

Refactoring Software Using Design Patterns

Masatomo Noborikawa

Under the direction of Dr. Eugene Wallingford
Research Project for Masters of Science in Computer Science
University of Northern Iowa

May 5, 2003

Table of Contents

1. Introduction.....	3
2. Overview of Tools	5
2.1 What are design patterns?	5
2.1 What is Refactoring?	7
3. Basic Facts	9
4. Initial Prototype	11
4.1 First Design Attempt	11
4.2 First Refactoring	16
5. Development Episode	25
6. Modified Prototype	27
6.1 Modified Design Attempt	27
6.2 Systematic Refactoring	32
6.2.1 Refactoring to clean up codes	32
6.2.2 Refactoring Creation of Quiz: refactoring to factory pattern	37
6.2.3 Refactoring the Problem-Quiz-Portfolio hierarchy	42
7. Conclusion	52
References.....	54

Chapter 1

Introduction

Designing a software system prior to implementation often results in designs that are incomplete or incorrect. Creating a good design is difficult because designers do not always know everything they need to know about the solution before writing the program. Often, the programmer must correct mistakes in the design as he comes to understand the problem and come up with better solution while writing the program.

Even when the programmer comes to know that something is not right in the design, he might not often be able to come up with the correct solution right away. The process of "refactoring" guides the programmer as he identifies weaknesses in the design and code and provides methods to improve them. This research project investigated the use of design patterns and the application of various refactoring techniques as a means to trigger modifications to designs and programs that preserve the functionality of the system while improving the quality of the software. This approach offers a powerful set of tools for programmers to use as they evolve their designs throughout the writing of a program.

In this research project software called *Basic Facts* was developed from scratch. Varieties of refactoring techniques and design patterns were applied to enhance its functionality. The details of the sequences of these developments are illustrated throughout the paper which is sub-sectioned in the form of chapters.

Chapter two provides an overview of what design patterns and refactoring are, and what are their advantages and disadvantages. They are essential tools for this development. Chapter three explains the fundamentals of *Basic Facts* and its very original software. Chapter four describes how the initial design was created and how design patterns were applied along with modification to the system. Refactoring in this chapter is used in a different context compared with the way of refactoring in chapter six: It is done in a more intuitive way. Chapter five talks about the different types of pattern usage and how they may affect the development of software. Chapter six shows step-by-step refactoring applied to the second design. Various refactoring techniques were implemented to enhance the usability of the design patterns that lead to the final design of the software. Chapter seven provides the conclusive thoughts of the research.

In this paper, italicized words are used to describe abstract classes, bold type represents normal classes, and words enclosed with $\langle \rangle$ signify interfaces. Methods (or message to which owning objects respond) are underlined.

There is a note for code explanation to be mentioned. This paper uses codes extracted from source to explain refactoring in step-by-step basis. In code tables that the reader will see in chapter 6, ‘...’ refers to repetitive codes that are previously mentioned. In addition, codes in bold type highlight changes as compared with ones in the previous table.

Italicized words followed by a number enclosed with parentheses (e.g. *Extract Method (110)*) mean that it is a name of refactoring techniques cataloged on the page of the given number in Fowler’s book (2000).

Chapter 2

Overview of Tools

2.1 What are design patterns?

Design patterns represent the best practices of experienced object-oriented software developers. Design patterns are recurring solutions to common software design problems. These solutions were discovered through trial and error by numerous software developers over a substantial period of time. Gamma, Erich, Helm, Johnson, and Vlissides (1995) catalog those best practices with 23 patterns. The authors describe each design pattern with a name and an intent, which defines what the pattern does and what design issues or problems it addresses. Just as object-oriented programming encourages code reuse, design patterns encourage design reuse. Design patterns help programmers capture the features of good programming solution and create more reusable and maintainable programs.

The following lists are the advantages and disadvantages of design patterns.

[Advantages]

- Programmers do not need to reinvent solutions to known design problems.
- Design patterns give novice developers access to the best practices proven by expert developers.
- Design patterns allow developers to think about their designs at a higher level of abstraction. For example, instead of focusing on low-level details, such as how to use inheritance, a developer can approach complex systems through a collection of design patterns that already make the best use of inheritance.
- Design patterns provide a common vocabulary for developers to discuss design. A design pattern vocabulary conveys a particular solution to a design problem more succinctly than explaining the solution with a lot of words.
- Knowledge of design patterns often brings insight in designing flexible software.

[Disadvantages]

- Design patterns are hard to understand. It takes a while for non-experienced developers to understand patterns and reach “Aha” moment.
- Preoccupation with design patterns sometimes keeps developers from finding simpler solutions (Kerievsky, 2002). In some case, there is a simpler solution to a facing problem than one using a pattern. If a programmer believes a solution with patterns is an ultimate one, he or she may miss small, simple, and straightforward solution.

Design patterns will benefit developing software and make it flexible, modular, and reusable. However, knowing design patterns does not always mean that they will be usable. Because of the nature of Object-Oriented development, the design tends to be changed and refined time to time. It is not rare that the designer finds it useful to apply some design patterns a while after the development starts. Yet compared with very early development stage, it is often harder to apply patterns in midst of the development. Design patterns do not fit right away. Program codes require some kind of cleaning up or transforming for the use of the patterns. Refactoring then comes into play. Refactoring cleans up codes and help a target design pattern fit in the program better. Refactoring is a powerful tool to increase the chances of using patterns and improving the quality of software. The following section will explain refactoring.

2.2 What is Refactoring?

Refactoring is the art and science of improving existing code. Refactoring, according to Fowler (2000), is the activity of reorganizing the design or internal mechanism of software in order to make the software easier to understand and modify, without affecting its external behavior. Typically when application development begins, prototyping is used to see how things work. As program requirements grow, the same code is tailored, modified and extended to add a new functionality. With the growing need for change, the software turns out to be lump of spaghetti code that can become a nightmare to manage unless handled with great care. Refactoring helps programmers to maintain the functionality of the code developed during prototyping and improves its quality for the next iteration.

During refactoring there are three disciplines that must be followed:

1. *Testing after each refactoring*

Testing is a precondition for refactoring. Since refactoring makes a change to the internal structures of the software, we need to have a way to check that the change does not alter its observable behavior. No matter how hard we try to avoid introducing bugs, we are still human and we have always possibility to make mistakes during refactoring. Therefore, we need solid tests to detect such bugs. A set of solid tests are essential for refactoring.

2. *Self-check testing as much as possible*

Testing should be done by self-checking. Self-checking test means that it should say “Ok” if the expected output from a testing method is a string such as “AbCdd” instead of printing the output literary. If the automatic testing is not used, programmers end up spending time hand checking the output from the test against an expected output. It then slows down their development. A self-checking test is a powerful bug detector to save the time it takes to find bugs.

3. *Refactoring and adding functionality separately*

When refactoring, it is advisable not to add new functionality to a code. This sounds easy, but it can become difficult at times. While refactoring, the programmer will occasionally find a place where a new functionality is called

for. The addition of new functionality during refactoring may require several steps of backtracking because adding new functionality by itself entails a change in the behavior of the system and requires other sets of testing procedures. This is not at the best interest of the programmer who presumably tries to maintain the very behavior that might change as a result of the new functionality being inserted. The obvious consequence might waste many hours of developments.

There are many benefits of refactoring as follows:

- Refactoring improves the design of software. Refactoring often cleans up codes by deleting duplicates, divides a big chunk of codes into several methods, and makes the program more understandable.
- Because refactoring makes a design cleaner, it helps the programmers understand codes better and see things that may have not been seen before.
- Refactoring helps spot bugs since it makes the software more comprehensible.
- Refactoring turns an adverse design into a good design, which in turn allow for rapid software development.

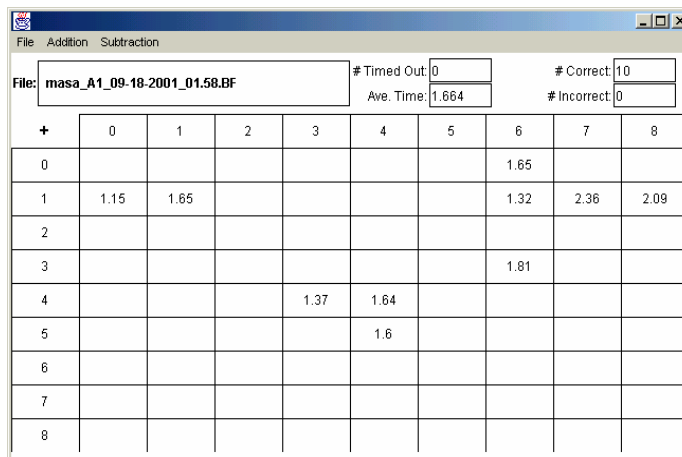
Chapter 3

Basic Facts

Basic Facts is a program that helps students learn fundamental math operations such as addition and subtraction. The software is intended to help teachers identify the type of mistakes students tend to make and combinations of numbers in an equation that slow down students' calculations. All questions consist of two numbers and one operation. There are two types of users in this program.

As a student user, the student first identifies himself by typing his name in a given textbox and kicks off the program. As the identification is done, the program makes his portfolio (a folder/directory) to store his work behind scenes. The user then selects a quiz to play from a menu. After his play, the result, such as how much time was taken to solve each question and which questions were missed, are stored in the user's portfolio.

A teacher user can select any of his student's results and view it in a table format. Each result shows the number of seconds taken to solve each question, how many questions were missed, and timeouts if the student spent too much time attempting to answer a question (Figure 1). A decimal number in a grid represents seconds for the student to complete a particular question. For example, a 1.37 across from the 4 and under the 3 means that it took the student 1.37 seconds to complete the problem $4 + 3$. By observing the outcome, the teacher may diagnose which problems the student needs more practice on.



	0	1	2	3	4	5	6	7	8
+									
0							1.65		
1	1.15	1.65					1.32	2.36	2.09
2									
3							1.81		
4				1.37	1.64				
5					1.6				
6									
7									
8									

Figure 1 Student's play result view

This program has a simple structure and easy-to-use interface. For student users, there are two operation menus to select from: Addition or Subtraction. Each menu has five subcategories that become more difficult as the level increases. In order to play, students select a sub category. They click the buttons on the screen or type the number keys on the keyboard to provide a solution to the question. After each question is answered, the program provides feedback and states whether the answer is right or wrong (Figure 2).

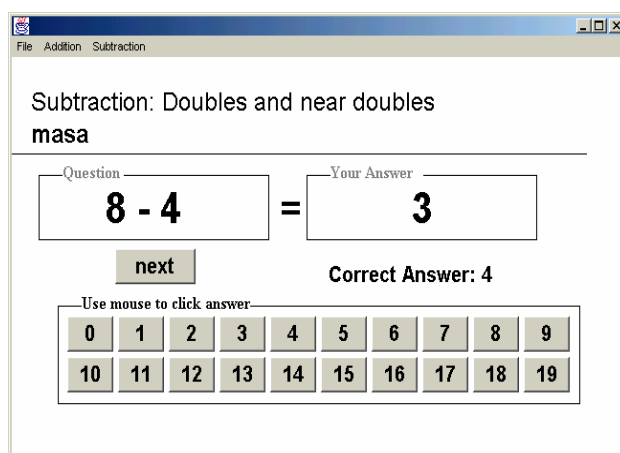


Figure 2 Screen shot of Basic Facts

Every time a student plays, his results are kept in files within the directories that are created when he build his portfolio. To switch to a teacher mode, the user selects “switch to teacher” from the file menu, or “teacher” when the program is run. The user must enter a password to gain access to the student’s portfolios. Selecting “Open File” from the file menu to view student’s work pops up a file dialog box. The teacher user searches for a desired file in the target student’s folder and clicks to open.

This program was first written in C++ in a Mac OS environment. It has appealing features and many potential ways to extend its uses. The only stumbling problem existed was that its source code including its documentation did not exist limiting future enhancements to it. I then wrote an application in Java mainly because Java supports Object Oriented programming. My first mission for the development of this application was “to make a Basic Facts application that works as close to the original as possible.”

Chapter 4

Initial Prototype

4.1 First Design Attempt

Basic Facts was my first attempt to write an object-oriented program without having an assigned specification. My lack of experience with OOP and Java programming made the assignment quite a struggle, yet one quote on a class note from my OOP class has always stayed in my mind: “Good design comes from experience. Experience comes from bad design.” Believing that quote, I assembled the first design of the Basic Facts program.

Based on my observation of Basic Facts the following points are requirements the software needs to meet:

- a) Basic Facts has two types of users, student and teacher.
- b) The first time the application runs, the user can choose to be either a student or a teacher.
- c) A user can change to either player by selecting “switch to teacher” or “new user.”
- d) A password is required to become a teacher user.
- e) Both users have the same menus on the menu bar.
- f) When a user is a student, “open file” on the file menu is inactivated.
- g) When a user is a teacher, all math problems are inactivated.
- h) A student can play two kinds of math problems with many subcategories that have the same organization and display.
- i) Feedback is given after each answer is submitted.
- j) Only a teacher can display a student’s play result.
- k) A play result shows its file name, time taken to solve each question, average time to solve a question, number of timeouts, and number of correct and incorrect answers.

Using these specifications, I designed and implemented Basic Facts in Java. This program ended up consisting of many classes and some interfaces. The class diagrams of BasicFacts are divided into three figures, Figure 3, 4 and 5. Note that all class

diagrams provided in this paper have been illustrated using Unified Modeling Language (UML). Some trivial classes used in this program have been omitted for convenience.

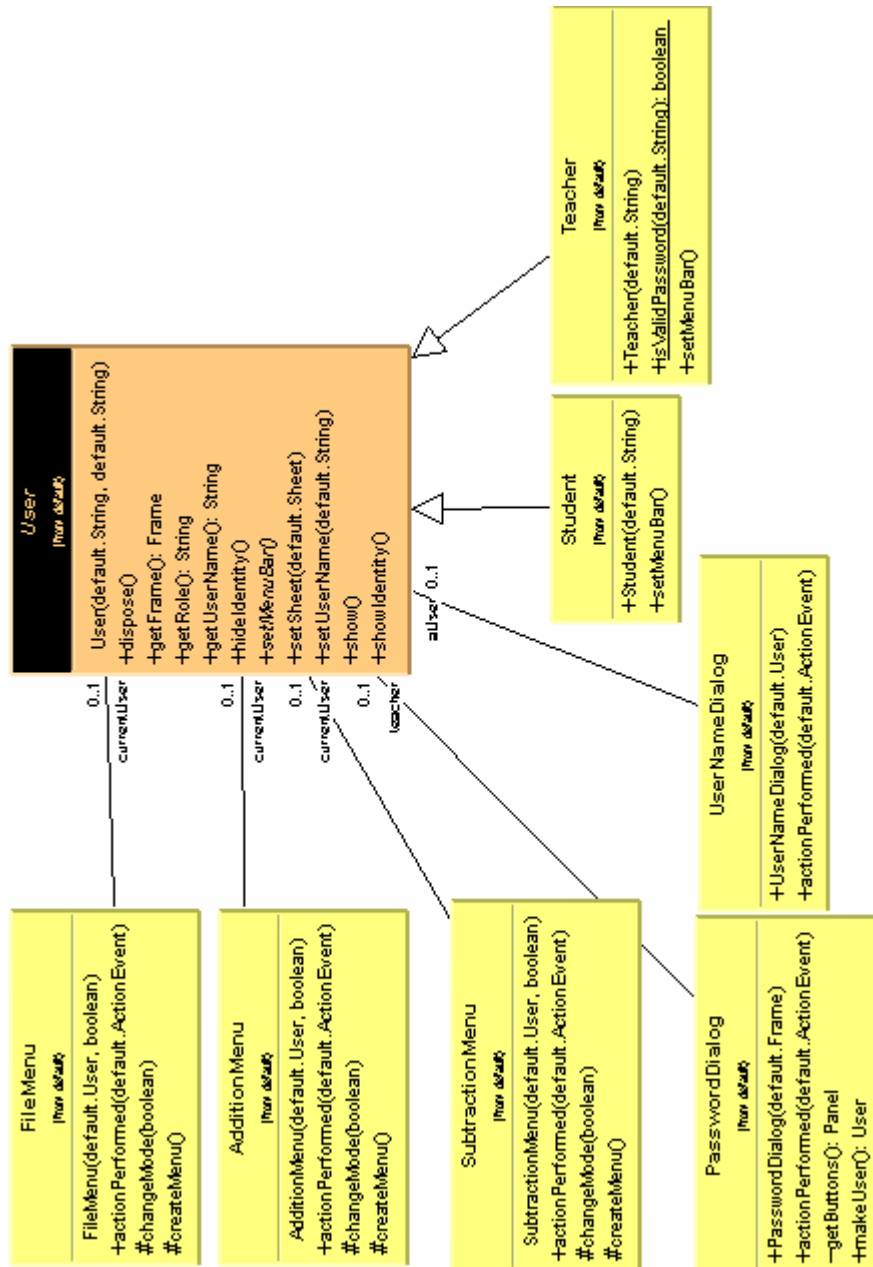


Figure 3 The class diagram around User class

User class is an abstract class that holds behaviors common to both Student and Teacher (Fig. 3). Both Student and Teacher are extended from *User*. Each of them specialize *User* class. Because Teacher must know the password to gain access to student's play results, the isValidPassword() method is implemented in Teacher (specification d). The setMenuBar() method is an abstract template method that defers the creation of menu objects to its subclasses. It was made to meet specifications e), f), and g). This class hierarchy models the structure described in specification a).

All Menus: FileMenu, AdditionMenu, and SubtractionMenu are set to a java.awt.Frame instance variable in *User* when *User* responds to a setMenuBar() message. FileMenu class has menu items that are responsible for changing one user mode to another (the specification c)), showing/hiding a user's identity (role or user name), opening a play result if the user is in teacher mode, and termination of the program. The *AdditionMenu* and *SubtractionMenu* classes have choices that allow the user to select different math programs (the specification h)). In the response to changeMode() method in these menu classes, some menu items are inactivated according to a boolean parameter to meet the specification f) and g).

PasswordDialog is a java.awt.Dialog that asks for a password to create a Teacher object. UserNameDialog is also an extended java.awt.Dialog to get a unique user name to create Student object.

Basic Facts in Figure 4 is a class to kick off this program. It creates a SelectUser object that allows the user to select either the student or teacher modes, and create a corresponding *User* object to play. The Sheet class is derived from a java.awt.Container class that provides a graphical user interface (GUI) allowing the user to interact with the program or view its contents. In response to a makeGUIContainer message, Sheet creates GUI objects and returns itself with the objects to its client. The AnswerSheet class specializes its super class Sheet for a student user to complete math problems.

All protected methods (starting with “#”) in the AnswerSheet class are used for creating and aligning the GUI objects displayed in Figure 2. Three private methods (starting with “-“) are needed to control interactions between a <Question> object and the GUI components needed when a user causes an action event to occur while solving a math problem. The ResultSheet class contains StatisticTable, ResultChart, FileNameBox to display a student's play result as shown in Figure 1. Both the AnswerSheet and ResultSheet classes use a <QuestionFactory> object to create <Question> objects.

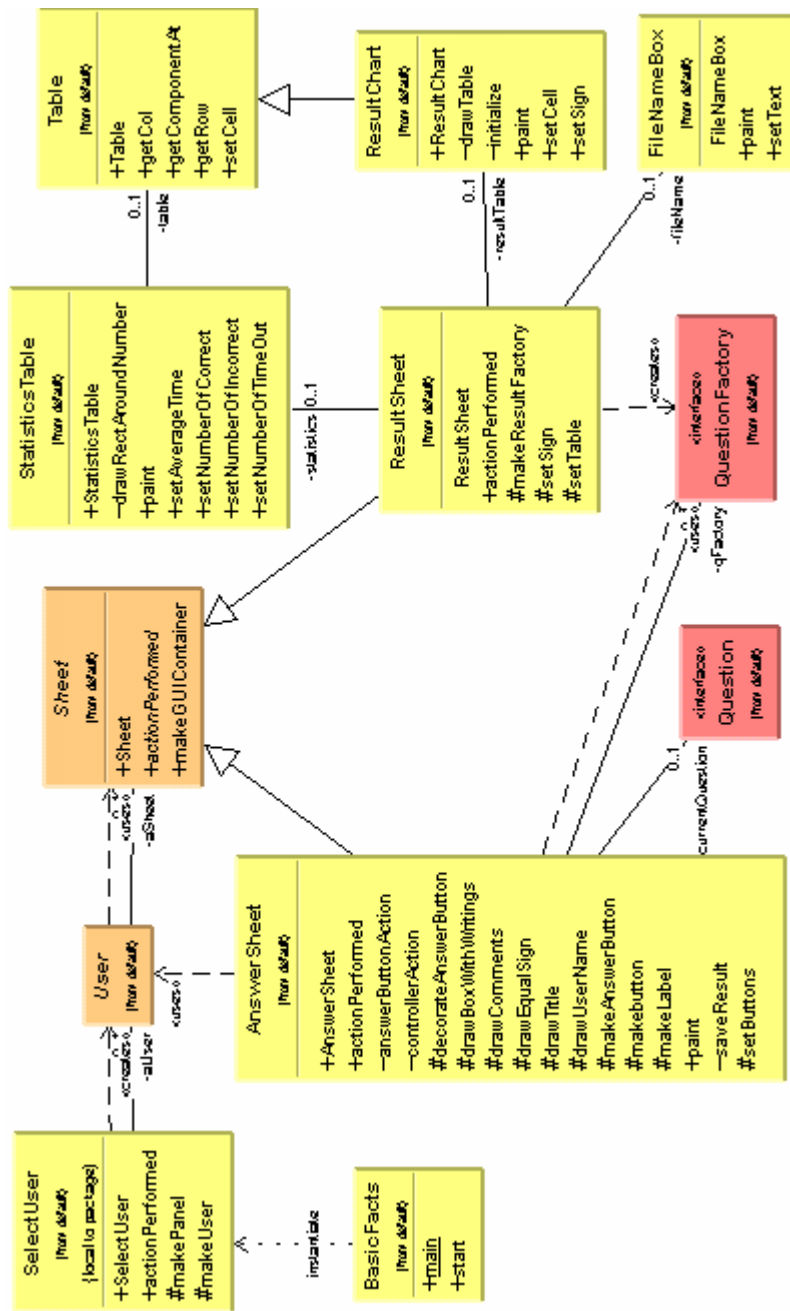


Figure 4. The class diagram around Sheet class

The structure of class relations between <Question> and <QuestionFactory> shows the *factory pattern* that a super class defers on instantiation to its subclasses (Figure 5). For instance, ResultFactory stores answers back to questions that have been solved and IntegerQuestionFactory produces not-yet-solved questions. The QuestionWriter class is used to create a file in the player's directory and store the information of play results there. QuestionReader reads the information from the file and instantiates ResultFactory for the preparation of displaying the play result.

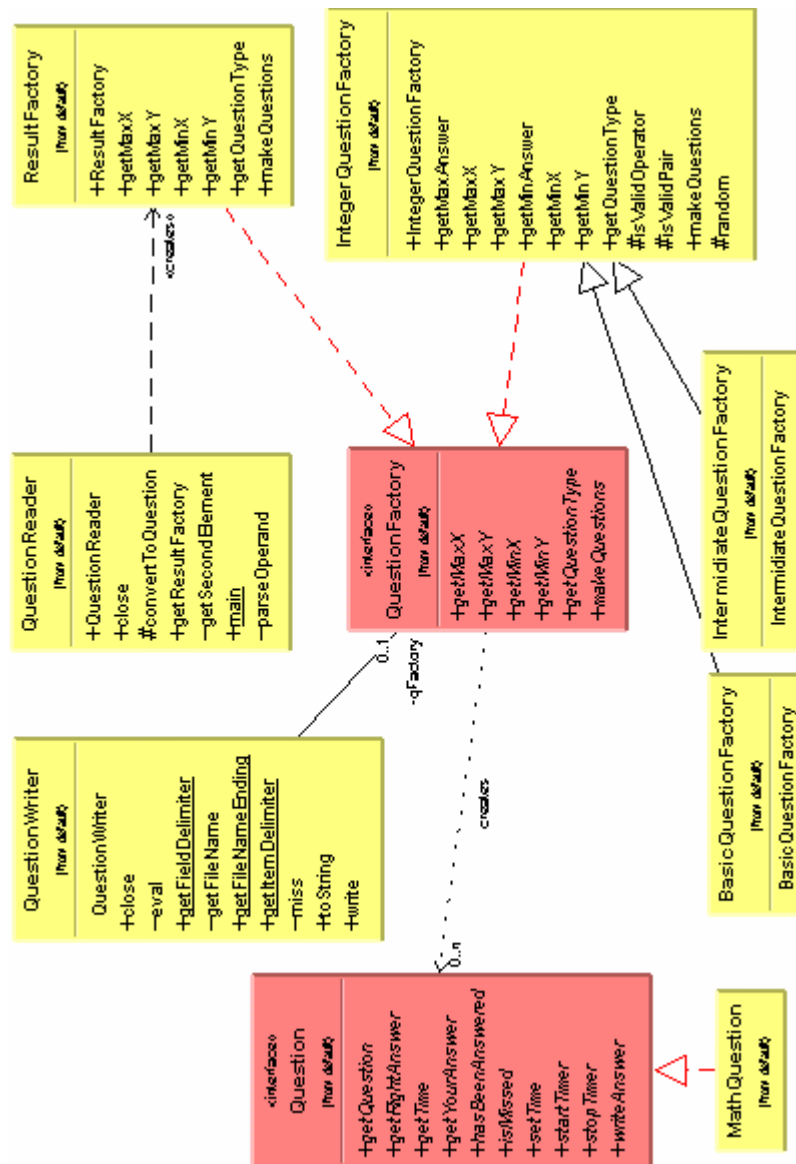


Figure 5. Question – Factory hierarchy with IO classes

4.2 First Refactoring

In my initial design of Basic Facts the program met the specifications outlined in the previous section and was able to perform its basic working functions. However when looking at the degree of reusability, this design needed better organization. I began to look for parts of the code that could be modified for improvement.

While looking through the class diagrams in Figure 3, it is noticed that AdditionMenu and SubtractionMenu consist of the same methods that behave in the same way to produce slightly different menus. How can their overlap be extracted and put into one class to gain generalization? The `createMenu()` and `actionPerformed()` methods in the class AdditionMenu are the key to that question.

As it is obvious, each menu's title is hard-coded in the instantiation of MenuItem's in `createMenu()` and in the creation of AnswerSheets class in `actionPerformed()` methods. Repeated code has a potential to cause problems and should be eliminated if possible. In the `actionPerformed()` method an AnswerSheet object is created by passing as a parameter an instance of QuestionFactory with slightly different arguments. Those arguments should be merged into one because they belong to the property of quizzes.

```
protected void createMenu()
{
    String letter = "A";
    MenuItem [] menuOne = new MenuItem[6];
    menuOne[0] = new MenuItem(letter+"1 Count on and zero generalization");
    menuOne[1] = new MenuItem(letter+"2 Doubles and near doubles");
    menuOne[2] = new MenuItem(letter+"3 Make ten");
    menuOne[3] = new MenuItem("-");
    menuOne[4] = new MenuItem(letter+"4 Mental Math1");
    menuOne[5] = new MenuItem(letter+"5 Mental Math2");
    for (int i=0; i<menuOne.length; i++)
    {
        menuOne[i].setActionCommand(letter+Integer.toString(i+1));
        menuOne[i].addActionListener(this);
        add(menuOne[i]);
    }
}
```

Figure 6. Creation of menu items in AdditionMenu class


```

public void actionPerformed(ActionEvent e){
    String qType = e.getActionCommand();

    if (qType.equals("A1")){
        currentUser.setSheet(
            new AnswerSheet(currentUser,
                "Addition: Count on and zero generalization",
                new BasicQuestionFactory(3,qType,"+"))
            );
        currentUser.show();
    }
    else if (qType.equals("A2")){
        currentUser.setSheet(
            new AnswerSheet(currentUser,
                "Addition: Doubles and near doubles",
                new BasicQuestionFactory(5,qType,"+"))
            );
        currentUser.show();
    }
    else if (qType.equals("A3")){
        .....
    }
}

```

Figure 7. actionPerformed method in AdditionMenu class

The instantiation of AnswerSheet objects with different quizzes is managed by a set of if-then-else statements producing a quiz type such as “A1.” Each of the if statement blocks look alike and are nearly identical. It is not good programming practice to have so many such if-then-else blocks in this way. Additionally, if the number of types of quizzes gets large, it will be difficult to maintain. Therefore we should look for a way to create specific quizzes using QuestionFactory without using if-then-else statements. How can all these problems be resolved? Through the use of *Singleton pattern* and *strategy pattern* these faults can be corrected. Singleton pattern ensures that a class has only one instance, and provides a global point of access to it. Strategy pattern encapsulates alternative strategies, or approaches, in separate classes so that each implements a common operation.

Here the QuizProperty class has been made to encapsulate a quiz’s property. It holds the quiz type, title, and operator as well as the range of literals to compute. LookUp class is extended from the java.util.Hashtable class and uses the singleton pattern to ensure that one and only one instance of a class exists and provides a global point of access to that class. QuizProperty is used in several places in this program to reference a property of quiz, but only one instance is necessary. LookUp class registers instances of all QuizPropertys and becomes a global point to it.

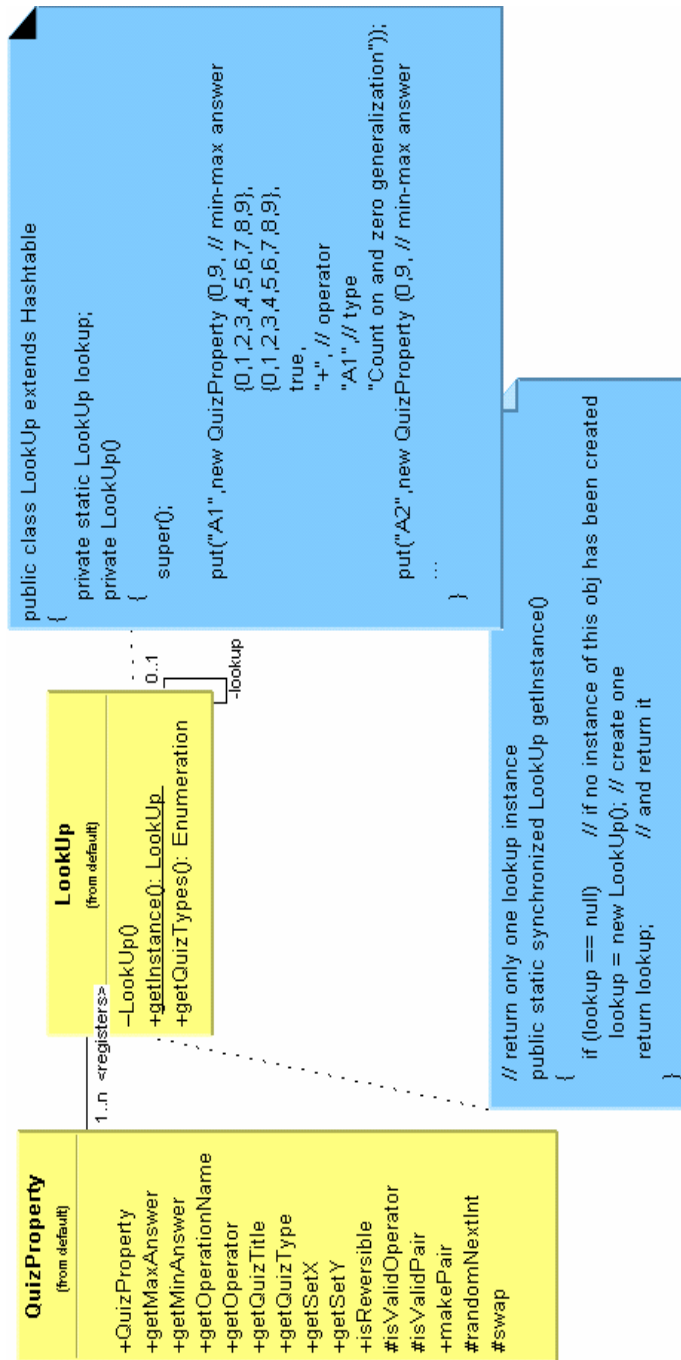


Figure 8. LookUp class modeling singleton pattern

The new createMenu() is shown in Figure 9. The getInstance() method creates an instance of LookUp if it has not been instantiated and returns it. The getQuizTypes() method returns an Enumeration object that holds various quiz types in String. Going through the quiz types, SelectionStrategy is used to capture a specific quiz type for the creation of its menu. This version of createMenu() is more generalized to produce many different kinds of menus.

```
protected void createMenu(SelectionStrategy selectionStrategy) {
    LookUp    lookup = LookUp.getInstance();
    Enumeration qTypes = lookup.getQuizTypes();
    QuizProperty aProperty=null;

    while (qTypes.hasMoreElements()) {
        String quizType = (String) qTypes.nextElement();
        if (selectionStrategy.hasFeature(quizType)) {
            aProperty = (QuizProperty) lookup.get(quizType);
            MenuItem quizItem = new MenuItem(quizType+" "+aProperty.getQuizTitle());
            quizItem.setActionCommand(quizType);
            quizItem.addActionListener(this);
            add(quizItem);
        }
    }
}
```

Figure 9. New createMenu method in QuizMenu class

Next let's compare the previous actionPerformed() method with a revised version shown in Figure 10. This new version is more concise and easier to understand. The instantiation of <Sheet> objects is done without the use of if statements. The creation of specific quizzes is delegated to MathQuizFactory that uses a quiz property extracted from a LookUp object.

```
public void actionPerformed(ActionEvent e)
{
    String command = e.getActionCommand();
    QuizFactory qf = new MathQuizFactory(command);
    currentUser.setSheet(
        new AnswerSheet2(currentUser,qf.makeQuiz(10))
    );
    currentUser.show();
}
```

Figure 10. New actionPerformed method in QuizMenu class

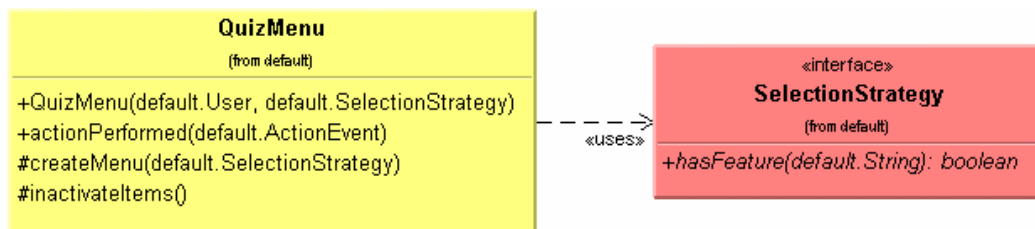


Figure 11. QuizMenu class diagram

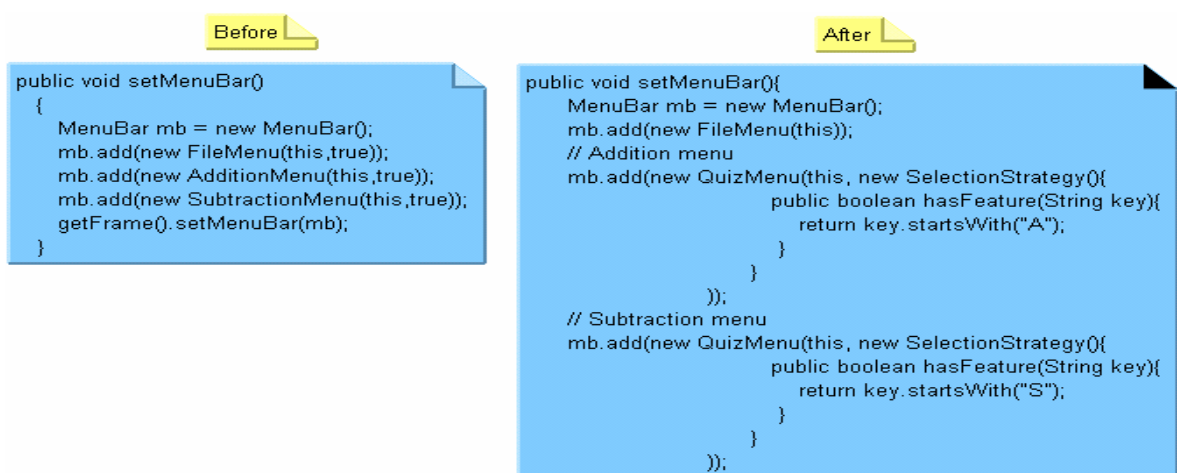


Figure 12. setMenuBar() method in User class before and after the modification

These changes enable the AdditionMenu and SubtractionMenu classes to become a newly created class: QuizMenu (figure 11). Each instantiation of the QuizMenu class is specialized using a strategy pattern. Strategy patterns decouple algorithms from clients. As a result, Strategy patterns allow the algorithms to vary independently and make them interchangeable. An instance of <SelectionStrategy> passed in the QuizMenu constructor determines a quiz type. Each SelectionStrategy object is made by means of an anonymous class. Figure 12 shows the comparison of two setMenuBar() methods in the former and latter *User* class used to create and set menu objects. At a glance, the latter looks complex. Yet in terms of flexibility QuizMenu class is more reusable than AdditionMenu and SubtractionMenu for the creation of different kinds of quiz menus. Consider if a new type of quiz menu needed to be added. The QuizMenu class can provide a new menu simply by calling its corresponding <SelectionStrategy> instance. The former way requires another new menu class that may only be slightly different from the others.

FileMenu class code has problems similar to the old AdditionMenu class. Here a *command pattern* is used to solve the problem. A *Command pattern* is an object behavioral pattern that allows complete decoupling between the user interface objects and the actions they initiate. Each subclass of <Command> in Figure 13 are extended from java.awt.MenuItem and implement the <Command> interface.

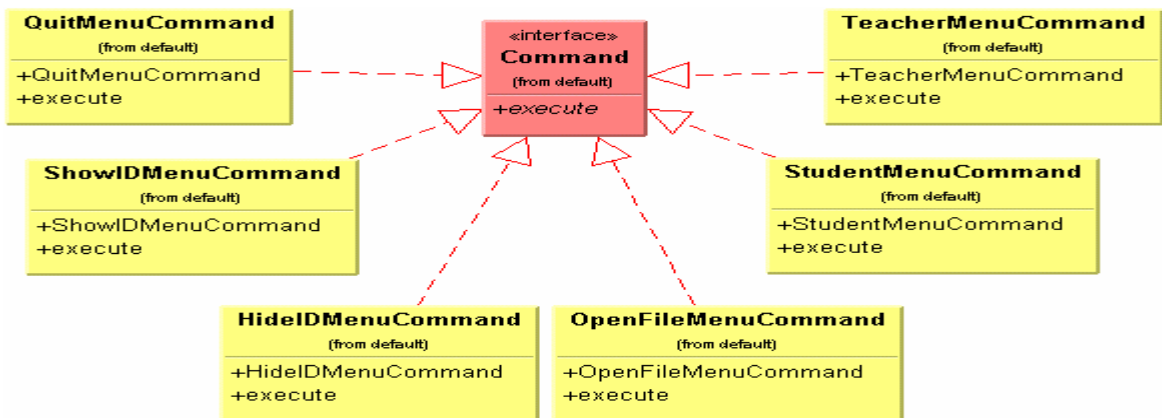


Figure 13. Class diagram of Command with its subclasses

Figure 14 shows settings for various menu items. As it can be seen, some menu commands have two parameters passed in their constructors. A “this” object is used in StudentMenuCommand, TeacherMenuCommand, to set an action listener. A currentUser is an instance of the User class whose services are needed for a specific task to be performed. The execute() method in each of the Command subclasses that is the interface of Command implemented to execute its own code when an action occurs on that object. The execute() interface is used to reduce the burden of the actionPerformed method. Since all instances of menu items added in the createMenu() method of FileMenu are a Command, an actionPerformed method can get an instance of command through the e.getSource() method call and perform cmd.execute() without knowing whose execute() method is being performed.

```

protected void createMenu()
{
    add(new StudentMenuCommand(this,currentUser));
    add(new TeacherMenuCommand(this,currentUser));
    add(new MenuItem("-"));
    add(new ShowIDMenuCommand(this,currentUser));
    add(new HideIDMenuCommand(this,currentUser));
    add(new MenuItem("-"));
    add(new OpenFileMenuCommand(this,currentUser));
    add(new MenuItem("-"));
    add(new QuitMenuCommand(this));
}

public void actionPerformed(ActionEvent e)
{
    Command cmd = (Command) e.getSource();
    cmd.execute();
}

```

Figure 14. createMenu() and actionPerformed() methods modified using Command pattern.

The initial design utilized only one *factory pattern*. A Factory method defines the interface for creating an object while retaining control of which class to instantiate. After this first refactoring, the program now consists of a combination of two *factory patterns* used in this modified program. A new Quiz class that is introduced in Figure 15 is a composition of Question objects. QuizFactory class is a factory of Quiz objects, which makes use of QuestionFactory to construct a Quiz object. The interaction of objects to produce a Quiz object using two factory method patterns is shown in Figure 16.

In response to a makeQuiz(2) method call from a client of the QuizFactory object, a QuestionFactory object instantiates a QuestionFactory object and sends a message *makeQuestion()* to it. Each time the QuestionFactory object responds to a message, it creates a Question and returns the question to the sender. When creation of all questions is complete, QuestionFactory creates a Quiz object with the questions and returns that object to the client.

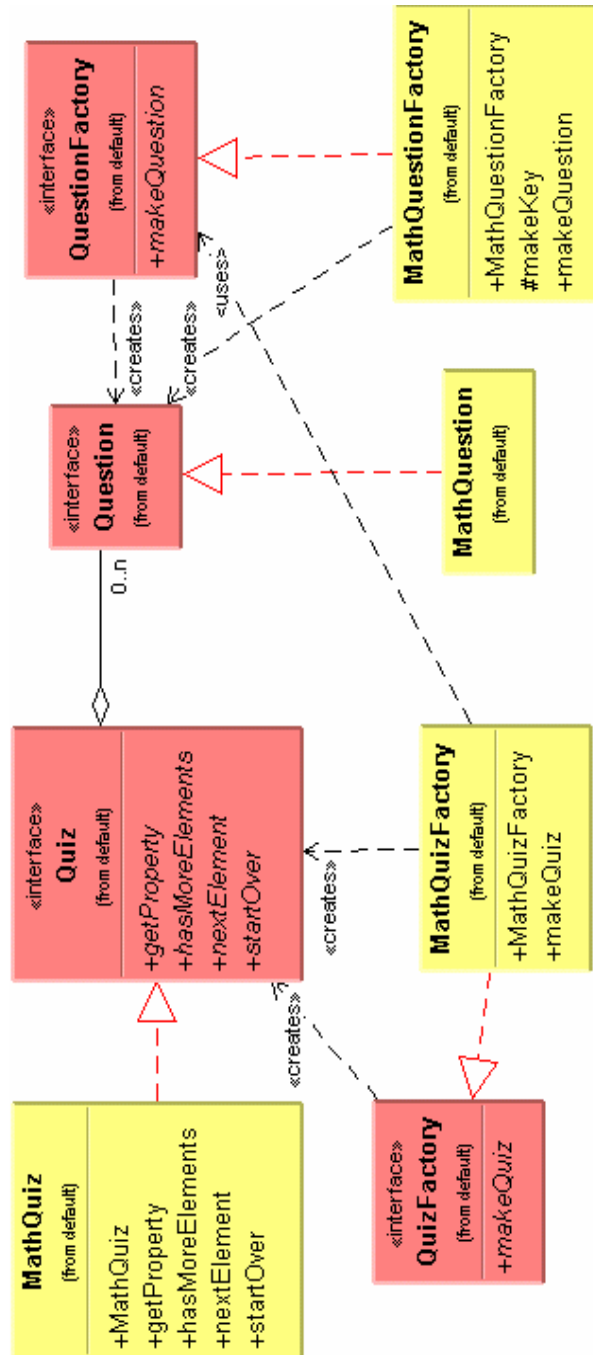


Figure 15. Class diagram of Quiz and Question with its factory

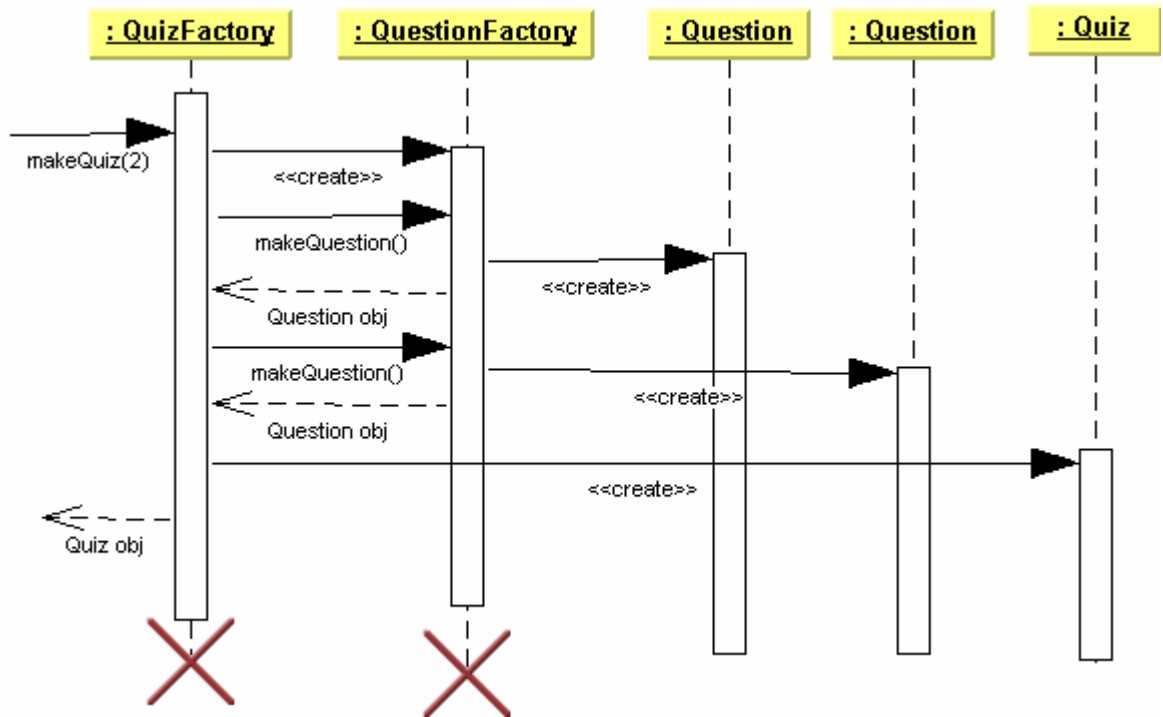


Figure 16. Interaction diagram demonstrating the creations of Quiz object

Although many parts of the program have been redesigned and refactored using several design patterns, there are still more pieces to refine. Several classes dealing with the graphical user interface (GUI) need modification. The Sheet class hierarchy could be completely redesigned. For instance, in the AnswerSheet class, there are too many private and protected methods to create GUI components. This suggests that this entity needs to be decomposed and split into several classes.

In the following 2 chapters, I will discuss the development stages from making a many designs as I could to the new addition to Basic Facts, which caused me to redesign the software. I will also show how the refactoring techniques will systematically improve the quality of software design and make the design clear allowing a design pattern to be applied more effectively.

Chapter 5

Development Episode

As Kerievsky (2002) mentioned in his draft, *Refactoring To Patterns*, there are two ways of using patterns: using them for up-front design and refactoring to patterns. Using the patterns for up-front design seems right choice, but all the hassle on the use of patterns may get wasted.

Why may patterns for up-front design tend not to work as intended? Designing object-oriented (OO) software is hard and it is harder to make it reusable. Crafting good OO software often requires several times of modifications for defining right-weighted classes and establishing key associations among them (Gamma, et al 1995). In another word, it takes a while for OO software design to mature. If the up-front design goes wrong, therefore, the design patterns used at early time has a risk of wasting development time.

When I first learned design patterns, they sounded a lot promising. So I tried hard to make use of as many patterns as possible in the first design. It took a long time to find a place where a certain pattern fit in my code. However, many of them were not worth usable as much time as I spent to apply them. Often time it made my design complicated: I ended up doing over-engineering.

I got confused with design patterns when this happened. I have learned design patterns because I wanted to be a good software engineer. While being frustrated with the result, I encountered Martin Fowler's book, *Refactoring: Improving the Design of Existing Code*. As mentioned in chapter two, it documents a rich catalog of methods of refactoring, each of which explains a common use for an improvement and the steps for making that improvement. This book changed the way I do programming drastically.

With following the catalog of different kinds of refactoring (merciless refactoring) and test-first programming, I refactored my code and then have started seeing things that I could not have found out before. As I refactored the code along with tests, my program got cleaner. The refined program started to show me another part of the code to refactor. And then the more refined program introduced the use of patterns naturally. The process of refactoring basically indicated to me the necessity of patterns in the right place at the right time: I started refactoring to patterns. "Instead

of thinking about a design that would work for every nuance of a system, test-first programming enabled me to make a primitive piece of behavior work correctly before evolving it to the next necessary level of sophistication.” Kerievsky (2002)

In an attempt to show “feeling” the power of merciless, disciplined, or systematic refactoring, the next chapter deals in detail a step-by-step refactoring with several case study examples that I encountered during the second design of Basic Facts.

Chapter 6

Modified Prototype

6.1 Modified Design Attempt

The first design of Basic Facts met the requirement found in the analysis phase. However, as mentioned before, it was not flexible enough to evolve and grow. In addition there were several problems with the first design.

One problem of the first design is that it was difficult to add a new type of a quiz: specifically multiplication and division. The algorithm of quiz creation in the first version could not accommodate a set of quizzes with many restrictions. For instance, x/y where x and y are interchangeable and the answer of x/y must be 0,2,4,8. That turned out to be more complicated than predicted.

In addition, many objects such as **User** and **AnswerSheet** had more responsibilities than they should have. They were implemented as a mixture of user interface code and the domain logics in one class. This complicated the entire system and made it harder to add new functionality. Therefore, the user interface code needed to be separated from the domain logic. These conflicts led me to redesign the system.

I chose to recreate the project from scratch, using the first design rather than refactoring. I did this because I came up with the first design without knowing much about object-oriented programming. My old mind set, which is procedural programming, got in the way. The class diagram for the second Basic Facts design is shown in Figure 17.

In the second design quizzes are created using a formatted file that contains quiz properties rather than using an algorithm to create a specific quiz. The format is shown in Table 1. In this manner it is much easier to create specific quizzes with more control. Although there is overhead requiring that all questions be created in the list, this approach simplifies the design drastically.

This change made the construction of Quiz systematic: as long as the format is followed, **QuizMenuProperty** can remain unchanged. **QuizPropertyReader** reads in data stored in the formatted quiz file, and **BasicProblemParser** parses the extracted data and stores it in **QuizProperty** objects. With this quiz construction process, any type of quiz may be constructed so long as it follows the defined format.

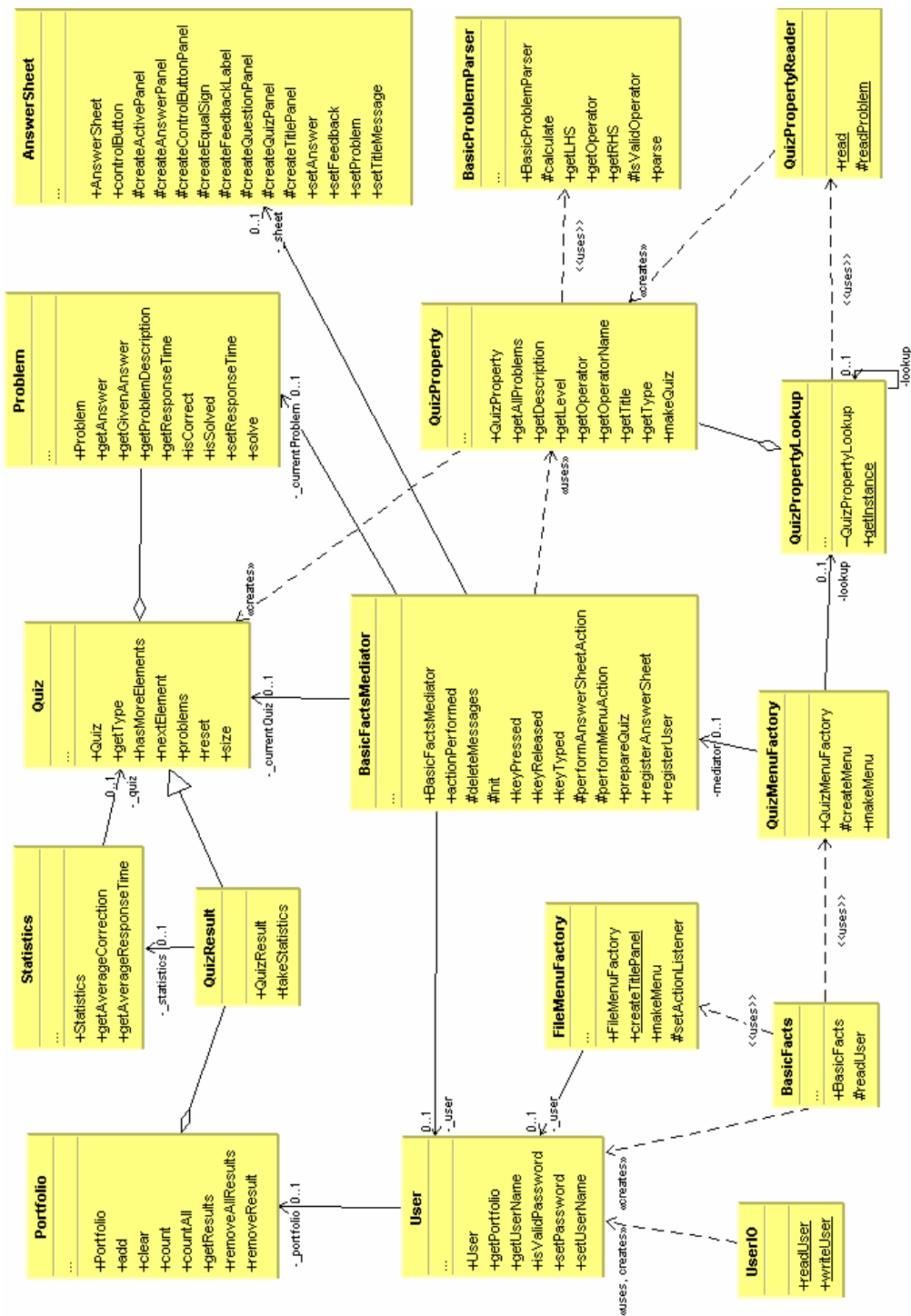


Figure 17. Second design of basic facts

Each line contains specific information described below:

1. An type of quiz (e.g. A1, M2 or S1)
2. The level of the quiz. Let's say, if it belongs to easy math, put 1, if it is hard math, put 2.
3. The operator of the quiz. (e.g. +, -, *, and /)
4. The name of the operator. (e.g. addition, subtraction, and so on)
5. The title of the quiz.
6. The description of the quiz. This is optional. If you do not want to enter a description, leave this line blank.
7. A list of all of the problems where each them is separated by a semi-colon (;). Problems can be listed all on one line or multiple lines. Do not leave blanks between lines. Each line from line 7 to below must have at least one problem.

Example:

```
==== A page starts here =====  
A1  
1  
+  
Addition  
Count on and zero generalization  
  
1+1;2+3;  
3+1;3+2;4+3;  
5+5;  
5+2;6+2;  
===== End of page =====
```

Table 1. The format of a quiz property

The names of many classes have remained the same as in version 1 of Basic facts, but their behaviors have been changed. **Problem** class is simplified version of the **MathQuestion** class implemented in the first design. An instance of **Problem** class holds the question of a problem and its answer as String. It also records the answer and the response time given by the student. My first goal of the second design was to simplify the design, so the interface <Question> is removed and the **Problem** class is introduced as a concrete class.

Quiz class is a collection of **Problem** objects stored in an instance of Vector. It implements <Enumeration> to iterate through the collection. The problems method returns the Vector. The reset method is defined to reset an exhausted iteration back to the beginning, so a student can iterate through the collection again. **QuizResult** is a subclass of **Quiz** class. The takeStatistics method returns an instance of **Statistics** class that defines the computation of average correction rate and response time for all the **Problem** objects held in an instance of **Quiz**.

Portfolio class works as a folder that holds the history of quizzes solved by a user. It defines several methods for adding, removing, and displaying a **QuizResult** instance. In Portfolio class, the clear method delegates to a Hashtable instance variable which maps a quiz type to a Vector instance. This vector holds a collection of the portfolios **QuizResult** instances. The count method returns the number of specific types of **QuizResult** instances dependent upon the quiz type passed in as a parameter. The method countAll returns the total number of quizzes in a **Portfolio** object. The getResults method returns as a list a given type of **QuizResults** as List type. The removeAllResults method remove all instances a given type of QuizResult class, whereas removeResult method removes a given individual instance.

This **Portfolio** class is defined to keep a record of quizzes taken, provide a method to gather a specific set of **Problems** solved and compute the corresponding statistics. Yet in order to solve the computation, Portfolio's client needs access to several other classes for example, **Statistics** and **QuizResult**. In a subsequent section, a composite pattern will be introduced and applied to **Problem-Quiz-Portfolio** hierarchy to simplify the computation.

User class defines a user that has a **Portfolio** instance and stores his name and password. The first design of the **User** class was implemented as a subclass of java.awt.Window. This version of User class is much simpler and represents only the logic of a user.

UserIO is a utility class that reads an instance of **User** objects and writes to a file using the java.io.**ObjectInputStream/ObjectOutputStream** class. The first design of Basic Facts stored a student's work in the file. This change was implemented to extend usability so a user object could be passed through the network to store in and retrieve from a server if needed in future use.

BasicFacts class subclasses javax.swing.JFrame class and conducts all creations of objects such as JMenuBar delegating to FileMenuFactory and QuizMenuFactory instances.

The **BasicFactsMediator** class coordinates the interactions of various visual components (buttons and text fields) and data models (Quiz, Problem, User). Even though, there are only a few visual objects, the interactions between the visual controls tends to be rather complex. This is because each visual object needs to know about other visual objects in order to update the related visual contents and data models dynamically. A *Mediator pattern* simplifies such a system by the creation of a mediator which is the only class that is aware of the other classes in the system.

This keeps the GUI components from referring to each other explicitly and isolates interactions into the only one instance of **BasicFactsMediator**.

Each control component the Mediator communicates with is called a colleague. **AnswerSheet** is a set of colleagues (gui components and quizzes) that provides access to each colleague. AnswerSheet class is, in a sense, a Facade pattern. Its intent is to simplify the complexity of the sub system by providing a simplified interface to other subsystems. AnswerSheet eases the communication of the visual components by providing a number of simple methods. This helps simplify the implementation of **BasicFactsMediator**.

The **QuizPropertyLookup** class holds instances of the **QuizProperty** class. Whenever the instance is created, the getInstance method returns only one instance of QuizPropertyLookup which keeps the global point of access to the object. This **QuizPropertyLookup** has changed very little since the last version of Basic facts.

6.2 Systematic Refactoring

After completion of the second design, several refactoring techniques documented in Fowler's book (2000) were applied to restructure the code. These refactoring techniques are well disciplined and easy to follow but indispensable in restructuring existing code. Unlike the refactoring in the first design, the refactoring applied to the second design of Basic Facts was performed in a strict manner. The following three sub sections will show how the process of refactorings enhanced the quality of the software. The first two sub sections will show changes to the codes step by step. The last will show the evolution of design through class diagram.

```
public class BasicFacts extends JFrame {
    public static Dimension defaultFullScreenSize = Toolkit.getDefaultToolkit().getScreenSize();
    public final static int width=defaultFullScreenSize.width*3/5;
    public final static int height=defaultFullScreenSize.height*3/5;
    public static Dimension defaultScreenSize = new Dimension(width,height);
    public static Point centerPoint = new Point(defaultFullScreenSize.width/2-width/2,
                                                defaultFullScreenSize.height/2-height/2);

    public BasicFacts(){
        User currentUser          = readUser();
        AnswerSheet ansSheet     = new AnswerSheet(defaultScreenSize);
        BasicFactsMediator mediator = new BasicFactsMediator(this);
        mediator.registerAnswerSheet(ansSheet);
        mediator.registerUser(currentUser);
        QuizMenuFactory qmFactory = new QuizMenuFactory(mediator);
        FileMenuFactory fmFactory = new FileMenuFactory(currentUser,this);
        Enumeration menus = qmFactory.makeMenu();
        JMenuBar mb = new JMenuBar();
        mb.add(fmFactory.makeMenu());
        while(menus.hasMoreElements()){
            JMenu m = (JMenu) menus.nextElement();
            mb.add(m);
        }

        setJMenuBar(mb);
        setLocation(centerPoint);
        setSize(defaultScreenSize);
        repaint();
        setVisible(true);
    }
    ...
}
```

Table 2. BasicFacts class before Refactoring

6.2.1 Refactoring to clean up codes

Basic facts class shown in Table 2 contained too many public static fields such as defaultFullScreenSize. Most of them seemed to be used locally. To find out who had

access to the public static fields, their access modifiers were changed to private and then all collaborating programs are compiled starting from the root class, BasicFactsApp.java as shown in Table 3.

```
$ javac BasicFactsApp.java
.\BasicFactsMediator.java:117: defaultScreenSize has private access in BasicFact
S
    _board.setSize(BasicFacts.defaultScreenSize);
                        ^
1 error
```

Table 3. Step to find who has access to the public static fields in BasicFacts class

The result indicated that the line 117 in BasicFactsMediator class attempted to access the private field defaultScreenSize as public (Table 3). As suspected, only one public static field, defaultScreenSize was accessed as intended, but the others were unnecessarily publicized. With this in my mind, the modifiers of the fields in basic facts class were changed back to public. Next I applied the *replace temp with query (120)* refactoring technique to the defaultScreenSize variable. In *Replace temp with query (120)*, *temp* refers to a temporary local variable, which is assigned a simple expression. Sometimes this temp gets in the way of other refactoring.

The problem with temps is that they are temporary and local. They can be seen only in the context of the method in which they are used; temps tend to encourage longer methods. Temps also opt to keep programmers from seeing duplicate codes. By replacing the temp with a query method, any method in the class can get to the information. This results in cleaner code for the class. Even with this issue, *Replace Temp with Query (120)* often is a vital step before using *Extract Method (110)* refactoring. *Extract Method (110)* is another refactoring technique that takes a clump of code and turns it into its own method. The use of local variables can make it difficult to extract the code, so it is better to replace as many local variables as possible with queries.

After the changes were made (Table 4) I compiled and tested the code. I then replaced all the instance of defaultScreenSize variable in the BasicFacts class and the BasicFactsMediator class with getFrameSize() method. After compiling and testing I could now apply *replace temp with query(120)* to the centerPoint variable and refactor again. Eventually I was able to eliminate all public static fields as shown below in Table 5:

```

public class BasicFacts ...
    public static Dimension defaultFullScreenSize = Toolkit.getDefaultToolkit().getScreenSize();
    public final static int width=defaultFullScreenSize.width*3/5;
    public final static int height=defaultFullScreenSize.height*3/5;
    public static Dimension defaultScreenSize = getFrameSize();
    public static Point centerPoint = new Point(defaultFullScreenSize.width/2-width/2,
                                                defaultFullScreenSize.height/2-height/2);

    public BasicFacts(){
        ...
    }

    public static Dimension getFrameSize(){
        Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
        screenSize.setSize(screenSize.width*3/5,screenSize.height*3/5);
        return screenSize;
    }
    ...

```

Table 4. BasicFacts after *replace temp with query* performed on defaultScreenSize field

```

Public class BasicFacts ...
    public BasicFacts(){
        User currentUser = readUser();
        AnswerSheet ansSheet = new AnswerSheet(getFrameSize());
        BasicFactsMediator mediator = new BasicFactsMediator(this);
        mediator.registerAnswerSheet(ansSheet);
        mediator.registerUser(currentUser);
        QuizMenuFactory qmFactory = new QuizMenuFactory(mediator);
        FileMenuFactory fmFactory = new FileMenuFactory(currentUser,this);
        Enumeration menus = qmFactory.makeMenu();
        JMenuBar mb = new JMenuBar();
        mb.add(fmFactory.makeMenu());
        while(menus.hasMoreElements()){
            JMenu m = (JMenu) menus.nextElement();
            mb.add(m);
        }

        setJMenuBar(mb);
        setLocation(getCenterPoint());
        setSize(getFrameSize());
        repaint();
        setVisible(true);
    }

    protected Point getCenterPoint(){
        Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
        return new Point((screenSize.width-getFrameSize().width)/2,
                        (screenSize.height-getFrameSize().height)/2);
    }
    public static Dimension getFrameSize(){
        ...
    }
    ...

```

Table 5. BasicFacts after eliminating all public static fields

The end result is that each method reveals its intent without referring to actual code in it. Now the code was ready for *Extract Method (110)* refactoring. But, before taking that step, there were still three local variables in the constructor that needed to be dealt with. I turned the local variables: `currentUser`, `ansSheet` and `mediator` into fields as described on Table 6:

```
public class BasicFacts ...
    private User currentUser;
    private AnswerSheet ansSheet;
    private BasicFactsMediator mediator;
    public BasicFacts(){
        currentUser = readUser();
        ansSheet = new AnswerSheet(getFrameSize());
        mediator = new BasicFactsMediator(this);
        mediator.registerAnswerSheet(ansSheet);
        mediator.registerUser(currentUser);
        QuizMenuFactory qmFactory = new QuizMenuFactory(mediator);
        FileMenuFactory fmFactory = new FileMenuFactory(currentUser,this);
        Enumeration menus = qmFactory.makeMenu();

        JMenuBar mb = new JMenuBar();
        mb.add(fmFactory.makeMenu());
        while(menus.hasMoreElements()){
            JMenu m = (JMenu) menus.nextElement();
            mb.add(m);
        }

        setJMenuBar(mb);
        setLocation(getCenterPoint());
        setSize(getFrameSize());
        repaint();
        setVisible(true);
    }
    ...
```

Table 6. BasicFacts after changing three local variables into fields

Then after applying *Extract Method (110)*, a sizeable chunk of the code turned into a method whose name clearly indicated its function: `setupMenuBar()`. There are several reasons to use extract method refactoring to refine your code:

- 1) It generates well-named methods and well-named methods increase the chances of method reuseability.
- 2) It created finely grained methods. Thus making the code more programmer-friendly.
- 3) It allows the higher-level methods to read more like a series of comments.
- 4) Overriding is easier when the methods are finely grained.

This is achieved by first applying the *Extract method (110)* refactoring technique to the code of `JMenuBar` Construction as shown Table 7:

```

public class BasicFacts ...

public BasicFacts(){
    currentUser = readUser();
    ansSheet = new AnswerSheet(getFrameSize());
    mediator = new BasicFactsMediator(this);
    mediator.registerAnswerSheet(ansSheet);
    mediator.registerUser(currentUser);
    setupJMenuBar();
    setLocation(getCenterPoint());
    setSize(getFrameSize());
    repaint();
    setVisible(true);
}

protected void setupJMenuBar(){
    QuizMenuFactory qmFactory = new QuizMenuFactory(mediator);
    FileMenuFactory fmFactory = new FileMenuFactory(currentUser,this);
    Enumeration menus = qmFactory.makeMenu();

    JMenuBar mb = new JMenuBar();
    mb.add(fmFactory.makeMenu());
    while(menus.hasMoreElements()){
        JMenu m = (JMenu) menus.nextElement();
        mb.add(m);
    }

    setJMenuBar(mb);
}
...

```

Table 7. BasicFacts after *Extract Method* applied

Next in Table 8, *Extract method (I10)* is applied to the method that set up the mediator code.

```

public class BasicFacts ...

public BasicFacts(){
    currentUser = readUser();
    ansSheet = new AnswerSheet(getFrameSize());
    mediator = new BasicFactsMediator(this);
    setupMediator();
    setupJMenuBar();
    setLocation(getCenterPoint());
    setSize(getFrameSize());
    repaint();
    setVisible(true);
}

protected void setupMediator(){
    mediator.registerAnswerSheet(ansSheet);
    mediator.registerUser(currentUser);
}
...

```

Table 8. BasicFacts after another *Extract Method* applied

The resulting code is more descriptive and easier to understand. If at a later time there is a need to add a new functionality or modify the code, it will be much easier to pin point where a new feature should be added or how to modify the existing code to accommodate revision. For example, if there is a need for the creation of menus to be decoupled from BasicFacts class, then the setupJMenubar() method is the place to change.

6.2.2 Refactoring Creation of Quiz: refactoring to factory pattern

At this point, the **QuizProperty** class handles multiple responsibilities (Fig 18 top). Along with maintaining the property of a quiz, it also has an extra feature that creates an instance of **Quiz** class: This means **QuizProperty** has taken over a factory job. When the class was first defined, it was thought that grouping related data and operations together would work well. Several weeks later I returned to this code, and realized that the class name did not represent its entire responsibility. This problem could be solved by splitting this class into two pieces. By using the *factory method* pattern and then applying the *Extract Class (149)* refactoring technique, makeQuiz() method for the creation of **Quiz** instance is going to be taken out of **QuizProperty** and moved to **QuizFactory** class where it should belong. Table 9 shows **QuizProperty** class before any refactoring was started.

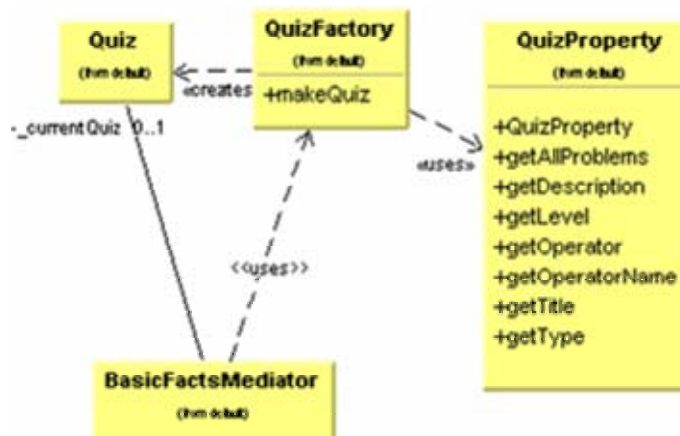
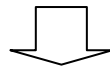
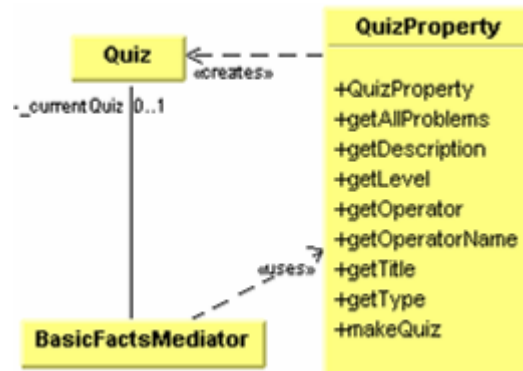


Figure 18. The comparison of before and after refactoring to QuizProperty class.

```

public class QuizProperty{
    private String _type;
    private int _level;
    private String _operator;
    private String _nameOfOperator;
    private String _title;
    private String _description;
    private List _problem;

    public QuizProperty(String type,
                        int level,
                        String operator,
                        String nameOfOperator,
                        String title,
                        String description,
                        List problem){
        _type = type;
        _level = level;
        _operator = operator;
        _nameOfOperator = nameOfOperator;
        _title = title;
        _description = description;
        _problem = problem;
    }

    public String getType(){return _type;}
    public int getLevel(){return _level;}
    public String getTitle(){return _title;}
    public String getOperator(){return _operator;}
    public String getDescription(){return _description;}
    public List getAllProblems(){return _problem;}
    public String getOperatorName(){return _nameOfOperator;}

    public Quiz makeQuiz(int count) throws IllegalArgumentException{
        if (count > _problem.size())
            throw new IllegalArgumentException("The number of problems in quiz must be less than "
            + _problem.size());

        Random rnd = new Random();
        Vector quizChosen = new Vector();
        Hashtable quizLooked = new Hashtable();
        BasicProblemParser parser = new BasicProblemParser();
        try{
            while (quizChosen.size() < count){
                int i = rnd.nextInt(_problem.size());
                String prob = (String)_problem.get(i);
                if (!quizLooked.containsKey(prob)){
                    quizLooked.put(prob,prob);
                    parser.parse(prob);
                    quizChosen.add( new Problem( prob,parser.calculate()));
                }
            }
        }catch(IllegalArgumentException iae){
            throw new IllegalArgumentException("The quiz type "+_type+" has an illegal format. & "+iae);
        }
        return new Quiz(this.getType(), quizChosen);
    }
}

```

Table 9. QuizProperty class before refactoring

As the first step of refactoring to factory pattern, there is a need to introduce a new class **QuizFactory** to express the split-off responsibilities. I then implemented **QuizFactory** as a stub class shown in Table 10 and then compiled. Note that there is a QuizProerty parameter added to makeQuiz() definition in QuizFactory class. The reason is that when all the creation of **Quiz** are moved from **QuizProperty** to QuizFactory, **QuizFactory** needs to know the type of quizzes it is constructing.

```
public class QuizFactory{
    public Quiz makeQuiz(QuizProperty property, int count) throws IllegalArgumentException{
        return null;
    }
}
```

Table 10. Initial QuizFactory class

Now using *Self Encapsulate Field (171)* refactoring, all of the problem field variables used in the makeQuiz() method of QuizProperty class were replaced with their equivalent getter method, getAllproblems() as shown in Table 11.

```
...
public Quiz makeQuiz(int count) throws IllegalArgumentException{
    if (count > this.getAllProblems().size())
        throw new IllegalArgumentException("The number of problems in quiz must be less than
+"this.getAllProblems().size());
    Random rnd = new Random();
    Vector quizChosen = new Vector();
    Hashtable quizLooked = new Hashtable();
    BasicProblemParser parser = new BasicProblemParser();
    try{
        while (quizChosen.size() < count){
            int i = rnd.nextInt(this.getAllProblems().size());
            String prob = (String)this.getAllProblems().get(i);
            if (!quizLooked.containsKey(prob)){
                quizLooked.put(prob,prob);
                parser.parse(prob);
                quizChosen.add( new Problem( prob,parser.calculate()));
            }
        }
    }catch(IllegalArgumentException iae){
        throw new IllegalArgumentException("The quiz type "this.getType "'s problem is in an illegal
format. & "+iae);
    }
    return new Quiz(this.getType(), quizChosen);
}
```

Table 11. makeQuiz method in QuizProperty class after *self encapsulate field* refactoring

I added a ‘this’ reference to each of the getter method calls to make it easier to change later when *Move Method (142)* would be applied in the QuizMenu class. Finally, the code is moved in the `makeQuiz()` method in the QuizFactory class and the required parameter list is updated for `makeQuiz`.

```

public class QuizFactory{

    public Quiz makeQuiz(QuizProperty property, int count) throws IllegalArgumentException{
        if (count > property.getAllProblems().size())
            throw new IllegalArgumentException("The number of problems in quiz must be less
than "+property.getAllProblems().size());
        Random rnd = new Random();
        Vector quizChosen = new Vector();
        Hashtable quizLooked = new Hashtable();
        BasicProblemParser parser = new BasicProblemParser();
        try{
            while (quizChosen.size() < count){
                int i = rnd.nextInt(property.getAllProblems().size());
                String prob = (String)property.getAllProblems().get(i);
                if (!quizLooked.containsKey(prob)){
                    quizLooked.put(prob,prob);
                    parser.parse(prob);
                    quizChosen.add( new Problem( prob,parser.calculate()));
                }
            }
        }catch(IllegalArgumentException iae){
            throw new IllegalArgumentException("The quiz type "+property.getType "'s problem is
in an illegal format. & "+iae);
        }
        return new Quiz(property.getType(), quizChosen);
    }
}

```

Table 12. QuizFactory class after *Move Method* applied

Next, construction of a **Quiz** in **QuizProperty** is updated to be delegated to an instance of QuizFactory as shown in Table 13:

```

public class QuizProperty ...
    public Quiz makeQuiz(int count) throws IllegalArgumentException {
        return new QuizFactory().makeQuiz(this,count);
    }
...

```

Table 13. QuizProperty delegates the construction of Quiz to QuizFactory

Again, I compiled and tested the QuizProperty class. Now, the makeQuiz method of QuizFactory class is ready to be used, instead of the makeQuiz method of QuizProperty class. This means that we need to change all the corresponding references in the code. For example, change a passing parameter in PrepareQuiz() method call in BasicFactsMediator class as illustrated in table 14.

```

BasicFactsMediator class ...
...
protected void performMenuAction(JMenuItem item){
    String quizType = item.getActionCommand();
    QuizPropertyLookup lookup= QuizPropertyLookup.getInstance();
    QuizProperty qp=(QuizProperty) lookup.get(quizType);
    PrepareQuiz(qp, new QuizFactory().makeQuiz(qp,5));
    _board.getContentPane().removeAll();
    _board.getContentPane().add(_sheet);
    _board.setSize(BasicFacts.getFrameSize());
    _board.setVisible(true);
}
...

```

Table 14. Change made to BasicFactsMediator class

After the changes were made, the code was compiled and tested. Next, the makeQuiz() method in the QuizProperty class was removed. See the class diagram on the bottom in figure 18.

6.2.3 Refactoring the Problem-Quiz-Portfolio hierarchy: refactoring triggers the use of composite pattern

In the current design, **QuizResult** is not doing much work: there is a slight difference between **QuizResult** class and **Quiz** class. To compute a quiz result, (compute average response time or average number of correct response) **QuizResult** holds an instance of statistics class who actually performs the work. The statistics class's responsibilities can be integrated into **Quiz** object. We can accommodate this integration by using another refactoring technique known as *Inline class (154)*. *Inline Class (154)* is used to fold the **QuizResult** class into **Quiz**. *Inline Cass (154)* works the opposite of *Extract Class (149)*. When a certain class does not have a reason for its existence, it needs to be folded into another class that can accommodate that responsibility. Figure 19 illustrates the association between the **Portfolio**, **Statistics**, **QuizResult**, **Quiz**, and **Problem** classes.

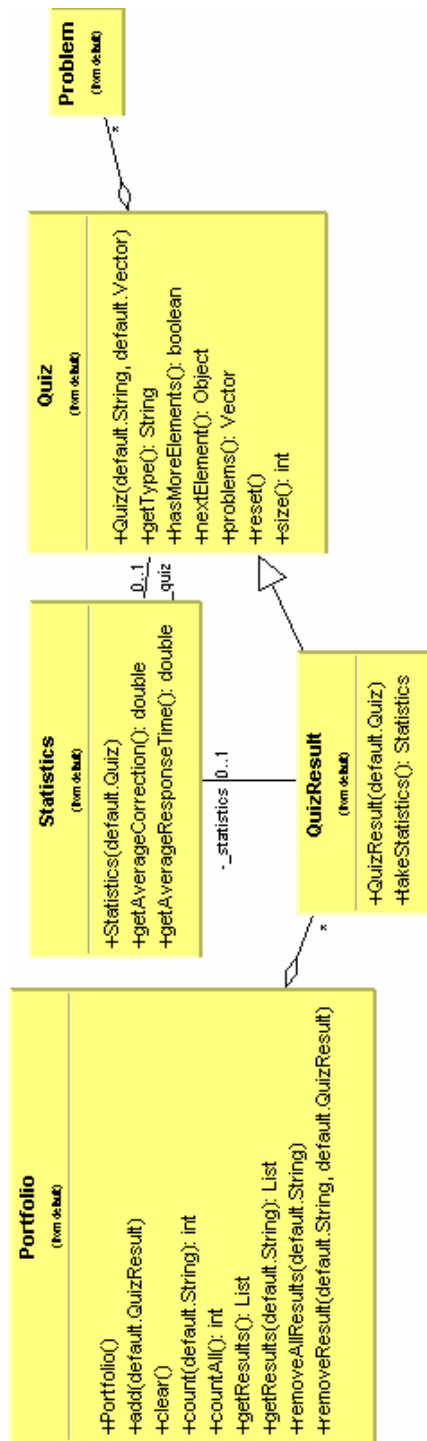


Figure 19. Class diagram for Portfolio, Statistics, QuizResult, Quiz, and Problem classes.

To prepare **Quiz** class to take on the responsibilities of **QuizResult**, getAverageCorrection() and getAverageResponseTime() methods are defined in the **QuizResult** class by using *Move Method(142)* refactoring as shown in Figure 20. Now **QuizResult** delegates to an instance of **Statistics** class for the computation of those values. Next I made sure that all calling objects sent a message to the **Statistics** object instead of calling the takeStatistics() method in **QuizResult** and receiving an instance of **Statistics** for the average response time and correction for a target **Quiz** object. Changing the access modifier of the takeStatistics() method to private ensures other objects are not using the method. Now the **QuizResult** class hides **Statistics** objects from **Portfolio** and other classes that may need the statistic information. Figure 20 shows the result of the refactoring to this point.

By applying *Move Field (146)* refactoring, the instance of **Statistics** is transferred from the **QuizResult** class to the **Quiz** class. Then move the takeStatistics() method to **Quiz** by again using *Move Method(142)*, and make takeStatistics() public so that the **QuizResult** class can delegate the work of statistics to **Quiz**. After the changes are made the code is compiled and tested. Finally I moved the remaining two methods getAverageCorrection() and getAverageResponseTime() out of **QuizResult** to **Quiz** and changed all the parameters and variables that were specified from **QuizResult** type to **Quiz**. The code was compiled and tested. Then I could safely remove **QuizResult**. The class diagram in figure 21 reveals the new class relations with **QuizResult** removed.

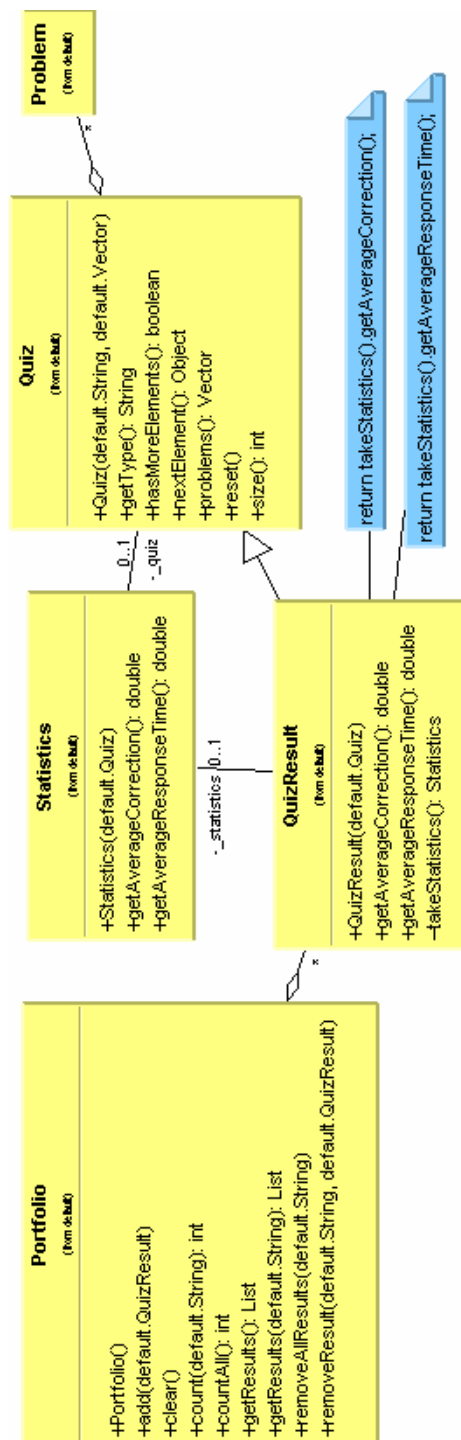


Figure 20. The class diagram for Portfolio-Quiz-Statistics-QuizResult relationship after *Move Method*

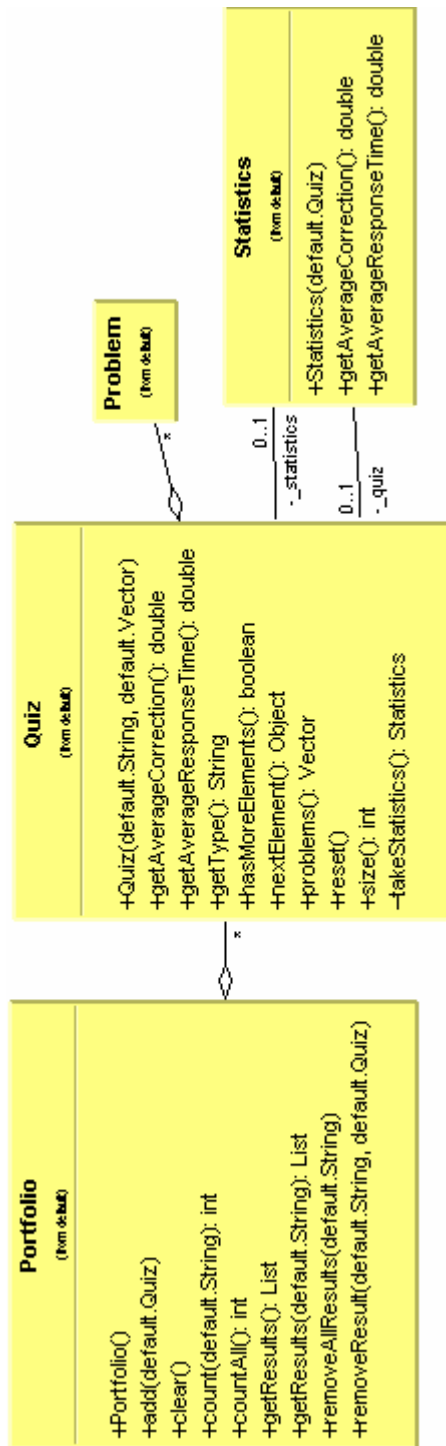


Figure 21. Class diagram after QuizResult class removed

The end result clearly shows that the three classes **Portfolio**, **Quiz** and **Problem** reside in the same hierarchy: **Portfolio** has instances of quizzes which in turn hold **Problem** objects. This relationship naturally triggers the use of composite patterns. *Composite pattern* arranges objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. Actual tests or exams are considered to represent a composite relationship. One quiz may contain one or more quizzes. This is the perfect place to apply a composite pattern.

The first quiz composite structure was implemented as shown in figure 22. **QuizComponent** defines all methods needed for **QuizLeaf** and **QuizComposite** that in turn subclass the super class and specialize necessary behaviors.

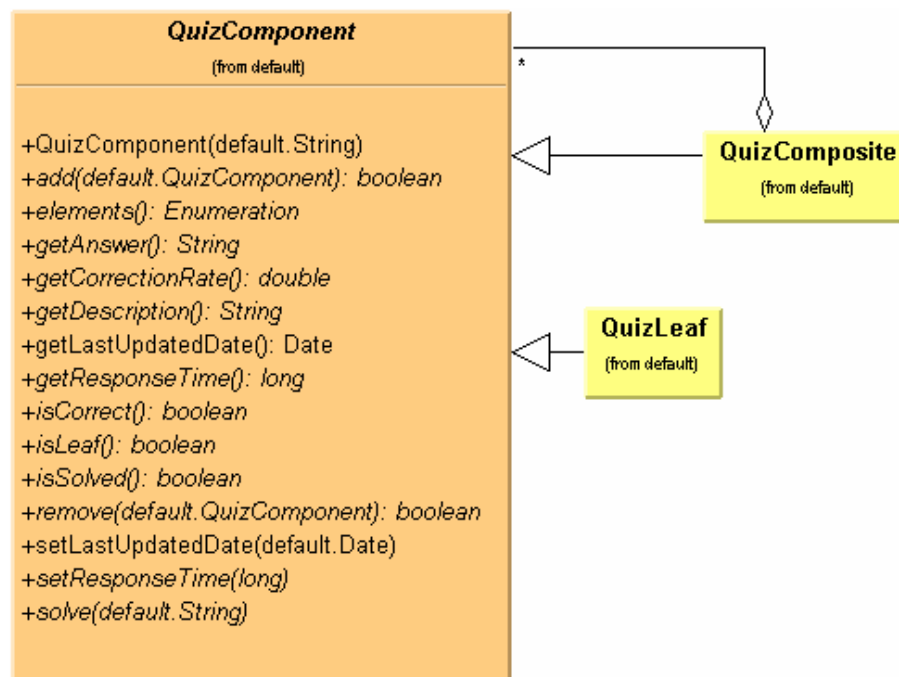


Figure 22. Quiz Composite class diagram

In order to accommodate the quiz composite into the system, Adaptor pattern is applied. The intent of Adaptor pattern is to provide the interface a client expects, using the services of a class with a different interface (Gamma et al. 1995). Using Adaptor pattern, I made **Problem** a subclass of **QuizLeaf** as shown figure 23.

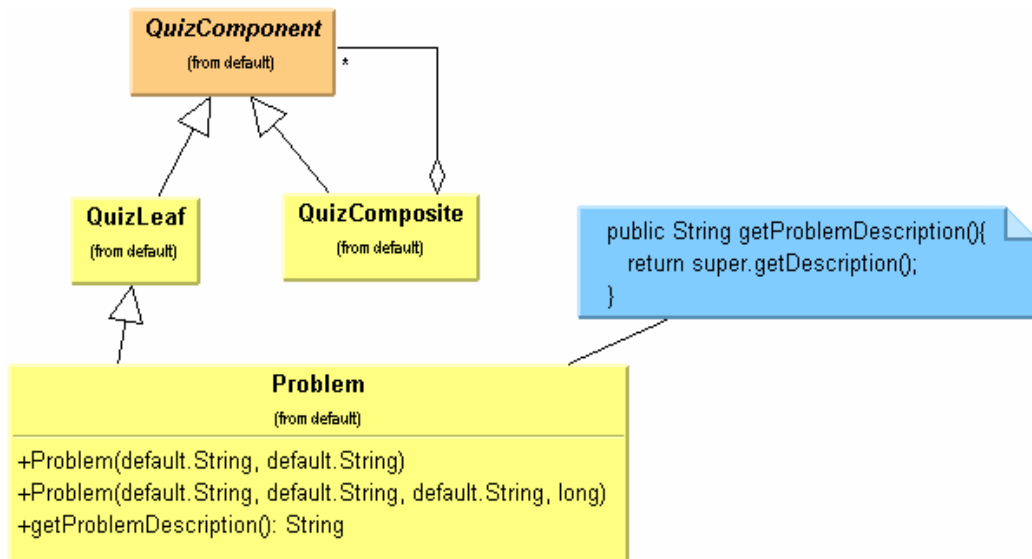


Figure 23. After adaptor pattern applied to make Problem a subclass of QuizLeaf

I replaced the `getProblemDescription()` method with `getDescription`. In order to control who calls `getProblemDescription`, I made it private and compiled as shown in Table 15 (of course, I deleted all class files before hand).

```

> rm *.class
> javac BasicFactsApp.java
.\BasicFactsMediator.java:86: getProblemDescription() has private access in Problem
    _sheet.setProblem(_currentProblem.getProblemDescription());
                                   ^
1 error
> rm *.class
> javac BasicFactsApp.java
>
  
```

Table 15. Compilation to check who accesses `getProblemDescription()` in Problem class

Only one `getProblemDescription()` method call had taken place. This method call was then replaced with `getProblem`. I removed the definition of the `getProblemDescription()` method from the Problem class. Then I compiled and tested. Now all **Problem** types can be replaced with **QuizLeaf** if needed. There is no reason to change, so I decided to leave it as it is.

In the same manner, I applied Adaptor pattern to **Quiz** class. The result is shown in Figure 24.

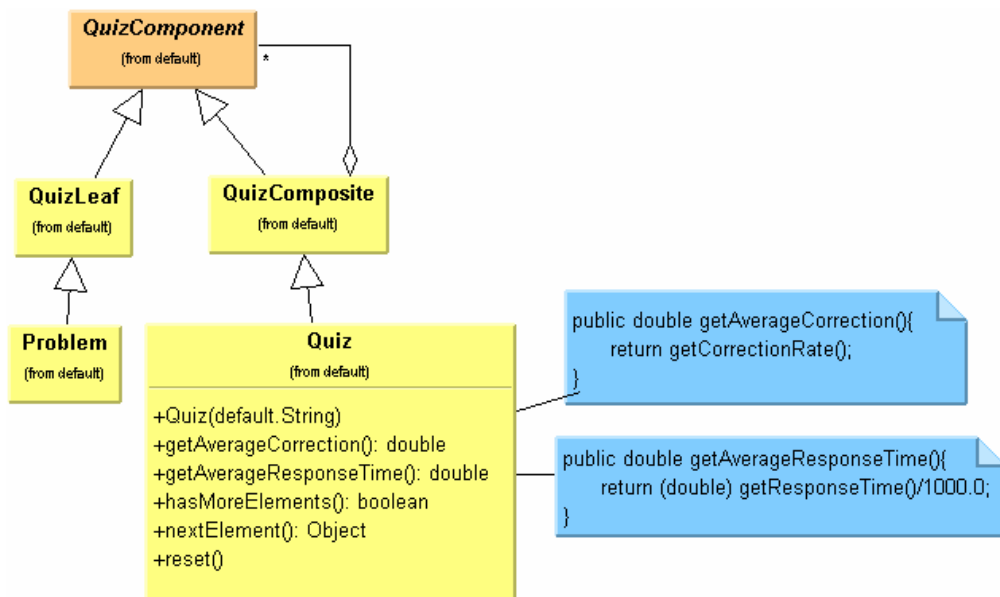


Figure 24. Quiz class with the composite after adapted

Continuing on in the same manner, **Portfolio** is subclassed to **QuizComposite** and replaced the old **Portfolio** class. The end result is shown in Figure 25.

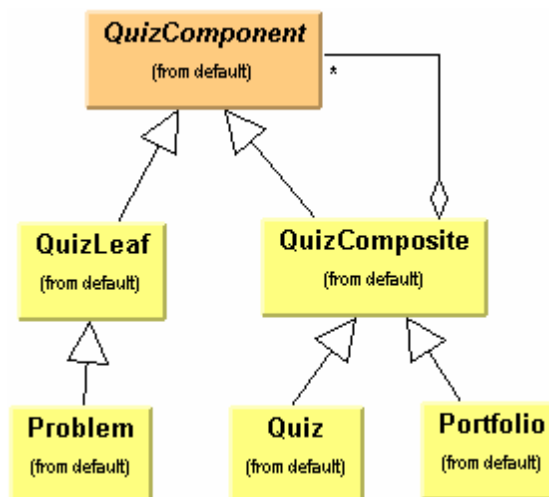


Figure 25. Portfolio class is adapted to QuizComposite

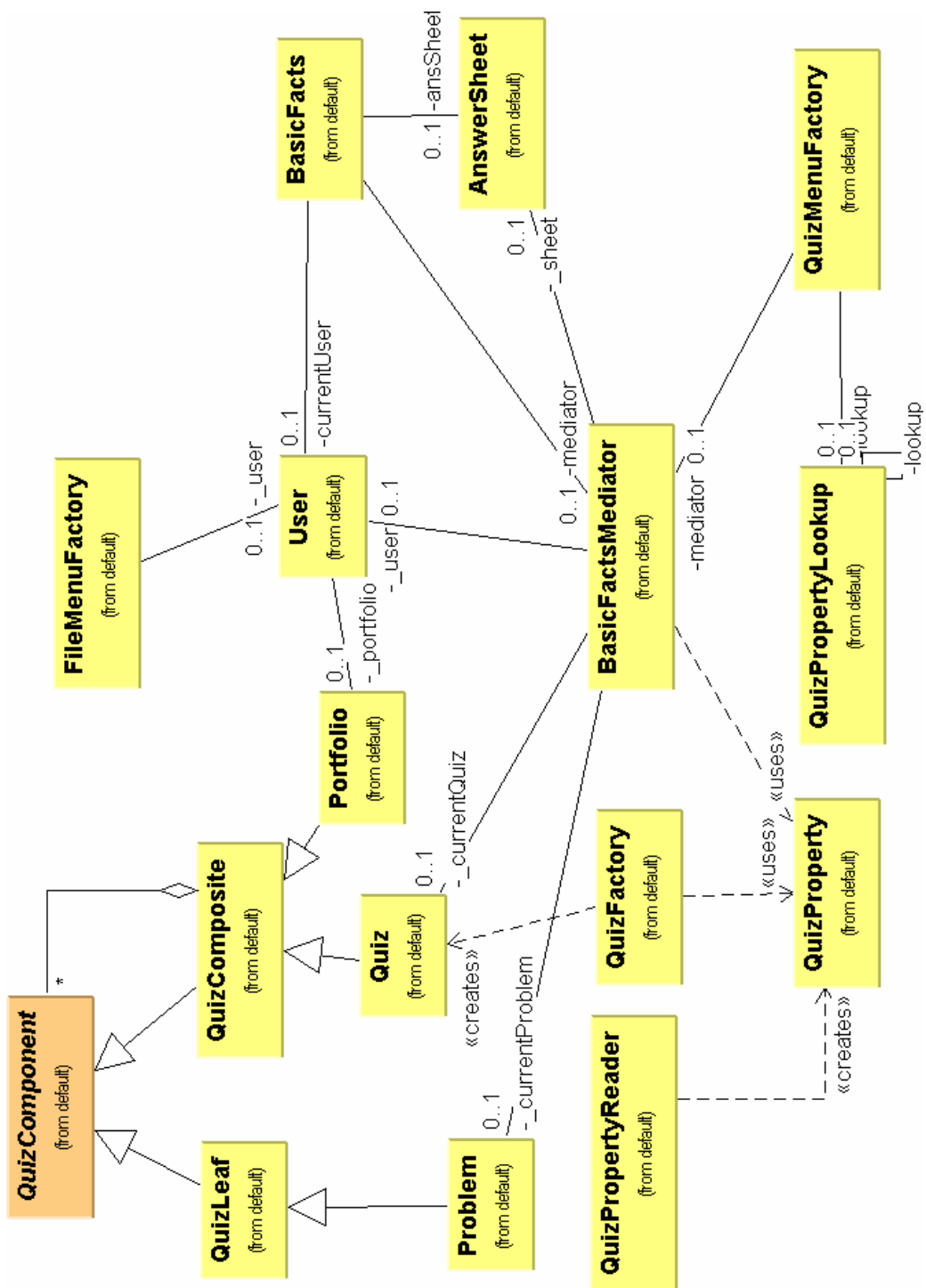


Figure 26. Class diagram for the second design of basic facts after all the refactoring

The new design, according to Fig 26, is very flexible and balanced. Since the hierarchy of quiz classes supports framework, with most of the classes remaining untouched, it is a simple matter to add a new quiz type. The new quiz type would not be required to be math related application since that behavior has been abstracted out of the quiz class. As long as the new application's quiz follows the format described before, most of the classes will remain untouched and be easily reused, except for **QuizFactory**, **BasicFactsMediator** and **AnswerSheet** that are specific to each application. Compared to the first design of Basic Facts, this latest version is greatly reusable and easily accommodates new requirements.

Conclusion

Through this research project, I learned design patterns and refactorings along with the development of *Basic Facts*. I applied several patterns and systematic refactoring techniques, which lead to several design attempts. I observed their benefits and drawbacks through out the evolution of the software. I discovered that applying design patterns blindly to the codes makes the design complicated. I came to the conclusion that it is better to focus on simple and concise design with well-disciplined refactoring rather than unnecessarily sophisticated design with design patterns. I also found that design patterns work greatly if they are applied after refactoring. The other discoveries and difficulties in my research are noted as follows.

[From my experience]

One of the biggest mistakes I made in designing *Basic Facts* was in struggling to accommodate a design pattern that did not fit the situation. I discovered that unless a design pattern absolutely makes tomorrow's job easier, one should not bother struggling to apply patterns to one's system just for the sake of flexibility. Instead, one should try to make a design simple. As the programming goes, the developer may refactor when his code starts smelling. Unnecessarily anticipating what may happen in the future will cause a lot of problems. My experience in this regard taught me that if a design pattern comes naturally, use it, otherwise let it go. I also learned not to obsessively think, "I have to use a design pattern to make my design more reusable." Refactoring is always right there for you.

I have found it is much easier to apply design patterns to a program after refactoring the code. The refactoring process helped me to see things that were not visible prior to refactoring. Refactoring often triggers modification to the system through the use of design patterns. Instead of thinking, "What pattern may be used here?" it is better to wait for an inspiration to occur during the refactoring process. In this way a suitable pattern appears more naturally and fits better with the code.

New functionality should not be added during refactoring. Sometimes, even though we know this principal, we are tempted to add new functionality to the code during refactoring. Adding new functionality to code while refactoring is dangerous in that if errors occur it can take a long time and much effort to return to the position

where initial refactoring began. Though it can be hard to resist the temptation, rather than modifying the code, write down any ideas for new functions. In this way you do not have to lose your idea but also will not break the refactoring principle and complicate the refactoring process.

I realized that when using search and replace in my refactoring, I need to do it with a great care. It is better to do such edits one by one instead of all at once. There always is a possibility that non-intended words can be replaced with a keyword, which is a possibility one with a different letter case (upper or lower case) or one that appears within a method or variable name. Replacing a wrong key word may not only slow down the refactoring process but also produce unnecessary bugs.

I used the common sense principle that if I see duplicate code occur three or more times within a program, it is time to refactor the code. Sometimes a programmer may favor development speed over conciseness of code. Even though he knows that duplicate code is a bad practice, he may tend to cut and copy some code when under the stress of a dead line. If duplicate sets occur several times throughout the program, refactoring becomes frustrating. Most importantly, duplication complicates code and makes it harder to maintain.

The refactoring techniques, *Extract method* and *replace temp with query* are simple to use but are the most effective tools in the refactoring process. One powerful effect of *replace temp with query* is that its use reduces temporary variables within the code and improves the readability of each method.

It is difficult to always come up with short and descriptive names for methods that explain what they are doing. Whenever I encountered that problem, I first named the function as best I could, then wait for an inspiration derived from refactoring. With the occurrence of changes to methods and with the growth of my understanding of the problem, naming methods becomes much simpler and obvious.

And last but not least, thinking too far ahead may not work well. Throughout the development of this project, I found it more productive to solve a problem which I currently face unless a solution to the problem is visible that might occur in the near future. Otherwise, it is easy to lose focus and introduce unnecessary interfaces or classes to the software.

References

Fowler, Martin *Refactoring: Improving The Design of Existing Code* Reading, Mass.: Addison-Wesley, 2000.

Cooper, James W. *Java Design Patterns: A Tutorial* Reading, Mass.: Addison-Wesley, 2000.

Gamma, Erich, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison-Wesley, 1995.

Kerievsky, Joshua.(2003) *DRAFT of Refactoring To Patterns*. [On-line resource]. Retrieved May 5, 2003, from the World Wide Web: <
<http://industriallogic.com/papers/rtp017.pdf> >

Metsker, Steven J. *Design Patterns Java Workbook* Reading, Mass.: Addison-Wesley, 2002.