May 2010

Methods for Educating Novices in Object Oriented Design

Matthew Allenson Netsch
Worcester Polytechnic Institute

Project Number: CS-KXF-EDU1

METHODS FOR EDUCATING NOVICES IN OBJECT ORIENTED DESIGN

An Interactive Qualifying Project Report:

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Matthew Netsch

Date: May 28, 2010

Professor Kathi Fisler, Major Advisor

1. Education
2. Programming
3. Object Oriented Design

Abstract

Object oriented design is often left out of novice level object oriented courses. The purpose of this project is to find a method that can be used to effectively integrate design in a novice level object oriented course. A group of high school novices were given a course in an attempt to demonstrate the project goals. The course was centered on two different methods; one involved a large project with abstract guidelines and the other involved a small project with specific guidelines. Results were obtained through observations of the students as well as a comprehension exam at the end of the course. The method that involved the smaller project demonstrated its potential to be used to effectively educate novices using design as part of an object oriented course. Future work is necessary to confirm this conclusion with a larger body of students and with a refined version of the second method.

Table of Contents

1 Introduction

The traditional approach for educating novices in object oriented programming leaves design out of the curriculum and reserves it for a more advanced course. Novices coming out of the traditional course end up programming object oriented programs as if they were writing in a procedural language. They have learned bad design and when they take an advanced course need to relearn many of their coding practices over again. The novice's learning progression could possibly improve if design was incorporated even at a basic level in novice courses. It is not the purpose of this project to determine the long term effects of such a course progression.

A method that has the ability to incorporate design in a novice level course and yet remain effective is the focus of this project. A prepared course is given to a small group of students in order to test out the methods and provide feedback. The small class size was chosen so that the instructor could focus more in depth on observations and have an easier time with adapting if assumptions were not found to be correct. However, this also means that even if the students demonstrate understanding using a particular method it cannot be statistically sound due to the small sample size. A follow up study should be made using more formal statistics to validate the results with a larger and diverse sample size.

The methods that are used are developed from the guidelines of several people's work on educating novices. The range of the work spans from the psychology of how people learn, to experts' object oriented conceptual models, to how people form conceptual models. All of these works served as general guidelines in preparing the content for the methods chosen.

Two methods were chosen to attempt to educate the class of students. The first method was the original and failed to yield any results. The method was a large project that had broad guidelines so that the students could make the design their own. They would write a program in Java in the way that they knew how and then take part in several code reviews. The instructor would encourage the students to elaborate the design in their own words and finally change the requirements slightly. The students would quickly see how much refactoring it would take with poor designs and would be motivated to search for better ones. There were several problems with this method that caused it to not yield the desired results; section 4 discusses these problems in detail.

The second method was developed learning from the mistakes of the first method. Instead of a large project it was a smaller one so the students ere not overwhelmed. The project was not open-ended and had very specific guidelines that the students were to follow. The designs were strict and they could not deviate from them after a code review. The concepts were taught to the students through instruction and backed by experience instead of through experience alone. The result of this method shows that novices have the possibility of learning object oriented programming with design integrated in with it.

2 Background

2.1 Design Principles

The goal of this project is to effectively instruct novice object oriented programmers to apply certain design principles. The design principles that the novices are expected to learn are listed below. There are variations to each design principle, therefore descriptions of them are provided. In order to demonstrate each principle, a chess application is used as example.

Design Principles

- Model, View, Controller
- Interface
- Observer Pattern
- Factory Pattern
- Service Locator Pattern
- Interface Segregation Principle

A chess program could be built with many objects that depend on details about the internal representation of the game board. Such a design, however, makes the objects tightly coupled or dependent, and subsequently fragile to change. A small amount of requirement change, which has a high chance of happening in practice, will cause a huge commitment in rewriting the program. For example, what would happen if the program was experiencing performance issues and the data structure for the board needed to be changed? Every single object that relied upon the board would have to be rewritten. What would happen if we needed to change or add new forms of I/O such as a GUI or a network app? What would happen if we wished to make a computer player? How would we configure all of these options? Imagine what would happen if this was actually a large program such as the developer's favorite video game.

In short, not following good design practices can require a lot of code refactoring later on. A lot of times we only care about fulfilling a general purpose and do not care about specific implementations. Using some basic design principles we can remove many dependencies or decouple the functionality of the program. Purpose and implementation can be defined separately and as a result we can make components dependent upon purpose. This is favorable because the abstract purpose of a program rarely changes, just the implementation does. The design principles given to the students for this project are just some of the basic ingredients to developing such a robust program.

Model, View, Controller is a general software design principle about separating key components of a system. The remaining principles focus on creating flexible designs in object-oriented languages. Object oriented programming incorporates the use of a defined set of classes in order to build or instantiate all of the objects in the program. Objects own their own data and functionality necessary to carry out their role. However, this structure can produce many problems if used without good design. The rest of this section describes the design principles and their impact on OO programming in more detail.

2.1.1 Model, View, Controller

Many patterns can make up the model, view, controller (mvc). The purpose of mvc is to separate out and to decouple the program's data from its logic, and display. Good design typically follows when a program exhibits these characteristics. This pattern is more of an abstraction pattern and can fit almost any application. There are many variations to this pattern but the basic concept remains the same. The model is usually where the user data of the program is kept. A model provides a way to create, retrieve, update, and delete (CRUD) user objects. Transforming user data is the main purpose of any program. The controller contains the state machine and logical flow of the program. This part is similar to a brain or overseer of the behaviors that a program exhibits. Finally in order for any program to be useful there must be some kind of view. This view allows for data to be passed in and out of the program.

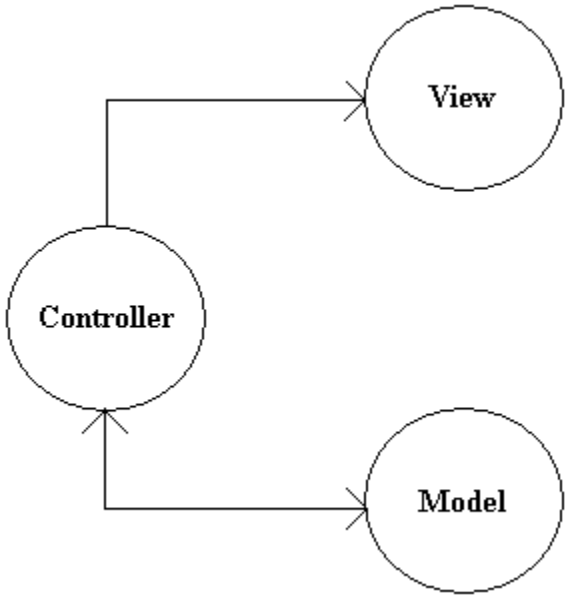


Figure 1 – Model, View, Controller

demonstrates one implementation of the mvc pattern. It is important to note that it is not typical to implement this pattern by simply using only three classes. The pattern typically consists of many classes and patterns grouped together into one of the three categories of the mvc according to their function. As for the game of chess example, the model consists of all of the factories and models required to represent a chess board and all of its pieces. The controller consists of all of the classes required to implement the game state and logic. Finally the view consists of all of the classes required to display the game.

2.1.2 Interface

An interface[1] is one of the basic design principles that exist in object oriented programming. This principle allows for the programmer to define an abstract purpose only. Classes which are concrete, or complete, can provide an implementation of this purpose. An interface simply contains the method declarations for performing a certain task. This guarantees to other classes that any class which implements this interface can be accessed in the way described by the interface. You can then make those other classes dependent upon an abstract purpose instead of upon the implementation contained in the concrete class. This allows for improved extensibility and maintainability as you will not have to change the classes that are dependent upon the interface when the concrete class changes. By removing this dependency you are decoupling your classes from those that use them.

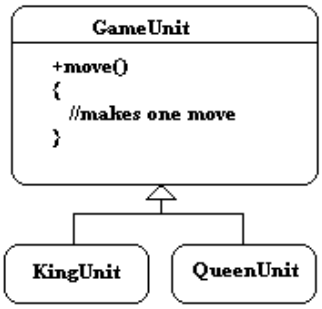


Figure 2 - Concrete Example

For example, game pieces could be implemented with a concrete base class seen in Figure 2; however that changes when new types of pieces are added. Let's say we implement a new type of chess piece called AwesomeUnit that can move twice per turn. The concrete base class has a move behavior which assumes that all game pieces would move only once per turn. Not only would there have to be a change the base class but there would have to be a change in all of the existing child classes as well.

[1] Freeman 04

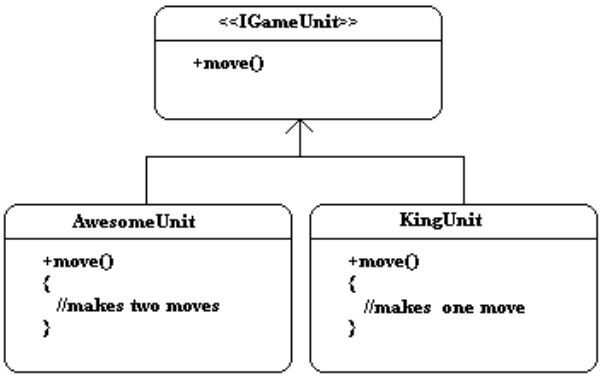


Figure 3 - Interface

A better solution would consist of not relying on inheritance but upon a more abstract concept called an interface. Diagrams usually denote interfaces with "<<>>" around the class names. The interface seen in Figure Figure , IGameUnit, defines only that a GameUnit must move. It delegates the actual algorithm to its implementing classes. Students learning Java abuse inheritance and do not fully understand interfaces. It should be noted that this solution is not the best design as it introduces problems of its own. Remember, that there is no perfect way to design. Many times finding the good enough design is better then finding the best way.

2.1.3 Observer Pattern

The observer pattern[2] provides a way for an object to receive updates from another. There are two main objects in this pattern; an observer, and an observable object. A decent way of implementing this pattern is to program to an observer and an observable interface. This allows each of the concrete classes that implement this interface the ability to post and receive updates without being tightly coupled to each other. This allows for modification in one without affecting the other, while still being able to communicate.

[2] Freeman 04

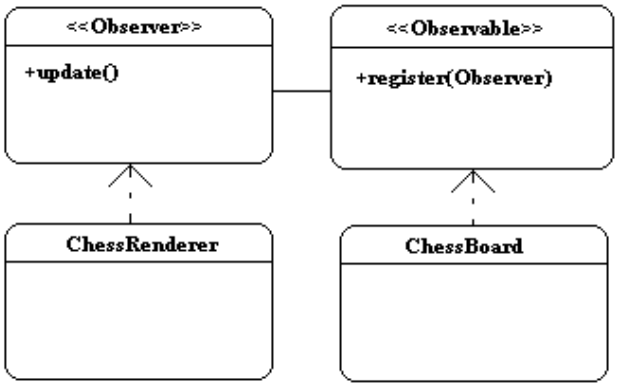


Figure 4 - Observer Pattern

Figure 4 demonstrates the observer pattern for the chess example. The renderer or view does not keep track of the underlying current game state. The view in this case is an observer of the board and is interested when it changes. ChessRenderer is a concrete Observer class, and ChessBoard is a concrete Observable class. Since ChessBoard is an Observable it can accept any Observer in its register method. When the ChessBoard wishes to post an update notification to its registered Observers, it calls update on them. Since ChessRenderer is an Observer it can have update called upon it by the Observable. In this way, when the board changes the view can then display the updated board state. This decouples the underlying board state from the current view so that if one is updated or modified then the other will not have to change. Also this allows for adding multiple observers in addition to the view without changing any of the ChessBoard.

2.1.4 Factory Pattern

The factory pattern[3] uses the same principle as interfaces in order to decouple the creation of objects from those using them. If there is a class that requires a product and it instantiates it you must change that class every time a new type of product is added. A better way would be to create a set of factories that can create each of the required products. Therefore when you add a new type of product you will only have to create a factory for it and not have to change any code that exists before.

[3] Freeman 04

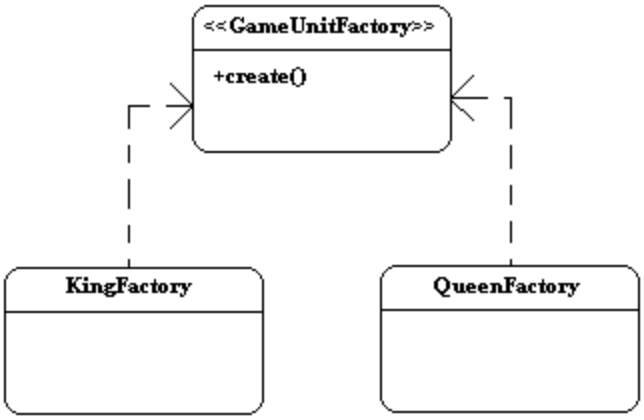


Figure 5 – Factory Pattern

Figure 5 demonstrates using a factory method pattern for creating the products listed in the interface section. GameUnitFactory is an interface which defines a create behavior for all unit factories. KingFactory and QueenFactory provide the concrete implementations of the GameUnitFactory. Each concrete factory builds their respective IGameUnit product. The client that needs to use these game piece products would be a game engine. Instead of having it know how to create each and every game piece, it would ask a GameUnitFactory implementing concrete class to create the desired piece. Providing concrete implementations of the GameUnitFactory to the game engine is a dependency issue in itself. This dependency issue is one of configuration and is discussed in the Service Locator Pattern section.

2.1.5 Service Locator Pattern

The purpose of the service locator pattern[4] is to separate configuration from use. A program can be designed with factories to remove the dependency of creating specific product objects. However, there is still the problem of depending on a specific factory in order to create the appropriate products. Therefore the service locator pattern is one method of inverting this dependency. It accomplishes this by handling the problem of selecting or locating a specific service, in this case locating a factory. Common methods of locating the desired factory are through a service name lookup or through a configuration file. The latter is used for the novices' assignment.

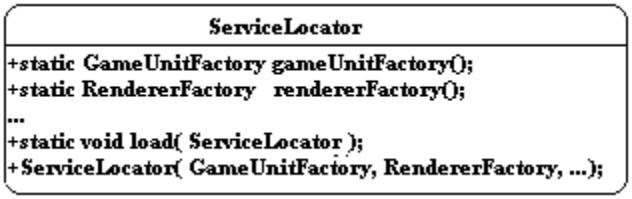


Figure 6 – Service Locator Pattern

[4] Fowler 04

The service locator in Figure 6 handles the configuration for all of the services the program requires. All of the objects that require a service can obtain that service once the ServiceLocator is configured. All of the dependencies that these objects had on configuration are now gone and therefore the program does not need to change when there is a change in configuration. Let's say the chess program reads in a configuration file when it starts up. The configuration file could say something like use the Chess Extreme 2.0 game units and render with OpenGL. The program would then create the ChessExtremeFactory and the OpenGLRendererFactory. The ChessExtremeFactory in this example implements the GameUnitFactory and delegates creation of an IGameUnit to all of its unit factories such as the KingExtremeFactory and QueenExtremeFactory. Note that this is a variation of the factory method pattern described in section 2.1.4, and is called the abstract factory pattern. The program would then construct a new ServiceLocator and load those factories into it. Finally it would store this constructed ServiceLocator object to the ServiceLocator class using the static load method.

2.1.6 Interface Segregation Principle

The purpose of the interface segregation principle[5] is to reduce the unnecessary dependencies an object inherits from using an interface that contains more functionality than is appropriate. For example, in terms of the ServiceLocator, an object that requires service A and not services B through F should not have to rely upon the entire ServiceLocator interface in order to use just service A. The interface segregation principle reduces this dependency by making the ServiceLocator implement several different interfaces. That way the object using a specific service or group of services need not rely upon the entire ServiceLocator interface.

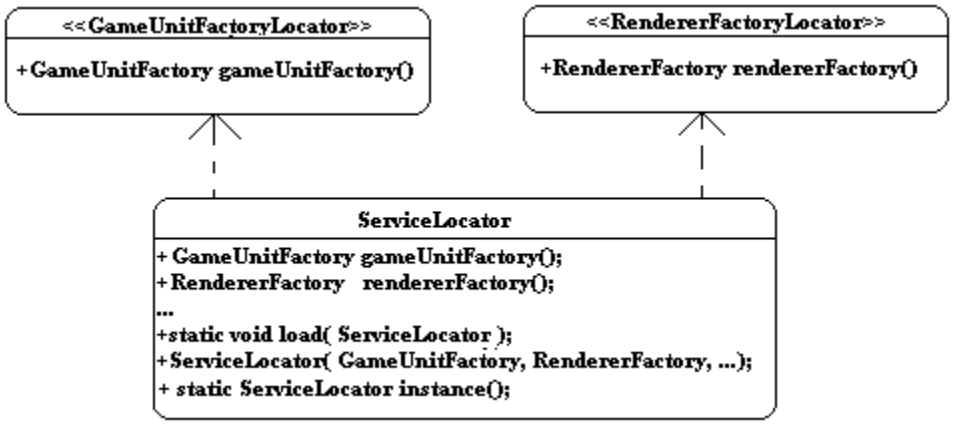


Figure 7 – Interface Segregation Principle

Using the service locator example in Figure 7, a View only requires the RendererFactory service and not the GameUnitFactory service. It is inappropriate that a View must depend on the gameUnitFactory interface in order to use the rendererFactory interface. Therefore the ServiceLocator implements several interfaces, GameUnitFactoryLocator and RendererFactoryLocator, so that the objects using a particular service need only depend upon the interfaces they require.

[5] Martin 96

2.2 Literature

The following contains summaries, some verbatim, of the relevant papers used for this project. They are here to provide reference as background material and to show the source of this project's methods.

2.2.1 How Novices Learn Computer Programming

Mayer[6] presented two methods, elaboration and concrete models, to aid novices in understanding computer programming. He created these method by using techniques developed by psychologists to back his experiments. This project deals with the same premise as Mayer's work. They both attempt to instruct novices in how to programming using techniques from psychology. However, newer research has been carried out since Mayer's work. Our understanding of computing has grown; especially into that of different paradigms and techniques. Therefore the procedural model he used is out of date; however his research is still noteworthy because people do not change. This project instead uses the object oriented paradigm as a platform for instructing the novices. However, the techniques Mayer presented for instructing novices are still useful today and serve as helpful guides for the methods included in this project. Several definitions from his paper are adopted, reproduced verbatim.

Definitions

- **Meaningful learning** - a process in which the learner connects new material with knowledge that already exists in memory
- **Schema** - Existing knowledge
- **Assimilation** - Process of connecting new information
- **Short Term Memory** - a temporary and limited capacity store for holding and manipulating information
- **Long Term Memory** - a permanent, organized, and unlimited store of existing knowledge
- **Novice** - Users who lack specific knowledge of computer programming
- **Rote learning** - Memorizing procedures to solve a specific problem
- **Understanding** - Analyzing a problem and ability to derive a solution for similar problems
- **Transfer Problem** - Problem aimed at testing the learner's ability of solving a similar problem that they have not seen before to determine if they have had meaningful learning
- **Manipulatives** - Concrete models such as coins or sticks that a learner can physically touch and interact with
- **Advance Organizer** - a short expository introduction, prior to text, containing no specific content from the text, but providing the general concepts and ideas that can be used to subsume the information in the text

Learning is only useful if the problem is understood. A problem that's understood can be used to derive a solution to similar problems not yet encountered. The usefulness of this transfer ability is what makes people competent in their field and therefore is valuable to learn. The opposite of meaningful learning is rote learning in which the novice does not make connections to information previously existing in

[6] Mayer 81

memory. Therefore the novice must access each piece of information individually and as a result cannot see the connections with various aspects of a problem. This means that novices will not gather the relevant information when they encounter a problem they have not seen before as well as experts would. According to Mayer, meaningful learning requires the following three steps.

3 Steps of Meaningful Learning

- **Reception** - Learner must pay attention to the information presented
- **Availability** - Learner must have previous related knowledge or schema
- **Activation** - Learner must retrieve previous related knowledge in order to make connections

Meaningful learning can only occur if first the novice is presented with relevant information. If he does not pay attention to that information or is uninterested then he will disregard it. The information must relate in some way to previous knowledge of the novice. If he cannot relate to the information provided then it will be difficult for him to produce any understanding. Finally the novice must be able to retrieve this related information or he cannot make connections. For example, he may be able to relate to the information presented however if he does not discover this relation he cannot make the connection.

2.2.1.1 Elaboration

This technique is aimed at influencing the activation requirement for meaningful learning. When a novice elaborates a concept he just assimilated in his own words he must gather familiar concepts previously stored in memory in order to describe the behaviors of the new material. The novice finds that this process is uncomfortable until he finds sufficient concepts that describe the new material and instinctively connects them. Mayer again presents several studies, but the conclusion is that the learner elaborating information in their own words stimulates activation.

2.2.1.2 Concrete Model

The concrete model was developed to provide a framework so that the novice has the availability requirement of meaningful learning. A concrete model helps the novice store relevant knowledge of a problem in order to make connections to this new information. Two main questions Mayer addresses are how concrete models influence learning, and how to make an effective model.

Manipulatives, a type of concrete model, physically demonstrates a concept in terms of actions that a novice can relate to. Mayer mentioned a study about students learning subtraction to demonstrate the use of manipulatives. One group learned subtraction using the traditional process and the other group learned using sticks. The tradition process includes explaining all of the rules of subtraction as well as two digit subtraction problems carried out in class. The sticks group associated subtraction with removal and borrowing actions which they were already familiar with. As a result the second group showed more of an increase in performance than the first group in transfer problems. This project attempts to reinforce the relations of the actions that novices are already familiar with and the actions of objects.

Another type of concrete model Mayer presented is the advanced organizer. This model provides the novice with background information of a problem before providing technical content related to it.

Results showed that advance organizers typically enhance retainability of information and increase performance in transfer tests. They do this by letting the novice become familiar with background information of a domain so that he can recall it later. This allows the novice to satisfy the availability requirement of meaningful learning. For example, let us assume that in the previous subtraction example that the novice does not know that sticks can have borrowing actions done upon them. They would not do well in the second group because there would be nowhere in memory for the new subtraction concept to get connected. An advanced organizer would ensure that the novice does have background information in memory by demonstrating that sticks can be borrowed.

However, according to Mayer, results showed that advance organizers are not as effective if the learner is not a novice, or if the organizer is presented after the target information. Experts already have their own schema in place and therefore gain no use from a newly provided one. Mayer also presented an effective demonstration of why the advance organizer must come before the new information. He provided a paragraph with instructions detailing how to do laundry but did not tell the reader what the paragraph is about until after. The paragraph is almost indecipherable without a title, however it is simple to understand when provided with a title. The title in this case is the advance organizer. The following are the guidelines for creating an effective advanced organizer verbatim according to Mayer.

Guidelines to Create an Effective Advance Organizer

- Should allow the reader to generate all or some of the logical relations in the text
- Should provide a means of relating the information in the text to existing knowledge
- Should be familiar to the learners
- Should encourage the learner to use prerequisite knowledge that the learner would not normally have used

There are a couple different kinds of concrete models that can be used to help novices connect familiar ideas. One model is the black box approach which represents a system as a block. The inputs and outputs of this block are clearly defined; however the internal workings are not. The black box approach causes the novices to rote learn as they basically are forced to memorize what outputs will be generated from a given input. This does not give the novice the tools required to generate or derive an output from a new input not seen before. The type of model that fixes this lack of transfer learning without exposing too much detail and overloading the novice is called the glass box approach. The glass box approach shows the internals of the system as a bunch of interconnected blocks. These blocks also have clearly defined inputs and outputs. Showing the internals in this way allows the novice to better understand how a system generates its outputs from a set of inputs. And as a result allows the novice to derive a solution instead of merely memorizing inputs and outputs.

Two properties for a Glass Box Model

- Simplicity - Small number of parts that interact in ways that can be easily understood
- Visibility - Users should be able to see the selected parts and processes in action

Mayer presented many other studies. One example of a concrete model he used involved showing the glass box model of a BASIC like language. This model included items that the novice would already be familiar with such as a memory scoreboard, an input/output window, a program list, and an output pad. The studies he presented all serve to reinforce his conclusion that advance organizers serve effectively as concrete models for computer programming.

2.2.1.3 Understanding Computer Programming

Mayer suggested that a statement in a language can be described as a series of transactions between memory and the processor. However, this view is procedural and formed at a time when memory resources were limited. In the more modern OO paradigm, a language is represented by different conceptual models than a procedural language according to Blackwell.

Mayer defined an expert as having 50,000 chunks of domain-specific information. This number seems arbitrary; however the direction in which novices need to push towards in order to become experts can be aided by examining how experts understand programming. Experts perform better than novices when analyzing a program that performs a meaningful task. However, experts and novices perform the same when analyzing a program containing no specific meaning. This result suggests that experts memorize meaningful patterns of domain-specific information. Experts organize chunks of information into larger chunks, while novices must remember each piece separately. Experts can see the interactions between the chunks of information in their memory because they are connected. Novices with unconnected chunks have a more difficult time when examining their interactions. Blackwell's findings backs this result as well and suggests that experts selectively pick out meaningful constructs out of the code where as novices have a hard time picking out the useful information. Mayer provided an example to demonstrate his own conclusion. He stated that experts develop a schema for tasks, such as calculate the sum of array X. Experts know the general structure for performing such a task and can interpret when they see that particular structure. Finally experts build up programs into macrostructures called modules using these substructures. They can then analyze the program after that point more efficiently. Mayer's result suggests that procedural languages are built up into hierarchies for performing tasks. However, classes in the object oriented paradigm are also built into hierarchies. There are also schemas for doing particular tasks called design patterns.

2.2.2 Conceptual Models of Java

Blackwell[7] investigated the frequencies of words in several Java library documentations to determine the mental models the authors have of programming through identifying the underlying conceptual metaphors. His research is used here mainly to learn about the concrete models that experts have in the object oriented paradigm. As well as try to construct a set of terminology and examples to help novices connect information. His research, combined with Mayer's, provides the basis of this project.

[7] Blackwell 06

2.2.2.1 Technique Used

Blackwell did not analyze too deeply into the meaning of the words he encountered, in particular archaic words because users are typically unaware of their true meaning. Therefore those words may not represent the user's mental model. There are many idioms and words used in English today that hold significantly different ideas than when they were introduced into the language. Blackwell also omitted other words that held no significant clues to the conceptual model such as identifier names, descriptions of application domains, and repetitions of repeated phrases.

Blackwell investigated the java.applet, java.beans, and java.util libraries to gain the vocabulary for his analysis of the models. He used manual interpretations for the smaller of the two libraries and automatic methods for the bigger one, java.util. Blackwell learned from experience from the manual interpretations and used that as a basis for constructing the automatic method. He chose these libraries as a subset because they express the Java computational model and are familiar to Java programmers. Also the libraries are not tainted by a visual domain such as the GUI libraries.

2.2.2.2 Results

One of the surprising things he found was few Java keywords in the passages. The majority of the technical vocabulary was derived from other sources such as OO design textbooks and tutorials. The list below enumerates the models Blackwell found verbatim with example terms.

Conceptual Metaphors Found

- **Components Are Agents Of Action In A Causal Universe** (perform, optional, likely, modify)
- **Programs Operate In Historical Time** (regular, recent, immediate)
- **Program State Can Be Measured In Quantitative Terms** (large, less, many)
- **Components Are Members Of Society** (reconcile, collude, accompany)
- **Components Own and Trade Data** (distribute, deliver, obtain)
- **Components Are Subject To Legal Constraints** (violate, contract, permit, impose)
- **Method Calls Are Speech Acts** (instruct, query, offer, advise)
- **Components Have Communicative Intent** (refer, describe, indicate, represent)
- **Components Have Beliefs And Intentions** (assume, consider, intent, desire)
- **Components Observe And Seek Information In The Execution Environment** (measure, observe, recognize, scan)
- **Components Are Subject To Moral And Aesthetic Judgment** (fair, malevolent, graceful)
- **Programs Operate In A Spatial World With Containment And Extent** (back, contain, position)
- **Execution Is A Journey In Some Landscape** (turn, ascend, flip)
- **Program Logic Is A Physical Structure With Material Properties And Subject To Decay** (dynamic, sufficient, hard, structure, form)
- **Data Is A Substance That Flows And Is Stored** (buckets, channels, fill, source, empty)
- **Technical Relationships Are Violent Encounters** (target, trigger)
- **Programs Can Author Texts** (annotate, abbreviate, write)
- **Programs Can Construct Displays** (render, exhibit display)

- **Software Tasks And Behavior Are Delegated By Automaticity** (automatic, code, manual, human)
- **Software Exists In A Cultural/Historical Context** (legacy, traditional, obsolete)
- **Software Components Are Social Proxies For Their Authors**

Blackwell went on to state that documentation in the libraries rarely addresses the reader; therefore components are social proxies for their authors. In the legal example, components are given constraints and act within a defined law of society. The use of legal terminology, theorized to be used subconsciously, encodes the components' hierarchy and power relations. OO Design also has an influence on the metaphors that were used, such as the economic ownership of data describes encapsulation.

One noteworthy result is that although the relative frequency of different categories are obviously influenced by OO, many metaphors that Blackwell found do not appear to be specific to OO libraries, or to Java; rather, they reflect the mental models of software operation. In addition, the high use of causal actions as opposed to declaratives indicates a procedural model. Therefore procedural models could be relevant in the expert understanding of Object Oriented design. This backs up the premise that Mayer's research, which is procedural in nature, can be adapted for this project.

2.2.3 Mental Representations of Programs

Fix, Wiedenbeck, and Scholtz[8] attempted to define a framework that can be filled in with content to produce conceptual models of computer programming. This research is similar to Blackwell's, which determined the content that could be used by such frameworks in order to produce conceptual models of object oriented programming. Mayer stated that people need to associate new information with previous information in memory. Experts borrow actions of a society that is used to connect OO design into their memory. But exactly how do experts determine which content to use as this connection point? Fix, Wiedenbeck, and Scholtz's research described the basic things that are necessary in a conceptual model in order for a person to be an expert. They stated that if a person does not have these items then that person cannot effectively retrieve data and make decisions. In other words, that person would not likely be an expert in the domain. However, like Mayer, this research is done with a procedural language and is out of date so this is just used as a guide when refining the models presented to the students. Below are the abstract features from their research copied verbatim. These abstract features are the items of the framework that are inherent in every expert's conceptual models.

[8] Fix 93

Abstract Features

- **Hierarchical Structure** - Understanding code as a hierarchical design
- **Explicit Mappings** - Understanding the connections between goals and code
- **Recurring Basic Patterns** - Understanding the common plans that solve specific problems
- **Well Connected** - Understanding interactions between plans
- **Well Grounded** - Understanding of position of structures in code text

2.2.3.1 Method

Their method involved a comprehensive exam after studying a program for a few minutes. Specifically 20 novices and 20 experts were given a 135 line Pascal program with no documentation and were told to study it. The novices had schooling in all of the areas the program covered. Experts were professionals with a median of 7 years experience. After 15 minutes the program was taken away and they all were given a comprehension exam that tested for the presence of the abstract features given above. Details on this exam can be found in their paper.

2.2.3.2 Results

It was clear that experts, but not novices, possess the abstract features given above. The experts extracted many kinds of data from the program to produce a good mental model of the program's tasks. The authors put forth that although the results could be that novices simply have less knowledge; there is a possibility that their lack of comprehension has to do with a different reading strategy that produces a poor mental model. They made an excellent point here and it is something to be aware about when instructing the novices. Mayer's work backed this up and provided further evidence that getting the novice's to elaborate the concepts in their own words will stimulate activation and allow the instructor to determine where they are weak upon in terms of extracting information.

3 Methods

This project involves testing the effectiveness of different methods in order to educate novices in object oriented design. This advanced concept is typically taught later in a student's education. However, given the research into how experts understand object oriented programming there should be a way to teach novices design earlier on. Teaching novices design early on may give them long term benefits; however, studying those effects is not a part of this research. There are two main methods this work experiments with to educate a group of novices.

The first method focuses on a single large project throughout the course. The assignments involve a series of design reviews to continually get the students to refractor their code. The theory behind this is that the students are forced to reevaluate their code as the requirements change. They see the reasons why some designs are good and why others are not so good. This provides for them relevance and motivation so they will have the drive to connect the information found into memory. The project is one that they can relate to so that they will have previous knowledge already in memory to satisfy the availability requirement. All that needs to be done by the instructor is to keep the students in a state of continually thinking about and elaborating their design. The project for this approach is the game of battleship.

The second method was developed to try a much more guided approach for the student. The class still involves a single large project that the students continually work on. However, the project is broken up into smaller assignments. The students do not get the full view of the project until the end. This method is used in order to try to avoid an overload for the novices. Also this allows them to focus on one design pattern at a time so that they can more effectively see the effects of introducing the new design into their code. Hopefully, at the end of the class they will be able to look back upon their program and see the power of following simple design patterns. The project for this method is the game of Tic-Tac-Toe.

The class this project focused on was a class of 3 high school students of different class years. All of the students have had a beginner programming class in a procedural language such as C. This is a requirement because it is not the goal of the project to instruct the students how to program. The population size is chosen so that the instructor has the ability to work closely with the students in order to observe them and change the material according to need. One of the project goals is for novice students to learn OO design; therefore they should not have any previous knowledge about OO.

The course is scheduled over one high school semester. The language chosen to implement OO concepts is Java. A comprehension exam is given at the end in order to test their knowledge of the design principles given. A major criterion in the exam is if the student is able to transfer the knowledge learned and can effectively interpret designs given. The design principles chosen to instruct the students are as follows; descriptions can be found in the background section:

Design Principles

- Interface
- Observer Pattern
- Factory Pattern
- Service Locator Pattern
- Interface Segregation Principle
- Model, View, Controller

3.1 Large Project Method: Battleship

The class organization of this method first starts off with teaching the students a small amount of Java syntax such as classes. The reason for this is that the students need to be able to write the initial version of the project on their own. After they have the base version then the class turns into a series of design reviews. Every few weeks the instructor changes the requirements and tries to guide the student to a solution. The student has no idea that they are learning a design principle. They only know that they are learning a design that makes their life easier in completing the new requirement. Battleship is chosen because there are a lot of areas to include the design principles above as well as be interesting to the student.

3.1.1 First Assignment

Game play is a turn-based, two human player, text game.

Pieces involved are 5 ships per player

- Minesweeper - Length of 2 (Displayed as [M][M] on the field)
- Submarine - Length of 3 (Displayed as [S][S][S] on the field)
- Frigate - Length of 3 (Displayed as [F][F][F] on the field)
- Battleship - Length of 4 (Displayed as [B][B][B][B] on the field)
- Aircraft Carrier - Length of 5 (Displayed as [A][A][A][A][A] on the field)

Each player's ships are contained in a separate 10by 10 field. An example field is shown below.

```
[  ][ 1][ 2][ 3][ 4][ 5][ 6][ 7][ 8][ 9][10]
[ A][  ][ M][  ][  ][  ][  ][  ][  ][  ][  ]
[ B][  ][ M][  ][  ][  ][  ][ +][  ][  ][  ]
[ C][ +][  ][ +][  ][  ][  ][  ][  ][ F][  ]
[ D][  ][  ][  ][  ][ +][  ][  ][  ][ F][  ]
[ E][  ][  ][  ][  ][  ][  ][  ][  ][ F][  ]
[ F][ +][  ][ A][ A][ A][ A][ A][  ][  ][  ]
[ G][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ H][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ I][  ][ *][  ][  ][  ][  ][ +][  ][  ][  ]
[ J][  ][ *][  ][  ][  ][  ][  ][  ][  ][ +]
```

Hits are designated as [*] and misses are designated as [+] on the field. Two ships cannot overlap and must be fully on the field. Also they are not visible as shown above until they are sunk.

During the **Setup Phase** both players are required to take turns placing their ships. The console will show the player's field and what ships are available to be placed. It asks the player which ship they would like to place and the board position for the corner of that ship. Then it asks which direction they would like the other end to go towards. If successful then print out the map and the list of available ships again and repeat until all ships have been placed. Then switch to player 2's field and repeat the above process. After that game play proceeds with alternating turns between players.

Turn Rules

- Console shows opponent's field (ships hidden) and asks player to enter a coordinate to attack. Should also print out the ships that the player has sunk.
- Console then tells the player if hit or miss.
- Miss: Turn ends go to the first step for the next player
- Hit: If all the grid points that the ship occupies have been similarly hit then the ship is declared sunk and removed from the opponent's field. Turn begins again with the same player.

Make sure you check for errors and that the resulting fields are valid after each turn.

End Game occurs when all the ships a player owns have been sunk. The winner is the player that still has ships left.

Example:

Welcome to battleship!

```
[  ][ 1][ 2][ 3][ 4][ 5][ 6][ 7][ 8][ 9][10]
[ A][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ B][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ C][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ D][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ E][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ F][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ G][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ H][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ I][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ J][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
```

Available Ships
1. Minesweeper (2)
2. Submarine (3)
3. Frigate (3)
4. Battleship (4)
5. Aircraft Carrier (5)

Player 1 please enter which ship you would like to place: 1
Please enter which coordinate you would like to place it: A3

Available Directions
1. Right
2. Down
3. Left

Please select which direction you would like the ship to go in: 2

```
[  ][ 1][ 2][ 3][ 4][ 5][ 6][ 7][ 8][ 9][10]
[ A][  ][ M][  ][  ][  ][  ][  ][  ][  ][  ]
[ B][  ][ M][  ][  ][  ][  ][  ][  ][  ][  ]
[ C][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ D][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ E][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ F][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ G][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ H][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ I][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ J][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
```

Available Ships
1. Submarine (3)
2. Frigate (3)
3. Battleship (4)
4. Aircraft Carrier (5)

Player 1 please enter which shi...

-Continues for bother players until all ships are placed-

Player 1's Turn

```
[  ][ 1][ 2][ 3][ 4][ 5][ 6][ 7][ 8][ 9][10]
[ A][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ B][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ C][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ D][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ E][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ F][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ G][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ H][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ I][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ J][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
```

No Ships Sunk

Player 1 please select the grid you wish to fire upon: G5

Miss!

Player 2's Turn

```
[  ][ 1][ 2][ 3][ 4][ 5][ 6][ 7][ 8][ 9][10]
[ A][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ B][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ C][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ D][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ E][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ F][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ G][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ H][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ I][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
[ J][  ][  ][  ][  ][  ][  ][  ][  ][  ][  ]
```

No Ships Sunk

Player 2 please select the grid you wish to fire upon: A2

Hit!

```
[  ][ 1][ 2][ 3][ 4][ 5][ 6][ 7][ 8][ 9][10]
```

```
[ A][   ][ *][   ][   ][   ][   ][   ][   ][   ][   ]
[ B][   ][   ][   ][   ][   ][   ][   ][   ][   ][   ]
[ C][   ][   ][   ][   ][   ][   ][   ][   ][   ][   ]
[ D][   ][   ][   ][   ][   ][   ][   ][   ][   ][   ]
[ E][   ][   ][   ][   ][   ][   ][   ][   ][   ][   ]
[ F][   ][   ][   ][   ][   ][   ][   ][   ][   ][   ]
[ G][   ][   ][   ][   ][   ][   ][   ][   ][   ][   ]
[ H][   ][   ][   ][   ][   ][   ][   ][   ][   ][   ]
[ I][   ][   ][   ][   ][   ][   ][   ][   ][   ][   ]
[ J][   ][   ][   ][   ][   ][   ][   ][   ][   ][   ]
```

No ships Sunk

Player 2 please select the grid you wish to fire upon: B2

Hit!

You sunk player 1's Minesweeper!

```
[   ][ 1][ 2][ 3][ 4][ 5][ 6][ 7][ 8][ 9][10]
[ A][   ][ M][   ][   ][   ][   ][   ][   ][   ]
[ B][   ][ M][   ][   ][   ][   ][   ][   ][   ]
[ C][   ][   ][   ][   ][   ][   ][   ][   ][   ]
[ D][   ][   ][   ][   ][   ][   ][   ][   ][   ]
[ E][   ][   ][   ][   ][   ][   ][   ][   ][   ]
[ F][   ][   ][   ][   ][   ][   ][   ][   ][   ]
[ G][   ][   ][   ][   ][   ][   ][   ][   ][   ]
[ H][   ][   ][   ][   ][   ][   ][   ][   ][   ]
[ I][   ][   ][   ][   ][   ][   ][   ][   ][   ]
[ J][   ][   ][   ][   ][   ][   ][   ][   ][   ]
```

Ships Sunk
1. Minesweeper

Player 2 please select the grid you wish to fire upon: ...

-Continues until all ships are sunk-

...

Hit!

You sunk player 1's Battleship!
Player 1's fleet is destroyed!
Player 2 has won the battle!

The first assignment used text for input and output based so the students were not distracted by complicated items such as a GUI. They were instead to focus on design and attempt to do their best on this first assignment. On later assignments, some of the requirement changes would have included adding a GUI and saving games. The students would have learned design principles through this process and would have ended up with a framework for a generic game. However, this assignment was the only one introduced to the students. This is because the method was a failure as discussed in the Results section. The next method attempts to learn from the flaws of this method.

3.2 Small Assignment Method: Tic-Tac-Toe

The class structure for this method is broken down into weeks and each week has a different concept that the students should learn. In the first two days of any given week, they are given to the students to write a particular program given to them by the instructor. They take the assignment and write a program that satisfies the requirements in the way they know how. The third day is a design review meeting and their design is looked over by the instructor. The instructor then makes a change to the requirements and asks them how they would change their design in order to satisfy it. Finally, the instructor suggests a better approach, a design principle in disguise, and instructs them to implement the new design. The following two days are then given to the students to complete this task.

The Model, View, Controller (mvc) pattern is the final pattern for the students to learn. This pattern is more of an abstraction pattern and can contain many classes and patterns inside of it. Therefore, the week to week schedule investigates one component of the mvc at a time. They learn all of the other principles along the way in order to implement all of the parts that consist of the mvc. Finally at the end the students can connect together all of the pieces and view how everything comes together in a final program. At the same time they do not feel overwhelmed looking at the big picture and break everything up into smaller design problems.

The program chosen for the students to work on is a simple Tic-Tac-Toe game. The project is simple and small enough that they can see the whole picture without getting overwhelmed and at the same time there is much room for extension to make the game interesting. The first week the students learn about the Model part off the mvc, the second week they learn about the view, and finally they learn about the controller. They learn items in this order so that they can implement the model and view as separate entities without worrying about how they connect together. Finally they develop the controller which links the two and can get the whole picture of how the mvc works. Splitting up design with these components allows the students to see a separation of data, display, and logic. Looking forward however, the time frames given above may be a little optimistic for a fast paced class.

The assignments are set up so the students and the instructor are acting out roles in a fictional story based around a software company. This method should provide distraction from the large nature of the project and allow them to focus on the little bits of code the instructor requires them to implement. Also this may provide motivation and combat the boring reality of high school courses Finally, this gives the students motivation to write the extra classes because they are forced to. The requirements given overwhelm the students; however the instructor always saves the day with the design principle. The students should be able to connect the concepts taught into memory.

3.2.1 Model Section

The goal of the model assignments is to aid the students in learning about interfaces, factories, and the model component of MVC. The model assignments show the students how those design principles are relevant in object oriented design by providing rationale through a real world scenario. The premise of the scenario is that a student is an entry level engineer at a company called Super Soft. Inc. The instructor of the class is the technical advisor and guides the students through designing their applications to compete with their competitor.

The first assignment simply asks the students to construct the central component of any Tic-Tac-Toe application; the game board. The students should see that the rest of the components developed are there to provide services in manipulating the game board. They find that the board class is fine enough on its own, until the situation changes and additional classes are needed.

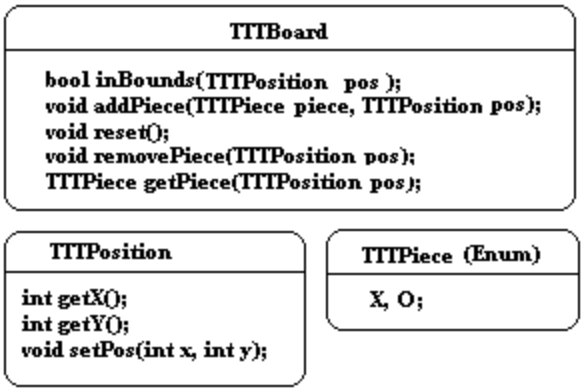
The second assignment changes the situation by having an external force change the requirements of the project so that it can satisfy a particular need. In this case, in order to get an edge over their competitor, marketing decides to advertise new features for the program. The board class must now have the ability to exist on several mediums of storage. This allows the company to market their product for a broader set of customers through supporting multiple platforms.

However, the benefit of this only occurs when engineering costs are low for adding new platforms. The students find out that the developer can quickly have a daunting task in front of him if the product was not designed well. The technical advisor has this foresight and guides the developer to design a system that can be extended easily. The main procedure of doing this is through the key design principle of isolating what components are expected to change from the ones that do not. This can protect the developer from refactoring more than he has to.

3.2.1.1 First Assignment

Congratulations on landing your first job as a software developer. You are now a proud employee of Super Soft. Inc. and have the honor of being assigned to work on their latest game Tic-Tac-Toe Extreme. Like any good company their requirements have a good likelihood of changing throughout the project to keep up with their competitors Awesome Game Co. You are therefore required to work with your supervisor and develop the sections he tells you to. Also you will be required to take part in code reviews with your technical advisor before your section can be incorporated into the code base. Thank you for coming on board and we look forward to working with you.

Any good program requires a strong backend to keep up with our competitor. Currently we only advertise support for having our game playable on a single computer without any persistence. In other words, only single player is supported for Tic-Tac-Toe Extreme and we cannot load a previous game after it has been restarted. Therefore, for the moment we are fine with keeping the entire game data in volatile memory. However, we are currently looking into other markets for our game such as embedded apps or network games so that we can get an edge over Awesome Game Co.



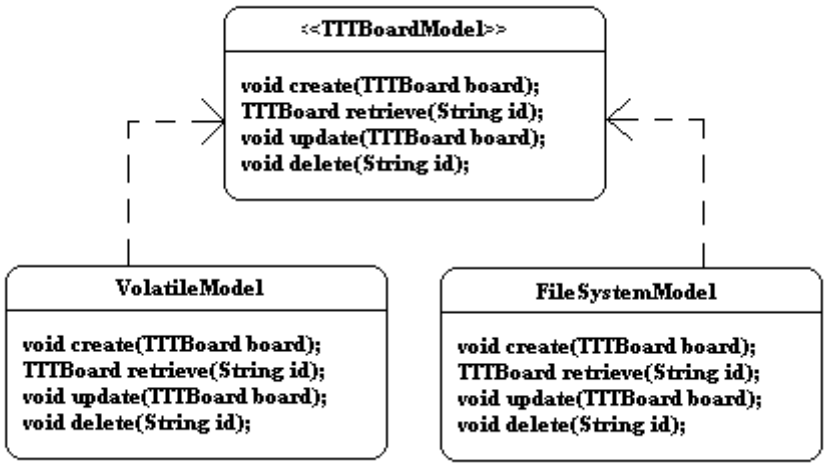
Your first assignment will be to develop a class that can store the game state for Tic-Tac-Toe Extreme. You are not the only developer working on this project so you must adhere to the chosen interface we came up with in our design meeting last week. As with any assignment you are required to implement test classes to test the functionality of your game. You must be thorough in your tests because it will look bad if your technical advisor's tests found a bug that you missed. The classes you must implement are in the diagrams that follow. Good luck!

3.2.1.2 Second Assignment

Excellent work on your first assignment, your employer is happy with his decision in hiring you. However marketing has decided to advertise the fact that Tic-Tac-Toe Extreme can have their games saved to your hard drive automatically so you can close the game at anytime and pick up right where you left off. To make matters worse they promised the fans of Super Soft. Inc. that version 2.0 will support online multiplayer. There's only one problem, you're the lucky developer that needs to implement this. Thankfully your trusty technical advisor is on your side.

The first rule of design is to identify what changes. Tic-Tac-Toe Extreme will always have pieces and will always be played on a 2-axis board. The way we represent the board will not change no matter where we store it. However, the location where we can store the board can change. The requirements state that we can place the board in memory at run time; on the user's file system to persist the game; or have the possibility for expanding onto a networked server.

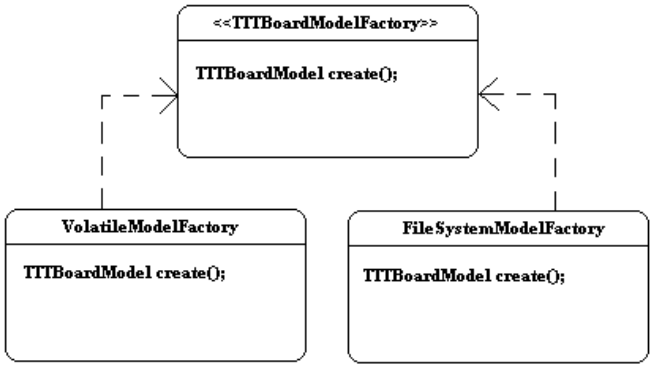
We could put the code to keep track of where the board currently is located inside the TTTBoard class. However what happens when we add new locations or ways a board can be stored? We are going to have to change sections of code that are dependent with all of the other ways of storing a board. The important realization here is that the way a board is represented and where a board is stored are two separate processes; especially since that the way that the board is represented will not change and where the board is stored will change.



Let us instead design a class whose sole purpose is to create, retrieve, update, and delete TTTBoards in one specific way. This means that we will create one of these classes (let's call them "models") for each way of storing the boards. Remember that the ways to store boards include in memory (volatile), on the file system, or on the network. However, that the network model is not part of the requirements at the moment therefore we should not include it. Right away we can see here that we have common functionality. We may be required to add new model classes in the future. We should program the

models to an interface is so that we can make the specific model independent from the class that uses it. A class that uses a specific model is tightly dependent to that model and as a result is not resilient to a change in the specific model. Therefore to avoid large amounts of code rewriting in the future we will eliminate this dependency. We do this by using the generic interface, TTTBoardModel. This interface will not change; therefore it is safe to make other classes dependent upon it.

However, there is a problem with this design. You cannot create an object of a generic interface. At some point we will need to construct the specific model. However, we are still in danger here of creating a bad dependency. The class that needs to use a specific model will have to construct it. We must design a way so that the construction of a model is also abstracted out of the class that uses it. Let us now design this class and call it a TTTBoardModelFactory.



This class knows how to construct a specific model. Therefore, we will have one factory for every model we can have. Again, we have common functionality now so all we need to do is program these factories to a single interface. Any class that needs to construct a model at will now only needs a TTTBoardModelFactory. However, it seems like we just deferred the dependency. We will still need to construct the factory. Later it will be decided just how to do this so at the moment you do not need to worry about that. Implement the classes that have been assigned and we will see you next week.

3.2.2 View Section

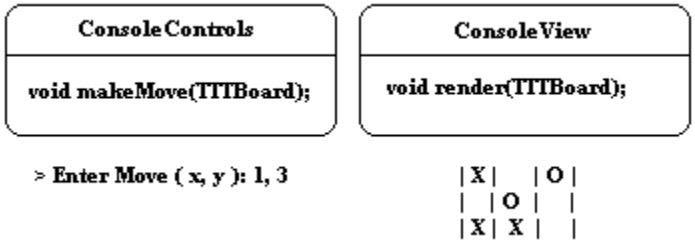
The goal of the view assignments is to reiterate the principles from the model assignment as well as introduce to the students the observer pattern and view component of the MVC. The view assignments show the students how those design principles are relevant in object oriented design by continuing the same real world scenario. The view assignments are meant to be similar to the model assignments so that the common material reinforces the information to the students.

The first assignment simply asks the students to construct classes that contain simple methods to retrieve user input to make moves and to display the board on the console. The students believe that these classes can be used "as is" in providing I/O functionality. Students seem to believe that specific implementations of user input and output are the most important components of any program. They often focus on developing those features at the sacrifice of a good back-end design as seen from the large project. The user interface is the most prominent feature of a program to a user probably because it is the part that they interact with the most. Developers, however, are responsible for maintaining and extending features of a program. Their task requires them to constantly interact with the back-end of the program. When the students switch roles, they should see the relevance of good design.

The second assignment changes the situation similar to the model section. Marketing advertises more features in a continuing effort to out perform their competitors. They do this by adding new ways of interacting with and viewing the board class. The student sees that this can quickly become a difficult task for a developer unless he were to put a little more thought into the back-end design of the user interface.

3.2.2.1 First Assignment

Your employer is very happy with the addition of last week's code. He sees that the cost vs benefits of adding new places to store the Tic-Tac-Toe board is low and grins as we can now keep up with Awesome Game Co. However, our game is far from complete! How can a user play the game if he cannot see it? This week your task is to design and write the view for the game. Unfortunately your technical lead will be gone for a few days so you will not be getting any advice until he gets back. However due to last week's accomplishments he is confident you have the ability to make the best console Tic-Tac-Toe view ever. Awesome Game Co. is moving fast producing their next version and rumors are starting to spread about what they are planning. Therefore you need to start making the view before your technical advisor returns. He will look at everything when he comes back.



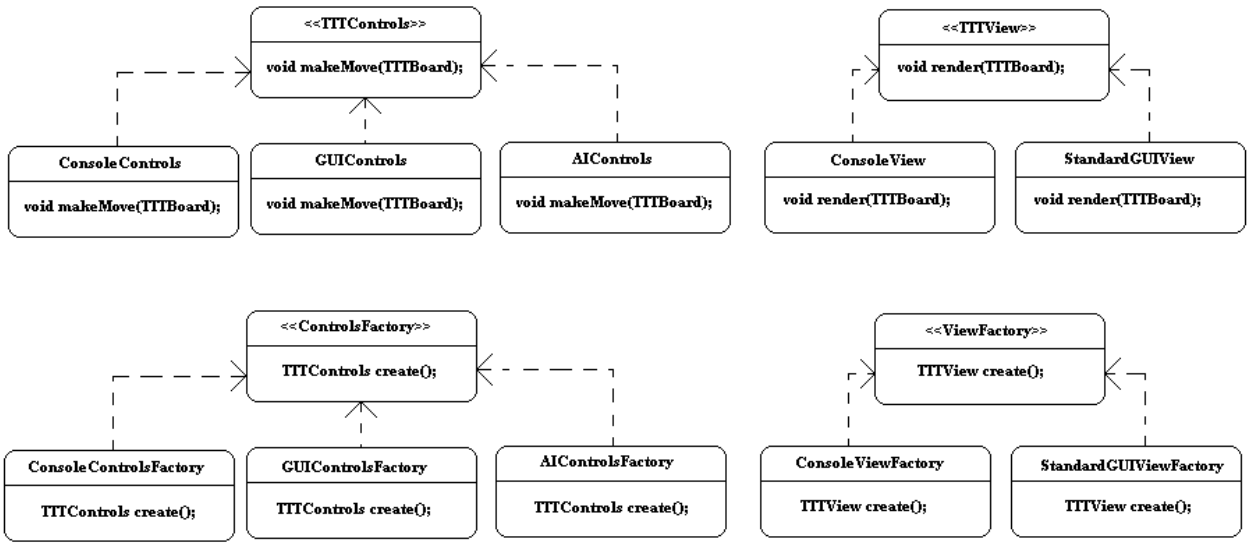
The ConsoleView class you are required to write is simple. All it needs to contain is a method to display a TTTBoard on the console. The ConsoleControls class is likewise simple and needs to contain a method for getting the next turn from the command line. Remember that JUnit tests are required to run your code and prove that it works.

3.2.2.2 Second Assignment

So far we've been keeping up with Awesome Game co. That is until they finally announced the features to the next version of their product. They advertise a variety of themes for their product. Their game can be shown with a cool ice blue theme, or their hot fire theme. Our console based game will not sell a lot when competing with these new features; therefore we need to improve our view. Marketing has decided to advertise our game can not only be played on the console but can support a graphical user interface (GUI). They also said that our game will incorporate a state of the art AI. Finally they promised that soon in the future Tic-Tac-Toe Extreme will support customizable themes.

Fortunately your technical adviser has just returned from his trip. He approved your console code; however a lot needs to be done to overhaul the current system to support the features that marketing promised. Again let's first identify what changes. In this new version, we have two things that can change. They are how the board is displayed and how we obtain our controls. Like last week with the models, a class that uses a view or has controls should be dependency on the specific implementation of either to protect that class from a change in requirements. Also, the construction of such classes needs to be independent as well to further protect the class using them. Fortunately, last week we have

already been exposed to the concepts of interfaces and factories that solve these problems. Let us now design the interfaces and factories for both the view and the controls.



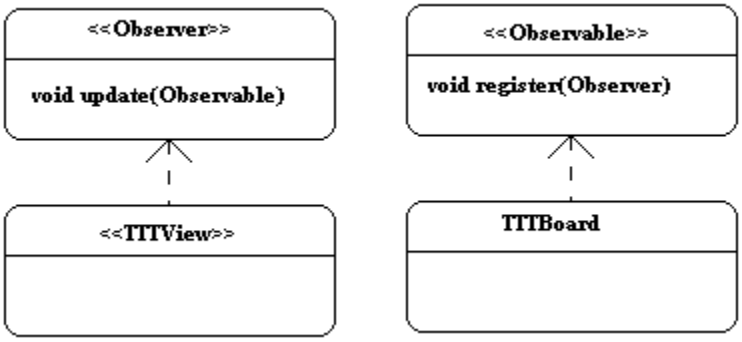
Again, we have a parallel structure between the products and the factories. Let's examine the controls branch first. We need three classes in the controls branch. The first is the ConsoleControls class which makes a move by prompting a user through the console. The second is the GUIControls class which waits until the user selects which move they wish to make through the GUI. Finally, there's the AIControls class which uses an algorithm to select which move to make when asked. Fortunately for you, the company outsourced the creation of the GUIControls and AIControls classes so you do not have to make them. Also you have already submitted the ConsoleControls class in the previous assignment. Now all that's left to do is to make the classes implement the TTTControls interface.

We need two classes in the view branch. The first is the ConsoleView class, which renders the board onto the console. The second is the StandardGUIView which repaints the display. The StandardGUIView has also been outsourced and you already written the ConsoleView. All that's left to do with this branch is to make the interface.

Finally we need the factories for all of the above products so we can construct them on the fly without any dependencies on the specific one we are creating. In doing so we can change or add as much controls and views as we please with little code rewriting. So make the factories just like we did with the models and submit all of the classes with JUnit tests when completed.

From previous experience, your technical advisor also believes you should make some additional improvements to the way your view is designed. What would happen if we wished to do something else every time the board is changed besides display it? An example of this would be if we wanted to preprocess the AI's next move in a background thread. The point here is that many items may be interested in updates to the board. However, the board should not care about who is interested so it does not gain unnecessary dependencies.

In order to accomplish this, we will classify the TTTBoard as an observable object. Therefore, anything interested in the board's changes, such as a view, will be classified as an observer. The board will send out a signal to all of its registered observers notifying them of a change. The observers will then decide what to do with the update such as render the scene if the observer is a view. Interfaces should be used here so that the observable object is independent from the observers. That way the board will not have to be changed anytime a new observer wishes to view it.



Great. Whenever the board is updated the view will change automatically now. All that needs to happen is for update() to be implemented in every view and for register() to be implemented in the TTTBoard class. Wait a minute, now we are duplicating a lot of code you say. Let us instead make Observable an abstract class and implement register() in there. Let us also make TTTView an abstract class and have update() call render(). Your technical adviser will help you along in developing this. Submit this with the rest of the assignment.

3.2.3 Controller Section

The goal of the controller assignments is to tie all of the components together through introducing the ServiceLocator, Interface Segregation, and controller component of the MVC. There is a lot of importance on this section so that the students can finally see a tangible, working system. Design has a lot to do with reducing dependencies on components that are expected to change. Dependencies are only the product of the interactions between components. This section should allow the students to see the complete system to get an understanding of those interactions.

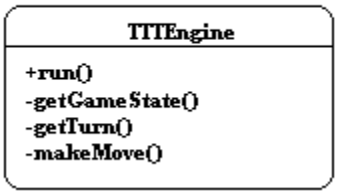
The first assignment simply asks the students to construct the game logic of Tic-Tac-Toe. The game logic is the controller of the MVC pattern. It uses all of the previously developed classes to accomplish its goal. The students see the role interfaces and factories play in removing the controller's dependency on specific views and models.

The second assignment addresses the final issue of choosing exactly which specific factories to provide the controller with. The method of choice is a configuration file in an effort to be simple to the students. They will have to write a parser for this file and develop a singleton registry called the ServiceLocator. This class serves as the provider for the factories, or services, that the controller requires.

3.2.3.1 First Assignment

Tic-Tac-Toe Extreme is looking amazing. Our sources tell us that Awesome Game Co. has started getting worried about our product. They simply cannot compete with our features. We are sure glad we have you coding for us. However, the key to making our product work is missing! If we ship the game now everyone will laugh at us. We have no game engine! Thankfully you have done so much work on the base code it will not cost us much for you to complete our awesome game. Just don't go getting any ideas about asking for a raise or anything. That was just a joke; the big boss has liked your performance so well that if you complete the game he is prepared to give you a share in the profits. Congratulations!

Alright let's get down to business. We are not out of the woods yet. Let's review what we have currently. We have our models, we have our views, and we have our inputs. The only problem is that we are missing the game logic! So let us write our game engine. This class will be our main controller class and will orchestrate all of the other classes together. Of course your trusty advisor will be there to help you out.



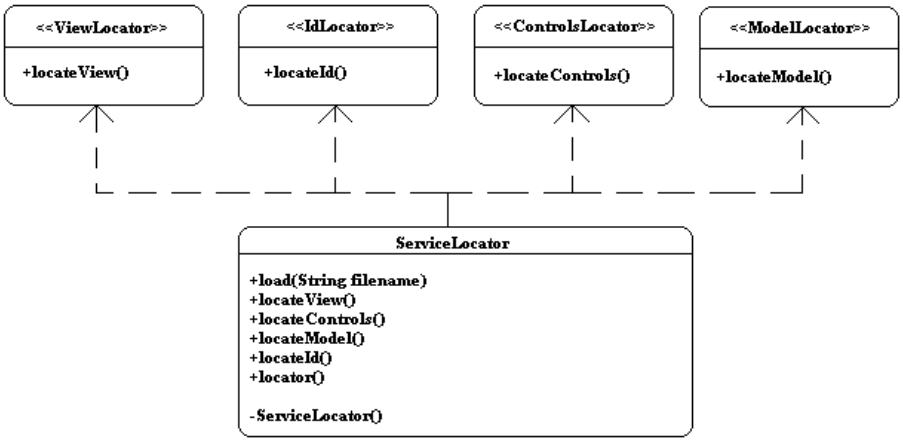
The game engine is pretty simple. Upon construction it will be required to construct a new TTTBoard. The id will be provided in a way to be later determined. Next a TTTModel, a TTTControls for each player, and a TTTView will be constructed from the appropriate factories. How the factories are provided will also be determined later on. The engine must then call TTTModel's create() method on the TTTBoard to allocate it inside its storage area. Next the engine must ask the TTTView to register to the TTTBoard so that it may start observing changes.

After this construction is complete the TTTEngine then has its run() method invoked. Run makes continual calls to the private methods getGameState(), getTurn(), and makeMove(). These methods will interface with the other classes in order to execute the Tic-Tac-Toe game. When the game is complete the run method will return and the program will exit. Please write this class and we will see you in the design review.

3.2.3.2 Second Assignment

Well it looks like we are finally done writing our Tic-Tac-Toe Extreme game. We have our views, controls, models, and game engine all written and working. All we have to do now is ship it. Or so that was the theory. The documentation department just emailed us. They asked how a user configures the settings of the game. We forgot to give the TTTEngine all of the factories it needs in order to construct the necessary objects required to run the game! Alright so let's finally tackle this one last problem.

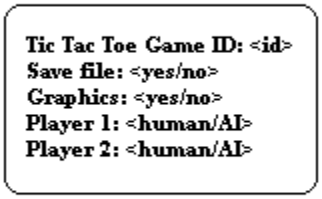
This problem at first seems difficult. We need to be able to supply the TTTEngine with the configuration specific factories for the current game session. However, we need to do this while the TTTEngine is dependent only on the interfaces we wrote before. Remember this allows us the ability to go back and add new features without changing TTTEngine. Let's do what we've always done before first. That is to isolate what changes. So what changes in this situation? The game we know we wish to stay the same. However, it is the configuration that we wish to change. Let us therefore make a class whose job it is to keep track of the current configuration. This way we will separate configuration from use.



The class in the figure that can do this is called the ServiceLocator. The factories we wrote before provide services or ways of executing processes related to Tic-Tac-Toe. The constructor of the ServiceLocator is private meaning that we cannot instantiate an object of ServiceLocator except from within the ServiceLocator class itself. This allows us to guarantee that there is only one instance of ServiceLocator in the program designating it as the authority of configuration. The load() method is static so you do not need an instance of ServiceLocator in order to use it. This method takes in a filename and will read the configuration file in order to construct a new ServiceLocator. This instance of the ServiceLocator will provide the appropriate factories when the locate methods are called as specified by the configuration file. A class will need to call the locator() static method in order to retrieve this sole instance.

We have also programmed the ServiceLocator to implement several interfaces. Doing this allows us to invert the dependency classes would have on the ServiceLocator. Typically a class would only be

interested in one service at a time; therefore it is inappropriate to force that class to become dependent on all of the services. Without the interfaces, a single change to ServiceLocator could end up making you rewriting much of your program. Finally we need to write how the configuration file will be written.



The figure specifies the configuration file. The file is text based so it can be human readable and modifiable. We want average users to be able to change options so it must be simple. For later versions we maybe able to save these configuration files and have the user in the program choose his options. For now this is how we are doing it. Everything in a "<>" should be replaced with the specific option you wish. The"/"means that either text on the right or left could be placed in that spot. For example, you could have either "Save file: yes" or "Save file: no". Please name this file config.ui and write ServiceLocator. Finally, once you do that we will be done.

4 Results

4.1 Battleship

This method seemed promising however there ended up being problems to this method. These problems were severe enough that the method was abandoned after the first design review.

The most obvious problem was that the students were completely overwhelmed by the size of the project. This was a surprising find because the students have previously done similar sized projects procedurally in C++. They were too focused on programming the correct solution from the beginning. This is probably a reason why they became overwhelmed. Specifically, they saw a new paradigm, experienced a few lectures on simple Java constructs such as classes, and as a result felt they needed to have the perfect design from the start. The result here is that exposing novices to a large project requires them to receive constant guidance. When revising this project, perhaps a smaller scaled project would be better such as Tic-Tac-Toe.

Another problem was that the students were surprisingly not motivated with random requirement changes. They would rather stick with familiar methods than learn new designs. The main reason for this is that some of the students found the extra classes caused by most design patterns to be unnecessary. Some of the students found these extra classes to be unnecessary. Developing the large project was not enough motivation in itself for them to continually make their program better. A better approach would involve isolating specific parts of the project. Each part of the project would be developed and then refractored as if the students are a part of a development team. They should see that there are meaningful changes to the requirements due to a change in circumstances in the development team. An interesting result that occurred is when the students were given small random constraints on what they can do, such as the students can only have 5 lines of code in the main method, they saw that as a challenge and were motivated to program a solution. One possible method of instructing in the future would be to base the whole class off of a competition.

The students also seemed to have spent too much time on parts that were irrelevant to design. They believe that the design classes were extra and unnecessary so they did not focus on using them. If the requirements had allowed them to have free reign over the controls and view of the battleship program, then they would have spent all of their time making the best looking GUI in the world. Their design underneath however would be junk. Therefore much of the complicated code such as the GUI or AI modules should be provided for the students.

It was apparent that the students felt the design patterns to be unnecessary. Part of this was because they could not visualize the patterns. This means that the students did not connect the patterns in memory because they were not getting the receptive part of the requirement. The students could not extract the concepts the instructor wished them to learn through experience. The students felt uncomfortable when asked to do such a process. They felt much more comfortable when they are not

grilled for the patterns. The point is that the instructor needs to explicitly state the concepts the students should learn. Expecting them to extract the concepts from practice is not productive

4.2 Comprehension Exam

All of the problems in the first method were fixed in the second method. The second method effectively eliminated all of the downfalls the larger project had for the students. Therefore, the best test to determine if the second method was effective in allowing the students to learn is the comprehension exam. Design is a hard skill to master well. Therefore given the lack of experience the students should not be expected to produce or generate a design. However, they should be able to recognize and elaborate on all of the designs they have encountered throughout the project. The goal of this project is to test if the students understand object oriented design. Therefore, they should be able to demonstrate the ability to recognize design patterns in programs they have not encountered before.

The problem types of the exam consist of matching and multiple choice. The exam contains two sections. The first section is multiple choice and asks the students questions about the uses of certain design patterns. This section covers the requirement of determining if the students can recognize the use of all of the design patterns. The second section contains a single problem. This problem shows a diagram of a large program and gives the student a word bank of all of the design patterns included in the diagram. The student is to write the appropriate design pattern next to its spot in the diagram. The student will have effectively demonstrated transfer learning if he can accomplish this task.

4.2.1 Exam

Section 1

Choose the design principle that is appropriate for the given use.

1. A class contains many methods that can be grouped for particular uses in separate abstract classes.

    A. Interface
    B. Dependency Pattern
    C. Interface Segregation Principle
    D. Service Locator Pattern

2. Several components need to be notified when a particular component posts updates.

    A. Factory Pattern
    B. Observer Pattern
    C. Messenger Pattern
    D. Service Locator Pattern

3. A component wishes to use an object but remain independent of its creation.

    A. Factory Pattern
    B. Assembler Pattern
    C. Interface
    D. Model, View, Controller

4. A component wishes to use a class but remain independent of its implementation.

    A. Dependency Pattern
    B. Interface Segregation Principle
    C. Interface
    D. Observer Pattern

5. A component requires the use of several classes but wishes to be independent of their configuration.

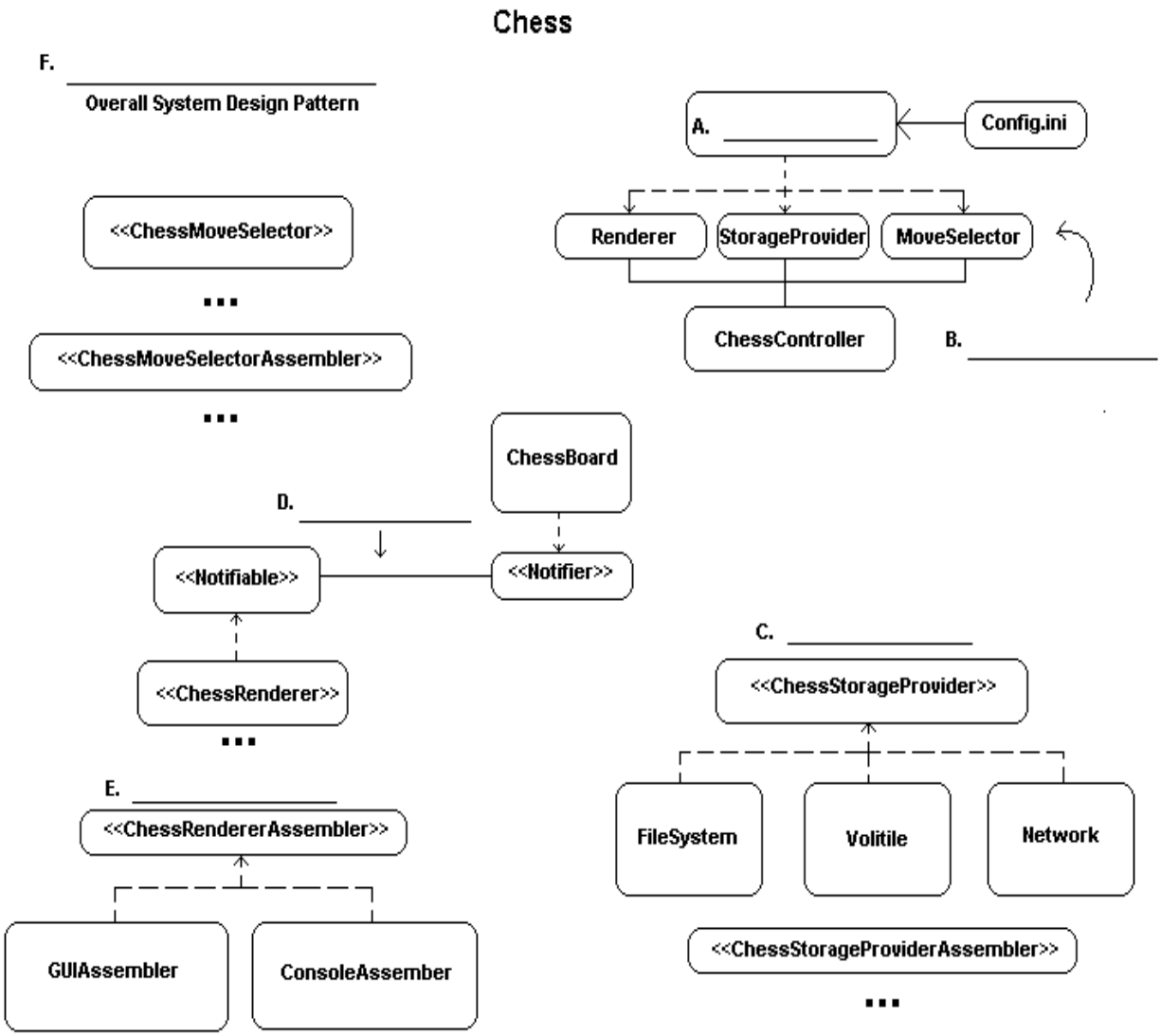
    A. Model, View, Controller
    B. Configurator Pattern
    C. Interface Segregation Principle
    D. Service Locator Pattern

6. A program requires its display to be separated from the program's logic.

    A. Factory Pattern
    B. Messenger Pattern
    C. Model, View, Controller
    D. Observer Pattern

Section 2

Fill in each blank in the system diagram (labeled A-F) with its descriptive term from the word bank below. There are terms in the word bank that will not be used.



Word Bank

1. Interface	2. Interface Segregation Principle	3. Dependancy Pattern
4. Model, View, Controller	5. Service Locator Pattern	6. Factory Pattern
7. Messenger Pattern	8. Observer Pattern	9. Assembler Pattern

4.2.2 Results

The average result the students received on the exam was an 83.3%. The couple of students that took the exam all returned the same answers. Since the exam was short in an effort to not make them panic and the fact that there was a small amount of students there simply is not enough data to determine if they cheated or not. However, even if they did cheat they would have to have collectively understood 83.3% of the material. The questions were arranged so that the students could not infer answers from other questions and limited guessing. The results only show the promise that the method could be effective. However, in order to gain a valid statistic a future study would have to be made.

In addition to the course, a single 3 hour review of the entire curriculum was given a few days prior to the exam. This may have contributed to the success of the project. The students in the review demonstrated a surprising amount of retention from the course. However, it seemed as though they had to be encouraged to activate it. Either they were unable or were unwilling to activate the knowledge just from the course alone.

This may be due to the students not elaborating the designs presented in their own words because the method gave room for no leeway in design. Mayer believes that elaboration helps in the activation of the student's knowledge. The method may be improved with a better way of incorporating elaboration without causing the students to panic.

Another reason that they have been lacking in the activation of knowledge is because they have underdeveloped learning skills at that age. Students in high school as seen with the previous method have a hard time with actively learning new information on their own. They just want to be over and done with their classes, even an interesting one. Not one of the students went out on their own and looked up a better way to program. The instructor could attempt to increase the level of interest in their class or it could just be that teaching students advanced concepts in novice courses at the high school level is impractical with the current system.

There are some areas in the project that could be improved for future work. For example, the results may contain some error just because of the small sample size. Another possible source of error is that the second method took place only after the first method failed. This means that the students were possibly corrupted by the first method thereby skewing the results of the second.

Future work should perform the second method on at least 30 students for the results to be statistically sound. That study would support that the use of the methods in this project could be effectively used to educate students. Another study to consider would be to have a much larger sample size of 100 students. 50 students can be used as a control and 50 students as a test group. That study would produce the result of how much more effective the methods of this project is compared to the traditional methods. Another interesting study to do would be to perform a follow up on those students 10 years later to examine the long term effects of such an education.

5 Conclusion

The purpose of the project was to demonstrate that there is a possibility for novices to learn object oriented programming with design integrated. The usefulness of this is to improve a student's progression through school. The traditional approach reserves design for later classes and as a result cause novice students to learn bad designs and are forced to relearn everything over again. It is possible that changing this progression will allow a student to progress further with later classes. However, long term effects of this approach are not the goal of this project.

The first method which was to instruct the students with a large, open-ended project failed to yield results. The students were overwhelmed and had difficultly with focusing on the design part of the project. It is apparent that at least at the high school level that open-end, large assignments are not an effective method. Also they had difficulty picking out concepts through experience alone. Students need the instructor to explicitly state the concepts they should learn.

The second method which was to instruct the students with a small, strongly guided project produced favorable results. The project was revised to handle all of the short comings with the larger project. As a result it was able to present results which could indicate that effective understanding in the students is possible with this method. The only short coming was that since elaboration was taken out, students had a hard time with activating the knowledge learned. There could be room for improvement in areas such as that if the method was to be tried again with a larger audience.

Given the results from the second method, the premise that students can learn object oriented programming with integrated design is still plausible. This project justifies a larger study to gather statistically sound results with a larger audience. The method should be refined further for such a study such as incorporate more elaboration from the students. A set of graphical elements would also be advised. The students will have a much easier and faster time understanding the concepts if the instructor had a good set of pictures to back it up with. However, this also may put them to sleep in terms of reception if the pictures are not appropriate for their age level. Along the same lines, more immersion in the story behind the assignments using elements such as acting and props could be beneficial to learning. Finally, the age group should possibly include students that are past the high school level. College novices may benefit more from the class as they already possess the methods and maturity for actively learning. This will eliminate any confounders when trying to determine if the method was truly at fault.

The short, guided project course shows promise in educating novices object oriented programming with integrated design. Determining if this is an effective method with further studies may help motivate a change in curriculum if the long term benefits are found desirable enough.

6 Bibliography

Blackwell, A.F. (2006). Metaphors we program by: Space, action and society in Java. *Proceedings of PPIG 2006*, pp. 7-21. http://www.ppig.org/papers/18th-blackwell.pdf

Fix, V., Wiedenbeck, S., and Scholtz, J. (1993). Mental representations of programs by novices and experts. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems* (Amsterdam, The Netherlands, April 24 - 29, 1993). CHI '93. ACM, New York, NY, 74-79. DOI: http://doi.acm.org/10.1145/169059.169088

Fowler, M., Inversion of Control Containers and the Dependency Injection Pattern. *Martin Fowler's Website.* [Updated Jan. 2004, Cited May 2010]. Available from: http://martinfowler.com/articles/injection.html

Freeman, E., Freeman, E., Bates, B., and Sierra, K. 2004 *Head First Design Patterns*. O' Reilly & Associates, Inc.

Martin, C. R. (1996). The Interface Segregation Principle. *Object Mentor Inc.* 2006, http://www.objectmentor.com/resources/articles/isp.pdf

Mayer, R. E. (1981). The Psychology of How Novices Learn Computer Programming. *ACM Comput. Surv.* 13, 1 (Mar. 1981), 121-141. DOI: http://doi.acm.org/10.1145/356835.356841