CSE 219
COMPUTER SCIENCE III

CREATIONAL DESIGN PATTERNS

SLIDES COURTESY:

RICHARD MCKENNA, STONY BROOK UNIVERSITY.

Design Patterns

- Design Pattern
  - A description of a problem and its solution that you can apply to many similar programming situations

- Patterns:
  - facilitate reuse of good, tried-and-tested solutions
  - capture the structure and interaction between components
  - Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved

Why is this important?

- Using proven, effective design patterns can make you a better software *designer & coder*

- You will recognize commonly used patterns in others' code
  - Java API
  - Project team members
  - Ex-employees

In addition, different technologies have their own patterns:
Servlet patterns, GUI patterns, etc …

- And you'll learn when to apply them to your own code
  - experience reuse (as opposed to code reuse)
  - we want you thinking at the pattern level

Common Design Patterns

Creational	Structural	Behavioral
• Factory • Singleton • Builder • Prototype	• Decorator • Adapter • Facade • Flyweight • Component Architecture	• Strategy • Template • Observer • Command • Iterator • State

Textbook: Head First Design Patterns

Creational Design Pattern

- Creational design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator.
- This gives program more flexibility in deciding which objects need to be created for a given use case.

Common Design Patterns

Creational	Structural	Behavioral
• Factory	• Decorator	• Strategy
• Singleton	• Adapter	• Template
• Builder	• Facade	• Observer
• Prototype	• Flyweight	• Command
	• Component Architecture	• Iterator
		• State

Textbook: Head First Design Patterns

The Factory Pattern

- Factories make stuff
- Factory classes make objects
- Shouldn't constructors do that?
  - factory classes employ constructors
- What's the point?
  - prevent misuse/improper construction
  - hides construction
  - provide API convenience
  - one stop shop for getting an object of a family type

What objects do factories make?

- Typically objects of the same family
  - common ancestor
  - same apparent type
  - different actual type

- Factory Pattern in the Java API:
  - BorderFactory.createXXXBorder methods
    - return apparent type of Border
    - return actual types of BevelBorder, EtchedBorder, etc …
  - lots of factory classes in security packages

Border Example

```
…
JPanel panel = new JPanel();
Border border = BorderFactory.createEtchedBorder();
panel.setBorder(border);

JPanel panel2 = new JPanel();
Border border2 = BorderFactory.createTitledBorder("Title");
panel2.setBorder(border2);
…
```

How could we implement a Factory Pattern?

```
public class BorderFactory
{ public static Border createEtchedBorder()
  {  return new EtchedBorder();
  }

  public static Border createTitledBorder(String title)
  {  return new TitledBorder(title);
  }
  …
}

public class EtchedBorder implements Border
{ // Border Methods }

public class TitledBorder implements Border
{ // Border Methods }
```

Factory Pattern Bonus

- The programmer using the Factory class never needs to know about the actual class/type
  - simplifies use for programmer
  - fewer classes to learn

- Ex: Using BorderFactory, one only needs to know Border & BorderFactory
  - not TitledBorder, BeveledBorder, EtchedBorder, etc.

- https://www.tutorialspoint.com/design_pattern/factory_pattern.htm

Common Design Patterns

Creational	Structural	Behavioral
• Factory	• Decorator	• Strategy
• Singleton	• Adapter	• Template
• Builder	• Facade	• Observer
• Prototype	• Flyweight	• Command
	• Component Architecture	• Iterator
		• State

Textbook: Head First Design Patterns

The Singleton Pattern

- Define a type where only one object of that type may be constructed:
  - make the constructor private.
  - singleton object favorable to fully static class, why?
    - can be used as a method argument
    - class can be extended
- What makes a good singleton candidate?
  - central app organizer class
  - something everybody needs

Example: The PropertiesManager Singleton

```
public class PropertiesManager
{
  private static PropertiesManager singleton
                                          = null;

  private PropertiesManager() {}

  public static PropertiesManager
                    getPropertiesManager()
  {
     if (singleton == null)
     {  singleton = new PropertiesManager();
     }
     return singleton;
  }
…
}
```

What's so great about a singleton?

- Other classes may now easily USE the **`PropertiesManager`**

```
PropertiesManager singleton =
    PropertiesManager.getPropertiesManager();
Singleton.dowhaterver()
```

- Don’t have to worry about passing objects around
- Don’t have to worry about object consistency
- **Note:** the singleton is of course only good for classes that will never need more than one instance in an application

- https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm

Common Design Patterns

Creational	Structural	Behavioral
• Factory	• Decorator	• Strategy
• Singleton	• Adapter	• Template
• Builder	• Facade	• Observer
• Prototype	• Flyweight	• Command
	• Component Architecture	• Iterator
		• State

Textbook: Head First Design Patterns

The Builder Pattern

- Use the Builder Pattern to:
  - encapsulate the construction of a product
  - allow it to be constructed in steps
- Good for complex object construction
  - objects that require lots of custom initialized pieces
- **Scenario**:
  - build a JavaFX component
  - put it in its container
  - register it by id to be retrieved later
  - add a style class
  - etc.

The Builder Pattern

```
public class AppNodesBuilder {
   public CheckBox buildCheckBox(
   public ColorPicker buildColorPicker(
   public ComboBox buildComboBox(
   public HBox buildHBox(
   public Label buildLabel(
   public Slider buildSlider(
   public VBox buildVBox(
   public Button buildIconButton(
   public Button buildTextButton(
   public ToggleButton buildIconToggleButton(
   public ToggleButton buildTextToggleButton(
   public TextField buildTextField(
   public TableView buildTableView(
   public TableColumn buildTableColumn(
   …
}
```

Using a builder

```
// INIT CONTROLS
HBox nameOwnerPane = tdlBuilder.buildHBox(…
HBox namePane = tdlBuilder.buildHBox(…
Label nameLabel = tdlBuilder.buildLabel(…
TextField nameTextField = tdlBuilder.buildTextField(…
HBox ownerPane = tdlBuilder.buildHBox(…
Label ownerLabel = tdlBuilder.buildLabel(…
TextField ownerTextField = tdlBuilder.buildTextField(…
```

Builder Benefits

- Encapsulates the way a complex object is constructed.
- Allows objects to be constructed in a multistep and varying process (as opposed to one step factories).
- Hides the internal representation of the product from the client.
- Product implementations can be swapped in and out because the client only sees an abstract interface.
- https://www.tutorialspoint.com/design_pattern/builder_pattern.htm

Common Design Patterns

Creational	Structural	Behavioral
• Factory	• Decorator	• Strategy
• Singleton	• Adapter	• Template
• Builder	• Facade	• Observer
• Prototype	• Flyweight	• Command
	• Component Architecture	• Iterator
		• State

Textbook: Head First Design Patterns

The Prototype Pattern

- Use the Prototype Pattern when creating an instance of a given class is either expensive or complicated.
- This pattern involves implementing a prototype interface which tells to create a clone of the current object.
- This pattern is used when creation of object directly is costly.
- For example, an object is to be created after a costly database operation.
- We can cache the object, returns its clone on next request and update the database as and when needed thus reducing database calls.

So what does the Prototype Pattern do?

- Allows you to make new instances by copying existing instances
  - in Java this typically means using the clone() method, or de-serialization when you need deep copies

- A key aspect of this pattern is that the client code can make new instances without knowing which specific class is being instantiated

Prototype Benefits

- Hides the complexities of making new instances from the client.
- Provides the option for the client to generate objects whose type is not known.
- In some circumstances, copying an object can be more efficient than creating a new object.

Prototype Uses and Drawbacks

- Prototype should be considered when a system must create new objects of many types in a complex class hierarchy.
- A drawback to using the Prototype is that making a copy of an object can sometimes be complicated.
- https://www.tutorialspoint.com/design_pattern/prototype_pattern.htm

A Prototype Pattern Example