SE 464 - Answers to Previous Exams

1005 - Final Exam
Question 1a
We have used the term “program to an interface, not to an implementation”.
Why is this desirable?

Answer:
Following this principle leads to code that exhibits loose coupling among classes. Examples of these benefits include:

a) Better testability. Classes that are not under tests can be substituted with fakes that implement the same interface. I.e you could substitute AJAXLibrary with FakeAJAX if the class just expects anything that implements an interface RemoteCallLibrary.

b) Better code reuse. If you have a routine that can sort anything that implements Comparable interface, you can now sort almost anything pretty easily instead of writing sorts for every type of object.

Question 1b
We also used the term “favor object composition over class inheritance”.
Explain this concept and indicate what possible advantages there may be. In your discussion explain what object composition is, discuss the advantages and disadvantages of class inheritance and the possible advantages of object composition.

Answer:

Two common advantages of Composition over inheritance:
Composition allows for code reuse even if two objects don't fit into a class taxonomy. In a video game, there could be a Monster class, Observer Camera class, and some utility search class. They can all maintain (be composed of) a strategy object to figure out how to traverse the level. But they don't have (not do they need) a shared ancestor. Fitting them into a taxonomy with a shared superclass is awkward and complicates the taxonomy by another level.

Composition allows to vary the behaviour of the object at runtime. Using the example of strategy above, the strategy of the Monster traversing the map can change depending on inputs by the user during the lifetime of a Monster instance.

Disadvantage:
Certain problems have a class hierarchy map well to a domain taxonomy. If different types of bank loans form a hierarchy of types in the problem domain, it might be awkward to split up a concept of a loan into pieces that compose each other. Inheritance thus would better represent the domain problem.

Question 2
We discussed architectural styles, architectural design patterns and basic design patterns. Explain the difference between these concepts and when (during the development cycle) they could be used. Why would one even consider applying these methods? Is there any useful result in the short or long term for the software system?

Architectural styles: Broad approaches to solving problems (pipes & filters, object-oriented, tiers, repository, etc.) that define which underlying tools are available to be used by patterns. These must be decided early, as they provide the general approach that is implemented.

Architectural design patterns: Higher-level than design patterns, describe construction of system as a whole and links between components. These should be decided early as they can have far-reaching implications. e.g. Should tiers be accessed through a façade?

Basic design patterns: Refer to how the internals of a component are arranged in order to solve problems. These are decided on as the problem they solve arises. e.g. An iterator is introduced when an operation must be applied to every component of a data structure.

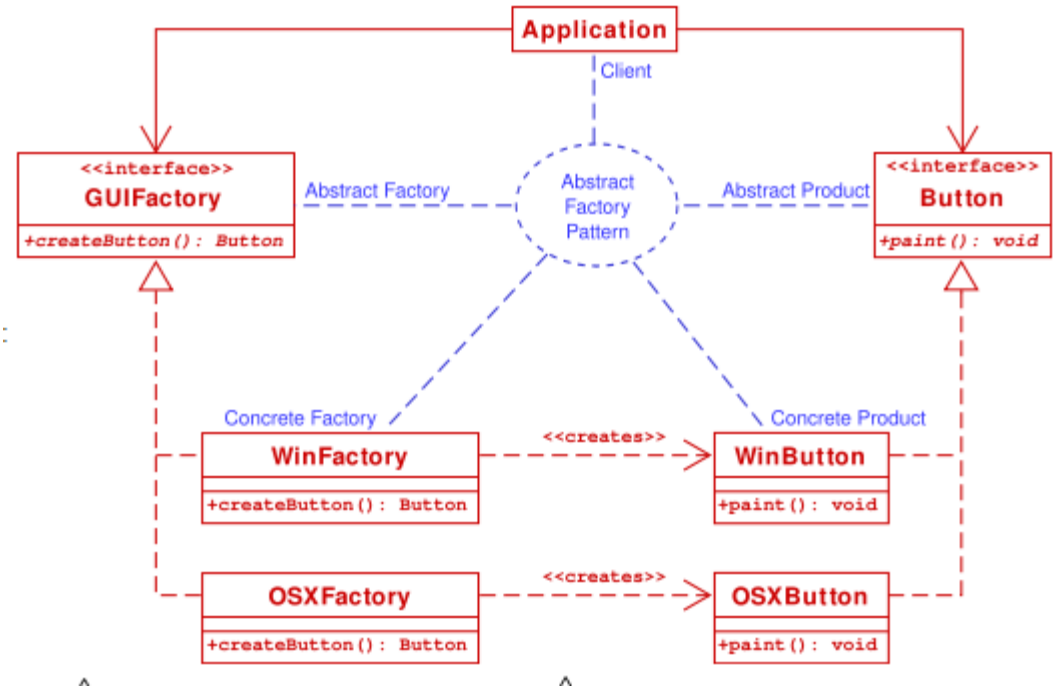
Rayside's opinion is that these are all design patterns.

Design patterns specify a general solution schema for problems that is known to have certain desirable properties. In the short term, the decision to use a design pattern provides a clear procedure for implementation. In the long term, the use of a standard design pattern provides a clear structure for maintenance programmers to help understand the code as well as use whatever extensibility procedures are available to the pattern.

3.

The **abstract factory** pattern is a software design pattern that provides a way to encapsulate a group of individual factories that have a common theme. In normal usage, the client software creates a concrete implementation of the abstract factory and then uses the generic interfaces to create the concrete objects that are part of the theme (courtesy of Wikipedia: http://en.wikipedia.org/wiki/Abstract_factory_pattern).

Essentially, the abstract factory pattern provides an interface for creating families of related or dependant objects, without exposing their concrete classes.



(http://en.wikipedia.org/wiki/Abstract_factory_pattern)

This pattern allows the client to ask a Factory to create an instantiation of the concrete class (in this case a button).

Pros:

- Client code has no knowledge of the concrete class (no need to include a header file, etc.). Instead a writer of client code would just use the abstract interface.
- Adding or changing to a new concrete type would involve simply using a different factory. This can be automated using configuration files, etc. to allow for multi-system compatibility (especially in this GUI example). See example below:

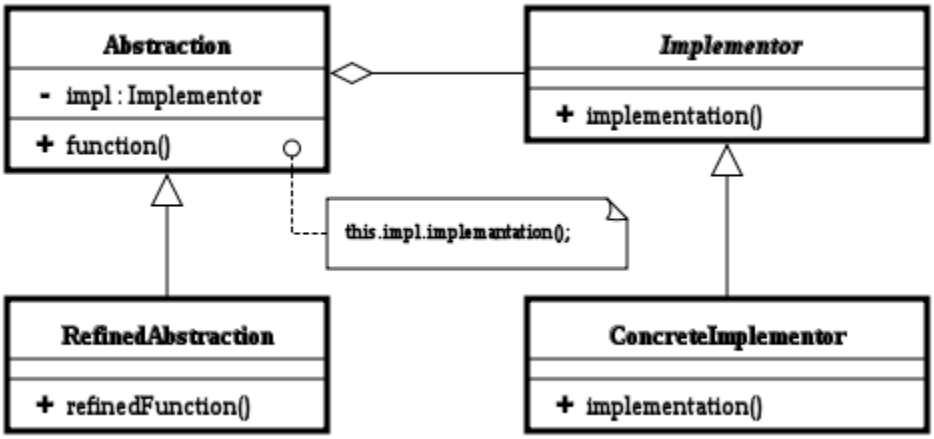
Cons:

- Usual overhead due to layers of indirection
- Concrete types must behave identically as far as the application is concerned

```
public static GUIFactory createOsSpecificFactory() {
      int sys = readFromConfigFile("OS_TYPE");
      if (sys == 0) {
         return new WinFactory();
      } else {
         return new OSXFactory();
      }
}
```

A factory created by this method could be used to create a button of either concrete type (depending on what was in the config file).

The **bridge pattern** is a design pattern used in software engineering which is meant to *"decouple an abstraction from its implementation so that the two can vary independently"*. The *bridge* uses encapsulation, aggregation, and can use inheritance to separate responsibilities into different classes (http://en.wikipedia.org/wiki/Bridge_pattern).



(http://en.wikipedia.org/wiki/Bridge_pattern)

In the example given on Wikipedia, the abstraction is a shape and the refined abstraction is a specific type of shape (circle, square, etc.). The constructors for these shapes take an Implementor as a constructor argument (one of many concrete implementors) that wraps specific GUI API calls with a single interface used by each abstraction.

Pros:

- The implementation and abstractions can be changed independently of each other.
- Easy to make code changes if if one is not familiar with the entire code base.

Cons:

- Often confused with the Adaptor pattern (and according to Wikipedia most often implemented using the adaptor pattern).

4.

Abstract Factory

Since the structure of the tree is the same regardless of the traversal type, we can use the abstract factory pattern to create objects that traverse trees in different manners. Taking the answer to question 3 as a reference: Instead of having an interface for button, we would have an interface for traverse, for which a method traverse(Tree tree) would exist. The classes that inherit from this interface would be InOrderTraversal, PostOrderTraversal, etc. These classes would implement the various traversals of a tree.

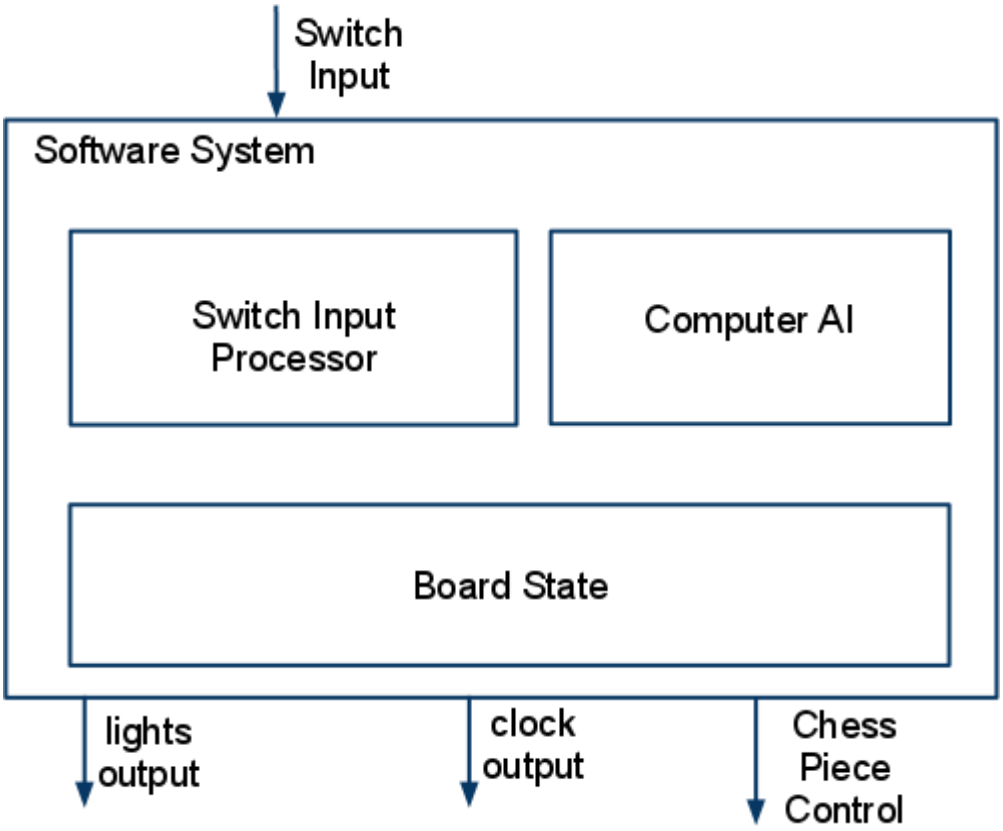
From here we would have an OrderFactory interface which would again have factories for each type of traversal that inherit from the interface, which in turn create the concrete objects required to traverse a tree in the desired way.

Here we could assume that the tree is given to us already created and we simply pass it into the traverse method of the created Traversal. However, we can apply this pattern once more to tree creation. For example, we could have a factory for each type of node. Creation methods for addition nodes for example would take two child nodes, and the factory would return an addition node with two children which could later be added together. From here we would create the tree from the bottom up; when we have created the root node, we can pass this into the Traversal created by a factory above.

Bridge

To use the bridge pattern, first consider the tree as the abstraction, and the action being performed on it as the implementation (which includes printing, and actually doing the math). Then consider that the implementation would be a traversal with concrete classes implementing specific traversals. Now for the abstraction, we could have a node as the basic abstraction and then refine it with specific types of nodes (multiplication, addition, etc.). In creation, we would construct a node and pass in any required children nodes. This means that we would create the bottom up again, since we would need to pass the children nodes into the constructor of a parent node. A node could be passed its implementor and when traverse is called on the root. The concrete implementer will run its course.

Question 5



Overall, the system uses a layer architecture. The bottom layer is the Board State component which provides the current state of the board to the Switch Input Processor and Computer AI components. Implicit Invocation can be used to communicate board state changes to the other components and to the outputs of the system.

The Switch Input Processor can be implemented using a pipe and filter style, with the first filter discarding user input that are invalid moves and the second filter processing a valid move.

Question 6

The Model-View-Controller pattern divides the system into the three components. The model is is responsible for controlling the access to the data. The controller contains the business logic of the application. The view is responsible for generating presentation from the controller and the model.

In the above diagram, the board state acts as the model of the system. The switch input processor and computer AI would fall into the controller component. since they contain the “business logic” of the application. The views are the outputs of the system.

In general, the MVC pattern has the benefit of separation of concerns and modularization. Each component can be update without changes to the other components. A classic sample would be adding multiple views to the system. However, in this chess game example, it is not clear that which component, if any, will change in the future. So aside from the general benefits from modularization, there is no clear benefit from using MVC pattern.