

Classifying and Measuring Student Problems and Misconceptions

Alexander Hoem Rosbach

Dept. of Informatics,
University of Bergen
aro037@student.uib.no

Anya Helene Bagge

Dept. of Informatics,
University of Bergen
<http://www.ii.uib.no/~anya/>

Abstract

In this paper we report on an attempt to classify student problems and mistakes, and measuring the frequency of particular problems in a first-semester programming course. We also propose a scheme for annotating student hand-ins which may be useful both in grading and in future research.

1 Introduction

Learning programming is a complicated task that requires the student to acquire a lot of difficult and interconnected knowledge. This makes the teaching of programming a serious challenge for educators.

First off, a student must learn the language itself, that is, the syntax and semantics of the necessary language features (constructs). Then, the student should also learn typical ways of applying them. In literature this is often referred to as *schemas*, or sometimes plans, depending on the author. We will only use the term schema, and we will use it to refer to a certain way of combining language constructs to solve specific problems, i.e. a schema may be a way of combining language constructs to calculate the sum of the numbers in a list (array) or read and validate user input etc. These trivial schemas will give the student both a better conception of the construct semantics, and knowledge of actual usage.

In the process of learning programming the students will encounter many difficulties and challenges. Du Boulay [2] describes a list of five overlapping problem domains:

1. *General orientation* – understanding what programs are, that they can be used to solve problems and what kinds of problems are solvable.
2. *The notional machine* – having a correct understanding of how the language constructs affect the computer, e.g. correct mental models of how an array is stored and accessed in memory, how references work etc.
3. *Notation* – the correct understanding of the syntax and semantics of the language. An IDE can help greatly with this.
4. *Structures* – knowledge of schemas that can be applied to solve small sub-goals of a problem, when and where to apply them and how to combine them.
5. *Pragmatics*. Planning, developing, testing, debugging etc.

Spohrer and Soloway [8] conclude that misconceptions about language constructs (i.e., notation) does not seem to be as widespread as is generally believed. Though the semantics of the constructs may prove a bit challenging at times, the student should be provided with good explanations from the course literature and lectures.

We have examined problems and misconceptions in the setting of the INF100 course, a first-semester programming course taught at the University of Bergen. The curriculum is based on teaching basic programming in Java, and includes the imperative features of the language; objects and classes – but with little emphasis on inheritance; exception handling; simple algorithms and recursion; exception handling. The students use the DrJava¹ environment, but without using any of the advanced features of the IDE.

Our study is based on examining handed-in solutions to weekly exercises and semester assignments. We inspected 349 assignment solutions by a selection of 79 students, in order to determine what the most common problems are. Our main motivation was a desire to provide tooling targeted at particular problems, but also to provide a basis for future evidence-based, targeted teaching. The work is based on the first author's Master thesis [6], which provides more detail, and also discusses a simple tool for step-by-step evaluation of Java programs.

Our main contributions in this paper are:

- A simple taxonomy of problems (Section 2)
- A study of problems and mistakes in student assignments (Section 3)
- A scheme for annotating problems in student assignments (Section 4)

2 A Taxonomy of Problems

In this section we present the taxonomy scheme that we will be using in this paper. We will base our work on that of Spohrer and Soloway [8] who built a taxonomy for student problems where they classified into two categories, *construct-based problems* and *plan composition problems*. These categories they describe as not mutually exclusive or exhaustive. To determine which category problems should be placed in, they identified plausible underlying causes for the problems and used them as determinants.

We will use some of the causes that Spohrer and Soloway identified, refine some of them and provide additional ones that we identify, as determinants for our taxonomy. However, we have chosen other categories, *concept-based problems* and *knowledge-based problems*.

Brooks [1] describes the program comprehension process as expectation-driven, by creation, confirmation and refinement of hypotheses. This is what we attempt to capture with the *concept-based problem* category; problems that can be identified and rooted out by creating hypotheses and confirm or refine them, and by adjusting the mental models of how programs and programming works. These are also the problems where interactive tools may be of use in teaching. This is in contrast to the *knowledge-based problems*, where the students lack the necessary knowledge to understand a problem or complete a task and may be unable to proceed without first filling the knowledge gap.

Concept-Based Problems

Typical *concept-based problems* are those that are related to incorrect schema applications or misconceptions of the notional machine. Problems we classify as *concept-based*

¹<http://www.drjava.org/>

are those where the student should be able to understand the mistake by observing the behaviour of the program. With a combination of reading, manually tracing the code and observing the results of executing the program, while experimenting with the data set, the student should be able to locate and correct the mistake. Instances of these problems are expressions, statements or constructs in a program that are either unnecessary and does not affect the result of the program, or in some way prevent the program from working as it should.

Resolving persistent concept-based problems may require some degree of help from a teacher or teaching assistant, for example by prodding the student into tracing the behaviour of the program. A debugger may be of help here; or even just teaching students to use print statements or assertions to verify their assumptions about program behaviour.

There may be several underlying causes behind concept-based problems. First of all, the student's mental models of the program itself or the programming language may be imperfect – what Du Boulay [2] calls having an *incorrect notional machine*. This may have unexpected obscure consequences that may be challenging to notice and understand. For example, the student may have a misconception of construct semantics, where the student may possess some conception of construct semantics that is incorrect, but seems to be correct based on previous experience. Though difficult, such problems are identifiable by observing the behaviour of the program.

A second issue may be *natural language translation*. The student may devise a solution to a problem in natural language, which is not straight-forwardly translatable to code. Observing the behaviour of the program, the student should be able to identify that it behaves differently than expected.

Third, we have *specialisation problems*. The student may find it challenging to apply the general schema correctly to a concrete problem. When customising the schema to the particular situation at hand, the student may deviate slightly from the correct application and create an incorrect solution.

Finally, and related to all the above issues, the student may not have enough *test data*, and may lack the skill or imagination to construct such data. The data set the student is executing the program with may not be good enough to expose the problem, and if it was supplied with the assignment, the student has no reason to suspect that it is insufficient.

Knowledge-Based Problems

Knowledge-based problems are those situations where the student does not have the proper knowledge or understanding to solve a problem. This may be due to lack of schemas to apply, or an inadequate understanding of them (thus not able to apply them). Instances of these problems may be segments in a program that the student has either left completely blank or placed an empty skeleton of a schema. Other examples of instances may be a dynamic problem solved in a static way, with the consequence that they only work for the given example data set.

Knowledge-based problems should resolve themselves through normal teaching efforts or self-study (or possibly by the student giving up and dropping out). In many cases, they should also be easier for the student to uncover and do something about – students will typically know if they have no idea how to solve a problem; figuring out that you have misunderstood something is a lot harder.

Central issues for knowledge-based problems include:

- *Interpretation of specification*. Correct interpretation of the assignment specification may be challenging. Explicit requirements may be neglected or perceived in-

correctly and implicit requirements may be missed.

- *Schema knowledge*. The student may not possess the necessary schemas or adequate conceptions of the schemas to solve the problem.
- *Existing environment*. Students may find it difficult to express their intention using the relevant existing parts of the program environment, e.g. identifying the correct variables and methods to use.

Commonly Encountered Problems

We'll now give a description of the most relevant problems encountered by our first semester students, and attempt to classify them as concept-based or knowledge-based. More details about the problem classification can be found in the first author's Master thesis [6].

$\swarrow_{\text{COND}}^{\text{BAD}}$ – **Wrong Condition** Any conditional expression that is in some way incorrect may be counted as a *wrong condition* problem. There are several special cases of this, including $\swarrow_{\text{COND}}^{\text{BDRY}}$ (below). Occurrences of this problem are primarily incorrect conditional expressions in if-sentences, though there are a few occurrences in loop-constructs.

Typical occurrences of this problem are conditions that are wrong because they use an incorrect operator, connective or method call. Concrete examples of this problem are solutions that check for reference equality instead of object equality, or over-complicated conditions that render the solution incorrect, etc.

$\swarrow_{\text{COND}}^{\text{BDRY}}$ – **Boundary Case Condition** Any conditional expression that evaluates incorrectly for some boundary case, either by going out of bounds or being too restrictive, e.g. incorrectly included or excluded boundary elements in a range. A concrete example of this is a condition that uses the inclusive less-than operator instead of the strictly less-than. This might result in an attempt at accessing an index outside of the array, or a guard that accepts too little or too much of a number range. As a consequence the program might suffer of index-out-of-bounds errors for arrays.

$\swarrow_{\text{PROB}}^{\text{LOOP}}$ – **Loop instantiation problem** Any situation where the student has had trouble instantiating a loop-construct to solve a given task, is counted as a loop instantiation problem. This includes cases where a loop is needed, but missing; partly instantiated loops, often the most trivial loop-construct schema; or loops that are far away from a working solution, often with conditions and bodies that does not resemble a solution in any way.

$\swarrow_{\text{CTRLS}}^{\text{CMPLX}}$ – **Unnecessarily complicated** Instances of this problem are those situations where the student devise an over-complicated solution that may or may not work. For example, deeply nested blocks, repetition of large segments of code, conditional structures with individual branches for each possible case the student can imagine and so on.

$\swarrow_{\text{CALL}}^{\text{NO}}$ – **Missing method call(s)** Any situation where the student has omitted a necessary, or useful, method call is counted as a *missing method call* problem. In some situations the student has still managed to create a working solution by re-inventing the wheel, i.e. implementing the functionality of the method specifically for that situation, and in others the student may have left out that part of the solution entirely. When solving assignments given in a course, the student will in some cases be provided with methods (or classes) and/or information about any library methods that could be useful.

$\swarrow_{\text{GRPNG}}^{\text{BAD}}$ – **Incorrect grouping** Any situation where the student has incorrectly connected, or failed to connect, constructs and/or statements is counted as *incorrect grouping*

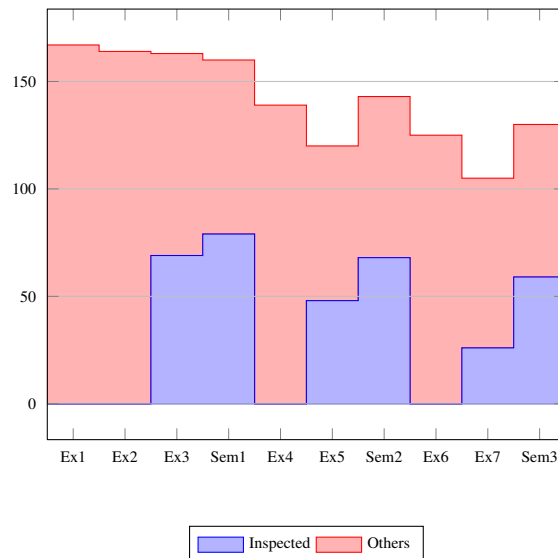


Figure 1: Inspected student solutions (INF100, Fall 2012). *Ex** are weekly exercises, *Sem** are semester assignments. Selected students scored <85% on one or more semester assignments.

problem. For example, incorrectly connected or unconnected `if` clauses; an action that should be performed after a loop is performed each iteration; placing a statement that also should depend on the condition of an `if`-structure, outside of that block.

↳^{BAD}ARITH – Erroneous arithmetic Any arithmetic expression that is incorrect with respect to the given assignment is counted as erroneous arithmetic. Common occurrences of this problem is when the student needs to calculate the difference between two numbers, and neglects that the result might be negative. The student may realise the problem but fail to find a correct solution, often statically multiplying by -1 or finding the absolute values of the individual terms instead of the entire expression. Other occurrences of this problem may be that the student fails to understand exactly what the numbers represent, what denominator they have and/or what they count.

↳^{BOOL}ACCUM – Accumulate Boolean Instances of this problem are situations where the student has neglected or failed to accumulate a result Boolean when performing an operation on a list of elements. The operation that is performed on each element returns a Boolean value that represents failure or success. This Boolean value should be accumulated by the loop-construct iterating over the list of elements in a way that detects any failure.

3 A Study of Student Problems

In order to get a clearer picture of the problems that faces first-semester INF100 students, we did a study of the submitted student assignments. We had a large number of submissions available to inspect from just one semester; in total 182 students enrolled for the course (Fall 2012) and there were 3 semester assignments and 7 weekly exercises during the course. This sums up to a total of 1820 submitted solutions, if all the students had submitted a solution for all the assignments. However since a portion of the students dropped out during the course, and students were only required to submit and pass 5 out of the 7 exercises, there was slightly fewer submissions, approximately 1400.

As this is far too many submissions to examine, we limited ourselves to the students

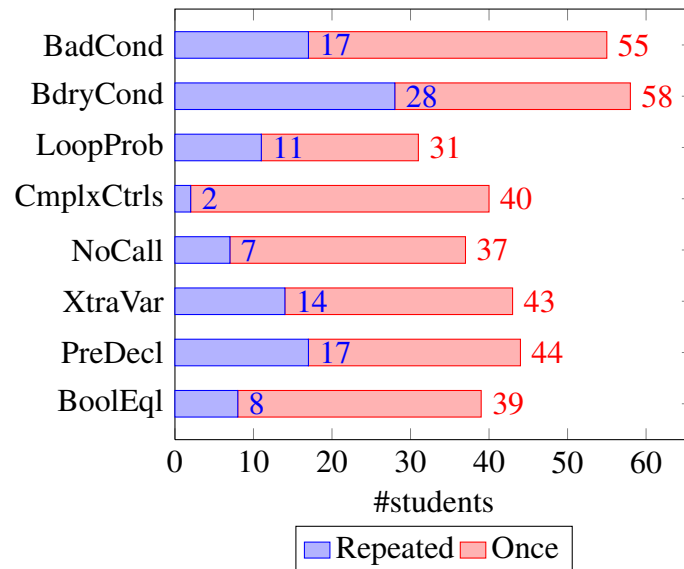


Figure 2: The most common annotations (those that the most students were annotated with).

scoring strictly less than 25 out of 30 (~85%) points on any of the three semester assignments. Students with higher scores had very few, mostly superficial mistakes. This gave us a set of 79 students, for which we selected six assignments to inspect. The first two exercises were trivial introductions to programming, and had little value to us as they only involve simple arithmetic in a *Hello World* way. The three semester assignments were sure choices, and to close the time gap between them we chose the three exercises that were given before each semester assignment. Fig. 1 shows the total mass of submissions, and the ones selected for inspection. The decrease in submitted semester assignments reflects students dropping out. For the exercises however, the decrease is mostly because the students are only required to submit solutions to five of them. Exercise 7 was especially affected by this.

Before we began inspecting the student submissions we created an initial list of typical student problems, based on our previous experience, and gave them each a unique identifier. However, during the process of annotating the source code we discovered other kinds of problems, that we had not initially identified, and also had to change some of the initial ones as they were either too general or too specific. The most common ones are presented in Section 2, see [6] for the rest.

Most Common Problems

In Fig. 2 we present a chart of the most common problems among the students, i.e. those problems that the most students was registered having. Conditions, over-complicated solutions, loop-constructs and method calls are those problems that by the data we collected was the most common problems. In the chart there are also some annotations regarding unnecessary expressions, statements and placement:

- ↘^{XTRA}_{VAR}, unnecessary variable
- ↘^{PRE}_{DECL}, unnecessarily pre-declared variable
- ↘^{BOOL}_{EQL}, checking Boolean expression for equality with Boolean literal

These three are only indicators that show that the student may have some misconception of the semantics or the notional machine. The students are quickly rid

of the ζ_{EQL}^{BOOL} and ζ_{DECL}^{PRE} misconceptions, most likely due to good feedback from the teaching assistants, as they are primarily only a “problem” in semester assignment 1 (and exercise 3 as well for ζ_{DECL}^{PRE}). The occurrences of ζ_{VAR}^{XTRA} are more thinly spread across the assignments, though most frequent in the subsequent assignments exercise 5 and semester assignment 2. We argue that the most plausible reason for this “problem” is that it is some remnant of some plan or previous attempt to implement a solution, and that the student most likely has forgotten the existence of the variable.

Conditional expressions is a concept that our data shows was a big challenge for many students. The most common kind of conditional expression problem was incorrect boundary case conditions, ζ_{COND}^{BDRY} , which as many as 58 students were registered having problems with. In addition 55 students were registered having trouble with conditions in general, ζ_{COND}^{BAD} , that is those that does not fall into a special category (like boundary case). Most of the students (48) who had troubles with boundary case conditions also had troubles with conditions in general. Respectively 28 and 17 students were registered with ζ_{COND}^{BDRY} and ζ_{COND}^{BAD} across multiple assignments.

We registered 40 students that created over-complicated solutions, ζ_{CTRLS}^{CMLPX} , though only two students created such solutions in multiple assignments. In many cases it is plausible that the reason that these kinds of solutions are submitted is that the student fears to change something that seems to work, even though it might be clear that the code segment is very complicated and/or repetitive. We also suggest that it is plausible that this is a problem that students have with newly learned concepts, and that with experience they will learn to apply the concepts properly.

Parameter Passing and References

Our experience in teaching suggested that the concepts of parameter passing and references would be a significant and common problem in the submitted student solutions. However this was not the case, and in fact very few students were seen to be having difficulties with parameter passing (less than 10%) and none with references.

The teaching assistants reported that parameter passing was a hot topic in their lab sessions, which may mean that most such problems were resolved with the help of teaching assistants before submission time.

There is one interesting case though, 22 students seem to have a small misconception regarding how parameter passing works (ζ_{PASS}^{PARAM} – more of a problem later in the semester, probably due to more advanced exercises). In some cases when they write method call statements, they declare an unnecessary variable with the same name as the formal parameter of the method, on the line above the method call.

We suggest that a reason for the lack of reference problems and the low frequency of parameter passing problems might be related to the design of the given assignments. Situations where references could have been a problem in the assignments were all related to immutable objects like String, Integer etc.

Conditions

Problems related to conditional expressions, both in if-sentences and loop-constructs, was very common among the students, see Fig. 3. Semester assignment 3 was the most difficult assignment given during the course, and not only did it stress most of the previously learned topics, it also required the students to implement a total order relation-method (compareTo from the Comparable interface). Almost all, 93.2%, of the selected

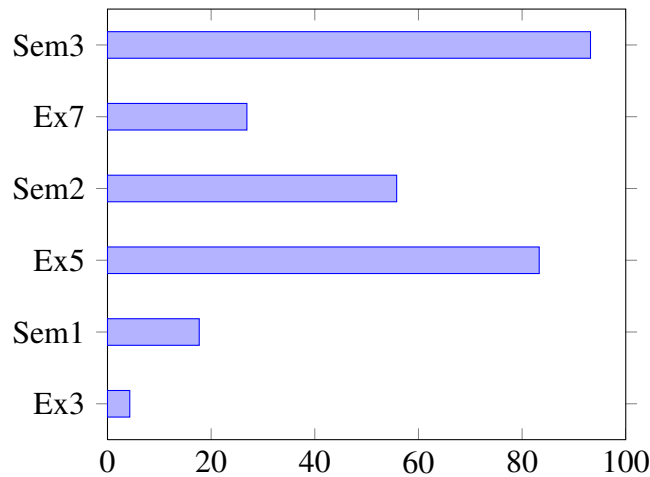



Figure 3: Percentage of the students (who submitted a solution) with some conditional problem ($\frac{\text{BDRY}}{\text{COND}}$, $\frac{\text{XTRA}}{\text{COND}}$, $\frac{\text{BRKNG}}{\text{SNTNC}}$, $\frac{\text{BAD}}{\text{COND}}$, $\frac{\text{ALMST}}{\text{CRRCT}}$).

students who submitted a solution for this assignment had some trouble with conditional expressions.

If we also take into account the problem of accumulating a Boolean value ($\frac{\text{BOOL}}{\text{ACCUM}}$), there was only one student who did not have problems. Exercise 3 and semester assignment 1 both had an introductory focus on if-sentences and loop-constructs, though the student was required to interpret and understand specifications for these structures, and instantiate them in the implementation. In these assignments few students had difficulties and most were able to express correct conditional expressions. We argue that this may show that the students are able to identify situations where these structures are required and at least able to interpret the concrete specifications of the conditions. But when the specifications and conditions grow more complex in the later assignments the students experience difficulties.

A large part of the solutions submitted for exercise 5 and semester assignment 2 also had mistakes related to conditions, respectively 83.3% and 55.8%. The listed goals of these assignments are respectively *Objects and methods* and *Classes and objects*, but there are few identified problems related to these – which may suggest that the actual focus of the assignment was incorrect.

Loop constructs

Implementing programs that requires loop constructs is a challenge for most novices, and bugs are more common in relation with these constructs than with other common tasks like input, output, syntax/block structure and overall planning [7]. This is reflected in our results as well, during the course 82.2% of the selected students had trouble related to loop constructs, either with the condition or the instantiation of the construct. The assignments where the most students struggled with loop constructs was exercise 5 and semester assignment 2 and 3 (). Though it was most difficult in semester assignment 2, where 63.3% of the selected students that submitted a solution struggled. The focus of this assignment was *Classes and objects* but there was a frequent need for loop-constructs, and the nesting between the provided Java classes was quite deep for a novice. This may have caused the student to have difficulties when creating the mental model of how the classes and objects communicate.

Method calls

During the course 60.1% of the selected students had misconceptions or difficulties with method calls. These problems were most common in semester assignment 3, where 59.3% of the selected students who submitted a solution had trouble. Most of the problems that the students had were situations where a necessary method call was missing ($\frac{NO}{CALL}$). In many of these cases the student had just written a copy of the method body (presumably unknowing) at the location where the method call should have been, and thus met the requirements of the assignment. This may be due to an underlying problem understanding the provided API. In other cases the program did not perform the task of the method, and did not meet the requirements.

There was also other problems related to method calls, though quite rare. A few students struggled with correctly specifying method calls, where both syntax and choosing which expressions to pass as arguments were problematic. Other cases were more subtle and did not affect the solution in any way, e.g. repeated calls to observer methods.

4 Grading by Annotation

In this section, we'll sketch a grading-by-annotation scheme, which may be used as a tool for teaching assistants, and which should make it easier for a teacher to identify particularly common problems faced by students.

In the normal grading scheme for our INF100 course, feedback to the students depends on the kind of assignment, with only pass/fail and short comments for weekly exercises. For semester assignments the teaching assistants fill out a form where they for each of the requirements of the assignment give a score and possibly a comment. The detail of the feedback for exercise assignments is quite low, and it may require a lot of text to describe what exactly is wrong and where the problem is located. For semester assignments it may be a lot more specific regarding location, depending on how the requirement is formulated and how specific it is.

With our scheme the teaching assistants annotate the exact locations of the problems in the submitted solutions, with annotations that describes the problems. This way the students will have an easier task of connecting the feedback with the problems, and it makes it significantly easier for the teaching assistants to give feedback to the students, and especially regarding lesser problems that with a large workload might be ignored. This is more akin to traditional paper-based grading, which we used earlier – but reviewing code on paper has the problem that it is impossible to test the submitted solution properly. An important feature of this scheme is that the graded solutions can be scanned for annotation occurrences, and a database can be built from the results – enabling future research, and targeted teaching.

Syntax

The syntax we have chosen for our annotation scheme supports a comma separated list of annotations and an optional comment for each location. In addition it is possible to include the optional brace parentheses to mark segments of code that the annotation describes. The code below includes all the possible syntax elements, notice the brace closure at line 3. The two optional elements do not depend on each other, and one or the other, both or none can be left out.

```
1  ///[<anno-id>, ...](<comment>){  
2  <code segment>  
3  ///}
```

Below we have given a list of two annotation identifiers, and have not included any of the optional elements. The bare minimum of an annotation is a list with one annotation identifier. Notice that both normal and brace parentheses are not present.

```
1  ///[fail, not_implemented]  
2  class Recipe {  
3      ...  
4  }
```

Grading

We have three types of annotations in our scheme, *file annotations*, *problem annotations* and *meta annotations*. A combination of these three should be used when grading and providing feedback to the students. In addition we suggest that the teaching assistants provide comments, using the syntax of our scheme, whenever it is necessary in order to provide the best possible feedback.

File annotations describes files and their contents, with the primary purpose of stating if the file contains adequately good implementations to pass or not. *File meta annotations* provide more details for a given annotation (e.g., the student failed because the file was not implemented).

Meta annotations are used to further describe the nature of a problem, and must always be connected with a *problem annotation* when used. For example, to state that a conditional expression is incorrect because the wrong logical connective was used, we would use the annotations *incorrect condition* together with *incorrect logic*.

Problem annotations mark occurrences of mistakes, possible misconceptions, uncovered requirements etc. We have created a list of annotations based on our study and previous experience. They are categorised into the following categories, based on what they relate to: *variables*, *scopes*, *conditions* and *control structures*.

Collecting the data

One of the most important reasons to choose this grading scheme is to be able to study what problems the students have, how frequent they are, correlations between the problems etc. To do this the graded solutions must be scanned for occurrences of annotations, and the results gathered in a database. In the study we performed in this paper we saved additional data with the annotation identifier:

- Assignment identifier.
- Student identifier (anonymous if necessary).
- Filename (requires that all submitted solutions have the exact same name for all the files, which was the case for us).
- Line number.

This level of detail allowed us to analyse the data in many interesting ways, and there are many more possibilities.

To be able to collect all these details we kept a folder structure, with one folder for each assignment that contained one folder for each student. Then we developed a simple Bash script that scanned the files for annotations, and through the folder structure created SQL insert statements with all the details listed above.

5 Related Work

Lathinen et al. [4] analyses a survey of students and teachers opinions of which elements they consider difficult when learning programming. Parts of this survey is strongly related to what we look at in this thesis, though we have chosen other methods of gathering data and we have a more narrow focus. Robins et al. [5] reviews much research regarding learning and teaching of programming and present a discussion of the findings. Spohrer and Soloway [8] present an empirical study of the general belief of novice mistakes.

Johnson et al. [3] developed a bug classification scheme that relate bugs to program constructions and underlying misconceptions and Spohrer et al. [7] created a problem-dependent classification scheme based on Johnsons scheme. Our grading scheme (Section 4) and taxonomy (Section 2), has similarities with both classification schemes, it relates problematic code segments both to the problem description (requirements) and the language constructs.

6 Conclusion and Future Work

When we started our study, we had several questions in mind:

1. Are some problems more common than others?
2. Is parameter passing and references as common as our experience suggested?
3. How does problems relate to misconceptions and missing knowledge?

We have found that some problems are more common than others, and that the two most common problems are related to conditional expressions, ζ_{COND}^{BDRY} and ζ_{COND}^{BAD} . These are also most prone to repetition. In general, the most common problems accounted for more than half of the problem occurrences, both in absolute terms and in terms of the number of students with those problems.

Problems with parameter passing and references are not as common as we expected, at least not for the assignments that we inspected in this study. Though we do suspect that it may be because the assignments are not particularly challenging regarding these subjects (the next course, INF101, becomes more challenging), and that it would be a more common problem if the assignments had given this more attention.

To relate problems to misconceptions and missing knowledge we used the taxonomy we described in Section 2. The category *concept-based problems* cover problems that are related to misconceptions, and the category *knowledge-based problems* cover those related to missing knowledge. Classifying the most common problems into concept-based (ζ_{COND}^{BDRY} , ζ_{COND}^{BAD} , ζ_{CALL}^{NO} , ζ_{GRPNG}^{BAD} , ζ_{ACCUM}^{BOOL} , ζ_{ARIT}^{BAD}) and knowledge-based (ζ_{COND}^{BAD} , ζ_{PROB}^{LOOP} , ζ_{CTRLS}^{CMLX} , ζ_{CALL}^{NO} , ζ_{ARIT}^{BAD}), we found slightly more students with concept-based problems than knowledge-based problems.

Based on the results that we have presented it is not possible to conclude that one of the categories is more important than the other, though that is not our intention either. What we can conclude is that the *concept-based problem* category is an important one, and that improving the environment that the student has available to discover and learn about these mistakes likely will yield better results. For example, an IDE like Eclipse or NetBeans will make it a lot easier to navigate both the standard library and the classes provided as part of the exercise. Another significant improvement would be to supply the students with enough tests and test data, so that they may eliminate more of the errors themselves.

Having a better understanding of what makes programming difficult, could also allow us to make educational tools that would give the students better feedback, and help improve their mental models of programming.

The grading-by-annotation scheme sketched in Section 4 should make it easier to collect data from assignments, and it's being applied in the 2013 iteration of the course. But studying handed-in assignments has its limits when it comes to uncovering problems and thought processes. It is likely that many problems have been solved, either individually or with the help of teaching assistants before the assignment is handed in. A future study should perhaps also consider observing and interviewing students as they attempt to solve problems.

Studying and classifying problems has an immediate value in that one can target the teaching to address such problems. In particular, if data is collected from the assignments every week, the teacher can immediately address common problems in the next lecture.

There are also open questions for future research: If we use targeted teaching to resolve common problems, will we uncover a new list of common problems that have yet to be addressed? If different students have different problems, can we identify a few large subgroups that share common problems, so that we could provide tailored teaching to each group?

References

- [1] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.
- [2] B. du Boulay. Some difficulties of learning to program. In E. Soloway and J. Spohrer, editors, *Studying the novice programmer*, pages 283–299, 1989.
- [3] W. L. Johnson, E. Soloway, B. Cutler, and S. Draper. Bug catalogue: I. Technical Report 286, Department of Computer Science, Yale University, New Haven, CT, 1983.
- [4] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen. A study of the difficulties of novice programmers. *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education - ITiCSE '05*, page 14, 2005. ISBN 1595930248.
- [5] A. Robins, J. Rountree, and N. Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, pages 37–41, April 2003.
- [6] A. H. Rosbach. Novice difficulties with language constructs. Master's thesis, Dept. of Informatics, University of Bergen, August 2013. URL <https://bora.uib.no/handle/1956/7167>.
- [7] J. Spohrer, E. Soloway, and E. Pope. A goal/plan analysis of buggy Pascal programs. In E. Soloway and J. Spohrer, editors, *Studying the novice programmer*, pages 355–399, 1989.
- [8] J. C. Spohrer and E. Soloway. Novice mistakes: are the folk wisdoms correct? *Communications of the ACM*, 29(7):624–632, July 1986. ISSN 00010782.