Team Meeting
Design Patterns Continued
Team Meeting

• Please spend a few minutes talking with your teammates about your project.
  – What progress have you made?
  – What will you be doing next?
  – What is holding you up?

• I will be available for questions
  – Class will resume at 2:15
**OO Class Patterns**

- Most patterns consider several aspects of classes:
  - Classes as entities
    - definition of object data (values)
    - definition of interfaces to the data (functionality)
    - method calls as messages between entities
    - definitions of behavior
  - Classes as types (classification of objects)
    - allows type relations
    - classifications / generalizations
    - inheritance and polymorphism
OO Class Patterns

- Consider object ‘behavior’:
  - Language syntax can define methods and variables, but additional information is needed to communicate behavior
    - Methods define type and value behavior through parameters and return values and functional behavior through code
    - Instance variables define value existence, modifiability, and visibility, but little else
  - Contracts and specifications (as simple text) are commonly used for more abstract descriptions of behavior
    - Representation invariants
    - Preconditions
    - Postconditions
Descriptions of Behavior

- **Representation Invariant** (also called “class invariant”)
  - A constraint on the state of an object
  - Defines ‘well-formed’ instances of an object
  - Code inside of the class can depend on (and must ensure the validity of) these invariants

Inside some class

```csharp
// Invariant: fs != null
private FileStream fs;

// Invariant: size > 0
private int size;
```
Descriptions of Behavior

● Precondition
  – Specifies conditions (object and parameter restrictions) that must be true when a method begins execution.
  – If preconditions are met, postconditions are guaranteed.
  – A well-formed object instance is an implied precondition.

● Postcondition
  – Specifies outcomes of the method call
  – Includes all changes of object state, return values, and side effects
  – It is implied that the object instance will remain well-formed.
**OO Class Patterns**

- How does inheritance restrict functional behavior?
  - Consider this simple example – what type of object can legally be returned from ‘dolt’ in class Gamma? How about in Beta?

```
<table>
<thead>
<tr>
<th>Alpha</th>
<th>Gamma</th>
<th>Beta</th>
</tr>
</thead>
<tbody>
<tr>
<td>+doIt():Gamma</td>
<td>+doIt()</td>
<td>+doIt()</td>
</tr>
</tbody>
</table>
```


Covariance

● Method ‘doIt’ in class Gamma can be defined as below.
  – This is covariance – the return type of a virtual method can be more specific when it is overridden.

```
Alpha
+doIt():Gamma

Gamma
+doIt():Beta

Beta
+doIt()
```

● A similar term is contravariance
  – Parameters in an overridden method can be more general than the parameters for the same method in the base class.
Liskov Substitution Principle

- The Liskov substitution principle
  - Subclasses should be usable through the base class interface without the user knowing they are using a subclass.
  - This is behavioral subtyping, and is not processed or enforced in C# or Java
Liskov Substitution Principle

How does inheritance restrict behavior?

- Subclasses must preserve the invariants introduced in the superclass.
  » Keeping superclass variables private helps ensure this.

- Subclasses must adhere to the preconditions and postconditions when it overrides a base class method
  » Have you noticed that you cannot make an overridden method private if the base class method is public?
  » Unfortunately, this is all that is done to ensure it. The onus rests on the programmer.
Covariance in Behaviors

- Note that covariance also applies to behaviors
  - The preconditions on a overridden method can be more general than those in the base class, *but not more restrictive.*
  - The postconditions for an overridden method can be more specific than those in the base class, *but not less specific.*
  - Derived classes cannot impose any new representation invariants on the base class, but it can add some of its own.
Covariance

- Question: Does covariance apply to generic data types?
- No – classroom discussion follows.
The Strategy Pattern

What if you want to create a class to represent a list? Easy enough:

class List<T>
{
    public List() ...
    public void add(T e) ...
    public void insert(T e, int pos) ...
    public T get(int pos) ...
}
The Strategy Pattern

- What is the cost of inserting an item at the beginning of the list?
  - It depends entirely on the data structure you choose (your strategy)

```java
class List<T>
{
    public List() ...
    public void add(T e) ...
    public void insert(T e, int pos) ...
    public T get(int pos) ...
}
```
The Strategy Pattern

- The strategy pattern decouples algorithms (solution strategies) from the problem they are trying to solve:
The Strategy Pattern

- Users of the list decide upon a strategy when they need to create a list, but then proceed to use the list abstractly.
The Strategy Pattern

● Pros:
  – Algorithms are decoupled from the objects that require them
  – New strategies can be added without compromising the abstract view of an object

● Cons
  – Programmers need to know details about the internals of an object to make an informed decision
  – Others?
The Call Super Antipattern

- When overriding methods, subclasses should not be *required* to call superclasses to implement some functionality.
The Call Super Antipattern

- It is common to use ‘super’ calls for the subclass to access superclass methods. This is not the antipattern – it is perfectly normal (if optional).

```
public void draw()
{
    // Assume this clears the window
    super.draw();
    // Do more stuff using an inherited
    // sprite batch
}
```
The Call Super Antipattern

- The pattern is when it is *required* for the subclass to access the superclass to complete the work.

**Inside Alpha**

```java
public void draw() {
    // Access private sprite batch
    // Do the drawing
}
```

**Inside Gamma**

```java
public void draw() {
    // Setup specific drawing
    // Call to do the drawing
    super.draw();
}
```
The Call Super Antipattern

- You don’t necessarily want to make the sprite batch variable accessible to the subclass (for encapsulation), so how do you solve this?

```
Inside Alpha

public void draw()
{
    // Access private sprite batch
    // Do the drawing
}

Inside Gamma

public void draw()
{
    // Setup specific drawing
    // Call to do the drawing
    super.draw();
}
```
The Call Super Antipattern

- Create a second method to do the specific drawing and reorder the workflow:
  - When `draw` is called, it is now always activated in the base class.
The Call Super Antipattern

Create a second method to do the specific drawing:

- Since calling the base class was desired, the initial call accesses the base class. The base class then allows the subclass to override some part of the drawing process through the ‘subdraw’ method.

```java
Inside Alpha

public void draw()
{
    // Access private sprite batch
    // Do local drawing
    // Do subclass drawing
    subdraw(batch);
}

Inside Gamma

public void subdraw(batch)
{
    // Do specific drawing
}
```
Another pair of my favorite ideas

- Don’t Repeat Yourself (DRY)*
  - The key idea is that duplication of any information, abstractions, or code is undesirable in software development

- The theory is that...
  - Duplication means that an effort has been redundant
  - Duplication leads to maintenance troubles – a change or correction may not be reflected in all logically related components
  - Imagine writing (or cutting and pasting) three nearly identical sprite classes to handle different sprites in your game – how do you debug and revise them?

* From “The Pragmatic Programmer” by Hunt and Thomas.
Don’t Repeat Yourself in practice

- So, how do you avoid repetitious code?
  - Don’t use cut-and-paste?
  - OO techniques come in to play – let’s follow the patterns we’ve discussed with an example

- Assume your three sprite classes all draw a sprite animation frame from a sprite sheet. How can you avoid repetition of code or representations?
Don’t Repeat Yourself in practice

● First thoughts of most programmers:
  – Move the common functionality into a superclass

● It is generally a bad idea to allow details to migrate up into the abstraction
  – The superclass should be a more abstract view of the solution, not dependent on derived class requirements
    » Refer to the “Dependency Inversion Principle”
  – In this case, drawing of sprites may require individualized solutions, so most subclasses would simply override the base class anyway
Don’t Repeat Yourself in practice

- A better approach – since each class uses a sprite sheet, create the notion of a sprite sheet
  - Create a sprite sheet abstraction, a class, to represent the sprite sheet
  - Add a sprite sheet variable to each sprite class

- Let’s discuss the difference:
  - A BlockSprite IS-A Sprite that can draw from a sprite sheet
  - A BlockSprite IS-A Sprite that HAS-A SpriteSheet object
Don’t Repeat Yourself in practice

- No rule would be complete without its exception:
  
  - Repetition, while avoided when writing code, can be very desirable when executing code in parallel
  
  - In parallel programming, inter-process communications often serve as a bottleneck.
    
    » It can be better to have processes repeat computations individually than to share a single computed result
Another pair of my favorite ideas

- This leads me to the second idea:
  “Separation of Concerns*”

  - In our example, the first solution integrates concerns, the second solution separates them.

  - The core idea is to avoid overlapping functionality within a single domain.
    » At the statement level, don’t interleave algorithms
    » At the class level, don’t interleave representations and/or functionality

* Dijkstra, 1974.
Separation of Concerns

- Consider this algorithm:

```plaintext
// Open a file for reading
// Open network channel for writing
// Build data structure for sorting data
// Get screen context
// While more data in file
    // Output data over the network
    // Draw the data to the screen
    // Insert data into correct position in structure
// Close connection
// Close file
// Flush screen output
```
Separation of Concerns

- Revised to separate concerns
  - This is something you should already be striving to do, but it is worth revisiting this idea from time to time.

```c
// Open file, read data into list, and close file
// Draw list to the screen
// Output list to network
// Sort list
```
Coming Up

• Tuesday
  – Another scored stand-up meeting
  – Continued look at software engineering patterns and practices

• You should be starting on your second sprint over the weekend.