Object-Oriented Design

Lecturer: Raman Ramsin

Lecture 20:

GoF Design Patterns – Creational
Software Patterns

- Software Patterns support reuse of software architecture and design.

  - Patterns capture the static and dynamic structures and collaborations of successful solutions to problems that arise when building applications in a particular domain.

- Patterns represent solutions to problems that arise when developing software within a particular context.

  - i.e., “Pattern == problem/solution pair in a context”
GoF Design Patterns – Principles

- Emphasis on flexibility and reuse through decoupling of classes.

- The underlying principles:
  - program to an interface, not to an implementation.
  - favor composition over class inheritance.
  - find what varies and encapsulate it.
GoF Design Patterns: General Categories

- 23 patterns are divided into three separate categories:
  - **Creational** patterns
    - Deal with initializing and configuring classes and objects.
  - **Structural** patterns
    - Deal with decoupling interface and implementation of classes and objects.
  - **Behavioral** patterns
    - Deal with dynamic interactions among societies of classes and objects.
GoF Design Patterns: Purpose and Scope

<table>
<thead>
<tr>
<th>Class</th>
<th>Purpose</th>
<th>Structural</th>
<th>Behavioral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factory Method</td>
<td>Adapter (class)</td>
<td>Interpreter</td>
<td></td>
</tr>
<tr>
<td>Abstract Factory</td>
<td>Adapter (object)</td>
<td>Chain of Responsibility</td>
<td>Command</td>
</tr>
<tr>
<td>Builder</td>
<td>Bridge</td>
<td>Command</td>
<td></td>
</tr>
<tr>
<td>Prototype</td>
<td>Composite</td>
<td>Iterator</td>
<td></td>
</tr>
<tr>
<td>Singleton</td>
<td>Decorator</td>
<td>Mediator</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Facade</td>
<td>Memento</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Flyweight</td>
<td>Observer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Proxy</td>
<td>State</td>
<td>Strategy</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Visitor</td>
</tr>
</tbody>
</table>
GoF Creational Patterns

- **Class**
  - **Factory Method**: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.

- **Object**
  - **Abstract Factory**: Provide an interface for creating families of related or dependent objects without specifying their concrete class.
  - **Builder**: Separate the construction of a complex object from its representation so that the same construction process can create different representations.
  - **Prototype**: Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
  - **Singleton**: Ensure a class only has one instance, and provide a global point of access to it.
Factory Method

- **Intent:**
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

![Diagram](image.png)
Factory Method: Applicability

- Use the Factory Method pattern when
  - a class can't anticipate the class of objects it must create.
  - a class wants its subclasses to specify the objects it creates.
  - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.
Factory Method: Structure

```
Product

ConcreteProduct

Creator

FactoryMethod()
AnOperation()

ConcreteCreator

FactoryMethod()

... product = FactoryMethod()
...

return new ConcreteProduct
```
Abstract Factory

**Intent:**

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
Abstract Factory: Applicability

Use the Abstract Factory pattern when

- a system should be independent of how its products are created, composed, and represented.
- a system should be configured with one of multiple families of products.
- a family of related product objects is designed to be used together, and you need to enforce this constraint.
- you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.
Abstract Factory: Structure
Builder

- **Intent:**
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations.
Builder: Applicability

- Use the Builder pattern when

  - the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.

  - the construction process must allow different representations for the object that's constructed.
Builder: Structure

Director

Construct()

for all objects in structure {
    builder->BuildPart()
}

Builder

BuildPart()

ConcreteBuilder

BuildPart()
GetResult()

Product

Department of Computer Engineering
Builder: Collaborations
Prototype

- **Intent:**
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

```
Tool
  Manipulate()

RotateTool
  Manipulate()

GraphicTool
  Manipulate()

p = prototype->Clone()
while (user drags mouse) {
  p->Draw(new position)
} insert p into drawing

Graphic
  Draw(Position)
  Clone()

Staff
  Draw(Position)
  Clone()

MusicalNote
  WholeNote
    Draw(Position)
    Clone()

HalfNote
    Draw(Position)
    Clone()
```
Prototype: Applicability

- Use the Prototype pattern when

  - the classes to instantiate are specified at run-time, for example, by dynamic loading.

  - building a class hierarchy of factories that parallels the class hierarchy of products should be avoided.

  - instances of a class can have one of only a few different combinations of state.
    - It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually.
Prototype: Structure

Client
    Operation()

Prototype
    Clone()

p = prototype->Clone()

ConcretePrototype1
    Clone()
    return copy of self

ConcretePrototype2
    Clone()
    return copy of self
Singleton

**Intent:**
- Ensure a class only has one instance, and provide a global point of access to it.
Singleton: Applicability

- Use the Singleton pattern when
  - there must be exactly one instance of a class, and it must be accessible to clients from a well known access point.
  - when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.
Reference