Software Tools & Methods
Class 5

Lecturer: Greg Bolcer, greg@bolcer.org
Summer Session 2009
ELH 110 9am-11:50am
Overview

• Reading for today was on UML, Larman
• Last Class: UML
• Up next
  – Quick Review
  – More UML
  – Class, Object, Package diagrams
  – Design Patterns
Software Tools and Methods

REVIEW
Iterative Development

• The development is carried out as a series of stages, each with its own design, implementation and testing phase.
• The output of each stage is a working, production-quality system but with reduced functionality.
• The customer is involved in evaluating each of the stages.
• Can accommodate changing requirements.
Iterative Development

• System is defined by use cases
  – A “use case” is a major way of using the system, or a major type of functionality

• High level planning needs to
  – Define what are the major use cases
  – Determine in what order they will be done
  – Estimate development time for each use case (“timeboxing”)

Larman
p. 15
Agile development

• A family of iterative development techniques that emphasise response to changing requirements (the opposite of the waterfall model)

• Makes heavy use of UML for sketching.
  - Usually by a group of developers using whiteboards.
  - The main purpose of using UML in this way is to understand the problems and solutions rather than to document them.
Software Methods & Tools

MORE UML
UML Goals

• The goals of UML are:
  – To model systems using OO concepts
  – To establish an explicit coupling between conceptual and software artifacts (objects)
  – To address the issues of scale inherent in complex mission critical systems
  – To create a modeling language usable by both humans and machines
UML Method

• A method needs a language, and a process to describe how to use the language
  Method = Language + Process

• The Rational Unified Process (RUP or UP) was designed to be used with UML
  – UP is an iterative process
  – Provides a structure for system development
UP Phases

- Inception
- Elaboration
- Construction
- Transition

Larman
p. 19
More in Larman

• Keywords («guillemets»)
  – Stereotypes
  – Examples: calls, interface, permit,…

• Responsibilities
  – Another compartment in class
  – Items prefixed by --

• Template classes
Examples

1. A dependency on calling on operations of the operations of a Clock

2. A dependency that A objects create B objects

**DataAccessObject**

- id : Int
- ... doX()
- ... exceptions thrown
  - DatabaseException
  - IOException
- responsibilities
  - serialize and write objects
  - read and deserialize objects
  - ...

**Clock**

- getTime()
- ...

**Window**

- «call»

**A**

- «create»

**B**

- ...

**Timer**

- «interface»
- getTime()
Hints for Class Diagrams

• Remember: models are for communication
• Remember: include only important stuff
• How do I find classes, attributes and so on?
  – Classes often correspond to nouns
  – Associations often correspond to verbs
• A class should
  – Represent a coherent concept
    • Principle: Low Coupling, High Cohesion
  – Have a small, well-defined set of responsibilities
  – Be named with a singular noun that says what each instance of the class is
  – Have no more than 10-20 operations
Hints for Class Diagrams

• Class diagrams should
  – have a single purpose
  – have a title that expresses the purpose
  – show only things that are relevant for this purpose

• Avoid
  – cyclical dependencies, if possible
  – generalization hierarchies with more than 5 levels
  – crossing edges
Hints for Class Diagrams

• Use colors judiciously
  – to highlight and group things
  – unless you have to print it in black-and-white!

• Lay out classes in a meaningful way
  – similar classes close to each other
  – top: closer to the user, bottom: closer to the data structures
Object Diagrams

• Show instantiation or specification of classes
• Associated with a particular use or instance of the model
• Differences between Classes and Objects
  – Name: class is underlined
  – Attributes and operations included as needed
  – Fields have data added
• Useful for showing interactions between interfaces, abstract classes, etc.
  – Where functionality is not clear until instantiation
Example: Class to Object Diagrams

- Person
  - owns
  - 0..1
- Toaster
  - n

- Cinderella
  - owns
  - :Person
- ProctorSilex
  - owns
  - :Toaster
Package Diagrams

• Package is a grouping construct
  – Most commonly used for class diagrams, but can be used with any UML diagram or elements
  – Used to create a hierarchy or higher level of abstraction
  – Corresponds to package in Java

• Each package represents a namespace
  – Like Java, can have classes with same name in different packages
Representing Packages

- Contents listed in box
- Contents diagramed in box

- Fully qualified package name
- Nested packages
- Fully qualified class name

Week 5, Slide 20
public class SalesLineItem
{
    private int quantity;

    private ProductDescription description;

    public SalesLineItem(ProductDescription desc, int qty) { ... }

    public Money getSubtotal() { ... }

}
Code to UML

public class SalesLineItem
{
    private int quantity;
    private ProductDescription description;

    public SalesLineItem(ProductDescription desc, int qty) {
        // Constructor implementation
    }

    public Money getSubtotal() {
        // Method implementation
    }
}

SalesLineItem
quantity : Integer
getSubtotal () : Money

ProductDescription
description : Text
price : Money
itemID : ItemID
...
public class SalesLineItem extends BasicLineItem implements Item {
    private int quantity;
    private ProductDescription description;
    public SalesLineItem(ProductDescription desc, int qty) {
    }  
    public Money getSubtotal() {
    }
}
public class Sale {
    ...
    private List lineItems = new ArrayList();
}

A collection class is necessary to maintain attribute visibility to all the SalesLineItems.

Code to UML
Software Tools & Methods

DESIGN PATTERNS
Design Patterns

• Reusable design component
• First codified by the Gang of Four in 1995
  – Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
• Concept taken from architecture
  – “A Pattern Language” by Christopher Alexander
  – “…a three-part rule, which expresses a relation between a certain context, a problem, and a solution.”
• Original Gang of Four book described 23 patterns
  – More have been added
  – Other authors have written books
Design Patterns Template

- **Context**
  - General situation in which the pattern applies

- **Problem**
  - The main difficulty being tackled

- **Forces**
  - Issues or concerns that need to be considered. Includes criteria for evaluating a good solution.

- **Solution**
  - Recommended way to solve the problem in the context. The solution “balances the forces”

- **The following are optional**

- **Antipatterns**
  - Common mistakes to avoid

- **Related Patterns**
  - Similar patterns; could be alternated solutions or work with the pattern

- **References**
  - Source of pattern
  - Who developed or inspired the pattern
Gang of Four Design Patterns

• Creational Patterns
  – Abstract Factory
  – Builder
  – Factory Method
  – Prototype
  – Singleton

• Structural Patterns
  – Adapter
  – Bridge
  – Composite
  – Decorator
  – Façade
  – Flyweight
  – Proxy

• Behavioral Patterns
  – Chain of Responsibility
  – Command
  – Interpreter
  – Iterator
  – Mediator
  – Memento
  – Observer
  – State
  – Strategy
  – Template Method
  – Visitor
Patterns in Java

• Chain of Responsibility
  – Exception handling
  – Try/catch/throw blocks

• Iterator
  – Container classes

• Observer
  – Listeners in GUIs
Gang of Four Design Patterns

• Creational Patterns
  – Abstract Factory
  – Builder
  – Factory Method
  – Prototype
  – Singleton

• Structural Patterns
  – Adapter
  – Bridge
  – Composite
  – Decorator
  – Façade
  – Flyweight
  – Proxy

• Behavioral Patterns
  – Chain of Responsibility
  – Command
  – Interpreter
  – Iterator
  – Mediator
  – Memento
  – Observer
  – State
  – Strategy
  – Template Method
  – Visitor
The Observer Pattern

• Context
  – When an association is created between two classes, the code for the classes becomes inseparable.
  – If you want to reuse one class, then you also have to reuse the other.

• Problem
  – How do you reduce the interconnection between classes, especially between classes that belong to different modules or subsystems?

• Forces
  – You want to maximize the flexibility of the system to the greatest extent possible
The Observer Pattern

```
- Observable
  - addObserver
  - notifyObservers

- ConcreteObservable

- Observer
  - update

- ConcreteObserver

- Observable

- Observer

- Forecaster
  - Observers are notified when a new prediction is ready

- WeatherViewer

Week 5, Slide 33
```
Observer

• Antipatterns (Don’t do this)
  – Connect an observer directly to an observable so that they both have references to each other.
  – Make the observers subclasses of the observable.

• Reference
  – Gang of Four
Observer in Java

• Observer interface and Observable class exist
  – java.util.Observer and java.util.Observable

• But people usually implement their own
  – Usually can’t or don’t want to sub-class from Observable
  – Can’t have your own class hierarchy and multiple inheritance is not available
  – Has been replaced by the Java Delegation Event Model (DEM)
    • Passes event objects instead of update/notify

• Listener is specific to GUI classes
The Façade Pattern

• Context
  – Often, an application contains several complex packages.
  – A programmer working with such packages has to manipulate many different classes.

• Problem
  – How do you simplify the view that programmers have of a complex package?

• Forces
  – It is hard for a programmer to understand and use an entire subsystem.
  – If several different application classes call methods of the complex package, then any modifications made to the package will necessitate a complete review of all these classes.
The Façade Pattern

• Solution
The Façade Pattern

• Solution
  – Provide a simple interface to a complex subsystem.
  – Decouple the classes of the subsystem from its clients and other subsystems, thereby promoting subsystem independence and portability
Using the Façade Pattern

• Hides implementation details
• Promotes weak coupling between the subsystem and its clients.
• Reduces compilation dependencies in large software systems

• Does not add any functionality, it just simplifies interfaces
• Does not prevent clients from accessing the underlying classes.
Façade Example
The Singleton Pattern

• Context
  – It is very common to find classes for which only one instance should exist (singleton)

• Problem
  – How do you ensure that it is never possible to create more than one instance of a singleton class?

• Forces
  – The use of a public constructor cannot guarantee that no more than one instance will be created.
  – The singleton instance must also be accessible to all classes that require it
The Singleton Pattern

• Solution

```
Singleton
static Instance()
SingletonOperation()
GetSingletonData()

static uniqueInstance
singletonData
```

return uniqueInstance
Singleton

- Example

  Pattern

  | «Singleton» |
  | theInstance |
  | getInstance |

  Instantiation of Pattern

  | WindowMgr |
  | theWindowMgr |
  | WindowMgr «private» getInstance |

  This is the code for getInstance

  ```
  if (theWindowMgr==null)
  theWindowMgr= new WindowMgr();
  return theWindowMgr;
  ```

  Constructor for WindowMgr is private
  getInstance is public and static
  theWindowMgr is private and static
public class WindowMgr {
    private static WindowMgr theWindowMgr;
    private String windowLabel;

    private WindowMgr (){ }
}

// Lazy instantiation
public static synchronized WindowMgr getInstance(){
    if (theWindowMgr == null){
        theWindowMgr = new WindowMgr();
    }
    return theWindowMgr;
}

...
public class WindowMgr {
    // Eager instantiation
    private static WindowMgr theWindowMgr = new WindowMgr();
    private String windowLabel;

    private WindowMgr (){ }

    public static synchronized WindowMgr getInstance(){
        return theWindowMgr;
    }

    ...
}

Week 5, Slide 45
Questions

• Why do you need the getInstance method? Why isn’t it enough to just make the WindowMgr static (i.e. one per class)?
  – This results in extra instances of WindowMgr, but still only one underlying theWindowMgr

• Why do you need an instance of WindowMgr at all? Why not just make all the methods static?
  – May need an instance, e.g. as an observer, for callbacks
  – More flexible when you discover later that you don’t want WindowMgr to be a singleton any more
Drawbacks

• Need to add synchronization to getInstance
  – Race condition could occur in if block
• Sub-classing becomes complicated
  – Private constructor violates normal Java design principles
  – Could change constructor to protected, but that would violate the security provided
    • Make a sub-class that is identical to parent
    • Can have lots of pseudo-WindowMgrs running around
  – Alternatively, each sub-class has own getInstance method
• Also need to prevent cloning by overriding Cloneable interface
• Erich Gamma doesn’t like Singleton any more
Singleton Design Pattern

• Related Patterns
  – Factory and Façade

• Reference
  – Gang of Four