Comparing Student Software Designs Using Semantic Categorization

Eckerdal, Anna; Ratcliffe, Mark Bartley; McCartney, Robert; Moström, Jan Erik; Zander, Carol

Publication date:
2005

Citation for published version (APA):
Eckerdal, A., Ratcliffe, M., McCartney, R., Moström, J. E., & Zander, C. (2005). *Comparing Student Software Designs Using Semantic Categorization*.

Comparing Student Software Designs Using Semantic Categorization

Anna Eckerdal
Department of Information Technology
Uppsala University
Uppsala, Sweden
Anna.Eckerdal@it.uu.se

Robert McCartney
Department of Computer Science and Engineering
University of Connecticut
Storrs, CT 06269 USA
robert@cse.uconn.edu

Jan Erik Moström
Department of Computing Science
Umeå University
901 87 Umeå, Sweden
jem@cs.umu.se

Mark Ratcliffe
Department of Computer Science
University of Wales
Aberystwyth, Wales
mbr@aber.ac.uk

Carol Zander
Computing & Software Systems
University of Washington, Bothell
Bothell, WA, USA
zander@u.washington.edu

ABSTRACT

This paper examines the problem of studying and comparing student software designs. We propose semantic categorization as a way to organize widely varying data items. We describe how this was used to organize a particular multi-national multi-institutional dataset, and discuss the implications of using such techniques for education researchers.

1. INTRODUCTION

A fundamental goal of computer science programs is that their graduates be able to design software systems. This suggests that it is important to assess whether this goal is being met. There is, however, no agreed-upon method to do this.

A fairly direct approach would be to analyze student-produced designs for a common set of tasks. If students are allowed to design as they wish, however, the data produced will be very rich and varied, even in controlled settings. Although this is positive in that it can illustrate the range of approaches that students take, the complexity and variability of the designs can make them difficult to analyze and compare.

In this study, we worked with written designs produced by near-graduating students under stringent interview conditions. These designs were collected as part of a larger study, the "Scaffolding" experiment; a multi-national, multi-institutional project looking at the approach students take to software design.

The Scaffolding study looked at two different groups of students, novices and near graduates, as well as their educators. Its overall focus was developmental—so its questions were largely comparative, as in "How do beginning students, finishing students, and educators differ in the way they design software?"

By contrast, we are specifically interested in students at the end of their academic training, when they are presumably prepared to work professionally. The overall question is: Can students near graduation design software systems? In order to address this and other related questions, it is necessary to characterize the designs for evaluation and comparison. We did this by grouping the designs into categories that were meaningful for our purposes.

The structure of this paper is as follows. First, we provide an overview of the Scaffolding experiment: the larger project in which the data were collected. Next, we describe our methods: the processes by which we categorized and analyzed the designs. Then we compare the results of these processes with data from other research. Finally, we present our conclusions, and provide suggestions for future study.

2. THE SCAFFOLDING STUDY

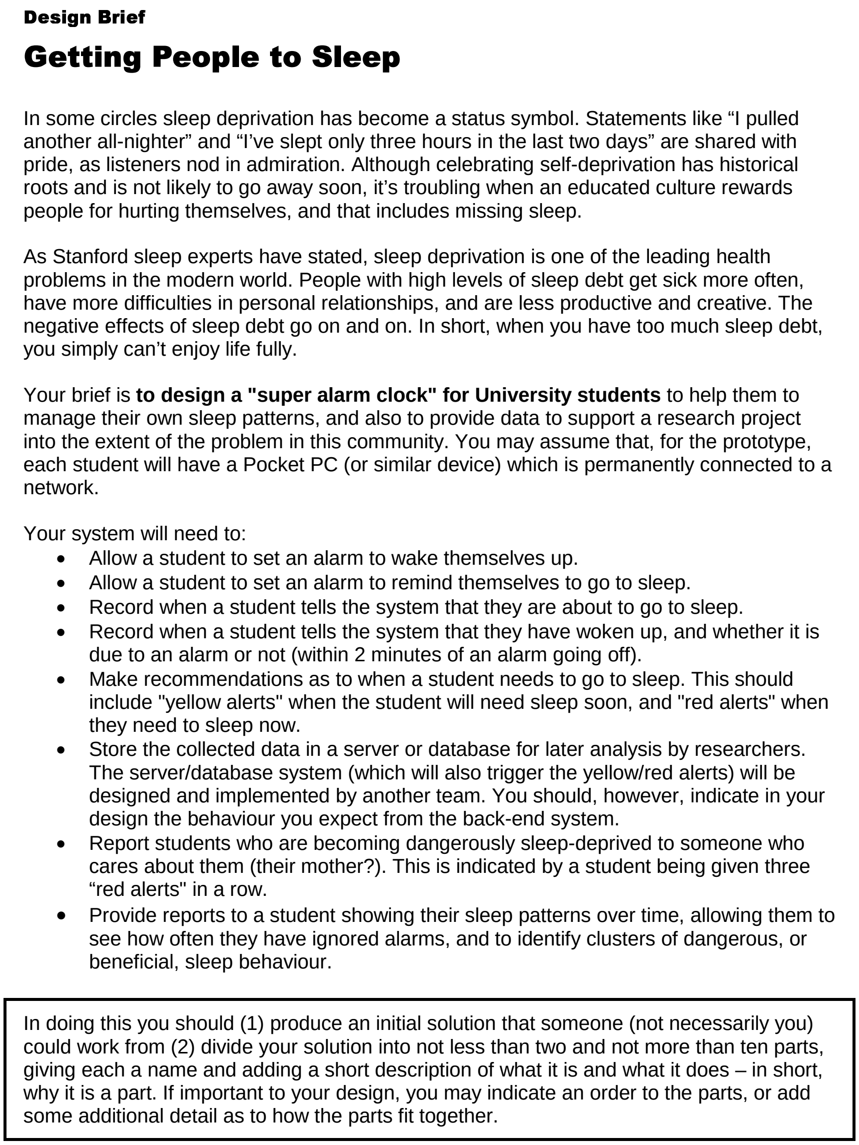
The Scaffolding study[1, 2, 12] was a multi-national, multi-institutional study that looked at various aspects of software design. It included 314 subjects from 21 institutions in the US, UK, Sweden, and New Zealand. The subjects were drawn from three pools:

First competency students (136 subjects) : students who could be expected to program at least one problem from the set proposed by McCracken et al. in [7].

Graduating seniors (150 subjects) : students within the last eighth of a Bachelor degree program.

Educators (28 subjects) : faculty members who teach computer science courses.

Each subject performed two tasks, a *design* task, in which they designed a software system from its description, and a *design criteria prioritization* task, in which they ranked software design criteria by importance under different design scenarios. These tasks were performed one-on-one with a researcher who took notes, answered questions, and made an audio recording of the session. In addition, academic and demographic information was collected for each subject: age, gender, grades, number of CS courses taken, knowledge of different programming languages, and so forth.



Design Brief

Getting People to Sleep

In some circles sleep deprivation has become a status symbol. Statements like “I pulled another all-nighter” and “I’ve slept only three hours in the last two days” are shared with pride, as listeners nod in admiration. Although celebrating self-deprivation has historical roots and is not likely to go away soon, it’s troubling when an educated culture rewards people for hurting themselves, and that includes missing sleep.

As Stanford sleep experts have stated, sleep deprivation is one of the leading health problems in the modern world. People with high levels of sleep debt get sick more often, have more difficulties in personal relationships, and are less productive and creative. The negative effects of sleep debt go on and on. In short, when you have too much sleep debt, you simply can’t enjoy life fully.

Your brief is **to design a "super alarm clock" for University students** to help them to manage their own sleep patterns, and also to provide data to support a research project into the extent of the problem in this community. You may assume that, for the prototype, each student will have a Pocket PC (or similar device) which is permanently connected to a network.

Your system will need to:

- Allow a student to set an alarm to wake themselves up.
- Allow a student to set an alarm to remind themselves to go to sleep.
- Record when a student tells the system that they are about to go to sleep.
- Record when a student tells the system that they have woken up, and whether it is due to an alarm or not (within 2 minutes of an alarm going off).
- Make recommendations as to when a student needs to go to sleep. This should include "yellow alerts" when the student will need sleep soon, and "red alerts" when they need to sleep now.
- Store the collected data in a server or database for later analysis by researchers. The server/database system (which will also trigger the yellow/red alerts) will be designed and implemented by another team. You should, however, indicate in your design the behaviour you expect from the back-end system.
- Report students who are becoming dangerously sleep-deprived to someone who cares about them (their mother?). This is indicated by a student being given three “red alerts" in a row.
- Provide reports to a student showing their sleep patterns over time, allowing them to see how often they have ignored alarms, and to identify clusters of dangerous, or beneficial, sleep behaviour.

In doing this you should (1) produce an initial solution that someone (not necessarily you) could work from (2) divide your solution into not less than two and not more than ten parts, giving each a name and adding a short description of what it is and what it does – in short, why it is a part. If important to your design, you may indicate an order to the parts, or add some additional detail as to how the parts fit together.

Figure 1: The design brief that was given to the subjects.

For the *design task*, subjects were given a one-page “design brief” that described the behavior of the desired system: a “super alarm clock” for college students. This brief is shown in Figure 1. Subjects were given as much time as they wanted to perform this task, during which they could ask questions of the researcher. After the subject indicated that they had finished, they were asked by the researcher to identify the parts of their design. The generic term *part* was used (as opposed to object, module, package, etc.) to reduce the introduction of bias. Information collected here was their design—everything they wrote on paper—plus the researcher’s notes which included the time spent, the subject’s part names, and any other observations that the researcher chose to make.

Most of the analyses of the designs in the scaffolding study were comparisons between groups. Some comparisons were based on the design artifacts, using features to partition the designs into groups, then examining the frequency of each partition across subject groupings. These partitions were based on:

- The form of the dominant representation used: text; standard graphical (such as UML); *ad hoc* graphical; code or pseudo-code; mixed (no one dominant)
- Whether the subject grouped parts into larger parts
- Whether the design showed any interactions between the parts
- Whether the subject recognized ambiguity or made assumptions (either explicit in the artifact or in statements to the researcher during design), and/or asked questions to resolve ambiguities

The results of these comparisons were mixed. Most of these showed significant trends from first-competency to graduating seniors to educators: increasing use of standard graphical representations and decreasing use of text-only descriptions; increased representation of interactions among parts; increased recognition of ambiguity in the design task and asking questions to resolve ambiguity. The grouping of parts, however was significantly more frequent among educators than students—the student groups were not significantly different.

All of these analyses involved extracting easily-recognized features from the artifacts—features that might be used to distinguish among groups of designs—but did not really attempt to characterize the designs *as designs.*

By contrast, the focus of this study is quite different: how do students design when they are at the end of an undergraduate computing program? To address this question, we undertook a detailed examination of the design artifacts (all of the material written by the subjects) produced by the graduating seniors group.

3. CATEGORIZATION METHODS

The goal of the present study was to examine students' abilities to design software, using their written designs as the primary data. To organize and simplify these data for analysis, we categorized them into groups of similar designs. We chose a data-driven approach for this categorization, with the intention that the categories reflect similarities that we observed in the data. Moreover, we intended that the observed similarities and differences be meaningful relative to the design task.

3.1 Categorization and classification

We categorized, as opposed to classified, these designs. We distinguish these terms as in Jacob[5]. Simplifying somewhat:

Categorization assigns items to categories based on semantic similarity in context (design in our case). Members have *graded typicality*, that is, they can be a more or less typical for their category, so categories can meaningfully be based on prototypes. Boundaries between categories can be fuzzy.

Classification assigns items to classes based on necessary and sufficient membership conditions independent of context. From this it follows that classes are disjoint with well-defined boundaries, and all members are equally representative of their class.

We grouped designs based on their semantics, that is *what* they communicate rather than *how*, and how well they met the stated requirement in the design brief that the design be something that "someone (not necessarily you) could work from."

Based on this approach, we developed six categories of designs, shown in Figure 2, which are ordered relative to the degree to which the above requirement was met.

These category descriptions have a distinctive characteristic: they include a description of category members in general, and refer to a typical example, or *prototype*[1]. The descriptions include qualitative terms without clear boundaries. For example, "add a small amount" in *Skumtomte*[2] and "include some significant work" in *First step* both refer to amounts of added information. The prototypes provide guidance in making such distinctions. In this example, if the amount of added information is closer to that in the *Skumtomte* prototype than either of the *First step* prototypes, then it would be considered "a small amount." This provides a mechanism for dealing with "fuzzy" category boundaries, but suggests that it may be difficult to precisely categorize some of the artifacts.

3.2 Developing the categories and tagging designs

The processes of developing the categories and assigning the designs to categories, were data-driven and integrated—both the category descriptions and the previous design assignments changed as the category assignment, or *tagging* progressed.

The initial categorization was based on examination of 20 randomly-chosen designs. After a number of other attempts based on syntactic features, one researcher proposed a categorization based on semantic design content. This first attempt had five categories, each described by both a written description and specific examples of the category. This proposed set of categories is shown in Figure 3.

The key features of this draft categorization (and the final categorization that followed) are that the features are based on understanding what the design communicates, and that there are prototypes given for each category. Based on these descriptions and prototypical designs, each researcher individually tagged (that is, assigned to a category) a group of 70 designs. Comparing the values from the different researchers, we observed that around 60% of the tags agreed (considering all pairwise comparisons). Of the disagreements, most were between *Restatement* and *Partial attempt of a design.* This level of agreement is actually fairly good, given that the categories are based on semantic features of the designs.

In order to reach agreement on the categories and assignments, four of the researchers spent a number of days together in a conference room, projecting designs on the wall, and discussing each design until we agreed on its category. This process was interleaved with the refinement and modification of the categorization to try and best accommodate the observed data.

The designs that were most difficult to categorize were those that were more than a *Restatement* but less than a *Complete* design—the proposed *Made a partial attempt* and *Made the first step* categories did not correspond well to the observed designs. Closer examination of these designs revealed a number of possible categories:

1. There were a lot of designs that added some insignificant information—more than a restatement, but not much that would be helpful to someone implementing the design.

2. There were designs that had most of the content of a *Complete* design, but were lacking some aspects–either the communications between the parts was not explicit, or the part responsibilities were missing.

3. There were designs that concentrated on one aspect of

[1] These prototypes are actual designs from the dataset, not general descriptions.

[2] The Swedish word *Skumtomte* refers to a pink-and-white marshmallow Santa Claus, a traditional Christmas confection. It looks like there is something there, but it is only shaped and colored marshmallow fluff.

Nothing This category has designs with little or no intelligible content. These tend to be very short, typically a single unlabelled diagram. There are very few of these.

Restatement These are designs that merely restate requirements from the task description (Figure 1). A typical example is a list of functions that correspond to the bulleted items in that description. These have no design content that was not stated in the description.

Skumtomte These are designs that add a small amount to restating the task. Some subjects added a small amount of information in text, or a drawing of a GUI with no description of its design, or some unimportant implementation details. There is no overall system view, nor is there any significant work on any of the modules. A typical member would have a list restating the bullet items, plus one or two details, such as a piece of code to calculate the difference between two times, a drawing of the alarm clock's screen, or the designer's preferences for programming language or development environment.

First step Designs in this category include some significant work beyond the description: either a partial overview of the system (identifying the parts, but generally not identifying how they are related in the system) or the design of one of the system's components, such as the GUI or the interface to the database. There are two prototypical members. One has a partial overview expressed as a UML-like diagram (or text on single page) that identifies the system modules but does not show their interactions; the other gives a fairly detailed description of the GUI, but provides no overview of the system.

Partial design A partial design provides an understandable description of each of the parts and an overview of the system that illustrates the relationships between the parts. The parts descriptions may be incomplete or superficial, and the communications between the parts is not completely described. A typical example is a design that describes the parts, and has an architecture diagram with links between communicating parts, but no description of what is being communicated.

Complete These designs show a well-developed solution, including an understandable overview, part descriptions that include responsibilities, and explicit communication between the parts. A typical example uses multiple formal notations (e.g. UML, Use cases, CRC cards) as well as text.

Figure 2: The six categories used for design artifacts.

the design, but little else. Most of these involved the design of the GUI or the database.

4. There were designs that provided some overview information—a drawing showing the system components, e.g., but did not show the relationships between parts.

Based on these, we replaced the proposed *Made a partial attempt* and *Made the first step* categories with three new categories: *Skumtomte* and *Partial design*, corresponding to 1. and 2. above, and *First step*, which are the designs corresponding to 3. and 4. Although the latter two (from 3. and 4.) could have been distinct, they were classified together as each of them showed significant progress past the *Skumtomte*, and neither provided a clear overview, or system architecture. Implicit in these choices is that *system overview*, *responsibilities of parts*, and *relations and communications between the parts* are important parts of the design process, a view consistent with software engineering texts such as [4, 11].

During the above process, we categorized 111 of the designs, reserving 38 for subsequent individual tagging, as a way to check how well we agreed on individual ratings after the categories were set. Comparing the four researchers' tags, we found that 55% of the tags agreed, that is, considering all possible pairs for each design, 55% were in agreement. To put this number in perspective, however, if three researchers agree on a tag and the other disagrees, only half of that design's tags match.

Finally, the researchers met and discussed the last 38, reaching agreement on all of the tags.

3.3 Student design results

The distribution of the designs among the categories is shown in the frequency plot in Figure 4. Given that the categories can be naturally ordered relative to the communicated design content, we can easily describe the overall performance:

- 21% of the designs were simply restatements of the specification or less—no value added at all.

- 41% of the designs were *Skumtomte*: those that added an insignificant amount beyond the specification, and, in particular, did not produce any usable "design content".

- 29% of the designs were in the *First step* category, showing some progress toward a design—a partial overview, or significant progress toward the design of one part of the system.

Failed *(Corresponds to* Nothing *in final categorization)* There is no coherent text/drawing in the design.

Typical example is design 292.

Restated the task *(Corresponds to* Restatement *in final categorization)* The problem has been restated, usually as a number of bullets: "It should sound an alarm after X minutes", etc. But there is no original work there. The bullet might be divided into "parts" but these parts are essentially the text of the design brief. Note that these might include graphics with connections but there is still no additional information compared to the task description.

Example: 756, 606.

Made a partial attempt at a design Here you can find some original work, he/she has attempted to add some information but has not continued to make an actual design.

Example: 925

Made the first step of a design He/she has started to make a design, drawing up different "objects", relationships, etc. Contains a substantial part of original work and the task has been analyzed, but the actual description is not finished and is in a "sketch" stage.

Example: 217

Have actually made a design *(Corresponds to* Complete *in final categorization* This group contains designs that are actual designs: the problems have been analyzed and a first design has been made. This means that a coherent description is made and it's possible to continue the design without re-analyzing the whole question. Does not need to make it possible for someone else to start implementing the system.

Example: 734

Figure 3: Initial draft of categories. Numbers are ID codes of particular designs. *These are the initial categories based on twenty designs: the final categories evolved from these during tagging the rest of the data.*

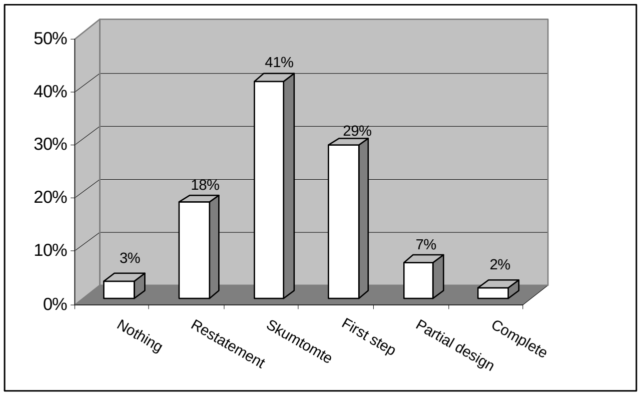


Figure 4: Frequencies of observations in each of the design classifications (based on 149 observations).

- 9% produced *Partial* or *Complete* designs: those including an understandable system architecture/overview, with parts and their interactions explicitly stated. Of these, less than a third produced *Complete* designs, with explicit part responsibilities and inter-part communications.

All in all, a poor performance from students who are near graduation: over 20% produced nothing, and over 60% communicated no significant progress toward a design.

In terms of analysis, categorizing the designs was quite time-consuming, even after the categories were defined, as it required extracting the meaning from the artifacts, many of which are poorly organized and nearly illegible. Gaining complete agreement between raters required extensive discussion for designs that did not closely match the prototypes or were difficult to read. Having placed designs in these categories, however, gives us useful information—the category that a design belongs to provides information about the communicated design content, and allows us to easily compare designs on that basis.

Regarding the distribution of designs in Figure 4, and what that says about students' design skills, it should be noted that the experimental situation might have affected their overall performance. The instruction said that "you should (1) produce an initial solution that someone (not necessarily you) could work from", but they were also told that they were going to explain the design to the researcher once it was completed—the latter might have caused them to make "sketches of solutions" instead of writing down a complete design. In the analysis, we only considered the written artifacts, and not any verbal explanations from the researchers' notes. All the authors agreed that a few of the designs seemed to indicate that the students knew much more than they communicated.

4. SYNTAX AND SEMANTICS

Extracting meaning from a design, which is necessary to categorize it, is difficult as it requires a global understanding of the artifact. Recognizing syntactic features, by contrast, does not require such deep understanding; there are syntactic features that are both local and visually apparent. With an eye toward reducing the amount of work required in analysis, we examined the relationships between our semantic categories and the recognizable syntactic features.

Before and during the categorization, we identified a number of candidate syntactic features that might be used to characterize designs. Listing these and collapsing similar features, we agreed to use the following set:

Algorithm. The design contains a step-by-step recipe, for example "1 - Add A and B, 2 - Store the result in C, 3 - Copy C to Z, . . . ".

Block. Box with text in it, usually a single word.

Bulleted list. Text is organized in a bulleted, numbered, or labeled list, where each list item contains a short textual description. *If the text descriptions are each long, this should be counted as running text instead.*

Class. The design includes a class, either represented in code or in a diagram. *This is not counted separately if in UML diagram.*

Code. Code snippets are included in the design, typically something similar to "tmp = a; a = b; b = tmp".

CRC. The student has included CRC (Class, Responsibility, and Collaboration) Cards in the design, typically in the form of a drawing of a filled-out CRC Card.

Database. The design contained a representation of the database. The representation was more detailed than a simple square with "database" inside, usually a description of the different tables in the datbase.

Event-Action. The design is described in terms of "when X happens the system should do Y", more elaborate than single sentences.

Flowchart. The design includes a graphical flowchart.

Methods. A method (function) is described. *Not counted separately if methods are included in the a class description or in a UML diagram.*

Other diagram. Miscellaneous drawings *(not counted elsewhere).*

Overview diagram. A *diagram* showing an overview of the main parts of the design.

Running text. More that a couple of sentences of text. *An outline or bulleted list where items contains more than a paragraph of text is considered to be running text.*

Simple user interface. A simple drawing of the user interface, for example a rectangle inside another rectangle symbolizing a PDA with a hardware button.

Text outline. The design contains an hierarchical outline where each item is described by, at least, a few words.

User Interface. An elaborate drawing of the user interface for example including data, labels, diagrams, colors, etc.

UML. The design includes a UML diagram. It does not have to be complete or correct.

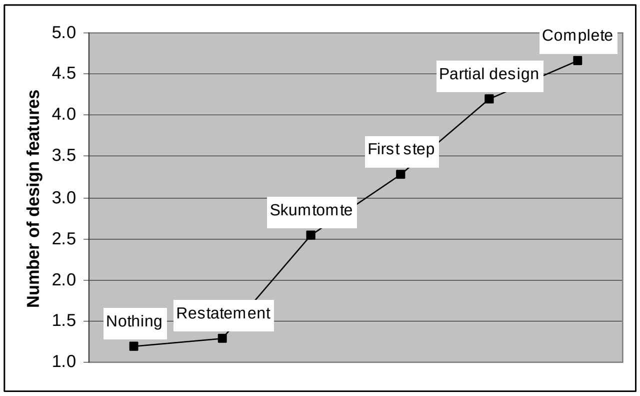


Figure 5: The average number of different features in designs of each category

Use case. A use case description is included.

User picture. The users of the system is explicitly drawn as a stick figure or in a similar way. *Not counted if part of a Use case.*

We then re-examined all of the designs relative to this feature set, determining, for each design, whether or not each of the above features was present. (We did not attempt to count how many of each feature were present.)

At the most abstract level, we compared the number of these features present in each design for each of our semantic categories. We summarize this in Figure 5. The diagram clearly shows the feature count increases as the designs become better. The designs in higher categories tend to be syntactically richer, in part because they tend to include some of the formal structures like UML and Use case diagrams. As Table 1 indicates, however, there is a large range of feature counts in each category.

If the number of features measures the syntactic richness of the design, the overall length of the design is a measure of the syntactic quantity of the design. As seen in Table 2, the higher category designs tend to be longer. As with the feature count, there is a great deal of variance, however, as illustrated by the minimum and maximum values for each category: there are examples of short *Complete* designs, as well as extremely long *Skumtomte* designs.

5. RESULTS AND RELATED WORK

We summarize the results, which concern developing and applying the categorization, as well as the comparison with syntactic features, as follows:

- It is possible to develop cogent and purposeful categories based on design content. These groups provide a meaningful basis of comparison in the context of our problem and concerns.

- Some designs will be hard to tag, as it is sometimes very difficult to extract information from their unstructured data. On the other hand, by creating a well-defined, yet broad categorization, a design's category assignment is independent of the methods used to express the information.

Table 1: The average, minimum, and maximum number of features counted per design, by category

No of features	Nothing	Restatement	Skumtomte	Start of design	Partial design	Complete design
average	1.20	1.30	2.57	3.30	4.20	4.67
minimum	1	1	1	1	2	4
maximum	2	3	6	5	6	6

Table 2: Length of designs by category. Entries are in number of pages except for last row, where entries are percent of designs at least three pages long.

Number of pages	Nothing	Restatement	Skumtomte	Start of design	Partial design	Complete design
average	1	1.9	2.9	3.7	5.2	5.0
minimum	1	1	1	1	2	2
maximum	1	4	9	9	10	9
≥ 3	0	18.5%	54.8%	66.7%	90%	66.7%

- Gaining complete agreement between raters is possible with discussion, but probably not without it. Since the category memberships are graded (based on closeness to prototypes), this is to be expected, as there will be designs that fall between prototypes and are hard to tag. Additionally, the legibility problems mean that not all raters will understand what is being presented, a problem that discussion can address.

- Semantic content does correlate with syntactic richness (number of features) and syntactic content (as measured by length); these syntactic features are easier to extract. However, due to the variability, syntax is a relatively poor predictor of category, so if the meaning is what is important, a semantic categorization should be more valuable.

5.1 Comparison with related work

As has been described above we have not tried to put the designs into some pre-defined categories. Instead we have tried to let the data speak for itself. Comparing our results with previous work suggests that this has been a judicious approach.

Whilst there has been much research over the years into student coding [9], there is far less available on design. Results from both of these areas, however, are consistent with what we have seen here.

Of particular interest to student design is recent work by McCracken [8] which focusses on the *learning* of design skills. He contrasts design and programming, and suggests study techniques for studying design behavior: *In situ* observation, Retrospective interviews, and Protocol analysis. The Scaffolding data collection was an example of *In situ* observation in his terms. The big difference between his approaches and ours is that his concentration is on the *process* of designing as opposed to the *results*. We chose not to use the collected process data from the Scaffolding study: apparent differences in experimental technique among the researchers make these data difficult to compare, and the extracting of the subject behaviors is quite labor intensive—too intensive given the size of this entire dataset.

DuBoulay comments on novice programmers' inability to grasp the whole program and the relation between the main parts: "This ability to see a program as a whole, understand its main parts and their relation is a skill which grows only gradually." [3, p. 59]. This is in line with our conclusions that overview of the parts and relations between parts are important features found only in the more advanced designs.

Soloway et al. [10] discuss how to teach design. Based on a study with expert software designers the authors advocate five phases in developing a design. The phases are:

- *Phase 1: Understand problem specification.* The goal here is simply to understand what the problem is asking for.

- *Phase 2: Decompose problem into programmable goals and objects.* During this phase, the objective is to "lay the components of the solution on the table", that is, decompose the problem and identify the solution components.

- *Phase 3: Select and compose plans to solve problems.* During this phase the pieces of the solution are woven together, that is, the components are composed to form a working whole.

- *Phase 4: Implement plans in language constructs.*

- *Phase 5: Reflect–Evaluate final artifact and overall design process.* When all is said and done, a good strategy is to look back over what has been done and learn from both the successes and failures.

It is interesting to compare Soloway's phases with the data driven classification in this study. Soloway's first phase can be directly mapped to our *Restatement* and *Skumtomte* groups, where the designs indicated that the students were trying to understand the problem and form a model of it. It is worth noting that the majority of the designs have not passed the first of Soloway's phases.

The second phase, to "lay the components of the solution on the table," we interpret as to get an overview of the solution. The overview sometimes occurs in *First step* and always in the two top categories in our study. These designs showed an understanding of the problem as well as how it should be solved.

In the third phase "the pieces of the solution are woven together, that is, the components are composed to form a working whole". This assumes that both the responsibilities and relations (connections and communications) among the parts are determined, things which are part of the top two categories in our study.

The fourth and fifth phases, implementation and evaluation, are not within the scope of the task given in our study. However, there were examples in designs of all categories but *Nothing* of students including code fragments or making other implementation decisions (fourth phase), especially in the *Skumtomte* and *First step* categories. Additionally, there were designs in the top groups that showed reflection and evaluation of their solutions, and so illustrated Soloway's fifth phase.

Soloway et al. describe the phases as "activities" and that "there may be (will be!) some jumping around – back and forth" based on the results from the study with the expert designers. This agrees with our data. We do not see a monotonic progression through the phases in the written designs, and we see traces of many of these phases throughout the documents. Our data here are not very strong, however, as it is not necessarily the case that the order of things in the design corresponds to the order in which the work was done. Since many of the designs seem to illustrate only the first of Soloway's phases, whether they jump around is a moot question.

6. CONCLUSIONS

The results of this study show that a semantic categorization is both possible and practical in studying designs. Such an approach has one fundamental strength: the categories developed are consistent with the information desired from the data. Additionally, these results suggest that useful design data might be extracted from the design artifacts alone. Although categorizing these artifacts can be somewhat labor-intensive, it is far less so than the standard alternative, extracting process information from observations of people designing, which means we might practically examine larger datasets.

The other strength of using semantic categories is that they are relatively insensitive to stylistic differences. In this study, researchers from three different countries and five different institutions were able to compare designs from 21 institutions from different cultural and linguistic traditions, and could compare designs with radically different presentations—from purely textual to purely diagrammatic. In the future, we intend to examine the range of applicability of such techniques in other multi-national situations.

7. ACKNOWLEDGMENTS

The authors would like to thank Sally Fincher, Marian Petre, Josh Tenenberg, the Scaffolding workshop participants, and the National Science Foundation (through grants DUE-0243242 and DUE-0122560) for their support and encouragement. Thanks also to Kate Sanders and Lynda Thomas for helpful comments and suggestions on drafts of this paper. Finally, thanks to the reviewers of this paper, and the participants of Koli Calling, for their questions and suggestions.

8. REFERENCES

[1] K. Blaha, A. E. Monge, D. Sanders, B. Simon, and T. VanDeGrift. Do students recognize ambiguity in software design? a multi-national, multi-institutional report. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 615–616, 2005.

[2] T. Chen, S. Cooper, R. McCartney, and L. Schwartzman. The (relative) importance of software design criteria. In *Proceedings of the 10th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2005)*, pages 34–38, Monte da Caparica, Portugal, June 2005.

[3] B. DuBoulay. Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1):57–73, 1986.

[4] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Pearson Education, Inc., Upper Saddle River, NJ, second edition, 2003.

[5] E. K. Jacob. Classification and categorization: A difference that makes a difference. *Library Trends*, 52(3):515–540, Winter 2004.

[6] R. McCartney, J. E. Moström, K. Sanders, and O. Seppälä. Take note: the effectiveness of novice programmers' annotations on examinations. *Informatics in Education*, 4(1):69–86, 2005.

[7] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bulletin*, 33(4):125–180, 2001.

[8] W. M. McCracken. Research on learning to design software. In S. Fincher and M. Petre, editors, *Computer Science Education Research*. Taylor and Francis Group, London, 2004.

[9] A. Robins, J. Rountree, and N. Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137 – 172, 2003.

[10] E. Soloway, J. Spohrer, and D. Littman. E unum pluribus: Generating alternative designs. In R. E. Mayer, editor, *Teaching and Learning Computer Programming*, pages 137–152. Lawrence Erlbaum Associates, Publishers, 1988.

[11] I. Sommerville. *Software Engineering*. Pearson Education, Ltd., Harlow, England, sixth edition, 2001.

[12] J. Tenenberg, S. Fincher, K. Blaha, D. Bouvier, T. Chen, D. Chinn, S. Cooper, A. Eckerdal, H. Johnson, R. McCartney, A. Monge, J. Moström, M. Petre, K. Powers, M. Ratcliffe, A. Robins, D. Sanders, L. Shwartzman, B. Simon, C. Stoker, A. Tew, and T. VanDeGrift. Students designing software: a multi-national, multi-institutional study. *Informatics in Education*, 4(1):143–162, 2005.