Problem: Window Appearance

Independent Features

Solution 1: Inheritance
Solution 2: Composition and Interfaces

Composition and the programming to an interface design idiom lets us change window properties dynamically.

Decorator Pattern: Basic Idea

Decorator (wrapper object) intercepts call to nested object.
- can add functionality before it invokes component's op
- can add functionality after it invokes component's op
- can add functionality before and after
- can decide NOT to call component's op
- can add new operations!

Decorator Pattern

Idea: Using composition, build a wrapper.
A wrapper implements new functionality, and includes original object as a component.

Design Pattern Instantiated
Using Decorators

Decorated objects are created by instantiating features and passing them the component object.

```cpp
// client code
int main () {
    VisualComponent* list = new FileListing();
    VisualComponent* bList = new Border(list);
    VisualComponent* bsList = new HScroll(new VScroll(bList));
    ...
    bsList->draw();
}
```

Summary

**Decorator Pattern**: encapsulates "features" or additional responsibilities or functionality that can be added to a class at runtime.

Design Patterns

**OO Basics**
- Separation of Concerns
- Loose Coupling
- Information Hiding
- Avoid duplicate code

**OO Principles**
- Encapsulate what is likely to change
- Single Responsibility Principle
- Law of Demeter
- Favour Composition over Inheritance
- Program to an Interface, not an Implementation
- Liskov Substitutability Principle

**Strategy**
- Encapsulate polymorphic decision-making

**Template Method**
- Define the skeleton of an algorithm in a separate class, defer some steps to subclasses

**Observer**
- Implement a one-to-many dependency relationship between objects

**Facade**
- Encapsulate a complex subsystem

**MVC**
- Model-View-Controller

**Composite**
- Represent a part of a tree structure via nested objects

**Iterator**
- Allow a client to traverse through a collection without exposing its underlying representation

**Decorator**
- Add responsibilities to an object dynamically

**Factory Method**
- Define an interface for creating an object, but let subclasses decide which class to instantiate
Instantiating Objects (of Concrete Type)

Program to an Interface, not Implementation is hard to follow when creating new objects—cannot instantiate abstract objects.

```cpp
void admitStudent (const string &name, const string &faculty){
    Student *s;

    // must instantiate concrete objects
    if (faculty == "Engineering") {
        s = new EngineeringStudent(name);
    } else if (faculty == "Math") {
        s = new MathStudent(name);
    } else if (faculty == "Science") {
        s = new ScienceStudent(name);
    } ...

    // Each student type has its own admission operations
    s->welcome();
    s->invoiceTuition();
    s->createTranscript();
}
```

Approach 1: Encapsulation

Encapsulate code that creates concrete objects in a Simple Factory. (Not a design pattern.)

Approach 2: Factory Method Pattern

Problem: encapsulate the code that creates concrete objects.
- Factories are polymorphic.

Solution: use the Template Method.
- Abstract class defines a method (template method).
- Factory method is a primitive operation of the template method.
- Subclasses override factory method to construct specific concrete objects.

![Diagram of Factory Method (Instantiated)]
Factory Method (Instantiated) 2

```
class ShapeFactory {
    virtual Shape* create() = 0; // Factory Method
    static std::map<std::string, ShapeFactory*> factories;
}
```
Factory Method Pattern (Instantiated)
From Xinhao Tian, SE_2014

Abstract Factory Design Pattern

Abstract Factory Example

Summary of Design Patterns

Design Patterns are about encapsulating what changes:
- code is more maintainable (changes are localized)
- client code is likely to have to change as well
- some patterns enable design decisions to be changed at runtime

XP rule of three applies to design patterns
- The first time you code a given task, just do it.
- The second time you code the same idea, wince and code it up again.
- The third time you code the same idea, it's time to refactor!
What You Should Get From This

Recognition
  • Know what problem a pattern solves.
  • Know a pattern’s UML model.

Comprehension
  • Select appropriate design pattern to solve particular design problem.

Application
  • Apply design patterns to modify given UML model.
  • Implement (code fragments) of design patterns.
    - Singleton
    - Template Method
    - Strategy
    - Adapter
    - Observer
    - MVC
    - Composite
    - Iterator
    - Decorator
    - Factory Method
    - Abstract Factory