CS 247: Software Engineering Principles

Design Patterns (Decorator, Factory)

Reading: Freeman, Robson, Bates, Sierra, *Head First Design Patterns*, O'Reilly Media, Inc. 2004
  - Ch 3 Decorator Pattern
  - Ch 4 Factory Method Pattern, Abstract Factory Pattern
Problem: Window Appearance

- **Quit**
- **Border**
- **Content**
- **Navigation**
- **Metadata**
- **Vertical Scrollbar**
- **Horizontal Scrollbar**
- **Maximize/Minimize**
- **Hide/Display**
Independent Features

- horizontal scroll
- vertical scroll
- content
- border
Solution 1: Inheritance

FileListing

draw ()

BorderFileListing
draw ()

HScrollFileListing
draw ()

VScrollFileListing
draw ()

HVScrollFileListing
draw ()

BorderNavFileListing
draw ()

HScrollNavFileListing
draw ()

VScrollNavFileListing
draw ()

HVScrollNavFileListing
draw ()

BorderMinFileListing
draw ()

HScrollMinFileListing
draw ()

VScrollMinFileListing
draw ()

HVScrollMinFileListing
draw ()

BorderHideFileListing
draw ()

HScrollHideFileListing
draw ()

VScrollHideFileListing
draw ()

HVScrollHideFileListing
draw ()

Aaaarrrgggh!
Solution 2: Composition and Interfaces

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

```plaintext
FileListing
  draw()

* FileEntry
    draw()

Decorator
  draw()
  decorators

Border

NoBorder
  draw()

PlainBorder
  draw()

FancyBorder
  draw()

HScrollBar
  draw()

VScrollBar
  draw()
```
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

- **VisualComponent**
  - +draw ()
  - +scroll() { }

- **FileListing**
  - +draw ()

- **FileEntry**
  - draw ()

- **File**

- **Decorators**
  - **Decorator**
    - +draw ()
    - +scroll ()
  - **HScroll**
    - -scrollPosition
    - +draw ()
    - +scroll ()
    - -drawScrollBar ()
    - -scrollTo ()
  - **VScroll**
    - -scrollPosition
    - +draw ()
    - +scroll ()
    - -drawScrollBar ()
    - -scrollTo ()
  - **Border**
    - -style
    - +draw ()
    - -drawBorder ()
Using Decorators

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

```c++
// client code
int main () {
    VisualComponent* list = new FileListing();

    VisualComponent* bList = new Border(list);

    VisualComponent* bsList = new HScroll(new VScroll(bList));
    ...
    bsList->draw();
}
```
Decorator Pattern: encapsulates "features" or additional responsibilities or functionality that can be added to a class at runtime.
Design Patterns

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

Separation of Concerns
- Loose Coupling
- Information Hiding
- Avoid duplicate code

Design Patterns
- Strategy
- Template Method
- Adaptor
- Facade
- Observer
- MVC
- Composite
- Iterator
- Decorator
- Factory Method
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();
}

Program to an Interface, not Implementation is hard to follow when creating new objects—cannot instantiate abstract objects.
Approach 1: Encapsulation

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

// friend of MathStudent
// friend of EngineeringStudent

// must instantiate concrete objects
if (faculty == "Engineering")
    return new EngineeringStudent(name);
if (faculty == "Math")
    return new MathStudent(name);
if (faculty == "Science")
    return new ScienceStudent(name);
...
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.
Factory Method (Instantiated)

```
create() : Shape
+ createShape (string) : Shape

ShapeFactory

Shape
+ draw()
+ erase()

Square
+ draw()
+ erase()

Circle
+ draw()
+ erase()

Circle::CircleFactory
- create() : Shape

Square::SquareFactory
- create() : Shape

Client Code
```

- factory method
- template method, calls factory method

Template method, calls factory method

return new ConcreteProduct;

U Waterloo CS247 (Spring 2017) — p.16/25
createStudent(name)
admitStudent (name)

Registrar
createStudent(name)

Math Registrar
createStudent(name)

Engineering Registrar
createStudent(name)

client

Math Student
<<creates>>

Engineering Student
<<creates>>

Student

factory method

template method

return new ConcreteProduct;
Polymorphic Factory
From "Thinking in C++, Volume 2", by Bruce Eckel & Chuck Allison.

class ShapeFactory {
    virtual Shape* create() = 0; // Factory Method
    static std::map<std::string, ShapeFactory*> factories_;

public:
    virtual ~ShapeFactory() {} // Template Method
    friend class ShapeFactoryInitializer;

    static Shape* createShape( const std::string& id ) {
        if ( factories_.find(id) != factories_.end() )
            return factories_[id]->create();
    }
};

std::map<std::string, ShapeFactory*> ShapeFactory::factories_;
Concrete Classes

From "Thinking in C++, Volume 2", by Bruce Eckel & Chuck Allison.

class Circle : public Shape {
    Circle() {} // Private constructor
    friend class ShapeFactoryInitializer;

class Factory;
friend class Factory;

class Factory : public ShapeFactory {
    public:
        Shape* create() { return new Circle; }
        friend class ShapeFactoryInitializer;
    };

public:
    void draw() { cout << "Circle::draw" << endl; }
    void erase() { cout << "Circle::erase" << endl; }
    ~Circle() { cout << "Circle::~Circle" << endl; }
};
// Singleton to initialize the ShapeFactory:
class ShapeFactoryInitializer {
    static ShapeFactoryInitializer si;
    ShapeFactoryInitializer() {
        ShapeFactory::factories_['Circle'] = new Circle::Factory;
        ShapeFactory::factories_['Square'] = new Square::Factory;
    }

    // Destructor deletes all factories
    ~ShapeFactoryInitializer() {
        for ( auto it : ShapeFactory::factories_ )
            delete it->second;
    }
};

// Static member definition:
ShapeFactoryInitializer ShapeFactoryInitializer::si;
Factory Method Pattern (Instantiated)

From Xinhao Tian, SE_2014

CharacterFactory
- factories
- create() : Character
+ createCharacter(type: string) : Character

HeavyFactory
- create() : Character

SoldierFactory
- create() : Character

MedicFactory
- create() : Character

Character

Heavy
<<creates>>

Soldier
<<nested Class>>

Medic
<<nested Class>>

client
<<creates>>

client

U Waterloo CS247 (Spring 2017) — p.21/25
Abstract Factory Design Pattern

AbstractFactory
+createProductA() : AbstractProductA
+createProductB() : AbstractProductB

ConcreteFactory1
+createProductA()
+createProductB()

ConcreteFactory2
+createProductA()
+createProductB()

client

AbstractProductA

ProductA1

ProductA2

AbstractProductB

ProductB1

ProductB2
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