme of activities that the children were engaged on, appears to have been specifically driven by constructionist principles which explicitly seek to encourage bricolage and concrete styles. To replicate or parallel such a scenario could not provide a valid or neutral testing ground for the hypothesis.

The student population does not any present such immediately obvious problems: representative populations were available, along with representative curricula. Again, this will be explored in more detail in the following chapter.

The choice of questions was also crucial: a key intention was to provide concrete exemplification of what it meant to use prepackaged routines. The questionnaires and results are discussed in more detail in the following chapter. What follows is presented as methodologically underpinning.

7.3.1. Hypothesis testing: terms of reference

The hypothesis that women are more likely to reject the techniques and 'way of thinking' of abstraction is questioned by the exposition of the theoretical contradictions and weaknesses of the hypothesis. This process undermines the hypothesis; however, it cannot render the hypothesis superficial, given its significance.

The hypothesis is reasonably clear, but it is equally clear that it requires practical interrogation; whether it is in a form that permits it to be thus tested, will depend on further exploration of the meaning of the terms in which it is expressed. It is necessary in particular to clarify what is meant by black boxes, and then to consider how a hypothesis about attitudes to black boxes might reasonably be tested.

7.3.1.1. What a black box means

The concept of black boxing in programming has a wide connotative remit: it is used, for instance, to describe a method of testing and debugging a program; and can also imply a top-down approach to problem solving. Turkle and Papert use the concept in its generally understood sense: they repeatedly equate blackboxing with opacity and abstraction – with detail being hidden. They explain blackboxing as "the technique that lets you exploit opacity to plan something large without knowing in advance how the details will be managed". Something that is "thoroughly black-boxed" has "both its inner workings and all traces of the perhaps messy process of its construction hidden from view"

(1992, p.16). The term 'black boxing' is directly conflated with the use of opaque library procedures: "black-boxing or using prepackaged programs".

The equation of black boxes and 'prepackaged programs' is repeatedly emphasised throughout Turkle and Papert's publications. Lisa and Robin (who feature as a prominent part of the argument in all of Turkle and Papert's joint papers, and in Turkle's two books) want to write their own procedures rather than "use prepackaged ones from a program library". Lisa "resents the opacity of prepackaged programs" (1990, p.128). Again, in Turkle's *Life on the Screen*, Lisa is said to prefer "to write her own smaller subprograms even though she was encouraged to use prepackaged ones available in a program library. She resented that she couldn't tinker with the prepackaged routines" (1996, p.53). What is described is a dissatisfaction with library procedures, and a desire to "take it all apart" and look at what's "going on at that low level". It is this attitude that Turkle and Papert characterise as more likely to be a female trait.

The use of prepackaged routines is not quite all that is meant by black boxing – but it is the strongest exemplification of it within the terms of Turkle and Papert's argument. Turkle and Papert do occasionally appear to treat 'top-down' as synonymous with abstraction: when a top-down approach is referred to, it is used loosely as a shorthand for an 'algebraic' approach that uses opacity. So Anne's approach is generally said to be opposed to "the top-down approach", but as an approach it specifically "dramatises what was so salient for Lisa and Robin: the desire for transparency". The use of variables to store sprite colours is identified as an obstacle to transparency, and is also referred to as 'algebraic' because it assigns the colour of each bird to a different variable. Anne's nervousness about black boxes is elaborated as her not wanting "to package her constructs into opaque containers"; it is explained that "in engineer's jargon, it [how the bird works] could be treated as a black box". Anne's example – analysed in Chapter 6 – indicates that black boxing is inimical to the concrete programmer not just in terms of the use of third-party library routines, but also as a strategy for writing and organising their own

code. In 'Life on the Screen', Turkle conflates the notion of 'structured programming' with the separation of subprograms:

> After you create each piece, you name it according to its function and close it off, a procedure known as black boxing. You need not bother with its detail again." (1996, p.51)

Programs, or segments of code, are "black-boxed and made opaque": the "black box" is "designed not to be touched".

However – as Chapter 5 illustrated - rejection of a top-down approach does not equate to a rejection of abstraction. The separation of different levels of abstraction does not have to be top-down in nature: it can commence at the top, the bottom, or somewhere in between (e.g. Clancy and Linn, 1992). Hayes-Roth and Hayes-Roth (1979) refer to this kind of approach as "opportunistic planning", in which, according to Pennington and Grabowski (1990), "the plan exists at different levels of abstraction simultaneously and the planner continually alternates between levels".

7.3.1.2. Exemplification

Exemplifying the use of black boxing would therefore appear to be relatively trivial: prepackaged routines are an intrinsic part of most introductory programming courses. Two aspects of this triviality, however, raise concerns: firstly, prepackaged routines are unavoidable in any high-level programming language; and secondly, bypassing such routines is undeniably more difficult. While both of these issues point to deeper weaknesses in Turkle and Papert's analysis, both must be addressed. It is impossible to ascertain what constitutes a satisfactory level of primitives: in her earlier work, Turkle refers to her prototype concrete 'programmer X' wanting all elements of the program to be available to the programmer "as primitives" (1980, p.17). While the meaning of this cannot be ascertained in the absence of concrete exemplification, it was felt that it would be unreasonable to focus solely on packaging built into either the programming language or the compiler. It was therefore decided that routines which had explicitly been prepackaged by a known third-party, over and above this inherent 'prepackaged' prepackaging, should be tested. The issue of

difficulty may appear to be more intractable: many students have a perhaps natural inclination to avoid difficulty! However, while there is the possibility that this might prejudice against wanting to take prepackaged routines apart, this should not impact on the question of gender preference.

Relative difficulty is something that Turkle explicitly (and perhaps proudly) recognises as a concomitant of soft mastery. It is acknowledged that Lisa and Robin's insistence on doing their own "'building-block' procedures...makes their job harder" (1988, p.49). Anne's way of making birds appear and disappear "doesn't make things technically easy", and on the contrary requires unnecessary "technical sophistication and ingenuity" (1984a, p.113). All of her soft masters do not need to look inside the black boxes, and they know that they are choosing the difficult route. The fact that students 'do not need to know' how something works is the crux of the black-box question: it is all the more necessary to try to establish who actually wants to know. A black box 'abstracts' the detail that the programmer does not need to know, so that they can concentrate on what it does, rather than how it does it. Turkle and Papert's hypothesis is that the learning and programming style of concrete programmers (such as Robin) compels them to want to know how it does it – to make the black box transparent, or else rewrite it themselves.

It was decided to devise concrete examples to test this inclination, and to ask learning programmers a small number of closed questions about their attitudes towards specific prepackaged routines, then conduct a statistical analysis of the results. While a quantitative strategy might be seen to imply a less contextualised and more 'abstract' approach, this was adopted precisely because of some of the problems seen to arise from Turkle and Papert's more qualitative research: the likelihood that discourse and explicit vocabulary could reproduce similar misunderstandings of transferred concepts (such as what abstraction really means in the context of programming); the prejudicial influence of the researcher's perspective and background, and the expectations of self-selected volunteers; and the probability that a self-conscious knowledge of gender as an issue would arise.

The 'concrete' style of programming is characterised by specific instantiation rather than general abstraction; the methods adopted here are founded upon the assumption that it is also desirable to make specific what is meant by the concepts of 'concrete' and 'abstract'. The experiential discourse of Turkle and Papert's first-year programming students inclines towards abstract generality and conceptualisation: in particular, it is characterised by an absence of clear exemplification.

7.3.2. Sampling

While the population of interest was in a broad sense people learning to program, the individuals who expressed the attitudes and styles that Turkle and Papert's thesis is based upon, were students in the earlier part of undergraduate programming courses. It was therefore decided to sample this population, and to do so in institutions with different profiles and different courses. On the one hand, Combined Honours students from an Arts background were chosen to reflect most closely the background of Turkle and Papert's sample. On the other hand, this particular population was largely female, and therefore unrepresentative of the national statistics. It was therefore decided to take a second sample, this time from a more traditional Computer Science course in a male-dominated student-body and Department. While they were also first-year programming students on Combined Honours degree courses, they were all studying for a BSc at a large 'new' University. The degree was located in the Department of Computing and Mathematics, which in turn forms part of the Faculty of Science and Engineering. Both the Department and the Faculty have –unlike Liverpool Hope – an overwhelmingly male profile.

To minimise extraneous pressure, it was decided to survey the students at a point in their course where they had not been explicitly taught that abstraction and black boxing was a good thing. In addition, to again minimise the assumption of predetermined positions, it was decided to conceal the gender theme of the questionnaire.

The first sample size was 41 (24 female, 17 male), and the second was 50 (11 female, 39 male). Given the use of quantitative methods, a larger sample would have been preferable. However, this was not available within the respective institutions, and the intention was to treat each group – as a product of a particular micro-culture - as a coherent whole.

While the samples were in many respects similar to Turkle's, the strategy adopted, both in formulating the questions, and analysing the responses, was very different.

7.3.3. The questions

In order to test the hypothesis concerning black boxes, therefore, attitudinal questions would have to be formulated that were specific and applied, in a way that would reflect the meaning of the central research question *and* be meaningful in the context of the students' actual experience. The specific detail would need to be such that a reasonably full range of students would understand the questions sufficiently to respond in a way that accurately represented their attitude. In order to quantify results, responses would be gauged on an ordinal scale, and a $\chi 2$ test then used to compare available female and male responses. Questions were therefore closed, and the attitudinal scale was limited to three points to better measure attitudinal dichotomy. Some triangulation would be conducted by means of interviews with respondents who indicated that they were willing to talk in detail about their attitudes.

The hypothesis would be tested by establishing whether female students learning to program largely shared Lisa and Robin's dissatisfaction with prepackaged programs from a program library, and whether they too wanted to 'take it all apart' or write their own. Prepackaged programs were exemplified both in terms of library routines provided within the larger software development environment (Visual Basic), but also in terms of higher-level routines that had been prepackaged at course level by a tutor known to the students.

7.3.4. Distribution

The programming units at both institutions were delivered by a team of several tutors. The questionnaires were therefore distributed, with a minimum of additional information, by the tutor with whom each group was familiar. Students completed and returned the questionnaire at the beginning of a laboratory session, at a time of year when they knew enough to understand the questions, but were not under immediate pressure to prepare assessed work.

7.3.5. Analysis

The objective of the sampling was to establish whether the sample of attitudes fitted the pattern proposed by Turkle and Papert – or whether it fitted a pattern that might be suggested by our critique As the original pattern was broadly conceived as a predilection on the part of one gender for or against black boxes – and the suggested alternative either no significant difference, or the opposite predilection - it was decided to use a Chi-Square test for goodness of fit. This test would measure the extent to which observed response frequencies differed from the frequencies one would expect, given the null hypothesis of no difference between women and men, and conclude whether or not the observed data fitted the distribution indicated in the null hypothesis.

7.3.6. Interviews

While it did not prove practicable to directly follow up anonymous questionnaires with interviews, it was still desirable to temper and triangulate the quantitative data with finer, more impressionistic accounts. It was decided to conduct a limited number of interviews about programming, with gender and abstraction explicitly on the agenda. To best fulfil this agenda, it was decided to restrict the interviews to female students who in terms of programming knowledge and experience had moved significantly beyond the level of the surveyed students, and who had some additional qualities. Knowledge of a range of programming languages, expertise in psychology, and musical expertise, for example, could all provide strong links to particular facets of Turkle and Papert's ideas and personal accounts. Interviews were also a means of bringing gender out into the open, to see if this made a difference. And it

offered a way of representing women's voices and subjective experiences, as a correlative for the more impersonal 'objective' analysis.

These interviews were arranged informally and by invitation, on the basis of the researcher's personal knowledge of individual students and their interests and backgrounds. In most cases the interviewees were well known to the researcher. Interviews took place within the respective Universities, and generally included an open question about the student's experience as a female programming student, followed by questions or comments related to their particular expertise, and to their experience of black boxes. The first interview took place in 1997, and the last one in 2002. One was conducted by e-mail, and the rest were conducted in person - of which one was recorded on tape, the others initially recorded in handwritten notes. While the topic of research was made explicit to interviewees, the author's own views and interpretations were not at any point discussed. At the time of interview, none of the students was studying on a programming course for which the author had any responsibility; two were studying on another (non-programming) course for which the author had responsibility; the rest were not studying – or going to study – on any course for which the author had responsibility. Two had already completed their degrees.

7.4. Conclusion

The research question has been justified as significant, and examined and clarified in several relevant contexts, including feminist methodology; qualitative and quantitative research methods; conceptual confusions across different discourses; and the researcher- respondent relationship. A method for testing the hypothesis that the teaching and use of abstraction techniques puts women off computer programming, and to a greater extent than men, has been carefully constructed in this light. Given the critical nature of the review of the literature on programming style, particular care has been taken to ensure that the test is reasonable and non-trivial. The rationale for the initial sample has been explained, and the data from this survey will now be examined.

8. Exemplifying the Concrete

8.1. Rationale

Turkle and Papert's central thesis is substantially based on interviews with undergraduate students in the relatively early stages of a programming course. To examine its validity, it was therefore decided to ask questions of another group of first-year programming students with an arts background, and follow this up with a similar survey of a more traditional student body. The choice of questions was crucial: a key intention was to provide concrete exemplification of what it meant to use prepackaged routines.

Within the context of the issues that have arisen, quantitative methods as such might be interpretable as an 'abstracting' of experience; on the other hand, it may be pointed out that the 'concrete' style is characterised by specific instantiation rather than general abstraction, and the methods are founded upon the assumption that it is also desirable to make specific what is meant by the concepts of 'concrete' and 'abstract'. The experiential discourse of Turkle and Papert's first-year programming students inclines towards abstract generality and conceptualisation: it is characterised by an absence of clear exemplification.

As we have seen, the interpretation of this discourse is also variously influenced by:

- a misunderstanding (on the part of the researchers as well as the respondents) of abstraction in computer programming;
- a simplistic transferral of gender ideas, from psychology and ethical reasoning, to computer programming;
- an uncritical acceptance of the transferral of styles from music and poetry to computer programming.

The methods adopted were therefore chosen with the intention of minimising the scope for subjective interpretation and prejudice by making concepts concrete and presenting a statistical analysis of responses.

8.1.1. The need to exemplify

The empirical research on which Turkle and Papert's claims are based, involved observation and interview of children, as well as degree students. This was conducted at Hennigan School, Harvard, and Massachussets Institute of Technology (MIT) in the 1980s, with students learning to program. It is not, however, described in detail in any of their publications (1980; 1984a; 1984b; 1988; 1990; 1992; 1996). While they cite 14 out of 20 girls favouring the 'soft' approach, as opposed to 4 out of 20 boys, it is not clear how the statistics were arrived at. In exemplifying 'hard' and 'soft', 'abstract' and 'concrete', there are few concrete programming examples. Those that are given are largely based on the use of a special-purpose language, LOGO, in a school context: they tend to be intrinsically visuo-spatial in nature, and do not reflect common real-world programming scenarios. Student programmers are interviewed, and their views form the practical support for the analysis. "Robin", who is said to be "frustrated with black-boxing or using prepackaged programs", gives typical testimony when she is quoted, talking about 'prepackaged programs':

> "I told my teaching fellow I wanted to take it all apart and he laughed at me. He said it was a waste of time, that you should just black box, that you shouldn't confuse yourself with what was going on at that low level." (1992, p.7; 1990, p.134)

There is no description, or even exemplary identification, of 'it' - the prepackaged routine she wanted to dismantle.

The basis of my research was to describe 'it': to give novice programming students practical examples of prepackaged routines, and find out who "wanted to take it all apart". In this way Turkle's hypothesis could be put to the test: if true, one would expect that women would be more inclined to look inside the black box and to take it apart.

We will look at each of the two sample groups in turn.

8.2. Liverpool Hope University College

8.2.1. The sample

As a beginners' course, the Hope programming unit did not immediately reflect real-world programming problems. In this respect, it would appear to be similar to the course referred to in Turkle and Papert's studies. The strategy adopted, both in formulating the questions, and analysing the responses, was, however, different. Specific and applied questions were formulated, gauging responses on an ordinal scale, and a χ^2 test used to compare available female and male responses.

As already indicated, the students surveyed in the first group were Combined Honours students at a liberal arts college of Higher Education in Liverpool, England. Their first year course of study consisted of Information Technology (IT) and two other subjects. The IT provision assumed no previous experience upon entry: students generally took up IT to learn more about computers, and not to become programmers. All students had completed a compulsory first year module on programming and problem-solving. They would know enough to understand practical examples, but had not encountered abstraction and black-boxing as explicit concepts. Students had no awareness either of the debate represented by Turkle and Papert, or that gender was a key part of the questionnaire. It was therefore possible to test the hypothesis with minimised contamination: students would not prejudice their responses in relation to the debate.

The sample size was 41, of whom 17 were male and 24 female.

8.2.2. The course

In previous years, the Liverpool Hope first year programming unit had used Pascal as its language, and had included more material and concepts. Some of these were dispensed with on the basis that students experienced great difficulty grasping the ideas: enumerated types and pointers were dispensed with first, then arrays of records. In the year preceding the development of the unit as run

during the research, a new case-study approach to teaching program design and development – still using Pascal – was adopted. This involved looking at much larger programs, understanding how to read the code, and how to work in groups to add to it. Common design patterns or 'templates' were identified and reapplied. The success of the approach was in part compromised by team tutors' reluctance to depart from the traditional syntax-based drill teaching with which they were familiar. Student results were still a matter of concern, so the language was changed, and yet another new unit developed.

The new unit employed a simple environment that was developed in-house at the college: PieEater provided a two-dimensional grid environment through which a small character (the eponymous pie-eater) could be directed, eating pies on request along its way. Students could see on-screen the immediate effects of their instructions. In this respect it shares some of the characteristics of the Logo environment used by some of Turkle's subjects. It does not, of course, reflect realistic or typical programming tasks, but presents a simple environment within which basic principles may be learnt, and immediate results can be seen. The implicit teaching approach would be described as 'concrete'. It is important that teaching and learning in a concrete way – by exemplification and practical manipulation – should not be confused with programming in a concrete way. The PieEater environment represented a strongly exploratory and concrete pedagogical approach to the teaching and learning of programming, yet, for ease of understanding, it made extensive use of black-boxing, requiring students to use pre-packaged routines written in Visual Basic. These routines were written by programmers other than the student, but the student is not taught how to make their own black boxes. The student would largely use these routines, with occasional direct use of routines native to Visual Basic itself. The questions to be asked would concern attitudes to the use of black boxes, not attitudes to learning by exemplification and manipulation, and not the temporary use of glass boxes for learning purposes.

8.2.3. Relationships and other contexts

The small size of the institution, and the high proportion of mature students, both led to relatively personal and informal relationships between students (including respondents) and staff (including the researcher). The questionnaire was not identified with a specific member of staff, and its concern with gender was not revealed or discussed. It was presented as part of general research into the ways students learn to program, and was distributed, in different groups by different tutors, to be completed either before or after the completion of the class. All staff were experienced programming lecturers, three male and one female. The classes were practical sessions, where student numbers were restricted to a maximum of 20. The response rate represented the attendance rate in the week of distribution.

8.2.4. The questionnaire

As noted above, the course made little or no explicit reference to the design concepts of abstraction or black boxes: it was therefore neither possible nor desirable to ask questions that referred explicitly to these concepts. The guiding principle in formulating the questions was to provide concrete exemplification of what it meant to use prepackaged routines. Preferences in relation to black boxes would be elicited by asking implicit and strongly exemplified questions. The learning environment (PieEater) made specific use of what Turkle and Papert call 'prepackaged routines', so practical examples of these were presented to students, and their reactions elicited.

Three simple questions were framed: the first asks whether the student is curious just to know the inner workings of typical PieEater routines:

> 1. In PieEater, routines were provided to enable you to *move* or *turn* the pie-eater. Have you ever wondered, or wanted to know, how the *move* and *turn* routines were implemented?

This is the basic 'glass box' question, testing the weaker end of Turkle and Papert's thesis: it entails a desire only to look inside prepackaged routines.

The second goes further, and asks if the students would prefer to actively write their own routine instead:

> 2. Would you rather have written your own routine to move and turn the pie-eater, so that you knew exactly how things work?

The third moves on to Visual Basic itself – would the student wish to write their own routine rather than use Visual Basic's pre-packaged routines:

> 3. Visual Basic allows you to create objects on the screen – such as buttons and windows – without having to write the code to do so yourself.
> Would you prefer to write your own sub-routine to draw a button?

This question tests the 'strong' end of the thesis.

The scale used was a simple three-point Likert scale, from definite rejection (1), through a willingness to consider the notion (2), to definite confirmation (3).

Personal data was also requested: age-group, sex, previous experience and qualifications. The additional data ensured that the spotlight was not on gender, and provided information concerning other potential variables.

8.2.5. Analysis of data

Higher scale points indicate respectively that the student would prefer to 1) see inside a prepackaged routine; 2) write their own routine instead of a routine prepackaged by the tutor; and 3) write their own routine instead of a routine prepackaged by Microsoft. If Turkle's thesis were accepted, there would be a marked difference between the response of female students and that of male students: females would register high scale points more frequently than males, and males would register low scale points more frequently than females.

As the responses were measured on a three-point ordinal scale, a X^2 test was used to examine how much the observed response frequencies differed from the

frequencies that would be expected given the null hypothesis of no difference between women and men.

8.2.5.1. Question 1

Observed frequencies for responses to the 'glass box' question are indicated in table 3.

Observed data O_{ij}

scale point	*1*	*2*	*3*	
Female count	2	14	8	24
Male count	1	10	6	17
	3	24	14	41

Table 3

The expected frequencies, under the hypothesis of no difference, are shown in table 4:

Expected data E_{ij}

scale point	*1*	*2*	*3*	
Female	1.756	14.049	8.195	24
Male	1.244	9.951	5.805	17
	3	24	14	41

Table 4

It may be noted from this that the number of female responses at points 2 and 3 on the scale was actually lower that would be expected, and higher for 1.

The calculation of the value of χ^2 is shown in table 5.

χ^2	*1*	*2*	*3*	$(O - E)^2/E$
Female	0.0339	0.0002	0.0046	0.0387
Male	0.0478	0.0002	0.0066	0.0546
	0.0817	0.0004	0.0112	
			Sum =	0.093

Table 5

With two degrees of freedom, and a 5% significance level, the critical value of χ^2 is 5.991. With *Calc* χ^2 0.093 < *Tab* χ^2 5.991, the null hypothesis was accepted for the glass box question, and it was concluded that there was no significant difference between the responses of women and men.

8.2.5.2. Question 2

Observed frequencies for responses to the question which asked students if they would prefer to write their own routines instead of the prepackaged routines, were as follows:

Observed data O_{ij}

scale point	1	2	3	
Female count	5	12	7	24
Male count	3	8	6	17
	8	20	13	41

Table 6

Expected frequencies, under the hypothesis of no difference, were calculated:

Expected data E_{ij}

scale point	1	2	3	
Female	4.683	11.707	7.610	24
Male	3.317	8.293	5.390	17
	8	20	13	41

Table 7

The calculation of the value of χ^2 is shown in table 8:

χ^2	1	2	3	$(O - E)^2/E$
Female	0.021	0.007	0.049	0.078
Male	0.030	0.010	0.069	0.110
	0.052	0.018	0.118	
			Sum =	0.187

Table 8

Again, *Calc* χ^2 0.187 < *Tab* χ^2 5.991, and the null hypothesis was accepted. There was no significant difference between the responses of women and men to the proposition that they write their own routine instead of a prepackaged routine.

8.2.5.3. Question 3

The observed responses to the question which asked whether students would wish to write their own routine in place of Visual Basic's own pre-packaged routines, are shown in table 9:

Observed data O_{ij}

scale point	*1*	*2*	*3*	
Female count	19	4	1	24
Male count	13	4	0	17
	32	8	1	41

Table 9

The null hypothesis expected frequencies are shown in table 10, and the value of χ^2 in table 11:

Expected data E_{ij}

scale point	*1*	*2*	*3*	
Female	18.732	4.683	0.585	24
Male	13.268	3.317	0.415	17
	32	8	1	41

Table 10

χ^2	*1*	*2*	*3*	$(O - E)^2/E$
Female	0.004	0.100	0.294	0.397
Male	0.005	0.141	0.415	0.561
	0.009	0.240	0.708	
			Sum =	0.958

Table 11

The null hypothesis is again accepted, with $Calc\ \chi^2 < Tab\ \chi^2$. There was no significant difference between the responses of women and men to this question.

8.2.6. Further Observations

It may be observed that there is a marked difference in the overall responses to questions 1 and 3 respectively: while question 1 tempted a third of respondents to peek inside a prepackaged routine, only one respondent was definitely inclined to self-author a routine prepackaged in Visual Basic. Black boxes appear to be more welcome in complex situations. It may also be noted that, apart from the fact that this single respondent to the most unpopular proposition was female, females were generally less likely to respond at the higher end of the scale than males – and therefore slightly **more** likely than males to prefer black boxes. This difference, however, is not statistically significant.

8.2.7. Conclusion: Liverpool Hope

University programming courses typically emphasise high-level understanding and transferable problem-solving skills, rather than quick-fix practical solutions specific to the individual problem, language, and programmer. The perception of principles that lead to greater reusability and maintainability, patterns in different representations, transferable problem solving strategies, techniques to facilitate team-working on large projects, are all important. Such objectives should lend themselves to – but by no means guarantee - a deep approach to teaching and learning. This approach may involve both concrete and abstract teaching styles. The PieEater environment represented a strongly concrete strategy, encouraging students to explore problems in a hands-on way, and providing concrete representation of actions. As in Kolb's learning cycle, this is the starting-point in a cycle that moves on to reflection and conceptualisation, and then links back to the experimental (Kolb, 1983).

Turkle and Papert view the concrete approach as neither a learning style nor a stage of development, but as a fully-formed "intellectual style" (92: 11). This is clearly debatable – as is the conclusion that this style is in contrast to and as good as the use of abstraction and black-boxed library routines. What is suggested by this study, however, is that women are not more likely than men to shun black boxes.

8.3. The Manchester Metropolitan University

8.3.1. The sample

The sample group consisted of first-year Combined Honours undergraduate students, all of whom were on an introductory Java course at The Manchester Metropolitan University (MMU). While other cohorts (Single Honours in Software Engineering, Computer Science, and Information Systems) were also studying on the same course, this particular group was chosen for two key reasons.

Firstly, the sample at Hope had involved Combined Honours students, and the intention was to see if the Hope results would be repeated at a different type of institution; secondly, whereas the Single Honours cohorts contained virtually no female students, there was a larger minority of female students on the Combined Honours programme. The sample therefore differed in that it was drawn from a student body that was predominantly male. This indeed was one of the reasons for sampling the undergraduate population of a British 'new' (post-1992) University in addition to the sample from the liberal arts college of higher education. The 'culture' within University departments of Computer Science – or, in this case, the Department of Computing and Mathematics – is more distinctively masculine than in the college sector. There are no female members of staff teaching programming. The few female students in the cohort are deliberately grouped together, and it is therefore not unusual for classes of 32 students to contain no female students whatsoever.

A further contrasting characteristic of the Manchester cohort is that, whereas a large number of the Hope students were mature, only two students over the age of 25 were recruited (of whom only one was retained). All female students were aged between 18 and 20.

All students were studying for a BSc Honours degree, and therefore their second principal subject would be drawn from sixteen other disciplines typically available at a new University. These subjects include Languages,

Sociology and Psychology as well as Chemistry, Biology, and Materials Science. Students were enrolled in one of two computing subjects: Computing Science, or Information Systems, with the former being generally characterised as more technical.

Although higher admission requirements were set for students accepted onto the Combined Honours degree [16 points at A level or equivalent, as opposed to 12 on the 'Modular' single honours degrees], Combined Honours students had traditionally performed less well than their single honours colleagues. This underperformance is usually more pronounced in programming courses, and courses generally regarded as being more 'technical'.

During the course, students had encountered black and glass boxing as concepts in the testing of programs, and abstraction only implicitly, in the form of stepwise refinement as a design technique, and the corresponding use of methods as an implementation technique. Students again had no awareness of gender as an issue of debate in programming, or as a significant part of the research.

The sample size was 50, of whom 39 were male and 11 female.

8.3.2. Further Background

The representation of women on the Combined Honours Computing (encompassing Computing Science and Information Systems) at the University, was only slightly above the national average. Of the seventy three applicants who were offered places, sixteen (22%) were female, and fifty seven were male. The number of males who were offered a place, and were retained until the end of the academic year, was 84%, as opposed to 94% for females.

The average mark of all Combined Honours students in the introductory programming unit was 40.7%. The average for females was 40.9%, with a slightly lower average for males of 40.6%. This was in contrast to the overall average (which also takes into account the students' other core unit – either

Systems Analysis or Computer Architecture), where male students scored an average of 42.56%, and female students an average of 41.73%.

Interestingly, more female students were enrolled for Computing Science than for the 'softer' Information Systems, so 66% of female students completed the more technical Computer Architecture unit, rather than Systems Analysis. And the average for female students on these units was higher – in defiance of conventional notions about what areas are 'female-friendly' - for Computer Architecture (49.6%) than it was for Systems Analysis (42.5%) 48:36.

8.3.3. The course

The teaching of programming to first years had been historically problematic at MMU, with high rates of failure. Up to the academic year in which the survey was conducted (2000-2001), the language used to teach programming was Modula-2. Previous strategies had included the extensive use of a Computer Based Learning package that adopted constructivist approaches to learning and facilitated relatively independent study. Results had been extremely poor, and some concerns were expressed that Modula-2 was not a professionally useful language in itself. It was therefore decided to increase student contact time, to switch language to Java, and to adopt a book that was not object-oriented, which simplified input and output procedures, and which declared it was aimed at non-technical students. The course was also changed in terms of staff-student contact time. In addition to the traditional single lecture plus practical laboratory session, students would have a weekly tutorial session and a further laboratory session.

The move to Java was primarily motivated by the perception that poor student motivation was central to poor performance, and that students would be more strongly motivated to learn a language that had such a high profile both in the media and in industry. The relatively early use of graphical routines, and the potential for web-based development, were additional motivating features of the language.

The new programming language presented two problems, however: firstly, it was perceived that Java's object-orientation would make it more difficult to learn; and secondly, implementing simple input – reading from the keyboard – is an inordinately complex task in Java. User input cannot be avoided even in writing one's earliest programs - a computer program is of no use if it cannot obtain data from outside itself - and the level of technical detail Java requires in order to implement basic reading from an input stream such as the keyboard would represent a serious distraction and obstacle at the early stages of learning to program. A broad analogy may be made with a natural language learning system whose instructions themselves require the learner to know more complex language than the system itself is intended to teach.

As well as deciding to teach Java in a procedural manner, it was therefore also decided to use a text book which provided – in the words of Turkle and Papert - prepackaged routines that would simplify input and output so that it was possible to write interactive programs from an early stage.

The way in which these two practical educational 'problems' were handled provides an interesting illustration of the complexity of the themes underlying the research question. On the one hand, it was decided to use data hiding in a practical way – by using opaque third-party routines to simplify input and output; but on the other hand, it was decided that teaching data hiding as a systemic, full-scale conceptual paradigm, would, contrarily, be too difficult.

This apparent paradox exemplifies the distinction between programming and conceptual 'abstraction': the use of the prepackaged third-party routines represents a 'concrete' application of abstraction, whereas programming in an explicitly object-oriented way requires a full conceptual understanding of abstraction. The perception of the course team was that students could not move onto this level of conceptual understanding without first experiencing and understanding concrete representation. Among a substantial section of the student population, this was also a problem in terms of stepwise refinement as a design technique. The use of stepwise refinement clearly represents the use of

abstraction. Indeed, the hiding of detail is in many ways inherent to the design process itself. To understand stepwise refinement sufficiently to make it a practical design technique, one tutor commented, would require a level of reflection and conceptualisation that beginners were unlikely to be able to accomplish without first acquiring practical experience of what it meant. Such experience would be the starting-point in a learning cycle that would eventually lead to the conceptual understanding necessary to design solutions to problems rather than just implement possible solutions.

The issues surrounding object-oriented programming in general, and Java in particular, have been dealt with in more depth in Chapter 6.

The programming course was similar to the Hope level two course in that it was as a beginners' course, and no previous programming experience was assumed. The course differed in several respects: it used Java as an introductory programming language; the students were predominantly male; and students were likely to have a higher level of entrance qualification and experience.

The general reason for choosing the course book, *Java Elements*, by Bailey and Bailey, was that its approach emphasised the elements of programming rather than the intricacies of Java. More specifically, it was chosen because it provided easier input and output routines, did not explore object-oriented programming in any detail at an early stage, and made early use of graphically oriented examples and tasks.

The authors make use of a high-level abstraction feature of Java called a *package*. The term clearly relates strongly to Turkle and Papert's concept of 'prepackaged procedures' and 'opaque containers'. A Java package is simply a named library of routines used for a particular purpose: in the words of Bailey and Bailey, "a collection of related software components" (2000, p.37). Turkle and Papert's Lisa writes her own procedures because she "resents" the "opacity" of "prepackaged ones from a program library". Java is fundamentally based on packages – every class (or 'procedure') in Java is part of a package.

Examples are the standard Java API (Applications Programming Interface), including the AWT (Abstract Windowing Toolkit), which provide the programmer with basic functionality and graphical user interface components respectively. The first lines of most Java programs consist of a statement or statements importing routines from one or more packages. An inexperienced programmer, writing only small programs, will use only third-party packages; with more experience and time, a programmer would also write their own packages, to organise a large program, to systemise and reuse building-block routines, and to customise their programming environment to their needs.

The early and extensive presentation in the book of graphics routines allowed for a 'concrete' style of learning: in general, students enjoyed calling routines that would immediately draw shapes and colours onto the screen. All of these routines were provided as part of the Elements package – although they were often not very different from the underlying AWT routines, setting up the drawing window was easier. This once again exemplifies a concrete learning environment made possible by high levels of abstraction.

8.3.4. The questionnaire

As noted above, the course made little or no explicit reference to the design concepts of abstraction or black boxes: it was therefore neither possible nor desirable to ask questions that referred explicitly to these concepts. The guiding principle in formulating the questions was to provide concrete exemplification of what it meant to use prepackaged routines. Preferences in relation to black boxes would be elicited by asking implicit and strongly exemplified questions. The customised learning environment (PieEater) made specific use of what Turkle and Papert call 'prepackaged routines', so practical examples of these were presented to students, and their reactions elicited.

Four simple questions were constructed: the first seeks to ascertain the extent to which students would like to look inside the black box of the package provided by the authors of the course book:

1. If time allowed, would you like to inspect and take apart the code that Bailey & Bailey wrote in implementing the element package, so that you could find the detail of how methods such as readLine were written?

This question was designed to test the weaker end of Turkle and Papert's thesis: it entails a desire only to look within a prepackaged routine, an inclination to "bother with its detail". The proviso 'if time allowed', was intended to offset simplistic negative responses based on the extra effort that was implied.

The second question asks if the students would prefer to actively write their own routine instead of using the one supplied in the *elements* package:

2. Would you prefer to write and use your own routines rather than using either readLine or another prepackaged method?

The third question refers to use of a somewhat lower level of prepackaging – the packages that the authors used in writing their own packages:

3. Would you prefer to use the Java AWT and IO packages directly yourself?

The packages included in the course book are particular to that book, and are designed to ease the learning curve for students. The AWT and IO packages, on the other hand, are what programmers 'in the real world' would use. For this reason – although it represents a lower level, and a level of greater perceived difficulty, it might in fact appear attractive to students to use the routines from these packages directly.

The fourth question tests the stronger end of the hypothesis, by asking whether the student would like, not just to use the AWT and IO routines, but actually to take them apart to see how they work:

4. Would you like to take apart the Java AWT and IO packages to find out how the underlying code has been implemented

The scale used was again a simple three-point Likert scale, from definite rejection (1), through a willingness to consider the notion (2), to definite confirmation (3).

Personal data was again requested for the purposes of correlating the potential variables of age-group, sex, previous experience and qualifications.

8.3.5. Administration of the Questionnaire

The issue of distributing copies of the questionnaire was discussed with the three tutors responsible for teaching the Combined Honours students, and with the Combined Honours course leader. On the basis that a small number of students were so weak that they might not understand what was meant by some of the questions, it was agreed that it would be useful if the tutors were to briefly recap on what packages were, and how the students had encountered them during their study, before they started to fill in the questionnaire.

The distribution of questionnaires took place at the beginning of the weekly laboratory session. Within the academic year, this took place at a point approximately three quarters of the way through: when there was no immediate pressure to complete an assignment or revise for an examination, but at a time when the students had studied sufficiently to understand what was being raised in the questions.

8.3.6. Analysis of data

Higher scale points indicate respectively that the student would prefer to 1) see inside a prepackaged routine; 2) write their own routine instead of a routine prepackaged by the authors of the course book; 3) use routines from standard Java packages for themselves 4) see inside the routines from the standard Java packages. If Turkle's thesis concerning attitudes to black boxes were accepted, there would be a marked difference between the response of female students

and that of male students: females would register high scale points more frequently than males, and males would register low scale points more frequently than females.

The responses were measured on a three-point ordinal scale, and a X^2 test used to examine how much the observed response frequencies differed from the frequencies that would be expected given the null hypothesis of no difference between women and men.

8.3.6.1. Question 1

Observed frequencies for responses to the weak 'glass box' question, asking whether the student would like to inspect and take apart the package provided by the course book, are indicated in table 12.

'See Elements Code'

Observed Data Oij

scale point	1	2	3	Total
Male	3	27	9	39
Female	4	7	0	11
	7	34	9	50

table 12

The data shows a strong distribution around the middle scale point. The expected frequencies, under the hypothesis of no difference, are shown in table 13:

'See Elements Code'

Expected data E_{ij}

scale point	1	2	3	
Male	5.460	26.520	7.020	39
Female	1.540	7.480	1.980	11
	7	34	9	50

table 13

It may be noted from this that the number of female responses at the highest scale point was zero, whereas the number of male responses was higher than would be expected at the top point (nine, rather than seven), and at the middle point. The inverse was true in terms of the lowest scale point: the number of male responses at this scale point was lower than would be expected (three observed, and 5.46 expected), and the number of female responses was higher (four observed, 1.54 expected). points 1 and 3 on the scale was higher that would be expected, but lower for the middle point 2. This tends to support – more strongly than the Hope data – the reverse of Turkle and Papert's hypothesis.

The calculation of the value of χ^2 is shown in table 14.

'See Elements Code'

Chi-Squared	1	2	3	
Male	1.108	0.009	0.558	1.676
Female	3.930	0.031	1.980	5.940
	1.108	0.009	0.558	1.676

***table* 14**

With two degrees of freedom from three scale-points, and a 5% significance level, the critical value of χ^2 is again 5.991. With *Calc* χ^2 1.676 < *Tab* χ^2 5.991, the null hypothesis was accepted for the 'Elements' glass box question, and it was therefore concluded that this difference between the responses of women and men, although stronger than the difference elicited by the Hope study, was still not statistically significant.

8.3.6.2. Question 2

Observed frequencies for responses to the question which asked students if they would prefer to write their own routines instead of the prepackaged methods, are shown in table 15:

2. 'Own Routine'

Observed Data Oij

scale point	1	2	3	Total
Male	19	18	2	39
Female	4	5	2	11
	23	23	4	50

table 15

The major difference that may be observed between these responses and those to question 1, lies in the much larger figure for responses at the lowest scale point – which is as strong as that around the middle scale point. It is clear that most students would not prefer to write their own routines. The expected frequencies, under the hypothesis of no difference between female and male respondents, are shown in table 16:

2. 'Own Routine'

Expected data E_{ij}

scale point	1	2	3	
Male	17.940	17.940	3.120	39
Female	5.060	5.060	0.880	11
	23	23	4	50

table 16

On this occasion, the proportion of female responses among the few at the highest scale point, is actually higher than would be expected. And the number of male responses at the lowest point was higher than expected, the number of female responses lower. This does not therefore reproduce the results of question 1, and instead tends to fit with Turkle and Papert's hypothesis. The numbers involved (two students from each gender) are small.

The calculation of the value of χ^2 is shown below:

scale point	1	2	3	
Male	0.063	0.000	0.402	0.464883
Female	0.222	0.001	1.425	1.648221
	0.285	0.001	1.828	2.113

table 17

Again, *Calc* χ^2 2.113 < *Tab* χ^2 5.991, and the null hypothesis was accepted. There was therefore no significant difference between the responses of women and men to the proposition that they write their own routine instead of a prepackaged routine.

8.3.6.3. Question 3

The observed responses to the question which asked whether students would prefer to use standard Java packages directly, are shown in table 18:

3. *'Use AWT'*

Observed Data Oij

scale point	1	2	3	Total
Male	7	27	5	39
Female	3	7	1	11
	10	34	6	50

table 18

As with question 1, the data show a fairly strong bunching around the centre point. The null hypothesis expected frequencies are shown in table 19:

3. *'Use AWT'*

Expected data E_{ij}

scale point	1	2	3

Male	7.800	26.520	4.680	39
Female	2.200	7.480	1.320	11
	10	34	6	50

table 19

This data shows – as with question 1 - a larger number of males than expected at the top and middle points, and a smaller number of females. The number of females at the bottom end was correspondingly higher than expected. This is, as with question 1, opposed to Turkle and Papert's hypothesis.

The calculation of the value of χ^2 is shown in table 20:

scale point	*1*	*2*	*3*	
Male	0.082	0.009	0.022	0.112619
Female	0.291	0.031	0.078	0.399287
	0.373	0.039	0.099	0.512

table 20

The null hypothesis is again accepted, with *Calc* χ^2 0.512 < *Tab* χ^2 5.991. There was no significant difference between the responses of women and men to this question.

8.3.6.4. Question 4

The observed responses to the 'strong' glass box question which asked whether respondents would like to take apart the standard Java packages, are shown in table 21:

4. *'see AWT'*

Observed Data Oij

scale point	*1*	*2*	*3*	*Total*
Male	5	27	7	39
Female	0	8	3	11
	5	*35*	*10*	*50*

table 21

The data here again show a very strong bunching around the centre, and the highest overall figure at the top point. This is perhaps puzzling, given that it is probably the most challenging proposition of the four. The expected responses for this question are given in table 22:

4. '### AWT'

Expected data E_{ij}

scale point	1	2	3	
Male	3.900	27.300	7.800	39
Female	1.100	7.700	2.200	11
	5	35	10	50

table 22

The number of females at the high point was higher than would be expected, and the number at the bottom point was lower. As with question 2, this tends to fit with Turkle and Papert's hypothesis.

The calculation of the value of χ^2 is shown in table 23:

scale point	1	2	3	
Male	0.310	0.003	0.082	0.395604
Female	1.100	0.012	0.291	1.402597
	1.410	0.015	0.373	1.798

table 23

The null hypothesis is again accepted, with *Calc* χ^2 1.798 < *Tab* χ^2 5.991. There was no significant difference between the responses of women and men to this question.

8.4. Interviews

Computer and departmental cultures undoubtedly contribute to the way people construct meaning. Interviews would serve to explore these and other situational contexts of the target population in greater detail, and to triangulate the data. However, as has been noted, the anonymity of the questionnaire compromised the scope for this. In the second study, no female students volunteered for interview.

It was therefore decided to conduct limited interviews, of several women at different stages of their computing studies – ranging from Masters level, to the second year of the undergraduate Combined Honours degree course at MMU. The intention was to elicit, from informal interview situations, how they felt as women about studying computing, and about programming.

8.4.1. Tania

Tania entered MMU's MSc conversion course at the age of 25, having already obtained an MSc previously, in Forensic Psychology. She noted that, whereas most students on the Computing MSc were male, there had been only two male students out of thirty on her Psychology course. Tania explained her reasons for undertaking a Masters course in computing in terms of her previous job as a Psychologist in a prison. This job had been a dangerous one, involving the potential of unprovoked abuse. One reason why computers seemed attractive was that it avoided dealing with unpredictable people, and involved no danger: "You are not dealing with any unpredictable people, you're just sitting in front of a computer, and the worst that could happen is that the system could crash." It is perhaps interesting that this reflection echoes the psychology often negatively ascribed to male hackers.

While she was not overly concerned that there were relatively few women on the course, Tania gave clear indications of a departmental culture that was male-dominated and inimical to women. At the outset of the course, she had asked if doing the Masters would prepare her for a job in programming. She had been told that "not many women actually go into programming because it is so male-dominated, plus they've not really got the capability to go into it". She

indicated that this put her off straight away – but that it had also made her more motivated to understand it (a phenomenon she described as 'psychological reactance'). She felt that it was true that programming was dominated by men – and that this was to some extent self-fulfilling. A further possible reason was "probably because it is quite boring, isn't it?". She felt that programming involved no social contact, and that the women she knew preferred jobs that involved contact with a lot of people

Students had problems with the diversity of programming languages on the course, and in particular found the Java module either difficult or problematic. Although Tania thought that this experience of difficulty was equal between female and male students, on further elaboration the "problems" appeared to be differentiated. While "the girls in general used to find that it was quite complicated", the difficulty identified with the boys was more often complaint-oriented. Some would test the tutor with questions they already knew the answer to. She felt that all but one of the men who appeared to 'know it all', genuinely did – and that the women were more likely to have genuine difficulties with understanding the material. The difference between a "swot" and a "nerd" was that nerds don't have social life.

Tania did not view the computer as a psychological machine, "because it can't talk back. You're controlling it at the end of the day. All it can do is just respond to your actions, which you've got control of...you're controlling that computer, whereas with a person you can't". Having dealt with inmates who "can answer you back even though you might not say anything to them", she sees "this PC or any computer as something that's just, I don't know, a whiteboard or something"

Tania's initial response to the issue of prepackaged procedures was that she "would prefer to see the overall picture", including the "source code". It emerged, however, that she conceived of this as simply seeing any code that was being talked about, rather than listening to it being discussed "on the

abstract level". When asked what seeing the code of writeLn would tell her, she responded: "Well, the syntax, how you write it: write 'L' – 'n', 'brackets'". She did not in fact think that she wanted to see any code at a lower level, and would prefer to draw a button in Visual Basic rather than write the code for it herself.

Tania had some experience with Multimedia Authoring software, and appreciated the extent to which it automated programming: "writing the code out myself would be a lot more difficult than already having it there and just adding bits in".

It was notable, during the interview with Tania, how readily the general and pedagogical meaning of the 'abstract' can be confused with its significance in programming. This observation had implications in terms of the clarity of the questions in the questionnaire. On revisiting these, it was clear that the questions were more specific and precise, as to what was meant by 'seeing the code'. Within the framework of the questionnaire format, there did not appear to be much else that could be done, and the margin of error did not appear to be any greater than would be normal with any series of questions. It may, however, be desirable in future to construct an extended programming situation that was intrinsic to the course, and to ask questions within that situation.

8.4.2. Maria

Maria was nearing the end of a Combined Honours degree at Liverpool Hope University College. She found pre-packaged routines "helpful". However, she did not in practice spend much time on planning and design. She felt that this was because the problems set were relatively small, and that anything bigger, covering, for instance, several screens in Visual Basic, would need planning. In her Applications Programming module, for example, she had been set the task of developing a temperature converter, and had gone straight onto the PC without doing anything on paper, apart from the calculations. Much of the work set, she indicated, was "about knowing the right commands or messages", and placing objects on a screen. Maria perceived screen design, in Access and

Visual Basic, to be entangled with data content, so she did this directly on the screen.

In Maria's experience we see the impact of teaching methods. If the problems that students are set are small and mechanical, they see little point in thinking about design and stepwise refinement (and, ultimately, in abstraction). The goal is to find out how a small set of commands can be implemented so that they work. It also appears that higher-end interface-driven tools may lend themselves more to a hacking approach.

8.4.3. Son-Yong

Son-Yong was in the second year of her BSc Combined Honours at MMU. She chose to do Information Systems and Psychology, as she didn't want to do "anything technical" – and she certainly didn't want to do Computing on its own. She perceived Information Systems as being potentially "useful" in the employment market.

She felt that computing was male-dominated, and that this reflected existing ideas about what people were good at: "this thing about men being 'more technical', and women better at literature etcetera". Like Tania, Son-Yong thought computing was "boring", (and also "scary"), and found the gender balance on the course to be in contrast to the psychology part of her degree (where males were in a small minority). The boys who were good at programming were also boring and scary. They were more adept because they grew up playing computer games, and did not find it boring. The people on the course who had prior experience were male – they 'know their stuff': she wished she could be their friend! She said that "writing bits of code in an exam was particularly difficult and challenging - that in effect it required prior experience, and therefore favoured the boys.

Son-Yong indicated that, while many women found programming "quite hard", she felt that she personally was "in the middle" – perhaps the opposite of the 'we can, I can't' phenomenon? She would quite like to be a programmer herself – but only if she was sure she was "quite good at it". She had done some work

experience that involved PHP programming, where she had worked in a team. Two out of 8 or 9 members of the team (including herself) were female; the other was a senior member of marketing staff. Although problems were occasionally tackled as a team, she felt that the work was 'not very sociable'.

In terms of the prepackaged procedures provided in the elements package, she "wouldn't mind seeing how they work" – although if it was 'hard', it could possibly confuse. She would not like to write it herself, though. That, she said, would be "quite scary".

Son-Yong also had experience of multimedia authoring software, and when asked to choose between a prepackaged drag and drop behaviour from the Library palette, and scripting her own code, she expressed the wish to go with whatever was easier.

Son-Yong did not consider the computer to be in any way a 'psychological machine'. It was 'just a machine', and if she ever talked to it, it was as she would to any recalcitrant object. She was interested in AI, but was disappointed that it appeared to be taught as a technical rather than psychological subject at the University.

8.4.4. Rupa

Rupa was completing a Masters degree in Computing at Manchester Metropolitan University. Her studies were part-time, as she worked full-time in a programming job. As such, she had considerable experience of using abstraction in the form of library routines. When presented with the idea – new to her – that there might be a difference in the attitudes of women and men to black boxes and abstraction, she indicated that she did not feel that there was a great difference. To the extent that there might be one, she thought that it might be correct "to assume that there would a greater number of men who would be interested the actual working of black boxes and abstractions, than women". In terms of gender and computing generally, Rupa felt that "men in general have a greater interest in the background working of equipment and such likes", and

added that " this does include how library routines are constructed and their working".

Rupa explained that "one of the reasons that library routines have been used is to ensure that within the different sections.... that make up an applications certain functions work in exactly the same way." For most library routines she had used, she had "just accepted their performance at face value", and there was in any case no time to do so. She had only looking into them when "the libraries did not perform in the exact manner I expected them to". Even on these occasions, however, she had not amended the libraries, "as this would inflict unnecessary problems onto other users' programs", but had "copied and then amended the necessary areas of the library and then incorporated them into my own program".

Rupa's experience reflected the reality of programming for clients in the real world: her work was essentially defined by the need for egoless programming, and team use of library routines. Her expectation was that it would be men rather than women who would waste time tampering with black boxes rather than simply using them.

8.4.5. Zaida

One of Turkle and Papert's interviewees, Robin, spoke of her experience as a piano player who came to programming as a second year student at Harvard. Zaida, a second year single-honours Computing student at Manchester Metropolitan University was also a highly trained and accomplished pianist. Her course content included modules on advanced Java programming, and on multimedia scripting. She talked with enthusiasm and clarity about areas of similarity between playing the piano and programming. She saw analogies between playing piano and the Java she was currently studying – with its emphasis on server-client protocols, getting the client and the server to talk to each other. Playing the piano is also a communication process. One also has to be insistent and patient to get the best out of both a piano and a computer.

In both playing and programming, she said, "it was necessary to plan, to think about it. You don't just sit there and play. It's the same with programming – you can't just sit down at the computer and write a program". She also saw both programming and playing the piano as "a kind of language". She explained that

> music has its syntax, just like a programming language – there is a grammar of music, and it is really important. It is essential that you understand this grammar first. Playing the piano is creative, but you must have the syntax first. To be able to play a piece, first you need to think a lot: a lot of logic, and maths, is involved.

Music has its own symmetry and insistent mathematics – it is only correct if there is, for example, strictly three beats to the bar; a crochet is only ever worth a crotchet. One also has to work out how and why the notes are distributed, and calculate the way to play the notes. Piano playing is also "like a program in that you execute a series of instructions", but some of these instructions existed at an interpretative level – an expressive overlay on top of the mathematical musical notation. This layer is conveyed by means of Italian prose messages (*andante*, *adagio*, *allegro*) rather than via notation. While the notation might be equivalent to program code instructions, she saw no meaningful equivalent in programming to the expressive layer.

Zaida experienced piano playing as a more exacting activity than programming:

> with programming, if your style is not strong, you can do it anyway. You can write it, and it may work. With the piano, it's not like that. You need to be really, really careful. For example, to play Bach, you need a different way of playing – you need to insist more with your fingers. For Mozart, you need to play 'andante'; Chopin has his own style. For a Bach fugue, there are several voices, and you have to distinguish between them.

Piano players are taught that they 'master' the piano: the piano is not attributed with any of its own features or agency. It is the notation that enables you to make the piano play music. When you play Chopin's notation, people say you're playing Chopin. The notation is the musical composition. One could still feel so relaxed with a piano that it became one's best friend: playing the piano is more enjoyable than programming, which was more rigid. But one could also feel frustrated with both: she had once done a performance, and been frustrated

that an easy piece did not come off the way it should have. Such feelings, however, did not endow the piano with any character or psychological presence. Unlike Turkle's Robin, Zaida concluded that there is no relationship in the case of either the computer or the piano.

Simultaneity is a major difficulty in piano playing: one had to follow instructions without seeming to follow them (so that the sound is expressive rather than mechanical), conveying both notation and interpretation at the same time, and handling a simultaneity of voices, keys, and scores. Although there was no relationship with either instrument, playing the piano was, unlike programming a computer, an art. In this respect, Zaida concluded that piano 'style' is **not** a good analogy for programming 'style'.

Zaida also has considerable language skills, speaking Italian, English and French fluently. She thought that she learned languages quickly because they were spoken and communicative, whereas programming languages were about a way of thinking.

When asked explicitly about gender issues and programming, Zaida said that she felt that men were stronger in technical skills, but did not know why. She speculated that it might be because no emotions were involved, and men were not as emotional, less likely to panic. She also thought that perhaps women generally made programming more complicated than it was.

As well as drawing out comparisons between musical notation and programming instructions or software, Zaida represents a fascinating counterpoint to Turkle's Robin. Where Robin communicated "**with** her instrument" [my emphasis], Zaida communicated **through** her instrument; where Robin felt emotional involvement **with** the piano, Zaida felt emotional while playing it. Zaida spoke of communication, interpretation, and style, but did not recognise the "language and models for close relationships **with** music machines" [my emphasis] that Turkle and Papert (1992, p. 28) boldly ascribe to the whole culture of music. It is not the machine or tool that is important. It may

be that the "close, sensuous, and relational" encounters with inanimate objects that Turkle and Papert ascribe to artists and musicians, and transfer to their model of "highly personal" feminine programming styles, represents an equally artificial imposition of object relations theory.

8.5. Conclusion

Both of the surveys failed to show any significant difference between female and male attitudes to prepackaged routines. In both samples, responses to the basic 'glass box' question showed that women were actually slightly less likely to want to see inside black boxes. The results for the other questions varied: MMU women were slightly more likely to want to write their own routines, while Hope women were slightly less likely; and Hope women were more likely to want to write lower-level routines, while MMU women were less likely. This cross-referencing suggests that there is no pattern in, and no consistency to, these marginal differences. In the extra Java question, testing the strongest end of the hypothesis, women were less likely to want to write their own routine.

Educationally, the concrete learning style in this case was a means rather than an end, and did not imply the teaching of a concrete approach to programming. What characterised the 'concrete' pedagogical approach of the PieEater and Elements graphical activities, was that the learner's activity produced immediate visual results. It is quite clear that such an approach could exist without reference to any desire to get inside black boxes. The activities provided by PieEater represented the starting-point in a learning cycle that would move on to reflection and conceptualisation, and then link back to the experimental (Kolb, 1983).

Learning style is therefore entirely distinct from programming style. Yet while a concrete style of learning does not require a 'concrete' programming style, any given programming style (including the concrete style) could potentially be used within a concrete pedagogical approach. For Turkle and Papert, however, the 'concrete' style of programming is not a manifestation of a learning style,

and it is emphatically not a stage of development: it is, rather, a fully-formed way of knowing how to program, an "intellectual style" (1992, p.11), at least as good as the use of abstraction. The responses suggest that there is no significant difference between women and men in their attitude toward a concrete style of programming.

9. Summary and Conclusion

What has been done

The research originates from two main ideas: that women are under-represented in computer programming; and that certain programming techniques have been characterised as inimical to women. The idea that women would be more likely than men to shun black boxes in programming, has been premised in the literature on psychoanalytic and gender analyses that accentuate difference. These analyses have been first traced back to their roots and critically examined at that point. Their influence on gender analyses of computing - and the bearing of gender theory in general on computing - has then been analysed. It was concluded that essentialist strains of gender analysis value (and often preserve) difference whether it is natural or constructed, and that the theoretical base for change requires an alternative perspective. Computing can change, but not by appealing to stereotypes simply because they are in place, and which are in fact part of the problem. The under-representation of women in computing cannot be laid at the door of abstraction in programming: indeed, abstraction can play a part in humanising the larger computing culture.

A critique of the limitations of psychoanalytic theory therefore informs the sustained focus on programming, as the activity and body of knowledge that is most definitive of the subject where women are under-represented. Such a focus is not a deterministic acceptance of a predefined agenda – it reflects the fact that programming represents the knowledge and skills whereby people make computers useful and give them purpose. Insofar as computers are important, programming is important: in the post-industrial world it is increasingly programmers who are remaking the world. Within this part of the study, these underlying limitations of psychoanalytic gender theory are seen to be accentuated by an over-liberal transference of terminology between disciplines. Part of this analysis has been to demonstrate the meaning and nature of relatively specialised programming terminology that is used over-generally in the literature.

The characterisation of abstraction as a male preoccupation has therefore been challenged in relation to its terms of reference; its faulty use of analogy from other disciplines; and in relation to the essentialist tendencies that appear to underlie such a characterisation. It has been argued that the basis of the idea is superficial and contradictory in significant respects; and that the gendered characterisations could just as easily be reversed. It has been precisely those features that have been caricatured as male and remote, which enable programmers to remove mechanical detail and understand and solve problems holistically, in terms of real-world and user-centred applications rather than in terms of computer operations. The same features that make for communicative and connected styles of software development, where programmers connect, not with the machine, but with the users of their product and the other members of the software development team.

This analysis has had to be painstaking, and the research has therefore placed greatest emphasis upon it. The analysis suggested that the major assumptions within gender and programming scholarship and discourse were mistaken. It was therefore decided to test the idea in order to establish whether this critical analysis could be borne out by survey and interview research among programming undergraduates. This part of the research required careful construction of concrete means of testing attitudes to black boxes and abstraction. Again, this was informed both by the review of the literature on gender and programming, and by the exemplification of what these programming concepts meant in practice. The conduct of this part of the research was constrained by practical logistics: one needed to know the particular programming course content and delivery in detail in order to develop specific concrete examples of the general concepts under scrutiny. While the choice of student bodies was a positive feature - embracing, for example, two different types of institutions and courses - it would have been desirable to have had a larger sample, and to have been able to interview a range of respondents specifically about their responses. The sample was representative of the Combined Honours first year students in the respective institutions. A larger sample size may have increased confidence in projecting

the findings onto a wider population, and significance is confined to the particular student bodies. The number of possible outcomes is also constrained by the three-point scale, and chance influences cannot be definitively excluded as an explanation of the results. The questions were designed to maximise clarity of attitudes and minimise the influence of researcher, but to some extent such limitations are intrinsic to attitudinal research.

As a feature of a teaching and learning strategy, both the Visual Basic PieEater environment and the Java Elements graphics package, facilitated a strongly concrete learning style, encouraging students to explore problems in a hands-on and sometimes experimental way, and providing concrete representation of actions. In order to produce such a concrete learning environment, very high-level routines were implemented and packaged that were simple to use: in other words, very deliberate use was made of abstraction and black-boxing. Crucially, there was little desire to look inside or take apart these "prepackaged programs", and – contrary to Turkle and Papert's central thesis - no greater desire to do so on the part of women.

A gendered characterisation of attitudes to black boxes was therefore not borne out by the applied research. A series of specially-targeted interviews - of female students who, unlike the survey respondents, were informed of the gender basis of the research, and had a very good knowledge of programming - confirmed these findings, and shed further light on gender perceptions of programming and abstraction.

What is to be done

The nature of programming may change, and may be changing; but our research concludes that change centres on the very modular building-blocks which theorists such as Turkle reject as inimical to women, and that a serious reconsideration of the received gender analysis of programming is therefore of crucial importance.

This in itself should be positive in terms of gender equality in programming- it is only the prevalence of the antithetical theory that would cast this finding as negative. We have sought to establish that the propagating of a definitively feminine programming style as opposite to the use of abstraction, planning, and structure, is not only theoretically and empirically without foundation, but also that it can be actively damaging to the position of women in computing. For change to happen, gender has to be explicitly addressed at every level of programming education, from marketing to graduation - and when this happens it is vital that what is said is authentic. To base changes on gender characteristics and stereotypes – however prevalent those stereotypes might be – offers at best only superficial change, and at worst exacerbation of inequality. Solutions that are based on supposedly essential differences ultimately do not necessarily intend to change inequality.

Alternative perspectives suggest themselves to make programming attractive to everyone, not just to women: a need to break decisively away from the machine - and from the spectre of the hacker that so paradoxically shadows the supposed feminine style. Distance from the computer can be reframed as a good thing, something that is fundamentally social, and as such not inimical to either women or men. Abstraction has the capacity to relate programming and the programmer to the real world instead of to the machine and to 'computational objects'; to overlay obsessive detail with the 'big picture'; and to conjoin software construction to software use. As a concept and tool, it has the potential to shift programming decisively from its perceived position on the male-obsessive spectrum, and move it towards people and the real world.

In order to fulfil this potential, abstraction needs to be recognised and presented as such – to be situated as a tool, within computing education, in meaningful contexts. Some means of doing this – such as the use of case studies within a holistic pedagogy - are traditionally linked to women, but in any case make pedagogical sense in this context. The key, however, to the holistic pedagogy is understanding that the 'whole' context does not reside inside the machine,

inside the machine's programmer, or within the 'relationship' between the machine and its programmer.

A truly holistic educational case study would start at the level of useful real-world functionality. As such it would have to be represented initially at a high level of abstraction, possibly with substantial parts of the lower levels already implemented. It is thereby possible to avoid mathematics – something that is generally desirable in that it liberates software development from the machine, and from a relatively specialised discipline. Teaching methods can start from the concrete – concrete situations, concrete illustrations of 'abstraction', and concrete experiences of its effects – in order to develop sound conceptual understanding. It is possible thereby to situate programming culturally as an artistic science or a scientific art that has to be systematic and communicative rather than individual.

The roots of inequality clearly go deeper than a single sub-discipline is capable of reaching. Macro-social, political, and larger cultural explanations for inequality cannot be denied: a focus on the particularity of a discipline can be myopic if these factors are ignored. In the more immediate outer context of the academic profession, for example, a shift away from an open-ended work culture could be an important trigger for greater gender equality in staffing, particularly so for a subject that requires such constant updating. Traditional divisions of labour have informed even feminist thought, and ambivalence and contradiction attend upon gender theory. Yet it is still possible for an important discipline to contribute to deeper social change, rethink its own discourse, and change itself.

APPENDIX A

Published article: McKenna, P. (2000). Transparent and opaque boxes: do women and men have different computer programming psychologies and styles? *Computers & Education 35*, 37-49.

PERGAMON

Computers & Education 35 (2000) 37–49

COMPUTERS & EDUCATION

www.elsevier.com/locate/compedu

Transparent and opaque boxes: do women and men have different computer programming psychologies and styles?

Peter McKenna

Liverpool Hope University College, Hope Park, Liverpool L16 8ND, UK

Received 28 September 1999; accepted 4 February 2000

Abstract

An orthodox 'hard mastery' programming style is a cornerstone of Sherry Turkle's influential psychoanalysis of different approaches to learning and practice in computer programming. Hard mastery consists of planning and design, documentation, structure, functional and data abstraction, and debugging, in the development of programs. Turkle is concerned that teachers of programming are trained to recognise hard mastery as the only real way to program, whereas it is only 'male mastery'. To bring women into computing, teachers are told to teach or facilitate the development of soft, hacking styles. This paper argues that this was a misconceived and impossible aspiration whose widespread influence has led, instead, to a deepening of perceptions of programming and computing as a masculine culture, and to the implicit and absurd identification of women as innately unsuited to the skills required for large programming projects in real organisations.

Keywords: Programming and programming languages; Gender studies; Teaching/learning strategies

1. Introduction

The only original attempt to psychoanalyse different styles of programming has been that of Sherry Turkle who, along with Seymour Papert, has developed the notion of 'hard' and 'soft' styles of computer programming. Gender values are attached to these different styles, on the basis of analogies with key concepts from psychology and psychoanalysis, in combination with

E-mail address: pmckenna@liv.ac.uk (P. McKenna).

0360-1315/00/$ - see front matter
PII: S0360-1315(00)00017-8

observations of children and interviews with programming students at Massachusetts Institute of Technology (MIT).

The influence that this gendered dichotomy of hard and soft has had is reflected in its frequent reiteration: commentary on gender and computing regularly emphasises, on the basis of Turkle and Papert's work, that a structured approach to programming alienates women (Sutherland & Hoyles, 1988; Kvande & Rasmussen, 1989; Frenkel, 1990; Grundy, 1996; Stepulevage & Plumeridge, 1998). This has inevitably given rise to explicit demands that structured programming be dropped from the computing curriculum in order to encourage more women to study computing (Peltu, 1993).

Turkle transfers theoretical concepts from psychoanalysis to programming (and learning) as if such a transfer were entirely unproblematic. Just as cognitive psychology has not necessarily transferred well into computer based learning in the form of instructional design (Laurillard, 1993, pp. 74–75), so the semiotic imposition of psychoanalytical theory rests uneasily on top of the intricacies and subtleties of programming discourse and practice.

2. Programming psychoanalysed

2.1. Hard and soft mastery

In her first publication about programming, Turkle describes two different programming styles. The first is exemplified by 'programmer X' — a man who equates his style with 'a kind of wizardry', and whose style is explicitly identified with the 'hacker' subculture (Turkle, 1980, p. 20). The second is exemplified by 'programmer Y' — again, a man, but one who likes to work on precisely defined and specified projects; he also 'enjoys documentation'.

By 1984, Turkle has identified Programmer X's hacker approach as a 'soft mastery' that is a communicative and feminine 'mastery of the artist' (Turkle, 1984b, p. 49). Programmer Y's 'hard mastery' conversely became 'masculine' — 'the mastery of the engineer'. Turkle complains that teachers of programming are 'trained to recognise hard mastery as 'real' mastery', and teach it as 'the right way to do things', whereas it is in fact only 'male mastery'. This, it is claimed, amounts to 'discrimination in the computer culture' (Turkle & Papert, 1992, p. 8). To bring women into this 'computer culture', she states, teachers must recognise that this canonical orthodoxy is inimical to the psychological make-up and preferred learning styles of women, and instead 'encourage students to develop' soft mastery (Turkle, 1984b, p. 49).

In apparent contradiction to the identification of Programmer X with a female-friendly style, Turkle in the same year describes the hacker's world as a 'male world ... peculiarly unfriendly to women' (Turkle, 1984a, p. 216). By 1990, Turkle and Papert reformulate soft and hard mastery as 'concrete' and 'abstract' styles of programming respectively. They now explicitly acknowledge the similarity between the supposedly feminine soft/concrete style and the 'culture of programming virtuosos, the hacker culture', but valorise hacking as 'countercultural' (Turkle & Papert, 1990, p. 141; 1992, p. 16). The masculine characteristics of the hacker 'counterculture' are therefore set aside, and it is structured programming that is labelled a 'Western male gender norm'. Hard masters treat 'computational objects' as 'abstraction', soft masters treat them 'as dabs of paint'. It is observed at an early stage that 'girls tend to be soft

masters, while the hard masters are overwhelmingly male' (Turkle, 1984b, p. 49). While both types are 'masters', the soft-master female is characterised by sensitivity and intuitive artistry: she will 'try this, wait for a response, try something else, let the overall shape emerge from an interaction with the medium' (Turkle, 1984b, p. 49). She will try things out at the keyboard instead of planning them first on paper. Indeed, she will not like any sort of documentation. This specific sense of 'hacking' is part of a culture that, as Turkle herself illustrates in *The Second Self*, is intensely masculine. The hacker — as Mahony and Van Toen, citing *The Second Self* as support, argue (Mahony & Van Toen, 1990, p. 326) — is the epitome of the competitive techie machismo that is indulged in mainstream computing culture. A masculine obsession with inanimate objects appears to be simply feminised as connected and fusional rather than disconnected and weird.

2.2. Style and realities: examples

One of Turkle's soft masters, Anne, is 'an expert at writing programs to produce visual effects of appearance and disappearance' (Turkle, 1984b, p. 30). In order to make birds of various colours disappear, she rejects the 'algebraic' method of assigning the colour of each bird to a different variable. (It is incongruous that Turkle's examples of 'structured' solutions use elementary rather than structured components: a structured programmer would not think of a multitude of individual variables, but of a single data structure such as an array of records; and that the variables would be designated by the letters Turkle suggests rather than by meaningful identifiers). Instead, Anne masks each bird with a sky-coloured sprite, makes each sprite travel with its assigned bird, and makes the sprite appear to render the bird invisible, disappear to render it visible. This, Turkle indicates, enables Anne to 'feel' and relate to the screen objects rather than use 'distant and untouchable things that need designation by variables': 'she is up there among her birds'. Each sprite is 'not to be commanded as an object apart from herself'. How the mask sprites are to be rendered visible and invisible without 'designation' and 'command' is not explained — nor how it differs significantly from showing and hiding the birds directly. This solution does not reflect reality, or treat birds as birds: birds 'disappear' and appear on the horizon, rather than fly around with shields; they are also liable to pass or land on objects that are not sky-coloured, or to fly across skies of irregular appearance. Anne's feeling that she is in the company of the birds may be a subjective feeling unconnected either to feminine 'connectedness' or to the fact that she makes mask sprites disappear rather than bird sprites.

3. The psychological machine

3.1. The psychological machine and the real world

Turkle's categorisation of the computer as 'a psychological being' (Turkle, 1984a, p. 54; 1984b, p. 50), or 'psychological machine' (Turkle 1988, p. 50) with an 'intellectual personality' (Turkle & Papert, 1992, p. 3), produces even deeper reality problems: if the computer really is 'a psychological being', it follows that a sensitive person would not tell it what to do, and

would certainly not expect it to be perfect. Just as Anne and the other girls would tolerate and negotiate around imperfections in people, as a programmer she 'makes no demand that her programs be perfect'. The soft master negotiates with the computer rather than the user 'about just what should be an acceptable program (Turkle, 1984a, p. 111). In Turkle and Papert's model of programming psychology, only proponents of 'male mastery' are obsessed with debugging — removing 'the small errors'. Computer programming as a profession is located in what Turkle casually refers to as 'real organisations' (Turkle, 1996, p. 51). To avoid accident or social and economic damage in this real world, the 'small errors' are critical: programs have to be not just debugged, but also debuggable. This means that, for maintenance purposes, people other than the individual programmer have to be able to understand how the program has been written, and ideally to pinpoint a clearly discrete sub-section that contains the bug. Furthermore, if it is to be of use, the program itself needs to translate to the users' experience of their real world environment and tasks. In Turkle's evaluation, programming is a means of 'working through personal issues relating to control and mastery' (Turkle, 1980, p. 15). It is the user who appears to be most significantly left out of this personal and tactile relationship between programmer and computer. And within a programming team, the individualistic commitment is at odds with the communicative practice essential to holding large projects together. Focusing on 'negotiation, relationship, and attachment' (Turkle & Papert, 1992, p. 9) with machines at the expense of real people is not something that is traditionally perceived as a feminine trait.

3.2. Real world or Disneyland?

In her later popular work, Turkle is concerned with MUDs, cybersex, and other fantasies, rather than the real world of programming. It is apparent in this work that she confuses the act of programming a computer with the usable software product. And it is consequently programming as well as software that is confused with Disneyland, where 'the representation exists in the absence of the real thing' (Turkle, 1996, p. 47). Post-modern theorists may well 'write of simulacra, copies of things that no longer have originals' — but however intangible software is, it does represent something — and the fidelity of the representation of real-world needs and environments is vitally important. Relating to computer objects, or 'lines of computer code' (Turkle, 1996, p. 59) in the way that Turkle suggests, may be at the expense of the real world people and things that those objects and lines represent: real-world environments are not like Disneyland's Main Street (Turkle, 1996, p. 47, 236).

3.3. Art, design, and the abstract

The metaphors and applications of programming suggest the possibility of artistry: we 'write' a program in a programming 'language'; and Logo, the language that Turkle and Papert focus on, is commonly used to draw shapes. Turkle and Papert exemplify soft mastery through people who are poets, musicians, potters, and artists. The real artist does not normally communicate the process of composition — nor is it usually desirable to do so: it is the final work of art that the artist 'communicates', multivariously, ambiguously, and indirectly. They ignore, however, the implications of transferring this pattern to software development.

Software produced in this way is unlikely to be maintainable: only the 'artist' would be able to fix or update it.

The 'soft mastery' school of thought not only denies the need to document software design, but the very need for a design stage at all. In general, Turkle tends to ignore or overlook the distinction between program design and programming. Because designs are made regardless of the language of implementation, they may be perceived as indicative of the 'abstract' style. Grundy goes so far as to say that soft masters 'do not get involved in advance planning' (Grundy, 1996, p. 142).

Edwards (1990, p. 105) sees computers as 'a medium for thought', and compares it as such with language: 'In order to think with a computer, one has to learn its language'. Design, however, also mediates between human thought and the programming languages 'understood' by the computer. The informal language of design — sometimes referred to as 'structured English' — is intermediate between natural language and high-level programming language, between a user's requirements and a computer solution.

3.4. Documentation and egoless programming

In order to communicate the purpose and functioning of program elements, programmers write not just the program code, but additional 'documentation'. This is written in natural language, and may be separate from or embedded into the program code. It is written, not for the computer (which skips over embedded comments), but for people. Documentation might therefore be reasonably generalised as a communicative practice. Turkle, however, characterises it as asocial, part of a culture of 'individualism': to the concrete-style hacker, it is 'a burdensome and unwelcome constraint' (Turkle, 1980, p. 21). The magician of the machine wishes to share the secrets of their wizardry only with the computer.

Egoless programming is based on the need for open, shared and understandable code within a team-working context. It lies at the heart of the dominant 'formal' culture of programming, and is essential to teamwork on large projects (Weinberg 1971, p. 72). Abstraction allows the parts of a large programming project to be shared out. The sense of a development 'lifecycle' — where the whole is larger than the individual parts — may reasonably be characterised as more holistic than the solipsism of the programmer who sustains an intimate relationship with the computer.

In the 'formal' culture that Turkle eschews, 'problem' programmers are ego programmers, they are 'territorial', and resist peer review (McConnell, 1998). Her 'concrete' programming culture, defined by 'negotiation and experimentation with the machine' and by an antipathy toward documentation and design (Turkle & Papert, 1990), is inevitably in tension with the 'egoless' aspect of this culture. The account Turkle and Papert give of children (Anne and Alex) and college students (Robin and Lisa) is of learner programmers with a narrow focus on self-satisfaction: while this is probably inevitable with learners assigned to small problems that do not require collaboration, Turkle and Papert are emphatic that this approach is a completely 'different way of knowing', an equal style rather than an early evolutionary stage of knowing. Software, however, is a people business, and fundamental problems arise if people — and women in particular — are led by Turkle and Papert's analysis into adopting an 'ego' approach. Software is developed by teams of people within real-life scenarios involving real

people. From benefit clerks and claimants to airline pilots and passengers, they have real needs. However unfortunate it may be, those needs are unlikely to be met by an approach that treats programming as a private, self-satisfying relationship with the computer.

4. Psychology, context, and tactility

4.1. What context?

Turkle frequently superimposes onto programming concepts from psychology and psychoanalysis. The concrete, hacker style is conflated with Carol Gilligan's description of a countercultural, concretely contextualised (as opposed to abstract) style of moral reasoning. The hacker style is thereby associated with a feminine, connected style. Also adapted is the notion, from object relations theory, that females are more given to attachment because they do not have to separate their identity from the mother. To Turkle, the 'images' of object relations theory 'suggest' a relation between programming style and gender (Turkle 1984b, p. 108). As women are assumed to be good at forming relationships, they are further assumed to want 'a personally meaningful relationship with a computational object', to instinctively need to 'treat the computer as much like a person as they can' (Turkle & Papert, 1990, p. 145, 149). If females are strongly linked to a concrete style of reasoning, and are less 'abstracted' from interior space, then, it is asserted, they will be naturally predisposed to a 'concrete' rather than an 'abstract' style of programming.

Gilligan, however, refers to concrete reasoning in terms of real-world contexts. Turkle translates Gilligan's sense of the 'importance of attachment in human life', into women's preference for 'attachment and relationship with computers' (Turkle & Papert, 1990, p. 157, 150); Gilligan's need to 'stay in touch with the inner workings' of arguments, into the need to stay in touch with the inner workings of the machine. The concreteness of the hands-on, experimenting programming style resides therefore in the computer rather than in the real world of the client or user. It may conversely be argued that it is actually abstraction which allows for Gilligan's 'contextual and narrative' mode of thinking. The expressiveness of abstraction allows reasoning within the context of the real world, and provides the 'language' to bridge between human understanding in the real world and its representation on computer.

The computer-centric culture of programmers is often blamed for the frequent failure of computer software: they have communicated with the technology rather than with the people who need and will use their software. Indeed, the 'millennium bug' would not have arisen if programmers had been more connected to the real world than to the immediate burden on their computer's 'memory'.

4.2. Bricolage

Turkle and Papert also harness Lévi-Strauss's use of the French term 'bricolage' to denote the 'concrete science' of non-Western 'primitive societies' (Turkle & Papert, 1990, p. 135). The hacker's penchant for 'experimentation' invokes a comparison with tactile experimentation in the sciences: the 'soft master' with a 'concrete' style is designated a 'bricoleur'.

Notwithstanding that scientific experimentation is an integral part of mainstream 'canonical' science — as is implicitly acknowledged by references to scientific discovery and Nobel laureates (Turkle & Papert, 1990, p. 130) — it is not in any meaningful way analogous to programming by trying things out on the computer. There is perhaps an implicit primitivism whereby women as well as 'primitive' societies are automatically associated with what is regarded as 'natural'.

4.3. 'Abstract': what's in a word?

Turkle and Papert present a 'concrete' style of programming as a hands-on style that is intimately closer to reality, lends itself to a communicative, 'soft' style, and is thereby a 'feminine' approach. The opposite of the 'concrete' is taken to be 'abstract', and is interpreted simplistically as an antonym to concrete — as in 'an abstract idea' (Turkle & Papert, 1990, p. 131).

Within this framework, however, *everything* in programming might be conceived of as 'abstract', in that it is the logic rather than the physical medium of a program that constitutes its essence. Software per se is 'abstract': it is, invisible and untouchable, available to the intellect rather than to the senses. But it is 'abstract' in a more significant sense: as a final product — and within its development — the computer-specific detail of how it works has been removed (abstracted) and replaced by detail that is understandable by people. Where Turkle and Papert associate the concrete style of reasoning with a 'closeness to objects' (Turkle & Papert, 1990, p. 147), it may be argued that the concrete 'objects' of the machine are either real microchips, bits, registers, buses, disks; or virtual interface 'objects'. Consequently, abstraction can actually be viewed as the means by which the 'concrete' objects close to the machine are used to represent the real world: it is only by abstracting low-level detail that sofware can move closer to the human user.

Abstract does not mean, as it does in fine arts, something that has no representational qualities. In program design, abstraction signifies the removal of complex computer-specific detail and its replacement by human-specific actions and things. In a digital world, it is abstraction that facilitates a recognisable representation of reality. Detail is concealed within 'abstractions' that are more 'concrete' and less complex. The sense of the abstract in programming discourse arguably is directly antithetical to that of 'abstract' art: while in abstract art it is figurative representational details of reality that have been removed, in program design and development abstraction translates detail of the computer's reality into details of the user's reality.

In breaking down a problem, abstraction also provides scope for creativity and expression. There is seldom one orthodox 'reality': the 'ideal' representation will inevitably vary according to the individual (Cox & Brna, 1995). Indeed, the stronger the machine's role in the solving of a problem, the less expressive and communicative the process is likely to be for the programmer(s).

A further confusion with abstraction is the frequent failure to distinguish between teaching programming in an 'abstract' way, and 'abstraction' as a tool in problem solving (Turkle & Papert, 1990; Stepulevage & Plumeridge, 1998). Discovery-based learning and reinforcement through practice do not represent an approach to programming, concrete or otherwise, but

rather a pedagogical approach to the *teaching* of programming. One of its advantages is that it can serve to build up a genuine understanding of abstraction: abstraction can be used and taught in practical ways. While exemplification and practice may frequently be inadequate in programming courses, there is also often insufficient emphasis on making sense of software development through abstraction and its practical benefits. Indeed, Hoadley et al. (1996, p. 109) advocate 'instruction that emphasises abstract understanding', and can indicate without inconsistency that learners need 'practice and concrete examples' in order to gain an abstract understanding of patterns and reusable templates of problem-solving.

5. Transparent and opaque boxes

In the same way as an assessment of 'abstraction' must depend on what it is that is being abstracted, so the nature of transparency is defined by what it is that can be seen through the transparency: in the case of the black-boxing of routines in programming, whether it is the machine or the task that is visible. Turkle briefly glimpses this ambiguity in *Life on the Screen* (Turkle, 1996, p. 42), but the confusion is compromising. On the one hand, the soft-mastery of her Harvard students is defined by their desire to experience programming as transparent — hence their resentment of black boxes: Lisa, a poet, 'resents the opacity of prepackaged programs' and wants to take them apart or write her own (Turkle & Papert, 1990, p. 128). She is 'frustrated with black-boxing or using prepackaged programs' (Turkle & Papert, 1992, p. 7). On the other hand, George, a hard-master physicist, in language remarkably reminiscent of that used to describe the resentment the soft master feels for black boxes, also 'feels threatened by opaque objects that are not of his own devising' (Turkle, 1995, p. 39). Maury, a sociology student, prefers the 'old-time modernist transparency' of MS-DOS and Windows over the opaque Macintosh because 'Windows is written in C' and he can program in C (Turkle, 1996, p. 38). Soft-master and Apple aficionado Joel, however, unlike George and Maury, loved 'to create programs that were opaque' so that one had no idea how it worked and one 'could forget that there was something mechanical beneath (Turkle, 1996, p. 40).

Robin is told by her teachers not to take apart the prepackaged programs, and not to concern herself with 'what was going on at that low level (Turkle, 1988, p. 59; Turkle & Papert, 1990, p. 134). In this way, teachers are said to impose the orthodoxy of black-boxing on unwilling women. Yet in *Life on the Screen*, Turkle favours the 'opaque' Macintosh interface because it 'hid the bare machine from its user' (Turkle, 1996, p. 23), and is antithetical to 'the traditional modernist expectation that one could take a technology, open the hood, and see inside' (Turkle, 1996, p. 35). She goes on to dismiss the 'reductive understanding' of those who want to 'open the box' (Turkle, 1996, p. 43), and approvingly observes that we 'have grown less likely to neutralize the computers around us by demanding, 'What makes this work?' 'What's really happening in there?' (Turkle, 1996, p. 42). Turkle herself was uncomfortable with the old command language interface of her Apple II because it embodies the hard master theory 'that it was possible to understand by discovering the hidden mechanisms that made things work' (Turkle, 1996, p. 33).

Black-boxing makes structured programmers 'exultant', and they 'feel a sense of power when they use black-boxed programs' because it is not to be changed by others (Turkle & Papert,

1992, p. 16). Yet Lisa herself, while she may resent programs packaged by others, and does not appear to consider letting other people use her programs, makes her own black boxes:

> she prefers to write her own, smaller, building block procedures even though she could use prepackaged ones from a program library (Turkle and Papert, 1990, p. 133)

It is baffling that this is taken to exemplify a non-structured approach: it appears in fact to be a clear example of a novice programmer carrying out structured programming and 'divide-and-conquer' techniques. These 'building block procedures' may not be prepackaged, but they serve as 'black boxes' nonetheless. Within the same article, Turkle & Papert maintain that it is the soft master who makes the 'demand for transparency' (Turkle & Papert, 1990, p. 133) and when confronted with a prepackaged program written by someone else 'wanted to take it all apart' (Turkle & Papert, 1990, p. 134); and then go on to indicate that it is hard masters who demand 'transparent understanding' of programs made by others, and want to know 'how the program works' (Turkle & Papert, 1990, p. 140).

To write 'her own, smaller, building-block procedures', Lisa would have to make use, at some level in her programming language, of procedures that have been prepackaged by others. Each level up the ladder represents a more 'abstract' level than the level below. The higher level 'language' allows the programmer to use functionalities or data structures without knowing the details of how they are implemented at a lower level. At each level, the unnecessary detail below has been abstracted. In a very general sense, the entire enterprise of computer language is to move towards people and away from computers via abstraction.

In using any feature of a language above binary code, programmers are using a 'prepackaged routine', an abstraction, a 'black box'. A simple integer calculation — such as 2 + 2 — makes use of an abstract data type that is usually built into a programming language: we do not think about the definition of the data type and that of the addition operation. As Sebesta points out, 'all built-in types... are abstract data types' (Sebesta 1993, p. 375). The *Write* procedure in Pascal, for example, enables Pascal programmers to write text and numbers to a screen, printer, or disk. They do not need to know *how* it does this — only how to use it. Such prepackaged routines are coded using lower-level language: a lower-level 'out' procedure would write a given byte to a given computer port. While the items which *Write* deals with will be of a humanly recognisable type — words and numbers — the code that implements it will deal with variables of computer-understandable types such as WORD and BYTE, and will include hexadecimal numbers, memory addresses, processor registers, and assembly-language commands.

From the use of a meaningful variable identifier rather than a memory address, to the use of a control tool for drawing screen objects in Visual Basic rather than the low-level plotting and drawing of that object — or third-party VBX custom controls that may be plugged in and used — abstraction and prepackaged black boxes are everywhere in user-friendly computer programming. Within a team working on a large project, where nobody is able or needs to understand everything at a low level of detail, a divide-and-conquer breakdown of the problem is fundamental. Each sub-group or individual will produce routines that are black-boxed so that others understand how to use it, rather than how it works inside. The future of

programming may well rely on reusable components which end-users can plug together to meet their needs.

Procedures may be 'prepackaged' as part of the language, by the software house that developed the language compiler, the writer of a commercial library of routines, or by another member of one's programming team. A procedure encapsulates a number of operations that work together to achieve a single purpose. It can be called by name when needed, without the necessity to repeat the full sequence of its constituent operations, and combined with other procedures to perform more complex functions. In design, a procedure usually corresponds to one of the stepwise refinement (divide-and-conquer) sub-task statements of what the program needs to do.

In many programming languages, routines are defined and implemented separately. This provides a means for hiding the internal workings of modules from any program (or programmer) that uses them. The program, or programmer, therefore only know what they need to know in order to make use of the routine. This sealproofs the server module, as a client program cannot inadvertently change some aspect of the server module. In this way, black-boxing dramatically reduces the possibilities and complexities of errors and side-effects, and greatly simplifies maintenance, modification, and the isolation and correction of bugs.

6. From object-relations to object-oriented?

6.1. Abstraction or encapsulation

Turkle and Papert herald object-oriented programming (OOP) as the fulfilment and 'revaluation' of the 'concrete' approach (Turkle & Papert, 1992, p. 29): it is, they say, 'more congenial to those who favor concrete approaches', and 'puts an intellectual value on a way of thinking that is resonant with their own' (Turkle & Papert, 1992, p. 155). Turkle goes so far as to indicate that the OOP paradigm 'associated computation with the object-relations tradition in psychoanalytic thought' (Turkle, 1996, p. 296). Grundy too equates object-oriented programming with a concrete (as opposed to abstract) style of programming, interpreting the growing popularity of the object-oriented paradigm as 'a surge of interest in a concrete style of programming' (Grundy, 1996, p. 142). Yet abstraction is fundamental to OOP: as Sebesta puts it, 'object-oriented programming...is an outgrowth of the use of data abstraction' (Sebesta, 1993, p. 375). While the philosophy is different from that of functional decomposition, OOP is nonetheless a fulfilment of the promise of the 'abstract' approach, not, as Turkle and Papert would have it, of the 'concrete' approach. The features of OOP are clearly indebted to the characteristics of hard mastery.

Turkle and Papert observe that 'hierarchy and abstraction are valued by the structured programmers' planner's aesthetic', and that the formal approach 'decrees' the design of 'a set of modular solutions' (Turkle & Papert, 1990, p. 136). On the other hand, soft mastery is characterised by a 'non-hierarchical style' (Turkle & Papert, 1992, p. 9). Hierarchy, abstraction, and modularity, however, define object orientation. Gorlen, Plexico and Orlow (1990, p. 1) refer to data abstraction as the 'necessary foundation' of object oriented programming. Encapsulation in the OOP paradigm is synonymous with 'information hiding' or black-boxing:

it provides a 'cover', in Satzinger and Ørvik's words, 'that hides the internal structure of the object from the environment outside' (Satzinger & Ørvik, 1996, p. 40). The object-oriented paradigm abstracts data as well as processes, and binds the processes to the data — in a way not dissimilar to the 'hard master' who would create her or his own abstract data types (ADTs) in separate modules with a public functional interface and a private implementation. An 'object', normally an abstraction of an entity in the real world, is therefore roughly analogous to a module containing ADTs: it is the ultimate 'black box'. Its defining characteristic is that no other object needs to be aware of its insides: its internal data structure, and the methods for manipulating instances of it, are hidden. It is the behaviour, rather than the internal implementation, of an object that matter. An object prevents client programs from directly accessing its internal elements; it makes data elements and their operations

> "behave analogously to the built-in or *fundamental* data types like integers and floating-point numbers. We can then use them simply as black boxes which provide a transformation between input and output. We need not understand or even be aware of their inner work ng." (Gorlen et al, 1990, p. 1).

What Turkle and Papert disapprovingly present as the formal, hard master's alternative to Anne's 'dazzling' (Turkle & Papert, 1992, p. 15) bricolage with the masked birds, is therefore in fact a basic object-oriented strategy:

> From their point of view, Anne should design a computational object (e.g., her bird) with all the required qualities built into it. She should specify, in advance, what signals will cause her bird to change color, disappear, reappear, and fly. One could then forget about "how the bird works"; it would be a black box....Structured programmers usually do not feel comfortable with a construct until it is thoroughly black-boxed, with both its inner workings and all traces of the perhaps messy process of its construction hidden from view. (Turkle & Papert, 1990, p. 139–140)

6.2. Inheritance: connected hierarchy

As with structured programming, decomposition is essential for any substantial OOP problem, particularly where large teams of programmers are involved. The decomposition is not traditionally top-down, but hierarchy allows and is expressed in inheritance, whereby different levels of abstraction have structure. Satzinger interestingly borrows the example of a (non-gendered) baby from object relations theory in order to illustrate the concept of object hierarchy and at the same time show that hierarchy is a natural way of learning and organising information.

The new-born baby's model of the world initially consists of mother (or primary carer) and other things. This gradually extends to mother–people–other things, refining to mother–people–living things–other things, to mother–big people–people–living things-things. The characteristics and behaviours of instances of each category are inherited down the

classification hierarchy: a person, for example, inherits the properties of a living thing, and adds some extra just for itself. This process, Satzinger and Ørvik (1996, p. 11) say, 'allows the baby to infer information about newly encountered objects' both in terms of new classes (e.g., small people) and instances of classes (e.g., a new aunt). This notion of a connected 'hierarchy', based as it is on set membership and linkage rather than on value ranking (as conceptualised by Gilligan), can be seen as more analogous to Gilligan's sense of context and association. Learning is premised on a facility to link new things to that which is already known:

> Learning something new often means associating a new concept with a previously known concept while the new concept 'inherits' everything known about the previous concept. (Satzinger & Ørvik, 1996, p. 42).

This in general is antithetical to the 'soft master' desire to break apart rather than reuse that which has already been accomplished and packaged.

6.3. Messages and polymorphism

Much of the language of object-oriented programming is based on communicative metaphor. Interactions with and between objects involve sending 'messages' back and forth (rather than 'input and output', or 'commands'). These 'object relationships' whereby objects may be associated with each other are often defined through inheritance. The terminology of communication theory is used to describe message-passing: a 'sender' sends a 'message' to a 'receiver'. Where a soft master such as Turkle and Papert's Alex insists on 'repetitions of instructions' (Turkle & Papert, 1990, p. 137) to get a feel for the program, messages obviate the need for duplication of data, and help to keep the detail packaged inside objects discrete.

Meaning is a dynamic process in object as in human communication: the same 'message' may be interpreted in different ways by a variety of different receivers. This principle is described in the OOP paradigm as 'polymorphism'. A corollary of inheritance-based messaging, polymorphism allows for the same method (message) to be reused and interpreted differently by different types of black-boxed objects. This cornerstone of authentic object-orientation — message-passing and polymorphism — therefore also falls, as Graham, (1994, p. 14) notes, 'under the general heading of abstraction'.

7. Conclusion

Turkle and Papert misquote Gilligan as speaking of the 'importance of attachment in human life' (Turkle & Papert, 1990, p. 157): like the male hackers they model the 'bricolage' style on, they curtail and lose sight of the larger 'life cycle' perspective (Gilligan, 1982, p. 23), and choose a narrow attachment to the computer rather than a holistic and social understanding of the real world and its human contexts. In so misconceiving the nature of computer programming, they unwittingly seek to consign women to an amateur computer role that is wholly inappropriate and more in keeping with the original literal French meaning of *bricoleur*:

one who tinkers around at odd-jobs (as in the French D-I-Y chain, Monsieur Bricoleur!). It is a damaging fallacy that structured programming, and abstraction in particular, is inimical to women or anyone with a holistic style of learning. The aesthetic and elegance of abstraction, its power to invest complex problems with context as well as understanding and resolution, is something that simply cannot be rejected, and indeed should be utilised to make programming, and computing, available to everyone.

References

Cox, R., & Brna, P. (1995). Supporting the use of external representations in problem solving: the need for flexible learning environments. *Journal of Artificial Intelligence in Education*, *6*(1).

Edwards, P. (1990). The Army and the microworld: computers and the politics of gender identity. *Signs: Journal of Women in Culture and Society*, *16*(1), 102–127.

Frenkel, K. (1990). Women and computing. *Communications of the ACM,1990*, *33*(11), 34–47.

Gilligan, C. (1982). *In a different voice*. London: Harvard University Press.

Gorlen, K. E., Plexico, P. S., & Orlow, S. M. (1990). *Data abstraction and object-oriented programming in C++*. Chichester: Wiley.

Graham, I. (1994). *Object oriented methods*. Wokingham: Addison-Wesley.

Grundy, F. (1996). *Women and computers*. Exeter: Intellect Books.

Hoadley, C. (1996). When, why and how do novice programmers reuse code? In W. D. Gray, & D. A. Boehm-Davis, *Empirical Studies of Programmers: Sixth Workshop*. Norwood: Ablex.

Kvande, E., & Rasmussen, B. (1989). Men, women and data systems. *European Journal of Engineering Education*, *14*(1).

Laurillard, D. (1993). *Rethinking University teaching*. London: Routledge.

Mahony, K., & Van Toen, B. (1990). Mathematical formalism as a means of occupational closure in computing — why 'hard' computing tends to exclude women. *Gender and Education*, 2 (3).

McConnell, S. (1998). Dealing with problem programmers. *IEEE Software*, 15 (2).

Peltu, M. (1993). Females in tuition. *Computing*, 12, 2 December.

Satzinger, J., & Ørvik, T. (1996). *The object-oriented approach*. Cambridge: ITP.

Sebesta, R. (1993). *Concepts of programming languages*. Redwood City: Benjamin/Cummings.

Stepulevage, L., & Plumeridge, S. (1998). Women taking positions within computer science. *Gender and Education*, *10*(3), 313–326.

Sutherland, R., & Hoyles, C. (1988). Gender perspectives on logo programming in the mathematics curriculum. In C. Hoyles, *Girls and computers*. London: Institute of Education.

Turkle, S. (1980). Computer as Rorschach. *Society*, *17*, 15–24.

Turkle, S. (1984a). *The second self: computers and the human spirit*. London: Granada.

Turkle, S. (1984). Women and computer programming: a different approach. *Technology Review*, 49–50.

Turkle, S. (1988). Computational reticence: why women fear the intimate machine. In C. Kramarae, *Technology and women's voices: keeping in touch*. New York: Pergamon.

Turkle, S., & Papert, S. (1992). Epistemological pluralism: styles and voices within the computer culture. *Signs: Journal of Women in Culture and Society*, *16*(1), 128–157.

Turkle, S., & Papert, S. (1992). Epistemological pluralism and the revaluation of the concrete. *Journal of Mathematical Behavior*, *11*, 3–33.

Turkle, S. (1996). *Life on the screen*. London: Phoenix.

Weinberg, G. (1971). *The psychology of computer programming*. New York: Van Nostrand Reinhold.

APPENDIX B

Published article: McKenna, P. (2001). Programmers: concrete women and abstract men?
Journal of Computer Assisted Learning, 17(4).

Programmers: concrete women and abstract men?

P. McKenna
Manchester Metropolitan University

Abstract Sherry Turkle and Seymour Papert have identified a hands-on and experimental 'concrete' approach to computer programming as feminine, and as a fully formed way of knowing how to program, rather than as either a learning style or as a stage in development. This paper differentiates between concrete styles of learning how to program, and the concrete style of programming. Learning strategy is decoupled from programming style, and the hypothesis that women are more likely than men to prefer a concrete style of programming is tested by means of examining responses to practical examples of concrete and abstract styles. The responses suggest that there is no significant difference between women and men in their attitude toward a concrete style of programming.

Keywords: Abstract; Concrete; Gender; Learning styles; Programming; Questionnaire; Undergraduate.

Introduction

The idea that women and men program in different (but equal) ways was first advocated by science sociologist and psychoanalyst Sherry Turkle (1984b), then developed in collaboration with Seymour Papert. Her analysis, along with the corresponding stylistic dichotomy between the 'soft' or 'concrete' and the 'hard' or 'abstract' style of programming, has been subsequently accepted without serious question (Sutherland & Hoyles, 1988; Kvande & Rasmussen, 1989; Frenkel, 1990; Peltu, 1993; Grundy, 1996; Stepulevage & Plumeridge, 1998).

The ready acceptance of such a stylistic dichotomy may be due to the automatic resonance it has in traditional cultural perceptions of women and men: where the 'hard' or 'abstract' programmer uses abstraction, decomposition, structured programming and algorithmic reasoning, the 'soft' or 'concrete' stylist engages and communicates with the computer (and 'computer objects') in an artistic and holistic manner. Soft stylists, it is alleged, are naturally given to this more human style, and, Turkle and Papert claim, are repelled by the coldness of the 'abstract' approach.

Turkle and Papert have sought to validate both the concrete programming style, and a polarised dichotomy between it and the 'canonical' abstract style. The gender mapping is linked to feminist epistemology, by generalised conflations with object relations theory and the ideas of Carol Gilligan and Evelyn Fox Keller. Such an exercise has been analysed elsewhere as a superficial process, fraught with internal

Accepted 19 March 2001

Correspondence: Peter McKenna, Department of Computer Science and Mathematics, Manchester Metropolitan University, Chester Street, Manchester, M1 5GD Email: p.mckenna@mmu.ac.uk

contradictions — propagating stereotypes, linking women's capabilities once again to their 'nature', and valorising a hacker style that actually invokes detachment from the real world as a corollary of attachment to the computer (McKenna 2000).

The issue is a serious one, in that if this widely accepted hypothesis were true, equal access to programming-related education and employment would depend on teachers and trainers accepting the equal validity of the 'concrete' approach. Turkle & Papert (1992, p. 3) would be correct to suggest that giving abstraction a privileged position results in the 'exclusion' of women from computing and commentators such as Peltu (1993) would have good reason to propose that structured programming be removed from the computing curriculum.

The research on which these claims are based was conducted in the USA at Hennigan School, Harvard, and Massachusets Institute of Technology (MIT) in the 1980s, with students who were learning to program. This research was essentially qualitative, consisting of observations and a series of interviews. The methodology is not, however, described in detail in any of Turkle and Papert's publications (1980; 1984a; 1984b; 1988; 1990; 1992; 1996). While they cite 14 out of 20 girls favouring the 'soft' approach, as opposed to 4 out of 20 boys, it is not clear how these figures were arrived at. In exemplifying 'hard' and 'soft', 'abstract' and 'concrete', there are few concrete programming examples. Those that are given are largely based on the use of a special-purpose language, *LOGO*, in a school context: they tend to be intrinsically visuo-spatial in nature, and do not reflect common real-world programming scenarios. Student programmers are interviewed, and their views form the practical support for the analysis. 'Robin', who is said to be 'frustrated with black-boxing or using prepackaged programs', gives typical testimony when she is quoted, talking about 'prepackaged programs':

> 'I told my teaching fellow I wanted to take it all apart and he laughed at me. He said it was a waste of time, that you should just black box, that you shouldn't confuse yourself with what was going on at that low level.' (1992, p. 7; 1990, p. 134)

There is no description, or even exemplary identification, of 'it' — the prepackaged routine she wanted to dismantle.

The basis of this research was to describe 'it': to give novice programming students practical examples of prepackaged routines, and find out who 'wanted to take it all apart'. In this way Turkle's hypothesis could be put to the test: if true, one would expect that women would be more inclined to look inside the black box and to take it apart.

Justification of methodology

In order to test Turkle and Papert's hypothesis about gendered attitudes to black boxes in programming, it is necessary first to clarify what is meant by black boxes, and then to consider how a hypothesis about attitudes to black boxes might reasonably be tested.

The concept of black boxing in programming has a wide connotative remit: it is used, for instance, to describe a method of testing and debugging a program; and can also imply a top-down approach to problem solving. Turkle and Papert use the concept in its generally understood sense: they repeatedly equate blackboxing with opacity and abstraction — with detail being hidden. They explain blackboxing as 'the technique that lets you exploit opacity to plan something large without knowing in advance how the details will be managed'. Something that is 'thoroughly black-

boxed' has 'both its inner workings and all traces of the perhaps messy process of its construction hidden from view' (1992, p. 16). The term 'black boxing' is directly conflated with the use of opaque library procedures: 'black-boxing or using prepackaged programs'. The equation of black boxes and 'prepackaged programs' is repeatedly emphasised throughout Turkle's and publications. Lisa and Robin (who feature as a prominent part of the argument in all of Turkle and Papert's papers, and Turkle's two books) want to write their own procedures rather than 'use prepackaged ones from a program library'. Lisa '*resents the opacity of prepackaged programs*' (1990, p. 128). Again, in Turkle's *Life on the Screen*, Lisa is said to prefer '*to write her own smaller subprograms even though she was encouraged to use prepackaged ones available in a program library. She resented that she couldn't tinker with the prepackaged routines*' (1996, p. 53). What is described is a dissatisfaction with library procedures, and a desire to 'take it all apart' and look at what's 'going on at that low level'. It is this attitude that Turkle and Papert characterise as more likely to be a female trait.

The use of prepackaged routines is not quite all that is meant by black boxing — but it is the strongest exemplification of it within the terms of Turkle and Papert's argument. The separation of different levels of abstraction does not, for instance, have to be top-down in nature: it can commence at the top, the bottom, or somewhere in between (e.g. Clancy & Linn, 1992). When a top-down approach is referred to, it is used loosely as a shorthand for an 'algebraic' approach that uses opacity. So Anne's approach is generally said to be opposed to 'the top-down approach', but as an approach it specifically '*dramatises what was so salient for Lisa and Robin: the desire for transparency*'. The use of variables to store sprite colours is identified as an obstacle to transparency, and is also referred to as 'algebraic' because it assigns the colour of each bird to a different variable. Anne's nervousness about black boxes is elaborated as her not wanting '*to package her constructs into opaque containers*'. It is ironic — given Turkle and Papert's lauding of object-oriented programming as the future for concrete bricoleurs — that their description of the rejected 'structured' method in this case, is an elegant example of object-oriented programming:

> 'she should design a computational object (for example, her bird) with all the required qualities built into it. She should specify, in advance, what signals will cause it to change color, disappear, reappear, and fly. One could then forget about "how the bird works." In engineer's jargon, it could be treated as a black box.'

This passage indicates that black boxing is inimical to the concrete programmer not just in terms of the use of third-party library routines, but also as a strategy for writing and organising their own code. In 'Life on the Screen', Turkle conflates the notion of 'structured programming' with the separation of subprograms:

> 'After you create each piece, you name it according to its function and close it off, a procedure known as black boxing. You need not bother with its detail again.'
>
> (1996, p. 51)

Programs, or segments of code, are 'black-boxed and made opaque': the 'black box' is 'designed not to be touched'.

Exemplifying the use of black boxing would therefore appear to be relatively trivial: prepackaged routines are an intrinsic part of most introductory programming courses. Two aspects of this triviality, however, raise concerns: firstly, prepackaged routines are unavoidable in any high-level programming language; and secondly, bypassing such routines is undeniably more difficult. While both of these issues

point to deeper weaknesses in Turkle and Papert's analysis, both must be addressed. It is impossible to ascertain what constitutes a satisfactory level of primitives: in her earlier work, Turkle refers to her prototype concrete 'programmer X' wanting all elements of the program to be available to the programmer 'as primitives' (1980, p. 17). While the meaning of this cannot be ascertained in the absence of concrete exemplification, it was felt that it would be unreasonable to focus solely on packaging built into either the programming language or the compiler. It was therefore decided that routines which had explicitly been prepackaged by a known third-party, over and above this inherent 'prepackaged' prepackaging, should be tested. The issue of difficulty may appear to be more intractable: many students have a perhaps natural inclination to avoid difficulty! However, while there is the possibility that this might prejudice against wanting to take prepackaged routines apart, this should not impact on the question of gender preference. Relative difficulty is also not something that Turkle and Papert appear to recognise or take into account as a factor. The fact that students 'do not need to know' how something works is the crux of the black-box question: it is all the more necessary to try to establish who actually wants to know. A black box 'abstracts' the detail that the programmer does not need to know, so that they can concentrate on what it does, rather than how it does it. Turkle and Papert's hypothesis is that the learning and programming style of concrete programmers (such as Robin) compels them to want to know how it does it — to make the black box transparent, or else rewrite it themselves.

It was decided to devise concrete examples to test this inclination, and to ask learning programmers a small number of closed questions about their attitudes towards specific prepackaged routines, then conduct a statistical analysis of the results. While a quantitative strategy might be seen to imply a less contextualised and more 'abstract' approach, this was adopted precisely because of some of the problems seen to arise from Turkle and Papert's more qualitative research: the likelihood that discourse and explicit vocabulary could reproduce similar misunderstandings of transferred concepts (such as what abstraction really means in the context of programming); the prejudicial influence of the researcher's perspective, and the expectations of self-selected volunteers; and the probability that a self-conscious knowledge of gender as an issue would arise.

The 'concrete' style of programming is characterised by specific instantiation rather than general abstraction; the methods adopted here are founded upon the assumption that it is also desirable to make specific what is meant by the concepts of 'concrete' and 'abstract'. The experiential discourse of Turkle and Papert's first-year programming students inclines towards abstract generality and conceptualisation: in particular, it is characterised by an absence of clear exemplification. Turkle & Papert's interpretation of this discourse is also variously influenced by:

- a misunderstanding (on the part of the researchers as well as the respondents) of abstraction in computer programming;
- a simplistic transferral of gender ideas, from psychology and ethical reasoning, to computer programming;
- an uncritical acceptance of the transferral of styles from music and poetry to computer programming.

The simplistic interpretation of 'abstraction' as meaning the same in programming as it does in mathematics or philosophy, is compounded by a further and occasional confusion of 'abstract' with formal maths: both miss the point of abstraction in

programming — that it is the 'mathematical' implementation detail of a problem that it 'abstracts' or removes, so that it may be readily understood within the human domain. The linkages to object relations theory and to Gilligan (1982, p. 50) resulting in a masculine 'separation between subject & object' are equally simplistic.

The methods adopted were therefore chosen with the intention of minimising the scope for subjective interpretation, misunderstanding, and prejudice, by making concepts concrete and presenting a statistical analysis of responses. It was hoped that such an 'objective' approach would minimise side-effects, focus on the specifics of programming in regular 3rd generation systems programming languages, and use gender as an independent variable.

Survey questions were therefore formulated to find out whether students shared Lisa and Robin's dissatisfaction with prepackaged programs from a program library, and whether they too wanted to 'take it all apart' or write their own. Prepackaged programs were exemplified both in terms of library routines provided within the larger software development environment (Visual Basic), but also in terms of higher-level routines that had been prepackaged at course level by a tutor known to the students.

While the sample was in many respects similar to Turkle's, the strategy adopted, both in formulating the questions & analysing the responses, was different. Specific and applied questions were formulated, gauging responses on an ordinal scale, and a χ^2 test was used to compare available female & male responses

The sample

The sample group consisted of first-year programming students (17 male and 24 female), most of whom came from an arts background. In this respect it was similar to Turkle's group of students. The programming course was also similar, in that, as a beginners' course, it did not immediately reflect real-world programming problems. The students surveyed were Combined Honours students at a liberal arts college of Higher Education in Liverpool. Their first year course of study consisted of Information Technology (IT) and two other subjects. The IT provision assumed no previous experience upon entry: students generally took up IT to learn more about computers, and not to become programmers. All students had completed a compulsory first year module on programming and problem-solving. They would know enough to understand practical examples, but had not encountered abstraction and black-boxing as explicit concepts. Students had no awareness either of the debate represented by Turkle and Papert or that gender was a significant part of the questionnaire. It was therefore possible to test the hypothesis with minimised contamination: students would not prejudice their responses in relation to the debate.

The course

The first year course employed a simple environment that was developed in-house at the college: PieEater provided a two-dimensional grid environment through which a small character (the eponymous pie-eater) could be directed, eating pies on request along its way. Students could see on-screen the immediate effects of their instructions. In this respect it shares some of the characteristics of the *LOGO* environment used by some of Turkle's subjects. It does not, of course, reflect realistic or typical programming tasks, but presents a simple environment within

which basic principles may be learnt, and immediate results can be seen. The implicit teaching approach would be described as 'concrete'. It is important that teaching and learning in a concrete way — by exemplification and practical manipulation — should not be confused with programming in a concrete way. The PieEater environment represented a strongly exploratory and concrete approach to the teaching of programming, yet, for ease of understanding, it made extensive use of black-boxing, requiring students to use prepackaged routines written in Visual Basic. These routines were written by programmers other than the student, but the student is not taught how to make their own black boxes. The student would largely use these routines, with occasional direct use of routines native to Visual Basic itself.

The questionnaire

As noted above, the course made little or no explicit reference to the design concepts of abstraction or black boxes: it was therefore neither possible nor desirable to ask questions that referred explicitly to these concepts. The guiding principle in formulating the questions was to provide concrete exemplification of what it meant to use prepackaged routines. Preferences in relation to black boxes would be elicited by asking implicit and strongly exemplified questions. The customised learning environment (PieEater) made specific use of what Turkle and Papert call 'prepackaged routines', so practical examples of these were presented to students, and their reactions elicited.

Three simple questions were framed: the first asks whether the student is curious just to know the inner workings of tutor-packaged routines typical to the custom PieEater environment:

1. *In PieEater, routines were provided to enable you to move or turn the pie-eater. Have you ever wondered, or wanted to know, how the move and turn routines were implemented?*

This is the basic 'glass box' question, testing the weaker end of Turkle and Papert's thesis: it entails a desire only to look inside a prepackaged routine, to 'bother with its detail'. The second goes further, and asks if the students would prefer to actively write their own routine instead:

2. *Would you rather have written your own routine to move and turn the pie-eater, so that you knew exactly how things work?*

The third moves on to Visual Basic itself — would the student wish to write their own routine rather than use Visual Basic's prepackaged routines:

3. *Visual Basic allows you to create objects on the screen — such as buttons and windows — without having to write the code to do so yourself.*
   Would you prefer to write your own subroutine to draw a button?

This question tests the 'strong' end of the thesis.

The scale used was a simple three-point Likert scale, from definite rejection (1), through a willingness to consider the notion (2), to definite confirmation (3).

Personal data was also requested: age-group, sex, previous experience and qualifications. The additional data ensured that the spotlight was not on gender, and provided information concerning other potential variables.

Higher scale points indicate, respectively, that the student would prefer to (1) see inside a prepackaged routine; (2) write their own routine instead of a routine prepackaged by the tutor; and (3) write their own routine instead of an opaque routine prepackaged by Microsoft. If Turkle's thesis concerning attitudes to black boxes were accepted, there would be a marked difference between the response of female students and that of male students: females would register high scale points more frequently than males, and males would register low scale points more frequently than females.

As the responses were measured on a three-point ordinal scale, a χ^2 test was used to examine how much the observed response frequencies differed from the frequencies that would be expected given the null hypothesis of no difference between women and men

Question 1 This was the basic 'glass box' question.

Table 1. Observed frequencies for responses to the 'glass box' question.

Observed data - Oij scale point	1	2	3	totals
Female count	2	14	8	24
Male count	1	10	6	17
Totals	3	24	14	41

Table 2. The expected frequencies, under the hypothesis of no difference.

Expected data - Eij scale point	1	2	3	totals
Female	1.756	14.049	8.195	24
Male	1.244	9.951	5.805	17
Totals	3	24	14	41

It may be noted from this that the number of female responses at points 2 and 3 on the scale was actually lower that would be expected, and higher for 1.

Table 3. The calculation of the value of χ^2

χ^2	1	2	3	$(O-E)^2/E$
Female	0.0339	0.0002	0.0046	0.0387
Male	0.0478	0.0002	0.0066	0.0546
Totals	0.0817	0.0004	0.0112	
				Sum = 0.093

With two degrees of freedom, and a 5% significance level, the critical value of χ^2 is 5.991. With *Calc* χ^2 0.093 < *Tab* χ^2 5.991, the null hypothesis was accepted for the glass box question, and it was concluded that there was no significant difference between the responses of women and men.

Question 2 This question asked students if they would prefer to write their own routines instead of the prepackaged routines.

Table 4. Observed frequencies

Observed data - Oij scale point	1	2	3	totals
Female count	5	12	7	24
Male count	3	8	6	17
Totals	8	20	13	41

Table 5. Expected frequencies, under the hypothesis of no difference

Expected data - Eij scale point	1	2	3	totals
Female	4.683	11.707	7.610	24
Male	3.317	8.293	5.390	17
Totals	8	20	13	41

Table 6. The calculation of the value of χ^2

χ^2	1	2	3	$(O-E)^2/E$
Female	0.021	0.007	0.049	0.078
Male	0.030	0.010	0.069	0.110
	0.052	0.018	0.118	
				Sum = 0.187

Again, *Calc* χ^2 0.187 < *Tab* χ^2 5.991, and the null hypothesis was accepted. There was no significant difference between the responses of women and men to the proposition that they write their own routine instead of a prepackaged routine.

Question 3 This question asked whether students would wish to write their own routine in place of Visual Basic's own prepackaged routines.

Table 7. Observed frequencies

Observed data - Oij scale point	1	2	3	totals
Female count	19	4	1	24
Male count	13	4	0	17
Totals	32	8	1	41

Table 8. Expected frequencies for the null hypothesis

Expected data - Eij scale point	1	2	3	totals
Female	18.732	4.683	0.585	24
Male	13.268	3.317	0.415	17
Totals	32	8	1	41

Table 9. The calculation of the value of χ^2

χ^2	1	2	3	$(O-E)^2/E$
Female	0.004	0.100	0.294	0.397
Male	0.005	0.141	0.415	0.561
	0.009	0.240	0.708	
				Sum = 0.958

The null hypothesis is again accepted, with Calc χ^2 < Tab χ^2. There was no significant difference between the responses of women and men to this question.

Further observations

It may be observed that there is a marked difference in the overall responses to questions 1 and 3, respectively: while question 1 tempted a third of respondents to peek inside a prepackaged routine, only one respondent was definitely inclined to self-author a routine prepackaged in Visual Basic. Black boxes appear to be more welcome in complex situations. This is hardly surprising, given that the reduction of complexity is a major reason for the use of black boxes. It may also be noted that, apart from the fact that this single respondent to the most unpopular proposition was

female, females were actually less likely to respond at the higher end of the scale than males — and therefore slightly more likely than males to prefer black boxes. This difference, however, is not statistically significant.

Conclusion

The idea that women would be more likely than men to shun black boxes in programming, resonates with several perhaps stereotyped generalities about learning: that women prefer a holistic approach to learning; that women seek out 'connection' and interiority while men seek out objectivity and separation; and that abstraction is a male preoccupation. The resonance, however, is decidedly superficial. In programming, it may be argued, it is the ability to understand and solve problems by black boxing and abstracting detail that allows software solutions to be more than the sum of the mechanistic parts — that enable software elements to be understood all at once, and in terms of real-world and user-centred applications rather than in terms of computer operations. And if people prefer a more 'connected' style, then surely it is the users — and other members of the software development team — that they would want to connect to. To connect unnecessarily to the internal workings of blocks of code that do what they are supposed to do, can only distract the programmer from communicating with the real people involved, and is more suggestive of stereotyped male hacker behaviour. Indeed, Turkle and Papert make explicit links between the alleged 'programming virtuosos' of 'the hacker culture', and the 'concrete' style (Turkle & Papert, 1992).

As a feature of a teaching and learning strategy, the PieEater environment facilitated a strongly concrete learning style, encouraging students to explore problems in a hands-on and sometimes experimental way, and providing concrete representation of actions. In order to produce such a concrete learning environment, very high-level routines were implemented and packaged that were simple to use: in other words, very deliberate use was made of abstraction and black-boxing. Crucially, there was little desire to look inside or take apart these 'prepackaged programs', and — contrary to Turkle and Papert's central thesis — no greater desire to do so on the part of women.

Educationally, the concrete learning style in this case was a means rather than an end, and did not imply the teaching of a concrete approach to programming. What characterised the 'concrete' approach of the PieEater activities, was that the learner's coding produced immediate visual results. It is quite clear that such an approach could exist without reference to any desire to get inside black boxes. The activities provided by PieEater represented the starting-point in a learning cycle that would move on to reflection and conceptualisation, and then link back to the experimental (Kolb, 1983). For Turkle and Papert, however, the 'concrete' style of programming is not a learning style, and it is emphatically not a stage of development: it is, rather, a fully formed 'intellectual style' (1992, p. 11), at least as good as the use of abstraction. Turkle and Papert's concrete style is therefore either a hacker style, or a learning style from the early part of the learning cycle. To present either as a fully fledged style of programming, is problematic enough: to identify it with women runs the risk of marginalising half of the population in a world where real programming projects simply cannot succeed without the tools of abstraction.

Acknowledgements

Thanks are due to: Brian Farrimond, for creating and producing PieEater; Janet Farrimond, and Mary Sephton, for much-needed help with statistics, and the students of Liverpool Hope University College

References

Clancy, M. & Linn, M. (1992) *Designing Pascal Solutions*. Freeman, Oxford.

Frenkel, K. (1990) Women and Computing. *Communications of the ACM, November*, **33**, 11, 34–47.

Gilligan, C. (1982). *In a Different Voice*. Harvard University Press, London.

Grundy, F. (1996). *Women and Computers*. Intellect Books, Exeter.

Kolb, D.A. (1983). *Experiential Learning: Experience as the Source of Learning and Development*. Prentice Hall, New York.

Kvande, E. & Rasmussen, B. (1989) Men, Women and Data Systems. *European Journal of Engineering Education*, **14**, 1).

McKenna, P. (2000) Transparent and opaque boxes: do women and men have different computer programming psychologies and styles? *Computers &. Education*, **35**, 37–49.

Peltu, M. (1993) Females in tuition. *Computing*, **12**, 2 December.

Stepulevage, L. & Plumeridge, S. (1998) Women Taking Positions Within Computer Science. *Gender and Education*, **10**, 3, 313–326.

Sutherland, R. & Hoyles, C. (1988) Gender Perspectives on Logo Programming in the Mathematics Curriculum. In *Girls and Computers* (ed. C. Hoyles) Institute of Education, London.

Turkle, S. (1978) *Psychoanalytic Politics*. Burnett, London.

Turkle, S. (1980) Computer as Rorschach. *Society*, **17**, 15–24.

Turkle, S. (1984a) *The Second Self: Computers and the Human Spirit*. Granada, London.

Turkle, S. (1984b) Women and Computer Programming: A Different Approach. *Technology Review* November/December pp. 49–50.

Turkle, S. (1988) Computational Reticence: Why Women Fear the Intimate Machine. In *Technology and Women's Voices: Keeping in Touch* (ed. C. Kramarae) pp. 41-61. Pergamon, New York.

Turkle, S. (1996) *Life on the Screen*. Phoenix, London.

Turkle, S. & Papert, S. (1990) Epistemological pluralism: styles and voices within the computer culture. *Signs: Journal of Women in Culture and Society*, **16**, 1, 128–157.

Turkle, S. & Papert, S. (1992) Epistemological Pluralism and the Revaluation of the Concrete. *Journal of Mathematical Behavior*, **11**, 3–33.

© 2001 Blackwell Science Ltd, Journal of Computer Assisted Learning, 17, 386-395

APPENDIX C

Pascal Questionnaire

Questionnaire

This short questionnaire is part of a project to explore different attitudes to programming/program design. Your participation is greatly appreciated.

General information

1. Please tick your age-group:

18-24	
25-29	
30-35	
36-39	
40-45	
46-49	
50+	

2. Please tick your sex:

Female	
Male	

3. What (if any) experience of programming did you have before starting the programming course?

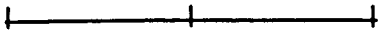
None whatsoever	
some self-taught knowledge	
a previous course	
work experience	
other (please specify)	

4. Please tick if you have any of the following qualifications:

GCSE computer studies	
GCSE mathematics	
an RSA IT qualification	
GNVQ in IT	
Access qualification	

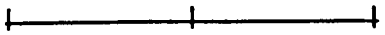
Please turn over ➔

5. In PieEater, routines were provided to enable you to *move* or *turn* the pie-eater. Have you ever wondered, or wanted to know, how the *move* and *turn* routines were implemented?



1 2 3
no way perhaps yes, definitely

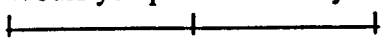
6. Would you rather have written your own routine to move and turn the pie-eater, so that you knew exactly how things work?



1 2 3
no way perhaps yes, definitely

7. Visual Basic allows you to create objects on the screen – such as buttons and windows – without having to write the code to do so yourself.

Would you prefer to write your own sub-routine to draw a button?



1 2 3
no way perhaps yes, definitely

==========

And finally:

I am interested in contributing to developments in future programming courses and in discussing my attitudes to computer programming. I would be willing to have an informal discussion in more depth at convenient time

YES	
NO	

[IF YES - my NAME is _____]

APPENDIX D

Java Questionnaire

Questionnaire

This short questionnaire is part of a project to explore different attitudes to programming. Your participation is greatly appreciated.

==========

General information

Personal data

1. Please tick your age-group:

18-24	
25-29	
30-35	
36-39	
40-45	
46-49	
50+	

2. Please tick your sex:

Male	
Female	

3. What (if any) experience of programming did you have before starting the programming course?

None whatsoever	
some self-taught knowledge	
a previous course	
work experience	
other (please specify)	

4. Please tick if you have any of the following qualifications:

GCSE computer studies	
A' Level Computing	
RSA IT qualification	
GNVQ in Computing	
A' Level Mathematics	
Access qualification	

Programming Questions

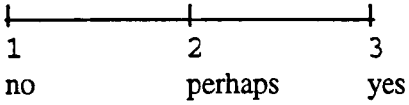
JAVA ELEMENTS

Bailey & Bailey's *element* package provides easy access to textual input and output facilities by 'packaging up' functionality from the Java AWT and IO for you.

If time allowed, would you:

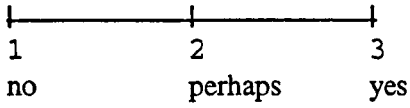
a) like to inspect and take apart the code that Bailey & Bailey wrote in implementing the *element* package, so that you could find the detail of how methods such as *readLine* were written?

Circle one number



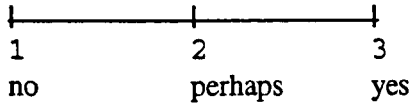
b) prefer to write and use your own routines rather than using either *readLine* or another prepackaged method?

Circle one number



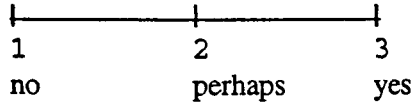
c) prefer to use the Java AWT and IO packages directly yourself

Circle one number



d) like to take apart the Java AWT and IO packages to find out how the underlying code has been implemented

Circle one number



And finally:

I am interested in discussing my attitudes to and experience of computer programming. I would be willing to have an informal discussion in more depth at a convenient time

YES	
NO	

[IF YES - my NAME :-_____]

REFERENCES

Andrews, G.R. (1997, March). *1996 CRA Taulbee Survey. Computing Research News*, (9) 2, 5-9.

Arch, E.C., and Cummins, D.E. (1989, March). Structured and Unstructured Exposure to Computers: Sex Differences in Attitude and Use among College Students. *Sex-Roles*, 20, (5/6), 245-254.

Atwell-Vasey, W. (1998). *Nourishing Words: Bridging Private Reading and Public Teaching*. Albany: State University of New York Press.

Bailey, D.A. & Bailey, D.W. (2000). *Java Elements: Principles of Programming in Java*. London: McGraw Hill.

Ball, G., Ling, D., Kurlander, D., Miller, J., Pugh, D., Skelly, T., Stankosky, A., Thiel, D., Van Dantzich, M., and Wax, T. (1997). Lifelike computer characters: The Persona project at Microsoft. In J.M. Bradshaw (Ed.), *Software Agents* (pp. 191--222). Meno Park, CA: The AAAI Press.

Barrett, M. and Phillips, A. (Eds.). (1992). *Destabilizing Theory: Contemporary Feminist Debates*. Cambridge: Polity Press.

Barry, H., Bacon, M. K., & Child, I. L. (1957). "A cross-cultural survey of some sex differences in socialization," *Journal of Abnormal and Social Psychology*, 55, 327-332.

Beauvoir, S. de (1954). *The Second Sex*.(H.M. Parshley, Trans.). Harmondsworth: Penguin. (Original work published 1949)

Bell, D. & Parr, M. (1998). *Java for Students*. London: Prentice-Hall.

Bem, S. (1993). *The Lenses of Gender*. London: Yale University Press.

Benjamin, J. (1995). *Like Subjects, Love Objects; essays on recognition and sexual difference*. New Haven: Yale University Press.

Bernstein, D. R. (1992). A new introduction to computer science. In C. D. Martin, and E. Murchie-Beyma (Eds.), *In Search of Gender Free Paradigms for Computer Science Education* (pp. 87-91). Eugene, Oregon: NECC Monograph.

Bertholf, C. F. and Scholtz, J. (1993). Program Comprehension of Literate Programs by Novice Programmers. In C. R. Cook, J. C. Scholtz, and J. C. Spohrer (Eds.), *Empirical Studies of Programmers: Fifth Workshop*. Norwood, NJ: Ablex Publishing.

Bishop, J. (2001). *Java Gently*. Harlow: Addison Wesley.

Bleier, R. (Ed.). (1986). *Feminist Approaches to Science*. Oxford: Pergamon.

Bolter, J.D. (1991). *Writing Space: The Computer, Hypertext, and the History of Writing*. Hove and London: Lawrence Erlbaum Associates.

Booth, P. (1989). *An Introduction to Human-Computer Interaction*. London: LEA.

Bourque, P., Abran, A., Moore, J.W., & Tripp, L. (1999, November/December). The Guide to the Software Engineering Body of Knowledge. *IEEE Software*, 16 (6).

Brooks, F.P. (1975). *The Mythical Man Month*. New York: Addison-Wesley.

Brooks, F.P. (1987). No Silver Bullet: essence and accidents of software engineering. *IEEE Computer*, 20(4), 10-19.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18, 543-554.

Bruckman, A. (1996, January). Finding One's Own Space in Cyberspace. *Technology Review*, 48-54. Cambridge, MA: MIT.

Brunner, C., Bennett, D., Honey, M. (1998). Girl games and technological desire. In J. Cassell, & H. Jenkins (Eds.), *From Barbie to Mortal Kombat: Gender and computer games* (pp. 72–88). Cambridge, MA: MIT.

Burgess, A. (1997). *Fatherhood Reclaimed*. London: Vermilion.

Butler, J. (1993). *Bodies that Matter*. London: Routledge.

Butler, J. (1997). *Excitable Speech: A Politics of the Performative*. London: Routledge.

Camp, T. (1997, October). The Incredible Shrinking Pipeline. *Communications of the ACM, 40*(10), 103-110.

Carmichael, H.W., Higginson, W.C., Burnett, J.D., Moore, B.G., & Pollard, P.J. (1986). Differential effects on male and female students. In H.W. Carmichael (Ed.), *Computers, children and classrooms* (pp. 106-126). Toronto, Ontario: Ministry of Education.

Cassell, J., & Jenkins, H. (1998). Chess For Girls? Feminism and Computer Games. In J. Cassell and H. Jenkins (Eds.), *From Barbie to Mortal Kombat: Gender and Computer Games* (pp. 2-45). Harvard: MIT Press.

Chivers, G. (1987). Information Technology - Girls and Education: A Cross-Cultural Review. In K.J.Davidson and C.L.Cooper (Eds.), *Women and Information Technology*. Chichester: Wiley.

Chodorow, N. (1971). Being and Doing: a Cross-Cultural Examination of the Socialization of Males and Females. In V.Gornick and B.Moran (Eds.), *Women in Sexist Society.* New York: Basic Books.

Chodorow, N. (1978). *The Reproduction of Mothering.* London: University of California.

Chodorow, N. (1989). *Feminism and Psychoanalytic Theory.* London: Yale University Press.

Cixous, H. & Sellers, S. (1994). *The Helene Cixous Reader.* London, Routledge.

Clancy, M. and Linn, M.C. (1992). *Designing Pascal Solutions: A Case Study Approach.* San Francisco: W.H.Freeman.

Cockburn, C. & Ormrod, S. (1993). *Gender & Technology in the Making.* London: Sage.

Cockburn, C. (1985). *Machinery of Dominance: Women, Men and Technical Know-How.* London: Pluto Press.

Collis, B. (1987). Sex Differences in the Association Between Secondary School Students' Attitudes Toward Mathematics and Toward Computers. *Journal for Research in Mathematics Education*, 18(5), 394-402.

Cox, B.J. (1995). There is a Silver Bullet. In N.Heap, R. Thomas, G.Einon, R.Mason, and H.Mackay (Eds.), *Information Technology and Society: A Reader* (pp.377-386). London: Sage. (Reprinted from BYTE Magazine, October 1990, New York: McGraw-Hill).

Cox, R. & Brna, P. (1995). Supporting the use of external representations in problem solving: the need for flexible learning environments. *Journal of Artificial Intelligence in Education*, 6(1).

Culley, L. (1986). *Gender Differences and Computing in Secondary Schools.* Loughborough: Loughborough University of Technology.

Curtis, B. & Walz, D. (1990). The Psychology of Programming in the Large. In J.M. Hoc et al. (Eds.), *Psychology of Programming.* London: Academic Press.

D'Andrade, R. (1966). Sex Differences and Cultural Institutions. In E. E. Maccoby (Ed.), *The Development of Sex Differences* (pp.173-204). Stanford: Stanford University Press.

Dain, J. (1992). Person-Friendly Computer Science. In Teaching Computing: Content and Methods. *Proceedings of the Women into Computing 1992 National Conference.*

Deitel, H.M., & Deitel, P.J. (1999). *Java: How to Program.* New Jersey: Prentice-Hall.

DeLoach, A. (1996, March). Grrrls Exude Attitude. CMC Magazine. Retrieved February 23, 2002, from http://www.december.com/cmc/mag/1996/mar/deloach.html.

Delphy, C. (1993). Rethinking Sex and Gender. *Women's Studies International Forum*, 16, 1, 1-9.

Delphy, C. (2000, July/August). Feminism at a Standstill. *New Left Review*, 4, 159-162.

Denvir, T. (1986). *Introduction to Discrete Mathematics for Software Engineering.* London: Macmillan.

Department for Education (1994). *Statistics of Education.*

DTI (1995). *Making the Most - Women in Science, Engineering and Technology: Building a Workforce for Sustained Competitiveness.* Department of Trade and Industry and Opportunity 2000, London.

Dworkin, A. (1987). *Intercourse.* New York: Free Press Paperbacks/Simon & Schuster.

Eden, C. and Hulbert, W. (1995, November). Gender and IT in the Primary Classroom: Building Confidence through Laptops. *Computer Education*, 81, 10–14.

Edwards, P. (1990). The Army and the Microworld: Computers and the politics of gender identity. *Signs: Journal of Women in Culture and Society*, 16 (1), 102-127.

Enloe, C. (1992, January). Silicon tricks and the two dollar woman. *New Internationalist*, 227.

Etzkowitz, H., Kemelgor, C., Neuschat, M. & Uzzi, B. (1990). The Final Disadvantage: Barriers to Women in Academic Science and Engineering. In W. Pearson Jr. and A. Fechter (Eds.), *Human Resources for Science.* Baltimore: Johns Hopkins Press.

Evans, M. (1983). Simone de Beauvoir: Dilemmas of a Feminist Radical. In D. Spender (Ed.), *Feminist Theorists (pp. 348-365).* London: The Women's Press.

Firestone, S. (1970). *The Dialectic of Sex: The Case for Feminist Revolution.* London: The Women's Press.

Foner, L. (1993). What's an Agent, Anyway? A Sociological Case Study. Technical report, Agents Group, Massachusetts Institute of Technology Media Lab.

Francis, L. J. (1994). The Relationship between Computer Related Attitudes and Gender Stereotyping of Computer Use. *Computers in Education*, 22, 4, 283-289.

Freedman, E. (2001). *Learning Styles, Culture & Hemispheric Dominance.* Retrieved on February 2002, from http://www.mathpower.com/brain.htm

French, M. (1978). *The Women's Room.* London: Andre Deutsch.

Frenkel, K. (1990). Women and Computing. *Communications of the ACM*, Nov 1990 33 (11), 34-47.

Gallop, J. (1997). "Women" in Spurs and Nineties Feminism. In E.K. Feder, M.C. Rawlinson, and E. Zakin (Eds.), *Derrida and Feminism.* London: Routledge.

Gelernter, D. (1998). *The Aesthetics of Computing*. London: Orion Books.

Gibbs, W.W. (1994, September). *Software's Chronic Crisis, Scientific American*, 72-81.

Gilligan, C. (1982). Woman's Place in Man's Life Cycle. In L. Nicholson (Ed.) (1997). *The Second Wave.* London: Routledge.

Goguen, J. (1992). The Dry and the Wet. In E. Falkenberg, C. Rolland, and E. Nasr-El-Dein El-sayed (Eds.). *Information Systems Concepts.* Amsterdam: Elsevier.

Goreau, A. (1983). Aphra Behn: A Scandal to Modesty. In D. Spender (Ed.), *Feminist Theorists.* London: The Women's Press.

Gorlen, K.E., Plexico, P.S., & Orlow, S.M. (1990). *Data Abstraction and Object-oriented Programming in C++.* Chichester: John Wiley and Sons.

Gould, S.J.(1977). *Ever Since Darwin.* New York: W.W. Norton.

Graham, I. (1994). *Object Oriented Methods.* Wokingham: Addison Wesley.

Grant, J. (1992). *Fundamental feminism: Contesting the core concepts of feminist theory.* London: Routledge.

Gray, W.D. & Boehm-Davis, D.A. (1996). *Empirical Studies of Programmers: Sixth Workshop*. Norwood: Ablex.

Grbich, C. (1992, Spring). Societal Response to familial role change in Australia: Marginalisation or social change? *Journal of Comparative Family Studies, 23,* 1.

Green, E., Owen, J., Pain, D. (Eds.). (1993). *Gendered by Design?* London: Taylor & Francis.

Green, T.R.G. (1980). Programming as a cognitive activity. In H.T. Smith and T.R.G.Green (Eds.), *Human Interaction with Computers.* New York: Academic Press.

Greer, G. (1970). *The Female Eunuch.* London: MacGibbon and Kee.

Griffin, S. (1992). Woman and Nature: The Roaring Inside Her. In M. Humm (Ed.), *Feminisms: A Reader.* New York: Harvester Wheatsheaf.

Griffiths, M. (1988). Feminism, feelings and philosophy. In M. Griffiths and M. Whitford (Eds.), *Feminist Perspectives in Philosophy,* (pp. 90-108). Bloomington: Indiana University Press.

Grint, K. & Gill, R. (1995). *The Gender-Technology Relation.* London: Taylor and Francis.

Grossman, R. (1985, August). Miss Micro. *New Internationalist* 150, 12-13.

Grundy, F. (1996). *Women and Computers.* Exeter: Intellect Books.

Gürer, D. (1995, January). Pioneering Women in Computer Science. *Communications of the ACM,* 38, 1.

Gurton, A. (1995, January). Business Comment. *Personal Computer Magazine.*

Hafkin, N. & Taggart, N. (2001). Gender, Information Technology, and Developing Countries: An Analytic Study. Academy for Educational Development/United States Agency for International Development. Retrieved February 23, 2002, from http://www.aed.org/learnlink/Publications/Gender_Book/html/3a_gender_access.htm.

Halpern, M. (1990). *Binding Time – Six Studies in Programming Technology and Milieu.* Norwood, NJ: Ablex.

Hamilton, D., McDonald, B.. King, C.. Jenkins, D., & Parlett, M. (Eds.). (1977). *Beyond the numbers game: a reader in educational evaluation.* Basingstoke: Macmillan Education.

Harcourt, W. (Ed.). (1999). *women@internet.* London: Zed Books.

Harding, S. & Hintikka, M.B. (Eds.) (1983). *Discovering Reality.* Dordrecht: Kluwer Academic Publishers.

Harding, S. & O'Barr, J.F. (Eds) (1987). *Sex and Scientific Inquiry.* Chicago: U of Chicago Press.

Harding, S. (1991). *Whose Science? Whose Knowledge?* Buckingham: OUP.
Hare-Mustin, R. T. & Marecek, M. (1988). The meaning of difference: Gender theory, postmodernism, and psychology. *American Psychologist*, 43, 455-464.

Hartsock, N. (1997). The Feminist Standpoint. In L. Nicholson (Ed.), The Second Wave. London: Routledge. Reprinted from Discovering Reality, pp. 283-310, by Harding and Hintikka, Eds., 1983, Dordrecht: Reidel.

Haughton, E. (2002, 5 March). Gender Split. *The Guardian Education, educ@guardian*, pp.2-3.

Hawkins, J. (1985, August). Computers and Girls: Rethinking the Issues. *Sex-Roles* 13, 3-4, 165-180.

Hayes-Roth, B. and Hayes-Roth, F. (1979). A cognitive model of planning. *Cognitive Science*, 3, 275-310.

Hayes-Roth, B. (1998, March). Interacting with Animate Characters: Puppets, Bartenders and Auto Salespersons. In: Proceedings of Imagina, The European Conference on Images and the Arts, Monaco, March 1998.

Henderson, J. (1989). Pagan Saxon Cemeteries: a study of the problem of sexing by grave goods and bones. In C.A. Roberts., F. Lee and J. Bintliff (Eds.), *Burial archaeology: Current research, methods and developments.* B.A.R. British Series No. 211, Oxford: British Archaeological Reports.

Henwood, F. (1993). Establishing Gender Perspectives on Information Technology: Problems, Issues and Opportunities. In E.Green, J. Owen, and D.Pain (Eds.), *Gendered by Design*? (pp. 31-49). London: Taylor & Francis.

Herring, S. (1994, June). Gender differences in computer-mediated communication: bringing familiar baggage to the New Frontier. Making the Net*Work*: Is there a Z39.50 in gender communication? Miami, FLA: *American Library Association Annual Convention*. Retrieved March 11, 1998, from http://cpsr.org/cpsr/gender/herring.txt

Herrmann, G. (1996). Women's Exchange in the U.S. Garage Sale. *Gender & Society*, 10 (6), 703-728.

Hess, R. & Miura, I.T. (1985, August). Gender Differences in Enrolment in Computer Camps and Classes. *Sex-Roles*, 13, 3-4, 193-203.

Hewlett, B.S. (1991). *Intimate Fathers: the nature and content of Aka Pygmy paternal infant care*. Ann Arbor: University of Michigan Press.

Hoadley, C., Linn, M.C., Mann, L.M., and Clancy, M.J. (1996). When, why and how do novice programmers reuse code? In W.D. Gray & D.A. Boehm-Davis (Eds.), *Empirical Studies of Programmers: Sixth Workshop*. Ablex: Norwood.

Hoc, J.-M. et al. (1990). *Psychology of Programming*. London: Academic Press.

Holderness, M. (1996, May 9). Baghdad bomb run. *The Guardian OnLine*.

Holmes, N. (2000, November). Some comments on the Coding of Programs. *Computer* 33(11), 126-128.

Hood-Williams, J. (1996, February). Goodbye to Sex and Gender. *The Sociological Review*, 44(1), 1-16.

hooks, b. (1984). *Feminist Theory: from margin to center*. Boston: South End Press.

Hoyles, C. (Ed) (1988). *Girls and Computers*. London: University of London.

Huang, Ring, Toich, Torres. (1998, March). The Gender Gap in the Computing field. *Gender Relations in Educational Applications of Technology*, 1 (1).

Huff, C., and Cooper, J. (1987). Sex Bias in Educational Software: The Effect of Designers' Stereotypes on the Software they Design. *Journal of Applied Social Psychology*, 17, 6, 519-532.

Hughes, M., MacLeod, H., Potts, C., Rodgers, J. (1985, 11 October). Are Computers only for boys? *New Society*.

Humm, M. (1992). *Feminisms: A Reader*. New York: Harvester Wheatsheaf.

Imperato-McGinley J., Peterson R.E., Gautier T., and Sturla E. (1979). Androgens and the evolution of male-gender identity among male pseudohermaphrodites with 5-alpha-reductase deficiency. *New England Journal of Medicine* 300, 1233-1237.

Irigaray, L. (1985). *Speculum of the Other Woman* (G.C. Gill, Trans.). Ithaca, NY: Cornell University Press. (Original work published 1974).

Irigaray, L. (1993). An Ethics of Sexual Difference (G.C. Gill, Trans.).Ithaca, N.Y.: Cornell University Press. (Original work published 1984).

Irwin, M.J. & Friedman, F. Taulbee Survey, Computing Research News, March 2000.

Israelsson, A-M. (1993). Educating Rita - the real challenge: some reflections on a female and human approach to teaching technology. In C. Payne (Ed.), *Education in the Age of Information*. Manchester: Manchester University Press.

Jenkins, H. (1998). 'Complete Freedom of Movement': Video Games as Gendered Play Spaces. In J. Cassell and H. Jenkins (Eds.), *From Barbie to Mortal Kombat: Gender and Computer Games*. Cambridge: MIT Press.

Kahn, K. (1999). From Prolog and Zelda to ToonTalk. In D. de Schreye (Ed.), *Proceedings of the International Conference on Logic Programming 1999.* Cambridge: MIT Press.

Keller, E. F. (1983). *A feeling for the organism: The life and work of Barbara McClintock.* San Francisco: W. H. Freeman.

Keller, E. F. (1985). *Reflections on gender and science.* New Haven, CT: Yale University Press.

Klein, M. (1946). Notes on Some Schizoid Mechanisms. In J. Mitchell (Ed.), *The Selected Melanie Klein.* New York: The Free Press.

Knuth, D.E. (1984, May). Literate Programming. *Computer Journal* 27 (2), 97-111.

Kohl, J. & Harman, M.S. (1987). *Attitudes of Secondary School Students toward Computers, Access and Usage: Do Gender and Socioeconomic Status Make a Difference?* American Sociological Association (ASA).

Kolb, D.A. (1983). *Experimental Learning: Experience as the Source of Learning and Development.* New York: Prentice-Hall.

Kozen, D. and Zweben, S. (1998, March). 1996-1997 CRA Taulbee Survey: Undergrad Enrollments Keep Booming, Grad Enrollments Holding Their Own. *Computing Research News.*

Kramarae, C. (1988) (Ed.). *Technology and Women's Voices: Keeping in Touch.* New York, Pergamon.

Kramer, P. Lehman, S. (1990). Mismeasuring Women: A Critique of Research on Computer Ability and Avoidance. *Signs: Journal of Women in Culture and Society,* 16 (10), 158-72.

Krechowiecka, I. (2002, 26 February). Never mind about maths. *Guardian Education.*

Kristeva, J. & Oliver, K.(Ed). (1997). *The Portable Kristeva.* New York: Columbia University Press.

Kvande, E. & Rasmussen, B. (1989). Men, Women and Data Systems. *European Journal of Engineering Education,* 14 (1).

Lacan, J. (1966). *Écrits: A Selection.* London: Routledge.

Laurel, B. (1997). Interface Agents: Metaphors with Character. In J. Bradshaw (Ed.), *Software Agents.* Meno Park, CA: The AAAI Press

Laurillard, D. (1993). *Rethinking University Teaching.* London: Routledge.

Lechte, J. (1994). *Fifty Key Contemporary Thinkers: From Structuralism to Postmodernism.* London: Routledge.

Levin,T. & Gordon, C. (1989). Effect of Gender and Computer Experience on Attitudes toward Computers. *Journal of Educational Computing Research*, 5, 1, 69-88.

Levi-Strauss, C. (1966). *The Savage Mind.* London: Weidenfeld & Nicolson.

Lévi-Strauss, C. (1969). *Elementary Structures of Kinship.* Boston: Beacon Press.

Liao, Y. (2000). A Meta-Analysis of Gender Differences on Attitudes toward Computers for Studies Using Loyd and Gressard's CAS. In J. Bourdeau and R. Heller (Eds.), *Proceedings of ED-MEDIA 2000* Montreal (pp.570-576). Norfolk, VA: AACE.

Linn, M.C. (1985). Gender Equity in Learning Environments. *Computers and the Social Sciences*, 1 (1), 19-27.

Linn, M.C., & Clancy, M.J. (1992). The Case for Cast Studies of Programming Problems. *Communications of the ACM*, March 1992, Vol. 35, No. 3, 121-132.

Lowe, M. & Hubbard, R. (1983). *Woman's Nature: Rationalizations of Inequality.* Oxford: Pergamon.

Lundeberg, M., Fox, P., and Puncochar, J. (1994). Highly Confident But Wrong: Gender Differences and Similarities in Confidence Judgements. *Journal of Educational Psychology.* 86(1): 114-121.

Maher, F. and Tetreault, M.K.T. (1993, January). Doing Feminist Ethnography: Lessons from Feminist Classrooms. *International Journal of Qualitative Studies in Education*, 6 (1), 19-32.

Mahony, K. & Van Toen, B. (1990). Mathematical Formalism as a Means of Occupational Closure in Computing - why 'hard' computing tends to exclude women. *Gender and Education*, 2 (3).

Margolis, J., Fisher, A., & Miller, F. (2000, Winter). The Anatomy of Interest:Women in Undergraduate Computer Science. *Women's Studies Quarterly Special Issue on Women in Science.* 104-126.

Matloff, N. (2001). Debunking the Myth of a Desperate Software Labor Shortage: Testimony to the U.S. House Judiciary Committee Subcommittee on Immigration, April 21, 1998. Retrieved November 12, 2001, from http://heather.cs.ucdavis.edu/itaa.real.pdf

Mayfield, K. (2000, May). Creating more women coders. *Wired News.* Retrieved May 1 2001, from http://www.wired.com/news/culture/0,1284,36372,00.html

McConnell, S. (1998). Dealing with Problem Programmers. *IEEE Software*, 15 (2) March/April.

McKenna, P. (1996). Desexing Computing: Gender, Myths, and Stereotypes. In R. O'Connor and S. Alexander (Eds.), *4th Annual Conference on the Teaching of Computing* (pp. 222-226). Dublin: Centre for Teaching Computing.

McKenna, P. (2000). Transparent and opaque boxes: do women and men have different computer programming psychologies and styles? *Computers & Education 35*, 37-49.

McKenna, P. (2001). Programmers: concrete women and abstract men? *Journal of Computer Assisted Learning*, 17(4).

McKenna, P. and Waraich, A. (2000a). Agents For Education: the Problem Of Synthetic Personality. In J. Bourdeau and R. Heller (Eds.), *Proceedings of ED-MEDIA 2000* Montreal (pp. 627-632). Norfolk, VA: AACE.

McKenna, P. and Waraich, A. (2000b). Social Agency: the Perils of Constructing Gendered Personalities for Intelligent Agents and Avatars. *BCS Conference on Cultural Issues In HCI*, University of Luton, 5 December 2000.

Meijer, I C. (1991). Which difference makes the difference?. In J.J.Hermsen & A. van Lenning (Eds), *Sharing the Difference: Feminist debates in Holland.* London: Routledge.

Mendelsohn, P., Green,T., and Brna, P. (1990). Programming Languages in Education: The Search for an Easy Start. In J.M. Hoc, T.R.G.Green, D.J. Gilmore, R. Samurcay (Eds.), *Psychology of Programming*. London: Academic Press.

Microsoft Corporation (2001, August). FIX: Invalid Page Fault in Msvcrt.dll (Q190536). Microsoft Product Support Services, Article ID: Q190536, Retrieved 11 November 2001, from http://support.microsoft.com/default.aspx?scid=kb;EN-US;q190536.

Miles, R. (1993). *The Women's History of the World.* London: Harper Collins.

Miller, L.M., Schweingruber, H. & Brandenburg, C. (2001). Middle School Students' Technology Practices and Preferences: Re-Examining Gender Differences. *Journal of Educational Multimedia and Hypermedia*, 10(2), 125-140.

Millett, K. (1970). *Sexual Politics.* New York: Doubleday.

Miura, I.T. (1987). Gender and Socioeconomic Status Differences in Middle-School Computer Interest and Use. *Journal-of-Early-Adolescence*, 7(2), 243-253.

Money, J. (Ed.). (1965). *Sex Research: new Developments.* New York: Holt, Rinehart & Winston.

Morahan-Martin, J. (1998, March). Women and Girls Last: Females and the Internet. In *Internet Research and Information for Social Scientists '98 Proceedings.* Bristol: Institute for Learning and Research Technology.

Morris, J. (1989). *Women in Computing.* London: Reed.

Myers, B., Hudson, S.E., and Pausch, R. (2002). Past, Present, and Future of User Interface Software Tools. In J.M.Carroll (Ed.), Human-Computer Interaction in the New Millennium (pp. 213-234). Oxford: ACM Press. (Reprinted from ACM Transactions on Computer-Human Interaction, 7(1), March 2000).

National Science Foundation (USA) (1998). Final Report, NSF Workshop on a Research Program for the 21st Century. Retrieved 25 May 2000, from http://www.cs.umd.edu/projects/SoftEng/tame/nsfw98/.

Naur, P. and Randell, B. (Eds.). (1968). *Software Engineering: Report on a Conference by the NATO Science Committee.* Brussels: NATO Scientific Affairs Division.

Neilsen NetRatings (2001). NetRatings, Inc. Retrieved 3 June, 2001 from http://www.nielsennetratings.com/.

Newton, P. (1991). Computing: An ideal occupation for women? In J. Firth-Cozens & M.A. West (Eds.), *Women at Work: Psychological and Organisational Perspectives.* Buckingham: Open University Press.

Nicholson, L. (Ed.). (1997). *The Second Wave.* London: Routledge.

Oakley, A. (1972). *Sex, Gender & Society.* Aldershot: Gower.

Oliver, K. (1993). *Reading Kristeva: unravelling the double-bind.* Indianapolis: Indiana University Press.

Ortner, S. (1972). Is Female to Male as Nature Is to Culture? In M. Evans (Ed.), *The Woman Question.* London: Fontana.

Parsons, D. (1998). *Introductory Java.* London: Letts.

Pearl, A., Pollack, M., Riskin, E., Thomas, B., Wolf, E., and Wu, A. (1990, November). Becoming a computer scientist. *Communications of the ACM* 33 (II), 47-57.

Peltu, M. (1993, December). Females in tuition. *Computing*, 12 (2).

Pennington, N. & Grabowski, B. (1990). *The Tasks of Programming*. In J.-M. Hoc, T.R.G. Green, R. Samurcay and D.J. Gilmore (Eds.), *Psychology of Programming*. London: Academic Press.

Pennington, N. (1987). Comprehension Strategies in Programming. In G. M. Olson, S. Sheppard, and E. Soloway, (Eds.), *Empirical Studies of Programmers: Second Workshop (pp. 100-113)*. Norwood, NJ: Ablex.

Personal Computer Magazine (1996, March). *The Bulletin Board.*

Petta, P. (1998, October). Emotional Software Agents: Principled Emotion Synthesis in Situated Software Agents, Austrian Research Institute for Artificial Intelligence (ÖFAI). Retrieved on Oct 23, 1999, from www.ai.univie.ac.at/oefai/agents/esa.html.

Pisanich, G. and Prevost, M. (n.d.). Representing Human Characters in Interactive Games. Photo-Genetic Labs Silicon Studios, Inc., San Jose, CA. Retrieved October 12, 1999, from http://reality.sgi.com/prevost_studio/personality.html

Plant, S. (1996). On the Matrix: Cyberfeminist Simulations. In R. Shields (Ed.), *Cultures of Internet.* London: Sage.

Pohl, M. (1997). The Internet: a 'feminine' technology? In R. Lander and A. Adam (Eds.), *Women in Computing*. Exeter: Intellect Books.

Provenzo, E. (1991). *Video Kids.* London: Harvard University Press.

Pruett, K. D. (1987). *The Nurturing Father: Journey Toward the complete man.* New York: Warner Books.

Questions Féministes Editorial Collective (1977). Variations on Common Themes. In E. Marks & I. de Courtivron (Eds.), *New French Feminisms: An Anthology* (pp. 212-230). Amherst: University of Massachusetts Press.

Rebelsky, S.A. (2000). *Experiments in Java: an introductory lab manual.* Harlow: Addison Wesley Longman.

Reilly, S. (1997). A Methodology for Building Believable Social Agents. *Proceedings of Agents* '97. Marina del Rey, CA.

Rich, A. (1981). *Compulsory Heterosexuality and Lesbian Existence.* London: Onlywomen Press.

Rodgers, C. (1998). The influence of The Second Sex on the French feminist scene. In R. Evans (Ed.), *Simone de Beauvoir's The Second Sex: New interdisciplinary essays.* Manchester: Manchester University Press.

Rossi, A. S. (1970). Sentiment and Intellect: the story of John Stuart Mill and Harriet Taylor Mill. In A.S. Rossi (Ed.), *Essays on Sex Equality* [by] John Stuart Mill & Harriet Taylor Mill. Chicago: University of Chicago Press.

Satzinger, J. & Ørvik, T. (1996). *The Object-Oriented Approach*. Cambridge: ITP.

Savitch, W. (2001). *JAVA: An Introduction to Computer Science & Programming*. New Jersey: Prentice-Hall.

Sax, L.J. (1995). Predicting Gender and Major-Field Differences in Mathematical Self-Concept During College. *Journal of Women and Minorities in Science and Engineering*, 1 (4), 291-307.

Schneiderman, B. (1998). *Designing the User Interface*. Harlow: Addison-Wesley.

Scott, J.W. (1988). Deconstructing Equality-versus-Difference: or, the uses of poststructural theory for feminism. *Feminist Studies*, 14, pp. 13-48.

Sebesta, R. (1993). *Concepts of Programming Languages*. Redwood City: Benjamin/Cummings.

Seeger, P. (1979). *Different therefore Equal* [Vinyl Recording]. Beckenham, Kent: Blackthorne Records.

Segal, L. (1999). *Why Feminism?* Cambridge: Polity.

Sellers, S. (1991). *Language and Sexual Difference*. London: Macmillan.

Shade, L.R. (1993, August). Gender Issues in Computer Networking. *Community Networking: the International Free-Net Conference*, Ottawa. Retrieved on 3 May 1999 from http://www.cpsr.org/cpsr/gender/leslie_regan_shade.txt.

Shashaani, L. (1993). Gender-based differences in attitudes towards computers. *Computers and Education*, 20(2), 169-181.

Sheehy, N.P. (1987). Nonverbal behaviour in dialogue. In R.G. Reilly (Ed.), *Communication failure in dialogue and discourse*. Amsterdam: Elsevier.

Siann, G., & MacLeod, H. (1986). Computers and children of primary school age: issues and questions. *British Journal of Educational Technology*, 2(17), 133-144.

Siann, G. (1997). We Can, We Don't want to: Factors Influencing Women's Participation in Computing. In R. Lander & A. Adam (Eds.), *Women in Computing*. Exeter: Intellect.

Skansholm, J. (2000). *Java from the Beginning*. Harlow: Pearson.

Sky Customer Magazine (2002, February). Our top 10 action babes. From Buffy to Lara Croft: who tops our list?

Smith, J. & Balka, E. (1988). Chatting on Feminist Computer Networks. In C. Kramarae (Ed.), *Technology and Women's Voices: Keeping in Touch.* London: Routledge & Kegan Paul.

Smith III, J.P., diSessa, A.A., & Roschelle, J. (1993) Misconceptions reconceived: A constructivist analysis of knowledge in transition. *The Journal of the Learning Sciences*, 3(2), 115-163.

Smith, R.B. & Ungar, D. (1995, August). Programming as an Experience: The Inspiration for Self. *ECOOP '95 Conference Proceedings*, Aarhus, Denmark.

Spertus, E. (1991). *Why are There so Few Female Computer Scientists?* Artificial Intelligence Laboratory at MIT.

Stanton, W. (1960). *The Leopard's Spots: Scientific Attitudes Toward Race in America, 1815-59.* Chicago: University of Chicago Press.

Stein, L. A. (1999, September). Challenging the Computational Metaphor: Implications for How We Think. *Cybernetics and Systems* 30 (6):473-507.

Stepulevage, L. & Plumeridge, S. (1998). Women Taking Positions Within Computer Science. *Gender and Education*, 10 (3), 313-326.

Stoller, R.J. (1968). *Sex and Gender: on the development of masculinity and femininity.* London: Hogarth Press.

Strober, M.H. & Arnold, C.L. (1987). Integrated Circuits/Segregated Labour: Women in Computer-Related Occupations and High-Tech Industries. In National Research Council (Ed.), *Computer chips and Paper Clips: Technology and women's Employment.* Washington DC: National Academy Press.

Strok, D. (1992, August). Women in AI. *IEEE Expert*, Vol.7, No.4.

Subrahmanyam, K., & Greenfield, P.M. (1994). Effect of video game practice on spatial skills in girls and boys. *Journal of Applied Developmental Psychology, 15,* 13-32.

Sutherland, R. & Hoyles, C. (1988). Gender Perspectives on Logo Programming in the Mathematics Curriculum. In *Girls and Computers*, ed. C. Hoyles. Institute of Education, London.

Turkle, S. & Papert, S. (1990). Epistemological Pluralism: Styles and Voices within the Computer Culture. *Signs: Journal of Women in Culture and Society*, 16(1), 128-157.

Turkle, S. & Papert, S. (1992). Epistemological Pluralism and the Revaluation of the Concrete. *Journal of Mathematical Behavior*, 11, 3-33.

Turkle, S. (1978). *Psychoanalytic Politics*. London: Burnett.

Turkle, S. (1980). Computer as Rorschach. *Society*, 17, 15-24.

Turkle, S. (1984a). *The Second Self: Computers and the Human Spirit*. London: Granada.

Turkle, S. (1984b). Women and Computer Programming: A Different Approach. *Technology Review*, November/December, 49-50.

Turkle, S. (1988). Computational Reticence: Why Women Fear the Intimate Machine. In C. Kramarae (Ed.), *Technology and Women's Voices: Keeping in Touch*. New York, Pergamon.

Turkle, S. (1996). *Life on the Screen*. London: Phoenix.

UCAS (1996). Press Release: *Applications for 1996 Entry*.

UCAS (1996-2000). Data Archive. Retrieved from http://www.ucas.ac.uk/figures/archive/index.html on 21 December 2001.

UCAS (2001). Annual Datasets. Retrieved from http://www.ucas.ac.uk/figures/fps/ads.html on 23 February 2002.

Ullman, E. (1997). *Close to the Machine*. San Francisco: City Lights.

Ungar, D. & Smith, R.B. (1991). SELF: The Power of Simplicity. *Lisp And Symbolic Computation: An International Journal*, 4 (3).

Urbanski, M. (1983). Margaret Fuller: *Feminist Write and Revolutionary. In D.* Spender (Ed.), *Feminist Theorists*. London: The Women's Press.

Wajcman, J. (1991). *Feminism confronts Technology*. Cambridge: Polity Press.

Wearing, B. (1984). *The Ideology of Motherhood*. Hemel Hempstead: Allen & Unwin.

Weinberg, G. (1971). *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold.

Wilder, G., Mackie, D. And Cooper, J. (1985). Gender and computers: two surveys of computer-related attitudes. *Sex-Roles*, 13, 3/4.

Willmott, R. (1996, November). Resisting sex/gender conflation: A rejoinder to John Hood-Williams. *The Sociological Review*, 44(4), 728-745.

Wilson, S. (1977). The use of ethnographic techniques in educational research. *Review of Educational Research,* 472, 245-265.

Wollstonecraft, M. (1792). *A Vindication of the Rights of Woman.* Boston: Peter Edes for Thomas and Andrews.

Woolston, C. (2001, October). The Gender Gap in Science. *The Chronicle of Higher Education.* Retrieved on February 23, 2002, from http://chronicle.com/jobs/2001/10/2001102201c.htm

Yeshno, T. and Ben-Ari, M. (2001, April). Salvation for Bricoleurs. In G. Kadoda (Ed.), *Proceedings of 13th Workshop of the Psychology of Programming Interest Group* (pp. 225-235). Bournemouth: PPIG.