

THE UNDERSTANDING AND MODIFICATION OF PROCEDURAL AND OBJECT-
ORIENTED PROGRAMS – WHEN DOES KNOWLEDGE HELP?

BY

Thomas D. LaToza

HONORS THESIS
for the Degree of
Bachelor of Science in Psychology
College of Liberal Arts and Sciences
University of Illinois at Urbana-Champaign

2004

THE UNDERSTANDING AND MODIFICATION OF PROCEDURAL AND OBJECT-
ORIENTED PROGRAMS – WHEN DOES KNOWLEDGE HELP?

BY

Thomas D. LaToza

HONORS THESIS
for the Degree of
Bachelor of Science in Psychology
College of Liberal Arts and Sciences
University of Illinois at Urbana-Champaign

2004

Acknowledgements

First and foremost I would like to thank my advisor, Alex Kirlik. As this project has traveled through numerous literatures and tried to find a balance between cognitive psychology and software engineering, he has really helped to get to the bottom of what matters and is important. He has always had something useful to provide despite not having a background in software engineering or the psychology of programming, and his perspectives have shaped this project. I would like to thank Ralph Johnson for providing advice and suggestions and for inspiring me to be interested in software design and the topic of this thesis. I would like to thank my roommates for helping me practice presentations, proof materials, figure out statistics, and whatever else I needed help with at a moments notice. I would like to thank my all of my friends who were willing to be participants, especially the two willing to be unpaid pilot participants. I would like to thank Susan Garnsey for teaching me what a research life is like and for always having something kind, understanding, and constructive to say no matter what the situation. I would like to Colleen Conley for having a handout for every question and all of the psych honors students for being good comrades. And finally I would like to thank my parents for always being supportive of everything I do.

TABLE OF CONTENTS

Abstract	vi
Introduction.	1
<i>The function of code</i>	2
<i>The structure of code</i>	5
<i>Object-Oriented vs. procedural code</i>	7
<i>Evidence for domain / code mapping</i>	9
<i>Questions and hypotheses.</i>	10
Method	11
<i>Design</i>	11
<i>Materials</i>	12
<i>Participants</i>	13
<i>Procedure</i>	14
Results	16
<i>Bowling knowledge manipulation</i>	16
<i>Main effect of code isomorph</i>	16
<i>Code understanding</i>	17
<i>Modification time</i>	17
<i>Bug creation</i>	17
Discussion	17
<i>OO and code length</i>	19
<i>Task limitations</i>	19
<i>Terminology</i>	20
<i>Conclusion</i>	20
References	22
Tables	25
Table 1	25
Table 2	26
Appendices	27
Appendix A	27

Appendix B	29
Appendix C	36
Appendix D	37
Appendix E	38
Appendix F	42

Abstract

Software bugs are an ever-increasing societal problem as computers become more prevalent in our homes and workplaces and programs grow in size and complexity. In response, many in the software engineering community have advocated a paradigm shift in the way programs are written: away from procedural or plan-like code (e.g., Pascal, Fortran) and toward more declarative and map-like Object-Oriented (OO) code (e.g., C++, Java). A central claim is that declarative, map-like code allows developers to better deploy their prior knowledge of the problem domain (what the code is ‘about’) than does procedural code, which typically appears as one succinct and specialized plan for solving a particular problem (a ‘recipe’ or ‘directions to a destination’ rather than map of the problem domain).

To evaluate this claim, we conducted an experiment presenting participants with procedural and OO code isomorphs of the same abstract algorithm, and crossed this manipulation with whether participants were primed on the concrete problem solved by the algorithm (scoring ten-pin bowling). In direct contrast to the claims of the OO community, our findings instead revealed that priming background knowledge relevant to the problem domain actually helped those presented with procedural, rather than OO, code. In their first programming task (code modification), the procedural group whose bowling scoring knowledge was primed created significantly fewer bugs, while priming the OO group led to, if anything, even slightly more bugs. We interpret this finding to suggest that, when working with abstractly represented problems, familiarity with the problem domain is more beneficial for those modifying highly specialized and compact solutions (e.g. detailed directions to a destination) than for those modifying more descriptive, declarative solutions (e.g. a map to a destination).

The understanding and modification of procedural and Object-Oriented programs – when does knowledge help?

Imagine that you are trying to direct your friend to a place to meet you where he has never been. You have two choices: you could give him the route traced out on a map or just a list of turns to make. What factors might influence your decision?

A map might help your friend more if he makes a wrong turn or would like to find an alternative route to avoid traffic. The map describes the surroundings and potentially provides information about distance. But a list of turns might take up less space on a page and provide a more succinct description of the actual route to be taken. While preoccupied with driving, it might be easier to just glance at a list to find the next turn than to look for one's current location on the map and trace out the next turn.

Another factor that might influence your decision is how well your friend knows the surrounding area. If you decided to give your friend a map, knowing the surrounding area might help a little, but likely not a lot. Even if your friend knew the area well, he would still have to trace through the map's path to find the route to the destination. Or he might trace the entire path out on the map and then remember only the list of turns. If you decided to give your friend directions and he didn't know the area, he might easily get lost after making a wrong turn. If he already knew the area, directions might be a more convenient way to remember where to go to. You might decide to be safe and give your friend a map if you were afraid he might get lost and just give him directions otherwise.

Software developers face similar decisions every day. When building a program, a developer must choose between different representations that describe the same system but make some subsequent tasks easier or harder. One particularly important decision is directly

analogous to the directions / map distinction. Like a map, Object-Oriented (OO) code provides a survey of the domain by structuring the solution to a problem (finding a location) in terms of the domain (the city). Like directions, procedural code directly and succinctly describes a plan (list of turns) to accomplish a goal. While most programs are a combination of both and the distinction is idealized, a developer must still choose whether particular functionality will be represented in a procedural or OO fashion. A frequent claim of OO proponents is that it is a better representation for those with domain knowledge in that its structure better facilitates deploying domain knowledge to understand the solution. This thesis first considers how grouping program structure affects developer performance and then experimentally tests this claim.

The function of code

Code is computationally precise and efficient representation of a system for a human developer and a compiler program. Code is a text written in a programming language that a developer reads and writes to describe a system. A compiler then translates code into a program to be used by a user. Normative models of programs, like normative models in general, are ultimately about cost and benefit – what is the cost of the program, and what benefits does it provide to its user? Programs provide value to a customer by satisfying requirements. A requirement is a particular feature, usually described in the terminology of the domain, that the customer would like the program to provide. A bowling lane owner might have a requirement that given the number of pins knocked down on each roll, he would like to know the score of the game. This is a feature and requirement of a desired system stated in terms of the domain, bowling. This system takes a list of bowling rolls and outputs the score of the game. A developer then writes code that implements this system which the compiler transforms into a program for the bowling

lane owner. The bowling lane owner can then place displays on each bowling lane depicting each bowler's score. This provides value to the bowling lane owner by increasing the number of bowlers interested in paying to use the bowling lane.

The utility of code can be divided into two parts – the utility of the corresponding program to the customer and the modification cost for a developer to change the code to provide marginal utility to the customer. In essence, both the system the code represents and the representation of the system have value. The system directly has some utility for the customer. A representation has indirect utility by reducing the cost of adding new features that have marginal utility. Modern models of software construction (e.g. Extreme Programming – Beck, 1999) view the process of building software as an incremental process in which features are added to a program one at a time. The customer of a system may choose to deploy the program after any number of features have been completed, but would then like to be able to add features in the future.

One consequence of code's large expressive power are isomorphs, pieces of code that all describe the same system. Two pieces of code may or may not be isomorphic depending on the user's requirements of the system. While code isomorphs of the same system provide the same value to the customer, they may have vastly different costs for a developer to add a particular feature. While writing code, a developer must choose between these potential isomorphs by making design decisions. A developer would like to make decisions that result in code that meets all of the requirements and that makes other features cheap to write.

What must a developer do to add a feature to a program? Before the system can be changed to add a new feature, relevant portions of the system and the corresponding code must first be understood. A developer builds a mental model of the system and code by forming

hypotheses and seeking evidence in the code to support or reject these hypotheses (e.g. Vans, Mayrhauser, & Somlo, 1999). This requires first reading the relevant code and then being able to understand the meaning of the code. Then the developer must make the actual changes to the code by finding the relevant code and modifying the text. Adding a feature to a program thus involves a developer first understanding code and then modifying code.

From the customer's perspective, having a developer modify code to add a feature has two costs. The developer must be paid commensurate with the length of the task. Yet, the resulting program may not meet all of the requirements and changes could result in the program no longer satisfying requirements that it used to. Adding support for a second bowler might result in the program no longer producing the correct score. This happens because developers make mistakes while understanding and modifying code (Ko & Myers, 2003). Developers may have incorrect or insufficient knowledge about the code. They may not correctly allocate their limited attentional resources to the correct sections of the code. And they may not correctly foresee how a particular piece of code translates into the actual system. These mistakes lead to breakdowns, where the developer's mental model of the code is different from the actual code. Breakdowns may then result in a developer making an incorrect change, or bug. If the developer and compiler both fail to detect this incorrect change, the program delivered to the customer then fails to meet all the requirements. A bug might result in the program crashing or might result in something more subtle, such as not producing the correct information, such as occasionally producing an incorrect bowling score. Detecting and correcting bugs requires a significant amount of developer time, increasing the customer's costs.

The difficulty understanding and changing code is often called the complexity of the code (Brooks, 1987). Part of this complexity comes from the requirements themselves – any

isomorph that correctly represents the system must contain this essential complexity. But one isomorph may be more complicated than other isomorphs, and this additional complexity is not essential but rather accidental in that it does not inhere in the program requirements themselves, but instead, results from the choice of representation. One goal of software engineering is thus to minimize this added, accidental, and unnecessary complexity.

In summary, particular code isomorphs of a system may be more or less complicated for a developer to use. Less complicated isomorphs are valuable to the customer in reducing the time to add a new feature and the chance that it will be incorrect and contain bugs. This reduces the cost that the customer must pay for the program and increases the likelihood that the program satisfies its requirements.

The structure of code

Modern programs have hundreds of thousands to tens of millions of lines of code – Microsoft Windows XP is about 50 million lines (Salkever, 2003). At 50 lines per page and 1000 pages per book, 50 million lines is equivalent to 1000 books. Without sufficient structure and organization, such programs become a Big Ball of Mud (Foote & Yoder, 1997) and may be nearly impossible to understand and modify. Thus, a central problem for software engineering has been to devise programming language constructs and norms for their use which structure code to make it easier to understand and change. Code can be viewed as a collection of hierarchically grouped statements. A statement is a programming language construct but is often the same as a line of code. Grouping constructs (e.g. procedures, classes, modules, packages, namespaces, aspects) decompose code by organizing it into groups. As developers modify code to add new functionality, they are constantly faced with grouping decisions about which functionality to group together and which to keep separate.

Like a library's Dewey decimal system and organizational schemes in general, code grouping norms state that functionality that is functionally similar, in that it is used together, should be grouped together. Groups determine the spatial locality of code - what code appears together on the same screen or otherwise spatially close in the same file that a developer is looking at. As programs become larger, this becomes increasingly important. Developers, being human, do not actually read 50 million lines of code. A central task a developer faces is being able to find and view the code relevant to the task at hand without being swamped by irrelevant code, while having confidence that other code is not also relevant.

Grouped code has a number of distinct consequences for the cost of understanding and modifying code. When a developer is testing a hypothesis or is trying to build a mental model of some functionality, grouped code is more likely to be examined at the same time by the developer. Thus, a developer must look at less code if all of the code for a particular feature of interest is all in the same place. The same is true when the developer wishes to modify code. Less code must be searched through and understood if all of the code that must be changed or might possibly be affected by a change is spatially local. If code that should always be the same is duplicated in two different places, one copy of the code may be changed and the other missed. This results in a bug. Groups also help to insulate code in the group from code outside the group. This decreases the chance that a particular change will then result in other dependent code elsewhere also needing to be changed. Finally, a feature completely implemented in a single group may be able to be reused when writing a new feature, making new features faster and shorter to write.

A recent emphasis in the software engineering literature has been that useful groups change over time as new features are required. As new functionality and code are added,

developers may need to change grouping decisions that they have made in the past to keep their code cheap to understand and modify. The tasks developer perform over time may change, resulting in locality benefits for different groups. Changing grouping decisions results in a refactoring (Fowler, 2000) which transforms one isomorph into another isomorph.

Object-Oriented vs. procedural code

As programming languages providing object-oriented constructs (e.g. C++, Java, C#) have eclipsed older procedural languages (e.g. Fortran, C) in the mainstream, the tradeoffs between object-oriented and procedural code have been frequently considered. The central grouping construct in procedural languages is a procedure. A procedure describes a plan, algorithm, or function that takes some information as an input and produces other information as an output. A “SIN” procedure would take an angle and output the Sin of the angle. Object-oriented languages still have procedures but also introduce a new central grouping construct – the class. A class groups features and procedures together to form a type of entity. Individual instances of the class are objects. A bird object might have blue feathers, a nest, and might have a procedure that chirps.

While OO has become the dominant paradigm and OO books abound, there has been little consensus and clarity into exactly why OO is beneficial. In particular, two camps have emerged. The Scandinavian view (Johnson, Balaguer, & Wei, 2004) is that OO code is more natural because people think in objects. Rather than model the world as sequential plans, this view argues that people naturally model the world using objects. Objects, as entities rather than plans, can be anthropomorphized. Objects, akin to agents, can be viewed as committing to a responsibility and then collaborating with other objects to carry it out (Wirfs-Brock & McKean,

2003). Procedures then become descriptions of how an object interacts with its collaborators to carry out its responsibilities.

Bolstering the claims that objects are natural is their similarity to a popular knowledge representation model for human memory – categories (e.g. Medin, 1989). Both categories and objects have a type / token distinction. Categories are a type and individual instances of the type are exemplars. Classes are a type, and individual instances of a class are objects. A robin sitting on the tree outside is an instance of the class or category of bird. Both also have inheritance and compositional relationships. A bird is subtype of the more general category of animal. A bird object is a composite of parts such as feet, beak, legs, and wings. Both use these hierarchical relationships to minimize the amount of information stored in a particular type. Asked if a bird has a heart, its supertype, animal, would be consulted to answer yes. Yet it is not clear that this similarity is important. Code is an external representation and categories are a representation for memory. Each serves a very different purpose.

A second view is that OO code is better than procedural code simply by providing an additional grouping construct in the programming language. Parnas (1972) proposed that a module, a type of group, exists to hide a difficult or likely-to-change design decision. This makes the design decision easier to change in the future by keeping any changes within the module. Classes are a programming language construct to implement groups and bring with them mechanisms for hiding some information inside the class. This promotes modularity by allow the hidden information to change without requiring other classes to also be changed.

Thus, there are two distinct stories for why OO code has a greater utility than procedural code. According to the Scandinavian view, grouping decisions should always be made to help the developer better understand the system. According to the modularity view, grouping

decisions should be made to minimize the cost of changing code. While these are not necessarily mutually exclusive, extreme emphasis on one or the other may result in very different choices for what objects and classes are created by the OO developer.

Evidence for domain / code mapping

A more specific claim about the naturalness of OO code is that it allows the developer to understand the code in terms of the domain (Rosson & Alpert, 1990; Evans, 2003). To understand the intent of code, a developer must be able to trace a particular piece of code back to the requirements and features that generated it. The same piece of code may or may not be correct depending on what requirements it implements. The mapping view of OO's benefits claims that OO code allows developers to better use and deploy their domain knowledge for these understanding tasks.

A number of studies have provided evidence that developers working with OO code make extensive use of the domain when understanding code. This has often been viewed as a distinction between top down and bottom up. Top down understanding involves reading code to test specific hypotheses and then iteratively generating more specific hypotheses. The domain provides reasonable hypotheses about the code. Bottom up understanding involves reading contiguous code and then slowly building a more abstract model of the code. Given novel code to read, developers initially spend more time testing hypotheses on OO code than procedural code (Corritore & Wiedenbeck, 2001). Developers reading OO code initially have more domain knowledge and less specific knowledge about code than the more balanced developers reading procedural code, but OO developers became more balanced over the course of a modification task (Corritore & Wiedenbeck, 1999). Developers using OO code may also spend more time

considering and refining their model of the domain than developers working with procedural code as they attempt to implement a corresponding version in code (Rosson & Gold, 1989).

Yet, making use of the domain may not always be beneficial. Using objects that are natural to describe the domain as grouping constructs for code may make writing the code more difficult. In one study (Kim, Lerch, & Simon, 1995), participants were first given a problem statement describing one of two isomorphs to the Tower of Hanoi problem. Objects in each isomorph had slightly different rules about how their operations worked. Participants were then asked to write an OO program to solve the Tower of Hanoi problem. Participants initially used their model of the domain to build their solutions. Yet, participants in the harder isomorph condition were unable to write the code to solve the problem and were forced to change their model to the other isomorph condition.

Questions and hypotheses

The central claim of the mapping view of OO's naturalness is that developers should perform better with OO code than procedural code because OO code allows them to make better use of their domain knowledge. Yet, directly comparing developer performance on OO and procedural code is fraught with peril. Confounded with differences in the ability to deploy domain knowledge are a host of other proposed and possible costs and benefits of a particular grouping of code. Direct comparison of OO and procedural performance makes it difficult to claim that domain knowledge is truly the cause of any benefits and to distinguish the mapping view from a modularity view of OO's benefits.

A more testable implication of the mapping view is that domain knowledge should benefit OO performance more than procedural performance. If OO code allows a developer to make better use of domain knowledge, providing domain knowledge should benefit developers

using OO code more than developers using procedural code. Rather than qualitatively interpret developer behavior, it is now possible to directly ask: does domain knowledge provide greater benefit for OO code than procedural code for (1) understanding code, (2) time to modify code, or (3) bugs created modifying code.

Method

Design

The design was a factorial, two (bowling or no bowling domain knowledge primed) by two (OO or procedural code isomorph) between subjects design. Participants were randomly assigned to one of the four conditions. All participants were presented with a description of a fictitious algorithm called the Simon function. Participants in the bowling knowledge condition were then primed with the information that the Simon function can be used to score bowling. Participants in the no priming condition did not receive this information. As all participants had already been screened for bowling knowledge, this manipulation was designed to prime participants in the bowling knowledge condition to deploy their existing domain knowledge to complete the following tasks, not to teach bowling. Rather than manipulate the presence of domain knowledge, only the perceived relevance of the bowling domain knowledge was manipulated. To ensure that participants did not spontaneously realize that the Simon function algorithm or code isomorphs were really the rules to score bowling, the Simon function algorithm and code isomorphs were changed such that the maximum number of pins that could be knocked down in a frame was 17 instead of 10. Additionally, in both code isomorphs, all comments, variables, and functions named in terms of bowling were renamed to use Simon function terminology. For example, Strike, Spare, and Open frames became HighIteration, MediumIteration, and LowIteration, respectively. For a selected list of renamed terminology, see Table 1. The code

isomorphs are shown in Appendices A and B. The success of this manipulation was measured by how many participants reported realizing that the Simon function could be used to score bowling on a questionnaire at the conclusion of the experiment.

Participants performance with each code isomorph was assessed with measures from three tasks. Participants initial understanding of the code was assessed by a quiz about how the code scores bowling. Participants in both code isomorph groups were given the same ten question quiz, shown in Appendix E. Correct answers for all but one question were the same for both groups. Both the responses and the total time for the quiz were recorded. In the second and third task, participants then modified the code to add two features. The time for each task was recorded. To measure the correctness of the changes, automated tests (unit tests) were run on the modified code. The procedural unit tests are shown in Appendix E and the OO unit tests in Appendix F. Some of these unit tests were from modified tests from the original articles to ensure that features still worked correctly. The rest were constructed to test the required modifications. For the third task, half of the bug score came from being judges as being an incremental implementation by saving and reusing a score after every invocation. To assess the amount of code required for the changes, the net increase in the number of statements in the program was recorded. This was measured by the net increase in statement ending semicolons, not including semicolons in for loop conditionals.

Materials

Rather than construct artificial examples of object-oriented and procedural code, an object-oriented (Jeffries, 2003a) and a procedural (Jeffries, 2003b) code isomorph were used from articles extensively discussed by expert software developers on a software engineering mailing list (Jeffries, 2003c). Both isomorphs score a game of American 10-pins bowling. The author's

purpose was to show that choosing class responsibilities from the domain before writing code leads to a much longer object-oriented solution and prevents discovery of the shorter procedural solution.

To adapt these isomorphs for this experiment, two changes were made to both, and one change was made to the OO code. Both were originally written in C# and were translated to Java to increase the number of potential participants familiar with the programming language. The original two implementations from the articles were not completely isomorphic. The OO code had methods which added data a frame at a time whereas the procedural implementation added data a roll at a time. To make the implementations completely isomorphic, a new method, `BuildIterations`, was added to the OO code. This method takes a list of rolls and builds the corresponding frame objects. The third change was to replace all the bowling terminology in both implementations with Simon function terminology.

Participants

Sixteen University of Illinois computer science, computer engineering, and electrical and computer engineering majors were recruited by email, a newsgroup posting, and fliers for an approximately one hour session. Participants were first sent a screening survey by email which gathered information about their programming experience. Participants ranged from juniors to third year graduate student and reported being proficient in from two to nine programming languages with a mean of 4.6. Participants reported having three to eighteen years programming experience with a mean of 8.5. Participants reported spending an average of 15.9 hours per week programming over the past year with a range of .5 to 40 and reported spending an average of 13.2 hours per week programming over the past five years with a range of 2 to 40. Participants who did not report having used Java for at least one project were excluded from the experiment.

Additionally, participants were asked to rate their level of familiarity with the rules of six different common sports including bowling on a five point scale. To ensure the success of the bowling knowledge manipulation, all participants who did not report that they at least “Know most of the rules” to bowling, the middle response, were excluded from the experiment.

All participants received \$8 for their participation and were informed that they would be competing with up to five other participants for a \$50 prize. Four prizes were awarded, one for each condition. Participants were instructed that they would receive points for three different measures on each of the two modification tasks. The rules were designed to increase motivation and establish specific goals for the participants in performing their tasks. Participants were encouraged to use their time efficiently by receiving one point for every unused minute of their allotted modification task times. Participants were encouraged to correctly complete the task by receiving three points for every passing unit test. Passing a unit test corresponds to correctly making some portion of the requested modification. Participants were encouraged to think deeply about what the given code was doing by being penalized one point for every statement added. Shorter solutions were possible if the participants make effective use of the existing code and wrote less new code.

Procedure

All participants first read a document describing an abstract mathematical function called the Simon function, shown in Appendix A. Participants in the bowling condition were then primed to view the Simon function as bowling by reading a short text that explained how the Simon function, suitably parameterized, can be interpreted as essentially identical to the algorithm for scoring ten pins bowling. This text is shown in Appendix D.

All participants then completed a table to calculate the value of an 11-valued vector input to the Simon function, resulting in five iterations of the Simon function. Performing this computation is equivalent to scoring the first five frames of a game of bowling, consisting of 11 balls rolled. Completing this table required understanding the equivalent rules for both “strikes” and “spares”. Next, participants read the instructions for the code comprehension and modification tasks. Any errors made by were then pointed out to the participant, and participants were asked to correct any mistakes that were not simple arithmetic mistakes. The number of times required to perform this task to criterion was recorded.

All participants then began the code comprehension phase. Participants were instructed that they would first have ten minutes to read the code and would then take a short quiz. Participants in the OO condition were given the OO isomorph and participants in the procedural condition were given the procedural isomorph. In order to minimize the effects of individual differences in familiarity and expertise with editors, the code was presented on a computer using a simple text editor, Notepad. All participants were encouraged to modify variable and function identifiers and add any comments to help them better understand the code. Participants were stopped after ten minutes, but others finished examining the code earlier. The time that all participants spent looking at the code was recorded. The text editor was then closed, any papers removed, and the participant given the ten question quiz.

All participants next began the code modification tasks. Participants were instructed that they would have 15 minutes to work on each of two modification tasks. Notepad was then reopened with the appropriate code isomorph. Participants were shown how to compile the program from the command line to ensure that it was syntactically correct. To control for differences in debugging strategy and expertise, participants were not allowed to run or debug

the program. The first modification task asked participants to write code to perform error checking on the input vector. The second task asked participants to implement incremental scoring. Both code isomorphs correctly produced the score only for a complete game of bowling. Incremental scoring required the code to be modified to produce the score after any sequence of rolls. The task also asked that the computation not recalculate any frame scores previously calculated but to resume calculating the score using the total score from the frame previously completed. Participants were stopped if they did not finish after 15 minutes.

Finally, all participants were asked to rate their perceived performance on each of the modification tasks and their overall understanding of the code on a five point scale. Due to the confidence scale being in the opposite direction expected, many participants inverted their responses. Participants who responded that they had done well when they did not finish or poorly when they did finish had all of their confidence responses coded as flips. Flips were confirmed by subjective impressions during the debriefing. All members of the no-bowling condition were specifically asked if they spontaneously made any connection between the Simon function and bowling.

Results

Bowling knowledge manipulation

Confirming the success of the bowling knowledge manipulation, all participants in the no bowling knowledge condition responded that they did not realize that the Simon function could be used to score bowling. Moreover, many participants expressed noticeable surprise upon making this realization.

Main effect of code isomorph

Participants with the OO code performed worse than participants with the procedural code on most of the measures. Participants spent significantly longer looking at the code before the quiz, $F(1, 12) = 5.59$, $p < .05$, and made significantly more mistakes on the quiz, $F(1, 12) = 20.25$, $p < .01$. Participants reported less confidence in their understanding of OO code with a difference that approached significance, $F(1, 12) = 4.24$, $p < .1$, and reported less confidence in their second modification task performance with a difference that approached significance, $F(1, 12) = 4.45$, $p < .1$. Participants with OO code wrote significantly more statements on the first task, $F(1, 12) = 4.94$, $p < .05$. The remaining main effects of code isomorph were not significant.

Code understanding

None of the interaction effects for time spent looking at code, number of mistakes on the quiz, time spent on the quiz, or confidence understanding the code were significant.

Modification time

Modification time differences for both modification tasks failed to reach significance.

Bug creation

The interaction effect for modification task 1 bugs failed to reach significance, $F(1, 12) = 2.99$, NS. But considered separately, bowling knowledge significantly decreased bugs written using procedural code from a mean of 3.5 bugs to 0.3 bugs, $t(6) = 2.67$, $p < .05$. Bowling knowledge increased the number of bugs written using OO code from 1 bug to 2.8 bugs, but the difference failed to reach significance, $t(6) = .67$, NS. A table of means of all measures for all conditions can be found in Table 2.

Discussion

The central claim of the mapping view of OO's benefits is that OO code is more natural than procedural code by enhancing the developer's ability to use his or her domain knowledge in

understanding and modifying code. Thus, domain knowledge should benefit developers using OO code more than developers using procedural code. The present experimental results do not support this claim. Moreover, the one relevant measure showing a significant benefit associated with priming domain knowledge, the number of modification task 1 bugs introduced, actually showed that those using procedural, rather than OO code benefited by this manipulation. Participants using OO code did not appear to benefit by having domain knowledge primed, although this result should be viewed cautiously due to the low number of participants studied. It would be hasty to conclude from this study that domain knowledge does not benefit OO code developers.

One interpretation of these findings is that developers benefit more from domain knowledge when the solution they are interacting with is a succinct, procedural plan (e.g., driving directions to a destination) as opposed to than those interacting with a more declarative problem solution (directions sketched on a map of a town). This interpretation makes sense in that procedural solutions tend to be fragile and do not necessarily provide resources for how they should or could be amended to account for unforeseen situations.

Only the chef with good cooking domain knowledge is able to effectively substitute novel, available ingredients for unavailable ones when following a recipe (a procedural problem solution), and only the driver with good domain (spatial) knowledge of a town is able to effectively detour around road construction while following list-like, procedural directions to a destination. In contrast, those faced with the task of modifying problem solutions that are represented in a more declarative, map-like form, may benefit less from domain knowledge because the external representation itself can often provide at least some of this information. If a driver following map-based directions to a destination had to detour around road construction, he

or she may be able to do so reasonably well regardless of whether he or she had familiarity with the town. The map, as an external source of domain knowledge, may compensate for a lack of internal domain knowledge. We suspect that only for our procedural code, rather than our OO code, did participants benefit from domain knowledge for similar reasons.

OO and code length

The numerous significant main effects of code isomorph clearly indicate that participants were less successful using OO code than procedural code. However, this does not necessarily indicate that the OO code is actually better than the procedural code. One particularly glaring confound is code length. The OO code was 191 lines long while the procedural code was a mere 38 lines.

Thus, it is not at all surprising that participants spent a longer amount of time looking at the OO code than the procedural code and that they did worse on the quiz. Participants reading OO code simply had more text to read and comprehend. Yet, while other studies comparing procedural and OO code have been plagued by similar problems (e.g. Boehm-Davis, Holt, & Schultz), it is not the case that OO code is necessarily longer than procedural code for the more common larger programs. Although overall code length would have to be significantly increased beyond the lengths used in the present experiment, this fact does open doors for future experimentation in which the code length confound could be eliminated.

Task limitations

One of the goals of the second and more ambitious modification task was to find a task complicated enough for OO code to show benefit over procedural code. The results indicate this was not the case. Moreover, the task was sufficiently complicated that five of the eight OO participants used all of their time. Much of the poor performance of OO participants on this task

may have arisen from not having sufficient time to complete the task with the OO code. Both code modifications tasks essentially tested performance with code that was novel to all participants. After a developer has had more time to become more familiar with code, the benefits of OO might begin to emerge. While a developer on a large program may constantly be exposed to novel code, it does not seem likely that it is typical of an actual software engineering environment for a developer to only spend 15 minutes with a piece of code.

Terminology

Another proposed benefit of OO code is that it better allows a “ubiquitous language” (Evans, 2003) to be shared between code and the domain. In this experiment, participants were forced to map from the Simon functionality terminology back to bowling to make use of the given bowling knowledge. This was necessary to provide the same code to participants whether or not they had been primed to use bowling knowledge. Thus, participants did not benefit from ubiquitous language shared between the code and domain. Being forced to map all of the terminology may have even made domain knowledge harm performance. Mapping between the two sets of terminology might have been more difficult than just determining what the code was doing directly from the code. The results suggest that any benefits of OO code in allowing domain knowledge to be better utilized may require ubiquitous language. Again, however, doors are open to further experimentation in which the ubiquitous language issue is addressed, by including conditions in which both procedural and OO code is written using the terminology of the domain knowledge.

Conclusion

We have interpreted our finding that domain knowledge benefits developers with procedural code and not with OO code and have proposed an alternative and novel explanation for code’s

naturalness. We contrasted the often fragile nature of compact, procedural problem solutions (e.g., cooking recipes; list-like driving directions) to the often more robust and informative nature of declarative problem solutions (e.g., driving directions presented in map-like form; assembly instructions including diagrams as well as, or in lieu of, a series of textual steps). When trying to amend or tweak often succinct, procedural solutions in response to unforeseen events (substituting items in a recipe, routing around a detour, improvising a replacement for a missing part during assembly), domain knowledge would seem to be of paramount importance, since the procedure provides only one potentially fragile solution and little additional information. In contrast, when performing similar tasks with a declarative problem solution such as a map, the map itself may mitigate the need for internal domain knowledge, since it provides an external representation of the problem domain that may compensate for any lack of such knowledge.

We have also discussed a number of features of the experimental design that may have unintentionally stacked the deck against the OO participants including a code-length confound, the use of code completely novel to participants, and the “whitewashing” of the code terminology to eliminate any reference to the concrete problem domain of bowling. We have, however, also discussed prospects for future research that could possibly address each of these issues, and thus may deepen our understanding of the problem solving activities involved in the extremely important and highly consequential area of software engineering.

References

- Beck, K. (1999). *Extreme programming explained: embrace change*. New York: Addison-Wesley.
- Boehm-Davis, D. A., Holt, R. W., Schultz, A. C. (1992). The role of program structure in software maintenance. *International Journal of Man-Machine Studies*, 36, 21-63.
- Brooks, F. P. (1987). No silver bullet: essence and accidents of software engineering. *IEEE Computer*, 20, 10-19.
- Corritore, C. L., & Wiedenbeck, S. (1999). Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *International Journal of Human-Computer Studies*, 50, 61-83.
- Corritore, C. L., & Wiedenbeck, S. (2001). An exploratory study of program comprehension strategies of procedural and object-oriented programmers. *International Journal of Human-Computer Studies*, 54, 1-23.
- Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. New York: Addison-Wesley.
- Foote, B., & Yoder, J. W. (1997). *Big ball of mud* (Tech. Rep. No. wucs-97-34). St. Louis: Washington University, Department of Computer Science.
- Fowler, M. (2000). *Refactoring: improving the design of existing code*. New York: Addison-Wesley.
- Jeffries, R. (2003). Adventures in C#: the bowling game. Retrieved May 1, 2004 from <http://www.xprogramming.com/xpmag/acsBowling.htm>.
- Jeffries, R. (2003). Adventures in C#: the bowling game revisited. Retrieved May 1, 2004 from <http://www.xprogramming.com/xpmag/acsBowlingProcedural.htm>.

- Jeffries, R. (2003, November 17). The bowling game -- new Article. Message posted to Extreme Programming electronic mailing list, archived at <http://groups.yahoo.com/group/extremeprogramming/message/84088>.
- Johnson, R., Balaguer, F., & Wei, J. (2004). CS497REJ Object-Oriented programming and design. Retrieved May 5, 2004 from <http://www-courses.cs.uiuc.edu/~cs497rej/notes/day01.ppt>.
- Kim, J., Lerch, F. J., & Simon, H. A. (1995). Internal representation and rule development in object-oriented design. *ACM Transactions on Computer-Human Interaction*, 2, 357-390.
- Ko, A. J. and Myers, B. A. (2003). Development and Evaluation of a Model of Programming Errors. *IEEE Symposia on Human-Centric Computing Languages and Environments*, Auckland, New Zealand, 7-14.
- Medin, D. L. (1989). Concepts and conceptual structure. *American Psychologist*, 44, 1469-1481.
- Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15, 12, 1053-1058.
- Rosson, M. B., & Alpert, S. R. (1990). The cognitive consequences of object-oriented design. *Human-Computer Interaction*, 5, 345-379.
- Rosson, M. B., & Gold, E. (1989). Problem-solution mapping in object-oriented design. *Proceedings of OOPSLA 1989*, 7 – 10.
- Salkever, A. (2003, April). For Windows, less fat means fewer bugs. *BusinessWeek Online*. Retrieved May 7, 2003, from http://www.businessweek.com/technology/content/apr2003/tc20030429_6540_tc047.htm.

Vans, A.M., Mayrhauser, A.V., & Somlo, G. (1999). Program understanding behavior during corrective maintenance of large-scale software. *International Journal of Human-Computer Studies*, 51, 31-70.

Wirfs-Brock, R., & McKean, A. (2003). *Object design: Roles, Responsibilities, and Collaborations*. Boston: Addison-Wesley.

Table 1

Selected Simon function and bowling terminology relationships

Bowling term	Simon function term
Strike	HighIteration
Spare	MediumIteration
OpenFrame	LowIteration
Score	Compute
FrameSize	IterationSize
Throws	X
Frames	Iterations
firstThrow	firstX
BonusRoll	ExtraX

Table 2

Group means for all conditions on all measures. All times are in minutes, and higher confidence numbers reflect greater confidence.

Measure	Procedural		OO	
	No bowling	Bowling	No bowling	Bowling
Code comprehension time	6.57	5.3	8.74	9.73
Quiz number correct	9	9	7	6.5
Quiz time	4.07	4.76	3.33	3.69
Code understanding confidence	8.25	7.5	6.5	6.25
Modification task 1 time	7.29	10.22	9.15	12.48
Modification task 1 bugs	3.5	0.3	1	2.8
Modification task 1 confidence	7.75	8.5	7.5	5.75
Modification task 2 time	11.9	13.6	13	13.9
Modification task 2 bugs	3	4	3.5	4.8
Modification task 2 confidence	6.8	7	4.3	4

Appendix A

Procedural code isomorph

```

import java.util.*;

public class SimonFunction
{
    public static void main(String[] args) {}

    private ArrayList X = new ArrayList();

    public void Append(int number)
    {
        X.add(new Integer(number));
    }

    public int Compute()
    {
        int xIndex = 0;
        int S = 0;
        for (int i = 0; i < 10; i++, xIndex += PopNumber(xIndex))
        {
            S += XAt(xIndex) + XAt(xIndex + 1);

            if (XAt(xIndex) == 17 || XAt(xIndex) + XAt(xIndex + 1) ==
17)

                S += XAt(xIndex + 2);
        }
        return S;
    }
}

```

```
private int PopNumber(int xIndex)
{
    if (XAt(xIndex) == 17) return 1;
    return 2;
}

private int XAt(int index)
{
    return ((Integer)X.get(index)).intValue();
}
}
```

Appendix B OO code isomorph

```
import java.util.*;

public class SimonFunction
{

    public static void main(String[] args)
    {

    }

    private ArrayList X;
    private ArrayList iterations;

    public SimonFunction()
    {
        X = new ArrayList();
        iterations = new ArrayList();
    }

    public void BuildIterations(ArrayList numbers)
    {
        int index = 0;

        while (index < numbers.size())
        {
            if (ListAt(numbers, index) == 17)
            {
                HighIteration();
            }
        }
    }
}
```

```

        index++;
    }
    else if (index + 1 >= numbers.size())
    {
        return;
    }
    else if (ListAt(numbers, index) + ListAt(numbers, index +
1) == 17)
    {
        MediumIteration(ListAt(numbers, index),
            ListAt(numbers, index + 1));
        index += 2;
    }
    else
    {
        LowIteration(ListAt(numbers, index),
            ListAt(numbers, index + 1));
        index += 2;
    }
}
}

private int ListAt(ArrayList list, int index)
{
    return ((Integer) list.get(index)).intValue();
}

public void LowIteration(int firstX, int secondX)

```

```
{
    iterations.add(new LowIteration(X, firstX, secondX));
}

public void MediumIteration(int firstX, int secondX)
{
    iterations.add(new MediumIteration(X, firstX, secondX));
}

public void HighIteration()
{
    iterations.add(new HighIteration(X));
}

public void ExtraX(int anX)
{
    iterations.add(new ExtraX(X, anX));
}

public int Compute()
{
    int total = 0;

    for (int i = 0; i < 10; i++)
    {
        total += ((Iteration) iterations.get(i)).Compute();
    }

    return total;
}
```

```
    }

abstract private class Iteration
{
    public ArrayList X;
    protected int startingX;

    public Iteration(ArrayList X)
    {
        this.X = X;
        this.startingX = X.size();
    }
    abstract public int Compute();
    abstract protected int IterationSize();

    protected int FirstBonusX()
    {
        return ((Integer) X.get(startingX + IterationSize())).intValue();
    }

    protected int SecondBonusX()
    {
        return ((Integer) X.get(startingX + IterationSize() +
1)).intValue();
    }
}

private class LowIteration extends Iteration
{
```

```
public LowIteration(ArrayList X, int firstX, int secondX)
{
    super(X);
    X.add(new Integer(firstX));
    X.add(new Integer(secondX));
}

public int Compute()
{
    return ((Integer) super.X.get(startingX)).intValue()
        + ((Integer) super.X.get(startingX + 1)).intValue();
}

protected int IterationSize()
{
    return 2;
}
}

class MediumIteration extends Iteration
{
    public MediumIteration(ArrayList X, int firstX, int secondX)
    {
        super(X);
        X.add(new Integer(firstX));
        X.add(new Integer(secondX));
    }

    public int Compute()
```

```
{
    return 17 + FirstBonusX();
}

protected int IterationSize()
{
    return 2;
}
}

private class HighIteration extends Iteration
{
    public HighIteration(ArrayList X)
    {
        super(X);
        X.add(new Integer(17));
    }

    public int Compute()
    {
        return 17 + FirstBonusX() + SecondBonusX();
    }

    protected int IterationSize()
    {
        return 1;
    }
}
```

```
class ExtraX extends Iteration
{
    public ExtraX(ArrayList X, int firstX)
    {
        super(X);
        X.add(new Integer(firstX));
    }

    public int Compute()
    {
        return 0;
    }

    protected int IterationSize()
    {
        return 0;
    }
}
```

Appendix C Simon function description

The Simon function, $S_k(\mathbf{X}) : Z^n \Rightarrow Z$, where k is a constant, is highly useful for describing processes in both physical and human systems. For example, S_{17} maps an n dimensional vector \mathbf{X} to an integer as follows:

While more numbers in \mathbf{X}

Let x_1, x_2, x_3 be the first 3 numbers on the list; x_3 may be null

Let $h \leftarrow 1$ iff $(x_1 = 17)$ or $(x_1 + x_2 = 17)$

Let $h \leftarrow 0$ otherwise

$S += x_1 + x_2 + h * x_3$

If $x_1 = 17$

Pop one from the head of the list

Else

Pop two from the head of the list

Return S

Appendix D Bowling priming

One particular use of a Simon function is to score traditional 10 pins bowling, the type played in most American bowling alleys. In bowling, multiple players roll a ball to earn points knocking down pins. \mathbf{X} is a list of the number of pins knocked down by each roll. Each player's turn is called a frame, equivalent to an iteration of the Simon function S_{10} . The maximum number of pins that can be knocked down per frame is 10, as opposed to the 17 in the Simon function example. If a player does not knock down all of the pins on the first roll of a frame, the player receives a second roll.

To score a frame of bowling, the score for all of the rolls in the frame is first summed. If all of the pins were knocked down on the first roll, the frame is called a strike, and the next two rolls are added to the score. If all of the pins were knocked down on the first two rolls, the frame is called a spare, and the next roll is added to the score. If all of the pins are not knocked down in a frame, the frame is open, and no additional points are added.

SIMON FUNCTION S_{17}	BOWLING (S_{10})
Loop iteration	Frame
17	10
$h = 1$	Strike or spare
\mathbf{X}	List of number knocked down pins
S after ith iteration	Total score after ith frame

Appendix E
Code quiz

An asterisk precedes the correct answer for each question except for question four, where the correct answer for each code isomorph is denoted separately.

Please circle the best answer. Each question is worth one point. There is no penalty for guessing.

1. How many times does the for loop in `SimonFunction.Compute()` iterate?

- a. 4
- b. 6
- *c. 10
- d. 14

2. Each iteration of the loop in `SimonFunction.Compute()`

- *a. goes to the next Simon function iteration
- b. goes to the next number in **X**
- c. goes to the next number with an odd index in **X**
- d. goes to the next number with an even index in **X**

3. What is the type of **X**?

- a. Array
- b. Vector

*c. ArrayList

d. LinkedList

4. What is the **minimum** number of values of **X** referenced by code called from each iteration of the loop in `SimonFunction.Compute()`?

a. 0

b. 1 [correct answer for OO isomorph]

c. 2 [correct answer for procedural isomorph]

d. 3

5. What is the **maximum** number of values of **X** referenced by code called from each iteration of the loop in `SimonFunction.Compute()`?

a. 1

b. 2

*c. 3

d. 4

6. For a given execution of `SimonFunction.Compute()`, how many times will it call `Append()`?

*a. 0

b. 1

c. 2

d. It varies

7. If the SimonFunction class has data for **more** iterations than 10 it will

- *a. Ignore the extra data
- b. Use the extra data and correctly calculate the Simon function
- c. Throw an exception
- d. Return a special value indicating it has extra data

8. If the SimonFunction class has data for **fewer** iterations than 10 it will

- a. Correctly compute the Simon function for the iterations it has data for
- b. Incorrectly compute the Simon function
- *c. Throw an exception
- d. It depends on exactly what data it has

9. Which of the following is the name of a member function of the SimonFunction class?

- a. Remove()
- *b. main()
- c. Prepend()
- d. Reset()

10. Which of the following is the import statement at the top?

- a. import java.math.*;
- *b. import java.util.*;

c. `import java.util.ArrayList;`

d. `import java.text.*;`

Appendix F Unit tests

Modification task 1 procedural isomorph unit tests

```
import junit.framework.TestCase;

public class SimonFunctionTest extends TestCase
{
    protected void setUp()
    {
        simonFunction = new SimonFunction();
    }

    public void RollMany(int count, int roll)
    {
        for (int i = 0; i < count; i++)
            simonFunction.Append(roll);
    }

    public void testGutterBalls()
    {
        RollMany(20, 0);

        assertEquals(0, simonFunction.Compute());
    }

    public void testThrees()
    {
        RollMany(20, 3);
    }
}
```

```
        assertEquals(60, simonFunction.Compute());
    }

    public void testSpare()
    {
        simonFunction.Append(7);
        simonFunction.Append(10);
        simonFunction.Append(5);
        simonFunction.Append(3);
        RollMany(16, 0);

        assertEquals(30, simonFunction.Compute());
    }

    public void testStrike()
    {
        simonFunction.Append(17);
        simonFunction.Append(5);
        simonFunction.Append(3);
        simonFunction.Append(2);
        simonFunction.Append(1);
        RollMany(14, 0);

        assertEquals(36, simonFunction.Compute());
    }

    public void testPerfect()
    {
        RollMany(12, 17);
    }
}
```

```
        assertEquals(510, simonFunction.Compute());
    }

    public void testXGreaterThanOrEqual()
    {
        simonFunction.Append(18);
        simonFunction.Append(7);
        simonFunction.Append(5);
        simonFunction.Append(3);
        RollMany(16, 0);

        assertEquals(-1, simonFunction.Compute());
    }

    public void testSecondXGreaterThanOrEqual()
    {
        simonFunction.Append(7);
        simonFunction.Append(18);
        simonFunction.Append(5);
        simonFunction.Append(3);
        RollMany(16, 0);

        assertEquals(-1, simonFunction.Compute());
    }

    public void testXLessThan()
    {
        simonFunction.Append(-1);
        simonFunction.Append(7);
```

```
        simonFunction.Append(5);
        simonFunction.Append(3);
        RollMany(16, 0);

        assertEquals(-1, simonFunction.Compute());
    }

    public void testSecondXLessThan()
    {
        simonFunction.Append(7);
        simonFunction.Append(-1);
        simonFunction.Append(5);
        simonFunction.Append(3);
        RollMany(16, 0);

        assertEquals(-1, simonFunction.Compute());
    }

    public void testComboGreaterThan()
    {
        simonFunction.Append(7);
        simonFunction.Append(16);
        simonFunction.Append(5);
        simonFunction.Append(3);
        RollMany(16, 0);

        assertEquals(-1, simonFunction.Compute());
    }
}
```

```
        private SimonFunction simonFunction;
    }
}
```

Modification task 2 procedural isomorph unit tests

```
import junit.framework.TestCase;

public class SimonFunctionTask2Test extends TestCase
{
    protected void setUp()
    {
        simonFunction = new SimonFunction();
    }

    public void RollMany(int count, int roll)
    {
        for (int i = 0; i < count; i++)
            simonFunction.Append(roll);
    }

    public void testIncremental1()
    {
        simonFunction.Append(7);
        simonFunction.Append(9);
        simonFunction.Append(5);
        simonFunction.Append(3);
        simonFunction.Append(14);

        assertEquals(24, simonFunction.ComputeIncremental());
    }
}
```

```
}

public void testGutterBallsIncremental()
{
    RollMany(2, 0);

    assertEquals(0, simonFunction.ComputeIncremental());
}

public void testThreesIncremental()
{
    RollMany(8, 3);

    assertEquals(24, simonFunction.ComputeIncremental());
}

public void testSpareIncremental()
{
    simonFunction.Append(7);
    simonFunction.Append(10);
    simonFunction.Append(5);
    simonFunction.Append(3);

    assertEquals(30, simonFunction.ComputeIncremental());
}

public void testStrikeIncremental()
{
    simonFunction.Append(17);
```

```
        simonFunction.Append(5);
        simonFunction.Append(3);
        simonFunction.Append(2);
        simonFunction.Append(1);

        assertEquals(36, simonFunction.ComputeIncremental());
    }

    private SimonFunction simonFunction;
}
```

Modification task 1 OO isomorph unit tests

```
import junit.framework.TestCase;
import java.util.*;

public class SimonFunctionTest extends TestCase
{
    protected void setUp()
    {
        simonFunction = new SimonFunction();
        rolls = new ArrayList();
    }

    private void AppendRoll(int number)
    {
        rolls.add(new Integer(number));
    }
}
```

```
public void RollMany(int count, int roll)
{
    for (int i = 0; i < count; i++)
        AppendRoll(roll);
}

public void testGutterBalls()
{
    RollMany(20, 0);
    simonFunction.BuildIterations(rolls);

    assertEquals(0, simonFunction.Compute());
}

public void testThrees()
{
    RollMany(20, 3);
    simonFunction.BuildIterations(rolls);

    assertEquals(60, simonFunction.Compute());
}

public void testSpare()
{
    AppendRoll(7);
    AppendRoll(10);
    AppendRoll(5);
    AppendRoll(3);
    RollMany(16, 0);
}
```

```
        simonFunction.BuildIterations(rolls);

        assertEquals(30, simonFunction.Compute());
    }

    public void testStrike()
    {
        AppendRoll(17);
        AppendRoll(5);
        AppendRoll(3);
        AppendRoll(2);
        AppendRoll(1);
        RollMany(14, 0);
        simonFunction.BuildIterations(rolls);

        assertEquals(36, simonFunction.Compute());
    }

    public void testPerfect()
    {
        RollMany(12, 17);
        simonFunction.BuildIterations(rolls);
        assertEquals(510, simonFunction.Compute());
    }

    public void testXGreaterThan()
    {
        AppendRoll(18);
        AppendRoll(7);
```

```
AppendRoll(5);  
AppendRoll(3);  
RollMany(16, 0);  
simonFunction.BuildIterations(rolls);  
  
assertEquals(-1, simonFunction.Compute());  
}
```

```
public void testSecondXGreaterThan()  
{  
    AppendRoll(7);  
    AppendRoll(18);  
    AppendRoll(5);  
    AppendRoll(3);  
    RollMany(16, 0);  
    simonFunction.BuildIterations(rolls);  
  
    assertEquals(-1, simonFunction.Compute());  
}
```

```
public void testXLessThan()  
{  
    AppendRoll(-1);  
    AppendRoll(7);  
    AppendRoll(5);  
    AppendRoll(3);  
    RollMany(16, 0);  
    simonFunction.BuildIterations(rolls);
```

```
        assertEquals(-1, simonFunction.Compute());
    }

    public void testSecondXLessThan()
    {
        AppendRoll(7);
        AppendRoll(-1);
        AppendRoll(5);
        AppendRoll(3);
        RollMany(16, 0);
        simonFunction.BuildIterations(rolls);

        assertEquals(-1, simonFunction.Compute());
    }

    public void testComboGreaterThan()
    {
        AppendRoll(7);
        AppendRoll(16);
        AppendRoll(5);
        AppendRoll(3);
        RollMany(16, 0);
        simonFunction.BuildIterations(rolls);

        assertEquals(-1, simonFunction.Compute());
    }

    private SimonFunction simonFunction;
    private ArrayList rolls;
```

```
}
```

Modification task 2 OO isomorph unit tests

```
import junit.framework.TestCase;
import java.util.*;

public class SimonFunctionTest extends TestCase
{
    protected void setUp()
    {
        simonFunction = new SimonFunction();
        rolls = new ArrayList();
    }

    private void AppendRoll(int number)
    {
        rolls.add(new Integer(number));
    }

    public void RollMany(int count, int roll)
    {
        for (int i = 0; i < count; i++)
            AppendRoll(roll);
    }

    public void testIncremental1()
    {
        AppendRoll(7);
    }
}
```

```
AppendRoll(9);
AppendRoll(5);
AppendRoll(3);
AppendRoll(14);
simonFunction.BuildIterations(rolls);

assertEquals(24, simonFunction.ComputeIncremental());
}

public void testGutterBallsIncremental()
{
    RollMany(2, 0);
    simonFunction.BuildIterations(rolls);

    assertEquals(0, simonFunction.ComputeIncremental());
}

public void testThreesIncremental()
{
    RollMany(8, 3);
    simonFunction.BuildIterations(rolls);

    assertEquals(24, simonFunction.ComputeIncremental());
}

public void testSpareIncremental()
{
    AppendRoll(7);
    AppendRoll(10);
```

```
        AppendRoll(5);
        AppendRoll(3);
        simonFunction.BuildIterations(rolls);

        assertEquals(30, simonFunction.ComputeIncremental());
    }

    public void testStrikeIncremental()
    {
        AppendRoll(17);
        AppendRoll(5);
        AppendRoll(3);
        AppendRoll(2);
        AppendRoll(1);
        simonFunction.BuildIterations(rolls);

        assertEquals(36, simonFunction.ComputeIncremental());
    }

    private SimonFunction simonFunction;
    private ArrayList rolls;
}
```