Object-Oriented Software Representation of Polymer Materials Information in Engineering Design

Thesis

UNRESTRICTED

Object-Oriented Software Representation of Polymer Materials Information in Engineering Design

Sean Paul Ogden, BE (elec)

Submitted as requirements for a Doctorate of Philosophy in the Discipline of Computer Science, Materials Engineering and Design

3 March, 1998

DATE OF SUBMISSION: 31 MARCH 1998
DATE OF AWARD: 29 JANUARY 1999

Abstract

The software application POISE, Polymer Objects in a Smalltalk™ Environment, integrates knowledge representation, user interfaces, and data management; a system of tools for the materials domain expert involved in design. Engineering design solutions initially build from generalisations. POISE represents multiple levels of generalisations from classifications of polymer information.

The class-instance paradigm classifies software objects. An object's behaviour is an exclusive function of its class. Polymer's behaviours are a function of multiple orthogonal factors, like chemistry and processing, therefore multiple orthogonal classes must represent polymers. Taxonomy only represents one of these factors. The Enhancer mechanism resolves this conflict between classification and representation.

Polymer classification is not well established, with new materials evolving. The software compensates by evolving the classification schema. Guided with a specialised interface tool, the domain expert updates the schema by adding new polymer families and re-classifying existing classes. Through analysing the generalisations in the classification, the domain expert can develop an appropriate classification. This analysis relies on the engineering properties differentiating the principal material qualities. Standard properties do not distinguish specific structural differences in polymer materials, necessitating new properties.

Material properties distinguish materials in the domain whereas the classes describe the properties of polymer objects. Domain experts add new properties to the polymer classes to distinguish polymer objects. Properties are independent objects that partially describe the class template; Partial Template Objects.

Persistence of personal design information and management of shared data requires dichotomous database management. Shared data requires multi-user access, and consequently transaction management. Transaction management in object-oriented systems often holds resources for a long duration. Transaction declaration hinders transparent access to storage, and corrupts the representation. For single-user design information, transactions are implicit with access. Database proxies provide transparent per-object transaction management to persistent design information. The WorkBase is an object-storage utility that utilises Enhancers as proxies.

Acknowledgments

This research was conducted in conjunction with research at the Open University between November 1990 and June 1994, funded by the research grant IED 4/1/1416. The research team included Adrian Demaid, Dr. John Zucker and Vanessa Spedding. I thank these people both as invaluable colleagues and as friends.

Finally, this thesis would not have been completed without the emotional and financial support of Anne Culhane.

Contents

Table of Figures

Trademarks

CAMPUS is a trademark of the CAMPUS consortium, Germany.

EPOS is a trademark of ICI Ltd.

Gemstone and OPAL are trademarks of Gemstone Ltd, California.

ISAM is a trademark of Georg Heeg, Dortmund.

PLASCAMS-220 is a trademark of the Rubber and Plastics Research Association, Plascams Technology Ltd, Shawbury, Shrewsbury, Shropshire, UK.

Smalltalk–80, Objectworks 4.0, and BOSS are trademarks of Xerox Intellegent Systems Laboratory, Palo Alto, California.

Tigre and Tigris are tradmarks of Tigre Object System; Santa Cruz, CA 95060.

Chapter 1 Introduction

Computer Aided Design (CAD) often refers only to the geometric design of shapes. The shape of a design is only one perspective on the design problem presented to the design engineer. An extended CAD system for managing all aspects of the design process must address other perspectives. One perspective is the choice of material. Within this perspective is CAD support of engineering polymer materials*. This thesis proposes a specification of polymer information, a region of knowledge or **domain**, which challenges customary computer representation. This representation, which includes evolving data types, is of particular interest to knowledge engineers. Its implementation describes new software patterns that challenge object-oriented language concepts that are of interest to software scientists. For similar reasons the database designers will find the approach compelling. The resulting application of this representation will interest polymer engineers, and potentially all materials and design engineers, for analysing their domains.

A variety of polymer properties characterises the domain of polymer information. They capture the diversity of the material and the dynamic technological advances still occurring within the industry. Each instance of a polymer material, a **grade**, has properties that are a highly complex consequence of the polymerisation reaction, chemical mix of additives and processing history. Distinguishing grades by this chemistry and history is not helpful to design. Designers need to relate properties to the behaviour of their final product. They design tests on samples of each grade for quantifying the properties that imply some behaviour of the product. The behaviour relates to the design purpose. Although there are behaviours common to many designs, such as the behaviour of strength, diverse products require different tests. Therefore, as well as developing new grades, the polymer industry dynamically develops new tests to describe the behaviour of polymers in diverse products.

A computer representation is a description of some part of the real world, the domain of the representation, on a computer. Customary implementations of computer representations, such as in many commercial relational databases, assume the description of entities in the representation do not change. Changing the representation, or "schema evolution", complicates database management. Logical inconsistencies, evolving storage in memory, integrating change with data manipulation tools and applications all generate an overhead unnecessary for most applications of database management systems (DBMS).

* Engineering polymers are synthetically produced solids composed of large molecules built from simple repeating chemical units (monomers). They have physical properties useful for many different mechanical and electrical engineering applications. References to polymer, for the remainder of this thesis, will imply engineering polymers

Computer representations supporting schema evolution are often object–oriented. Object–orientation is an approach to implementing software. Objects tie data with computer processes that manipulate the data. The description of the process is a **protocol**. Computer processing depends only on the local data and the protocol. Together they produce **behaviour** in the computer that characterises the object. A schema change in individual objects only affects the internal workings of that object. This localisation of change is a characteristic of objects called **encapsulation**, and makes schema evolution simpler to manage.

Schema evolution is still complex, even with encapsulation in the language model. Schema evolution is like software programming. The changes require knowledge of the schema to ensure each change is valid. The creator of the schema, the programmer, possesses this knowledge, not the user of the schema. A changing polymer domain needs schema evolution throughout its life, not just during programming. Empowering the user to manipulate the schema requires specialised software tools. Developing these tools requires a study of schema evolution in context of a specific schema for materials information to identify what needs to change and how to maintain a valid representation. In particular, the implementation of the language and database model considers the following general characteristics of the schema:

- Materials Classification
- Domain property inheritance
- Abstraction of domain generalisations

Software representations of materials classification exist, but representing the classification process is novel. Inheritance of domain properties and abstraction of domain generalisations are inference mechanism that follows on from classification.

Generalisation performs an important role during the conceptual analysis stage of design. For example, plastics and metal are both generalisations from the domain of materials. In the early stages of design, "the crucial decision-making steps [during design] in being able to deploy domain generalisations effectively are substantially *qualitative*."[1] A step such as approximating the design parameters to test the feasibility of the design concepts. Using the typical values of property performance to compare plastics verses metals is more effective than concluding from the specific value from a material test (like the test result for tensile strength of Huels' VESTOLEN high-density polyethylene).

Each of the listed characteristics is a unique development in the object–orientated representation of materials information. A database capable of evolving in a consistent manner while performing these tasks will question and challenge the very way objects are organised and communicated. This research presents answers to these questions in a chosen object–oriented language. The application developed to demonstrate this research, called POISE, includes the evolution of classification, evolves the descriptions of polymers, and maintains storage and user interaction throughout this evolution. To further introduce these issues requires a more detailed account of object–orientation.

1.1 Class–Instance Object–Orientation

Classification commonly organises the polymer domain. A majority of object–orientation supports classification of objects. In class–instance languages and databases, the **class** groups similar objects. The class formally defines the relationship between the structure of data and the protocols for a set of objects. An **instance** is one of these objects.

An object's behaviour depends only on the local data and the protocol. Each instance **inherits** protocols from the class and they add their local data to specialise the behaviour. Those with the same data behave the same. Different data produces similar behaviour, since the protocols are the same. The protocols describe the abstract behaviour of the class. Instances share this behaviour by **inheritance**. For software development, the motivation for inheritance is the re-use of a common protocol, which minimises coding. Inheritance also facilitates representation through the development of abstract behaviours.

Behaviour sharing complicates schema evolution. Instances share the behaviour from their class. A change in the class affects all the instances. No change in the class can apply to some objects and not others. The class can not define behaviours that only apply to some instances. For this reason, the principle of classification must be appropriate to the whole domain, and not some arbitrary portion. An appropriate classification also generates useful generalisations. Classification is important because generalisation is principle to the process of design.

Inheritance between classes generalises behaviours even further, forming levels of representation. Objects of different classes may have common behaviours. By placing the protocols for the common behaviours in a **superclass**, many **subclasses** can share the protocols by inheriting them. The result is a hierarchical classification, or taxonomy.

1.2 Object–Oriented Support of Hierarchical Classification

Taxonomy is the process of classification into an ordered hierarchy, forming the familiar family-tree shape. Each class in the classification groups similar information. This similarity is general to the members of the class and, therefore is a generalisation. The benefit of taxonomy, over arbitrary classification, is the branches of the hierarchy differentiate between classes. These differences then support comparison. A taxonomic classification of materials information can support design decisions by relating similar domain generalisations and distinguishing relevant differences.

Often a process of specialising generates the classification by distinguish the description of one class from others. Class–instance languages support specialisation in classification through **subclassing**. Subclassing extends the behaviour of a class. Each subclass inherits all the behaviour from one other class then adds its own specialised behaviours. Each subclass is a class that may be subclassed further. This forms a tree-shape hierarchy with each class branching to many subclasses. For example the subclass `Aeroplane`, a member of the class `Flying_machines` from which it inherits the general 'behaviour of flight', generates an instances *`DC10_NZ001`*. `Aeroplane` could be further subclassed by a class `DC10`, which contains behaviours that specialise DC10s from other aeroplanes. The inheritance relationship between classes and their subclasses forms a hierarchical organisation. All instances of a class share (inherit) the same protocols, thus share similar behaviour and satisfy classification.

The class provides the definition of a representation and a taxonomic classification. Classes can both represent and classify polymer information, such as those polymers considered nylons. The class `Nylon` is a template for an instance of polymer *`Nylon_Grade`*. Here an instance models a grade of polymer, a particular brand of a supplier's raw product that conforms to a set of properties. Equally, the `Nylon` class inherits behaviours describing properties from the class of `Partially_Crystalline` polymers. The network of inheriting polymer classes is a classification. The classes themselves define the structure and protocols for representing grades. This research poses the question whether the class can represent polymers exclusively without compromising the taxonomic classification.

The class–instance paradigm as interpreted by many object-oriented languages has drawbacks when representing polymer information. The drawback stems from the strict nature of inheritance between instances and classes. Classes exclusively define the properties of instances; they can not individually extend their properties. This limits the instances capability to model *`Nylon_Grade`*. All instances of a class must extend their properties together. An instance can change its membership to a subclass and add different properties to the subclass. Such ad-hock subclassing, solely for extending property descriptions of instances, conflict with the use of the class hierarchy as a taxonomy of the domain. Extending property descriptions will require a mechanism for behaviour sharing orthogonal to inheritance.

Conversely, Nylon is as much an entity as a grade. Although Nylon is an abstract concept, abstract materials have as much functionality as a grade in calculating a design. The class is not normally a computable object like instances. In some languages, the class is an object with the behaviour to create other objects and provide those objects with protocols. In addition to providing a description to grades, and evolving that description, the class of a polymer material needs to respond as a generalised material, giving responses typical of the grades it classifies.

A grade describes a brand of material, not the material itself. The material results from a common production process. It is probably subject to a quality control on a limited set of properties, a profile selected for the grade's intended use. The rest of the properties are generalisations that are similar due to the common production process. So, is a grade also abstract? Nylon is abstract because it does not reference specific examples, and it should even generalise unknown Nylon grades. Nylon is a common chemistry, and the principle of the classification is based on the belief that chemical composition strongly establishes the properties. The grade does reference specific examples. If a new sample of material does not fit the grade description, then supplier rejects the material, not the concept of the grade. If a new Nylon does not fit the abstract description of Nylon then the classification rejects the abstract description on principle.

There is no epistemological reason to distinguish a grade and an abstract material in the way object–orientation distinguishes the functions of classes and instances. Instances represent grades because the structural function of instances suitably represents the concrete property-values that data suppliers provide on grades.

1.3 Abstraction of Domain Generalisations

The application of domain generalisations, like Nylon as a design material, is a characteristic of the domain. Domain experts typically talk of the properties of Nylon in comparison to other general materials. Each generalisation from a classification forms an abstract concept. This concept abstracts a general behaviour for each property in the classification. Since designers use these abstract concepts, any knowledge base on polymer information should contain a representative entity for computation in design. The class Nylon should not simply create instances of grades but also behave as an object that abstracts the properties of those grades and provide them for design.

For example, a domain expert might consider the use of Nylon or Polypropylene for the manufacture of a washing bowl. Here, Nylon generalises the characteristics of the whole population of the class Nylon. If the properties of Nylon deem it unsuitable for the design of a washing bowl, then no grade in the class will be suitable. To a less formal degree, if Polypropylene shows desirable strengths in comparison to Nylon, then Polypropylene may be a better class to initially search for a solution.

With a populated classification, the properties of domain generalisations can be implicitly inferred through analysing the properties of member instances. Distributions of the explicit properties from grades are useful indicators of the generic behaviour of the domain generalisations. Quantitative comparisons of these distributions are possible between classes of different polymers, which provide support to qualitative decisions[1] during the search for a design solution. This process of generalisation is called abstraction, and the generalisations created are abstract polymers.

The properties of abstract materials are also a useful estimate for the value of a member grade where the property has not been measured. With the continual addition of new properties, the condition of data absence or "sparse data", is intrinsic to an evolving database. If specific data is not available then a default value may substitute. The default value is a property of a classification that sparse instances inherit. For a dynamic classification this inherited value is the same value the domain abstraction exhibits, eg the property of a Nylon grade expects a value similar to other Nylons, which is the value abstracted by the general concept of Nylon.

This relationship between the abstract concepts in the domain and member grades of a classification also strengthens the integrity of the knowledge base. New grades exhibiting a property outside the expected deviation of values in a classification are identifiable. The knowledge base can then query these entries, thus decreasing the chance of data entry error, and increase integrity.

1.4 Similar Properties

Object orientation supports sparse data through inheritance of a default value. It also allows processes specific to an object. A protocol inherited by an instance can distinguish a measured value from the sparse state and obtain a default value from the class generalisation. Alternatively, it can query the instance for other 'similar' properties, where similar is a subjective quality the knowledge developer encodes in the protocol. With this knowledge, a protocol can generate a specific process or behaviour of the instance that infers default data from a similar property or properties of the instance.

Similar properties in materials describe a different test measuring similar physical characteristics. This results in different values but similar properties will rank relative performance between materials the same. Occasionally correlation between properties can be determined within certain material contexts, eg tensile strength and hardness correlation of some polymer families. In the simplest case, a similar property may substitute for another in analytic calculations. The difficultly with using the correlation between properties of polymer materials is their uncertainty, poor accuracy and contextual dependency on the type of material and other perspectives, like geometry and environment. Calculations using the correlation, or any inherited default value, need to qualify their results, eg an audit trail. This element of the design domain has not been pursued further in this research, but its importance is identified.

The relationship between properties, such as these conditional correlations, has highlighted that besides the values describing grades the properties are themselves entities in the domain. The property contributes to the representation of grades, as a class contributes behaviour to instances. Therefore, representing the relationships between properties involves structuring and manipulating relationships within classes. In addition to a class structuring and manipulating the protocols of instances, now another representation must structure and manipulate the protocols representing properties in the classes, and thereby model the similarity between properties and their contextual application.

All class–instance languages manipulate instances. Instances are known as 'first class values'. Not all languages permit the manipulation of the class. The manipulation of classes as first class values permits the evolution of the schema for describing grades and permits experiments that relate the material properties. The schema describes domain classes as a collection of material property descriptions. These describe grades of polymers that collect or infer the values for each property. If the software class can be manipulated then an interface could empower the domain expert, not just the programmer, to add and remove property-objects that evolve a class, and propagating the schema change to subclasses and instances.

In many class-instance languages, the class is not a first class value and the class definition is static. In others, the class is an object capable of change that affects the schema of the instances they define. In these languages the classes can represent and evolve a classification of the domain. Smalltalk–80™[2] is a language belonging to the class–instance paradigm that allows classes to evolve their description. Both the description and manipulation of objects occur within a single environment without any separation of the two activities. This permits the development of a computing system that both manipulates and describes objects. Smalltalk is the language chosen for developing POISE, Polymer Objects in a Smalltalk™ Environment.

1.5 Smalltalk

Unlike most languages, Smalltalk™ is an interactive programming environment, so programming is an activity of small iterative changes to the definition of objects that are immediately active in the environment. Smalltalk™ is a large library of classes, and the objects in the environment with their source code constitute the software itself. This resource simplifies and shortens the software development process. An esoteric feature permits the manipulation of even the language compiler, which is also part of the library. It is possible to extend the language. Smalltalk has often been used as an experimental test bed for language research. Examples include extending Smalltalk for multiple inheritance[3] and developing new Actor language hybrids[4]. The research conclusions are not isolated to the Smalltalk environment but apply to any computing environment that can develop the features studied.

A section of this thesis (Chapter 4) is dedicated to language extensions. Although these extensions came about because of a need in Smalltalk, they are not believed to be unique requirements of Smalltalk for the support of materials representation. An examination into the nature of the behaviour sharing that the extensions support justifies this belief.

The first extension is the Enhancer, which is a very general mechanism for extending the messaging in Smalltalk. In a class–instance language the control of messages passes along a strict path from instance to class to superclass. The Enhancer enables individual instances to specify an extension to this existing path. Messages alien to the standard classification of the object can find meaning in the extension. The extension enhances the behaviour of the individual instances.

The second extension is the Partial Template Object (PTO). The PTO is an abstraction of the class template. Each class often defines categories of behaviours. The PTO defines an abstract category of behaviours independent of the class. The PTO then consistently installs itself on any number of specific classes. Further, the PTO maintains changes to the abstraction on the specific classes. Although this implementation of PTO affects classes, any object paradigm with a repository of behaviours could take advantage of this kind of abstraction. The mechanism is of particular interest to any system supporting schema evolution since it can quantify formal changes to the schema.

1.6 An Alternative Object–Orientation: Prototypes

The class–instance representation of materials is not the only possible course of action. An alternative approach uses **prototypes**. A prototype is an object that manages both data and protocols. Both the data and protocols are available for other prototypes to inherit. A grade as a prototype can add its own unique protocols like a class. An abstract material can respond with its own behaviours like an instance. Zucker[1] uses prototypes to represent the *purpose* of material selection in design.

The different virtues of class–instance and prototypes[5] have been well argued. The consensus is that they each describe a different type of behaviour sharing, and neither limit languages to these types. Therefore, it is more relevant to study the specifics of the behaviour sharing supported by a language. Zucker's work is reviewed for its unique contribution to behaviour sharing, which extends delegation with enforced classification. He uses this combination to represent the evolution of the design description, or the 'application perspective'. This thesis investigates the use of the Enhancer to see if it will support Zucker's objectives. The Enhancer extends classification with dynamic implicit empathy, a type of behaviour sharing similar to delegation.

An example of the application perspective of a disposable cup is the description; "Rigidly contains water at 100°C" and "Connects to a surface of less than 30°C" for a handle. This description of the application is independent of the material. It does not convey a restriction on the material properties explicitly, ie the disposable cup does not specify a material rigidity at 100°C. A prototype of this description combines information from other perspectives, such as a materials perspective, and deduces if the material satisfies the design purpose. Consider a polystyrene cup where the thermal conductivity of the material and the thickness from the geometry perspective could conclude the outer surface of the cup remains much lower than 100°C and maintains rigidity at this lower temperature. Other prototypes specialise the geometry, adding ribs to the cup, thus reducing the stiffness required of the material at 100°C. Decomposing prototypes into perspectives enables each to evolve independently and structures the design problem.

If the object model representing design abstracts design into an application perspective and a materials perspective, then the application perspective is an object that shares behaviour from objects representing materials. Other objects also share behaviour from materials, like the user-interface that displays a material. These objects are all users of materials' behaviours. The software design of the materials perspective impacts on all these objects. The software design also depends on the languages ability to traverse these object boundaries through behaviour sharing. The software design must also consider the effect schema evolution will have on the consistency of behaviours. Overcoming these difficulties in the materials perspective is the main modelling issue addressed in section Chapter 4, as it applies to Smalltalk.

1.7 POISE Tools

POISE provides a number of user interface tools for manipulating the classification. One browser empowers the user to define new properties. With another, the user assigns properties to classes within a hierarchy. The same browser also moves classes and defines new classes. A modified Smalltalk engine for schema evolution ensures consistency and supports the abstraction of new properties added to classes. These abstractions are then viewed in a third browser for comparing the general properties of classes, such as the tensile strength of Nylon.

To determine the general tensile strength of Nylon requires a significantly sized population of knowledge on Nylons. A data acquisition tool initially populates POISE by reading information from an existing materials database called CAMPUS™[6,7]. CAMPUS holds data on raw polymers from many major suppliers. CAMPUS was readily available and it contains a large population of polymer grades and a consistent set of properties. A class called `Polymer` defines these general materials properties. CAMPUS defines a chemical family property for each grade. The data acquisition tool uses this property to define a class that inherits from `Polymer`. This class generates an instance to represent and initially classify the grade.

With the tools developed and a population of grades, a separate study by Spedding[8] uses POISE to determine an "appropriate" taxonomic classification of polymers for engineering design. The classification abstracts the domain generalisations on which the designer visualises their qualitative judgements of similarity between properties. Hence, the nature of the classification affects the groupings of similarity within the domain. An appropriate classification is one that groups similar materials appropriate for the task, engineering design, and preferably design in general rather than specific design.

Spedding uses the tools to compare the abstracted properties of the CAMPUS polymer families. One observation was the wide-ranging effect additives had on the properties. So in a single polymer family significant deviation in property values were due to the different additives and masked any expected concept of similarity. The `Enhancer` was a consequence of this discovery. The `Enhancer` permits generalisation over secondary groupings orthogonal to the polymer classification. Orthogonal classes like `Film` and `Fibre` can be viewed independently of bulk engineering polymers, which are engineered for extreme geometric conditions. These grouping are orthogonal since they are a group of the whole polymer population dedicated to supporting a specific property of another perspective.

1.8 Summary of Objectives

The underlying objective is to resolve the software issues arising from implementing in Smalltalk a representation of polymer information intended for design. The essential requirements for design are the domain concepts of material properties, taxonomic and orthogonal classification, and abstraction. The domain expert declares the polymer classification and engineering properties, and the software evolves the schema accordingly. This user-defined schema represents a classification from specific grades to generalised polymers.

The intention is to build this representation into a working application. The POISE application requires a management system for the persistence of design knowledge contributed by the user, and effective graphical user interfaces for driving the tools, and import facilities for transforming relational data into the object-oriented representation. These features are significant because they must consistently perform their tasks as the schema evolves. Each esoteric phenomenon of the representation is thoroughly described, but a formal study into each is avoided since they are a consequence of performing the research into the representation. Special attention is given to the properties of the language extensions since they are a criticism of the underlying object model supporting the representation.

1.9 Introduction to the Literature Review

The use of classification in design is the principle that suggests a class–instance language strong on inheritance will provide appropriate support to engineering design decisions. This conceptual model of design is reviewed, putting classification in perspective with the task of finding a suitable material description to match the product specification. This in turn demonstrates the dependence on preconceived concepts of similarity because set theory limits classification. The question is then one of choosing a classification, an appropriate classification, for engineering design. Current literature only suggests the basis for a classification is on principles of the material's physical characteristics, and not on the use of the materials, which are only indirectly related to physical characteristics. The review then presents Spedding's in-depth analysis of classification that utilised the software product of the present research.

Substantial literature exists on the science of representing knowledge from basic computer data. Frames are introduced as a sample of this field. They are also an early application of inheritance for the representation of generalised descriptions. Inheritance is not a simple issue in knowledge representation. With classification, there are often exceptions between the generalised description and individual entities. If entities inherit characteristics from the generalised descriptions, mechanisms must permit exceptions to the inheritance of properties. This though can lead to logical inconsistencies in systems with multiple inheritance paths.

Data modelling studies the structures for containing data. All computer languages and databases are built on data models. Many still use simple record structures. Over large quantities of data, the relational model provides manipulation that is more flexible. In engineering, it has limitations due to the wide range of types of data. Each polymer property introduces another type of data to relate to the material.

Data modelling only structures data. The meaning of the structural components is simple and homogenous. Semantic data modelling classifies common types of relations. These types add more meaning to the structure of the data. The arguments for adding semantics to the data is equally applicable to developing a semantic model for adding properties to classes of polymers in an object–oriented system.

The concept of the object is addressed thoroughly. Encapsulation, messages, empathy between objects, and delegation are described. These are then related to the two main types of languages, the class–instance and the prototyping languages.

The rest of the thesis splits into three parts: Specification, Implementation and the Application.

The specification of POISE details all of the features built. Namely the data acquisition, grade instantiation, classification, generalisation, abstraction, user interface design, and data storage.

The implementation specialises on the language extensions, the Enhancer and the PTO. A number of applications utilise the Enhancer in particular, including enhancing grades with orthogonal description, enhancing classes with generalisation and abstraction, applying the Enhancer to a variation of delegation, and enhancing any object with persistence in an object storage. The PTO is part of a larger discussion on the mechanism providing schema evolution to the polymer classification.

Two chapters re-enforce the application of POISE. The first is a walk-through description that demonstrates the user interfaces and the underlying functionality. The second presents a domain expert's conclusions resulting from using the application. This domain expert, Spedding, examined the domain for appropriate classifications.

Chapter 2 Literature Review

"The claim that method may prompt inventive steps [in design] will seem rash, if not ridiculous, to some. But arguments, which can be built on the lines put forward here, will often reduce to a marching logic which leads inexorably to a minor but unmistakable invention"[9]

The representation of polymer materials information for design is itself a software design problem. The first stage of design methodology is to specify the objectives. The objectives then decompose into a number of specific software requirements for achieving each objective. One objective is to identify suitable materials. This literature review follows the current argument that browsing through a classification of materials meets this objective. Along the way, the review introduces other works contributing to other objectives in polymer materials representation.

Browsing introduces requirements on the representation of information. Browsing views groups of information. These groups need representing. Browsing traverses the relationships between groups. These relationships need representing. These groups and their relationships form a classification. The review analyses a conceptual model of the design problem to collect concepts of similarity to group materials for the designer to browse and identify the nature of the classification. The review then proceeds to review work defining similarity, abstraction, generalisation, classification, appropriate classification and problems with classification.

After specifying the software requirements, a broad solution is sought in terms of representation technology. Although the software methodology has already been identified, namely object-orientation, this should not be confused with the knowledge representation model. Although many knowledge representation models were developed in non-object–oriented software languages, an object–oriented language could implement them (and in some cases more effectively). Indeed, the review introduces knowledge representation features that object–oriented languages adopt in their object model, like inheritance, and will enhance the polymer information representation.

Finally, the review introduces in detail the object model for software development. This review provides the necessary background to convey the significance of the language enhancements found necessary to achieve the representation.

2.1 Polymer Materials Knowledge for Engineering Design

The suppliers of materials generate vast quantity of materials knowledge. Suppliers tailor much of this information to their customers, the design engineer. Demaid et al[10] characterise materials information as rich and complex, and much of it far beyond the capability of current databases to analyse. In Spedding's[8] extensive review of materials information available from suppliers, she describes "a wide variation in the form and level of detail of the information", usually disseminated by supplier data sheets. Within the information there is a subset of materials knowledge that software can help analyse. Suppliers of materials even tailor some information for software analysis.

Suppliers contributing to the CAMPUS[6,7] project agreed to a standard of data presentation. EPOS™[11] is a similar standardised system by the supplier ICI. They provide a uniform data structure and comparable data. This enables a database approach to information storage, retrieval and analysis by query. The CAMPUS database software supports materials comparison against a template. The template is a query that describes ranges of property values of interest. A query then selects all materials that satisfy all the range conditions. Once the selection result reduces to a manageable number, CAMPUS can retrieve a text description of each material. The text records information too complex to analyse by the query mechanism. At this stage, the domain expert must analyse the remaining information.

The query procedure dictates the extent a database system can analyse information before user intervention. The objective of a query procedure is to reduce the number of candidate materials. At the same time, the query should not reject materials that might not be optimal but could satisfy the design criterion through compromise.

Plascams–220™[12], a product of the Rubber and Plastics Research Association, has a similar representation to CAMPUS but advances on simple numerical comparison, and instead the query procedure ranks materials. The ranking could avoid rejecting any material, but in practice, many materials towards the bottom of the list are not useful, so an arbitrary limit is placed on list size to reject those materials.

Ranking materials against a single property criterion is simple. The difficulty arises when two design criterion conflict. This is common since optimising one property will rarely optimise another. In Plascams–220 the designer places a weighting on each property the designer wishes to optimise. The ranking algorithm can bias each property then sum the biased values for ranking. Zucker[1] analysed the ranking algorithms ability to promote suitable solutions and found they do not model well the activity of selection by designers. Consequently, potential candidates are lost far down the ranking. Further, Hopgood[13] found the inference mechanism gave a poor property with low importance a weaker ranking than a poor property with high importance. Hopgood suggests an alternative inference mechanism (AIM) that gives a ranking more in line with designer's expectations.

Others use "Fuzzy Logic" [14&15] to define a probability profile for measuring a material's suitability. Simply put, the weighting, or profile changes depending on the properties distance from a satisfactory value. All rely on the designer's judgement for weighting one property against another to correctly bias the ranking towards the desired design specification.

Ashby[16] addresses the problem of combining design constraints. First, he converts individual properties constraints to Dieter's merit indices[17]. A merit index applies the physics of the design problem to relate material properties, eg "specific stiffness, E/ρ (where E is Young's modulus and ρ is density) … large values of E/ρ are the best candidates for a light, stiff tie rod"[16]. The difficulty is often devising appropriate indices for specific design problems[18].

A design can have multiple merit index constraints. Ashby proposes that 'subjectivity is reduced or eliminated by employing the "coupling equation" method and the method of "currency exchange"'[16]. The coupling equation method combines multiple design constraints on the same merit index function. More commonly, designs have different design objectives, and therefore different unrelated merit index functions. For each objective, a judgement of value is given to its merit index. The judgement of value provides a common currency for trading off the design objectives. This currency exchange minimises the subjectivity of the judgment.

Software technology is still a long way from developing a query procedure that returns a list of ranked materials that satisfy a design specification. Judgement is still required to trade off between different criterion. In cases where there is no physical foundation for judgement, Ashby's currency exchange and Sargent's review of the problem of decisions and selection[19] are the only available approaches. Promoting judgements with physical foundation, which are always superior, will minimise subjective judgements. Although this is the objective of the computer aided design systems, there will always be need to support the subjective judgements.

Where the user is not able to specify requirements essential for conducting a database search, the metaphor of browsing[10] offers a different approach for obtaining a solution. Further, browsing information in a way that reflects the physics of the materials will promote judgements with physical foundation. Browsing has added benefits. It supports information both well represented and poorly represented, complete and incomplete. Interaction with the user is also more likely to support evolution with design.

Browsing to a solution depends on the presentation of the information to guide the designer. Software support for browsing needs to present the information in a useful way. Browsing therefore has different demands on information representation. A conceptual model of design identifies the objective of the designer, thereby identify what the software needs to present to the designer.

2.1.1 A Conceptual Model of Design

An engineering design application starts with a loose description of performance of the desired artefact. At this point, the artefact does not exist. What does exist, to a greater or lesser extent, is a Product Design Specification (PDS)[20,21]. The PDS is a functional and formal statement of requirements, not a description of the product itself. Inevitably such a specification will be incomplete and contain errors, eg a prototype of a kitchen appliance attracting dust will trigger realisation that electrostatic properties of the polymer are relevant for the saleable appearance of the appliance and the extra requirement added to the PDS. The use of a PDS to categorise the design process is discussed by Pugh[22].

As the design activity progresses the PDS will evolve. When gaining new information and correcting existing information the design problem changes and hence the PDS changes. Ill defined or ill structured problems change during the process of solving the problem, and are notoriously difficult[23,24].

Relevant parts of the PDS forms a Materials Design Specification (MDS), the materials perspective[1] of the design. Known material descriptions (MD) matching the MDS, partially satisfy the PDS. Demaid and Zucker[25] describe two measures of confidence when matching properties between descriptions:

> "How close is the description of an element in the MDS to the description of an element in the MD? **The relative description.**
> How close is the value of an element in the MDS to the value of an element in the MD? **The relative value.**"

A material can always further specialise differences in relative description. For both imaginable materials and existing material, the list of materials properties is potentially infinite.

Testing all materials across a large set of different property descriptions is impractical. It is costly for material suppliers, so they are selective in their choice of properties to test. Different polymer suppliers inevitably select different properties even for similar materials. Materials with different lists of properties, such as those between different suppliers, cannot be compared with equal confidence; they differ in relative description. To solve this problem within the polymer supply industry, four major suppliers developed CAMPUS, a database with a consistent list of polymer property values. Oberbach[7] describes the necessity of the CAMPUS development.

A consequence of a consistent set of properties is generality. The properties in CAMPUS are general polymer tests, which apply to nearly all polymer material.

"Materials descriptions from different sources are described in such a way that the individual attributes they contains are generally useful. This is done by attempting to make a material property as independent of a particular product or application as possible and is reflected in the test conditions used to determine that property. The description of a material required for an artefact on the drawing board is, however, asserted in terms of the functionality of that artefact **not** in generally useful terms. So, my plastic box of computer disks must not break when dropped onto a hard floor: this is not the way a general purpose test is formulated."[49]

Divorcing tests from any specific application means properties characterising atypical attributes of polymers are absent and unusual extremes in application geometry, processing or environment are not represented but these can be of particular importance to a design.

Indeed, the CAMPUS properties, although very general in that the test can apply to most material, are highly specific in their "relative description" in order to enable proper comparison between the "relative values". They are therefore not abstract descriptions of design. It is difficult to describe purpose-related MDS in terms of these properties, yet to compare with the generally described MDs within a general query requires this compromise.

French values the contribution of abstraction in design, but has this to say about generality:

"More abstract does not always mean more general. If we want to design an elastic beam, the highly abstract but very specialised view of a beam as two flanges and a web, the flanges taking all the moment and the web all the shear, is immeasurably more useful than the very general theory of elasticity. The key to ... the cruder concept here is its greater abstraction (only three areas and a depth) and its *purpose-related* nature."[9]

This quote ignores the design step that occurs before the elastic beam specialises the solution, which identifies the general theory of elasticity as a solution to the design problem. The theory of elasticity generalises the specialised behaviour of the beam. Initially, the designer must identify the general theory of elasticity as a general solution to the problem then infer, from the beam's association with elasticity, the specific beam solution. The details of the beams behaviour distinguish it from other solutions associated with the general theory. French's point is that design benefits from abstract solutions but these solutions may not be general. Specific use-related abstract solutions are more useful.

Why the specialised view of the beam is a greater abstraction is not so clear. The view of the beam is a geometric abstraction. The general theory abstracts over all geometry by applying finite analysis. By the reference to four geometric variables, French might be assuming a specific geometry with a simple solution, rather than the variables needed to solve an arbitrary geometry using finite analyse. The distinction between these cases is an example of generalisation in the geometric perspective of the design.

The beam solution is more useful because it specialises the geometry. Progress towards specific instances in any perspective is useful to design. Material properties that suggest a specialised class in another perspective will appear more useful when that perspective is a free variable in the design. If it is not a free variable then the design needs general properties, which have a more complex relationship with other perspectives.

Zucker[1] observed the domain expert disregards whole categories of properties, "those properties that were strictly electrical or load-bearing properties, none had any significance to the selection of kitchen containers". These categories relate to very general purposes. A design specification often addresses only a small number of relevant properties out of the many properties that describe materials. The designer makes a decision collectively on the categories of property, regarding its relevance to the design problem. Designers base these decisions on their physical knowledge of the categories and not from any explicit information about the properties. In the case of beam design, the category of elastic properties in the materials perspective and the cross section of the beam geometry perspective describe the application of the beam. The overall application perspective specialises each design perspective.

Specialising each design perspective depends on the level of detail in the application description. Matching general properties between MDS and MD identifies the general characteristics of materials, or selects or rejects whole categories of materials. As the classes of material become more specific, the problem of distinguishing materials requires properties that will match the level of detail in the application. Consequently, properties of more specific material classes depend more on the context of the other perspectives.

2.1.2 Logical Abduction of Properties in Classification

Inferring the behaviour of materials from the general behaviour in a class is abduction. Abduction [26] is synthetic reasoning in science, engineering, design and even in everyday life, which forms and accepts explanatory hypothesis that accounts for a set of facts. If a material behaves like a polymer (the set of facts) then the hypothesis forms that the material is a "member of the class of polymers". The hypothesis is useful for explaining material's behaviour. If a material is known to be a polymer, then abduction infers the behaviour of the material from the behaviour of polymers. Consequently, the behaviour of polymers is very general in order for the behaviour to apply to many members.

Abduction declares a concept, the class, which accounts for a set of facts and is a repository of general knowledge. Often facts are deduced from observing the members. Statistical facts, such as the minimum and maximum values of a numerical property, can contribute to the description of the class. The designer deduces the class relevance to design problems from the class description. A design for a furnace, for example, exposing the material to a temperature of 500 degrees Celsius can immediately disregard polymers if the melt temperature is always less than 500 degrees.

The properties in CAMPUS characterise members who, as a group, share a concept of similarity; they are engineering polymers. By selecting properties common to all polymers, CAMPUS identifies what is similar across all polymers in terms of those properties. This selection of properties can define a membership function. The function selectively defines the concept 'Polymer' and describes a class of 'Polymer materials'. The process of identifying similar characteristics and then the subsequent use of those characteristics as properties for grouping members, is classification.

The power of classification for inferring properties is well discussed by Fahlman[27]. If the designer knows some fact about the Nylon class of polymers, that excludes the material from the design, then this immediately excludes any material known as a Nylon. This inference by abduction occurs without referring to any physical properties of specific materials. The general concept of Nylon infers the fact in question upon the specific material. A search through all known instances of material is no longer required; instead, an on mass test applies to classes.

There is much more debate on the benefits and pit-falls of classification. Ackerman[28], and Smith and Medin[29] both analysis a more complete philosophical definition of classification and concepts. The benefits of classification to design are enough to justify representation in a materials database. The computer representation of classification needs a formal description.

Many software systems already represent classification, and it is a feature of all class–instance object–orientated languages. All build on a simple formal model of sets. Even this model uncovers some pit-falls of classification, which manifest themselves as conflicts in the representations. This model also characterises the limitations existing in contemporary computer data-models, and therefore the limitations of the proposed database system.

2.1.3 A Simple Formal Classification Model

Taxonomy is classification that refines each class into subsequently more specific "levels" of classes. The mathematical abstraction of the poset, a partial ordered set[30], models the relationship between classes in a taxonomic classification system. Category theory* is a more complex model of the relationship between classes and their properties. Morphisms, the formal description of properties in category theory and properties describing classes in computer languages differ significantly. The latter are much more expressive and do not obey a formal logic. The simpler poset model applies to a category in category theory if restricting

* The use here of category is not strictly consistent with mathematical semantics of category theory. The theory defines the descriptive functions (Morphisms) as transformations between valid members. This is more akin to object-oriented classes transforming the state of instances. Here the category is like a set but instead of the descriptive functions defining membership, they only specify membership.

morphines to membership functions. The membership functions of sets suffice to model the relationship between classes and their properties. To clarify the description between classes and their members, a member is a set of properties, a property-set. A class is a category of members. Therefore, a category is a set of property-sets that satisfy a membership function.

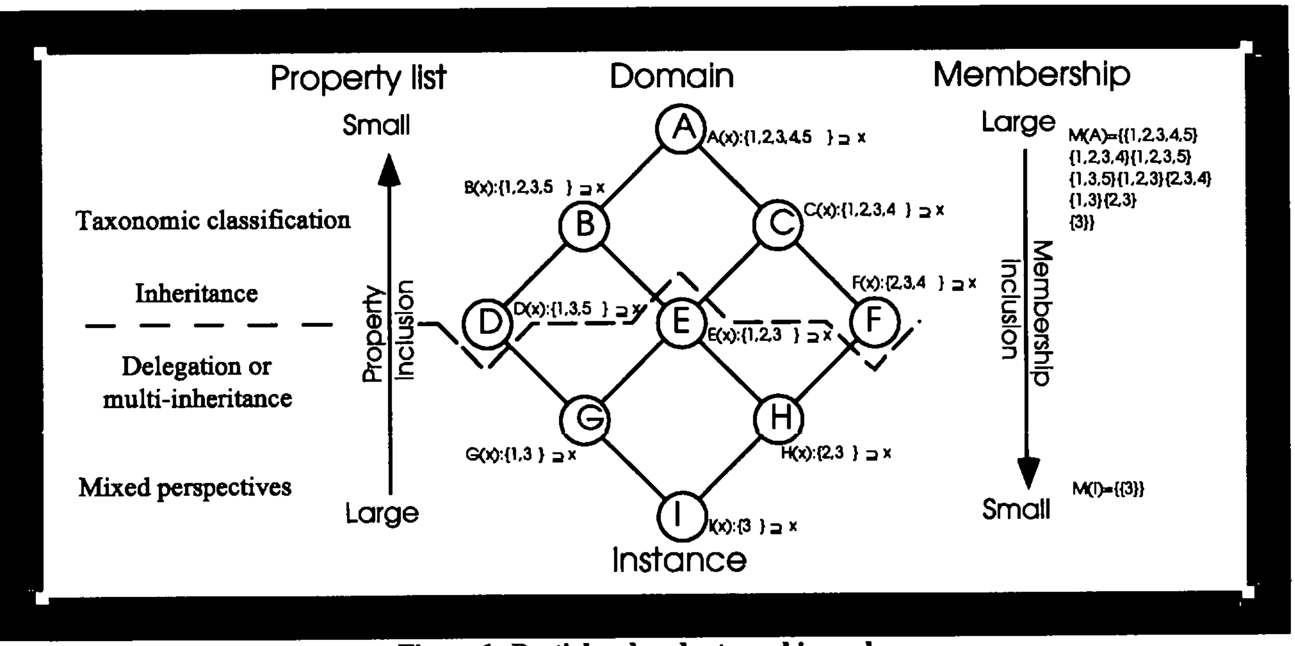


Figure 1: Partial ordered set as a hierarchy

Sets and categories related by inclusion can form posets. Inclusion[30] relates any set to its subsets and relates categories in taxonomy. The hierarchy in Figure 1 describes a classification of sets categorised by the inclusion relationship. The letters from **A** to **I** are categories. The categories' members are sets. The function **M** returns the membership, which for category **A** is all sets in the domain. Consider each set a material with a number of elements (the properties) as descriptions (eg, material {1,2,3} has property 1 and 2 and 3). The functions A(x) through I(x) specialise, or assert, the membership for each category (eg A({1,2,3}) is true since {1,2,3} is a subset of {1,2,3,4,5}). The properties inherit, so category **B** is subject to function A(x) and B(x). **B** is said to *subsume* the properties of **A**, by the process of subsumption. Category **I** is subject to all the functions. Of all the members in the domain, only {3} satisfies all the functions and belongs to category **I**.

The top portion of the graph in Figure 1 is taxonomic classification because the membership of lower (specific) categories are exclusively members of one higher (general) category. For example, the members of **D** are exclusively the members of **B** too, whereas the members of **E** are members of both **B** and **C**. A consequence of categories **B** and **C** not being mutually exclusive. Strictly, the exclusive categories are a requirement of a poset. **E** is said to mix the perspectives of **B** and **C**. If classification decides the subsets of {1,2,3} belong exclusively in category **B** or in **C**, as in Figure 2, it becomes taxonomic.

Touretzky[31] points out it is generally faster to search an ordered tree (a poset) than an unordered list. The efficiency depends on the organisation. "Often we will have more than one retrieval task in mind, with each task requiring a different organisation of the hierarchy". A search is efficient if the categories' assertions conclude which branches should be subsequently searched. A search becomes less efficient if the assertions are independent of the search criterion.

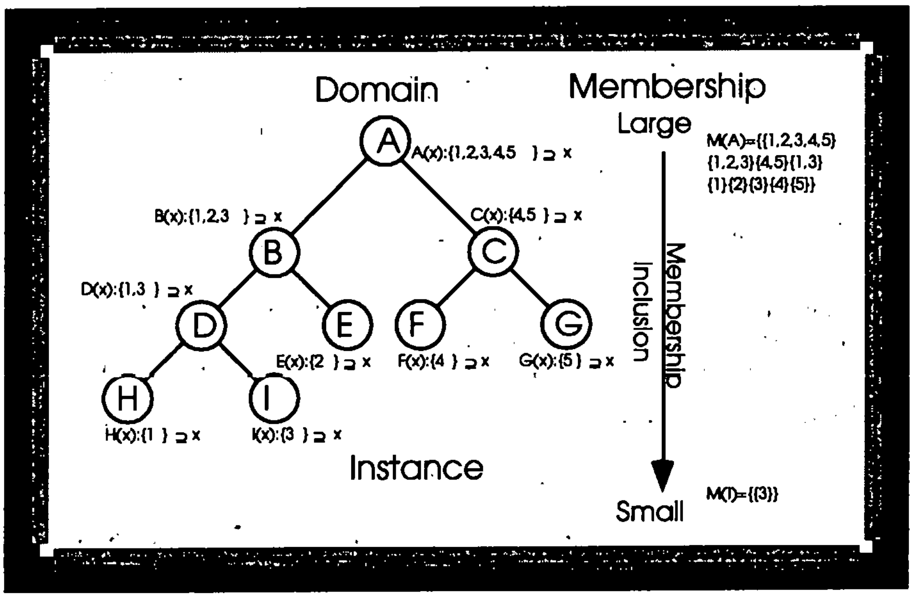


Figure 2: Taxonomic classification

For example, compare searches in Figure 1 for the following subsets: {3} {1,3,5} and {1,2,3,4,5}. The set {3} would require a complete search of the whole tree, through every node, until found in node **I**. The classification does not factor on the property of element 3. Whereas {1,3,5} is exclusively in categories **A B** or **D**. The set {1,2,3,4,5} is neither a member of **B** or **C** so the search can exclude the rest of the tree in two decisions.

For a materials classification, there is more than one retrieval task in mind, since many designs will use the classification. There is not one MDS, but many. At each level of the classification categories identify characteristics which clearly distinguish the categories. Each category must also characterise properties useful for matching MDS, thereby conclusively narrow the search for a suitable candidate material. The categories for one MDS may not be useful for another MDS. In practice, optimising the classification for all MDS is not possible, but attempts are made to make the classification **appropriate** for engineering design problems.

2.1.4 Appropriate Classifications

What makes an appropriate classification of materials for engineering design? Classification is a process of identifying "similar" characteristics, where similar at this point is an arbitrary common concept. Classification then uses those characteristics as properties for categorising members. In one sense, an appropriate classification discerns similar characteristics that ultimately result in assertions useful for searching. Although there are many retrieval tasks in mind, often the tasks themselves have similarity. One approach identifies similar features in MDSs. MDs then group depending on the MDSs they best match. This approach has been favoured by some[32] as it produces 'useful' application specific assertions. There are a number of problems with this approach:

1) MDSs do not classify in an exclusive manner. A MD can satisfy more than one MDS.
2) MDSs do not classify across the whole domain. New MDs might not satisfy any of the MDSs.
3) New MDSs can be defined that are not similar with any of the existing categories, thus needing ad-hoc classification to introduce a new category of MDs, which satisfy the new MDS.

The last point is relevant to a comment of Zucker's:

> "'Similarity' is known to philosophy as something of a snare and a delusion and we suggest that it cannot be used to group descriptions on an ad hoc basis — it is the context provided by the properties of the artefact which constrains the pattern of similarity in a selection system."[49]

In addition to ad hoc classification, Zucker says the purpose of the MDS, the properties of the artefact, constrains what is similar in materials. Representing similarity between MDS and MDs explicitly in the structure of the classification system is inappropriate. The act of classification is not an act in satisfying design criterion.

2.1.5 Conceptual Schema of the Cambridge Materials Selector (CMS)

Most software systems catering for polymer information have not developed advanced representations. They represent materials as a list of property values and focus on developing inference mechanisms that select materials using some satisfaction criterion in some way relating to a design specification. One exception is Ashby's[16] Cambridge Materials Selector (CMS)[33], which represents generalised data and focuses on presenting the information. The CMS demonstrates the effectiveness of generalised materials information at the initial stages of design. The CMS does not relate the general data to specific data on grades, which limits the CMS to the initial stages of design.

CMS rapidly accesses to a wide range of data at low precision, which supports the preliminary selection in design. The CMS diverge from selecting individual materials, with its precise and narrowly focused data. Instead, CMS provides a relevant level of information to questions raised in the initial stages of design, so answers with broad categories of material, with low precision.

> "The nature of the data needed in the early stages [of design] differs greatly in its level of precision and breadth from that needed later on"[33].

Whereas a specific material expresses a precise value for each engineering property, a category of materials can express a range of values in the category. The range is of low precision, but it expresses a broader scope of materials than the precise value of the specific material. The property profile of such a category reflects the property profile of the members in a broad and loose manner. In this way, the category generalises over the members it contains, and characterises an **abstract** material.

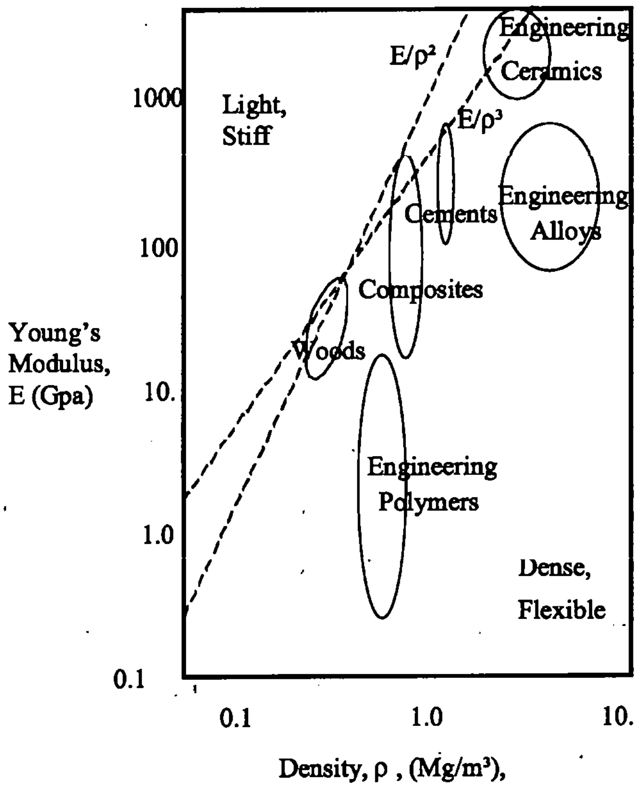


Figure 3: A sketch of an Ashby chart[33]

The CMS factors the materials domain into broad categories of materials, based on conventional material classes. These classes are founded on material principles of common chemical and structural composition, and are familiar to domain experts. They group similar properties, similar processing routes and often similar applications. They produce useful abstractions because generalising clusters the properties of the members across many types of properties. These clusters then differentiate the different abstract materials. This is important if the abstraction is to be useful in the selection process.

At the initial stages, it is more appropriate to answer design questions generally with an abstraction than with a specific material. CMS achieves this goal by visualising the abstract material families through a graphical user interface (GUI), Figure 3.

The GUI enables the user to plot two dimensions of the selection space. The CMS supports the process of evolutionary design through this GUI by allowing a progressive refinement of the selection space. The complete selection space in the database is multi-dimensional, each dimension a property distributing the domain of materials. The GUI displays any plane in this multi-dimensional space by choosing two functions of the material properties, typically factors from merit indices, defining a surface for graphing the abstract materials of the domain. An ellipse on the graph represents each abstract material family, mapping the extent of property variance within the family. The user can then mark an area of the graph to select or reject materials. Then the user can change the dimensions, graphing the selected materials against different properties. In this manner, the designer reduces the selection space from the whole domain of materials to a select few smaller categories of materials.

The information provided with CMS, the classes and generalisations, is the result of professional expertise and data analysis done elsewhere. The CMS does not support any database functions other than retrieval of this pre-defined information. The CMS is not a database system nor supports the process of classification. CMS only describes a hierarchy of abstract material based on classes of a classification system.

The main drawback of materials representation in the CMS is it can not support specialised property descriptions. The properties must be general to all materials in order to position each class in the selection space. Even some 'general' properties in the CMS have semantic differences. This could cause some error in the homogenous selection space. For example, the definition of yield point for metals and plastic is different, yet both share the same selection space. The precision of the general values CMS represents often permits minor differences in the relative descriptions of the properties within the same selection space.

2.1.6 Managing Property Pedigree and Test Data

CAMPUS, Pascams–220, EPOS and CMS all describe materials with highly general properties. They do not describe the purpose-related properties that fit design descriptions better. This is largely because these properties do not apply across all the materials they represent. A database capable of specialising the representation for smaller classes of material would be capable of representing properties that are more specific. An appropriate classification even needs these properties to distinguish the more specific classes of material. In the extreme, new properties will describe specialised classes of material. These new properties define new tests, and with this comes altogether new problems for database management.

Empowering the user to extend the descriptions of materials requires database management of both the grades and their descriptions. The M/Vision[34] system supports management for grades and their descriptions but does not support abstraction. Nor is it object-orientated, so the descriptions are just named values without any computational power. M/Vision though does demonstrate the complexities with managing even "simple" property descriptions.

Conditions and procedures qualify each test designed to quantify the physical properties of the material. The conditions normally include the environment, physical geometry and manufacturing technique of the tested sample. The procedure includes the technique and physical description of the test apparatus. Tests are standardised so the conditions are the same, and hence provide some consistency when comparing relative descriptions of different materials. Ashby[33] also emphasises the consistency of testing.

In order to test consistently, the test must identify all factors that affect the test result. Sometimes the factors affecting the result are not known. Lee[35] illustrates this in a case where a hydrogen environment, in the design for a rocket fuel pump, adversely affected the embrittlement of nickel-based superalloys. The relationship was not known at the conceptual stages of the design. Later, after the relationship was discovered, a new test was needed.

A poor test description can result in a poor property. Results of the new hydrogen embrittlement test were found to differ from different labs. Slight difference in test specimen microstructure resulting from different methods of making test samples caused different results. The test was subsequently modified. Lee coins the use of 'pedigree' to describe well-defined and understood property descriptions. Initially the property was not well understood. Discovering the new relationship with hydrogen and then the further refinement of the test description to standardise the microstructure of samples improved the test pedigree.

Quality management of test information has particular demands on materials information management, especially when acquiring data. Lee used the database management system M/Vision. M/Vision has multiple databanks. The test data enters into one data bank then passes through a spreadsheet that filters the test data ensuring the data meets the necessary pedigree, before entering a "materials" databank. For each design, another spreadsheet selects those properties appropriate to the design, creating the "materials design allowable" databank. For example, the materials design allowable databank could exclude data from an embritalment property that does not take into account of the hydrogen environment.

M/Vision is a database system with the purpose to store and disseminate consistent materials data. M/Vision's idea of a material is the tested substance, not the abstractions of the CMS. Data are one of an extensive, but limited range of data types, eg numbers through to graphs. M/Vision does qualify the relationship between material and data with a description of the test and the quality of the result. It supports categorisation by relating materials to a named category, but does not infer inheritance or represent abstract materials like CMS. Unlike most database systems, M/Vision can extend the description of materials to include new types of properties.

2.1.7 *Capricious Properties*

Sargent clearly identifies a difficulty in classification caused by properties known to be capricious.

> "The [Ashby] diagrams are most useful for selection at the conceptual stage of design because of the reliance on complete data being available for every property, for every material. The sparseness of real data implies that data from several closely related materials can be, and must be, merged as a material class to get a complete set. This implies that the method only works for those properties for which it is easy to identify classes of materials with similar property values. This is true for the properties, such as stiffness and thermal expansion, but largely false for properties such as corrosion or wear-resistance. These [are] capricious properties" [19].

Sparse data benefits from abduction, but the classes must group materials with similar property values. Classes may group the values of some properties, but capricious properties do not group. If the designer browses materials of a class, most properties will have similar values but the capricious properties appear random. Capricious properties are a problem for classification because the same kinds of materials do not have the same kind of properties.

In Lee's[35] case there was difficulty encountered in establishing a pedigree test for hydrogen environment embrittlement. Slight changes in microstructure were reported to have significant differences in property values from the different labs. Slight changes in microstructure between similar materials would have a significant effect on the property. Sargent calls this trait the capricious nature of some material properties.

Capricious properties describe a process that occurs during the test. An illustrative example is the process of crack propagation that must occur during any (successful) impact test. The propagation of the crack is subject to microstructure as much as the chemistry of the material. A slight change in the microstructure can cause very significant changes in the development of a crack and its subsequent propagation. Therefore, the test result relates more to the specific structure than the material of the sample. Some changes in microstructure may relate to processing in an unpredictable way, resulting in a chaotic variation in property results. Such properties will never lead to a property of high pedigree since the description of the test can never qualify the microstructure to a detail necessary for a repeatable result that depends solely on material composition.

In other properties, the slight differences in chemical composition between materials can result in radical changes of property value. Matching capricious properties is difficult at the initial stages of design, and best left for detailed changes in composition and processing. Therefore, despite the property not distinguishing a class from any other, the range of a capricious property may still be of interest.

Properties should encourage "incremental stability" — slight changes to the relative description result in acceptable changes in the relative value, otherwise confidence in the selection process will be lost[19]. The stability of a property is useful knowledge for determining the comparability of different relative descriptions, which can quantify the confidence a designer can have in a value comparison.

2.1.8 Classification of Polymer Information

Classification is an important structuring component of a KBS for design. In addition to researching software components for representing and managing classifications, the user requires interface tools in order to build the classification and then use the information it contains. After developing the software components and the tools, there remained the issue of how taxonomy of materials should classify. A separate study by Spedding[8] used the tools this present research developed (see Chapter 5) to determine an "appropriate" taxonomic classification of polymers for engineering design. Spedding provides the high level (human cognitive) judgement of similarity to develop an appropriate classification for design.

After populating the KBS by importing data from the CAMPUS database, Spedding used the KBS to evolve the classification by defining higher level classes of polymer. In addition, she extended the description by defining new properties and rules for those properties, and adding them to the classification. She also generated descriptions of new polymers and performed a number of data analysis on properties of the abstract polymers inferred from the classes.

Spedding classifies polymers by characteristics of chemical structure. The characteristics of simplified chemical structure satisfy Simon's[24] criterion for forming good hierarchies. A hierarchy needs to compose of identifiable sub-systems and the interactions, or properties, between members of the same sub-system should strongly correlate or identify stronger in magnitude than with members from different sub-systems. A simplified view of chemical structure composed multiple levels of sub-systems, namely atoms, molecules and grains. The interactions include spatial distribution and attractive forces. Among sub-systems of the same level the magnitude of the interactions, say between atoms within molecule, is similar. The interactions differ by an order of scalar magnitude when crossing different levels of sub-systems, say the atoms between different molecules. The difference in the attractive forces defines a molecule, so a hierarchy of chemical structure is based on principles of chemical science[19].

Chemical structure has far-reaching effect on a wide range of properties. Grouping grades by similar chemical structure affect the properties in the same way. The groups collect like with like grades as members of a class. Since the chemical structure is the basis of standard nomenclature, the generalisations from these classifications are also familiar.

Through Spedding's work it transpires that the characterisation of a polymer as a class according to chemical structure is, on its own, insufficient to fully differentiate types of polymer grades. For example, the addition of glass reinforcement has strong affects on some properties but not on others and is independent of chemical structure. These were properties where the mechanism of the internal process leading to the property was more dependent on the reinforcement than the material class.

In some cases general polymer properties could not predict the performance when the property was in an extreme geometric state, eg films and fibres. To the observer, the extreme geometry generates sufficient capriciousness in properties to cause the prescription of a new test. These tests though are only valid to materials capable of the geometric state, and not applicable to materials in general, yet chemistry does not exclude the property. The applicability of the properties is orthogonal to the materials classification.

A conflict was identified between the classification and the need to represent these properties on a per-instance basis. Conventional class–instance languages can extend the descriptions of grades by creating subclasses or by using multiple inheritance mechanisms to subsume orthogonal properties. In multiple inheritance a class might have two parents, for example, one contributing general properties of `Nylon` and the other adding the film properties to give a subclass `Nylon-Films`. The function of this multiply inherited subclass is no different to an explicit subclassing of `Nylon` with a subclass `Nylon-Films`, `Nylon` being the single parent. Although both are computable solutions, for knowledge representation of engineering properties of polymers both of these mechanisms are flawed[36] hindering the extensibility of the classification.

2.2 Knowledge Representation

Designers requires a taxonomic classification hierarchy of polymer materials knowledge that generalises the knowledge within the classes and then infers an abstract material useful for initial design, in addition to representing the individual properties of the specific materials. Frost[37] gives an excellent background covering general knowledge representation. This section of the review specifically examines the conceptual model of knowledge representation that addresses taxonomical hierarchies and the inference logic within them. This examination starts with defining some of the underlying concepts, before looking at work on the frame-based systems that introduced early taxonomic hierarchies with inheritance, and the problems they encountered.

2.2.1 Knowledge vs. Data

"Most knowledge bases are distinct from conventional databases in that they typically consist of *explicitly* states general rules as well as explicitly stated simple facts."[37].

A database only describes simple facts, such as tuples in a relational database, with implicit data modelling rules such as "tuples are unique in a relation". A knowledge base explicitly stores rules, such as "All `Nylons` are `Polymers`". Just as the computational functions calculate (eg sum and multiply) simple values in a well-defined manner, more complex computational processes manipulate the rules, according to a 'formal language', to **infer** new facts.

A number of formal computing languages logically process rules. They are formal because rules exist for the construction of legal expressions where the meaning of the expressions can be derived from the meaning of the components of those expressions. A formal language with axioms (standard rule of inference) that can deduce if rules in a theory (set of assertions as sentences of the language) are consistent is known as a formal deduction system. Deduction is a form of inference that infers a cause (like consistency) from a number of effects (the rules).

Most inference applies to a known set of related rules. A formal deduction system that groups rules need only check consistency within each set of rules, thereby reducing the computational load. Adding new knowledge is simpler since only the local effect of new rules would need consideration. Attempts have been made to structure the knowledge in formal languages. Both simple facts and rules were initially shown graphically structured in semantic nets[38,39]. The graph in a semantic net allows meaningful groups of rules about a common entity. "Slot and filler" representation is another approach to structuring rules in to entities, which frame-base systems are an example[40].

The knowledge structure in frame based systems complicates the axioms of the formal languages on which they were originally based. Additional axioms define the rules for inference between entities. One of these axioms models inheritance of rules between entities. The frame-based system NETL[27,41] is an early working example that demonstrated inheritance for knowledge representation. As will be shown, inheritance has lead to ambiguity and inconsistencies in these KBSs. These findings are relevant since frame based systems introduce a number of features that closely resemble object–oriented systems.

2.2.2 Frames

A frame structures data that represents an entity—a concept or thing being described. A frame consists of a collection of named slots. Values or 'pointers', which link to other frames, fill each slot. Copying frames creates a new frame of the same type, in a process called instantiation. In this, the frame is similar to a relation defining the structure of tuples (see §2.3.2), but different frames with the same state are possible and identity is not dictated by the values in the frame's slots.

Various kinds of deductive inference are supported through frame 'matching'. Frames were first developed for pattern matching, eg visual identification of an entity from observed properties, and understanding of analogies in text. The frame structure also supports deduction of consistency since local modifications to an entity only affect rules about the entity, and not the rest of the KBS. Matching MDS with MDs fits this category of inference.

Frame base systems also include a number of implicit rules to simplify knowledge representation. The rules include generic properties, default values, taxonomic structuring. Explicit rules are supported by slot conditions and procedural attachments (or 'demons').

Generic properties include universal rules, such as "All polyethylene are constructed from the monomer ethylene". A generic property is a specific property all instances of the frame must exhibit.

Default properties are similar to generic properties but may be over-ruled by instances. The default "All `Nylon66` have a melting point of 270°C" is copied by all grades of `Nylon66` but may be changed by individual `Nylon66` grades.

Slot conditions are explicit rules whose consistency depends on the state of the slot. The rule: "Material impact strength is a number greater than zero or No-Break" is such an example.

A procedural attachment is a mechanism for evoking a computing process upon change to a slot. For each frame based language the functionality can be different, but it is generally expected to aid the structuring of a knowledge base. For example, a component's material type might determine the production type for the component, so when the component's slot for material is filled, a type of production frame is instantiated, say injection moulder, and entered into the component's production slot.

2.2.3 Inheritance Hierarchies in Knowledge Base Systems

Frame base systems generalise common slots through hierarchical structuring. Rather than define the slots explicitly for each frame, a generalised frame, or 'parent', can define the common slots and the 'child' frames can *inherit* the slots through a special `is-a` relationship. All the slots in the parent, along with their generic properties, default properties, conditions and procedural attachments, are implicitly slots of the children by the mechanism of inheritance. For example, the child `Nylon` inherits the slot of impact strength from the parent `Polymer` through the "`Nylon is-a Polymer`" relation. These `is-a` relationships form a generalisations hierarchy of frames.

Hierarchies have long been seen as an important structure in knowledge representation. The hierarchy relates specific entities with more general entities by the `is-a` relationship. Inferring the behaviour of the specific entities from the general entities is inheritance. Although there are often some behaviours inherited that are abstract truths, typically they are only generalisations where exceptions are expected. The added complication of resolving exceptions clearly identifies that generality hierarchies and inheritance are not one and the same:

"ISA isn't inheritance and inheritance isn't ISA"[42]

For example, "*Clyde* is-a Elephant" is a classification[42]. Rarely is there a problem with the explicit statement that does not contribute any more properties than *Clyde's* membership to Elephant. When assuming *Clyde* has large ears, a property logical abduction infers from an abstract Elephant, there is a potential for inconsistencies and ambiguity.

A classification hierarchy describes the relationship of generalisation, and inheritance is only a mechanism for enforcing the principle of ***subsumption*** across that relationship (see 2.1.3). Subsumption occurs when one concept, say Polymer, collectively describes the properties of another, in this case Nylon, Polycarbonate, Polyethylene etc. Every property that defines Polymer also defines those subsuming Polymer. General rules for subsumption are still under debate. In particular, defining properties and describing properties are distinguished[43]. Some properties of Polymer do not define Polymer but are only descriptive; they are typical and used as default properties that are still inheritable. As they are only typical, there is cause to define contradictory properties in a subsumed concept, ie define an exception. "Crystalline polymers are not usually transparent. However, PET used in soda bottles, has such small crystallites due to processing conditions that it is transparent."

Some inheritance schemes allow for exceptions. Fox argues mandatory inheritance of properties is too inflexible for representing real-world knowledge[44]. Interpreting exceptions logically is complex. Standard first-order-predicate logic can not represent exceptions since this logic sees an exception as a contradiction with the inheritance rules. The more difficult nonmonotonic logic[45] provides a semantic that can model exceptions.

Exceptions can lead to poor modelling. They can over-ride all inheritance, leading to ludicrous statements that contradict the very purpose of the classification hierarchy[46]. Horty provides an alternative, by suggesting a mixing strict logical inheritance, which does not allow for exceptions, with a defeasible logical inheritance corresponding to a statement of expectation, "Birds should fly"[47].

Difficulties occur when a concept subsumes more than one other concept. Consider a material blown into a film. In many contexts film plastics are considered a raw material. A film-plastic subsumes both the concept of plastic and the concept of a film. In a hierarchical knowledge-representation, such as a semantic net, the film-plastic would be given both properties: `is-a` film and `is-a` plastic creating an a-cyclic graph. This is still considered a hierarchy (but not taxonomic) since a generalisation ordering is maintained. The difficulty is to resolve the subsumption of properties from both parents.

2.2.4 Problems with Inheritance in Hierarchical Representations

If multiple parents in a multiple-inheritance hierarchy are truly orthogonal then the properties of one parent is independent of the properties of the other parent. If the parents are not completely orthogonal, properties of one parent may conflict with the properties of the other.

Conflicting properties are either descriptive or definitions. If definitions, then conflicts should rule the subsumption invalid, eg a material can not be both `is-a` plastic and `is-a` metal. If properties are descriptive, then exceptions are possible and the conflicting assertions requires resolving. Resolving these issues is the task of the inheritance mechanism.

Semantics (the descriptive rules) for multiple-inheritance with exceptions were first proposed by Touretzky[31]. Earlier techniques for resolving inheritance in the system NETL[41,27], and many other knowledge representation systems, were based on a simple shortest path calculation. The shortest path algorithm assumes each link between child and parent has a unity weighting of specialisation, reflecting the strength of a parent's assertions. Shortest path algorithms can lead to unexpected results. By adding redundant statements, the properties of entities can change. For example, if "*Clyde* `isa Royal_elephant isa From_India isa Elephant`" and `From_India` has the property 'ears = small' conflicting with `Elephant's` 'ears = big', and then an extra redundant statement "*Clyde* `isa Elephant`", which changes the distance of 'ears = big' from three parents distant to one parent, would change the conclusion of the shortest path algorithm from 'ears = small' to 'ears = big'. Touretzky defined his inferential distance ordering to preclude inheritance along sequences if contrary intermediate sequences exist, ie precludes the inheritance along *Clyde* `isa Elephant`, while already inheriting along `From_India`.

Regardless of the system for determining the assertions from multiple parents, multiple inheritance with exceptions will always be bound by nonmonotonic logic[45], ie more than one logical solution can result creating ambiguity. Simply put, if film and plastic are equally strong parents and both are descriptors of property strength, which property should dominate as the property of film-plastic? Unless explicitly stated, there is no way to resolve the description of film-plastics into a single solution.

Touretzky's inheritance is more orthodox, formally describes its semantics and, more importantly, defines when ambiguity occurs. The implementation comes with the cost of a more complex algorithm for resolving inheritance.

Terminological logic studies hierarchical representations of knowledge. It is primarily concerned with generalisation by subsumption, which in turn has a strong inheritance flavour. The work has shown that inheritance has many representational problems. One suggestion for handling ambiguity, discards inheritance as an implicit mechanism and instead supports the inference of subsumption directly, by generating hierarchies with explicitly define inheritance[48]. Patel-Schneider points out two other problems with inheritance:

Expressive problems: Recognising the most suitable location on a hierarchy to express a specific instance is not a function of a hierarchy. The hierarchy does not prevent a specific entity from inheriting from a general description, and specifically defining properties that are also described in a more suitable subclass of the general description. The hierarchy does not enforce the recognition of similarity. This recognition is up to the user of the hierarchy[46].

Deductive problems: Inheritance, generally, do not address the combination of inherited properties. If the logical combination of two properties produces a third, then either inheriting or defining the two should result in the single third property: the two components should not be further inherited. Combining properties is necessary for the function of subsumption, eg subsuming $P(x):\{1,2,3\}\supseteq x$ and $P(x):\{2,3,4\}\supseteq x$ should give $P(x):\{2,3\}\supseteq x$.

2.3 Data Modelling

A data model is an abstract structure for containing data. One way of interpreting the data model is as a set of rules for combining data. These rules limit the expression of data, so the choice of data model needs to consider the purpose of the data. The rules limit expression because they are closely linked to the sequential way computers represent and access data in memory. This link between data model and memory is the physical model.

For the majority of computerised systems managing data, Data Base Management System (DBMS), the priority is on quickly processing large quantities of data from a storage system with slow (arbitrary or 'random') access. Consequently, these conventional DBMS limit the capabilities of data manipulation in the physical model, to simple access routes and data manipulations, which utilise the access routes. The data models do not support arbitrary computation involving many different types of knowledge, in contrast to the computation possible in many computing languages.

Many systems for supporting material selection use one of a number of standard data-models. Examples of some common models are given later. If a data model is suitable, any number of 'off the shelf' DBMS can be selected, optimising development and performance. Knowledge representation also has standard data models, such as the production rule representation common in many 'off the shelf' expert system shells.

Demaid and Zucker[25,49] question the appropriateness of adopting any of the common representation strategies for the development of systems that support the evolving nature of design. Their assessment of the common representation strategies is relevant since one component in their schema represents materials. A summary of their argument for a conceptual schema on which they designed their own representation strategy leads to the use of classification as a design tool. In general, the effect a data model has on the application of the data is well documented elsewhere:

> "It is important when choosing a DBMS that the user is aware of the data model underlying it. This is because the user of a DBMS must perceive the universe of discourse according to the view of the universe which is the basis of the data model of that DBMS"[37].

Most data models in conventional DBMS do not provide a diversity of modelling constructs. This weakness makes them inappropriate for engineering design[50]. Design involves computation with many different types of knowledge, eg processing, geometry and materials are all broad categories. These different types of knowledge would benefit from semantically richer organisations (see §2.3.4). For example, organisations based on knowledge entities (represented by data structures) rather than the data structures (representing many entities).

2.3.1 Hierarchical and Network Data Models

The automated data processing of the 1960s and 70s represented data as simple character strings and numbers and structured this data into hierarchies and networks. The hierarchical structuring of data mapped well into a physical model of records sequentially stored in files recorded on sequential storage medium such as magnetic tape[51]. This organisation enables quick searching for particular parts of the structure and simplifies automatic processing because of the uniform file format.

The hierarchy places limits on the knowledge represented. Only one-to-many relational structure are possible, eg a kettle design, with a plastic container, with glass lid, with a plastic handle with…etc. If other kettle designs were made using the same handle, the whole handle would need to be copied. Both designs could not access the same entity relating many kettles to one handle. Many to one and many to many relationships need a network structure, such as in CODASYL systems[52]. To form a network the CODASYL system introduces pointers between records.

Both hierarchical and network data models view information as entities with attributes. Records, physical space on disk, represent entities, and the binary information in the records translates into attributes. In a hierarchical model, the ordering of records (on secondary storage) describes all one-to-many relations. In a CODASYL network the binary data in a record can also be interpreted as a pointer to a set of records (DBTG sets, sets defined by the Database Task Group[37]) to form a many to one relation. The DBTG set forms a one-to-many relation as an ordering of records, like in the hierarchical model.

2.3.2 The Relational Model

The relational data model is more common, and originates from Codd's work[53]. Relational databases (RDBMS) manage tables of data. Each column of the table contains entities from a particular entity set. Entities are unique identifiers such as strings and numbers from an entity set. Entity sets defines all valid entity identifiers. The database stores these identifiers in tuples. Each tuple is a unique combination of entities from the entity sets in the table. Whereas the entity sets are the columns in the table, the tuples form the rows. The relation defines the associations between entity sets, hence the possible tuples and the relationships between entities, which gives the database structure.

The CODASYL network model represents many-to-one relations differently to one to many, causing an asymmetric performance when accessing. This benefits one use of the database over another. In addition, pointers make it very difficult to manage the movement of records in memory. In contrast, the relational model is 'flat', with entities associated in an equally commutative relation; ie the 'columns' of the table can be swapped without effect. Each column relates equally to each other column. The tuple easily describes a many-to-many relation. The relational model does not use pointers, but uniquely identifiable attribute values in the entity sets.

The relational model is suitable for financial records for two reasons. First financial information requires only a few simple data structures. Secondly, the number of individual records 'instantiated' from each of these structures is huge. Many financial database activities manipulate relations, not individual entities, which act on this large population of records as a group. These computations are operations of either a relational algebra or relational calculus, languages that manipulate relational tables.

Access to individual tuples is possible through a transaction that selects the desired tuple from a relational table. Such a transaction is a sequential search and compare of all tuples in the relation. Faster access is possible by 'hash key' list. A hash key list is a special ordering of entities indexing the tuples of a relation. A 'hash algorithm' calculates the position in the list for a particular entity and with the entity the desired tuple. Although hash lists aid the access to tuples, they are not part of the relational model, but extensions by typical RDBMS to the physical model.

2.3.3 The Relational Model in Engineering

Maier questions the suitability of the relational model for CAD, computer aided design[54,55], in contrast with the object–oriented model which will be discussed later. Maier argues CAD systems define large numbers of types with fewer instances. Transactions tend to follow paths from one individual record to the next (eg from car to the car's door — attributes form paths to other behaviours).

Frost identifies the same problem in the relational model as a performance asymmetry[37]. Information on entities is often spread across many relations. Although the relational model is 'flat' within a relation, combining the information from different relations requires an algebraic operation, whereas information within a tuple do not. The information on a particular entity spans across relations as well as the relationships within tuples. The asymmetry creates a difference in access performance for different attributes of an entity.

According to Maier, CAD tends to traverse between tables. In a relational system, this traversal requires an attribute value look-up, optimised through a hash key. The hash key is another source of asymmetry. DBMS only index selected entity sets in a relation. Although other physical models locate entities through a hash table, such as some object–oriented models, there is no asymmetry if the table consistently includes all entity relations. Object–oriented models optimise their access to objects since it is a prominent activity in object transactions. In relational systems, any overhead associated with each transaction (eg fetching a look-up table from secondary memory) effects performance.

Maier concludes the overhead with each CAD transaction tends to be large in a relational data manipulation language. Data processing computations of typical RDBMS applications tend to apply few transactions so the overhead has little impact on performance. CAD computations tend to be more complex, and the overhead has a larger impact.

This performance difference is at an extreme as the relations describing an entity increase. In CAD applications entities use many different relations, not one large relation. One reason is the different types of entities have some different associations and some the same. This forms a type-subtype hierarchy relating the similarity. The relations that are the same are kept in one table. Those that are different are in different relations. Hence, many relations describe entities with many differences.

An example is a Polymer entity in a relational model. A polymer may be considered a type of entity that exhibits the relationships to the properties for tensile strength, elasticity, conductivity and many other generic materials properties. A table is formed where each polymer grade is a tuple describing values in the property columns. Here a single relation is used. However, a grade may express unique knowledge that is not generic to polymer but some more specific classification, eg the `Crystalline` polymer property of melting point. Therefore, a second relation is created to record the `Crystalline` polymer properties. The process continues. At the extreme, properties may be defined to distinguish individual grades creating a large number of relations. This demonstrates one problem the relational model has with describing infinitely extendible descriptions of entities. More relevant is the distribution of information across many relations. Relational DBMS are good at managing large relations, not a large number of relations, therefore only a few types of entity.

Finally, Maier also argues the strategies for concurrency (data sharing) and recovery protection, work well in small transactions over large data populations where locking and logging can be applied and optimised, but work poorly on CAD data. These features put a lot of overhead on transactions in multi-user and multi-tasking computing systems.

The relational model could represent a polymer grade as an entity in a table that groups a number of entity-sets, one for each property. Rules could be associated with the table to ensure the grades correct behaviour. Beyond this, the relational model does not assist the knowledge representation of polymers. Developing a classification of many types of tables and ensuring they correctly subsume the rules from each class while evolving the whole representation would require a complex interface for interpreting the data in the model. The preferred approach rejects the relational model for a data model that supports the structure of the classification, such as semantic data modelling.

2.3.4 Semantic Data Modelling

The relational model fails to capture the semantics of an entity; the meaning of an entity as an atomic concept characterised by properties. An entity in the relational model is often spread over many relations. To display all properties of an entity requires an operation locating all the relations that attribute properties to the entity. Within tuples attributes and entities are not distinguished, so data manipulating can produce meaningless relationships, such as between tensile strength and conductivity taken from a tuple describing a polymer entity. One-to-many relations differ from the many-to-many a tuple represents. The one-to-many should constrain queries, like the relational operation 'projection'. This, and other semantic issues, are addressed by 'Semantic Data Models' (SDMs). In particular, SDMs focus on a database as a collection of entities.

SDMs provide structural abstraction[56] (as opposed to object oriented behavioural abstraction which will be addressed later), driven by a need for data representation as opposed to data manipulation, resulting in more complex types of data structure. An early semantic model is the Entity-Relationship model[57], distinguishing entities and relations. The semantics of relations are specialised to identify aggregation from association. Semantic modelling extends to distinguish groupings[58] from associations and aggregations. Further developments add generalisation. For an overview see Peckham and Maryanski[59], or Gardarin and Valduriez[60].

Aggregations and associations are semantically similar. They both attribute properties to entities. The aggregation though is not viewed as a number of parts, but an atomic semantic unit describing the entity. The polymer grade is an aggregation of engineering property attributes. Forming new relations from parts of the aggregation (between engineering properties eg conductivity and tensile strength of a polymer) is meaningless.

Associations are access paths between entities. A material may define a property linking a material to successful applications. The relationship does not define the entity. The attributes of associations are entities themselves. The attributes in associations may be used to form other relations to other entities forming other associations.

A relational model can support both aggregation and associations but does not distinguish them as the SDM does. In the relational model, each tuple is set of values, some that aggregate attributes and others that associate with tuples in other tables. The unique identity of the tuple is a function of all values, whether contributing to the aggregation or association.

In the SDM an entity changes its identity if attributes of the aggregation changes. If the change causes all the values to equal those of another entity, then the model will only represent one entity; the two entities become one. This is different to the relational model, which will maintain two tuples with the same aggregations if there is a difference in the associations.

Unlike changes to aggregations, changes in association should not affect the identity of the entity. In the SDM if two entities describe the same aggregation they should reduce to one, but how this affects their different associations is not so clear. Technically the two entities are the same. King suggests it should trigger some process[56] to resolve the associations.

Classification is a specialisation of association (also called grouping[58]) here the members are all of the same type of entity. A type defines the properties of entities, both associations and properties, derived from aggregations. A classification groups entities which exhibit the same properties but in a SDM not necessarily all entities with those properties, eg Polyethylene used at Lucas is a classification.

The generalisation is a classification that groups together semantic similarities, eg In a classification of materials, "`Polymer`" is a generalisation which includes the property tensile strength, exhibited by all `Nylons`, `Polypropylene` and `Polycarbonate` classifications since they are all `Polymers`. Every type is a generalisation. In addition, property intersections of types may define the similarities between the types, hence more general generalisations.

2.4 Object–orientation: A Background of Principles

Object–orientation is a technique of abstraction. The technique supports software design, in particular Graphical User Interface (GUI) development and knowledge representation, but may also be useful for product design. This section looks into object–orientation in software languages with a view to modelling design descriptions of products. Object–orientation composes descriptions in a similar manner to the composition of product designs. Classification is also predominant in many object-orientated languages; its relevance to materials information management has already been mentioned. In object–orientation the main unit of abstraction is the object. The principle of the object to formulate software behaviour follows.

2.4.1 The Software Abstraction of the Object

Programming is a design problem in itself. The problem is to get a computer to behave in a specified manner. An application is a software construction combining abstract behaviours, creating one solution to the problem. If the abstract behaviours model some other design domain, then within the constraints of that model, the software solution is also a valid representation of a solution in that domain. The question is whether the abstract behaviours a computing language provides for the construction of programming problems could form suitable models in other design domains: Are objects a generic representation?

Computer languages define a closed set of atomic behaviours. Computing machines construct atomic behaviours from boolean logic[37]. Consequently, they are individually invariant, precise and predictable. These qualities make them suitable for modelling formal mathematical logic. The logic of sets, for example, provides mechanisms towards generalisation, specialisation and abstraction.

Sequences of behaviours form sentences in the computing language. Although individually the atomic behaviours are invariant, the atomic behaviours affect the state of the machine, which in turn changes the sequence of behaviours. This allows variation in the behaviour of software. These variations quickly become very complex which makes understanding the behaviour of software difficult.

Consider the task of drawing a line between two points on a matrix of points. The computing behaviour determines which points in the matrix are between the two given points. The given points are a state of the computing machine that affects the computing behaviour to draw different lines. Whereas the programming task producing the behaviour is complex for the computer, the concept of line drawing is simply understood. The behaviour is complex in design but simple in concept. The concept is simple because the behaviour is limited to the task of line drawing. The behaviour changes, if given different points, by drawing a different line but always draws a line and, for example, does not draw curves.

Without the given points, the behaviour is abstract. An abstract behaviour represents a known variation of behaviour. An abstract behaviour conforming to a simple concept, though complex in construction, may be reused in further software constructions, such as drawing polygons requires line drawing behaviour. Through abstraction, software increases complexity while each abstract component may remain reasonably simple in concept.

Program design is mainly an activity of decomposing the design into identifiable abstract concepts. The example of drawing a square decomposes into drawing lines, which decomposes into drawing points. Programming then describes the behaviours of abstract concepts. Often a design encounters the same abstract concept many times. Computer languages support abstraction by allowing the reuse of a programmed abstract behaviour.

Once an abstract concept is successfully programmed, it is desirable to reuse it where possible. Designs rarely start from scratch, and languages supporting reuse of software makes it easier for programmers to build from previous software design. Such support is not limited to developing software but may address generic design problems.

Support of abstraction by computer language comes in many levels. At the lower levels the languages strictly define the abstract behaviour and maintain tight control over behavioural variations. Each level higher provides different kinds of abstraction, gradually increasing the ways a software abstraction can describe complex abstract concepts.

In an object–oriented language there are additional mechanisms of abstraction. The subject is well covered in many texts[61]. The following summarises the reasoning behind the object–oriented concepts.

In object–oriented languages, in addition to combining abstract behaviours to form specific behaviours, the behaviours themselves are grouped to form objects. The analogy is that objects in the real world are identified by a collection of descriptions. The behaviours are the descriptions of an object. These descriptions are in a language that uses other abstract objects as components in sentences. A behaviour is no longer reducible to a single complex combination of atomic behaviours, but depends on relationships with other objects and their behaviours, which can change.

Object–orientation recognises that few objects have unique descriptions. Their descriptions share similarity to other objects. This recognition leads to (currently) one of two kinds of description sharing: prototyping and classification.

There are many other issues in object–orientation besides sharing descriptions. The following section will highlight some of them. Two features dominate the control of object manipulation. Object manipulation changes the state of the object. A state is a specialised behaviour that depends on the history of transactions. The two features controlling the manipulation of the state are encapsulation and messages. Messages are the transactions and the object's encapsulation ensures only proper messages manipulate the object.

2.4.2 Encapsulation

Encapsulation is defined as the grouping together of various properties associated with an identifiable entity in the system in a lexical and logical unit, ie the object.

What encapsulation achieves in terms of modelling and program-structure is its most important benefit. Encapsulation provides a boundary called an 'interface'. This interface defines where an object stops and the rest of the world begins. The encapsulation defines rules for passing that interface. The rules ensure that the state of the content results from historical accesses to the object, where each access abides by the rules.

In terms of modelling, the interface enforces the grouping of related properties that constitute an object. Access to the properties is subject to the rules of the interface, so the state is well controlled and processes outside the interface can not change the state inside the interface. Although an object can be defined without encapsulation, it demands discipline from a programmer not to directly access internal components of an object thereby intertwining the object's internal world with the external world.

Not all encapsulation is equally effective. Some languages are better at encapsulation than others. A good test of the 'strength' is to try and side effect (change the state) the properties of an object by breaking the rules of the interface. Usually a language has weak encapsulation for reasons of efficiency. So even in this most typical feature of object–orientation, there can be variations.

2.4.3 Messages

A messages is a basic means of behaviour sharing between objects. Sharing descriptions of behaviours, mentioned earlier, is not the same as sharing the actual behaviour. To distinguish the difference, a description of a behaviour, as written in the syntax of the language, is called the **protocol**. While programming protocols, they are often termed behaviours since, when executed, the protocols generate the behaviours. They are very similar and subtlety different. The aim of this next section is to describe the message and distinguish this difference between behaviour and protocol. Later, the relevance to modelling will be highlighted.

A distinguishing feature of object–oriented systems is the ability to 'pass messages between objects'. A message originates as part of a sentence in a protocol that describes an object. The message specifies another behaviour to evoke. The message identifies another object, known as the **receiver**, where the behaviour resides. Additional information identifies the particular behaviour in the receiver.

The message does not directly access the protocols in the receiving object. Messages are received at the object's interface. At the interface, the additional information in the message interprets what will happen. This is fully under the control of the receiver, not the object sending the message. A useful interface will define a known set of possible actions.

The first step in message interpretation is to locate a protocol to continue the computation. The rules used to interpret the message differ from language to language, and are a major source of difference between them. In some the rules are programmable. The interface though should remain consistent, well known and published since it forms a contract between the receiver and the protocols of message senders. If a receiver can not locate an appropriate protocol, either the language generates some kind of error, or the receiver may specify a specific default protocol for messages it does not understand.

After locating each protocol, the receiver evaluates it. The evaluation generates the behaviour. The evaluation is a process of further message sends. Protocols generate behaviour, which locates further protocols for generating more behaviour, infinitum. Ultimately the software evokes messages to atomic behaviours that generate behaviour without further message sends, terminating the chain reaction.

Each protocol is a specific combination of other abstract behaviours. Additional objects augment the message evoking the protocol. These "arguments" and the receiver together specialise the behaviour that the protocol generates.

It may have been implied that the receiver of a message is the *owner* of a protocol. This is not the case. Protocols are often shared. This does not necessarily mean the behaviour is the same for all receivers sharing the same protocol. Differences in the receivers specialise the behaviours by sending different messages from the same protocol syntax. The syntax composing a protocol changes semantically by altering any objects in each message eg the object receiving the message. Each object sharing the protocol provides a different context of available objects, the receivers and arguments, for binding to messages in the protocol and determines the path of computation when evoking. There can be many differences, each a different path of computation.

The different paths a protocol generates are descriptively called its 'pattern of message passing'. A complete object–orientation system of objects is a flowing 'pattern of passing messages'. The nature of this flow is an important descriptive characteristic of any object–oriented language. Understanding the potential patterns is important for understanding the potential behavioural effects a protocol will have, so the semantics of protocols depend on the patterns.

Many languages define **types** of object to simplify understanding of computation paths. A type describes *what* behaviour a message should evoke, in general terms, for objects of that type. It is then up to the object to implement the behaviour (answering *how* to do it) as a protocol. In coding a protocol, the programmer relies on the specification of types. Protocols can send messages to types of objects (receivers of a type) knowing what will happen, not concerning with which object of the type binds to the message or how the behaviour is achieved. If all objects obey their type specification, then the protocol will link the correct behaviours and the protocol will evaluate correctly.

The difficulty in understanding the patterns of message passing rises as the number of variables affecting the pattern rises. The receiver is not the only variable. There are two further factors. The path can also depend on other objects (besides the receiver) sent with the message. Typing can help here by ensuring messages only send objects of the type expected by the protocols. In CLOS (Common Lisp Object System) for example, in the interface a messages must match all parameters of a protocol, which includes the type of objects the message carries. Smalltalk™ however does not check the arguments. Usually an incorrect argument type will, eventually, cause a message to be not understood.

The other factors affecting the pattern of passing messages depends on how a receiver shares its protocols. Recall there is a difference between sharing a behaviour through a message and sharing a protocol description of a behaviour. Objects that share protocols are said to have Empathy.

2.4.4 Empathy

The term empathy was coined in a paper called the Treaty of Orlando[5]. It reports a discussion between factions arguing the benefits of one sharing mechanism over another, namely inheritance and delegation.

In the quote from the treaty, which follows, the crucial feature is the assignment of the variable *self*. For empathy, this variable binds to the receiver of the message, not the owner of the protocol. This causes the receiver, not the owner, to fix the pattern of message passing.

"We say that object **A** empathises with object **B** for the message **M** if **A** does not have its own protocol for responding to **M**, but instead responds to **M** as though it were borrowing **B**'s response protocol. **A** borrows just the response protocol, but not the rest of **B**. That is any time **B**'s response protocol requires a message to be sent to *self* (or a variable to be looked up), it is sent to **A**, not to **B**; otherwise **A** and **B** respond in the same way [as if **B** received the message].

"Formally we say object **A** empathises with object **B** for **M** when the following holds: If **B**'s behaviour in response to **M** is expressed as a protocol function **P(B, M)** — that is, **B**'s method for **M** can be expressed as a function that takes *self* as an argument along with **M** — then **A**'s response to **M** can be expressed using the same function **P** as **P(A, M)** — **A**'s behaviour is derived by using **A** wherever **B** would have used itself"[5].

The implementation of empathy is asymmetric. **A** borrows from **B**. **B** does not borrow from **A**. The behaviour of empathy is symmetric. It does not matter if **A** borrows from **B** or **B** holds the protocol and **B** borrows from **A** . This raises the question of who should manage a protocol. In the case of CLOS[62], neither holds the protocol.

Where a protocol is stored and managed is not in itself empathy. Empathy only affects how the variable *self* associates with the receiver after locating the protocol. How a message finds and matches a protocol is a separate orthogonal issue. Often the two issues are related in particular language models. In a number of languages, for the convenience of the programmer, the same mechanism handles both look-up and binding to protocols.

If an object can change the set of protocols it shares dynamically, then the object can dynamically changes its description, ie the messages it will respond to (locate protocols for) hence the object's named behaviours or properties. Such change complicates inter-object communications. Delegation is conditional behaviour sharing.

2.4.5 Delegation

Delegation is a form of empathy. Whereas a **sender** sends a message to a **receiver**, a **client** delegates a message to a **proxy**. When a client delegates a message, the same mechanism locates the protocol as if the proxy was receiving a message. Instead of evoking a behaviour of the proxy, `self` assigns to the client, and the protocol binds with the characteristics of the client. This produces a specific behaviour of the client, not the proxy. The client is still the receiver, not the proxy.

There are two cases of delegation: Explicit and implicit. A protocol coding explicit delegation states the proxy as where to find the protocol, separate from the receiver which is the evaluator of the protocol. Implicit delegation is part of normal message reception. When a receiver gets a message that does not match any specific protocol of the receiver, the receiver can specify a **parent** proxy. The message then delegates to the parent. Implicit delegation models inheritance.

Consider an extreme case of implicit delegation: a client may delegate all messages sent to it, to the parent proxy. No other specific characteristics are contributed. Any attempt to locate specific behaviours of the client will fail resulting in immediate delegations to the parent proxy. Although any protocol found will have `self` bound to the client, the client still contributes nothing, with all messages delegating to the parent. All behaviour is the same as if messages were sent directly to the parent proxy. Now consider adding a single new property to the client. The client behaves just like the parent, but for the single new property. The client refines the property specification of the parent. The client is a software "prototype", an experiment in specification variation.

An even more generalised form of prototyping simplifies message sending. Consider an object receiving a message telling the object to do *something*. Does the object and message not define a more specific object representing "this object doing *something*"? Rather than define complex messages with the description of *something*, prototyping makes it easy to create a new object with the specific behaviour, "doing *something*", on every message send. The new object is characterised by a behavioural filter defining what the object is doing. Computing then becomes an activity of reduction. The object should then reduce to the result of that action, eg the function object [3 + x] receives the message "assign 5 to x", creates an object [3 + x , x = 5] and reduces to the object [8].

The feature of prototypes as a model for objects and messages is derived from the ACTOR formalism (see §2.4.8). The formalism does not specify delegation, but delegation is a mechanism for implementing the formalism.

Delegation has been shown helpful in modelling engineering design interactions[1]. It permits the submission of a query that needs to be answered through accessing some of the properties embodied by objects other than the original receiver of the message. As a knowledge encoding methodology, this use of delegation differs from inheritance because the latter provides an organisation of objects through anticipated connections whereas the former is a run-time technique to program dynamically established relationships. The computational difference between delegation and inheritance lies in the *localisation of processing*.

Delegation is but one mechanism providing sharing between objects. Much debate occurred over the virtues of various sharing mechanisms. However, a consensus was reached, and the dynamics of sharing in object–oriented languages concluded.

2.4.6 Dynamic Behaviour Sharing

'Dynamic behaviour sharing' is a term that describes a language mechanism that allows the patterns of computation to change at runtime. The issue was summarised by consensus between three arguing factions in the Treaty of Orlando[5]. The treaty describes three independent dimensions to characterise the nature of sharing mechanisms: STATIC VS. DYNAMIC, IMPLICIT VS. EXPLICIT, PER OBJECT VS. PER GROUP.

The orthogonality of the PER OBJECT VS. PER GROUP is more easily understood and the ordinate it describes is more discreet in the possibilities. Protocols are shared, thereby defining behaviour, either for individual objects or for a group of objects. In the middle, there are various degrees of a group guaranteeing some behaviour, but allowing idiosyncratic behaviour to individuals.

The orthogonality of the STATIC VS. DYNAMIC and IMPLICIT VS. EXPLICIT is less obvious.

STATIC VS. DYNAMIC: Static sharing is the fixing of the pattern of message passing. There are two possibilities: When specifying an object (protocol compilation), and when instantiating an object (object creation, see §2.4.7). All sharing that is not fixed is considered dynamic, determining the pattern of message passing as each message is sent at runtime.

Two different types of messaging mechanisms affect the patterns of message passing: binding and inheriting. Binding occurs when the message is sent, inheriting (or delegating) occurs on receiving the message. Both, either, or none can introduce dynamism to the pattern.

Static sharing is adverse to modelling and only an software optimisation. For developing prototype applications, static features should be avoided.

IMPLICIT VS. EXPLICIT: Implicit sharing is where a language provides a rule that is generally used when finding protocols and there is some assumption made in the rule as to how to find the behaviour and continue processing. The usual assumption is the recipient of the message gains control of the process flow. Explicit sharing is when the sender can specify all details; both the means of searching and who evaluates the code. Naturally, there are degrees as to the details provided with a message under control of the sender, which can affect the behaviour found. In systems where the searching rule is itself programmable, then both explicit and a pre-programmed implicit searches are possible.

2.4.7 Class–Instances

In general, three main features form the "classification paradigm"[63] and are often held to be essential to object–oriented programming:

- The ability to construct objects as a set of operations and a memory.
- The classification of objects, ie each object as an instance of a class.
- An inheritance mechanism defining superclass-subclass relationships.

This paradigm is synonymous with class–instance, object–oriented software mechanisms, ie systems oriented to objects as instances of classes whose definitions form templates from which many instances may be generated. The instances are intended to correspond to "real-world" information, responding to messages about their attributes and behaviours, while the classes are abstract specifications.

A class groups objects with common behaviour for the purpose of classification. The discussion on categories and taxonomies are equally valid to the classes in object–oriented languages. Objects belonging to a class are instances of that class, and must obey the common behaviour of the class. The instances obey because they depend on implicit inheritance from those classes for their behaviour.

Inheritance is a form of protocol sharing, as is delegation. Instances receiving messages look to their classes for protocols. Upon locating the protocol, the instance evaluates the behaviour as if it was its own. Inheritance is like implicit delegation, but applied per group and may be dynamic or static. Static inheritance is quite common and limits the evolution of instance behaviour.

Classification highlights the complex choice of abstraction technique presented to the programmer. The choice also exists with prototyping but is less obvious.

The behaviour sharing implicit in classification supports an alternative approach to behavioural abstraction. Similar behaviour sharing exists in prototyping languages, but the abstraction is more obvious in the class-instance languages. Classification emphasises the choice a programmer has when abstracting a software problem into objects. The choice is between sharing behaviour and sharing protocol. Should an object inherit a protocol, or should another object be created to exhibit the behaviour? Often the choice is not clear, especially when multiple inheritance is possible.

An object composed of three abstract parts could either models three objects or a single object inheriting from three independent classes. The latter is the philosophy of **multiple inheritance**. The inheritance becomes a mechanism for mixing behaviours.

Consider a clock as a gauge (display) and a timer. Classification says: "It is-a kind of timer or is-a kind of gauge". Alternatively, is it a device that "is-a timer" that "has-a gauge for display", or "is-a gauge" that "has-a timer as input". Alternatively, it could be a clock, an object, that "has-a gauge for display" and "has-a timer as input". These are all possible ways of modelling a clock. The first is multiple inheritance and the next two are different views of clock in a single inheritance system. The last example is not yet part of a classification, so it is just a composition. All provide the same behaviour.

The class originated in an ancestor of object-orientation, Simula™. The class has existed in many languages since, notably Smalltalk, which has the longest history of any object–oriented language still in commercial use today. The class in these languages is a template. The class generates objects in the image of the template. This function provides a guaranteed fixed structure to the objects produced. Fixing the structure gives two important advantages: consistency and optimisation.

The consistency provided by the class is more than simply a logical prevention of inconsistencies leading to errors. The class creates a syntactic grouping of concepts that all instances, objects of the class, abide by. The programmer uses the class as a guarantee that the instance will behave as specified. This simplifies handling objects, just as types mentioned earlier do which, without the enforcement of the class–instance relationship, would otherwise require a more exact understanding of the patterns of message passing.

Classes are generally considered static, in virtue of the assumption that real-world specifics change and generalisations do not; eg, new cars are designed but the idea of car remains static. Most objects change in "state" but are relatively static in their behaviour, described by the class. Cars move, but are still cars.

The assumed static nature of the class has made it the target for optimisation in many class–instance languages. Classes are implemented as static templates, and optimised, leaving the instances to represent the dynamic aspects of an application. Some languages do not even representing the class as an object. The class does not truly exist in the same way as instances exist at runtime. This viewpoint is taken to its extreme by the language C++, which continues the "C" philosophy of highly optimising code.

Enforcing a static structure has its disadvantages, namely when the structure needs to change. The boundary between class and instance is also a boundary between static and dynamic. Since nothing in the real world is truly static, there is always a point, if a model is to remain consistent with the real world, the boundary needs breaching. In some domains, this is more common than in others. In particular the domain of design is notorious for its dynamic nature of descriptions and specifications.

Many class–instance languages require all dynamic aspects of the implementation to be handled by instances, but this conflicts with the nature of design; a design describes the behaviour of an entity in the real world. In the class–instance language, the property of behaviour description, the protocol, is only held by classes. The only other dynamic changes in behaviour are by changing the relationships between objects, as recorded in instance variables. Therefore, the implementation of a design must be by an object that has the property of behaviour description and is capable of changing with the design. The whole point of classes is to provide a behaviour description, but one that is static so to guarantee the interpretation of messages to the class's instances, ie as a type definition. Classes are inappropriate for representing design under this criterion. This does not preclude a different mechanism, in a language supporting the class–instance relationship, for modelling design specifications.

Not all class–instance languages adhere to the strict static nature of the class. Languages allowing their classes to change are said to support schema evolution. Language supporting schema evolution carry a large overhead in terms of requiring compilers, consistency checkers, and error handling routines to enable the schema change and ensure the change is sound.

In Nierstrasz's[64] review of object–oriented concepts he defines schema evolution as an operation on a class hierarchy, not an operation on objects, ie not a consequence of messages. This follows the analogy of a database schema evolution, which is not a database transaction. A normal interaction between objects that dynamically changes the inheritance of behaviours within the object model is dynamic inheritance. Yet, if a class is a generic object, as it is in Smalltalk–80, then schema changes are a consequence of messages. Inheritance changes in Smalltalk are considered schema evolution because they involve coercing the underlying object model for each instance, despite the fact a complex series of normal message interactions achieve this coercion.

Schema evolution is often only available during initial program prototyping, such as in the database Gemstone™[65]. The effects of change on 'established' classes have far reaching consequences that return to the programmer as bugs. The reason for this far-reaching effect is the semantics of messages. The effective patterns of message passing are rarely well understood by the programmer making changes.

The semantics of a behaviour are not only defined by the objects that hold the behaviour, but also by the users of the behaviour. Viewing behaviours as an input output relationship, message goes into an object and the response comes out, then the implementation of all senders encodes the interpretation of the response. That is, the object a message returns is sent messages by the same sender and these messages are all part of the pattern that develops the sender's own behaviour. Theoretically, each message should return an object fully defining the semantics of the response. This is rare and usually messages return a simple data types with little semantic value. These messages rely on the recipients of the result (usually the message sender) to correctly interpret the result.

It is all very well to say that messages correspond to semantics, ie messages have precise meaning, and are separate from protocol implementation, but when programming starts, the semantics of a given message might differ by the time programming ends. In practice semantics of a program evolve with the implementation.

There are attempts to separate the implementation of a class from its semantic obligations[66]. This is believed to be a solution that will further prevent schema evolution from affecting other parts of a system. The semantic obligations are described in terms of type requirements for messages and message responses. The organisation of types can be handled quite separately from class descriptions. Type checking need only occur during schema evolution. If a change is made to the type hierarchy, consistency checks occur once before accepting the change. Ultimately the difference made by type checking is the determination of inconsistencies at the time of change rather than during execution of behaviours, which might occur much later when, the source of the error is forgotten. Typing introduces an overhead on the programmer who must define types to classes and in protocols.

2.4.8 Actor Semantics and Prototype Languages

In the late 60's and early 70's, Hewitt et al[67] developed the ACTOR formalism as part of the PLANNER research project into natural and effective means for embedding knowledge in procedures. They identified the modular nature of knowledge and its dynamic ability to combine the abstract to create the specific. This led them to the ACTOR, a computational model that allows an extendible description of knowledge. The ACTOR formalism is not a language but a computational model describing semantics for the foundation of computer languages.

An ACTOR is an active agent that plays a role on cue according to a script. The computational model conveys semantics similar to an object: modularity, messages, intentions (a conceptual model of behaviour), protection and privacy (encapsulation). Hewitt states that "control flow and data flow are inseparable" in an ACTOR. This is a concise description of encapsulation in that the control over processing (the control flow) and control of data change (data flow) are maintained inseparable in an ACTOR. Control passes between ACTORS through messages. Under these restrictions, the only way an ACTOR can achieve its intentions (behaviours) is either "Every ACTOR should act for himself or delegate the responsibility [pass the buck] to an ACTOR who will"[67]. It is through delegating that an ACTOR extends the representation of knowledge.

Experiments in programming styles have implemented some of the ACTOR philosophy in Lisp. Early examples include Kahn's Director[68] and Lieberman's Act 1[69]. These experiments are specific implementations of software machines using ACTOR. The concept of a prototype that delegates to a proxy (see §2.4.5) as a method of representing knowledge, came from these experiments. All prototypes are an ACTOR. Each knows a proxy, which is an ACTOR. Any message a prototype does not specifically know how to resolve will resolve the message by delegating the message to its proxy. This message delegation is more specific than Hewitt's "pass the buck" between ACTORS. Before a prototype delegates, control passes to the message (also an ACTOR) and assigns the variable 'client' to the prototype. Therefore, languages defining this delegation have standardised the intentions of messages.

The standardising of object organisations within languages has generated a lot of argument. Initially Lieberman[70] argued class–instance inheritance was inferior to prototype-delegation. Stein[71] countered that delegation is functionally the same as inheritance. Other languages implement various other organisations; Ungar's[72] Self, Mercado's[73] Hybrid, and Agha's[74] ACTORS are but a few. The arguments were clarified when Stien, Lieberman and Ungar produced the Treaty of Orlando[5](see §2.4.4), which abstracts the concepts of behaviour sharing. Each concept exhibits useful characteristics for software modelling. The important issue in designing or choosing a language is deciding which characteristics best suits the knowledge represented.

When Zucker[1] represented materials design he specified behaviour sharing that supports both searching through information and then experimentally combining information. He sought a classification to organise his knowledge, which supports searching. He sought the expressive description of prototypes to experimentally combine information into design solutions. Zucker got both these characteristics by starting with Scheme[75], a dialect of the Lisp language that adopts actor semantics. He modified the language to provide each object with a strict classification with inheritance, while the delegation of scripts between objects allow the dynamic combination of information from different classifications. This new language he called SPLINTER.

2.5 Selecting a Language

This review describes some of the software concepts applicable for knowledge representation. For the majority of polymer knowledge, these concepts are satisfactory, but inferring general polymer behaviours requires a language with highly abstract functionality. Object–orientation is reviewed because the philosophy of behaviour sharing encourages abstraction and classification, which the class-instance paradigm exemplifies.

This review starts by introducing a description of design as a method that uses classification and generalisation of information. As discussed, designers first identify suitable general materials during the initial stages of design. They attempt to generally satisfy the design, possibly by adjusting other design parameters, before attempting to satisfy it with materials that are more specific. This is a principle method of design, which "leads inexorably to a minor but unmistakable invention", as quoted at the beginning of this review. The method relies on a taxonomic classification, where each class generalises materials. The method proceeds as long as the designer can interpret design benefits from the abstract behaviour of the generalisations. Therefore, a language implementing this method of design requires a concept of classification and data abstraction.

A relational model can describe a hierarchy. Relational algebra can abstract properties of grades to give averages, maximums, minimums and general distributions. Then why is object-orientation chosen for representing polymer information rather than a relational model? The distinguishing features of these two data models are the way they manipulate data. A relational transaction processes many entities with the same query, while an object transaction evaluates many different messages over a few objects. The benefits of these features for representing polymer information lies in the way designers use polymer information.

If designers are able to translate a specification for a product into a material specification using general material properties, then a relational calculus query could represent the material specification and a relational database could effectively locate grades matching the query. This query approach was rejected as an uncharacteristic design method.

Designers can provide a loose specification of a product. Often the designer can translate them into desirable general material properties. The important difference is the designer knows the criterion is only approximate, and the criterion depends on other perspectives of the design, which can change and therefore change the criterion for the material. What the designer first wants to know is how classes of material behave in general. The designer can then translate the product specification into different material specifications that gives each class of material the best chance at solving the problem. A new specific criterion then applies to subclasses of a class, while other criterion apply to the subclasses of other classes. The designer decides on direction at each new source of information, which is a style of information browsing. Adjusting information processing according to the type of information is a general feature of object-oriented messages not supported by standard relational manipulation languages.

The object-oriented paradigm provides greater abstraction than the relational model. In a relational model the data definition of grades, their classification, consistency rules for inheritance, and rules to infer abstract polymer behaviour would all be represented as separate database objects (ie table definitions, tables and queries). In the object model, this level of abstraction exists too. Method objects represent the rules, while other objects represent the polymer abstraction. Unlike the relational model, the object model can abstract all these behaviours into a single object. For example, the behaviour of class objects, whose metaclass inherits the behaviour of inheritance and object representation (ie the grade-definition), can extend by the addition of new object behaviours to infer abstract polymer.

Of the object-oriented approaches, neither the class–instance paradigm or the actor formalism prevents complex modelling, but specific optimisations of individual languages might. Unlike the class–instance paradigm, which implies inheritance and classification, the actor paradigm does not naturally support a programming structure, though it does not prevent an actor language from developing one. Zucker started with a language with some actor semantics and enhanced the language with a taxonomic structure. In Zucker's case, the qualities of the prototype were a dominant benefit for his initial choice of language for modelling the evolution of design. The work presented here requires classification, and there exists many very good examples of languages that support classification.

The language will describe complex relationships between grades, classes of polymers and the abstraction of properties, but also evolve the description since the classification will continue to grow and develop. The language requires schema evolution. The schema includes the classification hierarchy and the description of polymer classes with polymer properties.

ObjectWorks™ 4.1, a variant of the class–instance language Smalltalk, was seen to be a suitable language for the representation of polymer materials information for design. Smalltalk had many characteristics deemed beneficial to the research in Table 1, including an expressive user interface capability. The interface to the knowledge base is important for browsing the information.

Everything is an object	all entities in the language are objects and can evolve.
∴ Classes as objects	As well as define protocols, classes can have their own behaviours, e.g. population generalisation and abstraction
∴ Protocols as objects	Can define engineering property objects as a kind of protocol
Strict classification hierarchy	encourages a cleanly principled taxonomy.
Runtime evolution of classes template	Grade structure can change at runtime, though not efficiently. Changes are per group, not per individual.
Dynamic protocol inheritance	Protocols change efficiently.
Large class library	Faster development time
Advanced user interface tool-kit	Encourages effective interface development.

Table 1: Known characteristics of Smalltalk deemed beneficial to the research

A suitable language means some of the expected functions may be difficult or impossible to achieve. Table 2 lists the characteristics considered challenging at the beginning. The absence of database support, which is necessary for an extendable knowledge base, suggests the Smalltalk data model may not be suitable for database application. Smalltalk supports a single inheritance classification and only implicit protocol sharing, so using the class to both classify and describe polymer materials with properties orthogonal to their classification will be a challenge. Encountering these barriers and others in the representation simply identifies how the language does not suit the problem. Where possible barriers are overcome and the research continues. Overcoming the barriers is also of interest since it characterises the problems not foreseen at the beginning of the research.

Only implicit protocol sharing	Object interface does not support explicit protocol sharing.
No prototyping	Classes must be used to manage protocols.
No Persistence	Requires a third party database service.

Table 2: Known challenges to Smalltalk at the beginning of the research

Chapter 3 POISE: Polymer Objects in a Smalltalk Environment

Overview of Objectives

For useful representation of polymer information for design applications, the following list identifies issues on:

1) managing a rich variety of informative descriptions, each with the potential to extend,
2) managing sparse data, and providing suitable defaults where possible,
3) encouraging descriptions that are independent of a particular purpose, through an appropriate classification which generalises similarities across the domain,
4) defining and managing many levels of abstractions from the domain generalisations.

The following chapter proposes a conceptual description, or loose schema. The schema describes software tools for achieving these objectives. Software conforming to the object–oriented class–instance paradigm provides the principles. (1) Objects encapsulate information, providing an independence that allows the information to evolve. (2) Objects share behaviour, typically following a concept of default inheritance, from more abstract objects. (3) In the class–instance paradigm classes enforce a strict classification of instances.

In addition, the class defines a consistent structure of objects, which is useful for supporting traditional information management tools, namely:

5) database support techniques for information storage, in a form that appeals to the organisation of information in the polymer industry, and
6) supporting interface design for reflecting the representations and appealing to the user through intelligent interaction.

In particular, an implementation of Smalltalk™ has characteristics deemed beneficial. The final application is implemented in Smalltalk™, and named POISE. Following this chapter is a discussion on the particular aspects of this schema that challenges the object model of Smalltalk™. This schema is not particular to Smalltalk™, and it should not imply Smalltalk™ is the only possible language for the implementation.

The schema follows the information flows from source through to the classification. The description, support for abstraction and grade representation of this classification is at the core of the schema. Extensions to this core add orthogonal descriptions and database management capabilities. The schema, visualised in Figure 4, shows the data acquisition on the right flowing into the classification and supported by an object management system on the left. The rest of this chapter follows this flow.

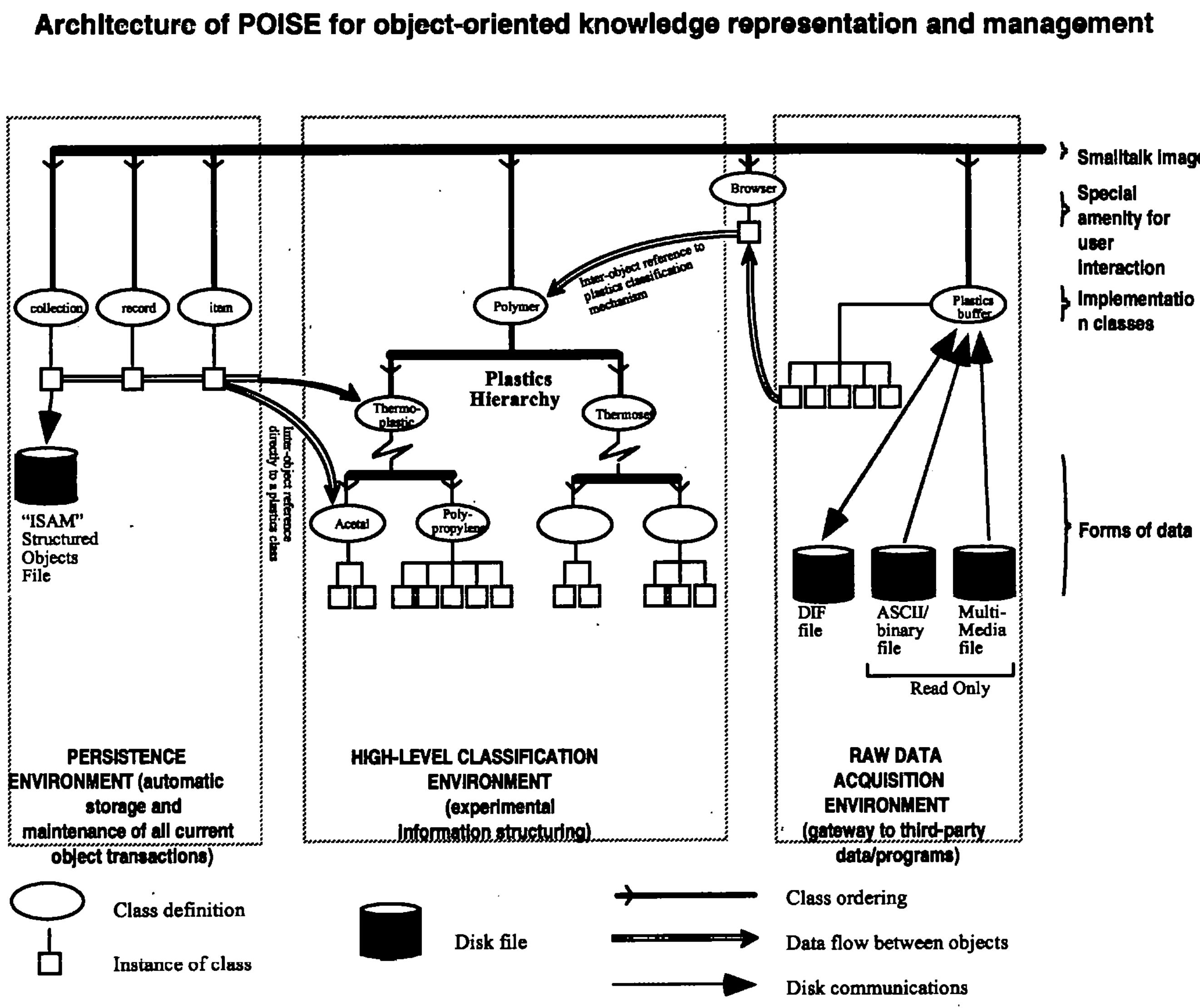

Architecture of POISE for object-oriented knowledge representation and management
Smalltalk imag
Special amenity for user interaction
Implementatio n classes
Forms of data
collection
record
item
"ISAM" Structured Objects File
Inter-object reference directly to a plastics class
Browser
Polymer
Inter-object reference to plastics classification mechanism
Plastics Hierarchy
Thermo-plastic
Thermoset
Acetal
Poly-propylene
Plastics buffer
DIF file
ASCII/ binary file
Multi-Media file
Read Only
PERSISTENCE ENVIRONMENT (automatic storage and maintenance of all current object transactions)
HIGH-LEVEL CLASSIFICATION ENVIRONMENT (experimental information structuring)
RAW DATA ACQUISITION ENVIRONMENT (gateway to third-party data/programs)
Class definition
Instance of class
Disk file
Class ordering
Data flow between objects
Disk communications

Figure 4: An initial architecture of POISE

3.1 Source Data

For experimenting with information management, POISE needs only a minimal strategy for data exchange, preferably accessing a single large source. Initially CAMPUS (§2.1.1.) was chosen. In principal, POISE requires a more general interface catering to many sources. To satisfy the principle, a CAMPUS specific interface passes data to a general data structure that may represent data from other sources. This general data structure is a binary relational table.

3.1.1 Reading Binary Relational Table from DIF Files

DIF, delimited interchange format, is a simple data file format that separates fields in a record with a delimiting character, allowing the fields to vary in size. Most spreadsheet applications and relational DBMS can produce output of this kind. POISE requires the fields within the records to correspond to the following binary relational data schema:

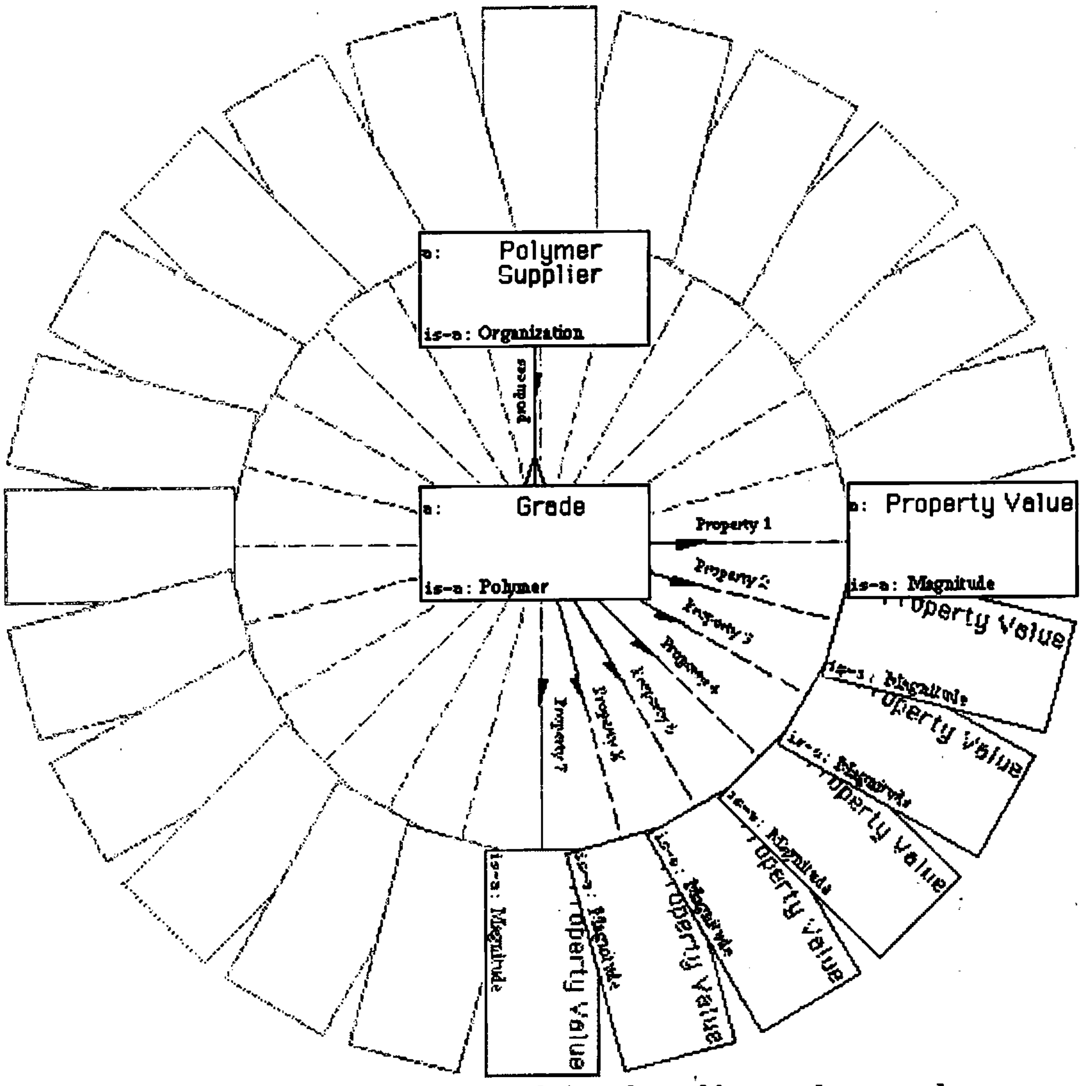


Figure 5: The open-ended relational view of an arbitrary polymer grade

{Unique Polymer Name}—{Unique Property Name}—{Property Value}

In this case, the binary relation maps a 'Unique Polymer Name' to a 'Property Value' under the named relation 'Unique Property Name'. The schema allows any property relation that is uniquely identified by the string in the 'Unique Property Name' field. Diagrammatically, this creates a model as shown in Figure 5. A `PolymerSupplier` is an object representation of a DIF file containing the source descriptions of grades. The grade's description takes the form of an aggregation of property relations to magnitudes, which are often numerical and described here as property-values.

The data structure adopts a binary relational data model, but not the inference engine that usually comes with relational databases. If an information source infers information from the data, it must explicitly export the inferred data in relations. For example, a database assuming a closed domain and closed world does not state what is false. POISE assumes what is not stated is unknown, so false statements must be given as relations in the input. The potential exists for information to be lost if the system generating the output makes assumptions on the schema of the receiving system.

The binary relational data structure does not rely on any assumptions or any particular domain. A DIF file containing such relations is quite capable of conveying the description of an arbitrary grade as a loose grouping of properties, ie the relations as described in Figure 5.

The binary relation file does rely on the unique name assumption for both polymer grades and the properties used to describe them, but only within defined sub-domains of data. The domain of POISE covers all knowledge of polymer materials. Any division of the domain creates sub-domains. Each `PolymerSupplier` is considered a sub-domain of grade descriptions. The boundary of the sub-domain simplifies the scope of an individual DIF file or aggregation of files, provided by suppliers. In the case of CAMPUS, there is a separate file for describing the semantics of the property relations and another for specifying textual descriptions of grades separately (see §3.1.3)

The knowledge content of a relational database will often include constraints over the domain entities belonging to an entity set. In particular, each property describes an entity set of property values. The source could define constraints over the property values hence define acceptable bounds.

Interpreting the 'Property Value' entity from a field in a DIF file relies on the semantic meaning of the property relation. In its raw form, the DIF field is a string of characters or bytes. In some cases, the entity only requires unique identification, in which case a string may be a reasonable representation of the entity. More often, the entity is a magnitude with other specific semantic qualities. The default behaviour when interpreting the 'Property value' is to convert the string into a real number. If the translation mechanism can not coerce the string to a number then the value is left as a string. This behaviour can change on a per property relation basis. The definition of the property relation in POISE can include a valid data type for the value acceptable for representing the property.

3.1.2 CAMPUS

The polymer data used by CAMPUS is available in two different file structures. The format found with the commercial distribution of CAMPUS is a binary file. CAMPUS was also available on request from the polymer suppliers in ASCII (American Standard Code for Information Interchange) file.

During the period of the project (1990—1994) the binary format changed when a new versions of the CAMPUS program, CAPS[76], was written. The ASCII format remained consistent, presumably because it is used to communicate the data to the CAMPUS software developers. The difficulty in obtaining the ASCII version though made it necessary for POISE to read either file format.

Both file formats contain the same information. CAMPUS portrays all polymers with the same list of properties. It classifies properties by type (eg mechanical vs electrical) and polymers by material family. Each polymer includes a textual description.

CAMPUS portrays a concrete aggregation of properties, so unlike the general schema, a static data structure could represent CAMPUS grades, and initially a class did. Instances had a fixed set of attributes, one for each property and one for the text. After the `PolymerSupplier`, which groups property-value associations, there was little need for the old class except for the code generating instances from the CAMPUS files. A subclass of `PolymerSupplier`, `CampusPolymerSupplier`, specialises the general representation with this code.

3.1.3 CAMPUS Data in ASCII Text Format

The text format can be likened to a simplified (or 'normalised') binary relational file. The file consists of tables, one for each polymer grade. The table has two columns, the first with integers uniquely identifying a property, the second associating the property with a value, see Table 3. The integers in the first column reference property descriptions in a second ASCII text file from the CAMPUS disk, see Table 4

```
301 Vestolen A 3512 F
19 5 89
101 0.932
102 17
103 10
104 >50
107 550
108 500
109 250
112 50
113 14
```

Table 3: ASCII Campus data file (edited)

The file differs from a DIF file. The fields are of fixed character size, rather than field delimited by a special character. There are also Boolean properties where the identifiers existence represents true, and its absence infers false (ie a closed world assumption). The absence of other properties, eg the mechanical (1) property (05) 'Tensile strength' in Table 4 is absent from the record in Table 3, infers no measurement exists.

This structure simplifies the task of the `PolymerSupplier` since the file groups together all the associations of one grade in sequence. A process iterates through the file without need to locate each grade for each property. A binary relational file does not necessarily group relations by grade. The `CampusPolymerSupplier` encodes this difference.

```
Huels Ag
3 Families
01* Vestolen   (PE-HD)                                     PE-HD
02* Vestolen   (PP)                                        PP
03* Vestolen   (PP + EP)                                   PP+EP
04* Vestolit   (PVC-P)                                     PVC-P
05* Vestolit   (PVC-U)                                     PVC-U
06* Vestolit   (PVC-HI)                                    PVC-HI
07* Vestyron   (PS)                                        PS
08* Vestyron   (S/B)                                       S/B
09* Vestamid   (PA 612)                                    PA 612
10* Vestamid   (PEBA)                                      PEBA
11* Vestamid   (PA 12)                                     PA 12
12* Vestodur   (PBT)                                       PBT
13* Vestoran   (PPE)                                       PPE
14* Vestoblend (PPE + PA)                                  PPE+PA
16* Dyflor     (PVDF)                                      PVDF
15* Trogamid   (PA-6-3)                                    PA-6-3
1 Mechanical Properties ( At: 23/50)
01* Density                                       g/ml     Dens
02* Stress At Yield      (50mm/Min)               N/mm2    Stssyi
03* Strain At Yield      (50mm/Min)               %        Strayi
04* Strain At Break      (50mm/Min)               %        Strabr
58* Stress At 50% Elong.(50mm/Min)                N/mm2    Stss50
05*-Tensile Strength      (5mm/Min)               N/mm2    Strgth
06* Strain At Break       (5mm/Min)               %        Strnbr
07* Young`S Modulus       (1mm/Min)               N/mm2    Ymod
08* Creep Modulus                1h               N/mm2    Ec1
09* Creep Modulus             1000h               N/mm2    Ec1000
10* Impact Strength (Izod)    +238C               kJ/m2    Imp+23
11* Impact Strength (Izod)    -308C               kJ/m2    Imp-30
12* Notch.Imp.Str.  (Izod)    +238C               kJ/m2    Nimp23
13* Notch.Imp.Str.  (Izod)    -308C               kJ/m2    Nim-30
14* Notch.Tens.Imp.Strength   +238C               kJ/M2    Tenimp
```

Table 4: CAMPUS property file

The property definition data, Table 4, is read by the `CampusPolymerSupplier` creating an automatic partial description of the properties. This meta-knowledge includes the full name of the property, a common abbreviation, and units. The file also describes a set of mutually exclusive properties corresponding to polymer family membership (the first 15 properties, prefixed with a 3, eg 301 for 'VESTOLEN (PE-HD)'). Each grade defines only one of these properties. The family allows the automatic placement of the polymer in the POISE classification, see (§3.2.3).

3.1.4 CAMPUS Data in Binary Format

The binary format represents each grade in a record with a fixed number of bytes. The main numeric properties are represented by two bytes each and identified by their index (position) within the record. This index corresponds proportionally with the identifying number found in the property file. Unlike the CAMPUS text representation, all properties are represented, even if not applicable or unknown. The two bytes only represent discrete values. These values include a range of numbers — both integer and float — and special states such as 'value unknown', 'value not applicable' and property specific states such as 'no break' for impact tests. The record also contains the name of the grade, in a fixed length field, the family of the grade, by integer corresponding to the property, and an encoded date to identify the version of the data.

The binary file uses the same property definition file (Table 4) as the ASCII file.

3.1.5 The Transitive Data Model

The `PolymerSupplier` reads DIF files, the `CampusPolymerSupplier` reads CAMPUS files. Both generate representations of grades. The TDM (Transitive Data Model) is the temporary representation of grades entering POISE.

On the first attempt at acquiring data from CAMPUS, the data was placed in objects that specified each CAMPUS property explicitly, so adopting a similar fixed data structure used by CAMPUS itself. The objects were rigid, requiring a redefinition of the objects data structure whenever new properties were encountered.

As new properties are a characteristic of the rich property descriptions of materials, a general transitive data-model was designed. This model, like the binary relations, adopts a set-like structure that collected relations. Any number of relations could be added. The model was not to be used for any inference so there were no restrictions on what relations were added since no meaning is attributed to them at this stage.

The requirements of the TDM are simpler than the representation of grades in the classification system of POISE, which does apply inference over the members. The TDM does not ensure consistency across properties. The concept of the property relation only requires unique identification.

The TDM model includes some mandatory property descriptions of grades. Most only simplify the development of POISE. We believe the software implementing POISE could be re-written so grades could exist within POISE without these properties but that it would introduce unnecessary difficulties when visually identifying grade entities. These properties are otherwise treated the same as any others. The mandatory properties include:

- A name for the grade
- The supplier of the grade.
- A text description
- A validation date.

Rather than enforcing the inclusion of these properties as input requirements, the TDM provides a default mechanism for each of these properties. The date is set at the current modification date of the file read. The name is either derived from the supplier as 'Unknown from <supplier>' or just 'Unknown'. The text is a copy of the name. The supplier defaults to the file name of the file read.

There is one exceptional property. Grades must belong to a chemical family. This relationship is the beginnings of a taxonomic classification. It is the only mandatory relation for automatic classification in the POISE schema. Any grade entity entering POISE without this relationship will not be able to take its place with other grades in the classification. Since there is little point is defaulting to 'is-a`Polymer`', and rather than make this relationship the sole input requirement, a browser was proposed to allow the user to place each grade without the property. Since all CAMPUS grades specify this property, the development of this browser was not a high priority.

3.2 System Data Flow

The data flows from source to a TDM, the temporary representation of grades. The TDM lacks any structure to support inference. The next step is to transfer the data in the TDM into a more knowledgeable structure in the classification architecture. This structure provides many different inferences. Restructuring and placing the grades requires the application of inference rules and occasionally some interaction from the user. As POISE collates more about polymers, the classification develops character. This section follows the flow of information and the effect it has on the classification.

3.2.1 The Grade

A transitive data model (TDM) initially collects the raw data on a polymer grade as arbitrary property-value pairs, managing them as a single group. The TDM acts as a flexible interface between the data acquisition system and the classification. The next step is to find a class for the grade. The `PolymerSupplier` object manages a collection of TDMs, and defines a mechanism for placing the grade into the classification.

The classification of grades divides into two steps. CAMPUS provides the information for the first step, which is to group chemically similar polymers into a family. This is the most specific level of class in the classification. Using the tools provided the user manually generalises more classes and completes the classification. A virtue of an evolutionary structure means these two steps can occur in either order. As soon as a CAMPUS grade enters POISE, the grade can automatically migrate to the class representing the family.

Each class in the classification describes a data structure for its member grades. This structure is a more formal description of the grade as an instance of a class. Each relation is unique and specifically described, unlike the general treatment in the TDM. Figure 6 shows the structure of a single relation, linking a grade with an attribute, with the relation qualified by a `Property` object. This object is the subject of the next section.

The TDM requests a new empty structure from the polymer family and fills the structure by matching properties in the structure with the properties in the property-value pairs. If the TDM defines a property that is not in the structure then there is the potential for the property to be lost.

However, POISE prevents the loss by checking the properties of all TDMs before adding. For example, as result of reading a CAMPUS file, the `CampusPolymerSupplier` object collects up the properties for each family class and compares them with the classification. Any discrepancies induce a request to modify the classification to provide for absent properties in the schema (see §3.3). Only then are the grades added. New families are also defined when not found in the classification. These families automatically inherit from the general class `Polymer`.

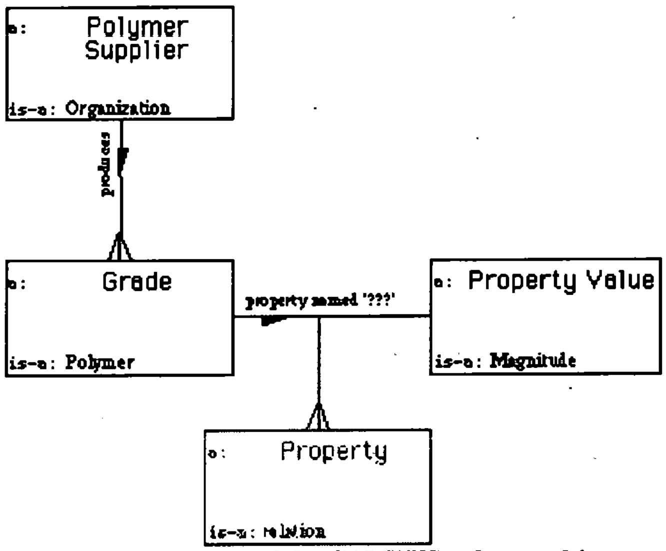


Figure 6: Schema of the CAMPUS polymer object

3.2.2 The Property Object

The `Property` object has the following roles in the implementation of POISE:

- An identifier of an engineering test applied to polymers.
- A unique key for property-value pairs in the TDM.
- Interprets values in the property-value field of a DIF file
- The ability to negotiate with a class on how instances represent grades.

By default, any two objects occupying separate locations are identifiable as different, but they may be semantically the same. Identifying semantically different properties requires information to differentiate between them. Simple attributes can be compared automatically, such as a name string, but a textual description of the test requires a user. Two different texts can have the same semantics, requiring a user to read and interpret the text to determine differences between properties. Either way, the information allows properties to be differentiated.

A unique `Property` specialises each association between the TDM and values of a grade. In the TDM, the `Property` object is a key in a look-up table. This key is the only distinguishable difference between different property data in a TDM.

When a `PolymerSupplier` reads a DIF file, the contents of the second field names a property. The `PolymerSupplier` locates the `Property` object matching the name. The third field containing the property-value is a string. The `Property` object converts the string to an object of the type representing a value of the property. The type is an attribute of the Property object, which also provides the behaviour to transform from a string. The TDM then associates the Property, as a key in a hash table, with the value object.

Properties are not pre-defined. Grades will always require more property descriptions. POISE is able to receive new properties at any time. New CAMPUS properties are no different from any others. CAMPUS describes all the properties in each database in a separate text file, Table 4. Each property in the file is a record with a name as a string, unique symbol (a shorter sequence of characters), and a string for the units of the property-values. POISE creates a new Property with this information as attributes to identify the property. Defaults are available for all other behaviours of a Property object.

CAMPUS mainly defines grade's property-values as a single rational number. Rational numbers describe an ordering and ordering is necessary for comparison; a prime function in design. So it is reasonable to assume all specific properties can be represented with a rational number, though other ordering representations may be found more appropriate. It so happens that all the CAMPUS properties are quantitative properties, which means the rational values are the result of some principled test. For some properties a measurable test has not been found, and these properties are often described qualitatively. In principle even these properties can be ordered and databases like PLASCAMS-220 use rational numbers as an abstract ranking to represent qualitative data. In this form they do not pose any more of a challenge to classification and abstracting as qualitative measures. Their absence in POISE is solely a consequence of the source of data. Nevertheless, it should be remembered that although the abstract use of rational numbers for measuring qualitative properties has a logical basis in ordering, there is no principle to the measure of qualitative properties.

The default type attributed to a property is an object representing rational numbers. A consequence of this default can be a loss of information, such as engineering units, in the representation of the values. Associating a value with units conveys more information. Instead, lost information is maintained as an attribute of the Property object. As POISE developed, the Property object became a repository of 'lost' information specific to the values. As the development of POISE evolves, this information finds a more appropriate representation, such as part of a value's type definition.

The default Property behaviour also makes it easier for users to define their own properties. Initially only a unique name is needed. The user can then refine the Property's attributes later.

Objects of any language could easily model all the roles in the above list. All are typical computing behaviours except for the last role. The last role, negotiating with a class, involves evolving the description of other objects. In a class–instance language, classes define the behaviour of other objects. The class describes the meaning of each value attributed to a grade of the class. However, properties also describe the meaning of a value attributed to grade. A class describes many attributes whereas a `Property` describes only one. Therefore, a materials class is an aggregation of materials properties.

A class describes objects as a single unit of description, or template. The template is not a composite structure, but a single description that has been contributed to by many properties. A `Property` requires some functionality where it may define behaviours and include these behaviours into the class template. The `Property` is a tool that adds behaviours to the class machinery that produce grades. The `Property` as a tool for constructing object templates is a unique object–oriented issue that arises from POISE. The `Property` is a partial-template object[77], see §4.4.3.

This approach to class definition is similar to the 'mixin' style of multiple inheritance of CLOS[78]. It differs as it does not enforce a membership behaviour with the `Property`. Grades have no relationship with the `Property` entity, only the behaviours the `Property` provides to the class template.

The process of installing, moving and removing properties over to classes is further described in §3.3.2. In the implementation of POISE, §4.4.1, addresses how classes add properties.

The description of a `Property` so far has been more as a tool in the machinery of POISE. The `Property` is also an entity of knowledge in the materials domain. Some of that knowledge is useful for identifying uniqueness across properties. As a representation of part of the materials information, a property should also provide:

- a text description of an engineering test, which is then translated into,
- a repository of behaviours that objects with the property may adopt.

The text, useful for identification, is also a repository of knowledge, which may be translatable into computable rules by a knowledgeable designer. These rules become behaviours of grades, but aggregate by property. The `Property` adds the behaviours to the classes of grades with the property. This means if the `Property` moves in the classification, so does its associated behaviours.

3.2.3 Automatic Classification Declaration

Initially POISE does not contain grades or classes except for the class represented by `Polymer`, which is the root of the classification hierarchy. By restricting this experiment to the domain to polymer entities, POISE can automatically classify TDMs under `Polymer`. The only other classification information is the polymer family property. The TDM demands a polymer family name from each entity. Each new name defines a class 'polymers belonging to polymer family named ...' and is subclassed automatically under `Polymer`.

Each new polymer family class is undescribed, with exception of its name and its membership to `Polymer`. Even `Polymer` initially describes only a name 'Polymer'. The concept of `Polymer` is empty and in its empty state a poor representation of a polymer category. POISE provides specialisation of the classes through a user interface, which is covered later, and by automatic inference that generalises from the TDMs.

A `PolymerSupplier` manages the first grades POISE adds as TDMs, grouping them by their polymer family "property". Each group requests the classification to provide a class with the collective properties of the TDMs. Each request is a transaction between the class and the `Property` object in the TDMs. This ensures that classes specify a template capable of storing the property information represented in the TDM.

The class template is common to all members of the class. Adding properties to the class modifies the data structure of existing members, thus keeping a unified representation of grades. This unification results in a relational de-normalisation by polymer family. Membership of the classification imposes a uniform property specification over the grades that did not exist in the TDM's unrestricted relational representation.

Unrestricted addition of properties aims to preserve all information obtained from the TDMs. Even if there is only one member with a `Property`, all members of the class will be modified to represent the `Property`. The automatic addition of properties assumes the existing grades and any other future grades of the class not specifying a property are simply sparse; the data is not available but may be specified.

The presence of properties in a TDM does not distinguish whether the property is a characteristic specific to the grade, to the polymer family or to polymers in general. The assumption of sparse data could be taken further; the absence of the property in other families is also due to sparse data and the property should characterise all polymers. Given any property, it is potentially a property of all polymers or specific to the polymer family class, or indeed specific to the grade itself. For a solution, POISE looks towards the nature of the source, the `PolymerSupplier`.

3.2.4 Transfer of CAMPUS Grades into the POISE Architecture

The following looks at the consequence the primary data source, CAMPUS, has on the architecture.

CAMPUS uses a fixed unified data structure to represent all engineering properties for all polymer grades. CAMPUS maximises the usage of the fixed data structure by tending towards generic polymer properties. By keeping to properties measurable (but not necessarily measured) for all grades avoids having useless slots allocated to properties that can not be measured for a grade.

In the case of data from CAMPUS, the majority of properties describe all polymers. Since the placement of TDMs, hence their properties, is under the control of the CampusPolymerSupplier object, it can specialise the rule for property placement. The default place for CAMPUS properties is under Polymer. This is where CampusPolymerSupplier puts them. Since the rule is general to all TDMs, CampusPolymerSupplier may sidetrack the PolymerSupplier's automatic property analysis of TDMs by class described earlier, placing all properties directly in the Polymer class

An interesting exception to generic properties in CAMPUS is the property water absorption. This property is usually applicable to Nylons. It so happens that many grades in CAMPUS are Nylons, hence its declaration in the data structure of all polymers is not inefficient. If this property is only applicable to Nylons, what do the other non-Nylon grades store in the space provided in the data structure? CAMPUS uses a special state, represented as a string 'NA' in the ASCII file, for 'not applicable'. This causes the TDM to specify the property with the value 'NA' for non-Nylon grades. To handle such nonsense, the TDM could be notified when the property translates the value that it is not an appropriate representation of the property. What then? If the property is just removed, then the CampusPolymerSupplier will assume the property is just unmeasured.

A state for 'Not applicable' is useful to the user for developing the classification of polymers. If one grade in a class is discovered that should not define a property then the property can not be related to the classification principle. Removing the property from the class will modify the data structure of all members, including those that define data for the property. Removal will cause loss of this data. It is simpler if the system just marks 'Not Applicable' until the property moves during re-classification by the user, rather than removed on an ad-hoc basis.

Two factors contribute to the population of properties settling down at the root of the classification. The assumption that the absence of data is sparse and that the CAMPUS database uses generic properties. Although the descriptions of the more specific classes are empty, they do not determine the principle of classification. The classification is based on a single property: polymer family membership. If the classification does indeed group like with like, then this similarity should be reflected in the property values of the specific grades grouped. This analysis of similarity was part of Spedding's[8] work. Generalising over the specific grades, to characterise each class, is a function of the hierarchy, §3.5.

All CAMPUS properties initially describe all polymers. This results in all grades sharing a homogenous data structure, just like the structure in CAMPUS. This is not surprising, since the information on classification originates from CAMPUS, and the details of this information are yet to enter POISE. Unlike CAMPUS, the polymers are not left in this homogenous state. The classification is not intended to remain static. Later, mechanisms for evolving the classification are given special consideration in (§3.3.2). The only issue at this stage is to prevent loss of data. The user must add knowledge absent from CAMPUS for further classification to occur.

A further consequence of choosing CAMPUS is that it leaves the classification as a shallow hierarchy. With no other source of information, it is up to the user to add abstract classes between `Polymer` and the polymer families, thereby creating the deep levels of representation found in the polymer domain (§2.1.8). User interaction requires, appropriate tools and these are considered in section §3.3.3.

3.2.5 Maintenance of Unknown Data

Sparse data and the concept of incomplete descriptions both have an identifiable state. For completeness, POISE considers four states exist for data:

- Known
- Not applicable
- Sparse
- Unknown

Two states exist for a property: $(P \in C) \cap (P \notin C)$ where C is any class template and P is a property. If $P \in C$, then data for the property is either known, not applicable or sparse. Unknown corresponds to $P \notin C$.

When a TDM represents a grade, there is no distinction between unknown and sparse data. Sparse data exists in the POISE data model because the data model recognises two states of absent data: when $P \in C$ and when $P \notin C$. When $P \in C$ is true, the property is known to be measurable, and POISE infers a value for the property. The contrary statement, $P \notin C$, does not mean the property is 'not applicable' (NA), ie not measurable. The domain is not closed and properties will always exist that are measurable but are not yet represented. Until represented, inferring defaults is not possible. Properties with values as NA are a subset of $P \notin C$. For all the properties in the domain, POISE expects $P \in C$ is true for all classes that can measure the property. The absence of a known property from a class is then assumed NA in that class. There is initially a period between a properties definition and its placement in the classification where this assumption is incorrect, ie a property is known but $P \notin C$.

The absence of a property from the system implies the descriptions are incomplete. If the property is absent from the grades description (class), but present in the system, then this indicates that the property is NA to the grade. When the system adds a new property, by default it is NA to all grades until it is correctly added to the classes. Even then, default values are not implicitly inferred until some grades attribute actual values to the property.

Requesting a NA property for a grade responds differently from a request for a known property of a grade which is sparse. A sparse condition results in a default value, and NA in an error message. With NA properties, it is semantically incorrect to request the property. Sparse data assumes the property has yet to be tested against the grade. POISE knows it is semantically correct to request the property of such a grade. The problem is how such a request should be handled when POISE does not know the value. These issues are addressed either explicitly as a behaviour of a property or generally through default values based on typical values of the property within the classification.

A response of a default value and a known value should also be recognised, as the two states have different accuracy. They both at least produce computable values.

3.2.5.1 Not applicable data

A grade describes a property as 'Not applicable' (NA) if the grade is inappropriately grouped in the classification or if the property is an inappropriate description in the class. The classification should normalise the property specification to remove such states.

The NA state is rare given the initial source of POISE. CAMPUS properties are mainly applicable to all polymers. The state does exist for a few properties, eg water absorption, which is applicable only to Nylons.

A property value representing the state NA is useful for flagging a possible fault of the classification as it can indicate an unusual grade that deserves a separate subclassing from other grades. Whatever the reason for its existence, a property behaviour that returns the state should generate an error, since a proper model would not normally respond to the behaviours of the property.

The NA is also useful during the process of property placement, either automated or manual. Placing a property in a more general class is possible without inferring all grades exhibit the property. Grades not exhibiting the property can be given the NA state. Later a process of relational normalisation can remove NA states.

3.2.5.2 Unstructured data: Text

All grades are partial descriptions. Information is usually available on individual grades before it is understood how the information relates to the polymer family or polymers in general. The information can also be of a lesser pedigree, and not available for general description. When the information is better understood it may describe a POISE property and add to the description of a class.

Unstructured information gains nothing more from classification than the association with the classified grade. POISE manages the unstructured information as text in a single collection. Such a collection of text descriptions exists in the CAMPUS system as a separate delimited file, with an entry for each polymer identified by name. A single text collection is a representation that benefits simple sequential searches through the text. A sequential search for a key word is a simple generic tool for finding similarity between grades. Individual grades are not attributed with the text. Instead, the collection maintains an index of grades and associates each grade with a block of text in the collection. Sub-collections of text for any group of polymer can be created when needed.

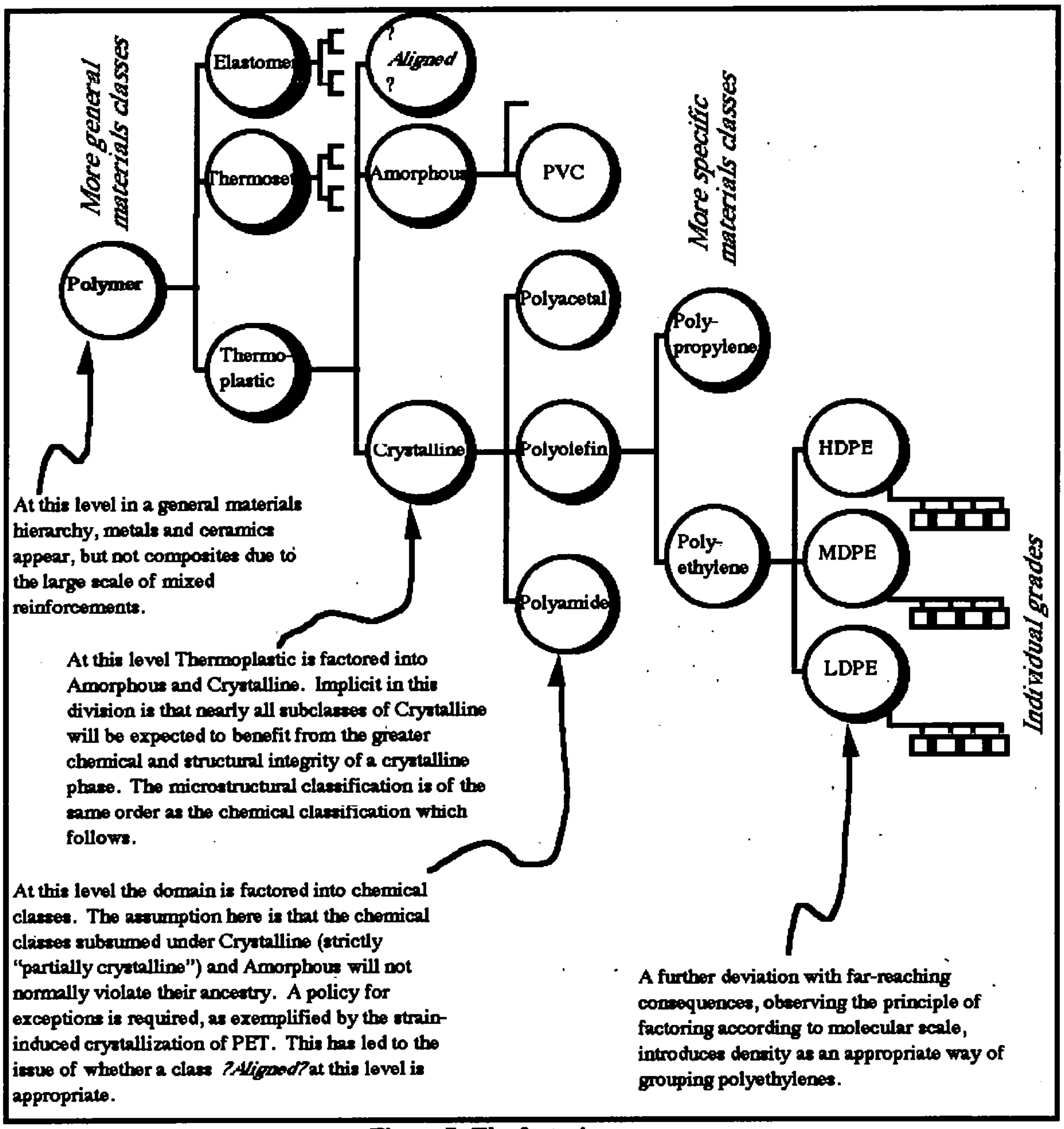
A user interface gives the domain expert access to the block of text for individual grades. The interface allows the user to annotate the grade with unclassified information in a piece-wise manner characteristic of prototyping. This interface would benefit from a Hypertext extension to the simpler text interface. Hypertext adds cross-referencing of key words and when the user selects one, the view automatically moves to the reference. Many other applications have proven Hypertext a very successfully browsing tool, eg HyperCard[79]. Although not considered an essential feature of the POISE system, it is viewed as being a potential future extension to the system. The hypertext facility could also provide cross-referencing to other POISE user interfaces providing immediate access to the knowledge in POISE on polymers and properties described in the text.

3.3 Evolution of POISE Architecture

3.3.1 Description of the Classification Architecture.

The hierarchical classification in POISE supports the management of domain information and generates abstractions. However, the nature of the domain complicates developing a classification because the principled concepts of similarity that provide an extendable classification do not always group similar engineering data useful for design. Without an acceptable principle for classification, and rather than enforcing a controversial classification, an alternative is to implement a classification that can change and evolve according to principles that are learnt from use. A philosophy of change compensates for a certain amount of absent knowledge still to be learnt.

An empirical decomposition of the domain aims to consistently classify like with like. The empirical approach considers existing record structures and documentary sources, such as the annotations from CAMPUS. Finding similarity within the domain is not a trivial exercise. Later, some tools are introduced to aid this process. Figure 7 shows a fragment of an initial classification. It illustrates an expanded portion of the polymer hierarchy together with some of the knowledge-domain arguments favouring the structure. The classifications of the domain, the classes between the polymer families and root class `Polymer`, are chosen according to principles of microstructural scale and composition, ie a domain principle suitable for extending the representation.



At this level in a general materials hierarchy, metals and ceramics appear, but not composites due to the large scale of mixed reinforcements.

At this level Thermoplastic is factored into Amorphous and Crystalline. Implicit in this division is that nearly all subclasses of Crystalline will be expected to benefit from the greater chemical and structural integrity of a crystalline phase. The microstructural classification is of the same order as the chemical classification which follows.

At this level the domain is factored into chemical classes. The assumption here is that the chemical classes subsumed under Crystalline (strictly "partially crystalline") and Amorphous will not normally violate their ancestry. A policy for exceptions is required, as exemplified by the strain-induced crystallization of PET. This has led to the issue of whether a class *?Aligned?* at this level is appropriate.

A further deviation with far-reaching consequences, observing the principle of factoring according to molecular scale, introduces density as an appropriate way of grouping polyethylenes.

Figure 7: The factoring process

The management of the classification, which is described next, automatically supports:

i. grade behaviour consistent with classification
ii. specification of grade implementation, including data structure,
iii. consistent placement within classification of grades acquired by data acquisition system
iv. generation and management of generalisation
v. abstracting default property specification from generalisation

whilst being able to coerce to a new classification as specified by the user.

Class–instance object–orientation supports this management already as follows: (i) Grades of the same class template define common abstract behaviour ensuring consistent properties (ii) and data structure. (iii)

Currently the placement of grades uses the unique polymer family name as a key to the polymer class, to ensure consistent placement of grades. Class–instance language do not generally support points (iv) and (v), which introduce abstract levels of representation besides specific grades and are discussed later in §3.4. Many class–instance language do not support schema evolution, which is the difficult task of coercing to a new classification.

The lack of a computable domain principle means there is no rule to automate classification. Only highly specialised chemical classes exist into which the grades are automatically placed. These families generally group alike grades but there is no guarantee so there may be exceptions where it is necessary for the user to manipulate the classification. The user will need to specify any other more general classes as well. The tools for supporting evolution of the class hierarchy under user control follows.

3.3.2 Creating the Hierarchical Classification

POISE adopts the class–instance paradigm to represent classification. The following sections discuss what specific functionality POISE requires from the class hierarchy for representing polymer classes, and the tools for managing the hierarchy from a domain, rather than software, point of view.

The behaviour of each polymer grade can change and must be modifiable at any time. It follows that the language implementing POISE will need to coerce object definitions at run-time as behaviours change. Changing object definitions at run time creates many problems for the stability of programs. If a language provides schema evolution, it must also provide consistency rules over change and provides a mechanism for handling instability.

POISE provides its own well-mannered mechanisms to govern object definition changes in the class hierarchy. From a domain viewpoint, these are the addition of new classes, the movement of class within the classification, and the movement of properties in and out of classes. All these issues are inter-related. A new class will involve all three. The movement of properties in an existing classification is considered first.

3.3.2.1 Adding and removing a property

A property is a partial template description that contributes to the complete template defined in a class. How the property achieves this is specific to the implementation language and not an issue here. Regardless of the implementation, the property will abide by rules of inheritance.

When adding a property to a class, POISE ensures the property is not declared more than once along the same line of inheritance. Besides the more specific declaration being redundant, if allowed by a class the repeated property would correspond to a repeated allocation of resources by the class in the grade's data structure.

If a specific polymer family specifies a property, and later it is found it applies to a more general class of polymers then the general class should add the property. The specific polymer family then removes the property, and instead inherits the property from the general class.

If all the subclasses of a superclass define a property, POISE infers that they should inherit the property from the superclass. The activity of moving a property defined in a subclass to its superclass is known in POISE as **promoting**. The inference to promote assumes a closed domain, eg initially the grades may all define yield strength, causing the promotion of the property to `Polymer`. Later an addition of a polymer that does not yield will invalidate the generality of the property.

The appropriate place for a property may be known before adding grades, so the promotion should also be under user control. Explicitly promoting each property is labour intensive, so an explicit denial of promotion is more useful than explicit promotion. Consequently, POISE does automatic promotion only when adding a property. The user can reverse or prevent this promotion by explicit removal of the property.

Removing a property from a general class of polymers has two possible consequences. Either the class and its subclasses no longer exhibit the property or each subclass adds the property so only the general class no longer exhibits the property.

Removing a property from a class is not the same as removing a property from the descriptions of all members of the class. The class only abstracts properties from its members. The members still express the property. When removing a property from a class, the more specific members of the class should automatically add the property. When a property moves from a superclass to its subclasses, the property is **demoted**. Demoting a property never affects the behaviour of grades.

Demoting is only possible if grades are in subclasses, since in most class–instance languages instances can not specialise their class description, ie cannot add properties to individual grades. When a property demotes from a class with instances (grades) the property cannot pass to the individual grades. In this case a destructive removal occurs. It is destructive because any values the grades maintain for the property will be lost, as their data structures will no longer support the property value.

Occasionally, the user wants to remove a property, rather than demote it. Such as when the user wishes to reverse an addition of a property. So both demotion and removal are supported. When removing a property from a superclass, it is possible that the user is unaware of the consequence to other subclasses. It is possible that the property was explicitly added to a subclass earlier and a promotion caused the inheritance of the property from a superclass. Removing the property from the superclass, rather than demoting the property, will conflict with the earlier addition of the property to the subclass. A similar case occurs when adding a property to a superclass that was earlier removed from a subclass. These actions are in conflict, but not directly reversed actions. In such cases, the user is notified.

For the user's convenience, the tool providing these facilities should visually convey an add as the reversal of a remove and a demotion as the reversal of a promotion. This eases the reversal of erroneous actions.

3.3.2.2 Adding abstract classes

Adding an empty polymer class as a specialisation of `Polymer` extends the classification. The class can then move to inherit from a class other than `Polymer`. Other classes can move to inherit from the new class. These functions allow the creation of a taxonomic classification.

The class needs a name, which as text, is a minimal visual representation of the class. If the class is a polymer family from CAMPUS, the name will link grades from CAMPUS to their class in POISE.

3.3.2.3 Moving classes

Moving a class is a major modification to the classification. A move changes the superclass-subclass relationships and the subsumed properties. With a change of superclass, the inheritance of properties changes. The effect on the class template is twofold: properties no longer inherited are added to the class, and the data structure extends to cater for the addition of newly inherited properties. Commonly inherited properties, between new and old superclass, do not cause change.

There are two kinds of move possible. The first moves a class and all its subclasses to a new superclass. The second moves a class but all its subclasses remain by inheriting directly from the old superclass. Moving the whole branch of the classification (the subclasses with a class) is thought to be better understood by a user.

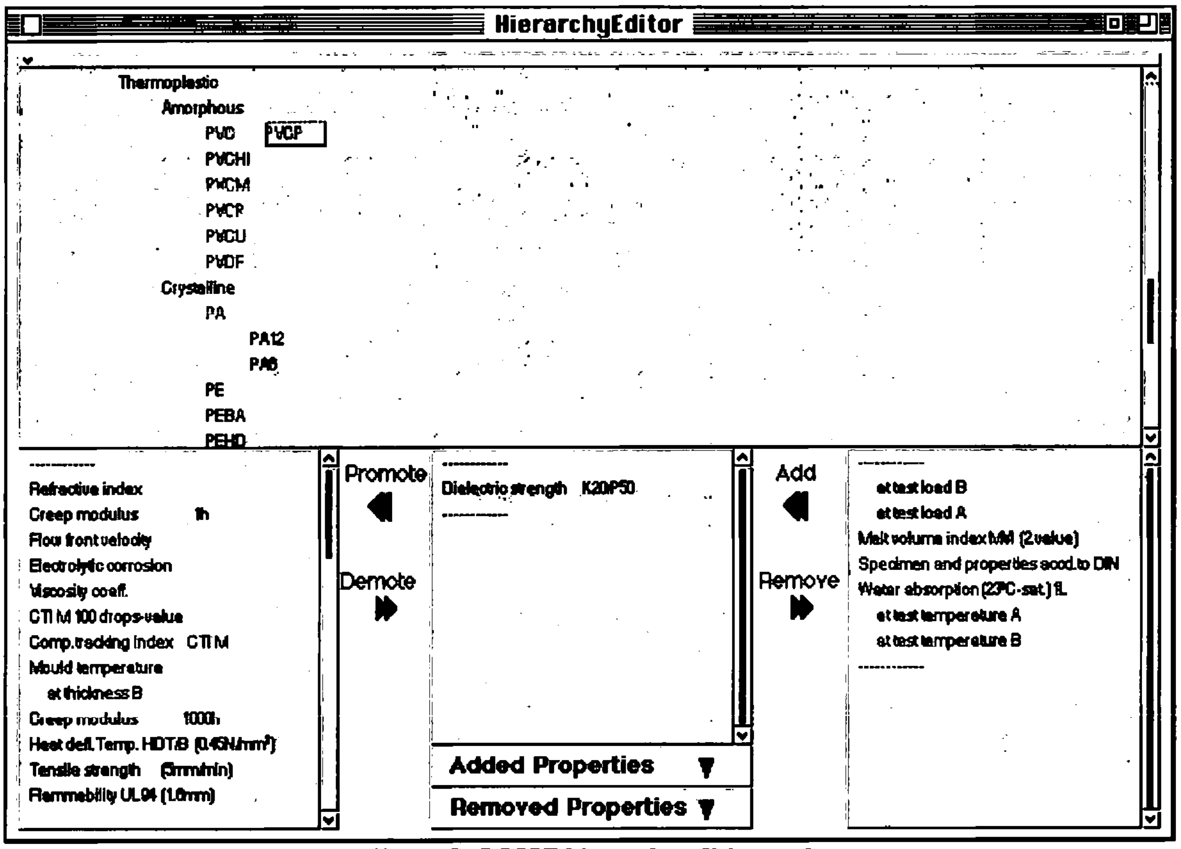
Moving whole branches prevents one inconsistency possible within a hierarchy: circular inheritance. Circular inheritance occurs when a superclass inherits from one of its own subclasses. By moving a whole branch and allowing the branch to inherit only from classes other than those in the branch, prevents circularity.

3.3.2.4 Merging classes

The naming of polymer families is not universal so there is the potential for the same real-world polymer family to be declared under two different names, eg Nylon and Polyamide. Merging two classes first requires both classes to share the same property structure. Normally this will be the union of the two property structures creating a more complete description. The next step coerces the structure of the instances (grades) to the unified structure.

3.3.3 The Interface

The hierarchy-editing interface is primarily a visual representation of the hierarchy. The simplest representation is a tab-indented list of the names of the classifications. The tabs are set according to the level in the hierarchy. The order of the list is such that any class inherits from the next class above it, which is printed at a lower level (one less tab space). An example is given in Figure 8 where PA6 and PA12 both inherit from PA, which is the next polymer class up the list at a lower level.



```
Thermoplastic
    Amorphous
        PVC    PVCP
        PVCHI
        PVCM
        PVCP
        PVCU
        PVDF
    Crystalline
        PA
            PA12
            PA6
        PE
        PEBA
        PEHD
```

```
------------
Refractive index
Creep modulus         1h
Flow front velocity
Electrolytic corrosion
Viscosity coeff.
CTI M 100 drops-value
Comp.tracking Index   CTI M
Mould temperature
   at thickness B
Creep modulus         1000h
Heat defl.Temp. HDT/B (0.45N/mm²)
Tensile strength   (5mm/min)
Flammability UL94 (1.6mm)
```

```
------------
Dielectric strength   K20/P50
------------
```

```
------------
   at test load B
   at test load A
Melt volume index MVI (2value)
Specimen and properties acod.to DIN
Water absorption (23°C-sat.) IL
   at test temperature A
   at test temperature B
------------
```

Figure 8: POISE hierarchy editing tool

Users select a class by choosing the appropriate line in the list with a pointer input-device (eg a 'mouse'). The view (an area displayed on a computer screen) then displays the information about properties of the class in the subviews below the hierarchy. From these subviews, the user can move properties around the hierarchy in an orderly manner.

3.3.4 User Interaction with Properties

Once the user has selected a class, three subviews are updated to display; the properties inherited by the class (the left view); the properties specialising the class (the centre view); and a view with all other properties, those considered not applicable (the right view). The right list will exclude the properties of orthogonal classes (§3.6.2) which can apply to any grade.

The editor keeps a record of all the property modification, for consistency checks. The interface checks each change and determine when the user requests conflicting actions. Pull-down lists marked "added-properties" and "removed-properties" display the history of actions on the selected class.

There are four dedicated buttons for manipulating the properties of the class; Add and Remove on the right, Promote and Demote on the left. The proximity of these buttons encourages the user's understanding of movements between the lists.

3.3.4.1 Adding a property

Properties listed in the pane on the right of Figure 8 are not subsumed by the selected class. The class could add these properties by selecting one and pressing the Add button next to the list. The lists are updated with the property removed from the right list and added to the centre list. The list in the centre pane is the properties declared specific to the selected class.

The editor checks all subclasses and recursively their subclasses, before adding a property to a class to see if any subclass has explicitly removed the property. If the property has been removed then inheriting the new property is a conflicting request. This raises a `Notifier` that provides the option to either abort the add, enforce the add despite the earlier remove (hence adding the property back via inheritance), or add the property to all subclasses except those in direct descendant from the class that had the property removed. This last option will cause all subclasses to have the property except the one that had the property removed.

After adding a property POISE analyses the complete hierarchy. If the property is common in all the subclasses of the selected class's superclass, (the subclasses will include the selected class) then the editor automatically promotes the property to the superclass. In which case, the property moves from the right list over to the left list in Figure 8. This process enriches the abstract classes and the hierarchy as a whole. If a record of the property removal exists for the superclass then this automatic promotion will be vetoed.

3.3.4.2 Removing a property

Removing a property, like adding, starts by selecting the property in the centre list and pressing the remove button. The property moves from the centre list to the right list. The editor first checks subclasses for any previous adds of the property that now rely on the inheritance from the current class. If a conflict occurs a `Notifier` opens with the following options: force the remove despite the previous add, abort the remove, or remove the property but add it back to the subclass that had add the property previously (like demotion, but only to the conflicting subclass).

Removing a property will eventually lead to loss of data. This has not yet occurred at this point in the procedure, but will happen later when the user accepts all changes to the hierarchy. Reversal is possible up to that point without loss. Any action providing the grades with the property later will maintain the data.

3.3.4.3 Adding and removing classes

The user can add a new class by selecting the menu bar (), and choosing the "Add Class" option. Defining a new class abstraction has two requirements: a name and a class from which to inherit. By selecting the menu, a field entry window opens for the user to type the name of the class. The class inherits by default from `Polymer`. The user can then add properties and move the new class.

3.3.4.4 Moving classes

Moving a class involves selecting it in the hierarchy and then designating a new superclass from which it inherits. A select and drag operation by the user can achieve this elegantly. The user selects and holds (keeps the mouse button depressed) a line containing the text of the class's name, then drags the mouse. As the mouse moves up with the text of the name, the classifications above are highlighted, conversely in the down direction. In Figure 8 PVCP can be seen over the classification PVC. When the editor highlights the new superclass, the user releases the mouse. If the move passes consistency checks then the class becomes a subclass of the new superclass. No changes occur if checks fail, eg releasing the class on one of its own subclasses. The editor updates the display.

3.3.4.5 Removing classes

By selecting a class then ordering the command via the menu, the user can remove classifications. The appropriate warnings are announced accordingly. The editor does not change the POISE classification, only the display, until the user commits all changes. All the changes can be aborted at any time, including class removals. Currently, aborting a single class removal requires aborting all modifications.

3.3.5 The Data Model underlying the Hierarchy Interface

A single change to the schema of the polymer data definition can affect from a single class containing tens of objects to every object in the schema, an order of thousands of objects. Often a user will not make one change but will have a number of modifications. Most of the changes will affect the same objects, eg removing a property from `Crystalline` and adding one to `Polypropylene`, both affect `Polypropylene`. It would be sensible then to use a batch processing technique rather than the interactive processing of each change.

Batch processing requires a description of all the changes to be made and then a single process to optimise by reduction and performs those changes in one step. A model, which is descriptively parallel to the polymer hierarchy, records changes entering the user interface and performs the consistency checks as described in the previous sections.

Two parallel hierarchies have two advantages over interacting directly with the polymer hierarchy. Firstly, the classes of the polymer hierarchy have instances that they must represent consistently at all times. So a single change requires as much coercion as many changes and, since it is more common to evoke many changes at once, batching the changes is more efficient. The parallel structure does not have to coerce instances. The second advantage is a separation of consistency checks and user interface protocols from the polymer hierarchy. These are properties of the parallel hierarchy. This leaves the polymer hierarchy a 'cleaner' structure for representing polymer behaviours.

The parallel structure, known in POISE as the `POISEHierarchyChanger`, performs the consistency checks and raises a `Notifier` when the user performs conflicting actions. It collects the changes from the editor interface until the user commits. The `POISEHierarchyChanger` only then updates the polymer hierarchy.

3.4 Levels of Representation

So far, the concept of the class imposes property descriptions on grades. Each class captures similar behaviour in a template of properties, common to all members. The hierarchy explicitly declares many levels of classification. Each level classifies a population into exclusive classes of similar grades.

The class template abstracts properties that all members of the class exhibit. These are the defining properties. Comparing the class template to a design specification can only establish if all members will meet the design requirements. For design, it is more useful to establish if any specific members might meet the requirements of a design. A similar logical statement is if no members meet the requirements. If this is false, it can be assumed a member might meet the requirements.

Two abstract inference rules determine if the requirements of a design are not satisfied. They do not determine if they are satisfied, but if they are true then the requirements are definitely not satisfied. The first rule for an unsatisfiable design is if there is a single property requirement not satisfied. This is true if a single property is not applicable to all members of a class, or the required value is not found in the class.

The second rule for an unsatisfiable design is if the union of all properties will not satisfy a property requirement, then no individual member will satisfy the requirement. The union of properties is a collection of every property descriptions exhibited by the grades and a union of the values for each property. The union of the values must infer if a given value is definitely not in the union, but does not need to infer if a given value is in the union.

This union of properties is a characteristic description of a class of grades. The union describes, as opposed to defines, the properties of a class. It is an abstraction capturing properties the class can at best satisfy.

The union is an optimistic abstraction of the members in a class. Logically, if the union satisfies any design rule then there is still a chance a member of the class can satisfy the rule. Additionally, if a member of a class satisfies a design rule then the union must also satisfy the design rule. Since the union can substitute for any member in a logical design rule, it is an abstract representation of those members. The union is an abstract polymer description.

The union of the values in an abstract polymer must infer a given value is definitely not in the union. A simple range can satisfy this criterion. If the given value is outside the range then the value is definitely not in the class. The range is a very course measure. Knowing the distribution of the values may give a measure of probability the value is in the class. Whatever the abstraction technique used, the abstract value is a function of the set of values from a population of grades. In an object-oriented system, this function can be specialised per property.

Each class abstracts a different population of polymers. A hierarchy organises these classes and the subclasses inherit defining properties from their superclasses. What is the relationship between the describing properties of subclasses and their superclasses? An inference mechanism similar to inheritance exists for inferring the properties of the abstract polymers that describe classes, but it applies in the opposite direction to inheritance. Rather than subclass inheriting from their superclass, the describing properties of superclasses "inherit" from their subclasses and the most specific subclasses "inherit" the descriptive properties of their instances.

The class template represents the defining properties of instances. Inheritance infers the class template by recursively appending the specific instance variables attributed by each class down the hierarchy. A template for the describing properties is quite different, but by reversing direction, the inheritance can also infer this template during abstraction. The descriptive templates from each subclass combine to create the template their superclass.

In a strict hierarchical classification, the grades need only be abstracted for their immediate classes, which produces an abstract polymer. These abstract polymers are further abstracted to produce the abstract polymer for their superclasses. The superclass abstracts the lesser abstract representations from its subclasses (eg `Nylon` abstracts all the specific nylon families, `Nylon66` and `Nylon12`, etc). Since the hierarchy is strict, a superclass's subclasses are mutually exclusive sets and there is no duplication of information. Mutually exclusive sets are easy to union, since the intersection is empty.

Abstracting grades for only their most specialised class makes it much easier when adding, or changing, a grade. POISE need only update the abstractions of the most specialised classes, ie the class of the grade. The more general classes then dynamically infer the update along the hierarchy.

When adding a grade, each known property-value adds to the corresponding set of property-values in the abstract polymer. When a property changes, the set must remove an occurrence of the old value and add the new value. The greatest effect occurs when a grade adds an orthogonal view. When this first occurs in a class, the abstract polymer description must also add an orthogonal view to represent the extra properties. Whatever property POISE provides to describe a grade of material, it must also be available to describe the abstract polymer.

3.5 Abstracting Knowledge Creating General Concepts

Abstracting reduces information, capturing the important concepts and discarding the details. At the initial stages of design, when the possible approaches to the problem are at their greatest and require assimilation of large quantities of information, these reductions are important. Assimilating abstractions rather than the large quantities of detailed information they capture makes the initial stages of design easier for the user.

In statistics, a normal distribution is an abstraction over a population of values, which reduces any number of values to only two numbers, a mean and a standard deviation. Comparing means and deviations is easier than population profiles. However, not all populations fit the character of a normal distribution. Care is needed not to lose important information in the process of reduction, so causing mis-representation.

The statistical quality of the values in polymer populations has not been the subject of a formal study. A study is appropriate only when given a quality population and an expert statistician determines an appropriate statistical method. Regardless of the method of reduction, the generation and management of the abstraction would still be a function over the population. Extensive statistical analysis for improving accuracy of abstract entities would probably be of little value while abstract entities are only used as approximate selection criterion. Consequently, POISE uses a simple representation of the values in polymer populations.

The abstraction POISE generates from a collection of single point data, properties with a single number value, is a histogram. A class `Histogram` instantiates objects which manage the set of single point data, and provide the behaviour necessary to display histograms and logically calculate with histograms. Although this technique is interesting in itself, the primary interest is in the management of abstracted general concepts.

A histogram is a set of consecutive intervals along an ordinate, each with a tally of occurrences within that interval. The `Histogram` constructs the occurrences of values falling into each interval from a population of values. The `Histogram` is usually viewed graphically as a series of bars on an ordinate, with the length of the bars proportional to the tally. The `Histogram` is an empirical reduction of a population rather than one based on the expert knowledge of the statistician.

3.5.1 Consistency between Histograms of the Same Property

An abstraction is best when it captures important information. Two objectives, the ability to compare and to query, judge whether an abstraction captures important information. For the Histogram, these two objectives depend largely on the ordinate intervals.

Comparing a property between two abstract polymers involves comparing `Histograms`. When comparing two `Histograms` the accuracy depends on the alignment. `Histograms` are said to be aligned if the intervals start at the same positions on the ordinate, and the intervals are of the same size. If two `Histograms` have different alignments, their comparison is visually distorted. When aligned, the tallies of individual bars are comparable.

Other visual issues affecting the accuracy of a comparison is the size of intervals. A `Histogram` with few large intervals has larger tallies. Consequently, the 'tally height' by 'interval width' area that a bar covers is larger, yet the data is the same as with a `Histogram` with many small intervals and low tallies. A `Histogram` with large tally seems visually to state that there is a number of members ('tally' of them) of the population at every point along the interval.

Whether large or small intervals should be used is a question of how accurate an answer is needed. If the interval is 1.0, and a query requires an answer of $\pm$ 0.5, then all members in an interval will satisfy the accuracy requirement and hence the tally is an appropriate response. Each query though has a different accuracy requirement.

The alternative is to derive an interval appropriate to the population. An interval too large creates one large tally. An interval too small creates a number of tallies equal to one. A visually appropriate interval is somewhere in between. Above all, the interval must be the same for all Histograms representing the same property. Both the distribution of the population and generally acceptable levels of accuracy for queries are qualities of each Property object.

Comparisons are always between two Histograms representing the same property. To ensure that the alignment is the same, POISE keeps the alignment as an attribute of each Property object. The attribute is a HistogramParameter object that POISE queries whenever it creates a new Histogram for a particular property. Once the alignment is set on a Histogram it is impossible to change the alignment. The Histogram only represents property abstraction. A more permanent source of knowledge is always the grades that make up the population.

Calculating suitable parameters for a particular property is not simple. Initially few values exist for a property and there is nothing to infer an alignment. The parameters can not be set until POISE achieves a suitable population. In the meantime, an ImmatureHistogram maintains the actual property values of the members of the population, rather than a set of interval-tally pairs. From these values, an ImmatureHistogram can calculate new interval-tally pairs for any alignment.

ImmatureHistogram depends on the HistogramParameter of the property they represent to ensure they all present the same alignment. The HistogramParameter maintains a reference to all the ImmatureHistogram. As the population of a given property grows, the HistogramParameter object recalculates the alignment using the record of values in all the ImmatureHistogram. Once the population for a property reaches a certain size, POISE deems the alignment accurate and the ImmatureHistograms mature. Each ImmatureHistogram coerces to the mature Histogram class using the latest alignment.

3.5.2 Consistency between Histograms and Populations

The general classes infer their Histograms from the Histograms of their subclasses. The Histograms merges by summing tallies for common intervals. These common intervals must align. For a given property, the alignment of the intervals in histograms must be consistent to support the recursive subsumption of histograms up the hierarchy.

The properties of more general abstract polymers are consistent with the population they subsume by dynamically merging histograms of the more specific abstract polymers. Each grade in the population is a member of only one specific polymer class. Each grade notifies its class of any change, which also represents the abstract polymer, so the whole hierarchy of abstract polymers maintain a consistent abstraction over all grades.

Finally, `Histograms` are polymorphic with the single-value properties of grades, because the median value in the Histogram answers any value-specific queries. Although this is not an accurate representation of an abstract value, it demonstrates the necessary polymorphic behaviour. The `Histogram` therefore satisfies all the objectives of a property-value describing an abstract polymer.

3.5.3 A Summary of Functionality for the Abstract Polymer

The following features have been identified:

- At the most specialised level of representation, collect the values for each property exhibited by members of the class.
- Reduce the populations of values into abstract `Histograms`.
- Maintain changes to individual grades and the addition of new grades.
- `Histograms` to be polymorphic with any grade's value and abstract polymers polymorphic with the grades.
- A subsumption mechanism that dynamically merges the populations of specialised classes to create the populations for the more generalised levels of representation.

The issue of polymorphism has multiple facets. In theory, it means the abstract polymer must respond to any message appropriate to any grade it subsumes. The abstract polymer subsumes many types of grades. The type of the response to the message must also match the type of response the grade would give.

The requirements for polymorphic behaviour and subsuming the populations for more generalised classes both suggest some kind of reversed inheritance of grade structure to cater for the wide range of property aggregations possible. Characteristics at the bottom, or specialised end of the hierarchy, inherit and merge to describe the top end of the hierarchy. This reversal reflects the bottom-up nature of generalisation, as opposed to the top-down nature of abstract property description imposed on grades.

From the domain viewpoint, the class defining the structure of the grades is the same concept as the abstract entity. If this is to be the case in the software model, the behaviour of the class needs an extension.

3.5.4 Extending Class Behaviour

The class has two roles within the POISE classification:

- As an abstract description, declaring behaviours and states for polymer grades.
- A domain representation of an abstract polymer.

With these two roles come a number of behaviours to maintain and manage the population of a class. The implementation of the two roles was found, on the whole, mutually exclusive. The same entity combines the roles because this is how the domain views them. It is also convenient to share the same hierarchical structure. It does not concur that the two roles relate functionally. As result, the class may be kept in the single hierarchy, while packaging the implementation of abstraction into a separate object. The class keeps this object as an attribute, and so indirectly keeping its hierarchical position. The object in turn provides the abstracting services for the class. The standard class–instance relation declares behaviours and states for polymer grades, while with minimal extension to classes, a separate object can extend the classes behaviour to include services for generalising and abstracting properties amongst grades of the class.

3.6 Orthogonal Property Classification

An appropriate classification distinguishes differences and similarities between classes with properties that depend solely due to differences in materials. Therefore, an appropriate polymer classification will describe properties that vary solely due to differences in material under test and are independent of other design perspectives. Grades inherit and specialise these properties. Many properties though are not solely dependent on a generic material perspective, requiring additional extensions to the grade's inherited template for these properties. These extensions are rarely unique to a single grade, but describe orthogonal classes of grades applicable to all. Properties depending on extreme geometry, like films, is an example. The nature of a material can prevent rolling into a film. Grades that can be blown into films forms an orthogonal class. Properties describing this extreme geometry should specialise the template of these grades.

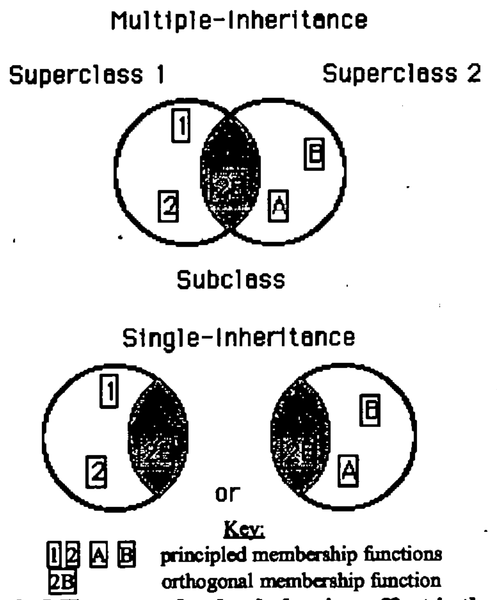


Figure 9: MI vs. standard subclassing effect is the same

The problem is how to extend the description of grades without losing the taxonomic separation. Orthogonal descriptions require a mechanism for subsuming different representations independent of taxonomic classification. Extending grade descriptions by subclassing and through multiple inheritance mixes the orthogonal classes with the taxonomic classes. Figure 9 shows multiple inheritance and subclassing in class based languages have the same effect on classification. Superclass 1 and Superclass 2 are orthogonal descriptions of the same domain. Both represent all grades but a grade can only belong to one immediate class. Each classification principle carves up the membership to subclasses differently and in conflict. In single inheritance, taxonomic classification chooses one or the other. Consider the membership function $\boxed{2B}$ which includes grades with both property $\boxed{2}$ and property $\boxed{B}$. If Superclass 1 classifies with single inheritance on the principles of Superclass 2, the members of $\boxed{2B}$ can form a subclass. With multiple-inheritance, the intersection subclass forms with members of $\boxed{2B}$. Once members of $\boxed{2B}$ separate from members of $\boxed{2}$ (or $\boxed{B}$) by subclassing, it is impossible to further classify grades with property $\boxed{2}$ (or $\boxed{B}$).

In an extreme case, it is possible the properties are very specific and further subclassing is not necessary. In an extendable classification, the only way to ensure no further subclassing would be to have a unique class for each grade (ie remove the class based premise), since the potential to further classify always exists in a class with two different entities.

Mixing the representation of grades and the classification in conjunction with multiple inheritance dilutes each taxonomic perspective. All the different orthogonal taxonomies mix, combining classes of all permutations. This dilutes the significance of classes that form individual perspectives. Variance within a perspective is independent of other perspectives and is therefore a valuable path to investigate design variation.

To illustrate the concerns of dilution and extensibility, take an example of a film made from a `Nylon` grade. Multiple inheritance would define classes `Film` and `Nylon` then a subclass `Nylon-Film`. Now consider if taxonomy classifies `Nylons` into `Nylon66` and `Nylon12`. An exclusive class of `Nylon66` and `Nylon12` is not possible since some are members inherit from `Nylon-Film` and some only from `Nylon`. The common solution is to remove `Nylon-Films` and add `Nyon-66-Films` and `Nylon-12-films`. Then consider if the user classifies on `Nylon-Fibres`. What happens if some `Nylons` fabricate both films and fibre? Are there then `Nylon-66`, `Nylon-66-Film`, `Nylon-66-Fibre` and `Nylon-66-Fibre-Film`? Such a structure complicates simple queries on `Nylon` properties. Moreover, the permutations do not stop here with this small example. Consequently the taxonomic classification loses structure on introducing multiple inheritance. What class a grade is finally a member of is no longer determinable from a top down search unless its form is known.

In conclusion, POISE requires the definition of polymer classes that do not completely describe the grades belonging to the class. Some describe the taxonomic decomposition. Others act as orthogonal classes describing properties templates for completing the description of the grades. Taxonomy requires these descriptions to remain separate. The implementation of grades requires the two to combine.

3.6.1 *Composite Structure for Orthogonal Descriptors*

Objects can alternatively share behaviours through a composite structure. Considered polymers as a general polymer description, "General Poly", that owns a specific polymer description, "Film properties", as illustrated by Figure 10.

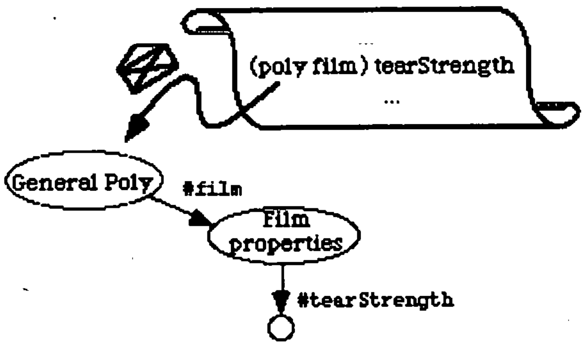


Figure 10: Composite template sharing

To access specific details — the film property '`tearStrength`' — the polymer, "*`poly`*", is first asked for the set of film properties with the message '`film`'. The message `film` might achieve this by returning the object `FilmProperties`. This object is then the receiver of the message `tearStrength`. There are the following problems with this representation:

- A `GeneralPolymer` is not a film. It does not respond as an object with film properties, ie it is not polymorphic with `FilmProperty`. Senders, objects that evoke the behaviour, must know where to send the message `#film`.
- `FilmProperty` is not a polymer. If asked for the name of the polymer (a property of `GeneralPolymer`) it can not respond. `FilmProperty` is not polymorphic with polymers.
- The general polymer has no control over the property access of film properties. After the message `#film` any message may be sent to the `FilmProperty` object and these messages do not pass through the `GeneralPolymer`'s interface. Hence there is a hole in the encapsulation around the polymer entity. It is an association rather than an aggregation. `FilmProperty` is not a separate part of a polymer, only part of the description of one whole entity.

3.6.2 *Management of Orthogonal Descriptors*

Polymer classes provide a template that describes the behaviour of grades. An orthogonal descriptor extends the descriptions of grades. They too contribute a template of behaviour. In class–instance languages, classes are templates of behaviour, so it is common to find classes representing orthogonal descriptors.

POISE extends the function of the polymer classes, and similarly classes representing orthogonal descriptors. The descriptions can add and remove properties. They have their own populations of grades and can abstract generalisations over those grades. Orthogonal classes differ from polymer classes in that the orthogonal classes are only meaningful in the context of a particular materials class. For example, asking `Films` for their density is only meaningful within the domain of polymers. The query is really the density of polymer films.

The query is more meaningful directed at the Polymer class. Efforts were made to extend the abstracting mechanism within the Polymer hierarchy rather than extending it to the orthogonal descriptors (§3.6.4).

3.6.3 Adding Orthogonal Descriptors to Grades

There is no such thing as a 'new' orthogonal entity. All grades are first classified taxonomically then classed orthogonally. Orthogonal descriptions extend existing grades.

Adding an orthogonal description to a grade extends the grade's existing data model by using the class template of the orthogonal descriptor. This class provides a data structure and methods, which extend the grade's behaviour. The problem of how to merge the templates, one from the polymer hierarchy and any number from orthogonal descriptors, under the same object interface remains.

3.6.4 Abstraction of POISE Knowledge in Orthogonal Descriptors

Although taxonomic classes of polymers do not define orthogonal properties as part of their template, individual grade entities do exhibit orthogonal properties. Although an orthogonal property is not a defining requirement of membership to the taxonomic classes, each property is a valid description of a subset of members in the taxonomic domain. Any property particular to a subset is a valid generalisation of a taxonomic class when the property itself fundamentally limits the domain. For example, take the property of minimum film thickness. The distribution of minimum film thickness over all Polyethylene is a valid generalisation of Polyethylene. The property does not exist for all kinds of Polyethylene but where the property does exist, its variance is a measure of Polyethylene in general.

When a grade adds an orthogonal extension, there is no need to affect the polymer class abstractions until a grade adds an orthogonal property-value. The grade then notifies the class abstraction mechanism as it does with all property updates. It is up to the abstraction mechanism to recognise that the property is orthogonal and to cater for the new property by extending the abstraction's own structure with the same orthogonal template. The abstraction mechanism maintains a separate population for each orthogonal view. The user can selectively view abstractions using an orthogonal perspective (ie select the orthogonal subset within the taxonomic class).

3.7 User Interrogation

3.7.1 Histogram Visualisation: The Comparator

Histograms are very easy to display and make good tools for conveying the abstract knowledge of a general polymer. The Comparator allows the display of any combination of Histograms of the same property in a resizable window.

From the `Comparator`, the user can select individual intervals or a single interval of any size across the whole `Comparator`. The tool searches for grades finding those in the selected interval(s). Displayed as a group, the user can browse them individually or even as a `Histogram` against different properties. This allows the user to dynamically specify any arbitrary abstraction besides those in the taxonomic classification.

There are a large number of groups of polymers in POISE, each with different property abstractions. The `Comparator` provides intelligent options to the user for specifying the abstraction to display as a `Histogram`. The initial selection might include all possible polymer classes, orthogonal classes or properties. If a property is chosen, then the `Comparator` limits further selection to classes supporting the selected property. This limits both the classifications and the orthogonal perspectives available for selection. Once a class is chosen only the properties in that class become available. The `Comparator` displays many abstractions but only against one property. If the user specifies many class abstractions, the only properties available are those common to all the abstractions. For this kind of consistency, the `Comparator` accesses global resources such as the `Polymer` hierarchy, a list of all properties and, more specifically, maintains a reference to the abstract polymer, not just the histogram displayed.

`Comparators` can scale the display of histograms, changing the visible size of the intervals. This is not only an issue of conveniently fitting windows on a computer screen, but to offset the effects perception has on information. Histograms can be deceptive depending on the interval size chosen. By modifying the scale a user can visually bias the interval size and tally for each comparison. Comparing properties that the user perceives as significant can be made larger. Although the technique is far from quantitative, it does provide a quick qualitative feeling as to whether polymer selection is satisfying design requirements. The technique also identifies polymers not belonging in their assumed groupings.

Scaling displayed views is challenging. The axis changes in only one dimension, keeping constant space for labelling which relates to the text size of the numbers displayed rather than the size of the view. The view labels more numbers as the axis gets longer and less labels if space is limited. Given an arbitrary maxim and minimum and the space for display, the view determines the numbers on the axis. Even the type of number on the axis affects clarity. The view avoids rational numbers preferring integers.

3.8 Database Management

POISE contains a database management system. Through intelligent management (classification and organisation) and through presentation with graphical interfaces, POISE conveys the meaning of new data informatively to the user, hence transforming data into information. Both the data entering POISE and the organisation transforming the data must **persist**. A classical program persists as an application stored on secondary storage and when the user commands the computer's operating system, it loads into primary memory. The user supplies the data each time the program activates and returns some result, so neither the data nor the result persist. Alternatively, the user may store input data on a file. Through iterative changes in the data on the file, the user changes the nature of the process. The result may also contribute to the data in the input file. A database is an application that manages files of persistent data.

One source of data in POISE is the descriptions of polymer grades. These already reside in files of a simple format for easy management although POISE also receives complex data from the user, especially on data structuring and organisation. Through the classification environment within POISE, the user adds value to the raw data by virtue of the structuring and behaviour associated with objects making up the classifications of a domain. The simple format of the polymer data is incapable of recording all the information within POISE.

POISE distinguishes between the format for archiving and exchanging data within or between industries (suppliers and users) and its own internal representation. For data exchange, the important factor in the format is its simplicity and universal acceptance. One example is the DIF structure of binary relations (§3.1.1), and although other formats exist, third party applications can convert data between simple formats. For the internal representation of the complex objects within POISE, the important factor for storage is an expressive structure capable of representing the diversity of these objects. The expressiveness is contrary to simplicity, hence distinguishes between complex internal and simple external representations. For persistence of the internal representation, a range of existing database management systems (DBMS) were examined. Although the internal representation is the focus for the remaining discussion, some of the issues apply equally to integrating data from external sources.

The search for a storage system starts with two extremes. Storage system functionality ranges from a fully-fledged object–oriented DBMS (OODBMS), to Smalltalk's simplistic file structures for exporting objects without any management. The initial preference was for an 'off-the-shelf' commercially available OODBMS, which provides program support, and convention. POISE though places high demands on even the most expressive data description language. The alternative; a file storage using the native data description language of the client, Smalltalk, required major extra development to incorporate a suitable management strategy.

3.8.1 Data Store vs. Database Management

General purpose OODBMS process behaviours remotely in a database server environment. The environment, primarily of the class–instance paradigm, features disk-storage management of a class hierarchy. Objects on the database are all instances of these classes. A mapping between classes in a client language (eg Smalltalk) to classes on the server allows copies of an object to cross from one environment to another. Typically, a mapping keeps to simple classes like numbers and strings. A totally alien language (ie not Smalltalk) describes the classes on the server.

The description language of a database is similar to a programming language. The class description includes the storage requirements of objects, manages versioning and schema evolution control, and interacts with system administration, such as memory management and security. The main differences are due to multi-user databases access. Policies for locking objects and accepting changes are designed to make each user's activities atomic. Most computing activities include a number of intermediate states, which other activities could corrupt or misinterpret. Computing in a multi-user environment requires composing all computing into atomic transactions. Within each transaction, the objects involved are locked and once the transaction is complete the final state of each object accepted and the objects unlocked for the next transaction.

Shortcomings of general-purpose OODBMS systems, their language's modelling power in particular, are criticised in an earlier paper[80] relating this management to domain modelling.

Data storage relies on the client's language for object definition and manipulation. The server for data storage stores only the state of an object. When read, it moves the state to a new object in the client environment and the client processes the behaviours. The responsibility of object integrity (valid states within an object) lies with the processing of behaviour within clients. An object store exports the object structure to a foreign environment. The protocols changing the states in one client can be different to those in another client; they may not be consistent and semantics can differ for different clients. In an OODBMS, clients do not access the state of the object. The OODBMS centralises consistent object behaviour.

A particular advantage of a simple object store over these large-scale OODBMS is the ability to reduce data administration overheads. For example, a multi-user object store is possible by locking the record of the object on the server until the client using the object finishes, but is unnecessary for a single user system. OODBMS tend to come as multi-user systems with the mechanisms tightly integrated in the server's language as a standard feature.

3.8.2 Evolution in a DBMS

The consequence of object definitions (classes) within the OODBMS is that they must meet POISE's requirements on schema evolution. An OODBMS using the same manipulative object model and environment as Smalltalk would suffice. Unfortunately, the memory management in primary memory, which makes Smalltalk's manipulative model possible, seems to compromise efficient transaction processing in a secondary storage system. Transaction processing is a benchmark, which relational database management systems use to judge a DBMS performance. Consequently no DBMS are as manipulative as Smalltalk, and contemporary DBMS cannot support the needs of POISE. The main problem is that they do not provide mechanisms for manipulating the schema while the databases is in use. Without schema evolution, adding and removing properties from polymer classes is not possible and the classification can not be re-engineered.

A grade's tendency to evolve even complicates the specification of a data store. All changes to the property structures of classes in POISE require automatic respect by the storage mechanism, which stores instances of those classes. A data store that returns an out-of-date data structure for an object in POISE is useless, so the storage mechanism must be able to migrate such out-of-date structures to the current versions in primary memory. For this, some interpretation of each data structure must be recorded, like the class template records the structure of instances in primary memory.

Schema evolution causes the same problems for object-storage as it does for languages. The problems exasperate when multiple users access the same evolving objects. Besides changing the semantics of objects, which has far-reaching effects in clients, evolving in a multi-user database starts a transaction that locks all active instances of the class and its subclasses. A change to the root class would lock the whole database. OODBMS systems, even those with supposed schema-evolution provisions like Gemstone[81], do not allow evolution when there are active instances.

The schema evolution in Gemstone lets the application programmer evolve class definitions before creating any instances. This schema evolution is a development function. The migration of a class definition from a client into the server is not a runtime function of Gemstone. Therefore, it is impossible to automatically add an object of a class not already on the server without programmer intervention. Some storage mechanisms provide a general object-storage process capable of managing new data types, but such a process is liable to fail for complex objects with 'global' references and cyclic paths in their structure. BOSS, see §4.5.3, is one mechanism with little program intervention that attempts to avoid such pitfalls.

Evolution of the DBMS objects is not the only problem. Whereas demonstrating such mechanisms as delegation within Smalltalk is possible, it does not necessarily hold that similar mechanisms are possible in other object–oriented environments. None of the current DBMS support delegation, only static class hierarchies. Other complex mechanisms like orthogonal descriptors would also be an issue.

3.8.3 Database Interaction for Memory Management

The data manipulation languages describing a transaction in a relational DBMS limits the manipulation to an explicit fixed scope of data, applying the rules of manipulation to each tuple in a relational table. This ensures that after processing a unit of data, the transaction is complete for that part of the data. The locking and committing of data is implicit in the structure of the transaction, not part of the manipulation language. Object–oriented languages manipulate data through messages, not transactions. A long-lived message generates many shorter messages, which in turn generate shorter messages. The end of a message can be conditional, thereby depend on an object to change state, which inturn may depend on other messages. The data accessed or the time taken can not limit the goal of a message. The message is therefore not like a transaction in a relational database.

Management of limited primary memory is an important feature of database management systems. All transactions occur in primary memory. When they finish, the database commits the changes to persistent secondary memory. The definition of a transaction has a consequence on the utilisation of primary memory. Defining a transaction as a message to an object requires careful consideration. Committing the state of an object to secondary memory after each message sent to the object will not affect the logic of any messages, but it is inefficient to commit objects subject to further change. Concurrently, changes in the primary memory are susceptible to loss until the database commits the object to secondary memory where it is persistent. Even in single-user systems, transactions affect the management of primary memory and the integrity of objects.

Something must trigger the database to commit objects to secondary storage. Since the applications using the objects evoke changes, they must also trigger the database. The efficiency of memory utility and object integrity depends on the regular commitment of objects by applications. Most OODBMS and data stores require an explicit interaction to activate and release objects. This means the application is constantly communicating with the database for each transaction.

In an object–oriented language messages follow an implicit path. They may potentially access any object, and the same path may repeatedly cause message to the same object. A message is not a transaction since it does not identify which objects to lock. A transaction could span many messages to the same object. In object–oriented languages there is no implicit structure to define the scope of a transaction. The protocols, which construct the path of processing, must explicitly encode transactions.

Persistent objects now differ from volatile objects, which do not necessitate the specification of processes into atomic transactions. The use of the persistent object is also a property of the protocols, which must define when the objects are locked and committed. The storage characteristics of objects are not a property of domain entities in the POISE application but must provide pre-processing and post-processing behaviours implementing the storage characteristics of the object on an object store. Even if objects are otherwise polymorphic in their behaviour, the activation of these behaviours within transactions with the object destroys the benefits of polymorphism. Protocols must distinguish stored object from memory resident objects that do not require this activity.

The situation where users of the objects do not see the interactions with the object store or OODBMS is termed 'transparent' database access. Although an object stores with only single-user access does not define transactions for multiple-users, primary memory is still a limited resource requiring the paging of objects back and forth from secondary memory. So both systems require transparent access, but OODBMS must also define effective transactions for multiple users.

3.8.4 Multiple Interfaces

The specification of a DBMS handling a national source of information is quite different to a DBMS for an individual designer. Different worlds of information pose different demands on an OODBMS. Differences include the number of users, security, data integrity, and the size and scope of information resulting in different access mechanisms and data models, all of which affect transaction management. POISE on the other hand does not need to identify where the data has come from. It is quite possible for POISE to access many different DBMS, and extend to add new DBMS at a later point in time.

Heterogeneous database management systems[82] manage the access to many different types of database using different data models. Since each DBMS has a different interface protocol, the main purpose of this manager is to provide a single consistent interface protocol for data manipulation in any of the databases. An object–oriented model is popular for this interface. Although the data model and manipulation language of the individual databases limits the behaviour of these objects, the objects provide a consistent, polymorphic interface across many different (hidden) access mechanisms. A management system for heterogeneous databases provides a uniform object interface as a **proxy** conveying transactions to database objects.

A third party handles proxy votes as though the owners of the vote had voted themselves. Similarly, a database proxy receives messages and, from the viewpoint of the message sender, the resulting actions are as if the intended object received the message. A message to a proxy triggers the memory management system within the DBMS. Within the ensuing communications the message transports somehow (depending on the particular DBMS) to the stored object, locks the object fro other processes and, when the message is complete, unlocks the object. The user is unaware of the nature of an object's storage so the DBMS becomes transparent to the users of persistent objects.

Current heterogeneous systems rely on each message to be an atomic transaction. Extending transactions beyond a single message requires a standardisation of locking semantics at the interface of the database objects. The standard makes the transaction locking independent of the actual database being accessed but protocols still explicitly specify the locking. In which case the proxy is no longer transparent.

3.8.5 Summarising the Storage for POISE

Both the persistence of user data and the interchange of data between users require some kind of database management. The management for both is quite different. If client languages support a transparent proxy access mechanisms, then different database management utilities can independently implement each of these requirements. The following two lists identify the two different storage requirements of POISE. The first list covers the management of private information gathered by a single user. The second list covers global data shared by many users:

1) Private single-user data
   a) Minimal transaction management. Lifetime of transaction only subject to primary memory management.
   b) Object behaviour integrity guaranteed by a single client.
   c) Complex highly structured data storage model for supporting any arbitrarily complex Smalltalk object composition.
   d) The evolution of object structures during runtime, trans-migrating class definitions between client and server and coercing objects of old versions to new versions within the client.
2) Global multi-user data
   a) A consistent protocol for accessing many heterogeneous databases.
   b) Object integrity guaranteed by individual server databases (usually read only with respect to Poise).
   c) Use simple data structure as the common denominator of many different client applications.
   d) Client process independent of server transaction management, eg security, locking and versioning.

A transparent interface between application and storage management is common to both storage mechanisms. A transparent interface is not only consistent with access to native objects but is also consistent between private and global stored objects. The interface, in this case a Smalltalk specific implementation of an object proxy, provides different services for each two types of storage:

- Transparent access and updates, managing transactions subject to local memory conditions.
- A translation from generic protocols to specific protocols of heterogeneous DBMS.

In order to simplify the development of POISE, the issues explicitly on global multi-user data were not considered further, ie issues on access and management of transactions within heterogeneous databases.

Others[82] have already addressed many of these issues. POISE instead imports external data and represents it with the private data. Issues considering transparent transaction management apply to both storage systems.

The behavioural complexity of objects within POISE and their tendency to evolve puts the representation beyond even the most advanced commercial OODBMS. Commercial OODBMS focus on the other issues, integrity and transaction management. For the private single-user data in POISE the objectives are more limited, and more powerfully focused on representation, than the objectives of a general-purpose management system. Consequently, even the simplest of data stores are as capable as the advanced OODBMS at representing and evolving POISE objects. Although data stores are less sophisticated, the client language implements most of them. Hence the data store gains the language's manipulation capabilities, including schema evolution, and it is possible to develop the management principles of these data stores.

In conclusion, the storage needs of POISE involves an investigation into the suitability for development of available data stores and an investigation into transparent transaction management. Initially the investigation of proxies for transparent interaction was separate to the development of a data store. As the issues involving the representation of complex object-relations became clear, the proxy was found useful as a representation for the relationship between objects on the database. The proxy became an integrated part of the database schema, see §4.5.2.

3.9 Summarising the Schema of POISE

Each aspect of the conceptual description, or loose schema, summarised below in Table 5, maps to a design specification. The design specification phrases the requirements in terms of object-oriented concepts, types of objects, classes, protocols classification and inheritance. During this research, the specification was implemented in Smalltalk to produce the POISE application. The following chapter discusses in detail the programming issues that arise from implementing this specification in this class-instance language.

Summary Requirements of POISE Schema	*Design Specification*
Adding value to acquired relational data through re-representation in a classification hierarchy with property inheritance.	Re-modelling relational data using principles of object–orientation. For example, extending the relational description of grades with default behaviours from their class.
Extend the description of the polymer domain: a) Define new engineering properties. b) Add new computations for describing and interpreting the semantics of an engineering property. c) Add new properties to abstract polymer descriptions. d) Add new classifications over the domain of polymers for abstracting similarity and generalising properties.	Instantiate new components of the schema: An object represents each engineering property. Compiling new protocol objects and assigning them to an engineering property. Assign the protocols from the engineering properties to classes of grades. Add new classes, modifying the inheritance structure, ensuring consistency and updating dependent class behaviours.
Support consistent evolution of the schema by the user: a) Define a language for describing engineering properties. b) Assign domain properties to classes of grades and manipulation of domain organisation c) Make schema changes persistent.	Develop specialised user interfaces and inference engines. Modify the native language programming tools and compiler for protocols defined in the context of a domain property, not a specific class. Reflect the hierarchical structure and inheritance rules within the classification through a graphical interface. Develop an object storage capable of recording the objects in the polymer classification and all engineering property's behaviours
Support both taxonomic classification and orthogonal classification for representation extending beyond the classification principle.	Investigate dynamic behaviour sharing to support orthogonal representation in a class–instance languages.
Support the design process through generalising the properties of domain classifications providing abstract levels of representation.	Develop an inference engine to generalise and abstract properties from a class and present these properties as an extension to the behaviours of the class. Again, complex behaviour sharing is an issue.
Provides expressive visualisation of generalised polymer properties.	Develop a user interface to explore the abstractions in the evolving classification.
Support persistence of design data and the complex evolving knowledge representations.	Develop an interface to secondary storage that is transparent, thereby creating the illusion of persistence of knowledge between sessions.

Table 5: Mapping requirements to specification.

Chapter 4 Implementation

The POISE schema presents a number of challenges to the class–instance language Smalltalk. The implementation of the POISE application successfully satisfied most of the requirements, like the data acquisition, and the user interface design, which fit well into the class–instance paradigm. This chapter focuses on the main features of the implementation and explains how each of these features was a challenge to the class–instance paradigm in general and Smalltalk in particular. This identifies the limits of this approach to taxonomic representation of materials information. For a complete summary of the POISE application, Chapter 5 describes the functionality from the user's perspective.

The main features of the implementation found challenging are summarised as follows:

Problem	**Feature Affected**	**Description**
Mix-in object behaviour	Orthogonal property classification	Combining polymer taxonomic description with orthogonal description in a single lexical unit.
	Extending behaviour for user interrogation	Protocols of domain entity with visual modelling protocols
	Extending behaviour for transparent memory management	Object behaviour + Database access and management protocols
Mix-in class behaviour	Implementation of levels of representation	Normal class behaviour plus abstraction of property generalisations and population maintenance
Composition of class template	Properties as class descriptors	Class behaviour encapsulated into shared Property Objects
Delegation	Design as a dynamic composition of shared behaviour	Behaviour dependent on a context of objects.

Table 6: Challenging problems to solve

Mix-in[83] of both object and class behaviour (§4.2.1) and delegation, in Table 6, are all solved using the same mechanism for enhancing the behaviour sharing capabilities of Smalltalk. The "composition of the class template" captures the existing class evolution behaviour in Smalltalk and packages it into an abstract object. This abstract object creates a new approach to change in a class–instance paradigm.

All these problems result somewhat from the limits to sharing between objects in standard Smalltalk. Fortunately, the underlying object model is flexible enough to implement a programmable extension to the standard sharing mechanism of inheritance.

4.1 Sharing in Smalltalk.

Chapter 2 introduced two types of sharing of protocols and behaviours. Instances inherit protocols from classes, and the classes inherit from superclasses, thus protocols are shared. The protocols describe the behaviours of objects. The behaviour is a composition of other behaviours (procedural abstraction) shared from other objects, evoked through messages.

The limits of the standard forms of sharing in Smalltalk are the subject of this next section, starting with the special inheritance relationship between the instance and the class.

4.1.1 The Smalltalk Object Model

Inheritance between classes and between a class and an instance is quite different, unlike in prototype languages where implicit sharing is uniform. The reason for the difference is that the instance is a specialisation of an object that optimises processing.

Instances hold specific differences and inherit abstract behaviours from their classes. The typical abstract behaviour is a protocol providing instructions on how to do something. The typical specific behaviour is a relationship with another object, or more simply, an attribute. Two instances can share the same abstract protocol, but behave differently due to specific differences in attributes.

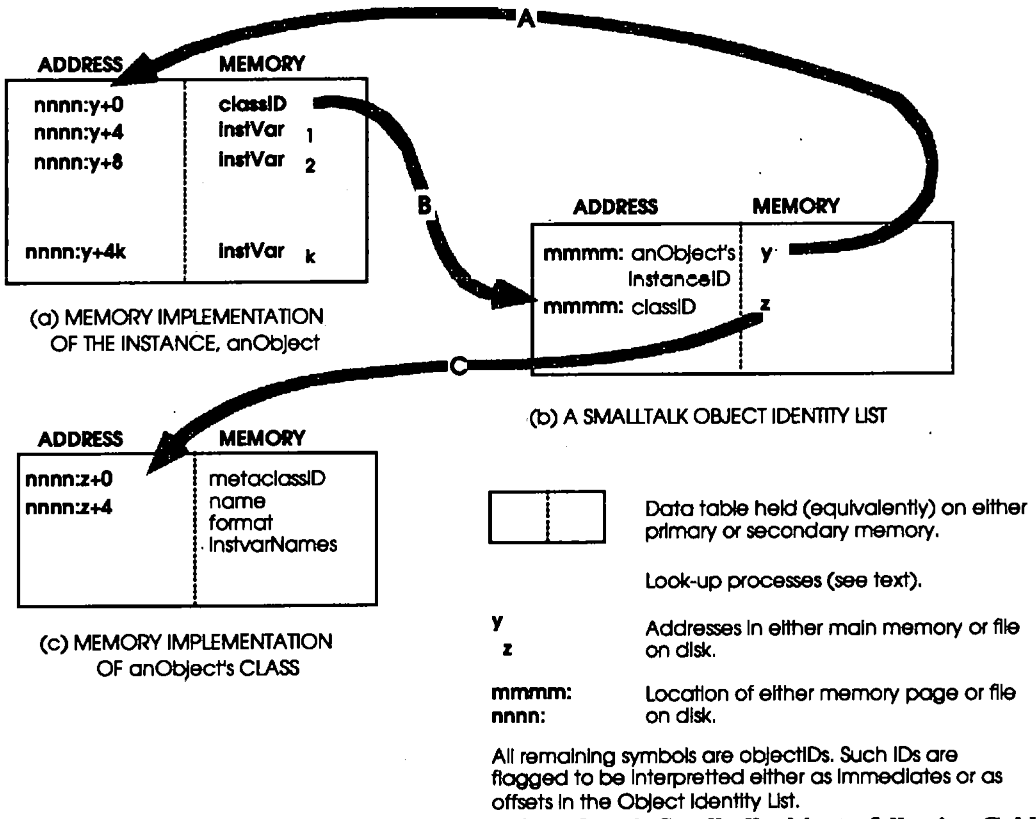

A
ADDRESS
MEMORY
nnnn:y+0
nnnn:y+4
nnnn:y+8
nnnn:y+4k
classID
instVar 1
instVar 2
instVar k
B
(a) MEMORY IMPLEMENTATION OF THE INSTANCE, anObject
ADDRESS
MEMORY
mmmm: anObject's InstanceID
mmmm: classID
y
z
C
(b) A SMALLTALK OBJECT IDENTITY LIST
ADDRESS
MEMORY
nnnn:z+0
nnnn:z+4
metaclassID
name
format
instvarNames
(c) MEMORY IMPLEMENTATION OF anObject's CLASS
Data table held (equivalently) on either primary or secondary memory.
Look-up processes (see text).
y
z
Addresses in either main memory or file on disk.
mmmm:
nnnn:
Location of either memory page or file on disk.
All remaining symbols are objectIDs. Such IDs are flagged to be interpretted either as immediates or as offsets in the Object Identity List.

Figure 11: Canonical memory representation of static Smalltalk objects, following Goldberg*

A contract exists between instances and the class behaviours. A class template contracts names to each attribute. The class protocols generate specific behaviour by referring to the attributes by the contracted name. Instances provide the necessary memory 'slots' to store attributes. The class template optimises the contract by defining a specific ordering of the names. This ordering generates a record structure for representing instances and an index for behaviours to directly access the attributes in instance records.

Figure 11 shows the basis of the object storage model for the static, record-part of object storage. The model can apply to both primary and secondary memory, as will be seen in §4.5.10.

In Smalltalk, the physical model of the object is an indexed table of *objectIDs*. The table represents named relations to other objects. These relations are collectively called *instance variables*. Every object has an `ID`, including integers, characters, any other instance, class or metaclass. The collection of object representations is known as Object Memory.

In order to access the complete description of a stored Smalltalk object (referenced as *anObject*) the following activities occur:

1) Look up *anObject's ID* in the Object Identity List, as Figure 11 (b), and obtains the address of the record (y) giving the location of *anObject's* record in Object Memory.
2) The first word of the record is an *objectID* to a class, the `classID`. Obtain this also by looking up the Object Identity List.
3) The remaining task is to obtain the location (z) of the `Class` record in Object Memory. The `Class` record includes (storage of) the data definition of *anObject*. The word marked `format` in this stored data definition determines the allocation size of *anObject*'s record. The data definition also includes `instVarNames`. These are the names of the locations identified in Figure 11 (a) as `instVar1` to `instVark`. Protocols are compiled to reference directly by index.

The effectiveness of this model as an object will become clearr after introducing how the object receives a message, locates a protocol and evaluates behaviour.

4.1.2 The Class as an Object

The `Class` in Smalltalk is an object constructed like all other objects in the language. It behaves like a `Class` because it inherits those 'class like' behaviours from the class `'Class'`. One of the behaviours a class inherits is the ability to generate other objects, their instances. This is a primitive behaviour (encoded in the Smalltalk kernel §4.1.3) that directly accesses the second instance variables of the class record, called the `format`, and must contain an integer. This integer describes the number of slots for the instance.

The new instance keeps a reference to the generating object (the class) in the instance's `classID` slot; ie the instance's class. For this instance to work as an object, its class must meet two other criteria. The class has another class object (or *nil*) in the third instance variable as the superclass, and a `MethodDictionary` object (a hash list of protocols) in the fourth instance variable.

These three instance variables are the basic requirement for getting a class to function as an instance template. Other requirements are necessary for the `Class` to function as expected within the Smalltalk environment, especially to compile protocols, but no more necessary to get an instance functioning in the Smalltalk environment.

4.1.3 Methods as Protocol Objects

All classes keep a `MethodDictionary` in the fourth instance variable. The `MethodDictionary` holds a hashed list of symbols (special strings of characters that the environment ensures have unique object id's) known as *selectors*. When an instance receives a message, it searches the dictionary for a selector matching the message. If the search does not find the selector, the search continues in the `MethodDictionary` of the superclass. Each selector in the `MethodDictionary` maps to a `CompiledMethod`, and by finding a selector, the `MethodDictionary` returns the `CompiledMethod`.

`CompiledMethod` are the essence of Smalltalk protocols. They are objects with code in their first instance variable and data in others. The code, at its most basic, is a byte array. The *virtual machine**, the Smalltalk kernel, interprets the meaning of the bytes in the array at runtime. The code may reference three different types of variables. As mentioned earlier, instance variables of the receiver are one type. The code also accesses temporary variables for the duration of method execution, which includes the receiver bound in source code to the name *`self`*. Finally, there are the global variables, which are a reference to any unique object by the method. A global reference does not change with each receiver. The reference is not empathetic.

Protocols in the object–oriented language compose primarily of message sends. Smalltalk behaviours evoke these messages through a sequence of pseudo-code commands encoded in the method representing the protocol. The method stores the selector in one of the method's instance variables and refers to it by index. First, a command pushes the receiver of the message, some *`objectID`* available to the method, onto a stack. A second command referring to the index of the selector causes the virtual machine to 'send the message'. For this the virtual machine gets the receiver's *`objectID`* off the stack, locates the record, locates the receiver's class record, locates the method dictionary, and starts the searching for the selector's *`objectID`* in the method dictionary.

Not all protocols result in message sends. There are a select number of primitive protocols, which call functions in the virtual machine, operate on simple objects like numbers and byte arrays, and communicate with the underlying operating system.

* On each computer platform there is a program running which dynamically compiles, caches, and executes Smalltalk™ code. This program is called the "*Virtual Machine*", as it emulates a hardware device which would directly execute Smalltalk™ code.

Some preparation occurs before evaluating a method, which involves creating a context in which the code can reference by index all the objects available to the method. The next section describes the objects used represent a single execution of a method.

4.1.4 The Process: Message Sends, Look-ups, and Patterns

Messages are sent causing classes to find protocols, that evaluate causing further message sends, hence creating a pattern of processing. An object called the `Process` follows this pattern through the Smalltalk environment. The `Process` represents, at any given time, a sequence of incomplete method evaluations, with the last one being currently evaluated. An object called the `Context` represents each incomplete method evaluation.

Once a message send finds a method, the virtual machine creates a `Context`. The `Context` immediately records the receiver, the method located, arguments sent with the message, and the `Context` which sent the message. The pattern returns to the sending `Context` once evaluation of the method is complete. During the evaluation of a method, the `Context` also maintains a stack (mentioned earlier) and the state of any temporary variables generated during the method evaluation. All these objects are accessible by the code evaluating the method by index.

Memory management of `Context` objects is special. A stack space, a sequence of equally sized records in memory, is reserved which adds and removes context objects efficiently. Since the virtual machine generates context objects, their data structure is beyond change by the user. Conceptually the structure fits the standard object model and can be viewed and manipulated by Smalltalk code and tools. One tool of importance is the exception handling system that will be introduced later §4.2.6.1.

4.1.5 Summarising Behaviour Sharing

The empathy between the receiver and the located protocol has been highlighted. The protocol binds `self` to the receiver and indexes the instance variable locations of the receiver. It is essential that all accesses to instance variable locations are consistent across every protocol a receiver shares. The class manages this consistency by naming the variables. These names link to the indexes when compiling methods for the class. It is therefore impossible for a receiver to empathise with methods from classes other than those from which the receiver inherits. This is only true of method accessing instance variables.

If a representation requires an instance to share behaviours with another class (that is not inherited), then the question that arises from the above point is whether an instance is the proper representation. Instance variables are unique properties of instances of a class and are meaningless to any other methods but those of the class. Methods do not need to access instance variables to represent knowledge, only to represent knowledge specific to instances of a class. To demonstrate this point and to place the behaviour sharing of prototyping languages into perspective, which does not classify and represents knowledge without instance variables, an experiment was carried out that specialised the meta-class, the definition of the class object.

Objects in prototype languages do not have instance variables, all attributes are stored as protocols (ie as a method references a global variable), always returning the same object. Attributes differ between child and parent by overloading the name of the protocol, just as methods are overloaded in the class hierarchy. A child with a protocol of the same name as the parent will never implicitly exhibit the parent's behaviour. A specialisation of the class object defines a class that is an instance of itself. When the "instance" receives a message, the *classID* points to itself, and the method search starts in the instance's own fourth instance variable; containing a method dictionary. This configuration successfully models prototypes in Smalltalk. The prototype defines its parent by referring to another prototype in the third instance variable (the superclass slot). If a selector is not matched in the prototype's own dictionary, the search passes to the second according to Smalltalk's standard look-up process.

The second point to highlight is that the look-up for methods (protocols) is a strict process. The virtual machine dictates what happens between the point a method in a context evaluates the code to send a message and, the point it creates a new context and evaluation starts. The critical part of this process is the locating of the method by the recursive search from class to superclass to superclass is well defined for all objects. Since the scope of the search is well defined, it is seen as a uniform and seamless *interface* to the objects. Only those selectors indexed in the method dictionaries will provide a key to access the receiver.

Since the look-up mechanism is a strict process, the prototype objects described do not give explicit delegation capabilities, and instances of a normal class can not explicitly look-up messages. All behaviour sharing in Smalltalk must locate the desired protocol using the standard look-up process. For extending the sharing capabilities for grades of polymers, an alternative scheme was developed from the "Encapsulator", §4.2.2, for enhancing message passing in Smalltalk.

4.2 Enhancing Message Passing in Smalltalk

As an introduction to the enhanced message passing that provides a solution to a number of problems in POISE, the issue of orthogonal classification descriptions over grades is re-introduced.

4.2.1 Mix-in Object Behaviour for Orthogonal Descriptions

Orthogonal classifications describe properties of grades. Objects describing the behaviour of other objects in a class–instance language are implemented as classes. Objects described by classes are instances. Therefore, the orthogonal classifications are classes and the grades of material are instances.

The relationship between class and instance is a one to one ordinal relation. This would suggest that more than one class could not describe a grade, thereby excluding orthogonal descriptions. 'Mix-in' object behaviour is the description given for objects that 'mix-in' the behaviours from multiple sources.

Mix-in object behaviour introduces an additional perspective to the description of an instance. While still only perceiving a single object, a mix-in object behaves as if it is an instance of two separate classes. Two interfaces seen as one, two implementations and one unified set of relationships.

The proposal is to place one instance from an orthogonal class and an instance from the polymer class under a common interface. The interface is the point at which messages are received. When receiving messages both instances are searched until a method is found. The interface would exhibit a concatenation of the instance's behaviours whilst each instance remains an inheriting member of their separate classes.

The common interface does not contribute any behaviour but does affect the pattern of message passing. Hence, the proposal is an enhancement of the message passing mechanism in Smalltalk.

4.2.2 The Encapsulator

Pascoe's `Encapsulators`[84] is a mechanism for controlling and extending the messaging powers of Smalltalk. It is not the first mechanism[3] of its kind, but has two advantages. First, it extends the behaviour of individual objects. `Encapsulators` are classes of objects that have the behaviour of isolating (encapsulating) another object. The degree of isolation and control on access depends on the implementation in different subclasses of `Encapsulators`. The second advantage is that it uses a standard Smalltalk kernel, so any Smalltalk environment can apply the implementation.

Pascoe solves two common operating system problems using the `Encapsulator`: queuing requests using a monitor[85] philosophy, and committing transactions in an atomic step. Both of these problems occur when managing message evaluation in multi-user operating systems to ensure the evaluation of one message does not conflict with the evaluation of a second message from a second concurrent user. These solutions have more relevance to the secondary storage requirements of POISE, see §4.5.

Pascoe applies the Encapsulator to the Model concept in Smalltalk's Model-View-Controller (MVC) scheme for user interaction. The Model is the source of information displayed in the interface. The View generates the screen display. The Controller handles users input. The Model has two roles. One role is as an **entity** in the conceptual schema of an application, which is the subject of the object's purpose. The second role is as a servant of the View. The user modifies the Model through hardware inputs interpreted by the Controller and the model reports changes visually through the View. The Controller and View send a range of messages to the Model for this purpose. Usually the Model mixes these message protocols with the protocols that describe the entity's role as part of some knowledge schema.

The two perspectives of the Model cannot separate through decomposition since the View requires notification when the application's entity part changes state. When the state of the entity changes the View updates. This requires extensive integration within the implementation of the entity, catching potential changes to the states.

The Encapsulator focuses on the interface of the object rather than the implementation by identifying message protocols known to modify the entity's state. All messages destined to the entity are sent instead to an Encapsulator. The Encapsulator then redirects the message to a dedicated object representing the entity. If the Encapsulator suspects the message changes the entity then it notifies the View after the entity evaluates the message. The implementation of the entity is free of the View's needs. The View displays only information it obtains from the interface of the entity (the Encapsulator).

The application of the Encapsulator in the MVC is a better object–oriented model because the object being viewed is independent of objects representing the View. If the View needs extra methods for combining information from the object on display, then they are separately implemented by another object, the Model. The Encapsulator combines the object on display with the Model under one object interface. Not only are the objects on display represented separately from the Model but can be classified separately. This separates the evolution of the MVC from the evolution of the objects they View. The separate MVC hierarchy can then classify interfaces of different generality, general MVC and specific MVC for viewing the same objects.

In POISE separate classifications represent different perspectives of a polymer grade. An Encapsulator can combine taxonomic classes and orthogonal classes of individual grades. Unfortunately, through common coding practices it was possible to break an Encapsulator accidentally. This would compromise the semantics of a polymer composed of different perspectives, and a major improvement was suggested, which also resulted in simplifying the `Encapsulator` as published by Pascoe.

4.2.3 Message Passing in Smalltalk

The `Encapsulator` extends the object interface through an exception in the message passing mechanism in Smalltalk. This exception invokes when none of the protocols, in any superclass of the receiver, matches a message's selector. The condition is not well defined by the class–instances paradigm and each language must specify some mechanism to handle the condition or ensure the condition never arises. In Smalltalk, the particular mechanism provides a very useful way for extending the behaviours shared between objects.

When messages are not understood by a receiver in Smalltalk, it generally opens a `Notifier` informing the user, usually the program developer, of the 'type' error. The mechanism generating this behaviour is not specific, as it involves many program controllable steps. After the failure of the first message, the Smalltalk virtual machine automatically sends a second message to the receiver. This message, called '`doesNotUnderstand: aMessage`', is a behaviour of all objects. The general `doesNotUnderstand` protocol, residing in the class `Object`, creates the `Notifier` for the user.

Like any other protocol, the `doesNotUnderstand` protocol can change. For example, rather than opening a `Notifier`, the failed message could be sent elsewhere. Consider object **A** sending a message `M` to object **B**. If object **B** does not have a protocol for `M`, **B** receives the `doesNotUnderstand` message. The `doesNotUnderstand` protocol for **B** does not open a `Notifier`, but instead sends the same message `M` to a third object **C**. Any message object **A** sends to object **B** can bind with either the protocols of object **B** or **C**. From the viewpoint of object **A**, the one interface at object **B** presents a subsumption of behaviours from object **B** and **C**.

Initially object **C** may be thought of as a proxy to the client object **B** and the message passing from **B** to **C** as delegation. Strictly, for delegation object **C** needs empathy for the behaviours of object **B**. Since the re-direction from **B** to **C** is a normal message send, this is not delegation. Object **B** shares the behaviours of object **C**, not the protocols. Empathy is not necessary for orthogonal descriptions and this extension of the interface is sufficient.

Changing the protocol for `doesNotUnderstand` in a class affects all instances inheriting from the class. Orthogonal descriptions require an extension of the interface per object, not per class. The `Encapsulator` provides an interface extension per object.

Pascoe's description of the implementation of the `Encapsulator` contains a flaw and involves some unnecessary modifications to the existing Smalltalk environment. The flaw allows access to the hidden object without passing through `Encapsulator`. The sender of messages can avoid the flaw if it knows the receiver is an `Encapsulator`. Requiring the sender to know the nature of the receiver is paramount to requiring the sender to know the receiver's implementation, ruining some benefits of encapsulation. Expecting POISE to assume any polymer could be a composition of orthogonal descriptions was unacceptable. The `Enhancer` is an `Encapsulator` that aims to solve this problem.

4.2.4 The Enhancer

The `Enhancer`[86] updates, simplifies and generalises the `Encapsulator`. The `Enhancer` takes a useful tool for combining the behaviour of two object for a specifically designed purpose, and creates a general enhancement to the messaging mechanism in Smalltalk–80.

Creating an object that exhibits the behaviours of another object it hides, while contributing its own behaviour is still the aim. In addition, the `Enhancer` attempts to do this as transparently as possible. Transparent means a sender will not be able to identify the composition of objects generating the behaviour, and only see a single object. The variable '*self*' is the main reason why the `Encapsulator` fails to achieve this objective. When a message binds to a protocol, the variable *self*, common in code, binds to the receiver. Although the `Encapsulator` initially receives the message, it re-sends the message to the hidden object. The hidden object is now the receiver and binds to *self*. Although this binding prevents empathy between the `Encapsulator` and the hidden object, it has a more serious consequence when the protocol finishes.

When a protocol finishes, unless otherwise specified by the programmer, it returns the reference to the object bound to *self*. Unless the `Encapsulator` intervenes, the variable *self* passes back to the sender. The hidden object, supposedly encapsulated, by default returns to the sender without the `Encapsulator`. A common programming practise in Smalltalk worsens the problem. Cascading messages sends the next message to the object returning from the previous message, often expecting it to be the same receiver.

An example illustrating the problem is a grade as an `Encapsulator` hiding a number of orthogonal parts. The `Encapsulator` passes any message it receives to each of the parts in turn until the message binds to one of them. A cascade of messages to a polymer could result in the first message binding to one of the orthogonal part of the polymer's description, and the behaviour returns that part to the sender. The cascade causes a second message to be sent to the returned object, in this case the part that responded to the previous message in the cascade, the message is not sent to the Encapsulator. Unless the cascade accesses only protocols from one part of the polymer, a message will not bind correctly.

4.2.5 Implementing the Enhancer

Unlike previous attempts to generalise the doesNotUnderstand mechanism[84] the Enhancer attempts to merge seamlessly with the Smalltalk object model with minimal disruption to the standard Smalltalk environment. With the exception of a few development tools (debuggers), the Enhancer is undetectable from an equivalent object inheriting solely from a single class hierarchy (ie based on the standard object model).

```
doesNotUnderstand: aMessage
"This method is a behaviour specific to Enhancers.  The doesNotUnderstand message is
automatically sent by the Smalltalk virtual machine when a method cannot be found to match the
message name.
The receiver, referenced by self, is therefore a variety of Enhancer.
aMessage is an object describing the message send which has been intercepted by the receiver.
aMessage comprises the selector (i.e. message name) and accompanying arguments."
"Temporary variables
hiddenObject...................Initially, the object hidden by the receiver.
me.............................Will be assigned the receiver.
answer.........................To store the result of aMessage."
        | hiddenObject me answer |
"Initial assignments."
        me := self.
        hiddenObject := me privateEnhancedObject.
"privateEnhancedObject is the retrieval operation specific to a particular subclass of Enhancer. that
the receiver inherits. "
        hiddenObject primBecome: me.
        "Swapping the receiver and the hidden object, so as to 'open the door' to the hidden object.
'me' is now a reference to the hidden object"
        [answer := me
                    perform: aMessage selector
                    withArguments: aMessage arguments]
"Executes the behaviour associated with message send described by aMessage."
        valueNowOrOnUnwindDo: [me primBecome: hiddenObject] .
"Regardless of the behaviour executed when aMessage is performed, the hidden object and me are
swapped back again.  The door is closed on the hidden object, no matter the outcome."
        ^answer
"The result is returned upon successful execution."
```

Figure 12: Message redirection for Enhancer

Consider an object without behaviours except the doesNotUnderstand behaviour of Figure 12 and one other named 'privateEnhancedObject'.

Any message that was sent to this Enhancer object (with the exception of privateEnhancedObject) would evoke the doesNotUnderstand message. The hidden object exchanges places with the Enhancer and the message is re-evoked but with the hidden object as the receiver. Upon completing, the exchange is reversed.

The only behavioural difference between an object hidden by an Enhancer and the hidden object on its own is the Enhancer will respond with the hidden object if the message privateEnhancedObject is sent. The semantics of this message would appear to be returning a copy of '*self*', the receiver. Since it is unlikely that the message privateEnhancedObject will have any other semantic meaning, this is a minor difference in behaviour

The advantage of the Enhancer is not an absence of behaviour. Semantically the Enhancer, as above, contributes nothing to object modelling. The Enhancer though differs in its implementation. The Enhancer is the sole reference to the hidden object. This allows the hidden object to change in a similar way to the message become:. The similarity is that when the hidden object changes, all owners of the Enhancer will experience the behaviour of a new object:

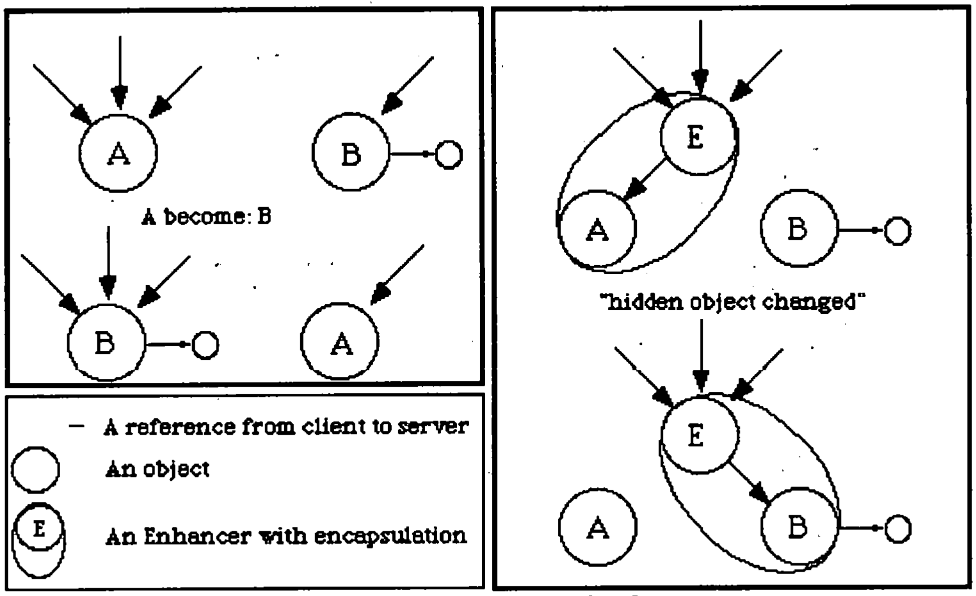

A
B
A become: B
B
A
E
A
B
"hidden object changed"
E
A
B
– A reference from client to server
An object
E
An Enhancer with encapsulation

Figure 13: Schema of Enhancer

In Figure 13, the Enhancer (E) can easily change its reference from one object (A) to another (B). This simple use of the Enhancer can help strengthen the encapsulation of all objects in Smalltalk. Currently any object can be the argument in a become: message. Without permission from either the object or any of the objects referencing the object, the object can be replaced by another. For the objects referencing, the change is an unauthorised change in state. With the Enhancer the become: primitive (§4.1.3) can be removed from general object behaviour, placed only in parts of the system necessary (eg to coerce instances during schema evolution). The Enhancer is then available for specific applications on individual objects that need the flexibility in changing object identity. For this, the Enhancer will keep its own private behaviour containing the become: primitive.

The Enhancer, as above, is an empty shell, into which each application writes a subclass with a different kind of behavioural extension. One application is the orthogonal description of grades.

4.2.6 *Implementing Orthogonal Descriptions of Polymer*

The primary description of grades is from taxonomic classes. For inappropriate properties, the POISE schema calls for orthogonal descriptions. An orthogonal polymer description is a modular extension of behaviour for individual grades. A class represents the orthogonal polymer description and each instance is an extension which individual grades may arbitrarily assign as part of their description. The grade distributes its description across many different instances. An Enhancer passes messages to each descriptive part until the message finally binds, thereby constructing the grade under one object interface.

A variant of the Enhancer combines the interface of two or more objects. The behaviours of these objects are independent of each other and pre-defined by their corresponding classes. This Enhancer creates the perception of a single object that combines the behaviours of the hidden objects. This Enhancer is a subclass called a CompositeEnhancer.

4.2.6.1 CompositeEnhancer

The CompositeEnhancer is like 'multiple inheritance on a per-object basis', or mix-ins. Multiple inheritance allows a single class template to inherit from more than one other class template. The CompositeEnhancer dynamically merges the interfaces of two (or more) objects without an official declaration of a class to unite the behaviours.

The CompositeEnhancer does not hide a single object but an ordered collection of objects. Upon receiving each message, the Enhancer iterates through each of the objects in order until the message binds satisfactorily. The iteration and testing for satisfaction are message pre-processing functions, functions quite specific to the multiple-hidden object nature of this subclass. The doesNotUnderstand: behaviour for the CompositeEnhancer requires re-implementation. To simplify the analysis of this behaviour, the following example, Figure 14, only comments on the new aspects of the protocol:

```
doesNotUnderstand: aMessage

"First I resend aMessage to the first object in my components. Any message that my
does not understand will be caught and sent to the next component until either it is answered or
have gone through all my components with the current
"1.     Set a temporary pointer named receiver to the first
        | receiver |
        receiver := 1.
"2.     Set up the exception handler to pass message to next object"
        ^Object messageNotUnderstoodSignal
        handle:
        [:ex |
"4.     A message has been sent during evaluation of the do: block that was not understood.
        exception, ex, occured in the context object 'initialContext'. iff the message not
        was aMessage sent in the do: block context below (equal to 'ex handlerContext'),
        there are still objects to pass aMessage to, then increment the pointer and restart the
        block"
        ex initialContext sender sender == ex
              handlerContext & (receiver == composite
              size) not
        ifTrue:
              [receiver := receiver + 1.
              ex restart]
"5.     Else reject this exception handler.  The signal will continue as if this handler did not
              ifFalse: [ex reject] ]
        do:
"3.     composite is an instance variable for access to an ordered collection  protential message
        receivers.  The perform:withArguments: sends a message as described by the
        the attributes 'selector' and 'arguements' of aMessage.
              [(composite at: receiver)
                    perform: aMessage selector
              withArguments: aMessagearguments].
```

Figure 14: Message redirection for Composite Enhancer

The CompositeEnhancer relies on the hidden objects supporting the standard Smalltalk behaviour for the doesNotUnderstand: message. The standard behaviour raises a signal, named messageNotUnderstood, as follows.

A CompositeEnhancer receiving a message will evoke its own doesNotUnderstand: method, in Figure 14. Under most circumstances, the do: block evokes and the first object in the OrderedCollection, named 'composite', receives the same message sent to the Enhancer, *aMessage*. If the message is understood the response returns. The Enhancer then appears to have the behaviour of the first object in 'composite'.

If *aMessage* was **not** understood by the first receiver in *composite* then the Smalltalk virtual machine decrees that this receiver will receive the message doesNotUnderstand instead. The first receiver is not an Enhancer but a standard Smalltalk object with the standard Smalltalk behaviour for the doesNotUnderstand: message, which raises a *signal*. This *signal* searches past contexts (§4.1.4) and finds the *handler* in the Enhancer's doesNotUnderstand context which evaluated the perform message. The *handler* is an object that holds the code described as step 4 and 5 in Figure 14. This code, unlike step 3 has not yet been evoked, despite the ordering in the source code. Now the *signal* tells the *handler* to evoke steps 4 and 5.

Getting the message, *aMessage*, to an appropriate receiver is the only purpose of the Enhancer. The standard Smalltalk behaviour for doesNotUnderstand might evoke for other reasons, at any time, because of another object not handling a sent message. There is no guarantee the cause of the signal is due to the attempt by the Enhancer to match *aMessage* to a receiver of the composite. This must be tested explicitly. This test examines the contexts created between re-sending *aMessage* and raising the signal.

For clarity, the context evaluating the do: block is the '*doContext*', which returns to a '*handlerContext*', the context that results from the whole handle:do: message. The 'sender' of the *doContext* is the *handlerContext*. In the *doContext* the perform: message creates another new context. This third context has the *doContext* as a sender. A chain of contexts is thus described: handler-do-new. If the receiver of the new context understands *aMessage*, the context will evaluate the protocol found for *aMessage*. If *aMessage* is not understood, the new context will evaluate the doesNotUnderstand protocol, which raises a signal.

'Raising a signal' is a message to a *signal* object. The *signal* creates an *exception* object. The *signal* passes the *exception* the current context (the *initialContext*) from which the *exception* can obtain the chain of parent contexts of the current process, ie the history of message sends leading up to the signalling. This includes the context that raises the signal, a *doesNotUnderstandContext*. The chain of contexts also includes the *handlerContext*. The receiver in the *handlerContext* is a *signal* object (see step 2 of Figure 14). The role of the *exception* is to search for *handlerContexts* and match their *signal* with the *signal* raised. The handler block, held as an argument in the *handlerContext*, then activates. The *exception* passes to the handler block through the argument **ex**.

In the handler block, there is a condition the Enhancer checks. This check determines if the *signal* is a consequence of a message the Enhancer sent and not some other object. Through the message initialContext to the *exception* (the **ex** argument), the handler block accesses the context raising the doesNotUnderstand signal. For the condition to be true, the sender of the *initialContext* must be executing the handler "do" block, and its sender must be the *handlerContext*. The *exception* determined this context when the *signal* was raised. If the message handlerContext to the *exception* matches the *initialContext*'s sender's sender, then the Enhancer must have given rise to the *signal*.

After ensuring that there are still objects in composite that have not received *aMessage*, the index increments and the do-block is evaluated again after removing (known as unwinding) all the redundant contexts down to the *handlerContext*. The do-block evaluates for each receiver until the doesNotUnderstand message and signal are no longer triggered, ie the message binds correctly. A receiver which binds will shadow the remaining objects in 'composite'. Any other objects also satisfying the message are not given the opportunity to express their behaviour.

The consequence of the first-object failing with *aMessage* and instead a second-object responding is the merging of two behaviours under a single interface, the CompositeEnhancer. To clients the CompositeEnhancer is a union of two or more object types. The example given dictates a particular rule for behaviour sharing in the intersection of the object types, so higher ordered objects override completely any object lower in the ordering.

4.2.6.2 CompositeEnhancer for supporting orthogonal descriptions.

The CompositeEnhancer is a single object interface. As such, it is identifiable as a single object, not a collection. It subsumes the behaviour from a number of other objects, not through inheritance but by delegating messages. Many objects explicitly subsume the behaviours of others through message passing, but the CompositeEnhancer does this implicitly. The types of the objects the CompositeEnhancer subsumes are unknown. Throughout the life of the CompositeEnhancer new subsumptions dynamically resolve as the objects themselves change. In addition, the subsumption is different for each CompositeEnhancer. The description of behaviour is per-object.

Resolving behaviour subsumption of orthogonal descriptions is simple since the orthogonality implies any intersection between the parts should be empty. The ordering within the `CompositeEnhancer` of objects has no consequence on sharing across objects with orthogonal properties. A `CompositeEnhancer` representing a grade can assign the component objects, for example, an instance from one general polymer description and one or more instances from orthogonal descriptions, in any order.

Even objects representing orthogonal descriptions of grades are not truly orthogonal in POISE. They all inherit from `Object`. An example of a common property inherited from `Object` demonstrates the susceptibility of the `CompositeEnhancer` to the ordering of objects. The property `hash` is a primitive behaviour that returns a unique integer for every object. It is important when placing and locating an object in a hash-table. Consider a `CompositeEnhancer` in a hash-table subsuming *`object-1`* first then *`object-2`*. When the `CompositeEnhancer` receives the `hash` message, it passes the message to *`object-1`*. The look-up of the selector `hash` starts in *`object-1`*'s class, down the super classes and locates the primitive behaviour in `Object`, returning an integer unique to *`object-1`*. That integer is used to place the `CompositeEnhancer` in the hash-table. If *`object-1`* and *`object-2`* where to swap places in the `CompositeEnhancer`, consider what the behaviour of `hash` is now. The `CompositeEnhancer` receives the message but now *`object-2`* receives the message first, passes it to *`object-2`*'s class, superclasses and finds the same primitive in `Object`, but this time *`object-2`* is the receiver. A different integer number is returned. A different integer number means the `CompositeEnhancer` is now in the wrong place in the hash-table. This problem is simply solved by defining the `hash` primitive as a property of the `CompositeEnhancer`, but it does demonstrate the related issues of orthogonality, property subsumption and empathy, (note if *`self`* was assigned to the `CompositeEnhancer` rather than *`object-1`* or *`object-2`* the primitive would have worked uniformly despite the look-up path).

The important aspect of the `Enhancer` is its ability to pass on arbitrary messages to individual objects. This facilitates the dynamic re-description of individual objects. In the next section, the message passing mechanism is attributed to the polymer classes for quite different reasons. Polymer classes are the sole instances of their class (the meta-class). Individual class can extend their behaviour by manipulating the meta-class, but the behaviours are subject to inheritance and affect all subclasses. As will be shown, the inheritance of protocols is in the opposite direction to the subsumption of property generalisations. The message passing mechanism cleanly separates the different roles of the class and the different subsumption of property generalisations.

4.2.7 Polymer Class Behaviour

The class has two roles (§3.5.4.) within the POISE classification:

- Explicit behaviours: As a definition of an object type, declaring behaviours and states for polymer grades, a class template.
- Implicit behaviours: Property abstraction for domain representation of an abstract polymer.

Property generalisations characterising abstract polymers features two complications. As polymer grades change their descriptions so too the abstract polymers hence the `Polymer` subclasses must all evolve their implicit property generalisations that compose each polymer abstraction. This evolution is even more complex than the evolution of grades, as will be shown later. The second complication is the abstraction becomes progressively more specific at every subclass down the hierarchy, and so the implicit properties list decreases. Subclasses do not subsume the abstractions of the superclasses rather the superclasses actually infer their abstractions from the subclasses. The superclass-subclass protocol inheritance is in the opposite direction to the inference by subsumption of property generalisations.

These two roles require separate implementation but represent the same entity. Although the `Enhancer` excels in this activity, it is unnecessary to use an `Enhancer` since all `Polymer` classes will exhibit both roles. Since the two roles are orthogonal, two separate objects could represent the two roles, the class and another object. They are combined by modifying the `doesNotUnderstand` protocol of the class so unbound messages pass to the other object. This allows the two roles to be kept separate, so allowing an instance of a specialised class to represent the evolving abstract entity with an explicit subsumption of the property generalisations. This new object is a `PolymerDataAbstraction`.

Separating the implementation of the two roles had a number of benefits. A `Polymer` class can change the type of `PolymerDataAbstraction`, which changes the abstract property subsumption for the `Polymer` class. There are different subsumption mechanisms for `Polymer` classes subsuming subclasses, instances and orthogonal classifications.

4.2.7.1 Abstract polymer objects

The `PolymerDataAbstraction`, or PDA, represents the abstract polymer part-behaviour of the `Polymer` class. The `Polymer` class receives messages pertaining to the abstract entity and delegates them to the PDA.

The PDA subsumes all properties of all grades in a population. The PDA is polymorphic with all grades in that population, which subsume the property descriptions of their `Polymer` subclasses. The PDA must be able to receive the same messages and respond in the same way as the grades. An additional complication is the evolution of the grade's behaviours. If the grades change their subsumption then the behaviour of the PDA must also change.

The simplest PDA is a single `Polymer` class with grades but no subclasses. The grades all subsume the same properties from the same class. The only difference between the behaviour of a PDA and the grades is the values held for each property. In the PDA each property holds a population of values. An instance of the `Polymer` class (not called a grade) could represent the PDA. This requires polymorphism between an object representing the population of values held by each property of the PDA and the specific values held by grades. The protocols inherited from the class behave correctly only if the object representing the population behave in the same way as the specific values.

In addition to presenting populations of values, the PDA provides the following management tasks:

- Receive and disseminate update messages when grades modify properties.
- Maintain a membership population over which the abstraction is valid, including adding and removing instances from the population.
- Merge with a fellow subclass's abstract polymer to provide abstract behaviour for superclasses, (to follow in §4.2.7.3).
- Manage the addition (and removal) of orthogonal property descriptions as grades in population extend their property descriptions, (also to follow in §4.2.7.5).

All these management tasks are additional to the behaviour of an abstract polymer instance. The tactics of the `Enhancer` extend the behaviour of a `Polymer` instance without compromising the classification describing its behaviour. Unlike other applications of the `Enhancer`, the management role requires access to the properties of the abstract polymer instance, ie the abstract values held by the abstract polymer instance. The PDA is a subclass of `Enhancer` that extends the message passing to both subsume and manage the properties of a polymer instance, representing an abstract polymer.

4.2.7.2 Conformity between population and abstract polymers

The `PolymerDataAbstraction` (PDA) is an `Enhancer` that embellishes the aggregation of polymer properties with a number of data management behaviours, which ensure that the properties of the aggregation are consistent with the population represented by the class. The PDA maintains a close relationship with the class it represents. Indeed, it accesses the class by simply sending the message '`class`' to the instance that it subsumes.

A class does not keep track of all its instances. However a primitive behaviour does exist that searches object space (primary memory) for objects with a class pointer matching a given class and returns all instances of the class in primary memory but not instances represented on secondary memory. It will also include the instance the PDA subsumes, which is not a grade. A more explicit approach is taken of recording grades' existence. There are two sources of grades in POISE. New grades can be instantiated, and the application can connect to a set of existing grades on a database. The `Polymer` class re-defines the standard instantiation protocol to notify the PDA when instantiating a new grade. When the application connects to a new database, the database notifies each class of the grades added, and the message passes to the PDA, which keeps a standard collection of instances representing grades.

An explicit approach is taken for the removal of grades, from primary memory or from a database. When a user directs the removal of a grade using a graphical interface (see §5.2.2), the interface notifies both the database concerned and the PDA.

The PDA views the addition or removal of a grade as the addition or removal of a set of property values. The PDA locates the generalisation for each property (see §4.2.7.4) and correspondingly adds or removes an occurrence of the value in the grade. On a lesser scale, individual changes in a property value of a grade cause a similar change in the PDA. The grade notifies its class of the change. The message passes to the PDA, which locates the appropriate property and updates by removing the old value and adding the new.

The function of the PDA, so far, manages the generalisation of data from a single class with instances but no subclasses. PDAs for `Polymer` classes with subclasses subsumes property descriptions from objects of different sub-types. For these classes, a PDA could subsume more than one instance, one from each subclass. This would complicate population management, so a new type of PDA that can subsume the properties of many other PDAs was created. Then only subclasses with instances require management with PDAs. The superclasses subsume the results of this management from the subclasses' PDAs.

4.2.7.3 Conformity across levels of representation.

The total population of a superclass is its own instances (if any) and the combined population from subclasses. The properties of abstract polymers for general classes are likewise the subsumption of the same properties from the specialised subclasses. These properties are already subsumed together in PDAs for those subclasses. A `CombinedDataAbstraction (CDA)`, subsumes the properties of any number of PDAs. One PDA represents the instances of the immediate class (if any) and one from each subclass with an instance.

Semantically there is no difference between a CDA and a PDA. Both are subtypes of any grade in their respective populations. Each PDA generalises engineering values (see §4.2.7.4). For the CDA to subsume the same property from many PDAS, the CDA resolves the subsumption by merging the generalisations of the same property from different PDAs. Resolving subsumption is a behaviour of the abstract engineering value. In §2.1.8 general subsumption resolution was considered a problem with inheritance representation. Here the CDA manages the problem explicitly with a specific merging algorithm.

The CDA is also an `Enhancer`. Any message sent to the CDA is sent to all PDAs it subsumes. Each message successfully binding to a PDA returns an abstract engineering value. Unsuccessful messages are simply ignored. The CDA combines the abstract engineering values and returns a single object as the response.

4.2.7.4 Abstract engineering values

The abstract engineering value (AEV) is an important description of the abstract polymer. In order to support the abstract polymer, the abstract engineering value must provide the following functions:

- generalise a population of specific engineering values
- presents an abstract value polymorphic with specific engineering values
- resolve subsumption by creating another abstract engineering value covering a combined population.

Although an AEV reduces the population into a generalisation in order to present an abstraction, for complete generality, it does not reduce the information content hidden within its own memory. It is not a memory saving device. It provides protocols for interrogating the complete population of values. The abstraction keeps a record of the engineering values from the population it represents.

The AEV is similar to the PDA. Both add a general functionality to a set of different object types. The PDA adds population management to different classes of polymer. The AEV adds the above functions to different types of engineering values - many of which the users of POISE will develop and are yet unknown. So again adding a common behaviour to an unknown type of object is a problem.

Any type could represent an engineering value. The user defines the type of an engineering value when they define the `Property` object. One behaviour of a `Property` is to return a class for representing the engineering value, (a class since Smalltalk doesn't define types). Strictly, this is a type specification for the argument of the `updator:` method, and the expected type of the `accessor` method response (see §4.4.3). The AEV collects several of these value types. Currently POISE assumes the values are arithmetic, and calculates a medium value. With the aid of the `Enhancer` behaviour sharing technique, the AEV subsumes the behaviour of the median value.

Whereas the PDA collects grades, so the abstract engineering value collects values. The management of the abstract engineering value is the direct result of a similar activity in the PDA. The abstract engineering value receives messages from the PDA to add or remove values as the population of grades change.

4.2.7.5 Applying orthogonal descriptions to abstract polymers

A grade's description extends with the addition of an orthogonal description. If a single grade is capable of extending its description, then so is the description of the abstract polymer. The mechanism for extending the grade subsumes the existing instance of the Polymer class with an instance from an orthogonal class. Yet, another Enhancer facilitates this subsumption.

A grade extends its description using an Enhancer to subsume two (or more) instances, for example one polymer and the others orthogonal descriptions. Since the composition of a PDA includes an instance of the Polymer class, the same mechanism applying to the specific grade also applies to abstract polymers. They both compose of an instance of the Polymer class. A PDA with orthogonal representation subsumes an Enhancer, which in turn subsumes a Polymer instance (the original abstract polymer) and a new instantiation of the orthogonal class. For each new orthogonal class that any instance in the population adds, the PDA must also add a single new instance from the same orthogonal class to its Enhancer.

The orthogonal descriptions provide a secondary classification. Unlike the class of the materials hierarchy, the members of the secondary classification mix with members from other classifications. A subclass, MultipleDataAbstraction (MDA), extends the behaviour of the PDA. The extension segregates the population according to membership to orthogonal descriptions. This allows queries to focus on grades subsuming a particular perspective. Besides some complications in management, there is little difference between MDA and the PDA.

4.3 Delegation in Smalltalk

Splinter uses delegation to combine the behaviours from multiple perspectives forming the behaviour of an artefact. Delegation is known to satisfy this objective. The question is what constitutes delegation. Does the behaviour sharing of the Enhancer constitute delegation? If it doesn't, then does the Enhancer, or some variation satisfy the objectives of a multiple perspective artefact? Bearing in mind complete delegation is not a goal of this thesis, though the ScopeEnhancer is the result of an attempt to capture behaviour sharing between multiple perspectives as closely as possible.

Delegation is a feature of a language implementation that supports empathy. Yet even empathy, as defined from the Treaty of Orlando[5], refers to the variable *self*, which is a common binding that languages implement. The variable *self* is a very important feature of an object–oriented language linking a protocol to the context of the receiver, allowing procedural abstraction of protocols, both abstract and specific, within the same entity. If the binding is outside programmable control and bound according to a rule of the language, then the binding is standard, a feature of the language implementation. Empathy defines the nature of a common binding, so describes a feature of a language implementation.

If both delegation and empathy are descriptions of a language implementation, it is impossible to use them as a description for a Smalltalk language where the implementation does not enforce such characteristics. Smalltalk always binds the variable `self` to the receiver of the message and the receiver of the message is always an instance of the class holding the code. It is impossible for another object to request a protocol from an instance and then take the role of the receiver (ie the other object binding to `self`). In conclusion, it is impossible for Smalltalk to support empathy or delegation beyond the implicit inheritance hierarchy.

Absence of delegation is not specific to Smalltalk but strikes contrary to the success of the class–instance paradigm. Even if the variable `self` could support empathy in Smalltalk, `self` is not the only variable associated with the receiver. Each instance variable maps to a relation of the receiver. Empathy does not define how such variables should bind. The binding of the `self` variable infers these relations should also come from the client and not the owner of the protocol. Such bindings simply do not exist in delegating languages and there are good reasons. The instance variables are just optimisations that benefit from a template-like relationship, such as found between the classes and their instance. They do not extend the expressiveness of a language. If a software model chooses to use the class–instance relationship, it must comply with the rules of instance variable classification.

If the existing Smalltalk environment, its protocols and message passing mechanisms, can not support delegation, could a separate mechanism within the Smalltalk language exhibit the intentions of delegation? Whether such a mechanism is considered to attribute Smalltalk with delegation is not the debate. Arguably, a procedural language can generate programs of an object–oriented fashion, but the language is not considered object–oriented. The same can apply to Smalltalk and delegation implemented as a language extension.

The implementation of Smalltalk can not change but emulation of delegation is possible. Instead of using the existing `self` variable, use a new variable. This variable changes according to who is the client, and in POISE the client is an `Enhancer`. If protocols are written using this variable, then these protocols may empathise with other receivers.

4.3.1 ScopeEnhancer: Delegation Emulation

A `ScopeEnhancer` adopts the simple object description of the `Enhancer`. It shares semantic similarity with the `CompositeEnhancer` but the implementation is different to allow experimentation in behaviour sharing management. The `ScopeEnhancer's` aims are also much more ambitious.

The ScopeEnhancer aims to support the sharing of protocols across a community of objects. The community forms an ordering from an object containing the most specific protocols to the last and most general object. Objects in the community can belong to other communities, forming an acyclic graph of behaviour sharing.

In terms of delegation, the ScopeEnhancer is the client and the objects in the community are proxies. The ScopeEnhancer does not contribute any behaviours itself, so the overall behaviour is the same as if the first object of the community, the most specific, was the client. The rest of the objects in the community are proxies of this client. For simplification, all the objects in the community are called 'proxies'.

The proxies are standard objects. Separate classes define the behaviour of each proxy. The classes reside in the standard inheritance hierarchies. Individually they do not share the protocols of others, yet in order for the sharing mechanism to work their protocols send messages to *self*, which are not behaviours of the class. In this way, classes of proxies are special.

The mechanism starts with a ScopeEnhancer and the proxies in a SequenceableCollection grouping the community. Messages to the community are sent to the ScopeEnhancer, which has the responsibility to locate the protocol within the community and 'overview' the evaluation.

Upon receiving a message the ScopeEnhancer sets up an exception handler and re-sends messages just like the CompositeEnhancer. Any messages, not just those re-sent messages, not found during evaluation of the ScopeEnhancer's behaviour will be caught by the handler.

If the messageNotUnderstoond signal is raised, the exception handler gains control. The handler identifies the message and object (receiver) causing the signal. The handler also identifies if the message was sent directly from the ScopeEnhancer, ie if it is the initial 'delegated' message re-sent. If the initial message springs the trap then the ScopeEnhancer re-sends the message to the next object in the community, just like the CompositeEnhancer. The ScopeEnhancer differs from the CompositeEnhancer when the message raising the signal is **not** the initial message.

When a protocol for the initial message has been found, the trap stays set during evaluation of the protocol. For empathy within the community, the ScopeEnhancer relies on the protocol evoking the messageNotUnderstood signal to intervene. This will happen if the proxy sends a message to *self* that it does not understand. The message raising the signal in this case is not the initial message.

The exception handler of the ScopeEnhancer identifies the message and object (receiver) raising the messageNotUnderstood signal. If the receiver is the object from the community that last received the initial message (the delegated message), and this message is different, then a new message delegation starts. The ScopeEnhancer searches the whole community for a new protocol for the unbound message. If the receiver is not the expected object, the ScopeEnhancer assumes the signal is a genuine type error.

The policy implemented is one of a few variations tried. They all rely on the handler trapping signals resulting in a protocol search in the community of objects. In this way, a message in one object can gain access to protocols of other objects within the 'scope of the Enhancer' whilst the object does not define the behaviour itself. This final condition is inconvenient since it prevents the ability for proxies to specialise behaviours existing in the class hierarchy. To further understand this limitation, the specific implementation of the ScopeEnhancer follows.

4.3.2 Implementing the ScopeEnhancer

Two separate parts of the ScopeEnhancer implement the interface and the delegation event. The class ScopeEnhancer, a subclass of Enhancer, implements the interface receiving the initial messages. ScopeEnhancer changes the doesNotUnderstand method to construct an instance of DelegationEvent with the message and the ScopeEnhancer's community of objects. The DelegationEvent is responsible for locating the protocol within the community and 'overview' the evaluation.

Separating interface and management of the delegation event allows specialisation of the DelegationEvent class. DelegationEvent inherits from Object like most Smalltalk classes. The subclasses create different delegation policies.

This scheme allows the ScopeEnhancer to create a copy of the delegation event to handle each message mutually exclusively. Each shared protocol has an event to manage the evaluation of that protocol.

The ScopeEnhancer's interface evokes the DelegationEvent as follows:

1) Enhancer receives message.
2) Enhancer does not understand message and receives doesNotUnderstand: message.
3) The Enhancer creates a new DelegationEvent object.
4) The ScopeEnhancer gives the event the message selector to search for and the community as an ordered collection of objects
5) The Event is told to search and evaluate.

The dominant behaviours of the DelegationEvent are search, evaluate and 'trap'. The 'trap' is an exception handler.

6) When an event receives a message to search it initialises a pointer to the top of the collection (of objects to delegate to). This is the current receiver.
7) An exception handler monitors for a does not understand signal.
8) The do-block is evaluated.
9) The message being delegated is sent to the current receiver.

From this point until the message is complete, the trap is set. The following occurs if any object triggers the trap by raising the signal.

10) The handle-block is evaluated with an exception object as argument.
11) The handle checks if the initial context, the context sending the message which was not understood, is the do-block context from in step (8).
12) If the same then the receiver is set to the next proxy receiver. If no more proxies then reject the exception (normal does-not-understand behaviour occurs) otherwise re-evaluate the do-block (back up to step 8).
13) If not the same, the handle checks if the originator of the exception, the object which did-not-understand, is the current receiver. If not then reject the exception (normal does-not-understand behaviour occur).

If the originator is the current receiver, the ScopeEnhancer directs the message to the whole community.

14) A new message is being sent. Simply pass the message to the ScopeEnhancer (start at step 1) and proceed with the response.

Step (14) causes the creation of a new DelegationEvent. The current event is still active until that event's message is complete.

It is possible for the parts from the community to message each other and send various parts of the community as arguments in those messages. The various DelegationEvents handle the situation well, with each part providing it's specific behaviour first, then the communities collective behaviour afterwards.

The above DelegationEvent was the first implementation. This implementation attempts to extend the variable *self*, by sending messages to *self* that are not understood within the local object's classes. An alternative is to define a new variable. A specialisation of the above DelegationEvent demonstrates one way of achieving this.

The new variable is named the *client*. Unlike the *self* variable, which binds to the receiver automatically by the virtual machine, *client* binds explicitly through a message sent to *self* in each protocol. The class Object defines the message client. All objects (bar Enhancers, which do not inherit from Object) can bind to the protocol. The semantics of the *client* message is to return the empathetic *self*, the ScopeEnhancer. If there is no ScopeEnhancer, *client* simply returns *self*.

A temporary variable can be assigned to the response to the message:

```
| client |
client := self client.
```

If *client* is now sent messages instead of *self* then the protocol is fully empathetic. Messages sent to *client*, the ScopeEnhancer, immediately form a new DelegationEvent that directs the message to the first object in the community, which is the most specific.

The message '*self* client' finds the ScopeEnhancer by raising the doesNotUnderstand signal. The specialisation of the DelegationEvent detects the *client*-message, and treats it specially. The DelegationEvent returns the ScopeEnhancer as the response to the *client* message. If the doesNotUnderstand signal is not handled, then there is no ScopeEnhancer. The *client* protocol detects this situation and returns *self* in response to the message. Protocols using the *client* variable without a ScopeEnhancer have all messages sent to *self* rather than a community. In this case, *self* is the most specific in a community of one.

With *client* the ScopeEnhancer can emulate explicit delegation. In delegation, a *client* delegates a message to a specified proxy. A specialisation of the ScopeEnhancer, with a single object as the community, emulates the proxy. The message is sent to this ScopeEnhancer. The specialised behaviour notes the object sending the message, the delegating *client*. The ScopeEnhancer finds this *client* by accessing the message contexts, just as exceptions access signal handlers. When the proxy object sends the *client*-message, rather than returns a ScopeEnhancer with just the proxy, this specialised ScopeEnhancer adds the delegating-*client* to the community as the most specific object. The delegating-*client* then overrides all messages sent to the *client* and the remaining proxy behaviours in the community acts as defaults.

Even with `client`, the programming style needs to change. If the programmer uses the variable `self`, then the programmer does not expect delegation. If the programmer does expect delegation, then programming must change to allow the `ScopeEnhancer` to support the empathy.

4.4 Hierarchical Schema Evolution

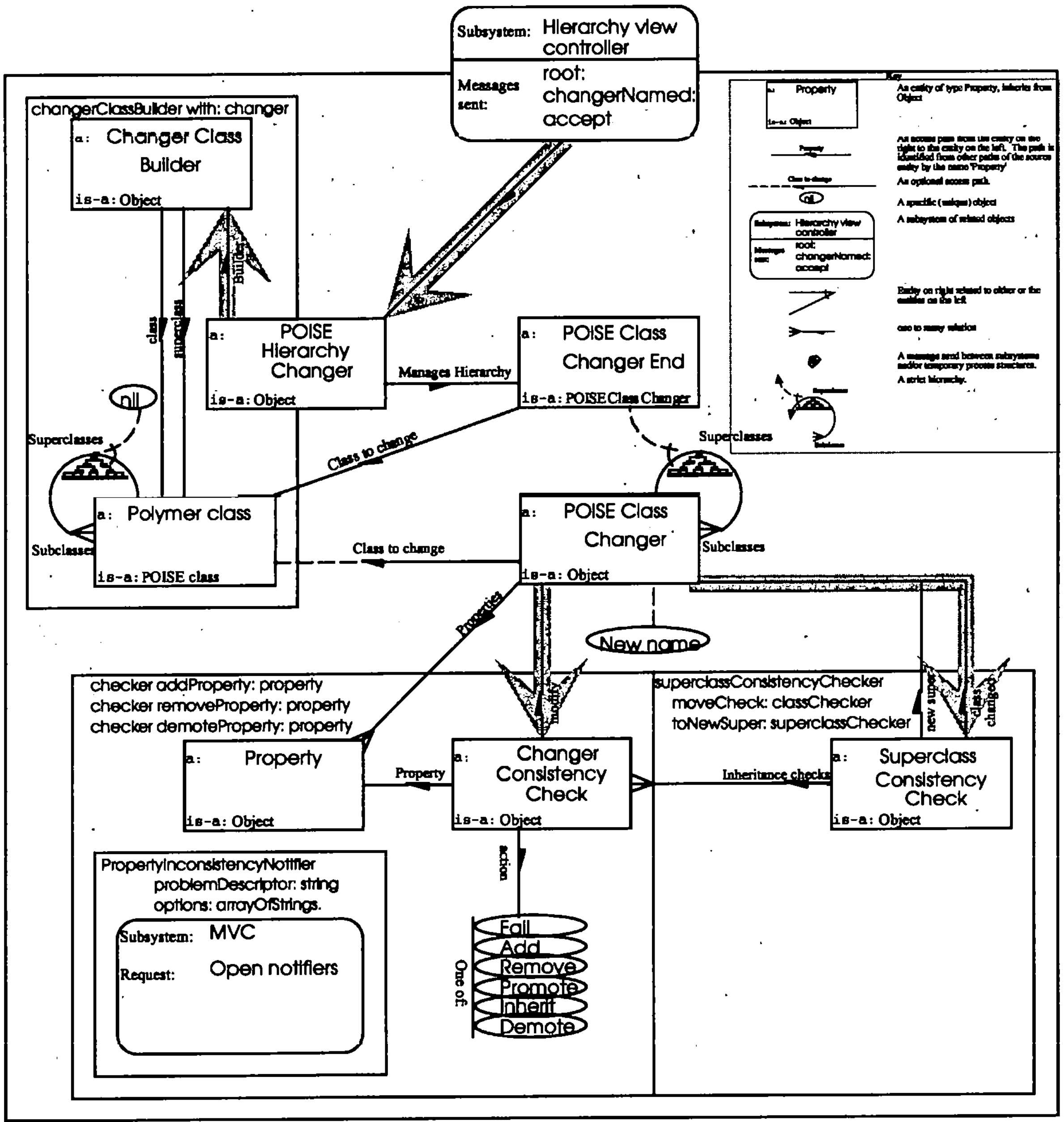
Schema evolution in POISE is not a common activity. Once the classification is initially set up, only occasionally will it change when a user adds new classes of polymers or properties. Even so, user interactions must be effective. For example, excessive delay would be unacceptable when processing a change. Excessive delays were incurred when making a series of changes using the development system's mechanism for schema evolution, the `ClassBuilder`. Long waits occur after each change to the schema, so a source of optimisation was sought.

The development system's `ClassBuilder` evolves class specifications; in particular, the scope of variables declared by a class and accessible by its instances. Changing the class's name and global's scope, such as class variables and pool variables, only requires minor changes to the state of the class object. Methods too only affect the method dictionary of the class. Subclasses implicitly inherit these changes and they need not change themselves. This is not the case for changes to the format of instances. The most common change is the number of instance variables. This changes the integer format descriptor (that encodes the storage layout of instances) and has much more extensive consequences.

Subclasses and instances inheriting instance variables, and any other format information, from a superclass must explicitly coerce their own format to match changes in the superclass. A change in a class can affect a number of subclasses and many more instances. The addition of an instance variable, for example, requires each subclass to change their format. For each instance the class generates a new instance under the new format, copies across the states of the instance variables and makes the old instance become the new. With the change in instance variable position within instances, each method of the subclasses requires recompiling. The `ClassBuilder` coordinates all these modifications to the hierarchy.

The schema of the `ClassBuilder` shows significant inefficiencies when moving an instance variable from a subclass to a superclass. The schema dictates that the name of an instance variable is unique within the inherited scope of a class. The subclass must first remove an instance variable before adding it to a superclass. Otherwise adding the same named instance variable to the superclass will conflict with the existing named variable in the subclass. Removing the name reduces the number of instance variables causing a reduction in data structure of all instances of the subclass. All methods are also recompiled. Only then can the superclass add the instance variable. This adds the instance variable back to a subclass, by inheriting from the superclass. All the instances of the subclass restructure again. Now the structure holds no data for the moved variable. The values were lost when the instance variable was removed from the subclass and the structure reduced. All the subclass's methods are recompiled a second time.

When properties move, the variables they define move. Moving properties up and down the inheritance graph is called promotion and demotion (§3.3.2.1). Moving instance variables up and down the inheritance graph is inefficient and causes a loss of data. Since this is a significant activity when modifying the classification, a new schema was developed to schedule the changes to the classification.



Subsystem: Hierarchy view controller
Messages sent: root: changerNamed: accept
changerClassBuilder with: changer
a: Changer Class Builder
is-a: Object
Builder
class
superclass
a: POISE Hierarchy Changer
is-a: Object
Manages Hierarchy
a: POISE Class Changer End
is-a: POISE Class Changer
nil
Superclasses
Subclasses
Class to change
a: Polymer class
is-a: POISE class
a: POISE Class Changer
is-a: Object
Properties
New name
modify
new super
class changed
checker addProperty: property
checker removeProperty: property
checker demoteProperty: property
a: Property
is-a: Object
Property
a: Changer Consistency Check
is-a: Object
superclassConsistencyChecker
moveCheck: classChecker
toNewSuper: superclassChecker
Inheritance checks
a: Superclass Consistency Check
is-a: Object
action
One of:
Fail
Add
Remove
Promote
Inherit
Demote
PropertyInconsistencyNotifier
problemDescriptor: string
options: arrayOfStrings.
Subsystem: MVC
Request: Open notifiers
Key
An entity of type Property, inherits from Object
An access path from the entity on the right to the entity on the left. The path is identified from other paths of the source entity by the name 'Property'
An optional access path.
A specific (unique) object
A subsystem of related objects
Entity on right related to either or the entities on the left
one to many relation
A message send between subsystems and/or temporary process structures.
A strict hierarchy.

Figure 15: Schema- hierarchy editor

The HierarchyEditor is POISE's user interface for evolving the polymer classification (§3.3.3). The interface is the user's view into a mechanism that radically changes Smalltalk's traditional class evolution. The new mechanism boasts the following features:

- Batching changes by class to minimise processing.
- Abstracting inheritance by property rather than by method.
- Prevent loss of data through instance restructuring
- Improving re-compilation efficiency by extending method representation.

Figure 15 is a schematic representation of the mechanism supporting the first two points, batching and inheritance checking by property. The ChangerClassBuilder does the actual evolution of the polymer classification. Improving the re-compilation is addressed later, in §4.4.5.

4.4.1 Assigning Properties to Classes

When a new interface on the hierarchy opens it initiates a new POISEHierarchyChanger (PHC), the batch manager of the session's changes. The PHC creates and records ClassChangers (CC) upon request. CCs record the changes for each class. The CCs form a hierarchy transposed from the existing class hierarchy. A ChangerEnd object terminates the scope of the PHC at the root of the hierarchy, thereby limiting the range of class modifications to some domain in the Smalltalk hierarchy. For the polymer hierarchy, this limit is set to the POISE class, the superclass of Polymer.

```
"The instance varrible 'current' is a ClassChanger currently being checked during recursive
behaviours.  The instance varrible 'offenders' are ClassChangers that conflict with the current
process.  The instance varrible 'superseders' are ClassChangers that have redundent states due to
the current process.  The instance variable 'originator' is the changer the process wil act on."

consistencyCheckForAdding
"current is a subchanger of an originator who is adding a property. Has current removed the propert
explicitly this session, causing a direct conflict?"
            (current hasRemoved: property) ifTrue: [offenders add:
                current].
"Does current define the property that it will now inherit"
            (current properties includes: property) ifTrue:[superseders
                add: current].
"Recursively check subclasses"
            current subChangersDo:
                [:ch |
                current := ch.
                self consistencyCheckForAdding]

consistencyCheckForRemoving
"current is a subchanger of an originator who is removing a property.
Has current added the property explicitly this session, and now should define the property"
             (current hasAdded: property) ifTrue: [offenders add:
                current].
"Recursively check subclasses"
            current subChangersDo:
                   [:ch |
                current := ch.
                self consistencyCheckForRemoving]
```

Figure 16: Code- ClassChanger consistency checking for adding and removing

Modifications to the schema are directed to the individual CCs concerned. The view requests the CC by class name in the message changerNamed:. A CC can add, remove, demote or promote a property. A CC can change its superclass or its name. New CCs are created for non-existent classes that will be added to the hierarchy.

Each request changing the schema initiates a consistency check. A `ClassConsistencyChecker` (CCC) is set with the appropriate check flag: add, remove, promote, or demote. The descriptions of these consistency checks are given in §3.3.2.1. Any inconsistency causes a `Notifier` to open giving the user appropriate options or else for aborting the request for change. After completing the consistency check, the checker updates the CC. The code implementing the checks in this schema is given in Figure 16 and Figure 17. Note these methods recursively call themselves as the check goes through the hierarchy of `ClassChangers`.

```
demoteCheck
"The originator is any one of the subclasses of a superclass that defines property.  The demote will
remove the property from that superclass and add the property to it's subclasses such that the
originator defines the property.  Hence all superclasses of the originator will not inherit or define the
property."
"Current starts as the immediate superclass (changer) of the originator"
"Any superclass that has had the property explicitly added is now having the property removed -
hence a conflict.  Add conflicts to offenders"
            (current hasAdded: property) ifTrue: [offenders add: current].
"Collect up the superclasses of the originator.  Their subclasses (not in the line of inheritance of the
originator) will require property to be added"
            superseders add: current.
"Continue recursion until the superclass that defines the property is found"
            (current properties includes: property)
                 ifFalse:
                 [current := current superChanger.
                 self demoteCheck]
promoteCheck
"Current is a class that will have a property added. (Often it is initiallt the same as originator when
adding).  Check to see if current can inherit the property from it's superclass (superchanger).  To dc
this, all subclasses of superchanger must also define property.
Current will be left at the highest superchanger which will accept property"
            | superChanger subChangers|
            superChanger := current  superChanger.
            (superChanger hasRemoved: property) ifTrue:
"Property has been explicitly removed. Do not promote"
                 [^self].
"Check the subclasses of this superclass, excluding current which is having the property added.
Check they all define property.  If a single one doesn't, promotion is not possible"
            subChangers := OrderedCollection new: 10.
            superChanger subChangersDo:
                 [:ch |
                 ch == current ifFalse:
                     [(ch properties includes: property)
                           ifFalse:[^self].
                     subChangers add: ch]
                 ].
"Promotion possible. All these subclasses will need to have the property removed so they can
inherit it from superchanger.  Add them to superseders. "
            superseders addAll: subChangers.
"Now recursively check to see if the property can be promoted to the next superclass"
            current := superChanger.
            self promoteCheck
```

Figure 17: Code- ClassChanger consistency checking for demoting and promoting

The primary consistency checks are concerned with inheritance conflicts. The checkers are also able to interact with `Property` objects to ensure that mutually exclusive properties are not both accessible to the same class. Each `Property` determines the existence of other `Properties` it depends on.

When a class changes its superclass, it affects the inheritance of the class and its subclasses. The properties subsumed from the new superclass are checked against the properties defined by the class and each subclass. Coordinating these checks is a `SuperclassConsistencyChecker` (SCC). Essentially this object iterates through all the properties of the class being moved. If the property is not inherited from the new superclass then it is defined on the class being moved. The moved class will only increase its property base. The SCC then checks each subclass to see if the newly inherited properties conflict with any subclass property definitions.

SCC utilises a number of CCCs to ensure the new superclass does not conflict with the class's properties. Unlike user driven property changes, `Notifiers` are not raised. Instead, default actions are taken, such as removing properties that are now inherited and adding properties that were previously inherited. Such changes are visible in the hierarchy view before any permanent change is made to POISE, allowing the user to make adjustments.

The SCC also checks for an invalid inheritance structure. The new superclass must not be the moved class or any of its subclasses, thereby creating an inheritance loop. At all times the changes are consistent with the inheritance rules and any other rules imposed by the properties.

4.4.2 Building Classes

The benefit of a separate model for representing the changes to the hierarchy is that POISE can control the order the changes occur in the classification hierarchy. When the user decides to accept the changes, POISE always begins modifying the most general class first, which are the classes at the top of the hierarchy. Another benefit is that there is no need for consistency checks as they have already been made, unlike the `ClassBuilder` of the Smalltalk development system. Instead a new `ChangerClassBuilder`, (CCB) does the changes. The individual CCs specify the new classes to the CCB.

Each CC specifies a new class object. This specification includes the superclass and an aggregation of property objects. The property objects specify the behaviours and state variables of the new class. The CCB collects the instance variables and defines the new class, but installing the methods is the responsibility of the property objects.

POISE allows behaviours specific to `Polymer` classes that are not specific to a property. These behaviours use standard Smalltalk methods. Any code not derived from a property object, but is specific to the class, requires copying over from the old class's method dictionary. The builder does this after installing the properties, allowing the class specific behaviour to over-ride a property specific behaviour.

The CCB starts at the root of the hierarchy. Building superclasses occurs before building subclasses and every new class built is initially absent of subclasses. After building each class, building starts on its subclasses.

The subclasses inherit the changes of the newly built superclass. A subclass will not expect changes in the superclass after building the subclass, thereby ending repeating evolution of subclass structure and methods. Also, there are no subclasses when building a (super) class. The builder does not recursively update any subclasses.

Once the new class objects are built, the builder coerces the instances (if any in primary memory, §4.5) of the old classes across to new instances of the new classes. Since this is only done once at the end of all schema changes, no data is lost. Data associated with an instance variable in the old class moves to the same named variable in the new class, regardless of the variables position in the instances data structure.

After accepting the hierarchy and the POISE class hierarchy has been rebuilt, the new classes substitute the old classes, then the old classes along with the old instances, the PHC and all its CCs, are all garbage collected (see §4.5.8).

The only part of the story left to tell is how property objects describe the methods of the class. Each CC, which collects the properties, passes the new class to each property. It is up to the properties to install their behaviours on the new class.

4.4.3 Properties and Partial Template Objects

A material class is a template for the behaviour of grades. This template comprises of an aggregation of properties. Each property contributes a part to the template. Objects with the ability to partially describe classes, and thereby the instances, are abstractly known as Partial Template Objects[77], (PTO).

The Hierarchical Schema interface collects the properties as the user directs for each class. With PTOs, the user defines the material classes template, and thereby the behaviour of grades. The process of composing the material class involves the information in the PTO, and some coercion of the instances.

Smalltalk's development environment provides the programmer with schema evolution for dynamically adding instance variables and methods to classes. At its simplest, the PTO is a similar description of change, where the declaration of instance variables and methods using text, as entered by the programmer, generates a macro like function.

Each property is an independent collection of behaviours. Instance variables in a property description support the implementation of the property's methods. Usually the variable holds the specific engineering value for each grade and methods provide the interpretation. The instance variables are few and specific to the property. The description of the methods are in turn limited to accessing these instance variables, some global variables and accessing other states of the receiver through messages to `self`. Global variables always bind to the same objects. `Self` always binds to the receiver. Instance variables though bind to locations within the structure of the receiver. This structure is different for different classes of object. Since the PTO's protocols are not defined for any particular class, the instance variables complicate the compilation of PTO protocols.

For each PTO the scope of variables the methods may access, the instance variables, globals, arguments and temporaries etc, are consistent regardless of the class of receiver. Only the physical binding of instance variables is unknown. Initially the programmer represents each protocol as text (the source) which is then compiled. The PTO could keep the text representation though any errors in the text would not become known until the text was compiled for a particular class. Instead, POISE extends the compiler to cater for PTO protocols. The compiler optimises the protocols, converting text to pseudo code, see §4.4.5.

If the protocols are correct, they are only correct for classes that support the instance variables required by the property. Before a `Polymer` class adds the protocols of a PTO, the class adds the instance variables. Then the PTOs can 'install' the protocols on the class and each instance in memory.

With the help of the `ChangerClassBuilder`, part of the hierarchy schema, PTOs also simplify the addition of instance variables. They remove the responsibility from the `ClassBuilder` and the rest of the hierarchy system from dealing with the complexities of the development system's compiler, scoping rules and naming conventions.

4.4.4 A Mechanism for Partial Template Objects

In POISE any instance of the class `PartialTemplate` or its subclasses, such as `Property`, is a PTO. The abstract description of `PartialTemplate` is:

```
Object subclass: PartialTemplate
instance variable names
        templateName
        classMethods      instanceMethods     classScope      instanceScope
        prerequisites     preclusions         classesInstalledOn
instance method for installing
        installOn: aClass
```

`PartialTemplate` supplies the following abstract specification of a PTO:

Ability to insert methods. — A PTO associates a dictionary of `instanceMethods` and a dictionary of `classMethods`. These methods are "partially compiled" (see §4.4.5) for extra portability and efficient compilation into any class installing the PTO. The `instanceMethods` contribute behaviours of instances, while `classMethods` contribute behaviours to the `Polymer` class itself.

Protected scope of variable reference, local to the set of inserted methods. — `instanceScope` and `classScope` express variable definitions accessible to the instance methods and class methods (respectively). A class–installing the PTO adds these variables as necessary. The scopes comprise (i) instance variables visible to all methods affecting the instances of the class, (ii) community-pool variables visible to all `instanceMethods` and `classMethods` of the PTO, (iii) Smalltalk global variables visible to all Smalltalk methods generally. The community-pool variables are a local enclosing scope of the PTO behavioural "community". These local variable definitions are not otherwise available to other methods of the affected class.

A record of all classes that the current PTO *has been installed.* — `classesInstalledOn` stores this set of classes as part of a mechanism ensuring changes propagate to them when modifying the PTO.

Any given class template may install more than one PTO. In order to control multiple installation the specification of `PartialTemplate` also incorporates the following:

Prerequisites. — These are other PTOs which a class must install (or inherit) before installing the current PTO. This attempts to provide for control over inter-module dependencies that arise if methods of the current PTO call methods in other PTOs.

Preclusions. — These are other PTOs with conflicting behavioural definitions.

This specification of PTOs supports a cohesive description of the partial contents of `Polymer` classes. They attempt to provide the classes they affect with a well-composed character, in the sense that each set of installed properties observes a scoping regime common to members of the set but otherwise private.

4.4.5 Generating a Behaviour of a PartialTemplateObject

Within POISE, the `Property` object implements the PTO as part of its role. The user, using a `PropertyEditor` browser, creates a new property. Then a second specialised `PropertyMethodBrowser` browser adds behaviours to the property. See Figure 45 for examples of the browser. Each property behaviour generates a `PartiallyCompiledMethod` (PCM). In Figure 18 the schema of the process generating the PCM from source code is the subject of this section.

The schema is a modification of the process that compiles standard Smalltalk code. The schema shows both compilations, with a different method activating each. A dashed line demarcates the two methods in the initial context.

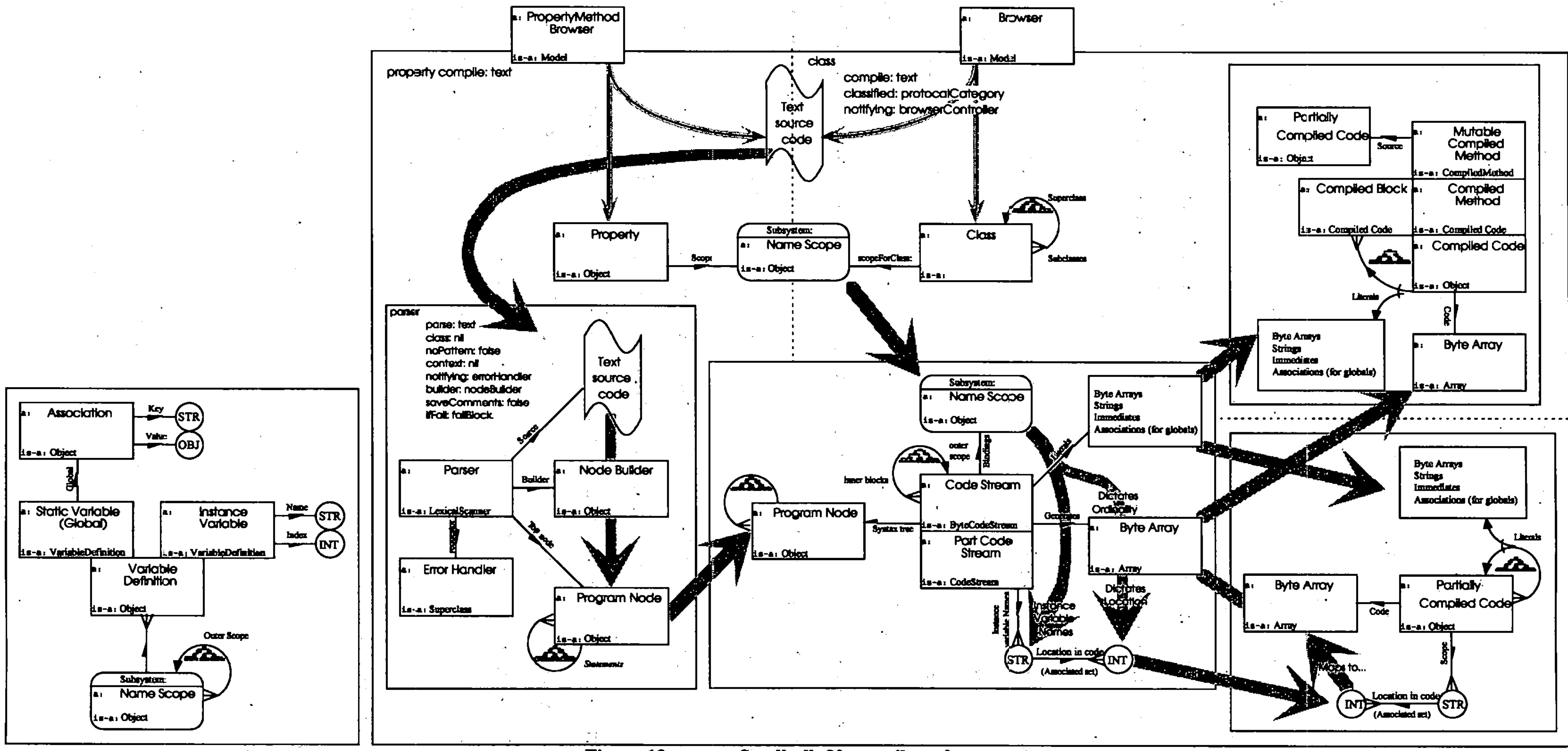


Figure 18: **Smalltalk 80 compiler schema**

The behaviour specialising the PCM compilation is mainly in the class PartCodeStream, a subclass of the CodeStream that normal class-bound compiled methods use. The PartCodeStream class captures the extra information that the PCM requires from the source.

Compiling code requires the text source code and a NameScope. The NameScope contains the mapping between variable names and their storage location. The NameScope is a nested structure, an extendible ordering of the variables descriptions. The nesting allows the addition of temporary variables while generating code. This ordering has little effect on the compilation other than when optimising some structures in code that do not require access to parts of the NameScope. In most cases, only the overall variable visibility is of any concern.

NameScope contain two basic types of variable definitions, which are StaticVariables and InstanceVariables. The StaticVariables are typically globals. Access to globals is the same for all methods, so will not be any different for a classless PCM. InstanceVariables are class dependent. InstanceVariables define the name and the index within instances of the class. Since a class is not known, the PCM compiles for a PTO, and the PTO provides the names and indexes of valid instance variables. As long as these variables are unique and the indexes are unique, the standard compiler will accept them. Later the PCM will explicitly modify the indexes to complete the compilation for given classes.

A compiler is a translator of 'high-level' source code to 'low-level' code. It typically consists of a lexical analyser that converts the source text into tokens, a Parser that converts the sequence of tokens into a syntax tree, an attribute collector and distributor that apply the contextual constraints of the source language, and a code generator and optimiser that translates the syntax tree into the low level code[37]. In the schema, Figure 18, these modules of behaviour can be seen as follows. The lexical analyser is the general behaviour of the Parser's superclass, LexicalScanner. The specific subclass Parser describes the syntax of Smalltalk's one-look-ahead grammar language. Subclassing off the scanner makes the schema amenable to other language syntax.

A sentence in the Smalltalk language composes of a sequence of tokens called terminals. The Parser applies rules for grouping terminals with other terminals and other groups of terminals. These rules are called production rules.

Production rules generate a 'syntax tree', which is a hierarchy with the terminals as leaves and the groups as nodes. The simplest representation of a syntax tree is a hierarchy of terminal and non-terminal symbols for the nodes. Each node is marked by a 'non-terminal' name that identifies the production rule used to produce the node. These names convey the semantics of the sentence. Later, they instruct the compiler how to construct the code in the output language.

The compiler applies denotation semantics, which means each component of a sentence corresponds to a component of the language's semantics. The syntax tree is a decomposition of the sentence into semantic components. The production rules identify each node and their corresponding semantic. The language's semantics are rules for constructing low-level code from the components of nodes. Each node combines simpler nodes and terminals. At each node, code combines until the compiler constructs code for the whole sentence.

With the importance of the nodes evident, it becomes clear why compilers often create elaborate syntax trees. Each production rule in the `Parser` maps to a type of node. In Smalltalk, rather than use a symbol, the `Parser` creates a message corresponding to the production rule, and sends the message to the interface of a `NodeBuilder` object. Messages to the `NodeBuilder` instantiates `Node` objects and generates the syntax tree. As the `Parser` scans the source code, each application of a production rule causes a cascade of message sends to the `NodeBuilder`, which builds the tree. After scanning and all messages to the `NodeBuilder` are complete, the `Parser` is left with one distinguished node, or top-node, which is the root of the syntax tree. In Figure 18, the syntax tree is the hierarchy rooted in `ProgramNode`, the top-node of the hierarchy. More specifically, this hierarchy composes of instances from subclasses of `Node`, each class distinguishing different types of node.

The `NodeBuilder` provides an interface between the production rules of the language and the `Node` objects used in the syntax tree. The `Parser` uses none of the node's instance behaviours. The `NodeBuilder` only uses instantiation behaviour of the `Node` classes. This leaves the behaviour of `Node`, and its subclasses, a clean representation for the language semantics. Therefore, although only syntactic information is used to generate the syntax tree, it is already a semantically powerful structure.

One abstract `Node` subclass represents variables. This node is specialised for temporary variables, arguments, instance variables and globals. The production rules do not provide the information to differentiate between them. This information comes from the variable scope, which the class usually provides. Instead, the PTO provides the initial variable declarations.

The `Parser` returns the top node of the syntax tree to the underlying context, which then initiates checks on the contextual validity of the tree. Smalltalk is not a context-free grammar. The choice of production rules at a point in a program depends on rules previously applied, eg a temporary variable must be declared before it can be assigned a value. The requirement that a variable is declared forms part of the contextual description at some point in the program. This 'program-context' description is part of the behaviour of the `CodeStream` object. Since the formulation of the program-context is an important part of code generation, the `CodeStream` also generates code while the contextual checks are being performed by the syntax tree. So, although the checks are initiated by underlying compiling context by passing a new `CodeStream` with the `NameScope`, messages pass back and forth between each node and the code stream, some for checking, some for code generation.

Code generation is a combination of the following activities:

- Binary instructions are sequentially added to a byte array.
- Collect literals (objects accessible by any code: integers, characters, selectors, references to global variables,) the code uses.
- For each inner-block the code uses the compiler generates a new `CodeStream`.

The inner-block is a unit of code within a protocol. An example is in Figure 14, (pp 109), which contains a handle-block and a do-block inside the main protocol. A separate `CodeStream` compiles each block. Later the code for these blocks will join the literals as attributes of the main protocol object.

In addition to these activities of the `CodeStream`, the `PartCodeStream` collects every reference made in the byte array to an instance variable. Two kinds of instructions in the code refer to instance variables, and they are either an `accessor` or an `updator`. The `NameScope` supplies the index of the instance variable, which follows the instruction code. The `PartCodeStream` collects the location in the code of this index (the 'location in code' in Figure 19) and associates it with the name of the instance variable.

After the checks and the initial pass of code generation, the final step usually makes the method, a Smalltalk protocol. The `CodeStream` constructs the components of a method. It does not present these components in a way the virtual machine can execute. Executing code is the independent intention of another object, the `CompiledCode`. Each `CodeStream` makes a `CompiledCode` object. The `CompiledCode` presents the components simply and uniformly within its own instance variables for the virtual machine to access. For the PTO, `PartCodeStream` constructs a `PartiallyCompiledCode` (PCC) (Figure 19) that is only partially of the same type and do not execute. This object is not a subclass of `CompiledCode` but it can generate a kind of `CompiledCode` when the PTO transfers protocols to a class.

A PCC defines a subset of CompiledCode behaviour so it can masquerade as a method in code browsers. To this behaviour, the PCC adds its own behaviour for re-compiling the code for a given class. This behaviour creates a mapping between instance variable names of the PCC and the indexes of instance variables of the same names in the class. A simple recursive Class behaviour collects the ordered lists of instance variable names defined by the class and its superclasses. The position of the names in this list provides the correct ordinal number of the slot in the instances.

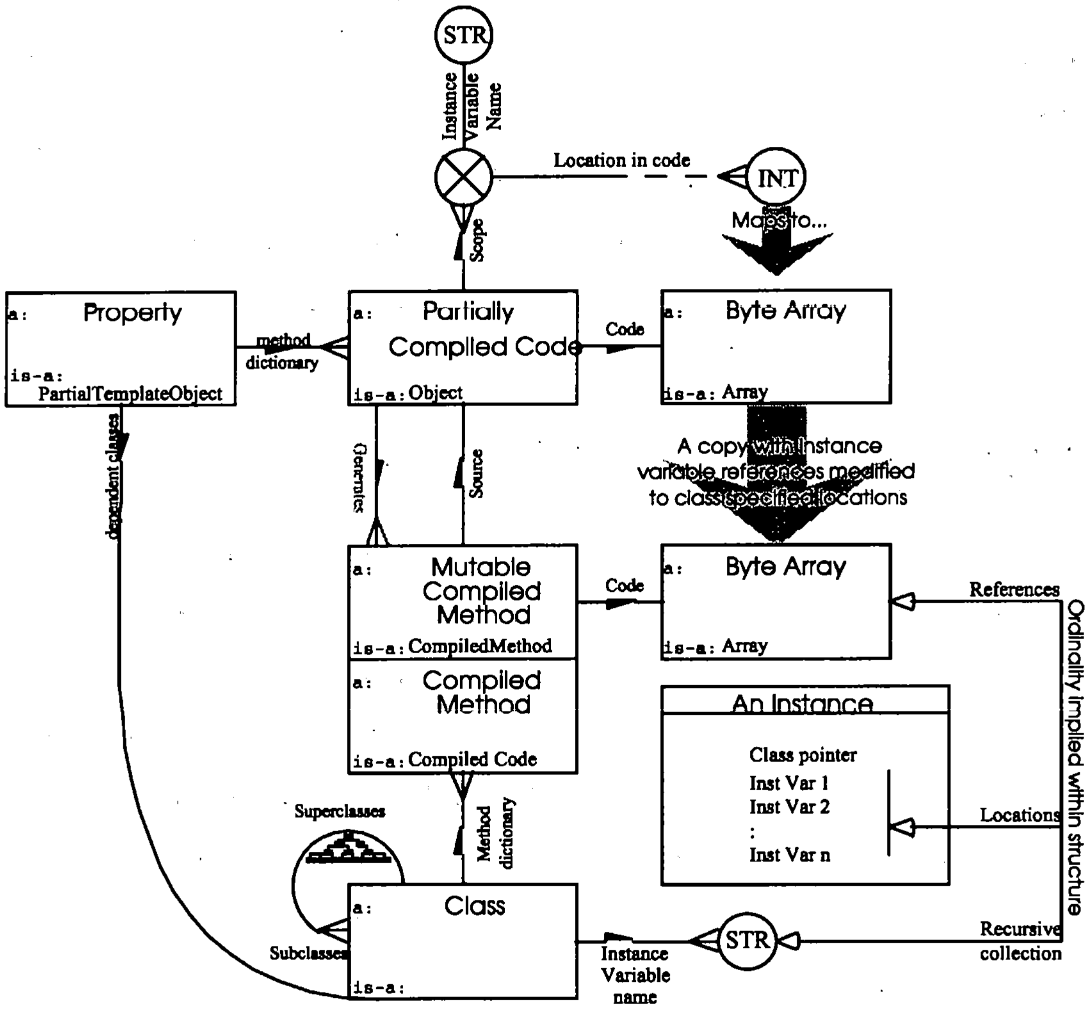


Figure 19: Schema- PTO linking

Like CompiledCode, the PCC can contain inner-blocks of code with the literals, but they too are PCC, not CompiledCode. Each separate PCC keeps a mapping of instance variable in their local code. The PCC is therefore a hierarchical structure. With the literals of a root PCC (an attribute not shown in Figure 19) there are PCCs as inner-blocks, which in turn can hold other PCCs with their literals. When a PTO installs PCC onto a class, it sends a re-compile message with the class as an argument to each root PCC. This creates the mapping for the class. A copy of the local code updates for the new indexes. The collection of literals in each PCC are copied (the literals themselves are not copied). The re-compile message then passes to each PCC found in the new collection of literals, which recursively pass on to their inner-blocks down the hierarchy. With the copy of literals and modified copy of code, each inner PCC creates and returns a new `CompiledBlock`, the subclass of `CompiledCode` representing inner-blocks of code. The exception is the first outer PCC, the root PCC, which generates a `MutableCompiledMethod (MCM)` object, a kind of `CompiledMethod`, and not a compiled block.

The PCC could make a standard `CompiledMethod`, which is the normal kind of `CompiledCode` that classes manage and code browsers manipulate. The `MutableCompiledMethod` adds a more efficient re-compilation behaviour. When instance variables change, or the class moves to a new superclass, all the methods of the class re-compile. A standard method re-compiles by sourcing the original text (from secondary storage) and going through parsing, syntax tree construction, and compiling. The `MutableCompiledMethod`, on the other hand, keeps a reference of the PCC, which can re-compile the method for any mapping of instance variables by changing instance variable indexes in the byte stream. When the MCM receives the message to re-compile from a class, it passes the message to the PCC, which returns a new MCM. The class then replaces the old MCM with the new one.

4.5 Data Storage

Data storage is a broad research topic. The issue at hand though is object storage for POISE. The requirements in §3.8.5 summarise the issues. Attempts to find a commercial system satisfying these requirements failed. This is due to two factors: first, Smalltalk is a particularly expressive object-oriented language, supporting large and complex object relations. A database supporting applications in Smalltalk requires an equally expressive data description language. Secondly, many OODBMS compete with RDBMS. Consequently, their design emphasis is on data retrieval not data modelling, so the expressiveness of the data description languages is secondary to the access speed associated with the data manipulation language. The development of POISE necessitated research into the field of database design, though with the very specific goals specified in §3.8.5.

The object storage for POISE is an issue of persistence for portable objects in the Smalltalk environment. Research into persistence of Smalltalk Objects is a broad topic in itself, with contemporary work often involving implementing a new Smalltalk kernel[65]. These persistent objects will not port between different Smalltalk sessions and a choice of Smalltalk kernel had already been made.

An alternative approach implements a storage mechanism within Smalltalk. Smalltalk code manipulates an external file structure that is portable between sessions. Since the mechanism is in Smalltalk code, the mechanisms benefits from Smalltalk as the data description language. Any data management POISE needs can be added later. The search was for a storage mechanism that satisfies most of POISE's data storage requirements.

Within the development environment of ObjectWorks 4.0, there is an `Object` protocol for representing instances on a byte stream. This protocol provides a fundamental record format for general representation of objects similar to the structure mentioned in §4.1.1. It does not provide any management of the representation. Applications storing objects in records even need to create the medium for the record (the disk file) and remember where the record is in that medium. In contrast, the commercial class library ISAM provides for the management of object storage. ISAM is one of many task specific class libraries commercially available for enhancing the productivity of Smalltalk development.

4.5.1 Attempt 1: ISAM

Two different commercial class libraries, or 'Toolboxes', supporting storage mechanisms were examined. The first commercial mechanism was the ISAM—Indexed Sequential Access Mechanism—Toolbox[87]. The toolbox focuses on management of objects stored in records on a file. The classes in the toolbox define a technique for creating and accessing records programmable from Smalltalk. The main class defines a set of access protocols for collecting and iterating through records in sequence and by index on attribute, so ISAM is both indexed and sequential access. For example, grades could be sequenced by polymer family keeping similar grades together and indexed by trade name for direct access.

In order to store and retrieve an object in a file, stored as a record, ISAM requires them to abide by a type specification. All subclasses `ISAMrecord` class inherit this type specification. Typically, only objects inheriting from `ISAMrecord` are stored. The type specification includes a specification for the structure of each record as a template aggregating items. An item is an attribute that is an instance inheriting from the `ISAMitem` class. Subclasses of `ISAMitem` represent basic Smalltalk objects, such as text and numbers, in binary form.

The ISAM representation is like a hierarchical data model. The class hierarchy of `ISAMrecords` form a hierarchy of structural description of different records. The CODASYL network model (§2.3.1) extends the hierarchical data model with many to one relations represented by pointers between records. Likewise, the first experiment in POISE extends the expressiveness of `ISAMrecords` with a pointer-item for representing complex structures.

ISAM uses Smalltalk as a data description language but each description is explicit. It stores only objects inheriting from `ISAMrecord` and as a type of ISAM object they explicitly define their behaviour of logical representation, ie the attributes representing the state of the object. Each subclass of `ISAMrecord` transforms instances into a set of attributes and visa versa. The inheritance between subclasses forms the hierarchical relationship between records and is the only relation available to ISAM. The addition of the pointer-item allowed other orthogonal relations, but they too must relate to an `ISAMrecord`. Although ISAM uses Smalltalk as a data definition language, the specification of behaviour or protocols of the `ISAMrecord` define the syntax and limit the semantics of records. As will be seen, representing the full semantics of Smalltalk addresses many issues beyond simply the network of relations between objects.

ISAM semantics are those of fixed aggregations of attributes and they are less expressive than the binary relational file discussed. ISAM is incapable of representing arbitrarily complex data structures on a file, let alone supporting change to those structures. Storing the representations in POISE needs complex data representation and transparent access §3.8, so the ISAM toolbox was extended.

In principle, by extending ISAM to include pointer-items, a general mechanism is possible for encoding on disk specialised subclass of `ISAMrecords`, provided each instance variable in the subclass was a type of `ISAMrecord` or `ISAMitem`. The network of inter-related records on disk directly model the inter-relationships of Smalltalk objects in memory. These pointers though introduce a number of complexities, which will be addressed later. 'Circular references' are a particular problem if objects are not identified as already stored.

ISAM does not define an independent portable data store. The data model, the subclass defining the structure of the records and the semantics of the objects into which the records transform, is a property of the application, not the data store. Specific applications specify items and record types explicitly, but these specifications are part of the application, and not integrated into the data store. Data stores can port only between applications that share common class definitions.

ISAM provides both an ordered access and a random access interface to stored objects. The interface is a characteristic of the data manipulation ISAM supports. Before an application manipulates an object on the data store, it must access the object via the interface. This distinguishes the manipulation of objects on ISAM from the manipulation of objects in memory. The access requires the application designer to identify the persistent objects prima-facie, to cater for the interface protocols. A stored object can receive a message, only after ISAM retrieves the object. The task of telling ISAM to retrieve the object falls on all classes of object, which send messages to `ISAMrecords`. After processing the message, the senders must also tell ISAM to save the object.

'Transparent access' is the process of data manipulation of stored objects that does not require explicit interaction with the storage management. Accessing ISAM is not transparent. An initial attempt at transparency uses an `Enhancer` as an object proxy responsible for communicating with the ISAM storage management.

4.5.2 The Role of Database Proxies

Just as a proxy vote is handled by a third party as though the voters had voted themselves, the database proxy receives messages in place of an object stored on the database. As a subclass of `Enhancer`, the proxy is a small primary-memory resident object, and all messages evoke its `doesNotUnderstand:` protocol, which creates a primary memory representation of the persistent object and passes it the original message. The proxy communicates with the object storage manager in order to achieve this task.

Objects can only receive messages from the other memory-resident object that reference them, ie their **referencers**. Therefore, proxies exist only for stored objects with memory resident referencers. In consequence:

- There can be many more objects on the database referenced by other objects on the database for which there are no proxies.
- Proxies consume less space than the database objects they represent and so do not compromise the purpose of the database to achieve primary-memory economy.

Evoking only the one behaviour, regardless of the message is a trait of the `Enhancer`. The employment of `Enhancer` as a proxy—usually one per stored object with a memory resident referencer—is transparent to the referencer, and hence the application in primary memory, which sends the messages. Stored objects appear to receive messages like any other object.

Upon receiving a message, the proxy requests from the database management system, a memory resident representation of the stored object. The proxy then passes the message on to this object. Upon completion of the message, the proxy requests the database to store the current state of the object. Both the importing and exporting of the object to the object manager occurs transparently with respect to the message sender.

The structure of an object in primary memory is as an aggregation of object relations, see Figure 11. In accord with other network models, an object on secondary memory is a record of pointers to other records. When reading an object into memory, each of the objects in the aggregation previously without a memory resident referencer now have one; the object aggregating those relations is a memory resident referencer. By the above rule, a proxy represents each relation in primary memory. An inter-record pointer on the data store represents an Enhancer as a future proxy. This finding simplified the problem of object retrieval, and led to a re-write of Tigris™[88] and BOSS™[89], which were parts of the second data-store examined.

4.5.3 Attempt 2: Tigris and BOSS

ISAM still had the problem of requiring an explicit declaration the record structure for each type of object it stored. Tigris stores objects without need for an explicit declaration of object structure. Tigris is an indexed access mechanism with general object storage capabilities, and in conjunction with transparency through proxies, initially seemed to satisfy POISE's storage requirements. Unfortunately, problems were found with the identity of objects retrieved.

The strength of the Tigris interface is its similarity to a Smalltalk `Dictionary`. Natural language dictionaries sorts words by character order for consistent access and associates the words with their meaning. A Smalltalk `Dictionary` is a collection of object pairs, one sorted for access with the other object associated for retrieval. Tigris stores each object against a unique name used for retrieving the stored object.

Transparent access to Tigris uses the same mechanism, the Enhancer as a database proxy, which extended ISAM. The Enhancer keeps the key for looking up the object in the Tigris database. When the Enhancer receives a message it sends the key to the database to retrieve the object. The message then passes to the object returned.

Unfortunately, the Tigris behaviour was found to differ from the behaviour of a true Smalltalk `Dictionary`. The object a Tigris collection retrieves from may or may not be a copy. A copy is acceptable if the original object no longer exists within the Smalltalk environment (ie it has been garbage collected, see §4.5.8). If the original object exists, it is possible to test for the identity difference between the original and the copy. A true dictionary stores the original and retrieves the original, so no difference is detectable.

Tigris maintains a small cache for efficiency. If an object is in the cache, a subsequent request for the same object produce the same object. So, in some cases, a copy is not generated depending on the number of different objects requested from Tigris and the size of the cache.

Copies have an adverse effect on many to one relations, converting relations to many-to-many-copies. If two different referencers both request an object from Tigris they may or may not reference different objects, depending on the cache. If they reference the same object, the behaviour of one referencer can influence the behaviour of the other. Otherwise, their behaviours are mutually exclusive. Consequently, behaviours differ depending on the activity in the Tigris cache! In addition, writing one copy will over-write the other, if using the same key name to the dictionary, resulting in possible information loss.

Two parts compose the Tigris toolbox. An outer shell provides the dictionary interface and object cache. The inner part is a version of a public domain toolbox called BOSS[89], Binary Object Storage System.

BOSS receives an object from the interface. As mentioned, each object can be viewed as a record of the other objects it references. From any given 'root' object, BOSS successfully traces the network of object relations, identifies circularity, and generates a linear sequence of records. A byte stream represents this sequence of records.

BOSS stores whole object compositions and reads whole object compositions. Within each composition BOSS recognises and assigns to each object a unique identifier. These identify the relations between the objects on the stream. BOSS handles multiple references within a composition correctly.

Once a record of an object composition is on the stream, Tigris orders the BOSS to forget all assignments of object identifiers. If BOSS remembers these assignments, Smalltalk does not garbage collect the original objects and release primary memory. Consequently, Tigris does not maintain relationships between different compositions or between compositions and primary memory except to the root object, which Tigris explicitly associates in its dictionary interface.

Tigris stores each object as an independent compositional unit. If the unit is not independent, if objects elsewhere refer to parts of the composition, then Tigris will not maintain the relationship. The original part will remain in memory and when BOSS reads the composition back into memory, it will return an identifiable copy. If the references elsewhere are also saved to Tigris, then BOSS will record a second copy of the common part.

The proxy, providing the transparent access to Tigris, is also a potential solution for maintaining object identity between different object compositions within BOSS. The proxy already maintains object identity for the compositions by keeping a single reference between a proxy for each composition and Tigris. The proxy is the only object that interacts with Tigris, so it does not matter if Tigris returns the object from the cache or from disk, only one copy is ever in memory. The Enhancer logic ensures a second referencer cannot ever hold onto an 'old' copy. By applying the same principle to a finer granularity, from object composition to individual objects, a similar solution is found for BOSS.

4.5.4 The Use of Proxies to Maintain Object Identity: an Application View

Multiple objects referencing one object is a many-to-one relationship. It is quite different to multiple copies of one-to-one relationships. If each referencer held a copy of the object, and one copy changes, the other referencers would continue to hold obsolete versions. Multiple references are generally dependent on the changing state of the common object.

However, using Enhancers as database proxies can partially solve the problem. When saving an object with multiple references, after copying the object to a record, replace it with a new Enhancer with the appropriate key. Replacing an existing object with a new object without referencers causes Smalltalk to garbage collect the original object and therefore preventing a second memory copy from existing. All the saved object's referencers now access the same Enhancer. If any of them are then subsequently stored, BOSS will discover the Enhancer as part of their composition and can identify the part already recorded on file. The Enhancer maintains a unique 1:1 relationship to the stored object. This relationship holds regardless of the stored object's memory state. When reading a composition, Tigris remembers the Enhancers it generates. Before generating any new Enhancer, Tigris checks the Enhancers already in memory. If a second composition attempts to read the same Enhancers a second time, Tigris substitutes the existing Enhancers in the second composition, thus preventing copies of the Enhancer to the same object.

By securing uniqueness of the proxy, if an object changes state, all references both on and off the database, can locate the new states through the key kept by the proxy. The behaviour of the proxy reflects the change and all the referencers will reflect the change in their own behaviour.

4.5.5 Attempt 3: The WorkBase

Although the majority of objects are uniquely owned (ie in one-to-many relations), any object is potentially a future member of a many to one relation. At the point of storage there is no guarantee an object will not multiply share in the future. A provision for a general object store must preserve the identity of all objects stored. Changing the BOSS system to maintain identity of every object throughout the data store requires a major change to the Tigris-BOSS model. Instead, aspects of the BOSS system were used in a new custom-built database called the WorkBase. The WorkBase takes advantage of the proxy concept, introduced to give the store transparency, for maintaining object identity. Whereas POISE applies the proxy concept to all databases, the management of proxies for object identity is particular feature of the WorkBase.

The object–orientation and management of database proxies is an original feature the `WorkBase` contributes to object storage systems. From an applications viewpoint of data base storage, the key advantages of the `Enhancer` technique is as a proxy for stored objects:

- The proxy separates all data management activities from the persistent objects. The management of the object on secondary storage is not a property of the object. The objects class does not define the behaviour, and objects of the same class may either be persistent or not.
- Database access is transparent to the applications using the object. Code manipulating objects does not specify the storage conditions of the objects it is manipulating. Changing the management of an object from primary memory management to a proxy that accesses secondary storage is transparent to the code manipulating the object. Smalltalk code appears to handle native objects in the same way, irrespective of whether they are memory-resident or a proxy retrieves them secretly.

The data management activities—the data-retrieval strategy, object caching, object updating, housekeeping of the store, etc.—are functions of a database's storage model. The proxy aids an application's interaction with the database but does not improve the storage model yet. The `WorkBase` storage model uses the proxies in the design of its storage model to solve many difficult problems that object data structures introduce. It helps in maintaining object identity, in handling circularity and in caching.

In the storage model of Smalltalk, §4.1.1, objects record relationships by reference with other composing objects. The record is physically an ordering of *`objectID`*s. The storage model of a `WorkBase` is the same. The *`objectID`* in Figure 11, pp. 98, is a different number but the `WorkBase` uses its *`objectID`* in the same way. Records representing an object are a list of *`objectID`*s. The `WorkBase` finds the storage location of any object from the *`objectID`*. Proxies reference persistent objects by remembering the *`objectID`*.

4.5.6 The Use of Proxies to Maintain Object Identity: a Database View

Multiple referencers can exist outside the database, and these hold a common proxy. Multiple referencers can also exist within the database. If the correct data model to be built when reading a referencer, it must hold the **same** proxy as all other referencers in primary memory, whether that proxy is representing an object in memory or not. That same proxy will guarantee the behaviour, which it replaces, is common to all sharing referencers.

Consider two database objects that both reference a common third object. When the database reads the first object into primary memory it creates a proxy that references the common object. When the database reads the second object, it cannot create a second proxy to the common object, since multiple proxies will create copies of the common object. How does the database find out whether a proxy already exists in memory for a given database reference? This question needs to be answered for all references the database creates when reading an object into primary memory. For the majority of cases, there will not be any other object sharing the reference, but every reference is potentially shared.

The WorkBase keeps a record of every proxy in memory. Each request to reference a database object the WorkBase searches the records to see if the proxy already exists.

The WorkBaseMapping is responsible for finding existing proxies. Based on a hashed dictionary, the WorkBaseMapping keeps an index of proxies against *objectID*.

When reading an object into memory, the WorkBase, checks each *objectID* against the WorkBaseMapping. Finding the *objectID* also locates, by association, the current proxy for the object in primary memory, otherwise the WorkBase creates a new proxy. Other objects on the data base may share the new proxy, so the WorkBaceMapping adds the *objectID* associated with the new proxy.

Creating a new proxy does not imply that the WorkBase reads the persistent object, which the proxy references, into primary memory. Only if the proxy receives a message will the proxy read the object into primary memory. The referencing object, the object currently being read, must send a message to the proxy. For the majority of the new proxies, this will not happen. The majority of the proxies in the WorkBaseMapping, and hence in memory, are **passive**. They represent a link to an object on the WorkBase that the pattern of message passing has yet to cross.

When repeatedly accessing an object, the WorkBaseMapping schema provides an efficiency benefit. Once a passive proxy receives a message and the object the proxy represents is in primary memory, the proxy holds the copy of the object in primary memory. The proxy is then said to be **active**. Active proxies do not read from secondary storage but use the memory copy for further messages and so respond much faster. Locating an active proxy in the WorkBaseMapping is the same as for passive proxies. A passive proxy reads from secondary storage, whereas a message passing to the active proxy uses its primary memory copy.

A special case of active proxies is when referencing classes. Every object references a class to provide the data description and behaviour for objects. Like all objects, *objectID* (specifically called *classIDs*) identify the memory resident classes. The WorkBaseMapping index them in the same way as other shared objects. The WorkBase finds classes like any other object. The object associated with the *objectID* is not a proxy. The primitive interaction of instantiation with the class structure prevents the use of a proxy. Instead, the association is with either the class or an object representing an obsolete version of a class. Class versions are described later, §4.5.11.2. The WorkBaseMapping cross reference minimises the retrieval of classes for representing objects on the database, and also manages versions of data definition between objects on storage and in memory.

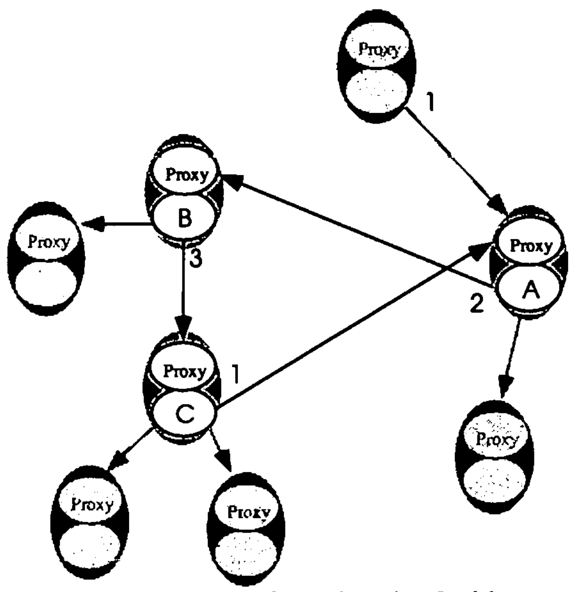


Figure 20: Scanning circularities

4.5.7 Object Circularity

Object circularity is a special form of multiple references. An example is given in Figure 20. Object circularity originally arises from practical difficulties with the deepCopy concept within Smalltalk. A deepCopy is a recursive copy of an object and all its composite objects. A deepCopy attempts to make a copy of an object that is totally independent of the state of the original, down to the finest detail. Usually applied to strictly hierarchical compositions, the result is a copy of all members of the hierarchy. A strictly hierarchical composition is one where each 'higher' object composes of 'lower' objects. Problems occur when an object is not a strict hierarchical composition. Since Smalltalk does not guarantee such structures, it is possible for a lower component object to refer to a higher object creating a loop, or circularity. When deep copying, the higher object is already copied, and unless the recursive copying is coded otherwise, the copy of the lower object will request a recursive copy of the higher object. A procedural loop forms, which runs infinitely — or at least until memory is no longer available.

A similar difficulty exists in the BOSS system. BOSS reads a whole object composition at a time so creating a copy similar to the deepCopy, but differs in the medium the original is on in secondary storage. Circular references could exist and an extensive mechanism is necessary for identifying the condition and structuring the composition correctly.

The general mechanism using proxies (§4.5.6) manages all shared references, not just circular ones. This mechanism manages object identities, and it has made the majority of the BOSS code dealing with circularity redundant. Instead, the database management was re-developed, adapting ideas from BOSS, to create the WorkBase.

An object read from the data store often refers to other objects on the data store. Reading those objects and the objects they reference *ad infinitum* results in large and proliferate transactions, in comparison to other data models[90]. Instead, the WorkBase substitutes proxies for the object references (except numbers and characters and some other special cases), so it only reads the object receiving the messages. Therefore, proxies need only read the next object, never the whole composition.

4.5.8 Proxies and Memory Management

The proxy's purpose in the architectural design of POISE is to mediate indirect database references at the interface between Smalltalk applications and persistent data. A further development of this design is to augment the definition of proxies with a control strategy for fine-tuning the **lifetimes** of resident object representations. Effectively, this implements rules of permanent versus temporary memory residence.

The underlying memory-reclamation controller of the application language monitors the lifetime of objects. Generally, an object dies when there are no other objects referencing it. In Smalltalk, this controller is a Garbage Collection Manager[91].

The primary memory life of a persistent object dies to free memory. The Garbage Collection Manager role extends to keeping track of active proxies occupying primary memory. The manager initiates the removal of old, active proxies by requiring that they 'commit' the memory-resident data to the data storage mechanism. Both the number of message-sends and a FIFO (First In First Out) queue determine the expiry choice, or age, of active proxies. Overall, the manager does not force such a decision until available primary memory runs below a threshold. The manager determines the length of the FIFO queue according to the platform's main memory characteristics, and dynamically adjusts it when necessary.

The WorkBaseCache manages the FIFO queue. It provides tuneable parameters for deciding the number of active proxies to commit. If they commit too eagerly, it causes run-time penalties. At the extreme, it commits immediately after the proxy services a message. Objects that receive messages many times within an enclosing context[92] will lose the benefits of caching between messages.

The lifetime starts whenever a process sends a message to the proxy. A passive proxy will become active or an active proxy will reset its lifetime. Proxies accessed often record a short lifetime, whereas proxies that are not will quickly grow old in primary memory and return to secondary storage.

The reason for committing objects is to release main memory, not for transaction integrity. At present, WorkBaseCache implements the policy of committing the oldest objects whenever the Garbage Collection Manager notifies a WorkBase that memory is running low. A complementary facility — implemented as a Smalltalk background process — utilises spare processor time to keep occurrences of WorkBase objects down to a maximum number. This then lessens the effect of system failure causing loss of transaction changes.

4.5.9 Implementation of the Database Proxy

The proxy provides a service to any object requiring persistence. The proxy behaviour is inappropriate under Object because not all objects will persist and persistence is not a behaviour of an object. Persistence is a service the language provides to objects, just as the ability for objects to receive messages is a service of the language. An alternative to the proxy would be to re-write the virtual machine so *objectIds* could point to representations on a file in the same way as they point to representations in object memory.

The database proxy is the simplest of the Enhancers in POISE. Specifically, the Enhancer implementation of the proxy is a subclass called the WorkBaseEnhancer. Each proxy has two exclusive states, either active or passive. The behaviour of the proxy is significantly different, requiring a change in protocol depending on the state. When active, messages pass directly to a memory resident persistent object, but must reset its lifetime. The WorkBaseCache can also direct the active proxy to commit changes, and become a passive proxy. The passive proxy interacts with the WorkBase and changes to the active state. Normally two different classes of object define differences in protocol. A proxy could reference different classes of instances to change state. The different instances would provide the differences in service described. Two objects would construct each proxy with a proxy interface and an associated object for the different management. An alternative was found which does not require using memory for extra objects. Smalltalk provides a primitive for changing the class of an object. A proxy changes its class and thereby changes the protocols it inherits.

There are two different subclasses of the WorkBaseEnhancer. One is the ActiveWorkBaseEnhancer when the proxy is active and the other is the PassiveWorkBaseEnhancer. When the PassiveWorkBaseEnhancer receives a message, it requests from the WorkBase a memory resident copy of the object on file and changes itself to an ActiveWorkBaseEnhancer to service the message. The ActiveWorkBaseEnhancer records the time it receives the message then passes the message to the persistent object. The active proxy is part of a cache system that uses the last access time to determine which proxies to commit to disk, so releasing primary memory. When committing, the proxy passes the hidden object to the WorkBase, which checks if the hidden object differs from the record on file and updates it accordingly. The active proxy then changes to a passive proxy. Since the passive proxy no longer keeps record of the memory resident object and the active proxy was the only referencer, Smalltalk garbage collects the hidden object and releases memory.

4.5.10 File Representation: Adaptations from BOSS

BOSS represents objects in byte arrays. Typically, the arrays of bytes compose a sequence of records on a disk file. Each array starts with an identifying signature. The objects on the file can always be found by searching for the signatures. Normally the byte count from the start of the file locates the objects. The signatures help overcome corruption, an event common in a developing system.

Bytes support 256 different states and grouping the bytes into sets of four gives 2^{32} states. Each of these states in an *objectID* uniquely identifies a different object. This is a finite number, which limits the number of objects a WorkBase represents. BOSS groups pairs of bytes, as the *objectIDs* are only unique within each object record.

After the signature, the WorkBase starts the object record with the object's own *objectID*. There is a key from *objectID* number to the location in the file. If ever this key is corrupt, the WorkBase can iterate through objects in the file by locating signatures and re-constructing the key.

The remaining representation is like the representation in Smalltalk primary memory (§4.1.1). Each reference is an *objectID*, starting with the class of the object or *classID*. Like the primary memory model, characters and integers have special references that are uniquely encoded to identify the character or integer without further reference. Consequently, numbers and characters do not require proxies.

BOSS differs from the WorkBase in the structure of its records. A record contains many objects, each contributing to a composition. After representing one object, BOSS immediately represents the next until it completes the whole composition. The WorkBase puts each of these objects in their own record since any one may be shared in the future. The consequence is a need for more *objectIDs* and the increase from 2-byte representation of *objectIDs* to 4-bytes.

The storage model is a relatively direct extension of the Smalltalk internal primary-memory representation policy. The internal primary-memory representation derives from the format in the class definition of stored instance variables. Trans-migrating the class definition from the Smalltalk environment into the database is important in automating object-storage.

4.5.11 Storing Class Information

The storage model discussed so far captures only the relationships between instances. This is an incomplete description of an object as it inherits protocols from its class, which defines the semantics of each object.

4.5.11.1 Requirements for class data definition storage

In a class-based inheritance language, the templates classes supply the data definition of instances. The retrieval of stored instances also requires sufficient template information for interpreting the stored structure. A class name is sufficient information to find a class in a Smalltalk application and hence the class definition with a template and protocols. This is fine if class data definitions are static, but POISE provides for evolutionary modification of domain-modelling classes, so classes are not static. The template in memory may no longer match the structures used in storage.

Maintaining the behavioural integrity of all objects inheriting from a class throughout its evolution requires much more representation than simply the name of the class. The class name representation is the simplest object specification using an application-based class, and it provides the least integrity. Complete integrity is possible with a data-based class. Data-based classes completely represent both protocols and structure on the database and re-construct the class in primary memory on demand.

Initially the `WorkBase` only considers objects inheriting from application based classes. These are simpler and faster to retrieve but the class name alone provides insufficient integrity. To entertain evolutionary class descriptions, the `WorkBase` supplements the class specification with a **version template**.

The name of an application-based class is an insufficient representation for an evolving class. When a class changes its structure, instances in primary memory immediately coerce to the new structure while both the new and old structure are known. The instances in secondary storage remain unchanged. The `WorkBase` needs the information about the old structure when it encounters these obsolete instances. Their data structure differs from the current class structure. The order of *`objectID`* relations in the record depends on the structure of the class when saved and the order can have no correlation with the current class structure. Without a valid class template, the ordering of *`objectID`* relations is lost, and with it the semantics of records stored on the database.

The version template provides integrity, or more correctly a consistency between the behaviour of instances in memory and the stored instances of the same application-based class name. The `WorkBase` coerces stored objects to the application's class structure. The `WorkBase` stores a version template for each class of any stored instance, which encodes:

- The class's *name*.
- The class's `format` number.
- A sequence of names of all instance variables defined and inherited.

The version template is only a partial description of the class. For it to be of any use the class's storage *name* must match the name of a class currently in the application. The `format` memorises the size of the object on the `WorkBase`. The instance variable names relate the stored object with its content, the *objectIDs* in the record. If names in the list match names in the application class specification, the stored object adopts the semantics for the name in the class.

If a class has not evolved, then the order and names of instance variables is the same in the class template and the class specification in memory. The records on store order the relations the same as new instances and a simple transcription of information from the record to a new instance re-creates the stored object in primary memory.

If the data structure of the named class changes then this causes the addition, removal and change in sequence of named instance variables. A difference means the records on the store are old **versions** belonging to an obsolete class definition. Instance variables common between the version template and the current class specification can map data from the obsolete object on the database to a current object in memory on retrieval. The stored object is then correctly consistent in behaviour with the current class specification. Since this may not have been the intended behaviour of a stored object, the integrity may still be in question.

4.5.11.2 Version management of evolving data definitions
When encountering an instance of an old version for the first time, the `WorkBase` creates `ClassVersion` object and records it in the `WBMapping` under the old class's *classID*. The `WorkBase` gives the current class a new *classID* as soon as it saves a new instance to distinguish it from old versions. New instances map to the class in the `WBMapping` and further encounters with the old version immediately map to the `ClassVersion` object.

The `ClassVersion` object and `Class` are polymorphic with respect to protocol for creating new current instances from `WorkBase` records. On instantiation a `ClassVersion` object compares the given old class definition with the current definition and makes a map between instance variable names. The `ClassVersion` keeps a reference to the current version of the class in memory. With this information, it can generate any current instance from the obsolete `WorkBase` record.

The `WorkBase` represents all classes by a record containing the version template, even the current classes. They therefore have *objectIDs* associated with them or, more specifically, *classIDs*. When the `WorkBase` reads an object of the class for the first time, it reads the version template first. As with all objects the database reads, the `WorkBaseMapping` makes a reference. If the class of the stored object is the same version as currently in memory, the `WorkBaseMapping` keeps a cross-reference between the *classID* and the memory resident class. If the class is an old version, it keeps a cross-reference between the *classID* and the `ClassVersion`. The `WorkBase` will use the object the `WorkBaseMapping` associates with the *classID* to generate instances from the record.

Whenever a class changes, Smalltalk notifies `WorkBases` through a 'dependency' link (a dictionary associated relation as opposed to an instance variable). The `WorkBase` removes the association between *classID* and the changed class from the `WorkBaseMapping`, because that *classID* no longer designates the current class version. If the `WorkBase` subsequently writes an instance of this latest class version, it will treat the instance the same as if the class had never been written to the `WorkBase` before. Hence, it writes a new version-template and assigns a new *classID* for the class.

Consequently, a `WorkBase` may hold multiple versions of the same class with a distinct version template and unique *classID* representing each version. The `ClassVersion` object performs the translation

- From the instance variables, the stored version template describes the ordering in the arrays of stored objects.
- To a fresh instance of the current application's class, matching where the descriptions are similar.

As each version template has a unique *classID*, it creates a unique `ClassVersion` instance to perform the conversion of instances that refer to that version template.

4.5.11.3 Data migration of instances

An accidental consequence of the `ClassVersion` management is the longevity of named instance variables. The removal and addition of the same instance variable leads to loss of data in primary memory, §4.1.2,4.4.2. This data survives in persistent objects on secondary memory and the `ClassVersion` can correctly return the data on return of the instance variable.

Data migrates from the record on file to a new instance of a named application class. In most cases, the `WorkBase` finds the application-based class matching the *classID* in the record is current. In this case, the migration is simple. The record, an ordering of *objectIDs*, migrates to an array of the same order containing `PassiveWorkBaseEnhancers`, with the corresponding *objectIDs*. The `WorkBase` then coerces the array to the named class by the changeClassToThatOf: primitive method. This changes the *classID* of the array in memory to that of the named class.

If the record is obsolete then, instead of the application-based class, the WorkBase finds a ClassVersion. The ClassVersion matches the named instance variables in the version template with those in the current representation and associates the names with the current index. Using the name-index associations, the ClassVersion translates from an 'old instance index' to 'new instance index', (equivalent to a relational projection). The rest of the process is the same as current versions, but the process re-orders the proxies in the array according to the translation. This translation is precisely what happens to instances in memory during schema evolution §4.4.

4.5.11.4 Limitations of application inherited classes

The application-based classes are still the only repository of protocols defining the semantics of instance variables. This is fine if there is only one application. Problems occur if there is more than one Smalltalk session where the application-based classes in each differ. Since the WorkBase does not distinguish classes of the same name in different applications there is the potential for semantic differences.

Relying on the definition of classes within the application can cause behavioural discrepancies in persistent objects in a common data store, accessed by different applications. This can breach the encapsulation of the stored objects. The greatest risk is if one application permits a state in the object not acceptable to the behaviours in another application. Typically, one application's behaviour assigns a type of object to a named relation that other applications do not permit. This is possible since although the two applications must both have classes that agree on the data structure of the objects, there is no agreement on the method code that accesses and changes the data in the structure.

Inheriting behaviours from application-based classes is not satisfactory for a distributed system where applications could access the data structures incorrectly. The WorkBase does not provide any means of ensuring correct, consistent access. An alternative is to store a class completely on the database that defines the complete behaviour of the objects stored. Complete storage of a class is more in line with the object representation of fully-fledged OODBMS.

4.5.11.5 Requirements and limitations of behaviour storage

A notable example of OODBMS that stores the complete class behaviour is within the architecture of Gemstone 2.1[65]. Gemstone runs an object manager on a server machine. It executes services (in a custom language, OPAL) remotely in the server, upon request by an application (eg in Smalltalk) running on a client machine. In contrast, the `WorkBase` strategy aims to escape the need of a programmer or end-user to establish whether a computation occurs as part of an application or part of a remote object manager. A research attempt involving this same goal was the Rekursiv project[93] into producing a seamlessly integrated object memory and secondary data storage, by developing special hardware. In the absence of a ready hardware solution, POISE can execute message-sends to an object only in the main memory occupied by the application, since the application contains the appropriate class manager and its the compiled code of its methods. From this follows the pragmatic language design decision to manage only structural-definitions of instances as the main data management task.

The `WorkBase` is not, essentially, a computational vehicle; it does not provide computation in addition to that with a Smalltalk application. Nevertheless, it provides a storage format for byte-compiled Smalltalk code, such as the `CompiledMethod` class discussed in §4.4.5, and for the syntax of a message-send to an object. It exploits these formats to provide commands for executing services when under authorisation by an application. In this way, the `WorkBase` can store a sequence of code as an object for later evaluation.

A method is a sequence of code associated with a class of objects. In particular POISE provides the option of selecting particular methods of a class and makes them persistent along with the class version template. This selective policy is suitable for the evolutionary information-modelling requirements of POISE, since the major part of an application running in Smalltalk will be the behaviours of domain-modelling objects, which in turn describe the peculiar activities of that application. This was superseded quickly by the more general mechanism of complete class storage.

4.5.11.6 Storage of a Smalltalk class

POISE evolves the description of polymer classes. In order to make these changes persist the `WorkBase` must also store the class. The storage of the class is especially complex because of the relationships it has with the client image.

Smalltalk constructs the class like all other objects in the language, §4.1.2. It behaves like a class because it inherits those 'class like' behaviours from the `Class` class. One of the behaviours a class inherits is the ability to generate other objects using the information contained in the class object. This is a primitive behaviour that directly accesses the second instance variable and must contain an integer which describes the format of the instance. The instance keeps a reference to the generating object (the class) and it is known as its class. For this instance to work as an object the class object must meet two other criterion. The class has another class object (or *nil*) in the third instance variable as the superclass and a method dictionary in the fourth instance variable. This is the most basic requirement for getting a class to function. Other requirements are necessary for the object to function as expected in the Smalltalk environment, but are not necessary to get the class's instances functioning in the Smalltalk environment.

The format, being an integer, is easily stored. The superclass can use any of the aforementioned class representations but not via a proxy. The superclass must inherit directly from the application or the database. The method dictionary though contains many difficulties. First, a proxy cannot be put in the method dictionary place since the virtual machine expects a dictionary. The dictionary links protocols to their names. With the list of names, the `WorkBase` creates a special proxy `MethodDictionary`.

Reading all the protocols of a class and its superclasses is unnecessary. All objects the `WorkBase` reads are by request from a proxy receiving a message, so the `WorkBase` need only read the protocol matching the message. The proxy `MethodDictionary` contains all the names of the protocols, but associates them with a `CompiledMethod` that requests the real protocol from the `WorkBase`. The message look-up occurs as normal, and so causes the `WorkBase` to import the protocol. The rest of the difficulties are with representing protocols on the `WorkBase`.

`CompiledMethods` are, at their most basic, a byte array, which contains pseudo-code, compiled at runtime, and hidden from the user. `CompiledMethods` are simple sequences of code that the `WorkBase` can easily represent, which are complicated by references to variables that are outside the scope of the immediate calculation: instance variables and global variables.

`CompiledMethods` refer to instance variables by an index, which must correctly correspond to the indexes in the receiver's class. It is for this reason that methods associate uniquely with a single class. Smalltalk searches for methods by class and the structure guarantees that the method is compiled for the class. Without the class, the method is meaningless since it refers to instance variables by a number that have arbitrary meaning in any other classes.

Globals include class variables and pool variables[2]. All methods that access a unique global variable share a reference to a common association, which contains a name for the global in the key and an arbitrary object in the value. When a method sends a message to a global association, the value of the association receives the message. As this is a primitive function the association must be an association not a proxy. The object, as the association's value, that receives the messages may be a proxy if the global is only within the domain of the database. However, a difficulty arises if the global is meant to be an application resident object. In this case, the Workbase must find the application global before any messages are sent to the global, ie when the method is read into memory.

The most complex issues in storing a class is the class's relationship to a superclass and any global variables. In fact, the superclass is just another global variable. References to globals should be resolved by a separate policy object from the server that has explicit knowledge of the global name space on every client and the server. Other than that, the class is treated like any other object on the WorkBase.

4.5.12 Summarising the WorkBase

The WorkBase satisfies the private storage requirements for a single POISE user (§3.8.5). The unique feature of the WorkBase is that when it reads objects it resolves differences in the schema between client and server, which allows the client schema to change independent of the schema of individual objects represented in the WorkBase. Implementing this feature was simplified by the single connection policy between the POISE application and the private single-user WorkBase. Most DBMS focus on supporting multiple connections and consequently complicate the client's dependence on the server's schema, which the server endeavours to maintain consistent for multiple clients.

An advanced object storage system is a better description of the WorkBase than a DBMS because of its single-user restriction, and the application executes all object behaviours, not the WorkBase. The WorkBase advances object storage because it is capable of representing complex objects, including the classes of polymers in the hierarchy and the behaviours of engineering properties developed by the user. In addition, with the help from the database proxy, the objects maintain their unique identity, usually lost when object storage systems remove objects from the application environment.

Any object is a candidate for storage by a database management system. The DBMS must retrieve the object back into primary memory before processing any messages directed at the object. A general proxy Enhancer provides a transparent interface between objects of an application and objects held in the DBMS. Messages sent to database objects via the proxy Enhancer activate the enhanced behaviours for requesting the DBMS bring the object into primary memory and for updating the database with any changes. This role of the database proxy is an abstract feature that can apply to any application-database interface.

A specialisation of the proxy manages object identity on behalf of the WorkBase. This lets the WorkBase retrieve objects individually, rather than whole compositions. The WorkBase only retrieves the objects necessary for the active process. The majority of proxies only remain in memory as long as a process using them remains active, thereby promoting memory management. Other specialisations include an object lifetime property of the proxy for collaborating with the WorkBase's transaction and memory management.

Although the WorkBase is an object storage system, it does adopt many database management features. In addition to the requirements of the knowledge representation, there are database management requirements, which manage the limited computing resources. The WorkBase collaborates with the Smalltalk memory management, only committing transactions when memory is low thereby maximising the utilisation of primary memory, and committing all transactions when the user terminates the application. Transaction management is a complex feature of many OODBMS. This simple policy takes advantage of the single-user restriction of the WorkBase, since in multi-user applications long transactions prevent other user access.

The one resource the WorkBase does not manage effectively is the disk file it uses to store the state of objects. This aspect is not pursued because there was an ample resource for experimenting and many DBMS address the problem adequately.

Another weakness in the design of the WorkBase was the efficiency of the DBMapping. This object provides the primary index for the database. Currently the DBMapping adopts the indexing behaviour of the Set to provide a simple hashing algorithm with linear probing. This is known to be one of the least effective mechanisms and doesn't take into account the future growth of Set like objects. Further research[37], concludes a dynamic hash table[94] is more appropriate. It also allows for many smaller WeakArrays of one size, rather than one big array that needs to change size. This advance to the WorkBase was unnecessary for the experimental purpose of POISE.

4.6 Summarising the Implementation of POISE

A drawback of the class–instance relationship is that the class usually defines instance behaviour exclusively. The Polymer classes in the POISE classification do not define grade behaviour exclusively. Grades have the properties from the taxonomic classes extended with orthogonal properties by a technique of manipulating messages sent to the grade object. Orthogonal properties are any property not related to the classification based on chemical and molecular structural composition, eg those relating to general geometric shape and process. Instead, a separate class template defines orthogonal properties that can apply to any polymer. POISE abstracts this manipulation of messages, which makes orthogonal behaviour possible, into a class of objects called the Enhancer.

The Enhancer provides a behaviour sharing that differs from the explicit dynamic messages between individual objects and differs from the static implicit behaviour shared between classes and groups of instances. The Enhancer provides implicit dynamic empathy between individual objects. These are general descriptions of types of behaviour sharing from the Treaty of Orlando[5]. The Enhancer is therefore a general enhancement to the class–instance paradigm, which Smalltalk implements. The implementation of the Enhancer uses the error trapping mechanisms built into Smalltalk. Although these details are specific to Smalltalk, the behaviour sharing that the Enhancer's characterises is significant for representation.

The Enhancer is a general tool for enhancing any object behaviour. Classes inheriting from the Enhancer add specific functionality to enhance the behaviour of a number of objects independent of their classes. The inheritance statically binds these specific behaviours to the class. An Enhancer called the ScopeEnhancer dynamically binds behaviours from multiple objects. The ScopeEnhancer shares behaviours with the flexibility often found in languages with delegation. Zucker has already demonstrated delegation useful for representing the evolution of the design description, or the 'application perspective'. The ScopeEnhancer demonstrates a similarity with delegation by sharing an enhanced behaviour between more than one object. This initial experiment suggests the Enhancer can support Zucker's objectives in a class–instance languages.

Another class inheriting from the Enhancer resolves deductive inheritance (§2.2.4). The CombinedDataAbstraction is one of a number of objects with enhanced behaviour that produce the abstract polymer behaviour. The CombinedDataAbstraction inherits many abstract properties and deduces a single abstract property. An enhanced instance from each concrete Polymer classes (the ones with instances) represents their abstract properties. The Polymer classes themselves manipulate their messages so they inherit the abstract properties. The result is a hierarchy of abstract polymers that generalise properties typical of the grades they classify, from which the designer can interpret design benefits.

POISE takes advantage of Smalltalk features that are not characteristic of the class–instance paradigm. Smalltalk does not distinguish between the development and runtime states of software. For this, the Smalltalk environment includes tools normally associated with development, such as a compiler. The compiler and other supporting classes let Smalltalk define, declare and instantiate objects during runtime. This promotes the development of software by prototyping, but does not distinguish prototyping during development from the application of prototyping during runtime. POISE uses these features to evolve classes through interaction with the user, thereby providing dynamic schema evolution. POISE specialises the software tools to orientate schema changes around polymer properties, thereby empowering the domain expert, rather than the Smalltalk programmer, to manipulate the polymer classification.

The development tools for evolving Smalltalk classes were found highly inefficient. While specialising these tools for the polymer classification, a new type of protocol objects was defined that is independent of a physical model. These protocols did not require re-compiling when the physical model changes. Therefore, these protocols are independent of the class, which defines the physical model of instances. This permits the definition of partial template objects, PTOs, which are a tool for managing protocols outside the class. Partial templates are a re-useable set of protocols that may be installed consistently on many classes. The PTO in POISE represents polymer properties. They provide a classification independent way of relating the similarity between properties and their contextual application.

Chapter 5 A Populated, Fully Functional POISE.

POISE supports the user with a number of interfaces. The aim of these interfaces is to reflect the data models discussed. There has been no formal cognitive design of these interfaces, and this is not an argument supporting them as the best way to display the data models. They provided a way to learn about the polymer domain and the models used. They were used in the course of developing the data model and by Spedding[8] in her research into appropriate classifications of polymer information.

The first user interfaces developed were the `Comparitor` and the `HierarchyEditor`. These interfaces had direct relevance to the representations of the abstract polymer and classification. Initially a command line interpreter started a Smalltalk process that opened these interfaces. Command lines are very flexible but require a specific skill for use. Consequently, a central interface was developed for the designer and other novice users. This central interface represents an active POISE session. The first task of the novice user is to start the POISE session.

5.1 Entering the Smalltalk Image

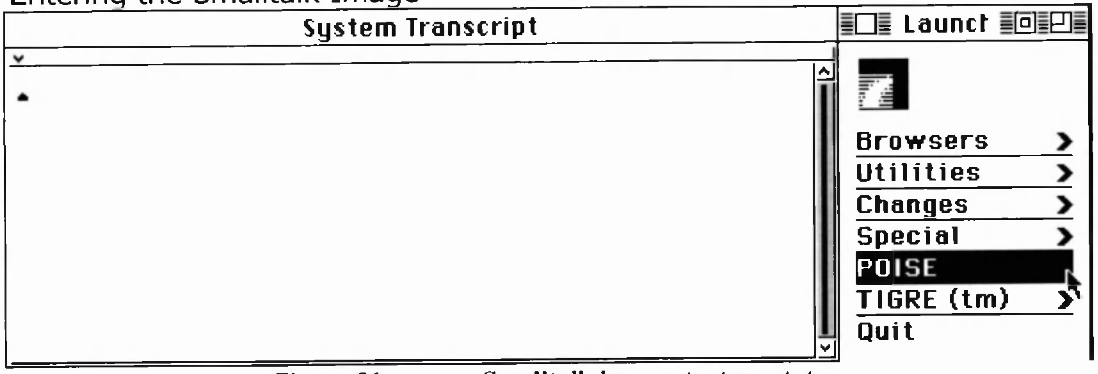


Figure 21: Smalltalk image start-up state

The application known as POISE resides with the development tools in a common Smalltalk image*. The user's access to the development tools is through a window known as the `Launcher` (Figure 21). The `Launcher` provides a list of options for the user to select. POISE adds an extra option as a gateway into the world of POISE. In a Smalltalk image containing only POISE, and without the development tools, loading the image automatically evokes this option.

The second window in Figure 21 is the System Transcript. This window provides a general display of messages to the user. The window is also a text editor providing the programmer with a place to type and request the evaluation of Smalltalk syntax. In an image containing only POISE, such a tool would not be available since it would enable the user to modify the image in an unpredictable manner. Later we introduce a specialised window for notifying the user of POISE's activities.

* A Smalltalk Image is the description of all objects in primary memory when starting Smalltalk. The Image plus the Vertial Machine make up the whole Smalltalk Environment.

On entering, POISE opens Tigre's `FileChooser` screen (Figure 22), asking the user to locate the `WorkBase` file containing the polymer information and session details.

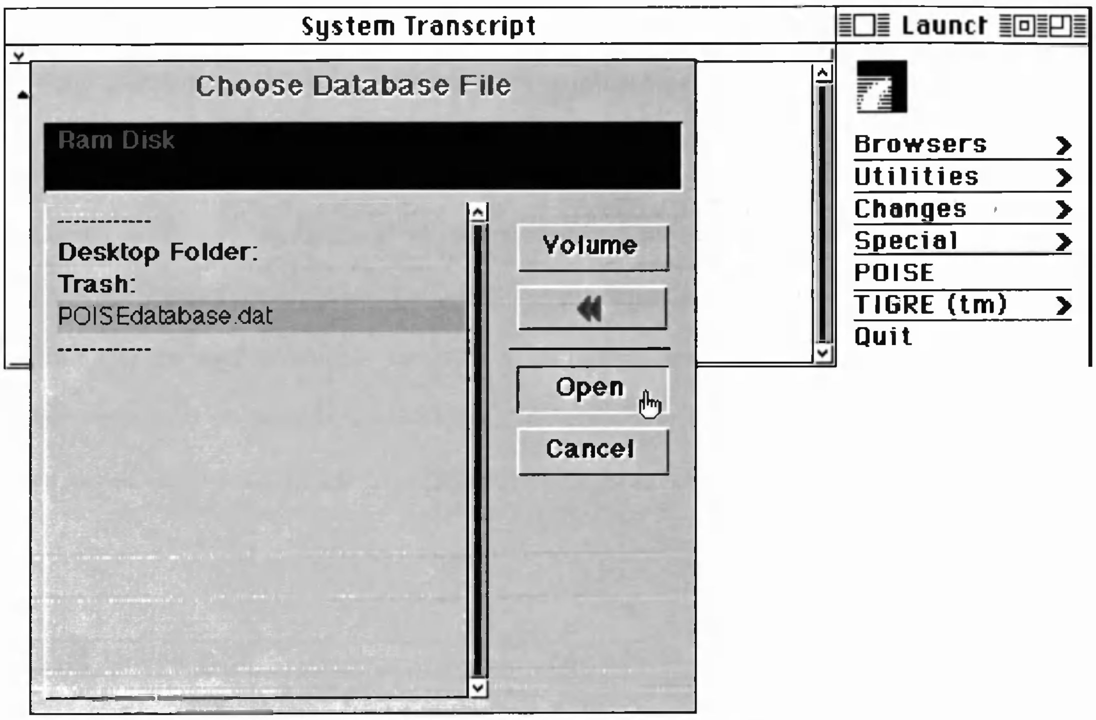


Figure 22: Selecting the WorkBase

At this point POISE locates references to polymers in the `WorkBase` and adds to each `Polymer` classes an index of their polymers. Opening the `WorkBase` creates a background process responsible for managing the garbage collecting in the `WorkBase` file. A message appears in the transcript notifying the user of this process and gives the number of grades POISE finds.

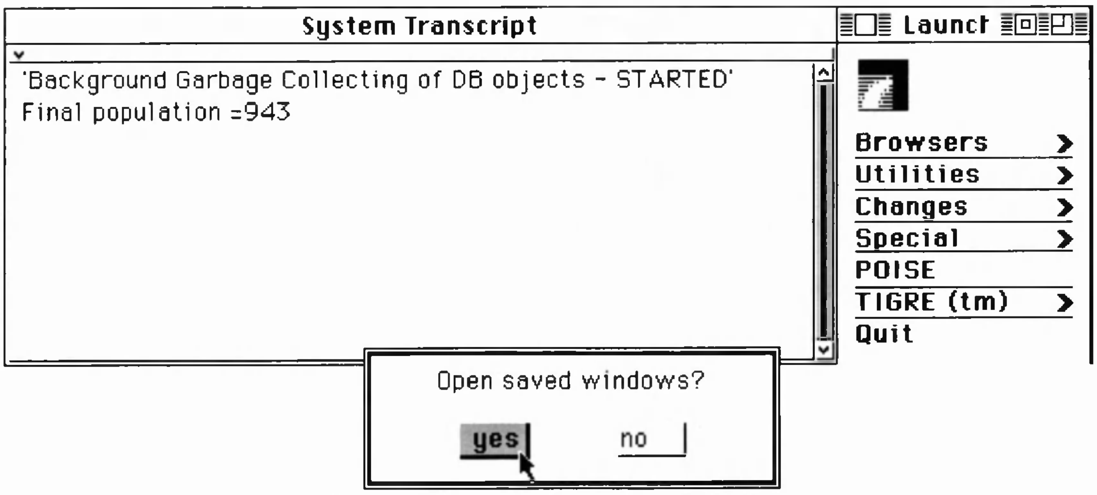

```
'Background Garbage Collecting of DB objects - STARTED'
Final population =943
```

Figure 23: Re-starting POISE

A new window opens, asking if the user wishes to open the windows stored in the `WorkBase` (Figure 23). These windows were saved in the last session, recording any lists of grades or particular comparisons of polymer families the user was processing during the last session.

If screens were not saved, or the user opts not to open them, then only the POISEsession screen opens (Figure 24). The Launcher and System Transcript screens close automatically.

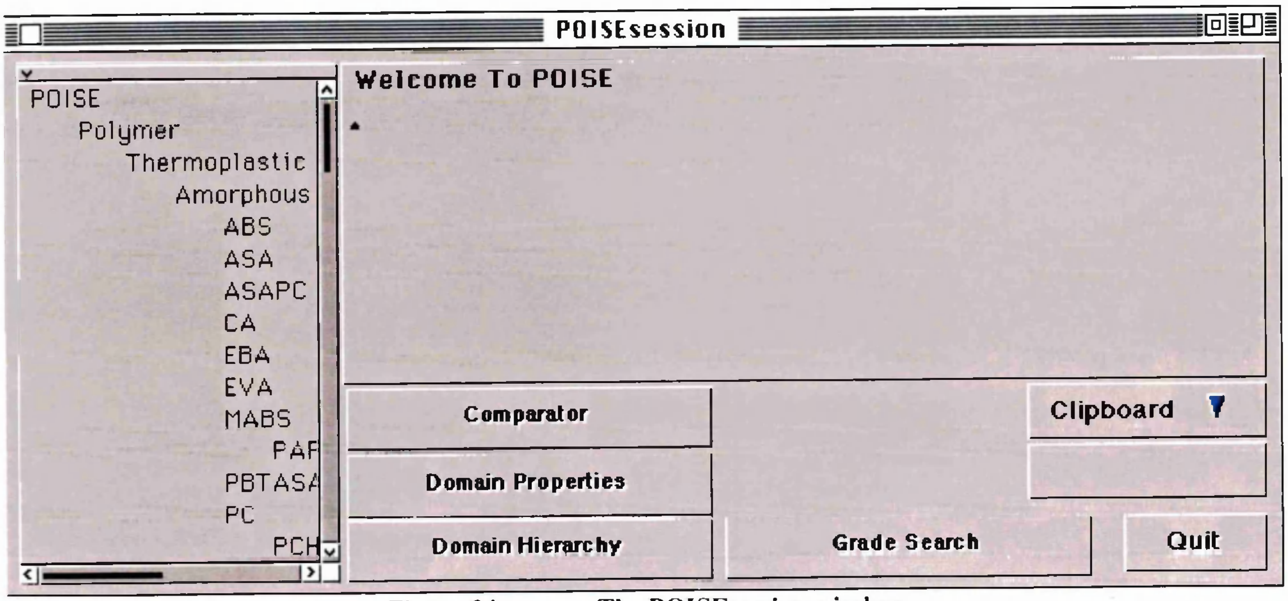


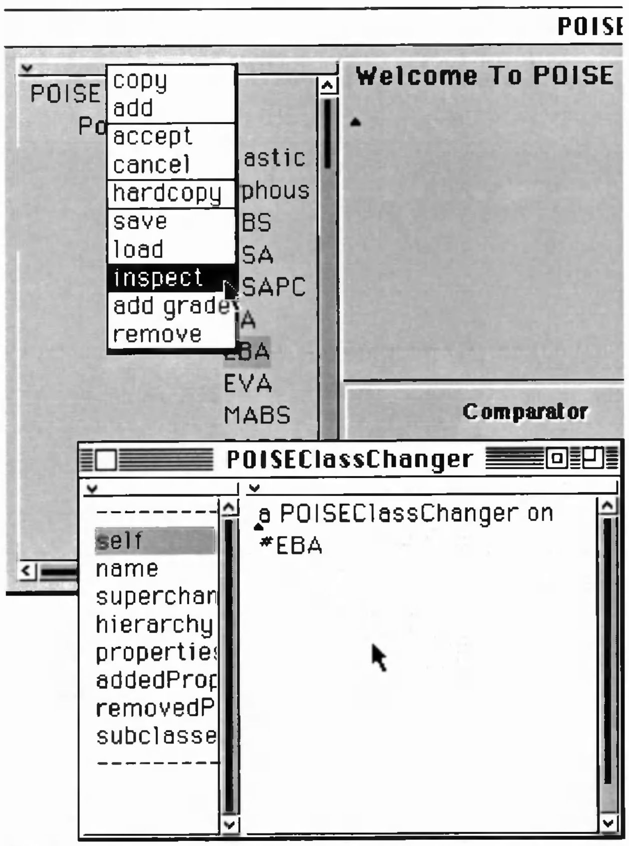
Figure 24: The POISEsession window

5.2 POISEsession

The POISEsession window (Figure 24) is a central access point to all other tools in POISE. The POISEsession has three parts. On the left is a sub-view containing a hierarchy of Polymer classes. Top-right is a specialised subview that replaces the functionality of the transcript. The third part is a set of 'button' views, which open various types of POISE screens when the user selects them. The button marked 'clipboard' and the button below are exceptions, providing functionality to the transcript part of the POISEsession (in §5.8). Quit returns back to the Launcher, offering to save any open screens.

5.2.1 The User Defines the Classification

The sub-view on the left of the POISEsession is a complete hierarchy. The hierarchy is inside a ScrollingWrapper that provides scrolling functionality. This sub-view is a functionally cut-down version of the HierarchyEditor window described earlier (§3.3.3). The user evokes functions by selecting the menu-bar, ⌄—— in Figure 24, causing a menu of options to appear. A complete list of the functions is in Table 7. Figure 25 demonstrates the 'inspect' option. The inspect command provides visual access into individual Polymer classes. In this case, the text for 'EBA' was previously selected in the sub-view, providing the context for the inspect command.



```
--------
self
name
superchan
hierarchy
propertie
addedProp
removedP
subclasse
--------
```

```
a POISEClassChanger on
#EBA
```

Figure 25: POISEsession- viewing the schema

The inspect option opens a general inspector on an instance of the class `POISEClassChanger`, in Figure 25. These objects record the state of changes to the hierarchy and the inspector provides a concise report of changes per class. A general inspector provides reports the state of the object's instance variables for this purpose. The sub-view on the left lists the instance variable. Selecting one displays the contents of the instance variable in the right sub-view as text, (the '`printOn:`' text behaviour all objects have.)

The list shows the instance variables that store changes to the POISE classes while the user operates on the hierarchy (in schema §3.3.2, editor window §3.3.3, implementation §4.4). The *`superchanger`* references the `Changer` of the superclass the classes inherits. *`Hierarchy`* is a reference to the overall hierarchy model. On

Copy	Assigns the variable 'Clipboard' to the object currently selected (in §5.8).
Add	Adds a new polymer class by first asking the name of the polymer family and subclassing off the class POISE.
Accept	Compiles all changes to the hierarchy. To this point only a description of the changes are kept. POISE does not change the classes and instances of polymers that the hierarchy describes until the user selects the accept option.
Cancel	The record of changes is reset.
Hard-copy	The text in the subview is sent to the printer.
Save	A disk file saves the configuration of classes.
Load	The `Hierarchy` compares the configuration of classes in a disk file with the current configuration and records the necessary changes.
Inspect	Opens a general object inspector on the record of changes for the selected class.
Add grade	Creates an instance of the selected class. The user is prompted for the name of the new instance and left with a `Grade` inspector window on the new grade (in Figure 26 below)
Remove	Marks the selected class for removal from the image. (Grades remain on the `WorkBase` but the key to them is lost until the user adds a class of the same name and re-opens the `WorkBase`.)

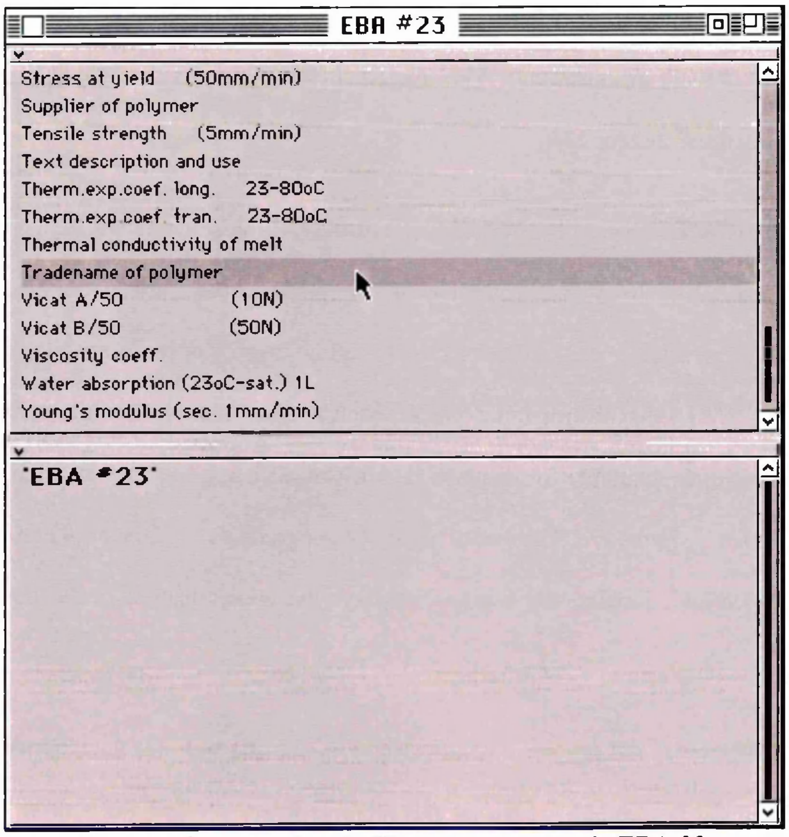
Table 7: User menu-functions over hierarchy editor

every change, the `POISEClassChanger` creates a `Checker` to consult the hierarchy and ensure changes are consistent with the inheritance and class naming rules of a Smalltalk hierarchy. Properties, `addedProperties` and `removedProperties` are used by the `HierarchyEditor` for confirming modification to the properties of the class.

5.2.2 Adding a Grade

The hierarchy interface is an easy place to identify a class of polymer. If a user selects a class, they can add a grade to the class by selecting the appropriate menu item.

When adding a grade, POISE provides a default name; simply the number of grades known plus one concatenated with the grade's family name. In Figure 26, a grade inspector views the new EBA grade. At this point, the grade inherits properties from the polymer family, but no specific values are known except for the grades name. The view provides the list of properties in the top sub-view. Scrolling to the property 'Tradename of polymer' and selecting causes the bottom sub-view to display a text representation of the property's value. The user can change the name here.


EBA #23

Stress at yield (50mm/min)
Supplier of polymer
Tensile strength (5mm/min)
Text description and use
Therm.exp.coef. long. 23-80oC
Therm.exp.coef. tran. 23-80oC
Thermal conductivity of melt
Tradename of polymer
Vicat A/50 (10N)
Vicat B/50 (50N)
Viscosity coeff.
Water absorption (23oC-sat.) 1L
Young's modulus (sec. 1mm/min)

'EBA #23'

Figure 26: Grade View over new grade EBA 23

The user can select any of the properties in the top list, modify the text in the bottom text view and, through the menu of the text view, accept the change. The view, in conjunction with the property object, parses the text, interprets a value for the property, and assigns it to the grade. In this way, the user can fully specify a new grade or modify or delete an existing grade. An example of a CAMPUS grade is given in Figure 35.

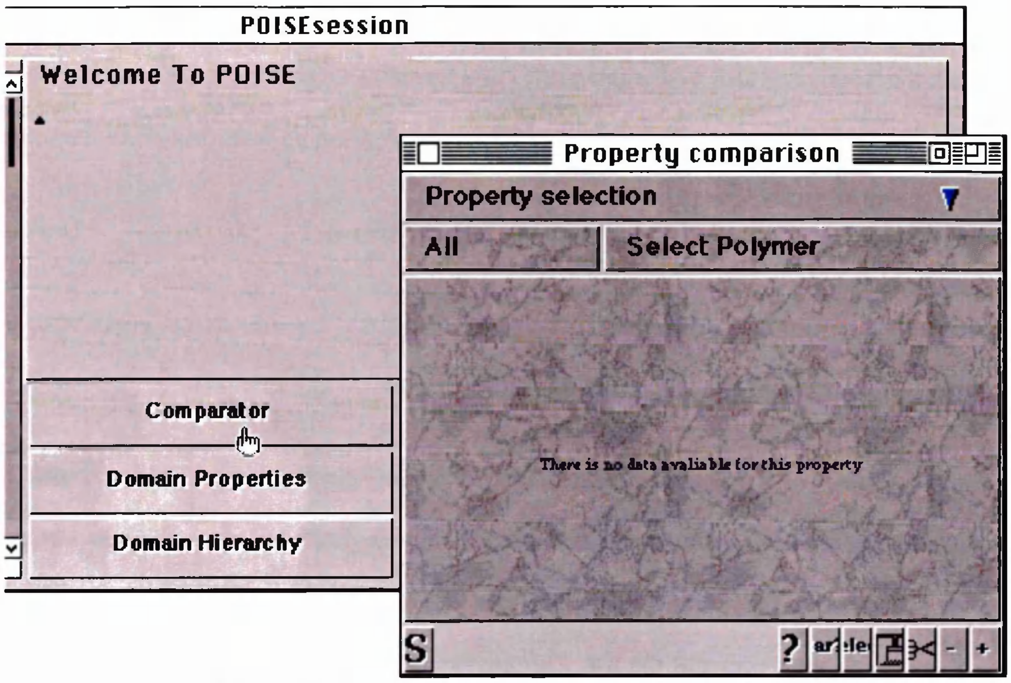


Figure 27: Starting a property comparison

5.3 The Comparator

The hierarchy in the POISEsession window provides the user with the tools for defining the initial database schema and data entry. As the user changes the schema and enters data, POISE constantly modifies abstractions over the domain of knowledge. The 'button' in the POISEsession named Comparator provides access to these abstractions (Figure 27).

The property comparison window, or Comparator, is a display of the generalisations derived from the grades in the domain. POISE generalises the properties of the grades from each polymer family, forming an abstraction. POISE merges these generalisations, and without further domain analysis, forms higher order abstractions (§3.5,§4.2.7.1). An abstraction exists for each Polymer class in the hierarchy, each contributing to a list of grade categories available for display. The list of abstractions appears when the user selects the button 'Select Polymer', Figure 27. The list of property generalisations appears when the user selects the button 'Property Selection'. Finally, the window displays the generalisation in the main centre sub-view, Figure 28.

In the example of Figure 28, the property Young's modulus is selected and the classification Crystalline (more correctly partially crystalline). Like all the other property objects, the Young's modulus property specifies a generic histogram subview to display the generalised data. The Comparator window locates the generalisation from the Crystalline class, which transparently accesses the WorkBase. The WorkBase stores the generalisation as a set of value occurrences. The view coerces the set into a data type suitable for display. A histogram object in this case.

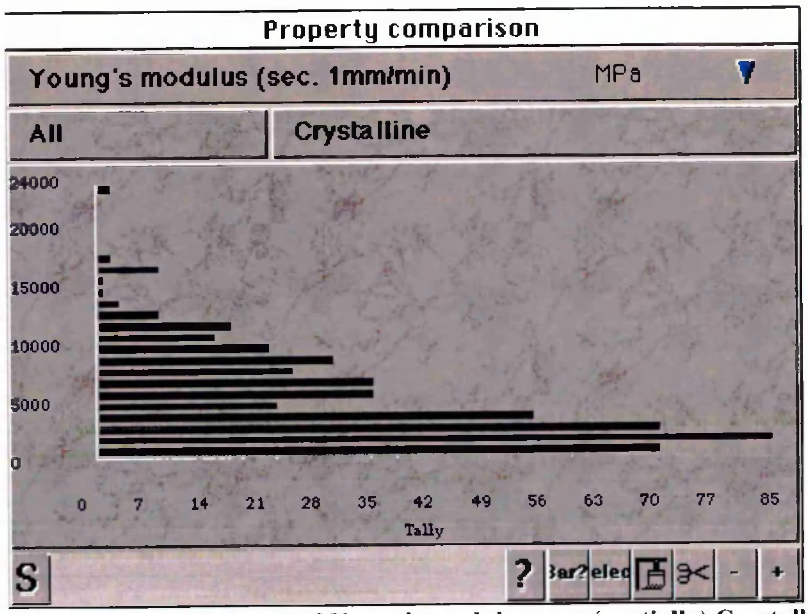


Figure 28: Abstraction display of Young's modulus over (partially) Crystalline polymers

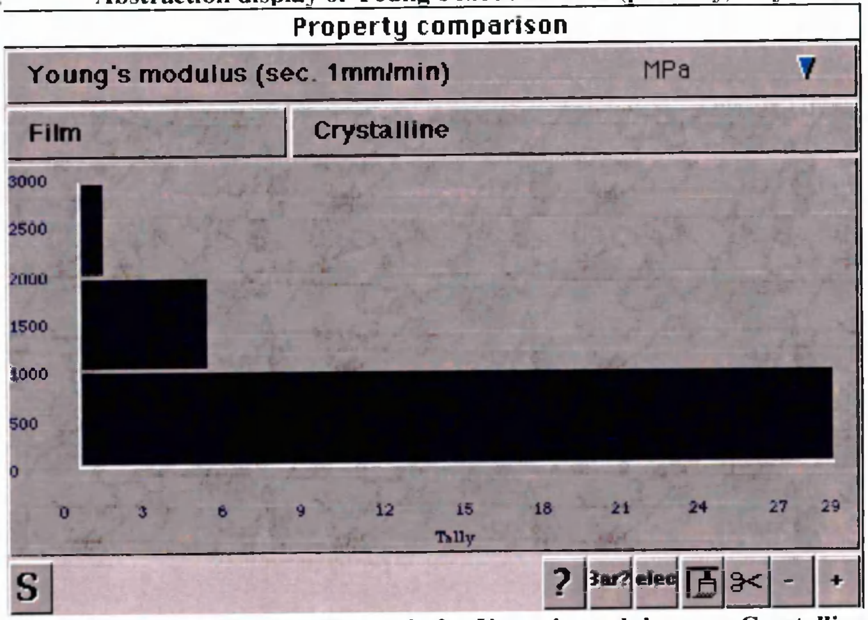


Figure 29: Viewing films only for Young's modulus over Crystalline

The histogram subview is fully scalable. When the user adjusts the size of the window, the sub-views size scales equally, changing the size of each ordinate proportionally. Increasing the window size increases the room available for displaying the ordinate. The number of labelled intervals also increase as room becomes available to accommodate the font size, which does not scale.

The axis on the left is the units of the property, displayed next to the property name, MPa for Mega-Pascals pressure. The bottom axis is a tally of occurrences so is unitless.

The button labelled 'All' refers to the whole classification of `Crystalline`. Any `Polymer` class with grades exhibiting orthogonal properties will generalise the orthogonal properties into `MultipleDataAbstraction` (MDA) objects, as described in §4.2.7.5. An MDA will report each orthogonal class template in use within in the polymer classification. Currently orthogonal classes include `Fibre`, `Film` and 'used-by `Lucas`'. The selection of this button allows viewing of one of these classes or, as seen here, all. In Figure 29 the selection changes so to view only polymers used as film.

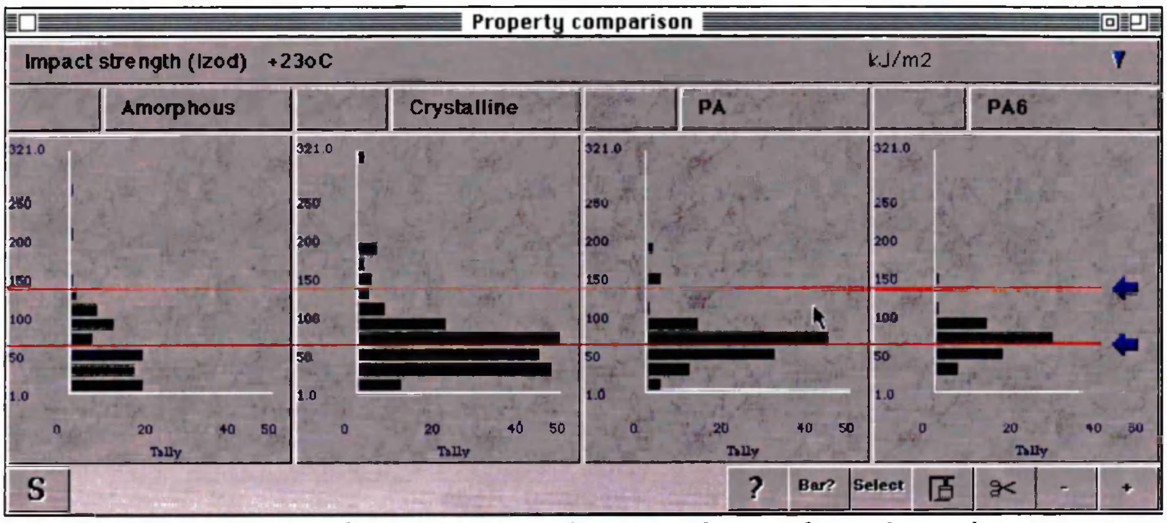


Figure 30: Comparing abstractions strength across four polymer classes

The `Comparator` allows comparison across abstractions sharing the same property in the same window. By selecting the '+' button, in the bottom right corner, the window adjusts the size of the sub-views to accommodate a second histogram display. Each display has its own classification buttons for selecting the abstraction. In Figure 30 four classifications compare their impact strength. In Figure 31 the films of `Crystalline` contrast against all of `Crystalline`. Note the property value axis scales across the largest range over all abstractions in the display.

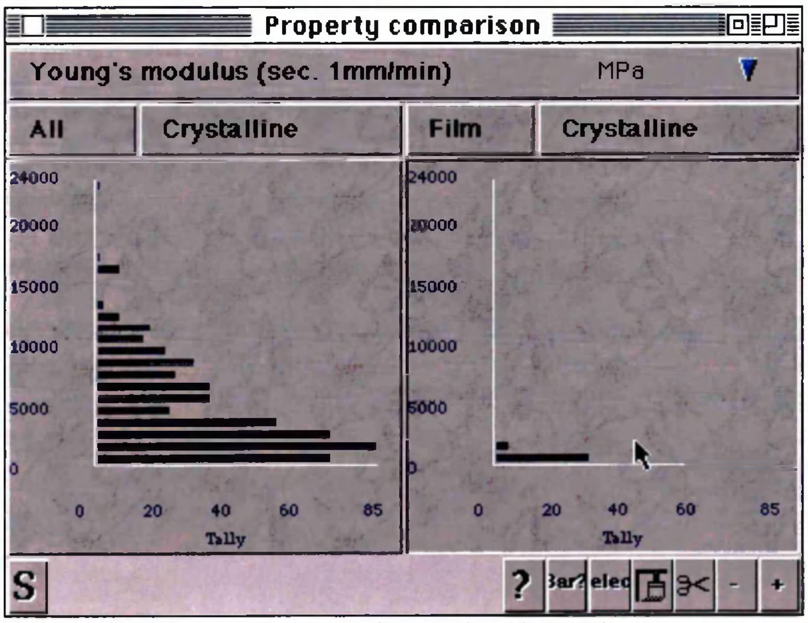


Figure 31: Comparing abstractions

In Figure 30 there are two red lines across all the histograms with a blue arrow at the ends. These bars appear when the user selects the 'Bar?' button. The red lines move along the property axis forming an upper and lower limit. While these bars are active, the user can select the grades that fall between the lines. While the lines are not active, the user can select the individual bars of the histograms, which change their colour to red when selected.

The selected grades are not collected until the user presses the 'select' button. Finding the grades requires a search through all the grades in the appropriate classes in the classification. The process can limit the search by inferring the absence of grades in a subclass that falls outside the range being searched. If the selection is across the class `Polymer` then potentially the search covers every grade in the `WorkBase`. This is time consuming compared with selections across specific classes, eg `PA`, which are quick.

The search results in a window listing the classes where grades match the selection. Selecting the class displays a sub-list of the grade's Tradename. The complete list of grades forms a sub-shortlist, Figure 34 p169, which can contribute to a global shortlist available to all POISE windows when the user closes the window (in §5.5 below).

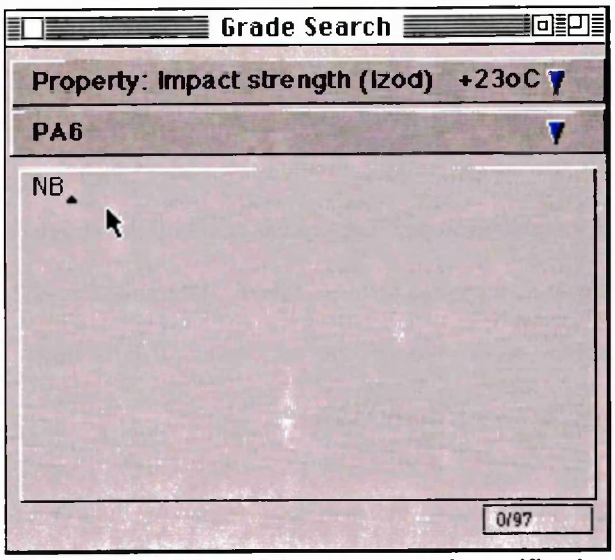
The `Comparator` can display the global shortlist as a user-generated abstraction. If the 'S' button is pressed in the bottom left corner of the `Comparator`, it displays only the grades in the shortlist. Each abstraction view still limits the display by classification and property. To display the whole shortlist, the classification would need to be '`All`' and '`Polymers`' with the shortlist button selected.

Clicking on the background of any subview in the `Comparator` selects it. The border of the subview inverts to indicate the selection. In conjunction with the scissors button the subview can be cut to the clipboard, or deleted with the minus ('-') button. The `Comparator` can paste in a sub-view from the clipboard, which has the same function as adding a new subview and setting the classification.

Finally the '?' button displays a text window containing help information. Help is generally seen as an important function but not critical to this research. The button demonstrates the simplicity of integrating an auxiliary support system, such as help, into POISE.

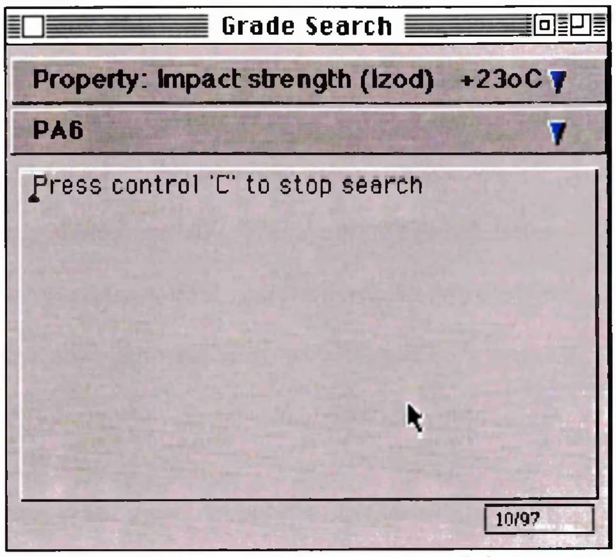
5.4 Grade Search by Query

The `POISEsession` screen provides access to an alternative to the `Comparator` for finding grades. The 'Search' button opens a Grade Search window. Like the `Comparator` it is possible to limit the search to a selected classification. The search is currently limited to the domain of a single property but this was only to simplify the tool. The potential exists for a complex query at the same level as the `POISEsession's` transcript window (in later §5.8).



NB

Figure 33:Grade search by query search specification



Press control 'C' to stop search

Figure 32: Grade search by query search in progress

In the example, Figure 33, the property selected is impact strength and the classification PA6. The user enters a query in the larger central view. The view compiles the text into values, as interpreted by the property object. In this case, the property interprets 'NB' as 'No Break', the extreme result of the impact test where the specimen fails to break.

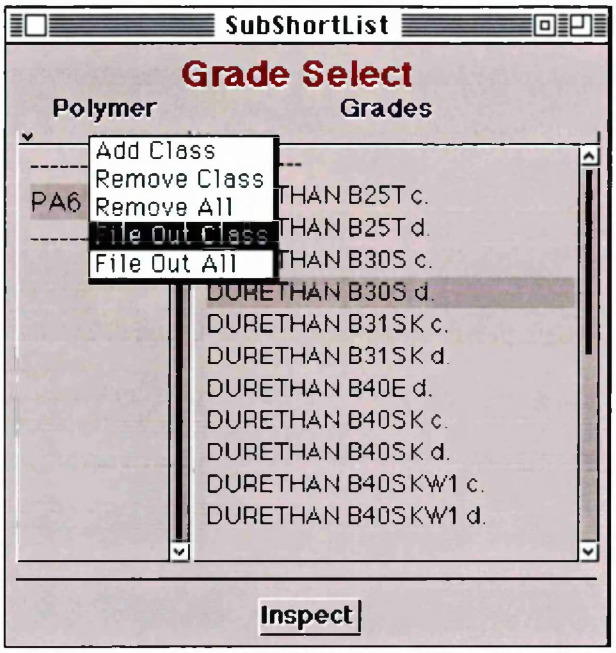
The view displays the size of the search in the bottom right corner as the number searched/total to search. As the search proceeds the number searched is periodically updated as in Figure 32. The user can terminate the search by typing control 'C'. Such facilities are necessary in a system with the potential for large linear searches for which it is not optimised.

5.5 Shortlisting

When the search is complete, a `SubShortList` opens with the results. In Figure 34 the only class is PA6 since the domain of the search was limited to this class. When the user selects PA6, it displays the grade's Tradename in the second list. At this point, the menu above PA6 allows the removal of the class or the generation of a disk file containing the set of grades. The menu also allows the addition of a whole classification or for all classes to be removed or filed out.

The menu above the grades enables the user to clear the list or remove individual grades. Grades may also be filed into a text file in a DIF format (in §3.1.1).

When closing the `SubShortList` window, POISE asks if this set of grades is to join the global shortlist. The global shortlist is a set of grades like the sub-shortlist, but it is unique for a single POISE session. The user may perform various searches generating sub-shortlists, which are logically `ORed` together in the global shortlist. The `Comparator` can limit the domain to just the global shortlist which provides a logical `AND` with the Comparator's own selection criterion. With these browsing tools, the user obtains some binary logic over the selection.

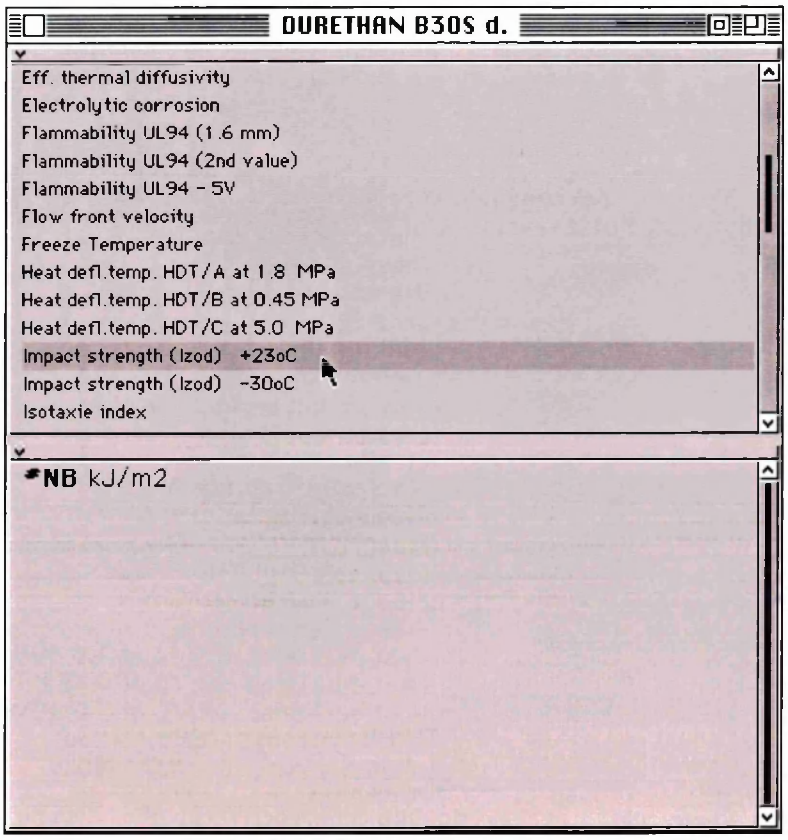


THAN B25T c.
THAN B25T d.
THAN B30S c.
DURETHAN B30S d.
DURETHAN B31SK c.
DURETHAN B31SK d.
DURETHAN B40E d.
DURETHAN B40SK c.
DURETHAN B40SK d.
DURETHAN B40SKW1 c.
DURETHAN B40SKW1 d.

Figure 34: Sub-shortlist a user defined set of grades

5.6 Grade View

The 'inspect' button in Figure 34 becomes active on selecting a grade, and opens a grade view window. This is the same type of window as used in Figure 26, for creating a grade. Figure 35 gives a complete description of the DURETHAN B30S d grade.



Eff. thermal diffusivity
Electrolytic corrosion
Flammability UL94 (1.6 mm)
Flammability UL94 (2nd value)
Flammability UL94 - 5V
Flow front velocity
Freeze Temperature
Heat defl.temp. HDT/A at 1.8 MPa
Heat defl.temp. HDT/B at 0.45 MPa
Heat defl.temp. HDT/C at 5.0 MPa
Impact strength (Izod) +23oC
Impact strength (Izod) -30oC
Isotaxie index

*NB kJ/m2

Figure 35: Grade view initial text description and specific property

Until the user selects a property to view, the bottom subview of Figure 26 displays the text property on the grade (in §3.2.5.2). When the user selects a property, the subview displays the value. In the case of Figure 35, the property impact strength matches the query. The behaviour generating a text representation of the value belongs to the property object and the value object, not the window or the subview. This allows different text formats to exist for different properties.

5.7 Property Definition.

The user can add properties and modify existing properties using a `PropertyEditor`. Returning to the `POISEsession`, the user opens an editor through the "Domain Properties" button. A menu opens for selecting the subject property, either a new property (Figure 36) or an existing property, which lists either the classified domain (Figure 38), an orthogonal class (Figure 37) or unassigned.

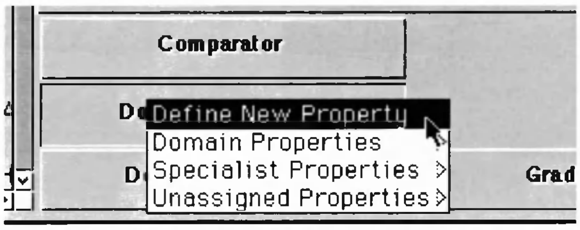


Figure 36: Selecting subject property- Start new property

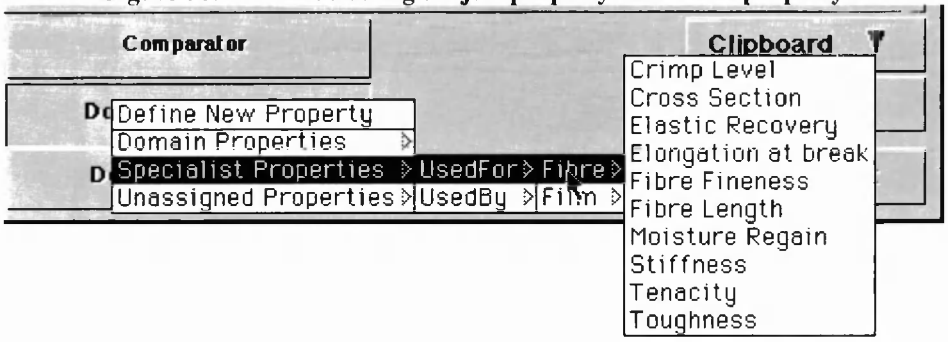


Figure 37: Selecting subject property - orthogonal class used for fibre

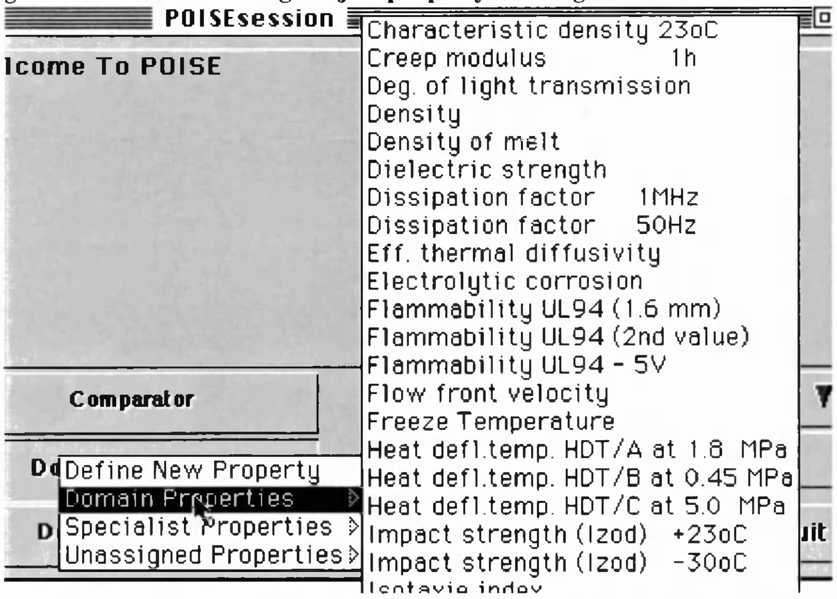


Figure 38: Selecting subject property- classified domain

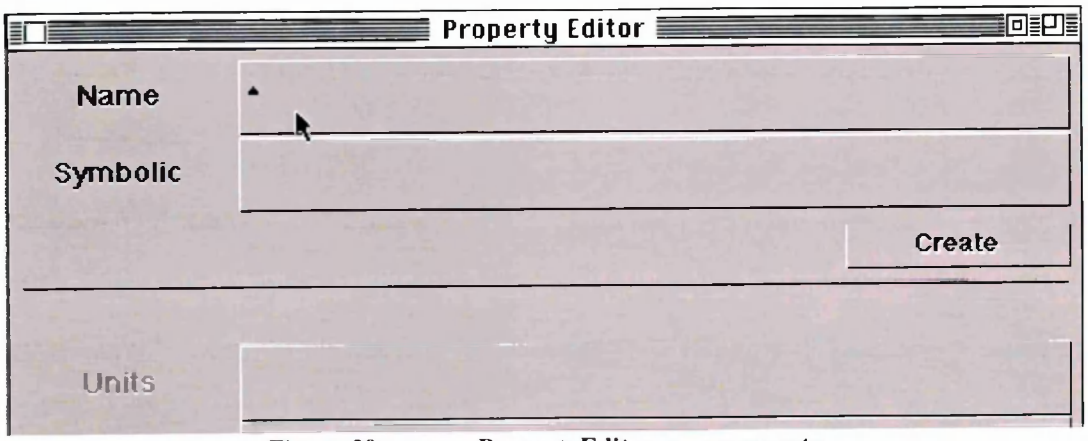
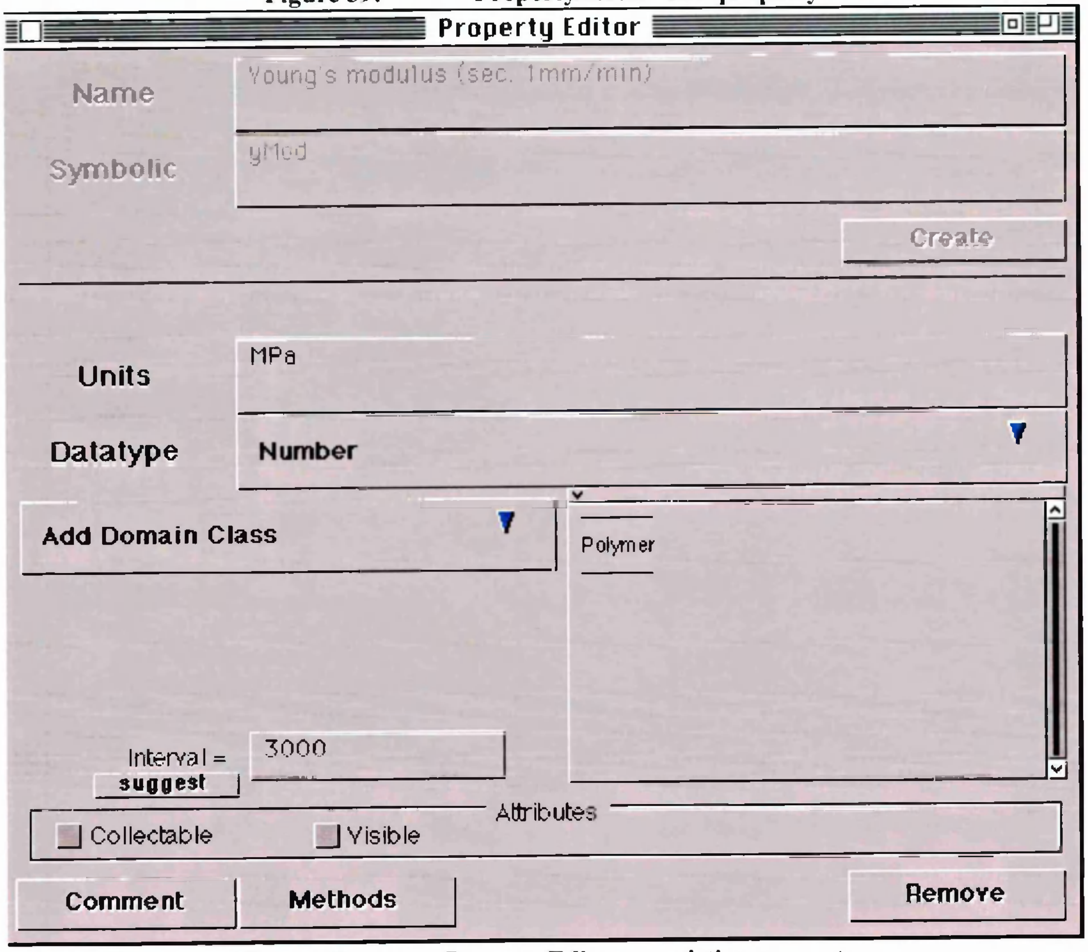


Figure 39: PropertyEditor- new property



Name: Young's modulus (sec. 1mm/min)
Symbolic: yMod
Units: MPa
Datatype: Number
Polymer
Interval = 3000
Collectable
Visible

Figure 40: PropertyEditor on existing property

Selecting a new property will open an editor with most options greyed-out and inactive. The user must fill in the active fields before accepting the property, adding it to the domain. These active fields, Figure 39, include; the property name, the string used by interfaces for property selection, eg Figure 38; the property symbol, which the editor checks for uniqueness and `Polymer` classes use to name instance variables and message selectors for the property.

After the user enters the essential descriptions, the create button generates the property object, setting other attributes to the default states. The rest of the window becomes active allowing the editing of these defaults. This is the same for windows on existing properties, eg Young's modulus in Figure 40.

Editors on existing properties do not permit the descriptive name string to change or the name symbol. The symbol identifies the property syntactically, the string semantically. Other property attributes can change through the life of the property.

If it is possible to generalise a property into histograms, then the user can specify the value in the interval field for distributing the histogram bars. The suggest option causes the interface to query the `Polymer` class for the property-values expressed by all known grades. If any grades define the property and assign values, then the query returns a set of those values. From this set, a rule of thumb calculation, derived by trial and error, suggests a value for distributing the histograms. The interval field displays the value.

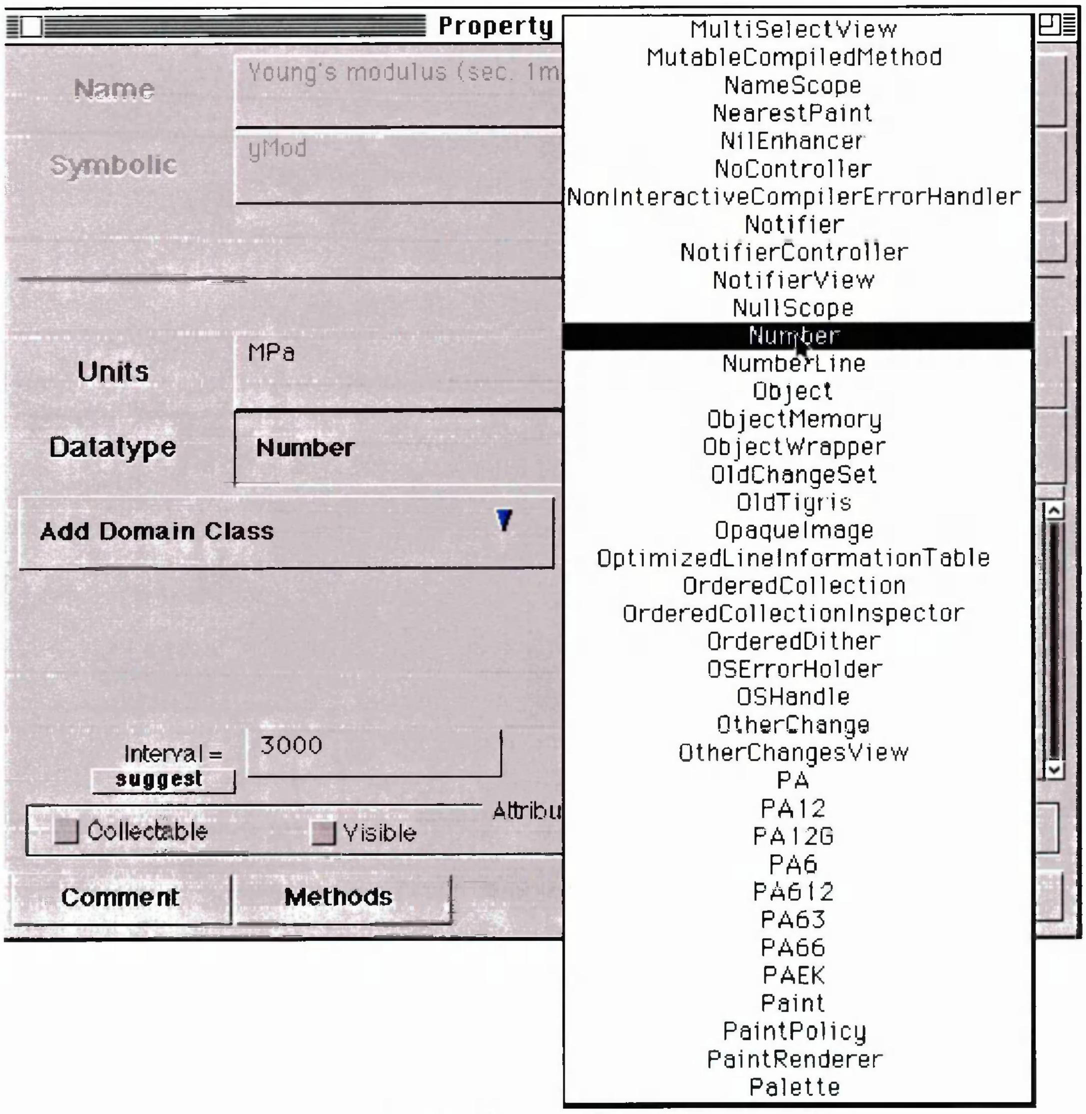

Name: Young's modulus (sec. 1m
Symbolic: yMod
Units: MPa
Datatype: Number
Interval = 3000

Figure 41: Datatype

The datatype attribute (Figure 41) is a Smalltalk class. A property can select any Smalltalk class to represent the value. Theoretically, the attribute should be a type, not a class. Standard Smalltalk does not distinguish types beyond a single class so POISE uses a class. Polymorphic classes, of the same type, which do not share a superclass, can not both represent a property since currently only one class can be selected.

If the data of the property can abstract into histograms then the "Not Collectable" attribute (Figure 42) can be turned off. The polymer abstraction mechanism and the `Comparator` check this binary flag. Strictly it is an attribute of the property's datatype, not of the property, but the development of POISE did not address user defined data types for engineering values.

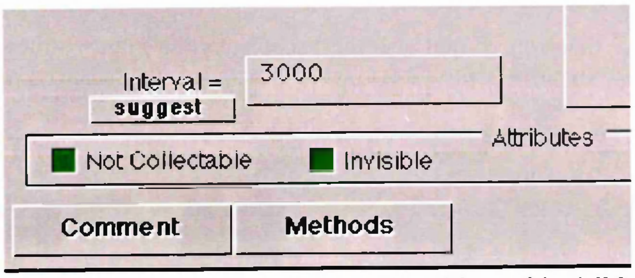
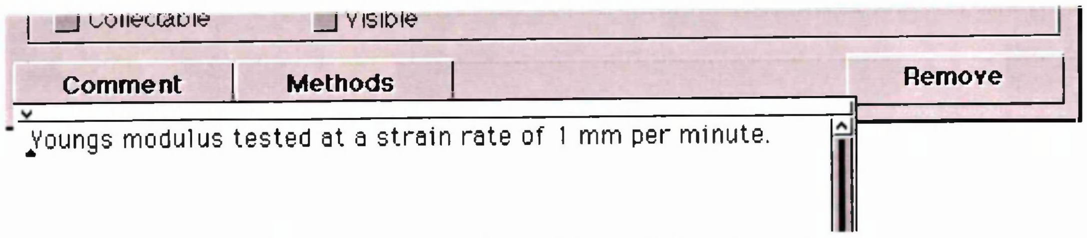


Figure 42: Interval, not-collectable, and invisible

The fields in the `PropertyEditor` are immediately active. Changing the interval will cause all histograms on the property to update. Any `Comparators` displaying the property will also update. The immediate feedback can make the selection of an interval much easier, and allows the user to modify the emphasis of a property histogram (§3.7.1). When working on a particular design, the emphasis of specific properties is different. Changing the interval can reflect the different emphasis.

The "Invisible" attribute prevents various POISE interfaces displaying the property as an option. Many properties that grades describe are of no interest to a designer with a particular design problem. Removing these properties from view lets the designer focus on the properties of interest.

The string describing the property is very brief, ensuring easy display through the interfaces. A complete description of the meaning of a property can take a large section of text. The "Comment" button provides just such a space. Although the example in Figure 43 is only displaying a single line, the child-window is capable of unlimited text.



Youngs modulus tested at a strain rate of 1 mm per minute.

Figure 43: Comment, method and remove

Properties describe classes. The list of classes this property describes is given in a list sub-view. In Figure 40 the property describes the class `Polymer`. The list does not include classes inheriting the property from `Polymer`. Alternatively, the property could list an orthogonal class. The "Add Domain Class" adds the property to a class. The button lists the orthogonal classes and a selection to open the `HierarchyEditor`. By definition, a property can not belong to both an orthogonal class and a taxonomic class. The view will not add a property to an inconsistent class until the other orthogonal class removes the property.

A class removes a property by selecting the class in the list. A removal option is found in the menu of the list. A property can be removed from the whole domain by selecting the "Remove" button (Figure 44). Not only do all classes remove the property but also the `Property` class removes the property from a list of all assigned domain properties.

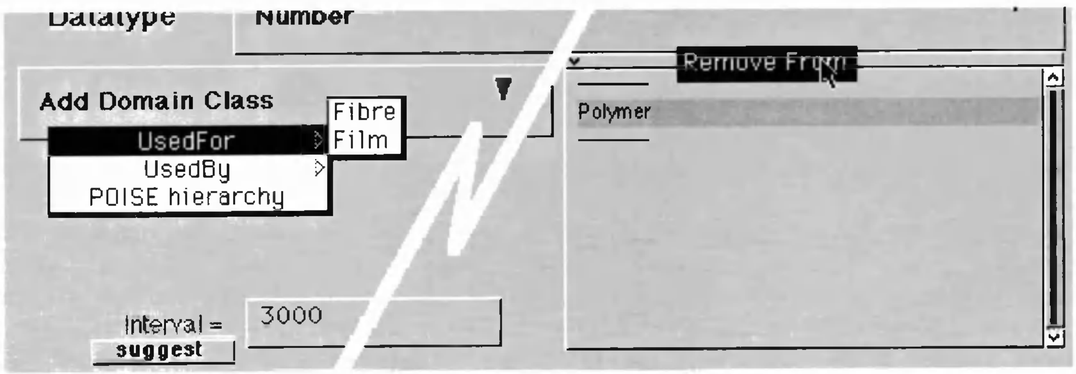
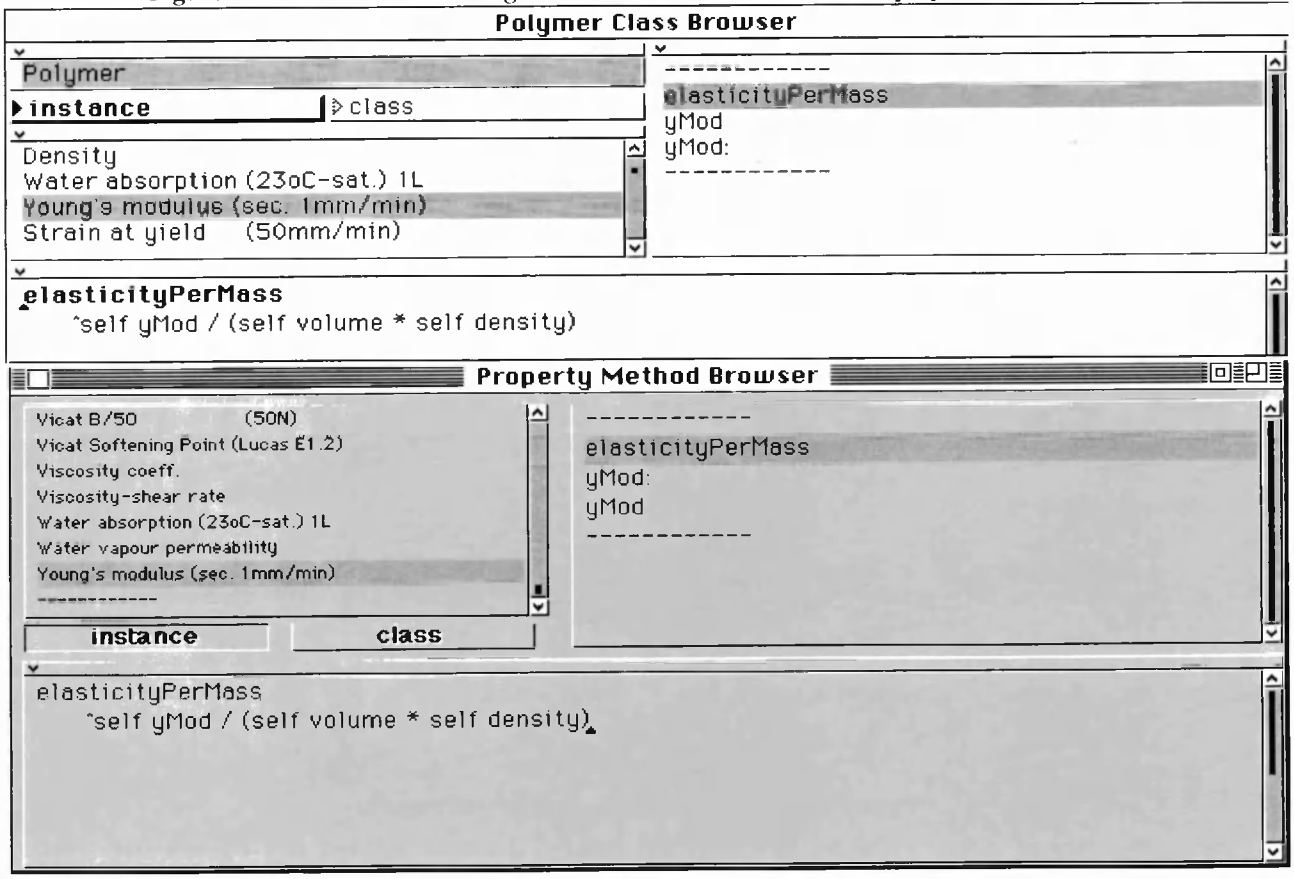


Figure 44: **Add to orthogonal classification / remove from polymer classification**



```
Density
Water absorption (23oC-sat.) 1L
Young's modulus (sec. 1mm/min)
Strain at yield    (50mm/min)
```

```
------------
elasticityPerMass
yMod
yMod:
------------
```

```
elasticityPerMass
    ^self yMod / (self volume * self density)
```

```
Vicat B/50              (50N)
Vicat Softening Point (Lucas E1.2)
Viscosity coeff.
Viscosity-shear rate
Water absorption (23oC-sat.) 1L
Water vapour permeability
Young's modulus (sec. 1mm/min)
------------
```

```
------------
elasticityPerMass
yMod:
yMod
------------
```

```
elasticityPerMass
    ^self yMod / (self volume * self density)
```

Figure 45: **Property method browser**

The `Property` is a special PTO (§4.4.3). Each `Property` has a single instance variable. The default behaviour is two methods. An `accessor:` method retrieves the contents of the instance variable, and an `updator` to set the contents. Both use the symbol name of the `Property`, the `updator` adding a colon as is the convention for methods with one argument. The `Property` methods for Young's modulus shown in Figure 45 (bottom window) are the `accessor:` (`yMod`), `updator` (`yMod:`) and a user defined behaviour `elasticityPerMass`, a unit of energy absorption.

Young's modulus is a property of the `Polymer` class. When a property compiles a behaviour, the behaviour immediately installs on the `Polymer` class. Figure 45 shows the method in `Polymer` through a standard Smalltalk browser (top window). The browser groups protocols, naming each group and listing the names in the top-left list. All the methods from the Young's modulus property are together under the name of the property.

5.8 Transcript

The transcript part of the `POISEsession` window provides the designer with a computational interface.

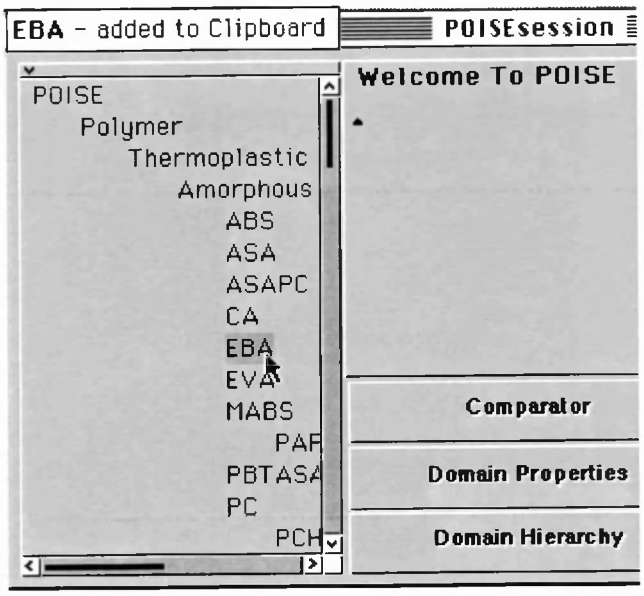


Figure 46: Transcript- selecting abstract polymer for clipboard

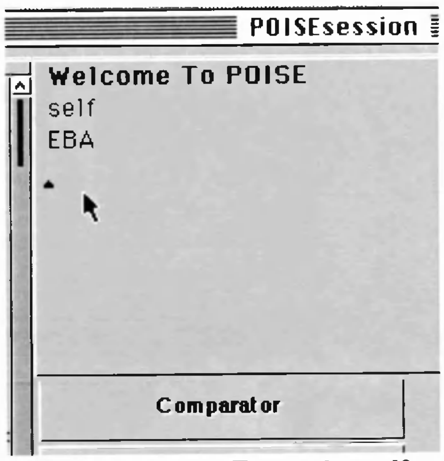
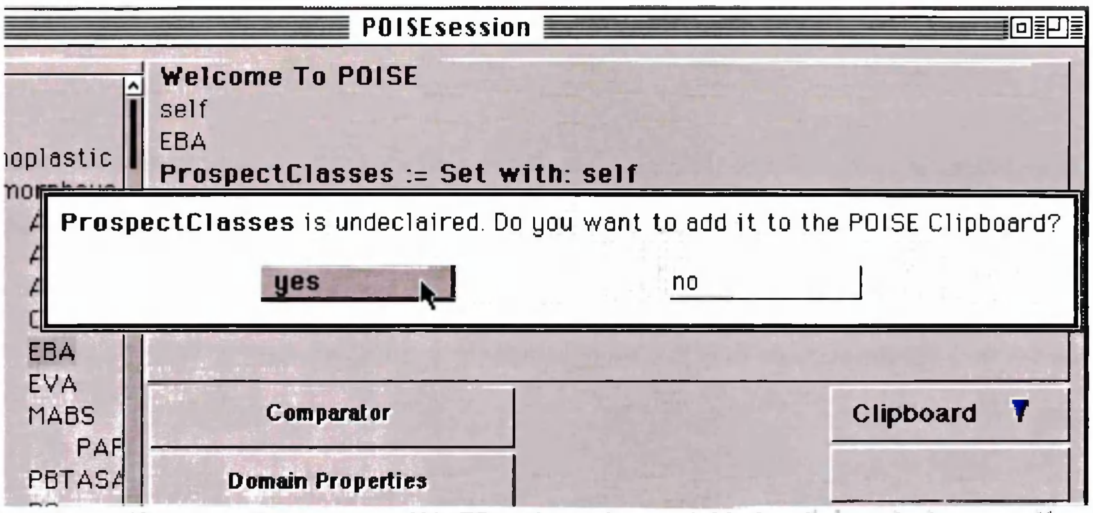


Figure 47: Transcript- self binds to clipboard contents

Access to objects for computation is through a clipboard. In Figure 46, the abstract polymer class EBA is placed on the clipboard by selecting the name in the hierarchy. The `Comparator` can also place populations of polymers on the clipboard. The clipboard notifies the user whenever an object is put there by opening a small `Notifier` window with the print-string of the object, and the words 'added to Clipboard'.

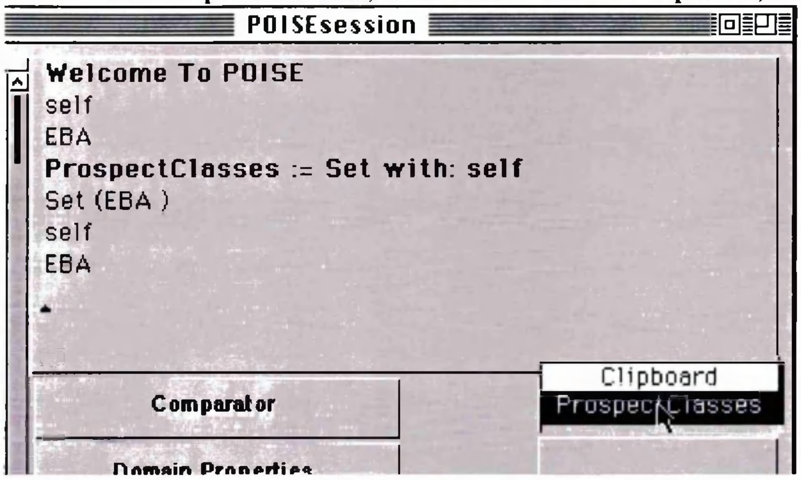
The variable '*`self`*' automatically binds to the object on the clipboard, when code evaluates in the `Transcript`. In Figure 47 coding '*`self`*' returns the object EBA. The `Transcript` prints the print-string of the returning object.



```
Welcome To POISE
self
EBA
ProspectClasses := Set with: self
```

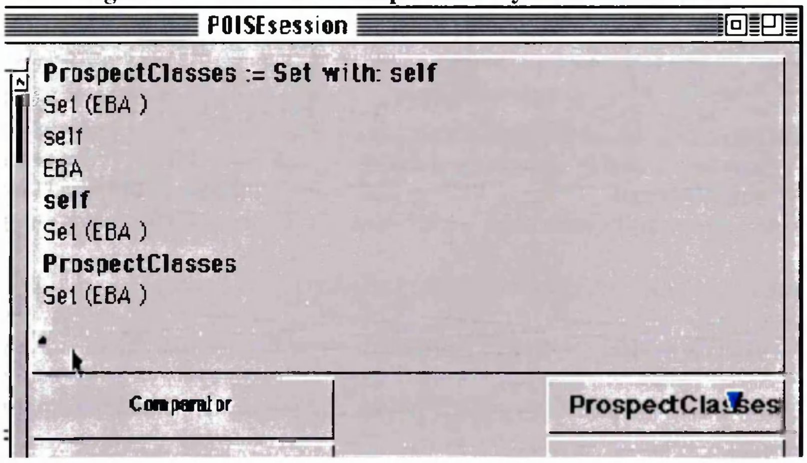
ProspectClasses is undeclaired. Do you want to add it to the POISE Clipboard?

Figure 48: Transcript- self is EBA, then select variable for clipboard, changes self



```
Welcome To POISE
self
EBA
ProspectClasses := Set with: self
Set (EBA )
self
EBA
```

Figure 49: Transcript- define your own variables



```
ProspectClasses := Set with: self
Set (EBA )
self
EBA
self
Set (EBA )
ProspectClasses
Set (EBA )
```

Figure 50: Transcript- self and ProspectClasses bound to Set with EBA

Any variable the `Transcript` does not recognise raises a `Notifier` (Figure 48) asking if the variable is to be 'added to the POISE Clipboard'. The clipboard can record a number of different objects under different names, and one 'active' object under the name "Clipboard". Together with global variables, the clipboard forms the variable scope of the code compiled in the `Transcript`. In Figure 48, a variable `ProspectClasses` assigns to a set with *`self`* (the EBA). Not recognising the variable `ProspectClasses`, the user is given the opportunity to add it to the clipboard.

After assigning a variable, the clipboard's active variable is left unchanged (eg still EBA). The active variable changes from the variable 'Clipboard' to another variable (eg `ProspectClasses`) by selecting the button currently marked 'Clipboard' (Figure 49). The variable *`self`* binds to the value in `ProspectClasses` and the button displays the new active variable's name, in Figure 50.

5.9 Summary

The walkthrough illustrates the functionality built into POISE as presented to the domain expert. The domain expert can define new schema components: grades, properties and classes. The user can re-distribute and re-define any of these components. While POISE changes, the effects of these changes immediately affect the inference mechanisms including the Smalltalk standard inheritance of grade properties and the POISE specific abstraction of abstract polymer behaviour.

The `Comparator`, also illustrated, is a window for browsing abstract polymer behaviour. The following chapter discusses Spedding's use of this window to contrast the polymer families while investigating appropriate classification.

The `POISEsession` lets the designer evolve a complex query in a `Transcript`, and a simple example is given.

POISE records the state of any activity in the `Workbase` when the designer leaves the session and re-instates the session when the designer returns. The designer can continue developing the complex design queries, searches, shortlists, and views on polymers on return to the session.

Chapter 6 Using POISE to Analyse the Polymer Domain

Spedding[8] uses the POISE application during her analysis of the polymer domain. Although her objective of determining an appropriate classification differs from the objectives of a designer, they both require a similar analysis of the domain, which identifies similarities and differences between polymer grades. A number of the relationships and characteristics of the polymer domain she reports from her analysis illustrate different ways of using the POISE tools not initially conceived when they were designed. These ways of using the tools are likely to benefit the designer since the analysis is similar.

The `Comparitor` was initially intended to determine relationships between abstract polymers. For example, Spedding illustrates the tensile strength of `Thermosets` is generally less than thermoplastics. Unfortunately, this particular application of the `Comparitor` was not as effective as expected at extending the classification beyond the polymer families. Often the nature of the data obtained from CAMPUS restricted further comparisons. Analysis between amorphous and partially `Crystaline` showed fewer differences than expected whereas the `Comparitor` did distinguishes classes at the chemical level. The standard polymer tests may not measure the effect of crystalinity, possibly to prevent capriciousness. Despite the inability to further classify, Spedding found other uses for the comparisons.

Although CAMPUS populates POISE with over 1000 polymers, each describing 50 properties, there are a large number of `Polymer` classes and the grades are not evenly distributed amongst them. Additionally, many properties are universally unpopular, ie often sparse of data, with only 12 giving adequate populations. Further, the suppliers of polymers produce grades for specific markets. Suppliers generate more grades for profitable markets, therefore the number of grades with a certain property profile is not a measure of the polymer's typical properties but a measure of the market that uses the polymer. Therefore, the abstract behaviour of `Polymers`, for example, is highly distorted by the behaviour of `Polyamides`, which are highly populated. The `Comparitor` does not hide this bias, but the median or average value from an abstract property will hide the distortion. A comparison of the populations of `Polyamide` and all polymers for any property will show `Polyamide` as a strong contributor.

Although popular markets distort the total number of grades with a particular property profiles, the range of property profiles of grades is reasonably represented. It is possible a market driven source of data, such as polymers for the automotive industry, will only represent particular property performance profiles. A range of different markets for polymers ensures the data source represents a range of property profiles.

Spedding illustrates an example of a significantly different application of polymers while using the orthogonal classification in POISE. Spedding used the search utility to find the words "film" and "fibre" in the text description of CAMPUS grades. Besides the complication of locating "Fibre-reinforcement", the search located a significant number of these grades. Spedding declared an orthogonal class for films, initially with no properties. She then added each of the grades located with the "film" text to the orthogonal class `Film`.

With the grades classified under `Film` the `Comparitor` can display just those grades. From the `Film` class, Spedding found they generally had relatively poor mechanical properties. A class with a range of applications will include grades with an extreme in a property's performance, and grades where the same property performance is not significant, which will distribute the property. The lack of data prevented any further generalisation, but again the `Comparitor` and the classification demonstrated their roles in determining this case.

Additives are another distorting effect on the abstractions. Some properties are more significantly affected by their additives than the polymer chemistry. Further orthogonal classification could remove this factor, but often the exact composition of additives is a polymer supplier's secret. Spedding highlighted the inclusive nature of the polymer classification prevents the `Comparitor` from excluding an orthogonal class of polymers, which is necessary to remove a distorting factor. For example, the `Comparitor` does not support browsing for "strongest polymer not glass filled".

While browsing a property with the `Comparitor`, Spedding found the text comments useful for relating the extreme grades to other properties. This is how she determined the effect of additives. The text comments can also include the application of the grade, and therefore a type of property profile, or even specific property profiles. For example, while browsing the density of `Polyvynalchlorides`, the text of grades with high density had lead stabilisation. In the class of `Polystyrene` high density grades were noted for stability and rigidity implying a high Young's modulus, which was confirmed in another comparison. By relating the different interface tools, Spedding inferred different types of property correlations, such as a specific correlation between density and Young's modulus for PVCs.

In another case, the `Comparitor` clearly identified miss-placed grades and classes. While investigating the extreme impact strengths of `Polyethylene`, Spedding found the two grades of the PESU subclass had a higher impact strength. The text confirmed they were not a polyethylene but a `Polyethersulphone`, which gives a higher impact strength.

Chapter 7 Conclusions

The representation presented focuses on the classification and abstraction of polymer grades. Classification and abstraction precede abduction, an inference method commonly used during design, which infers facts about members of a class from information abstracted from the class. Conclusions about large volumes of information are inferred from a few abstract facts. Therefore, both classification and generalisation are intrinsic to the representation of polymer materials for design. Although information representation commonly includes a classification, the schema in POISE builds a classification and abstracts general properties from the classification into many levels of representation.

An object-oriented software model was adopted to implement POISE. The object is a highly abstract computing element that provides a number of benefits to knowledge representation. Behaviour sharing between objects encourages abstraction and classification of knowledge and object encapsulation simplifies the evolution of a knowledge representation. Class-instance languages specialise on the classification of objects, and Smalltalk is an example. Class-instance languages implement a strict classification for explicitly describing software, whereas real classification is stereotypical, and maintains a level of generality. This difference raised the question whether the class-instance classification can represent real classification, or is it only a software design mechanism?

The majority of polymer grade information depends on the grade's chemistry, but there is also information relating to additives, processing and applications of the grade. Since an instance inherits from one class, which dominates the instance's behaviour, an instance can only represent data from one class of information. Therefore, the class-instance language can represent separate orthogonal parts of a grade, but not a complete representation of a grade because the complete grade does not belong to a single classification. The obvious solution is to unite the orthogonal parts into a single object. The behaviour of this object depends on the components and not a class, therefore does not fit the class-instance model.

A single materials classification does not define all the information on grades. Therefore, the class-instance relationship is too restrictive for an instance to represent a grade. Instead a number of instances from different classifications could contribute to a grade representation, but this depends on the languages ability to traverse these object boundaries through behaviour sharing. POISE enhances the behaviour sharing in Smalltalk with an object capable of unifying the behaviour from multiple objects, thereby extending the languages representational ability. This unique object inherits the ability to share other object's behaviours from the class called `Enhancer`.

Improving the representational abilities of a class-instance language is not the only benefit of the `Enhancer`. A specialisation of the `Enhancer` extends the empathy between the objects it unifies. The `Enhancer` provides behaviour sharing that differs from the explicit dynamic messages between individual objects and differs from the static implicit behaviour shared between classes and groups of instances. The `Enhancer` provides implicit dynamic empathy between individual objects. These are general descriptions of types of behaviour sharing from the Treaty of Orlando[5]. The `Enhancer` is therefore a general enhancement to the class–instance paradigm.

Like the `Enhancer`, delegation is an example of implicit dynamic empathy between objects. Work by Zucker identifies delegation as an important representational tool for prototyping design. Therefore, opportunity exists to represent design prototyping in a class-instance language using the `Enhancer`. Other research combining delegation with the class-instance relationship calls this language hybridisation, since they are normally contrary approaches contending for the right to describe an object. The `Enhancer` is an enhancement since it leaves the existing Smalltalk class-instance objects unaffected by the introduction of implicit dynamic empathy. Objects must explicitly permit implicit-binding, by using the `client` message rather than `self`.

The `Enhancer` is not equivalent to delegation. The name `self` always refers to the proxy object that owns a behaviour. In delegation, `self` refers to the delegating object. Using the `Enhancer`, the proxy's behaviour must look for the delegating object with the message `self client` before there is any empathy. Although an alternative approach was investigated, where a Smalltalk class models a prototype by becoming an instance of itself, the use of the `client` message was not considered a problem for explorative design. The `Enhancer`'s ability to re-program the way it handles messages was a dominant advantage over the alternative.

An initial experiment suggests the `Enhancer` can support Zucker's objectives of modelling explorative design in a class–instance languages. In this experiment, the class-instance structure represented concrete knowledge on materials, processes and geometry while an `Enhancer` represents the design, which dynamically explores ways of combining the knowledge. The `POISEsession` lets the designer evolve a complex query in a `Transcript`. An example query, reported elsewhere[80], tested the design property of `cost`, a function of all the perspectives. The test design was specialised by refining the materials perspective, thus demonstrating the dynamic binding between design and the perspective. Experiments other than cost were hindered by the lack of abstract knowledge in the public domain that combines information from different perspectives.

Another specialisation of the `Enhancer` resolves deductive inheritance (§2.2.4). The `PolymerDataAbstraction` inherits any named property from the instances of a class, including any orthogonal properties the instances may have. The abstraction behaviour then deduces a single abstract representation of the property from all the values it inherits. The `CombinedDataAbstraction` is a similar object, but infers an abstract representation from other `PolymerDataAbstraction`. In the hierarchy, the `CombinedDataAbstraction` infers the abstract properties of a superclass from its subclass's `PolymerDataAbstractions`. The `Polymer` classes themselves manipulate their messages so they inherit their abstract behaviour from either a `CombinedDataAbstraction` or a `PolymerDataAbstraction`. The result is a hierarchy of abstract polymers that generalise properties typical of the grades they classify, from which the designer can interpret the design benefits of the class.

The dynamic empathy of the enhancer was a distinct advantage when evolving the schema. The Smalltalk environment may evolve class inheritance hierarchies, which has a consequence on abstraction. When `Polymer` classes change their inheritance patterns, they also change the pattern of abstraction. `CombinedDataAbstraction` dynamically compose their abstractions from the subclasses, which ensures the abstractions are always consistent with the classification hierarchy.

Schema evolution in Smalltalk is a complex manipulation that substitutes all affected classes and instances with a new modified copy. The development tools for evolving Smalltalk classes were found highly inefficient, often replacing the same hierarchies for each change. While specialising these tools for evolving the polymer classification, a new type of protocol objects was defined that is independent of a physical model. These protocols are independent of the class, which defines the physical model of instances protocols therefore they do not require re-compiling when the class schema changes.

The improvement on schema evolution was a bonus feature of the independent protocols. These protocols have a representational role abstractly describing polymer properties. A partial template object, PTO, collects any set of interacting independent protocols. A PTO collects a re-useable set of protocols that may be installed consistently on many classes. The PTO in POISE represents polymer properties independent of `Polymer` classes and encapsulates the computing concepts of protocols with a concept familiar to the domain expert.

The PTOs translate the concepts of the `Polymer` class and the polymer property in the `Polymer` domain to the class and protocols in the software domain. Moving properties between classes is a taxonomic function, which now has an equivalent process in the software domain. Presenting a hierarchy populated with domain concepts, and without software concepts, lets the domain expert evolve the classification. Presented appropriately, the domain expert can create and position specific properties in classes and specific classes into a hierarchy, and POISE translates these actions into a manipulation of the software schema.

POISE supports the classification process with a visual analysis tool called the `Comparator`. The `Comparator` is a window for browsing abstract polymer behaviour, which displays the data abstracted by the classification hierarchy as histograms. Spedding's research[8] utilises the `Comparator` extensively to analyse the polymer domain for similarities and differences between classes of polymer grades. The `Comparator` showed that additives had a distorting effect on the abstractions supporting the need for orthogonal classes. Unfortunately the data on polymers did not consistently indicate the nature of additives, and a high majority had some kind of additive, so classifying to remove the effect of additives from 'natural' polymers was not possible. Since the additive classes were not created, the `Comparator` was never programmed to exclude orthogonal classes. For example, the `Comparator` does not support browsing for "strongest polymer not glass filled". There is no technical reason preventing orthogonal class exclusion.

The concurrent research into appropriate classifications tested and advanced POISE as a complete system. Orthogonal classes, new properties and new grades were added for `Films` and `Fibres` and 'Used by `Lucas`'. The abstraction mechanism automatically updated to include the descriptive properties contributed by the new orthogonal classes and properties, so for example, the `Comparator` could display the `Polymer` class against film `tear-strengths` from grades enhanced with the property, although the polymer hierarchy does not define the property.

POISE imported the bulk of the data from CAMPUS. The nature of data from CAMPUS identified some problems for distinguishing classes at different levels of generalisation. Though a property distinguished polymers at the chemical level, they did not distinguish the polymers significantly at higher levels of generality, which characterised the material structure. No other polymer representation represents materials at different levels of generalisation. Consequently, the properties currently describing polymers tend to generalise over all polymers, and often over all materials. These general properties are unable to distinguish the specific structural differences in polymer materials, hence classification by structure are not distinguished by these general properties.

Along with a population of over 1000 grades imported from CAMPUS, new `Polymer` classes and properties were generated, which tested the database management. POISE's database facility stored all the new objects, and the storage remained transparent to all POISE activities, thereby hiding memory management issues from the knowledge representation and the domain user. A general proxy `Enhancer` provides a transparent interface between objects of an application and objects held in the DBMS. Messages sent to database objects via the proxy `Enhancer` activate the enhanced behaviours for requesting the DBMS to bring the object into primary memory and for updating the database with any changes. Classes do not define this behaviour, thereby making database storage available to objects of all classes. Objects using database objects as part of their own behaviour do not need to define transactions. They can treat the database object like any other object of the same type.

A database proxy substitutes for any relationship between the application and the database. The database is object-oriented, only ever reading one object and substituting all its relations with proxies. A proxy only retrieves an objects if a process sends a message to it.

POISE specification identifies a dichotomy in database management requirements. Database management for persistence of user data differs from the interchange of data between users. The main difference is persistence is single-user data, and data-interchange is multi-user data. Managing multiple users requires the definition of a transaction, and a transaction distinguishes a database process from other computing processes, which complicates transparency. This type of database management is common to commercial OODBMS, with a focus on transaction management and its integrity. Schema evolution is a type of transaction that is typically very large and causes problems for these transaction bases systems. The behavioural complexity of objects within POISE and their tendency to evolve puts the representation beyond even the most advanced commercial OODBMS. For the private data of the single-user in POISE the objectives are more limited, and more powerfully focused on representation, than the objectives of a general-purpose management system. Consequently, POISE has a single-user database for object persistence called a `WorkBase`, which adopts Smalltalk's manipulation capabilities, including schema evolution.

The unique feature of the `WorkBase` is that when it reads objects it resolves differences in the schema between client and server, which allows the client schema and server schema to independently update individual objects. Implementing this feature was simplified by the single connection policy between the POISE application and the private single-user `WorkBase`. Most DBMS focus on supporting multiple connections and consequently complicate the client's dependence on the server's schema, which the server endeavours to maintain consistent for multiple clients.

An advanced object storage system is a better description of the `WorkBase` than a DBMS because the application executes all object behaviours, not the `WorkBase`. The `WorkBase` advances object storage because it is capable of representing complex objects without prior declaration of file structure, including the classes of polymers in the hierarchy and the behaviours of engineering properties developed by the user. In addition, with the help from the database proxy, the objects maintain their unique identity, usually lost when object storage systems remove objects from the application environment.

The proxy manages the active lifetime of the object. The WorkBase in collaboration with the Smalltalk memory management, commits the oldest proxies when memory is low thereby maximising the utilisation of primary memory. This simple memory management policy is a consequence of the single-user transaction restriction of the WorkBase.

The WorkBase also commits all proxies when the user terminates the application. POISE records the state of any activity when the designer leaves the session and re-instates the session when the designer returns. The designer can continue developing the complex design queries, searches, shortlists, and views on polymers on return to the session.

The one resource the WorkBase does not manage effectively is the disk file it uses to store the state of objects. Another weakness in the design of the WorkBase was the efficiency of the DBMapping. This object provides the primary index for the database. Improving the DBMapping and managing the disk file were both unnecessary for the experimental purpose of POISE.

The focus of this thesis has been the software development of POISE. The software demonstrates the feasibility of the thesis but its overall success of supporting design, which is the reason behind its development, depends largely on the information it contains. The software principles are well established with the CAMPUS data, but its general nature will not thoroughly test the design principles. The information POISE contains must start to answer design questions. This may require specific properties that better distinguish classes for specific applications or properties that negotiate with other perspectives to compromise the design. At least the software now exists that equips the domain expert with tools to perform these experiments.

Chapter 8 FutureWork

This thesis has followed the argument that designers require advanced software descriptions of classification and abstraction to support their decision tasks. Along the way the research raised many questions within the computer and material sciences. One question is the suitability of material properties to describe abstract materials. The lack of distinction between amorphous and crystalline polymers was cited as a visually indistinguishable example. The need for further research into the relationship between classification and properties describing its members is not conclusive without detailed research into how designers use information. One reason for this lack of research is the absence of an historical link between property use and design outcome to measure the effectiveness of properties in design. Software like POISE that is capable of manipulating complex materials information could help to determine the effectiveness of specific design methods using properties. The software can record the historical application of methods towards a design. It is possible that such research will find general properties, like those in CAMPUS, do not answer design questions effectively and lead to more appropriate materials research.

8.1 Extentions For Further Design Support.

Design requires contributions of information from many perspectives. The design behaviour is a complex combination of behaviour from different perspectives. POISE only addresses the representation of materials information principled upon the material's chemistry (and composition in the case of filled polymers) and it limits its design support to providing an example of tools for classifying and visualising abstract materials. Each new design perspective introduces its own challenges. Further research is required to represent the other perspectives, which contribute to a design, and develop useful design methods that integrate their various sources of information. Only then will computers aid the process of design and properly record the design history, which could measure design effectiveness for studying case based reasoning in design.

8.2 Furthering the Role of Object Orientation in Knowledge Representation

POISE is an example of a knowledge representation tool that does not build on traditional expert systems theory. POISE represents more specific manipulation rules than rule based expert systems where the general manipulation logic is encoded in an inference engine. POISE is not restricted in structuring information for the engine. The manipulation rules in POISE are object orientated. Classes contain the rules developed for each type of object, which contain the information so inference is a specific relationship between an object and its class evoked through well-defined patterns of message passing. POISE demonstrates the diverse patterns generated by object oriented systems forms a highly expressive knowledge representation. From this direction, object orientation needs more study into the role diverse message handling, which drives the patterns of behaviour, could have in knowledge representation.

POISE has also introduced some unique patterns of object behaviour. Patterns are quickly becoming important descriptions of abstract object oriented software solutions. Patterns often have some interesting attributes that make them particularly suitable for solving certain kinds of software problems. The patterns in POISE are interesting because they dynamically compose object interfaces. Most objects have a static type, whereas the Enhancer permits dynamic type construction, as demonstrated by the orthogonal description of grades. The PTO permits the runtime re-engineering of object types. POISE is able to create these patterns in Smalltalk, which is a non-typed language. Typed languages require specific type definitions to validate program execution. Currently valid execution of these patterns is not guaranteed and valid execution requires careful implementation and application. Further research would be necessary to determine how a typed language might support dynamic typing generated by a similar pattern.

This thesis has a highly focused agenda for representing materials information in a class instance language. This focus tackles some of the more complex issues in representing the domain. There are many other issues that fit the object-oriented paradigm very well that at first appear to be less of a challenge. Object orientation has much more to offer CAD development. These include managing engineering measurements that include units and accuracy

Engineering design also involves the application of design calculations. Objects can represent not just a result, but the whole calculation as a method that combines other calculations with new input parameters. The results of calculations are no longer subject to external interpretation. Ironically, the result does not need to derive a specific value until the designer needs it. Since most design decisions are a trade off between parameters, the specific values are not important until the designer contrasts specific combinations of design attributes. The object calculates the result dynamically when needed and not just when the input parameters are available since these parameters may change.

The benefits of a completely object oriented representation of a design method is that the design process is recordable and reusable. If the designer decides that the input parameters or method must change, a traceable route of dependent design discussions can be determined, which in turn can be re-evaluated.

From this scenario, we can see that an evolving database of materials might in fact notify the designer when potentially better materials have entered the database for specific designs, simply because the system records the methods used to in design conclusions.

8.3 Extensions Within the Materials Domain

POISE assumes the domain of engineering polymers can be classified into a hierarchy. This assumption ignores some complex material concepts, which challenge a hierarchical classification, as highlighted in Spedding's analysis of the domain. Co-polymerisation and alloying of polymers greatly complicate the relationship between a material's parentage and the material's physical properties. Whether the parentage should influence the classification of these kinds of material is still subject to further work that may suggest alternative software mechanisms for representing these concepts.

The POISE design could extend the domain to include metals. The characteristics of metals properties are probably better understood than polymers and the domain is also highly characterised by alloys. Both these characteristics of the domain would lend to experimenting with the classification of alloys and the development of more complex behavioural descriptions of metals. For example work hardening and annealing are well studied in metals, and are possibly suitable for computer modelling. The benefit of integrating the modelling with a database is that it is much easier to relate the test data to the computer models and therefore derive where the model deviates from reality.

Metals also have a simpler chemical description than polymers, based on crystalline atomic arrangements rather than molecules. This means abstraction techniques might be more precise. Examples could include abstracting the phase diagrams, commonly used by metallurgists, from experimental data as an alternative to histograms.

Ceramics are probably the least well understood of materials. Their properties are dominated by the kinds of lattice structures the chemistry produces, which is subject to the process used to create the material as much as the chemical composition. The study of their behaviour will be a subject of further work for quite a while. The difficulty is deriving a principled classification, but once determined, there is nothing to suggest the software patterns presented in this thesis will not be able to represent ceramics as well.

---

[1] **J. Zucker (1989)**: Engineering design computed by prototypes and descriptions; Library, Open University, Milton Keynes, UK.

[2] **A. Goldberg and D. Robson (1983)**: Smalltalk-80™ : The Language and Its Implementation; Addison-Wesley Reading, Massachusetts.

[3] **D. H. H. Ingalls, A. H. Borning (1982)**: Multiple inheritance in Smaltalk-80; Proceedings of National Conference on Artificial Intelligence, Pittsburgh, PA, pp234-237.

4 **J. P. Briot (1989)**: Actalk: a Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Enironment; in ECOOP'89, 2.3-2.4.

5 **L. A. Stein, H. Lieberman and D. Ungar (1988)**: The Treaty of Orlando: A shared view of sharing; in ibidi 100, pp31-48.

6 **J. Schmitz, E. Bornschlegel, G. Dupp and G. Erhard (1988)**: CAMPUS plastics database; in Plastverarbeiter 39(4), pp50-58.

7 **K. Oberbach (1989)**: Plastic Properties for Design - a Database from the Raw Materials Suppliers; in Polymer Properties For CAD/CAM, London, PRI, 3/1-5.

8 **V. Spedding (1995)**: An Object-Oriented System for Engineering Polymer Information; Library, Open University, Milton Keynes, UK.

9 **M. J. French (1971)**: Engineering Design: The conceptual stage; HEB London, ISBN 0 435 71650 6, pp 8.

10 **A. Demaid, S. Ogden, J. Zucker (1992)**: Materials Selection: Object-Oriented Structures for Factoring Polymer Information; in Computerisation and Networking of Materials Databases: Third Volume, ASTM STP 1140, Eds. Thomas I Barry and Keith W Reynard, Philadelphia.

11 **EPOS™** : a product of ICI Ltd;.

12 **RAPRA**: Plascams-220™; Rubber and Plastics Research Association, Plascams Technology Ltd, Shawbury, Shrewsbury, Shropshire, UK.

13 **A. Hopgood (1989)**: An inference method for selection, and its application to polymers; in Artificial Intelligence in Engineering 4(4), pp 197-204.

14 **D. Bassetti (1995)**: Fuzzymat User Guide; Laboratoire de Thermodynamique et Physico-Chimie Métallurgiques, ENSEEG, ref 16.

15 **P. Pechambert, Y. Brechet (1995)**: "Etude d'uneMethodologie de Choix des Materiaux Composites" and "Conception d'un Logical d'Aide à la Formulation des Verres"; Laboratoire de Thermodynamique et Physico-Chimie Métallurgiques, ENSEEG, ref 16.

16 **M. F. Ashby (1997)**: Materials Selection: Multiple Constraints and Compound Objectives; in American Society for Testing and Materials, STP 1140.

17 **G. E. Dieter (1983)**: Engineering Design: A materials and processing Approach; Mc Graw Hill, ISBN 0-07-016896-2.

18 **G. Lewis (1990)**: Selection of Engineering materials; Prentice Hall, ISBN 0-13-802190-2.

19 **P. Sargent (1991)**: Materials information for CAD/CAM; Butterworth-Heinemann Oxford, Chapter 5.

20 **Open University (1985)**: Manufacture, Materials and Design; Open University Press.

21 **R. Ayres (1974)**: Materials Process Product Model; International Research and Technology Corporation Ref Endean, 1989#22.

22 **S. Pugh (1986)**: Curriculum Design: Specification phase; Open University Press.

23 **J. Zucker, A. Demaid (1989)**: A software machine designed for selection; in Knowledge Based Systems 2(3), pp178-184.

24 **H. A. Simon (1981)**: The Science of the Artificial; MIT Press Massachusetts.

25 **A. Demaid, J. Zucker (1988)**: A conceptual model for materials selection; in Metals and Materials 4(5), pp191-271.

26 **C. S. Peirce (1958)**: Collected Papers of Charles Sanders Peirce; Harvard University Press, 1958.

27 **S. E. Fahlman (1979)**: NETL: A system for representing and Using Real-World Knowledge; in Proceedings of the National Conference on Artificial Intelligence, pp4-9

28 **R. Ackerman (1965)**:Theory of Knowledge: A critical introduction; McGraw-Hill, pp63-68 & 76-79, (ref 1).

29 **E. E. Smith, D. L. Medin (1981)**: Categories and Concepts; Harvard University Press, (ref 1).

30 **I. Stewart, D. Tall (1977)**: The foundations of mathematics; Oxford University Press.

31 **D.S. Touretzky (1984)**:The Mathematics of Inheritance Systems; PhD, Computer Science, Carnegie-Mellon, Pittsburgh, PA 15213, also Pitman/Morgan Kaufmann, Research Notes in Artificial Intelligence series (1986).

32 **M. M. Downs, R G. Greene, D. Rishel (1991)**: Development of an On-Line Data Dictionary Using Conceptual Data Modelling; in NSF Workshop Internal Report, Alcoa Technical Centre, U.S., November pp11-14.

33 **M. F. Ashby (1989)**: Material Selection in Engineering Design; in Material Science and Technology 5(June), pp517-525.

34 **M/Vision (1995)**: A product of PDA Engineering; Magnetic House, Waterfront 2000, Salford Quays, Manchester M5 2XW.

35 **J.E. Lee, D.E. Marinaro, M. E. Funkhouser, R.M. Horn, R. P. Jewett (1992)**: Creating a Common Materials Database; in Advanced Materials & Processes (November),.

36 **A. Demaid, J. Zucker, S. Ogden (1992)**: Object-Oriented materials Engineering Information Modelling and Management; in TOOLS, pp119-134.

37 **R. Frost (1986)**: Introduction to Knowledge Base Systems; Collins London.

38 **B. Raphael (1968)**: A computer program for semantic information retrieval; in ibidi 102,.

39 **R. Quillian (1968)**: Semantic memory; in ibidi 102,.

40 **M. Minsky (1975)**: A framework for representing knowledge; in The Psychology of Computer Vision, Eds. P. H. Winston, McGraw-Hill.

41 **S. E. Fahlman (1989)**: NETL: A System for Representing and Using Real-World Knowledge; MIT Press, Cambridge, MA.

42 **R.J. Brachman (1983)**: "What IS-A is and isn't": An analysis of taxonomic links in a semantic networks; in IEEE Computer 16(10), pp30-36.

43 **E. Decio, P. Petrin, L. Spampinato (1990)**: Pushing the Terminological Barrier; in ibidi 101,.

44 **M. S. Fox (1979)**: On Inheritance in Knowledge Representation; in International Joint Conference on Artificial Intelligence, pp282-284.

45 **D. McDermott, J. Doyle (1980)**: Non-Monotonic Logic I; in Artificial Intelligence 13(1,2), pp41-72.

46 **R. J. Brachman (1985)**: "I lied about the trees": Or defaults and definitions in knowledge representation; in AI Magazine 6(3), pp80-93.

47 **J. F. Horty (1990)**: A Credulous Theory of Mixed Inheritance; in ibidi 101, pp13-28.

48 **P. F. Patel-Schneider (1990)**: What's Inheritance got to do with knowledge representation; in ibidi 101, pp1-10.

49 **A. Demaid and J. Zucker (1989)**: Selection of engineering materials; in Scandinavian Symposium on Materials Science , Copenhagen, Danish Society for Materials Testing and Research.

50 **D. Hartzband (1985)**: Enhancing knowledge representation in engineering databases; in IEEE Computer, pp39-48.

51 **D. S. Tsichritzis, F.H. Lochovsky (1982)**: Data Models; Prentice-Hall, Englewood Cliffs, New Jersey.

52 **DBTG (1971)**: The Database Task Group of the CODASYL Programming Language Committee Report; Available from ACM, BCS and IAG, (in ibidi 37).

53 **E. F. Codd (1970)**: A relational model of data for large shared data banks; in CACM 13(6), pp377-387.

54 **D. Maier (1989)**: Making Database Systems Fast Enough for CAD Applications; in ibidi 100, pp573-582.

55 **S. Ahmed, A. Wong, D. Sriram, R. Logcher (1991)**: A Comparison of Object-oriented Database Management Systems for Engineering Applications; in R91-12, MIT, (Order no. IESL 90-03, 91-03).

56 **R. King (1989)**: My Cat is Object-Oriented; in ibidi 100, pp23-30.

57 **P. P. Chen (1976)**: The Entity-Relationship Mode-Towards a Unified View of Data; in ACM Transactions on Database Systems 1(1), pp9-36.

58 **M. J. Smith, D. C. P. Smith (1977)**: Database Abstraction: Aggregations and Generalisations; in ACM Transactions on Database Systems 2(2),.

59 **J. Peckham, F. Maryanski (1988)**: Semantic Data Models; in ACM Computing Surveys 20(3), pp153-189.

60 **G. Gardarin, P. Valduriez (1989)**: Relational Databases and Knowledge Bases; Addison-Wesley.

61 **G. Blair, J. Gallagher, D. Hutchison, D. Sheperd (1991)**: Object-Oriented Languages, Systems and Applications; Pitman, London, ISBN 0-273-03132-5.

62 **D. G. Bobrow et al. (1986)**: Common Loops: Merging Lisp and Object-Oriented Programming; in ibidi 95, pp17-29.

63 **P. Wegner (1987)**: The object-oriented classification paradigm; in Research Directions in Object-Oriented Programming, Eds. B. Shriver and P. Wegner, MIT Press, Cambridge, MA, pp479-560.

64 **O. Nierstrasz (1989)**: A survey of Object-Oriented Concepts; in ibidi 100,.

65 **R. Bretl et al (1989)**: The Gemstone Data Management System; in ibidi 100,.

66 **W. R. LaLonde (1989)**: Designing Families of Data Types Using Exemplars; in ACM TOPLAS 11(2), pp212-248.

67 **C. Hewitt, P. Bishop, R. Steiger (1973)**: A universal, modular Actor formalism for Artificial Intelligence; in International Joint Conference on Artificial Intelligence, pp235-245.

68 **K. Kahn (1979)**: Creation of Computer Animantion from Story Descriptions; PhD thesis, MIT.

69 **H. Lieberman (1987)**: Concurrent Object-Oriented Programming in Act 1; in Object-Oriented concurrent programming, MIT press.

70 **H. Lieberman (1986)**: Using Prototypical Objects to implement Shared Behaviour; in ibidi 95, pp 214-223.

71 **L. A. Stein (1987)**: Delegation Is Inheritance; in ibidi 96, pp 138-146.

72 **D. Ungar, R. B. Smith (1987)**: Self: the power of simplicity; in ibidi 96, pp 227-242.

73 **A. Mercado Jr. (1988)**: Hibrid: Implementing Classes with Prototypes; Master's thesis, Tech Report CS-88-12, Brown University, Providence, RI, July 1988.

74 **G. A. Agha (1987)**: ACTORS: A model of concurrent computation in distributed systems; MIT Press, Cambride, MA.

75 **K. J. Lang, B. A. Pearlmutter (1988)**: Oaklisp: An Object-Oriented Dialect of Scheme; in Lisp and Symbolic Computation 1(1), Kluwer Academic, pp. 39-51.

76 **S. Kuldoff (1990)**: CAPS; Aachen and Dublin, Polydata Ltd.

77 **S. Ogden, J. Zucker, A. Demaid (1993)**: Adding partial templates to class templates: modelling property commonalities in a product-engineering information system; in ibidi 96,.

78 **S. E. Keene (1989)**: Object-Orientation Programming in Common Lisp: A Programer's Guide to CLOS; Reading, MA, Symbolics Press/Addison-Wesley.

79 **HyperCard (1985)**: Apple Computers Inc; California, http://hypercard.apple.com/

80 **S. Ogden, J. Zucker, A. Demaid, (1994)**: Access Enhancement Objects for Data Management in Smalltalk; in Internal Report, Design Department, Open university, UK.

81 **D. J. Penney, J. Stein (1986)**: Class modification in the Gemstone object-oriented DBMS; in ibidi 95, pp111-117.

82 **J. Grant, T. Sellis (1987)**: Deductive Heterogeneous Databases; in Methodologies for Intelligent Systems, Ed. Zbigniew W. Ras and Maria Zemankova, Elsevier, ISBN 0-444-01295-8.

83 **D. A. Moon (1986)**: Object-Oriented programming with Flavors; in ibidi 95,.

84 **G. A. Pascoe (1986)**: Encapsulators: A New Software Paradigm in Smalltalk-80; in ibidi 95,.

85 **C. A. R. Hoare (1973)**: Monitors: An operating system structuring concpt; in Comm. of ACM 18(10), pp549-557.

86 **S. Ogden, J. Zucker, A. Demaid (1993)**: Access Enhancement Objects for Storage and Visualisation in a Smalltalk Information System of Engineering Properties; in IEA/AIE 93,.

87 **A. Tönne (1990)**: The ISAM Toolbox version 2.1; vender Georg Heeg, Dortmund, http://www.heeg.de/

88 **Tigris™ (1991)**: Tigre Interface Designer, Tigre Object System; 3004 Mission Street, Santa Cruz, CA 95060, Apparently Discontinued.

89 **BOSS© (1989)**: Binary Object Storage System for Smalltalk-80; Xerox Intellegent Systems Laboratory, Palo Alto, California. Also published by UMIST Smalltalk Goodies Library.

90 **S. Khoshafian, D. Frank (1988)**:Implementation techniques for object-oriented databases; in Advances in Object-Oriented Database Systems, eds. Banerjee, J., Kim, W. and Kim, K., Bad Münster, West Germany, Springer-Verlag.

91 **A. Goldberg, D. Robson (1983)**: The Implementation; in ibidi 2, pp 541-566.

92 **A. Goldberg (1984)**: Finding Out About System Classes; in Smalltalk™-80 the Interactive Programming Environment, Addison-Wesley, pp161-194.

93 **D. M. Harland, B. Drummond (1991)**: Rekursiv Object-oriented Hardware; in ibidi 61, pp270-298.

94 **G. Larson (1978)**: Dynamic Hash Algorithms, ibidi 37.

95 **OOPSLA (1986)**: Object-Oriented Programming Systems, Languages and Applications; Conference Proceedings of American Computing Machines (ACM), eds., SIGPLAN Notices, 21(11).

96 **OOPSLA (1987)**: Second Object-Oriented Programming Systems, Languages and Applications; Conference Proceedings of American Computing Machines (ACM), SIGPLAN Notices 22(10).

97 **OOPSLA (1993)**: Object-Oriented Programming Systems, Languages and Applications; Conference Proceedings of American Computing Machines (ACM), eds., SIGPLAN Notices, ?.

98 **ECOOP (1989)**: European Conference on Object-Oriented Programming; .

99 **IEA/AIE (1993)**: Industrial and Engineering Applications of Artificial Intelligence and Expert systems; Proceedings of the Sixth International Conference, Edinburgh, Scotland, June 1-4, 1993, Gordon and Breach Science Publishers.

100 **W. Kim, F. H. Lochovsky (1989)**: Object-Oriented Concepts, Databases, and Applications; ACM press, Addison-Wesley, NY, ISBN 0-201-14410-7.

101 **M. Lenzerini, D. Nardi, M. Simi (1990)**: Inheritance Hierarchies in Knowledge Representation and Programming Languages; Wiley.

102 **M. Minsky (1968)**: Semantic Information Processing; MIT Press.