CS3505/5020
Software Practice II

Team Stand-Up Meeting
Brief look into Scrumworks
Design Patterns
Stand-Up Meeting

- Meet with your team for 10 minutes at the start of class.
  - Each team member should state what they have accomplished since your last meeting
  - Each team member should describe what they are working on next, and if there are any barriers to completion
- After, fill out the stand-up meeting worksheet
  - List only people who are in class
  - Answer the questions on the sheet
  - Turn it in
- Finally, teams may reschedule their weekly appointment, if needed
  - Class will resume at 2:15.
A Brief Look into Scrumworks Progress

• At this point you should have your first sprint populated
  – Teams should have all three sprints populated by the end of the week

• Look at your burn chart – what progress are you making?
  – It should be a downward slope headed towards the bottom by the time of your next TA meeting.

• For teams with no progress:
  – You are in danger of not earning the points for the first sprint. (Don’t just cram it in in one evening – make this technology work for you!)
Design Patterns

What makes an expert programmer an expert?

- Experts can repeat their success quickly when faced with a programming problem.
  » How?

- Experts know what solutions to a problem will succeed, and they reuse those solutions.

- Experts can predict the outcome of a body of code, knowing that they have made a design error before they test their code.
  » Have you ever had the feeling that ‘this is a bad idea’ when you’re coding?
Design Patterns

- What makes an expert programmer an expert?
  - To sum up, experts use design patterns.
    » They (we?) do not solve every problem from first principles – they reuse what they’ve learned.
  - Good patterns stem from known successful solutions.
  - Antipatterns arise from failures.
Design Patterns Defined

- A design pattern is a named description of a solution design for a general design problem in a specific domain. What?
  - It’s a reusable outline of a solution!
  - It specifies high-level design, not algorithms or code

- A good design pattern describes:
  - The domain in which the problems occur
  - The problems to which the pattern applies
  - The software organization of the solution
  - The consequences of the pattern
Suppose you have an abstract application that manages documents (sources of data) and windows (which display data):

```
Window
Application
+open(name)
Document
```
A Design Pattern Example

- Assume you subclass these to build two coexisting applications, perhaps as part of an IDE:
  - Also note that the Application class is abstract. Every instance will be a TextApp or ImageApp.

```
Window
  +open(name)

Application

Document

TextWin ImageWin

TextApp ImageApp

TextDoc ImageDoc
```
A Design Pattern Example

- Now, open is called with a filename.
  - What would the code look like in ‘open’?
  - How would you know what type of Doc objects to create?
  - What if additional subclasses are added later?
A Design Pattern Example

- **The problem:**
  - A base class knows *when* to create objects, but not *what type* of objects to create.

- **One possible solution:**
  - Simple. Use ‘if’ statements to determine type, and act accordingly within the superclass:

  ```java
  Inside of Application open method
  if (this is TextApp)
    doc = new TextDoc(name);
  else
    doc = new ImageDoc(name);
  ```
A Design Pattern Example

- Bad. Bad, bad, bad.
  - The base class must be changed for every additional subclass
  - The base class is bound to the implementation details of every subclass
  - An antipattern example – avoid making superclasses dependent on subclasses implementations

Inside of Application open method

```java
if (this is textApp)
    doc = new TextDoc(name);
else
    doc = new ImageDoc(name);
```
A Design Pattern Example

- Another solution:
  - Override the open method in each subclass

  \[
  \begin{align*}
  \text{Inside of TextApp open method} & : \quad \text{doc} = \text{new TextDoc(name);} \\
  \text{Inside of ImageApp open method} & : \quad \text{doc} = \text{new ImageDoc(name);} \\
  \end{align*}
  \]

- Better, but still not great:
  - Subclasses must be able to change the ‘doc’ variable in the superclass. This may allow subclasses to violate an invariant in the superclass.
  - Subclasses must either replicate all the ‘open’ functionality from the superclass, or rely on a ‘super’ call to do that work.
    » This is the ‘super call’ antipattern – more another day
A Design Pattern Example

- The **factory method** pattern:
  - The superclass declares an abstract method for building the needed object
  - The subclasses each override the method for building the object.

```
Application
+open(name)
+makeDoc(name)

TextApp
+makeDoc(name)

ImageApp
+makeDoc(name)
```
**A Design Pattern Example**

- The **factory method pattern**:  
  - The superclass declares an abstract method for building the needed object  
  - The subclasses each override the method for building the object.

```
Application
+open(name)
+makeDoc(name)

TextApp
+makeDoc(name)

ImageApp
+makeDoc(name)
```
A Design Pattern Example

- The code in the superclass now has no knowledge of what will be built, but can control all other aspects of the ‘open’ method call.

`Inside of Application open method`

```java
    doc = makeDoc(name);
```

- It works because ‘this’ is a subclass object and the correct makeDoc(name) method will be called.
Factory methods

- The factory method pattern can be further generalized into abstract factory pattern.

- In the abstract factory pattern, the subclasses create a factory object (from some abstract class) to create further objects.

\[
\begin{align*}
\text{Application} & \quad +\text{open}(name) \\
& \quad +\text{makeFactory}() \\
\text{Factory} & \quad +\text{makeWin}(name) \\
& \quad +\text{makeDoc}(name) \\
\text{TextFactory} & \quad +\text{makeWin}(name) \\
& \quad +\text{makeDoc}(name) \\
\text{ImageFactory} & \quad +\text{makeWin}(name) \\
& \quad +\text{makeDoc}(name) \\
\text{ImageApp} & \quad +\text{makeFactory}() \\
\text{TextApp} & \quad +\text{makeFactory}()
\end{align*}
\]
Factory methods

- Many variations of factory method patterns exist:
  - Factory Method – seen it
  - Abstract Factory – seen it
  - Builders – methods that remove the complex details for building objects
  - Prototypes – copy an existing object to get a new instance of the correct type – rarely used in practice
Question

- ‘Design patterns’ look like various abstractions. What’s the difference?
  - None really
  - The goal is not to memorize a cookbook of useful tidbits, but to recognize patterns when you see them and reuse them when appropriate.
  - Many patterns exist because language support for some idea is weak. Consider...
Another pattern: Observers

- Imagine you have three GUI windows that all show some aspect of a data set:

- ![Diagram showing three GUI windows connected to a data object](image-url)
Another pattern: Observers

- When the data changes, each window also needs to change. What solutions address this problem?
Another pattern: Observers

- Simple. Repeatedly call each GUI window to allow it to update its data.
Another pattern: Observers

- Bad. Repeatedly call each GUI window to allow it to update its data.
Another pattern: Observers

- You’ve seen this pattern – the event in C# or listeners in Java.
Another pattern: Observers

- Create a list of objects that should be notified when the data object changes. (We will discuss forms of this.)

Some statistical data object
One of Peter’s favorite ‘patterns’

● Generalization in object-oriented programming
  – Not officially a pattern, more like a practice
  – Looks like a pattern, smells like a pattern
  – Fundamentally implied by object-oriented programming
‘Generalization’

- Consider the transmission of events in the last assignment:
  - Two types of events were sent, keyboard and mouse
  - Many stages of processing needed to occur
‘Generalization’

- How do/did you write the communication layer?
  - Do you write code to handle both KeyEvent and MouseEvent objects?
  - How do you avoid repetition?
‘Generalization’

- Choice 1 – Generalize through inheritance
  - Pros and cons discussion

```
Event

KeyEvent

MouseEvent
```
‘Generalization’

- Choice 2 – Generalize through interfaces
  - Pros and cons

<table>
<thead>
<tr>
<th>IEvent</th>
</tr>
</thead>
<tbody>
<tr>
<td>+someOp()</td>
</tr>
<tr>
<td>+otherOp()</td>
</tr>
</tbody>
</table>
Coming up

* More on Thursday
  - Common patterns
  - Some important antipatterns