UECS2344 Software Design: Lecture 9

Design Patterns

Before looking at design patterns, consider the idea of **patterns** in terms of code.

Example 1: Suppose we want to write code that displays a menu of options, accepts the user's choice, validates it, and then performs some processing according to the user's choice.

Here is a template of pseudocode for achieving this:

```
do {
     // display the menu
     print "1. xxxx"
     print "2. xxxx"
     ...
     print "n. Exit"

     // get the user's choice
     print "Enter your choice"
     input choice

      // validate input choice
      while (choice < 1 || choice > n) {
           print "Invalid choice. Please enter again"
           input choice
      }

     // perform some processing according to user's choice
     switch (choice) {
         case 1: call func1(...)
                 break
         case 2: call func2(...)
                 break
         . . .
         case n: break
     }
} while (choice != n)
```

Example 2: Suppose we want to write code to search an array for a particular value.

Here is a template of pseudocode for achieving this:

```
bool found = false
int i = 0

while (i < sizeOfArray && !found) {
  if (searchValue == arrayElement[i].value)
         found = true;
     else
         i++
}
```

These example pseudocode solutions could be used as a template for other similar problems.

Design Patterns

Design Patterns

- are known and well-established solutions applicable to common problems
- might need adaptation to specific problem context
- can be applied many times in many slightly different problems
- provide a way to talk about a design
- provide a way to reuse knowledge and experience of other designers

"Patterns and Pattern Languages are ways to describe best practices, good design, and capture experience in a way that it is possible for others to reuse this experience" (taken from http://www.hillside.net/patterns)

A group of four famous software experts

- called 'Gang of Four' (GOF) (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides)

described design patterns in a book

- called 'Design Patterns: Elements of Reusable Object-Oriented Software'

where the design patterns are divided into three categories:

- **Creational Design Patterns**
  1. **Singleton Pattern**
  2. Factory Pattern
  3. Abstract Factory Pattern
  4. Builder Pattern
  5. Prototype Pattern

- **Structural Design Patterns**
  1. **Adapter Pattern**
  2. **Composite Pattern**
  3. Proxy Pattern
  4. Flyweight Pattern
  5. **Facade Pattern**
  6. Bridge Pattern
  7. **Decorator Pattern**

- **Behavioural Design Patterns**
  1. **Template Method Pattern**
  2. Mediator Pattern
  3. Chain of Responsibility Pattern
  4. **Observer Pattern**
  5. **Strategy Pattern**
  6. Command Pattern
  7. State Pattern
  8. Visitor Pattern
  9. Interpreter Pattern
  10. Iterator Pattern
  11. Memento Pattern

Design Pattern Template

Four essential elements used to describe design patterns are:

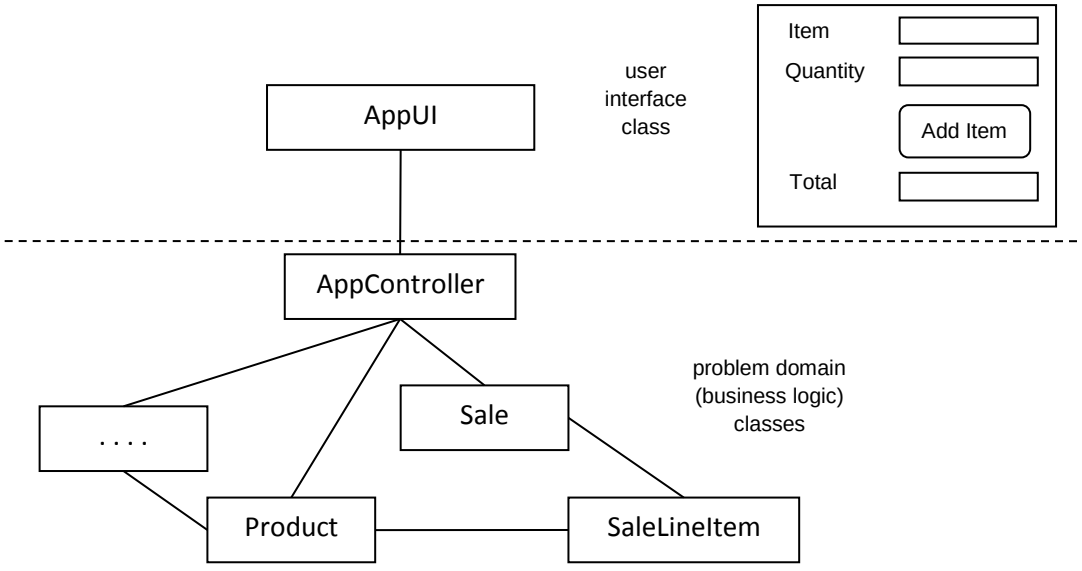
1. A **name** that is a meaningful reference to the pattern.
2. A **description** of the problem area that explains when the pattern may be applied.
3. A **solution description** of the parts of the design solution, their relationships and their responsibilities. This is not a concrete design description. It is a template for a design solution that can be instantiated in different ways. This is often expressed graphically and shows the relationships between the objects and object classes in the solution.
4. A **statement of benefits and consequences** - the results and trade-offs – of applying the pattern. This can help designers understand whether a pattern can be effectively applied in a particular situation.

Simple Design Patterns

Controller Pattern (not in patterns list from Gang of Four)

Name:	Controller
Problem:	Domain or entity classes have the responsibility for realising or achieving use cases. However, since there can be many domain classes, which one should be responsible for receiving the input messages from the user interface or boundary classes? User interface classes become very complex if they have access to all domain classes. How can the coupling between the user interface classes and the domain classes be reduced?
Solution:	Assign the responsibility for receiving input messages from user interface classes to a class that receives all input messages and acts as a coordinator to forward (delegate) the messages to the correct domain classes. There are several ways to implement this solution: (a) Have a single class that represents the entire system, or (b) Have a class for each use case or related group of use cases to act as a use case handler.
Benefits and Consequences:	Coupling between the boundary classes and the domain classes is reduced. But, if not careful, business logic will be inserted into the controller class.

Controller Pattern Example



Singleton Pattern (one of the Gang of Four patterns)

Name:	Singleton
Problem:	Only one object of a class is allowed to exist. Several parts of an application may request creation of an object of the class. The first request should create a new object. Later requests will not create new objects of the class but instead they will receive the single already created object.
Solution:	A singleton class has a static variable that refers to an object of itself. All constructors are made private. A getInstance() method checks if an object already exists. If it does not, an object is created and a reference to the object is returned. Otherwise, it just returns the reference to the object without creating any other objects.
Benefits and Consequences:	The singleton controls itself to ensure that only one object is created. There are other times when only one object of a class is needed, but if it is created from only one place, then a singleton may not be required.

Singleton Pattern Example

We want one and only one instance of the DatabaseConnection class in the application.

```
public class DatabaseConnection {

    private static DatabaseConnection instance;

    public static DatabaseConnection getInstance() {
        if (instance == null) {
            instance = new DatabaseConnection();
        }
        return instance;
    }

    // private constructor
    private DatabaseConnection() {

    }

    . . .
}
```

The first time the static method *getInstance()* is called:

```
DatabaseConnection connection = DatabaseConnection.getInstance();
```

the static variable *instance* will be null. So a new *DatabaseConnection* object is created and the object is assigned to static variable *instance*. This object is returned.

The next time the static method *getInstance()* is called, static variable *instance* will already have an object assigned and so that same object is returned.