Handling Objects: A Scenario Based Approach
Thomas George Kannampallil
College of Information Sciences and Technology,
The Pennsylvania State University,
University Park, Pa 16802
thomasg@psu.edu
John M. Daughtry III
College of Information Sciences and Technology,
The Pennsylvania State University,
University Park, Pa 16802
jmd514@psu.edu

ABSTRACT
We report on the development of a tool which supports software programmers in programming to an interface by providing just in time (JIT) solutions. The development of the tool was grounded in a scenario-based analysis of programming scenarios and a procedural task analysis of these scenarios. The scenarios were analyzed using the theoretical perspectives of mental models and the production paradox of the user. Based on the scenarios and analysis a new refactoring and a tool were developed to support software programmers.

Categories and Subject Descriptors
D.2.2 [Design Tools and Techniques]: Object Oriented Design Methods, D.3.3 [Programming Languages]: polymorphism

General Terms
Documentation, Design

Keywords
Mental Models, Object-Oriented Programming, Prior Knowledge, Refactoring

1. INTRODUCTION
Agile software development processes have been getting increasing attention in the last few years. The drive behind agile development is to make development teams more agile, specifically leaner and more adaptable, by focusing the process of software development on quality incremental development that is flexible while maintaining robustness. This is a dramatic shift from previous methodologies and development models where it was mandatory for requirements elicitation and system design to be completed up-front.

A result of this shift from top-down models to agile development is that design decisions are moving from being made during system design stages to being made just-in-time (as the implementation takes place). Because of these just-in-time design decisions, integrated development environments (IDEs) are now being used not only as development environments, but also as problem-solving environments. Rather than being used as the environment in which a design is reified, they are being used to clarify and derive the entire solution. With software developers making on-the-fly design decisions, it is important for them to make accurate decisions as fast as possible. In addition, it is just as important that programmers be able to change their original decision at a later point. With the idea of changing a prior decision in mind, Fowler introduced the idea of refactoring [8].

Mastering Object Oriented (OO) programming languages require significant time and effort. Additionally making effective use of the programming language requires a deeper understanding. We present a specific example of declaring Objects. Through a task analytic approach we describe the challenges of software engineers to determine the correct interface for declaring objects. We have designed a tool, which will help programmer to make faster and more informed decisions when declaring objects. Though the tool has been developed for the specific purpose of supporting Java programmers to declare objects, we believe that such tools would be helpful for educational purposes.

The rest of the paper is organized as follows. The next section describes certain scenarios of use. The scenarios are used to develop a task analytic model to study object declaration. The possible theoretical issues, which we infer from the task analytic model, are also described. We also discuss how we developed a system that addresses the issues identified, and examine the trade-offs between the current process and our tool. In conclusion, we discuss future research directions that have arisen out of this work.

2. PROBLEM STATEMENT
Gama et al [9] say that one of the two most important principles every software engineer should follow is to program to an interface as opposed to programming to an implementation. To be more specific, a class should be written against a contract for its usage. In Java, this contract is called an interface. In fact, a class can be coded to conform itself to many interfaces, and within its code it promises to abide by these contracts by explicitly promising to abide by them. Since classes can have many interfaces, and these interfaces can have many superinterfaces, the process of selecting an interface to represent an implementation is labor-intensive and error-prone.

One extreme of programming to an interface is to program to the highest-level interface that includes the methods that are needed. In other words, committing only to the interface that includes only what they need to use with as few additional methods as possible. The other extreme of programming to an interface is to program to the implementation only, never to an interface. There are trade-offs between these opposing options which will be addressed later. However, we would like to note that the tool we
developed supports various views of what interfaces are and the level of the interface one should commit to.

Before analyzing the process of choosing an interface to represent an implementation, it is necessary to discuss some issues and trade-offs when programming to various levels of interfaces. Specifically, we discuss the definition and importance of polymorphism, type proliferation, and type specificity.

2.1 Polymorphism
Polymorphic functions are functions whose operands (actual parameters) can have more than one type. Polymorphic types may be defined as those whose operations are applicable to operands of more than one type. Polymorphism [17] is a feature that allows one java interface to be used for a general class of actions [11]. This means that it is possible to design a generic interface for a group of related activities. Polymorphism is the ability of an entity to take on different forms at run-time. The forms it can take are the forms of its interfaces. This is useful in situations where entities are of the same general form, but differ in specific detail [1]. Polymorphism is, in fact, one of the three core ideas behind object-oriented programming [11]. An example of its power is the Factory Pattern introduced by Gamma et al [9].

Polymorphism is affected greatly by object declaration. At the time of object creation, a programmer is committing the object to a variable that controls to what interface the rest of the system has access. The more specific the commitment, the potential the variable has for polymorphic behavior. By programming to an implementation you can never make use of polymorphism, so it is imperative that programmers move into the interface hierarchy in cases requiring polymorphism at a minimum.

2.2 Type Proliferation
The patterns community has addressed class proliferation as a problem that can and should be solved at the root level. However, considering the uses desired of software, it is impossible to reduce the number of classes beyond a certain point without losing flexibility. Java, for instance, has been refining the included API classes since its inception. They are flexible and well-designed. However, there are still thousands of them. In this respect, today’s object-oriented technology inherently causes problems with type proliferation.

2.3 Type Specificity
Type specificity is the notional identification of the level of an implementation. For instance, two different levels of type specificity are when an ArrayList is placed in a Collection variable or an ArrayList variable. There are advantages and disadvantages to each strategy. The purpose of this paper is not to examine or promote either strategy, but to better support the processes required by either (or any varying level of specificity).

The main advantage of declaring an ArrayList as a Collection (being very general in specificity) is that as a programmer you have not committed to an implementation, allowing for polymorphism. You can easily swap out a LinkedList for an ArrayList, or any method that has been added. The disadvantage is that you have not limited the usage of your object to a predefined set of operations. You can easily use any method available within the ArrayList, or any method that has been added. The disadvantage is that you have promoted the use of those methods over potentially sufficient similar methods, thereby limiting the potential of the object for polymorphic behavior without refactoring of the method calls.

However, there are a variety of options when choosing a variable type to which to code. The most general option is to code to the highest possible interface. The most specific option is to code to the implementation. Examples of options that fall between these are the option to code to the highest interface for inferred potential usage or to code to the most specific interface (but not the implementation).

3. SCENARIO BASED ANALYSIS
We use a scenario-based approach to understand and analyze the issues created by type proliferation and the need for supporting polymorphism. The development of our tool is based on the results derived from the scenario analysis. These scenarios are used to understand the usage of objects in programming contexts. They are then analyzed using a procedural task analysis (PTA) to derive the effectiveness of the methods employed by programmers.

3.1 Example Scenarios of Use
In this section, we use scenarios to understand three different programming situations. Scenarios are narratives that describe the details of a user interaction with a system or application. In more detail, a scenario is “a concrete description of an activity that the user engages in when performing a specific task, a description sufficiently detailed so the design implications can be inferred and reasoned about” [3]. The essence of scenario-based methods is that system design is grounded in the concrete use scenarios for which it is intended.

Figure 1. Screen shot showing the class lists, and interfaces
The three scenarios we describe are not the only usage scenarios for our problem. However, they are a representative subset. In these scenarios, a software engineer declares an object and decides to which interface (or implementation) it should be coded. The java.util.collections package was chosen for discussion as it is commonly used, mature, and a great example for object-oriented concepts. Scenario 1 describes Bob as he declares an
ArrayList and programs to the implementation of the ArrayList. The advantage to this approach is that it is quick to implement. However, the result violates the notion of programming to an implementation as discussed by Gamma et al [9].

**Scenario 1: Programming to an implementation**

**Goal:** To make the program function per an implementation  
**Actor:** Bob, a Software Engineer  
**Narrative:** Bob is writing an application in Java and needs to collect objects in one section of code and use them in another. He is aware of the ArrayList and knows that it will suffice for his needs. He declares the object using the statement:

```java
ArrayList bag = new ArrayList();
```

Bob then proceeds to use the object he created throughout the rest of his code.

Scenario 2 describes Andy, as he declares an ArrayList as a Collection and programs to that interface. The advantage of Andy’s knowledge about the collections package is that he can make quick decisions without having to look up anything. However, there is an issue associated with this scenario. Namely, no programmer can be reasonably expected to be familiar with every package and class in Java given our current problems with type proliferation.

**Scenario 2: Expert Usage**

**Goal:** To make the program function using an implementation that abides the rules of an interface that matches the conceptual level of the object’s usage  
**Actor:** Andy, a Software Engineer  
**Narrative:** Andy is writing an application in Java and needs to collect objects in one section of code and use them in another. He is aware of the Collections package and knows that it will suffice for his needs. He declares the object using the statement:

```java
Collection bag = new ArrayList();
```

Andy then proceeds to use the object he created throughout the rest of his code.

Scenario 3 is the most complex. Mike, a software engineer, knows that he cannot make the same naïve decision that Bob made in scenario 1. However, he knows that his knowledge of the collections package is incomplete, possibly incorrect, so he refers to documentation. The advantage of this process is that he takes into account all the options that currently exist. The disadvantage is that it is time consuming and open for errors or partial solutions.

We began our research by looking more carefully at scenario 3. Indeed, we focus on scenario 3 throughout this paper, as it is the most complex scenario that we are aware of, for selecting an interface at object declaration.

**Scenario 3: Expert Usage**

**Goal:** To make the program function using an implementation that abides the rules of an interface and to commit to the interface that defines the least set of methods while including those he needs to use  
**Actor:** Mike, a Software Engineer  
**Narrator:** Mike is writing an application in Java and needs to collect objects in one section of code and use them in another. He is aware of the ArrayList class, and that it will suffice for his needs. He needs to use the add(Object) method to put things in the class and the get(int) method to access those objects later. However, he is not sure which interface he should use. He decides to find an interface using the Java API documentation.

Mike opens the Java API documentation (See Figure 1) and uses the Class List frame to locate the ArrayList class. The interfaces that ArrayList implements are Serializable, Clonable, Iterable, Collection, List, and RandomAccess. Mike is already familiar with the Serializable and Clonable interfaces and knows that they do not include the methods he wants to use. He then opens the link to the Collection interface. He does not notice that this is a superclass of the List interface, although the API does include this fact. He looks through the methods and does not find the get(i) method. He knows that if this interface does not contain the methods that superclasses will not contain them either, so returns to the ArrayList class and follows the link to the next interface, the List class. In the List interface, he finds both the methods he needs. He now needs to go through the interfaces that List are subclasses of to be sure that he has found the highest possible interface in the class structure. The superinterfaces are Collection and Iterable. He knows that he has already looked at the Collection interface, and is familiar with the Iterable interface, so he knows that he has found the highest-level interface in this path.

```java
List bag = new ArrayList();
```

Mike then proceeds to use the object he created throughout the rest of his code.

### 3.2 Analysis of Scenarios

PTA allows us to further examine the procedure [16]. Specifically, it helps us in determining what the user must do and know to complete the process. In addition, we identified partial solutions in the PTA so that early-committal to a partial solution could be identified. In this section we will analyze scenario 3 (See Figure 2) using PTA and discuss the results, focusing on partial solutions and the complexity of the process.

You will notice that the process contains only seven steps between the start and stop states. However, the task is deceivingly more complex upon closer examination. There are several issues that the PTA reify, namely, that there is an exponential computational complexity, an exponential increase on what needs to be remembered throughout the process, and potential for early-committal.

The looping that takes place as the software engineer identifies matching interfaces and examines their superinterfaces results in a computational complexity of $N^2$. For scenario 3, the complexity is $N^2$ because he examines the interfaces of the ArrayList and the
superinterfaces of the List. This complexity results in an exponential increase in the number of steps.

In addition, Mike has to remember which interfaces he has previously examined, and which interface he currently believes to be the best choice. Admittedly, this could be aided by the browser changing the color of links which have already been followed. However, it is still an exponential amount of facts that Mike has to determine and remember.

Figure 1. Procedural Task Analysis for Scenario 3

One of the most important issues of the current process is that there is the possibility of early-committal to a partial solution. For instance, if Mike was trying to find the interface for the add(Object) and iterator() methods and examined the List interface before the Collection interface, he would have identified the List interface as a candidate. Having reached the goal of finding an interface, there is potential for him to not look farther and choose the List interface, which is the optimal solution for his goal.

4. ISSUES WITH THE PROCESS

As we have seen from the scenarios above, programmers can select non-optimal strategies (scenarios 1 and 2) or they can use a strategy that is extremely labor-intensive and still could lead to a less than optimal solution (scenario 3). These scenarios point to the fact that conceptual models of programmers could be incorrect or ill-formed with respect to the programming language itself. In addition, programmers try to achieve the simplest solution (Occam’s razor) within the constraints of time. Below we analyze the scenarios we developed with respect to the two main challenges we identified – users' mental models and production paradox.

Mental models are often described as mental representations that users form of a system, process or object. From a programming perspective, mental models of users are conceptual models that users develop about programming strategies. We examine how incorrect conceptual mental models of problem situations can lead to errors in judgment, thereby resulting in incorrect or less than optimal problem solutions. Thus, irrespective of the problem domains, it is critical for users to have as accurate as possible mental models of problem solving. With respect to the software engineering domain, software programmers are constantly engaged in problem-solving exercises. An interesting aspect of this domain is that a software engineer has to conceptually understand at a minimum two parallel threads of ideas – a conceptual model of the software engineering problem she is trying to solve and a conceptual model of the programming environment (i.e. language) itself. An incorrect perception of either of these would lead to incorrect or less than optimal problem solutions.

The study of mental models have been described in several areas from electricity [5], use of calculators and mechanical devices [4], computer usage [15], space and motion [7], and learning and problem representation [10]. The importance of programmers’ mental representation has also been studied in great detail [13]. As it has been explicitly narrated in scenario 2, Andy uses an object declaration based on his conceptual mental view of the programming language (Java in the case of this scenario). But, he does not consider the trade offs in using the object declaration. Thus, as it will be later explicated, his mental model about the programming language was flawed and that can result in him choosing a less than perfect solution for his design. Thus ill-formed mental models hamper effective design process of a software engineer, and it is required to have a tool which could help the user to overcome this problem.

Scenario 3, describes a situation in which the programmer uses an external feature to support his programming task. However, externalizing the task requires a huge overload in the number of iterative steps required in reaching the ideal solution. Also, this externalization need not lead to a perfect solution, as the search can be mired in local optima. Thus the labor-intensive process of externalizing the process of finding the solution may not lead to an accurate solution strategy.

Carroll and Rosson [2] describe the paradox of the active user as a principle of human cognition according to which people are driven to produce direct, meaningful results from their work,
based on their prior experiences. The users are, whether expert or novice, driven by two paradoxes – production and assimilation. The production paradox is that most users of software tools are focused on managing their immediate workload concerns and are less interested in understanding and learning about the tools that they use, though these tools may be very useful for them in the long term.

Rieman [14], for example, describes a field study of the behavior and attitudes of computer users in everyday working situations that focused on the exploratory learning behavior of users. The exploratory behavior is limited by time and task constraints. Neerincx and de Greef [12] evaluated the on-line help system of the statistical software package SPSS/PC, and their results showed that an integrated help system did not improve the performance of the users because both the volume and content of technical manuals did not match user goal requirement. System users want the information needed at a certain moment and that is relevant in the current task context [6]. Research literature does not provide specific examples of production paradox with respect to software programming. We believe that the above-mentioned scenarios of computer usage, statistical packages etc., are similar environments to programming environments.

Scenario 2 described above provides an example of how a user (Andy) uses an object (based on prior knowledge or his expertise). Both assimilation and production paradox can be attributed to this activity. It could have been a result of a production paradox and assimilation paradox because the user was trying to complete the task in the quickest manner matching as closely as possible to his requirements. Scenario 3 however shows a labor-intensive process in which the user tries to find an optimal design solution to his problem. This mechanism could provide very accurate results but is time-consuming and error-prone. The tool we developed provides a mediating strategy to solve the problem of assimilation and production paradox.

5. SOLUTION REQUIREMENTS

Our analysis within the theoretical framework of mental models provides insight as to how a solution can be reached. Specifically, the goal should be to ease the burden placed on the developer of maintaining a mental model of the programming environment. To reduce the reliance on such a mental model requires that we simplify and improve the quality of the task performed in scenario 3.

Our PTA of scenario 3 showed that the process was computationally complex, required the storage of exponential facts, and contained points of partial completion. To improve the process, we set out to address each of these negative effects. We also kept in mind additional requirements, namely not interrupting the existing flow of events, implementing as a just-in-time system, and that the results should not be filtered.

We believed that the existing process could be improved using an enabler without interrupting or altering the existing conceptual views or goals. More specifically, that we could create a tool that would allow the user to maintain their current goals and only eliminate sub-goals and tasks within the process.

Complementary to the requirement that it not alter the existing goals is that the tool operates as a just-in-time system. This tool has to operate just as a developer is making a decision and should provide an answer to an inquiry fast enough that using the tool does not cause the developer to have to wait on the response. In waiting, the developer could move on to other tasks and further complicate the process.

The last requirement we identified is that programmers should be able to program to any interface they wish, and that filtering the list to match any single usage scenario would be inhibiting the usage of the tool. Additionally, providing all of the information would perhaps have the added benefit of better supporting the mental models the user does have of the particular class hierarchy.

6. PROOF OF CONCEPT

We have implemented a system that simplifies the task of choosing an interface for object declaration that does not interrupt the flow of the existing process, operates in just-in-time scenarios, and does not restrict its feedback to a set of results. The system consists of a refactoring formalizing the steps of selecting an interface, and a tool that supports this refactoring.

6.1 The Tool

To support the refactoring, we developed a tool that helps users understand the problem space of declaring objects. It allows users to examine a class and derive the potential interfaces based on its usage. Figure 3 shows the main screen of the tool. The programmer enters the name of the class defining the object they are creating and click Find Now. Upon clicking, combo-boxes on the left side of the screen are populated with the methods available to this object. These methods could be defined at any level is the class hierarchy. The user then selects the methods that they are using or intending to use and clicks Get Results. At this point a list of all available interfaces is populated in the text-box on the right-hand side of the system.

![Figure 3. Screen shot of the tool, showing the results of a search](image)

6.1.1 Usage Scenario

The scenario below explains an actual usage situation of the tool. A task analysis is performed to compare this scenario to scenario 3 used earlier.
Scenario of Use (tool)

**Goal:** To make the program function to an interface

**Actor:** Bob, a Software Engineer

**Narrative:** Bob is writing an application in Java and needs to collect objects in one section of code and use them in another. He is aware of the ArrayList and knows that it will suffice for his needs. He uses the tool and types java.util.ArrayList in the text box and clicks Find Now (See Figure 3 above). The list of methods for ArrayList appears in drop down menus. He chooses the method he needs (add()) and clicks the Get Results button. The list of interfaces that he could probably code to appear on the right side of his screen (with the ideal solution appearing lowest – List in this case). He chooses list as the interface.

6.1.2 Claims analysis of resulting system

Based on our evaluation of the tool, we argue that the tool provides considerable advantages for a software programmer.

+ By externalizing the possible relationships explicitly we are supporting the exploratory behavior of the users
+ Externalization also reifies the mental model of the programmer and also helps in overcoming prior biases
+ Programmers will potentially better understand polymorphic potential of the objects they are selecting to use
+ Since all levels of abstraction are provided to a programmer when he/she uses the tool, it is useful for all programmers

6.1.3 Task Analysis of the Tool and Claims

As can be seen from Figure 4, the number of steps involved in the process (using scenario 4) is three. Compared to scenario 3, the task has become significantly less complicated. As opposed to the computational complexity of \(N^4\), the complexity of the process reduces to \(N\). Also, the possibility of reaching a partial solution is avoided since the solutions are presented to the programmer (in complete form) and then he/she can decide on whether to use it or not.

![Figure 3. Task Analysis using the tool](image)

As the different steps in the process of finding the optimal solution are presented in a simple and less labor intensive manner, it reduces the possibility that a person is affected by his/her prior learning, thus reducing the assimilation paradox. The assimilation paradox is reduced to a situation in which the programmer has to choose between the different options that are available to him. This also provides an opportunity to correct his ill-formed mental model.

From our task based evaluation of the tool, we reduced the computational complexity of the tool to the order of \(N\) (from \(N^4\)), reduced the possibility of partial solutions and provided Just in Time solutions to the users.

### 7. FUTURE DIRECTIONS

Based on our evaluation of the tool, we believe that there are several additions that can be made. Our primary focus now is to incorporate the tool into the Eclipse IDE as a plug in. This would help us in doing evaluation studies as the tool would be integrated with the programming environment. The evaluation studies would help us in more accurately characterizing the task, thus providing us with a mechanism to understand the levels of complexity associated with other scenarios of this tool.

Since this tool is essentially a recommender system, we would like to provide a mechanism for the users to understand the levels of importance of the different recommendations and perhaps provide recommendations based on some pre-defined preferences. We propose to implement a representation mechanism for the display of recommendation choices.

Another aspect of the tool which we are interested in pursuing is over-riding – both human over-ride of the system recommendations and the system override of the human choices. The first case, human override, would be easier to implement and would be like any standard implementation in current Eclipse versions. The system override (a forced override) of human choices would be interesting from the perspective of empirical evaluation. A forced over ride could be applied to all the three scenarios and it would lead to potentially differing responses from the programmers. We think that this would be an interesting domain to do further research and evaluation.

### 7. ACKNOWLEDGMENTS

We would like to thank Mary Beth Rosson and Joshua Gross for providing constructive comments for improving this paper.

### 8. REFERENCES


