Patterns in Software Engineering

Lecturer: Raman Ramsin

Lecture 2

GoF Design Patterns – Creational
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>
</tr>
</thead>
<tbody>
<tr>
<td>Factory Method</td>
<td>Adapter (class)</td>
</tr>
<tr>
<td>Builder</td>
<td>Bridge</td>
</tr>
<tr>
<td>Prototype</td>
<td>Composite</td>
</tr>
<tr>
<td>Singleton</td>
<td>Decorator</td>
</tr>
<tr>
<td>Scopes</td>
<td>Purpose</td>
</tr>
<tr>
<td></td>
<td>Creational</td>
</tr>
<tr>
<td></td>
<td>Structural</td>
</tr>
<tr>
<td></td>
<td>Behavioral</td>
</tr>
</tbody>
</table>

- **Creational**:
  - Factory Method
  - Builder
  - Prototype
  - Singleton

- **Structural**:
  - Adapter (class)
  - Bridge
  - Composite
  - Decorator

- **Behavioral**:
  - Interpreter
  - Chain of Responsibility
  - Command
  - Iterator
  - Mediator
  - Facade
  - Memento
  - Observer
  - State
  - Strategy
  - Visitor
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.
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

Diagram:

```
Product
    ▲
  Creator
    ▲
ConcreteCreator
    ▲
ConcreteProduct
```

Code:

```
... product = FactoryMethod()
...
return new ConcreteProduct
```
Factory Method: Consequences

- It provides hooks for the subclasses.
- It connects parallel class hierarchies.
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
Abstract Factory: Consequences

✓ **Concrete classes are isolated.** Clients manipulate instances through their abstract interfaces.

✓ **Exchanging product families is easy.** Different product configurations can be used simply by changing the concrete factory.

✓ **Consistency among products is promoted.**

✗ **Supporting new kinds of products is difficult.** The AbstractFactory interface fixes the set of products that can be created.
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

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

Diagram:

- Director
  - Construct()

- Builder
  - BuildPart()

- ConcreteBuilder
  - BuildPart()
  - GetResult()

- Product
Builder: Collaborations
Builder: Consequences

✓ *It lets you vary a product's internal representation.*

✓ *It isolates code for construction and representation.*

✓ *It gives you finer control over the construction process:* Since the Builder pattern constructs the product step by step under the director's control.
Prototype

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

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

```
WholeNote
   Draw(Position) Clone()
   return copy of self
```

```
HalfNote
   Draw(Position) Clone()
   return copy of self
```
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
```
Prototype: Consequences

✓ *It hides the concrete product classes from the clients, thereby reducing the number of names clients know about.*

✓ *It lets a client work with application-specific classes without modification.*

✓ *It lets you add and remove products at run-time.*

✓ *It lets you specify new objects by varying values.*
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.
Singleton: Consequences

✓ It provides Controlled access to sole instance.

✓ It reduces the name space by avoiding global variables.

✓ It permits refinement of operations and representation through subclassing.

✓ It permits a variable number of instances.

✓ It is more flexible than class operations.
Reference

Gamma, E., Helm, R., Johnson, R., and Vlissides, J., Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley, 1995.