Design Patterns

Catia Trubiani, Antinisca Di Marco
{catia.trubiani, antinisca.dimarco}@univaq.it

Roadmap

- Background: what's a design pattern?
- Elements of Design patterns:
  - Name, Problem, Solution, Consequences
- Catalog of design patterns:
  - Creational Patterns
  - Structural Patterns
  - Behavioral Patterns
- Conclusion and References

Background

- Design patterns represent solutions to problems that arise when developing software, in fact they refer to problem/solution pairs within a particular context
- Describes recurring design structures by facilitating reuse of successful software architectures and designs
- Patterns capture the static and dynamic *structure* and *collaboration* among key participants in software designs

Origin of Design Patterns

"Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it in the same way twice"

Christopher Alexander, A Pattern Language, 1977,
Context: City Planning and Building architectures

History of Design Patterns

- 1987: Cunningham and Beck use Alexander ideas to develop a pattern language for Smalltalk
- 1990: Gang of Four (Gamma, Helm, Johnson e Vlissides) start to create a catalog of design patterns
- 1991: Bruce Anderson shows the first design patterns at the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)
- 1993: Kent Beck and Grady Booch organize the Hillside Group, i.e. the first meeting for discussing design patterns challenges
- 1994: First Conference on Pattern Languages of Programs (PLoP)
- 1995: The Gang of Four publishes the book “Design Patterns”

Elements of Design Patterns

- Design patterns have four essential elements:
  - Pattern name
  - Problem
  - Solution
  - Consequences

Pattern Name

- A handle used to describe:
  - a design problem
  - its solutions
  - its consequences
- Increases design vocabulary
- Makes it possible to design at a higher level of abstraction
- Enhances communication

  "The Hardest part of programming is coming up with good variable [function and type] names"

Problem

- Describes when to apply the pattern
- Explains the problem and its context
- May describe specific design problems and/or object structures
- May contain a list of preconditions that must be met before it makes sense to apply the pattern

Solution

- Describes the elements that make up the
  - design
  - relationships
  - responsibilities
  - collaborations
- Does not describe specific concrete implementation
- Abstract description of design problems and how the pattern solves them

Consequences

- Results and trade-offs of applying the pattern
- Critical for:
  - evaluating design alternatives
  - understanding costs
  - understanding benefits of applying the pattern
- Includes the impacts of a pattern on a system's:
  - flexibility
  - extensibility
  - portability

Design Patterns are NOT

- Designs that can be encoded in classes and reused as is (like linked lists and hash tables)
- Complex domain-specific designs (for an entire application or subsystem)

- They are "*Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context*"

Describing Design Patterns

- Graphical notation is generally not sufficient
- In order to reuse design decisions the alternatives and trade-offs that led to the decisions are critical knowledge
- Concrete examples are also important
- The history of the why, when, and how set the pattern for the context of usage

Design Patterns summary

- Describe a recurring design structure
  - Defines a common vocabulary
  - Abstracts from concrete designs
  - Identifies classes, collaborations, and responsibilities
  - Describes applicability, trade-offs, and consequences

Categorization Terms

- Scope is the domain over which a pattern applies
  - *Class Scope*: relationships between base classes and their subclasses (static semantics)
  - *Object Scope*: relationships between objects
- Some patterns apply to both scopes.

Types of Patterns

- **Creational patterns:**
  - Deal with initializing and configuring classes and objects
- **Structural patterns**:
  - Deal with the structural implementation of classes and objects
  - Composition of classes or objects
- **Behavioral patterns**:
  - Deal with dynamic interactions among societies of classes and objects
  - How they distribute responsibility

Some examples of Design Patterns

	Creational	Structural	Behavioral
Class	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

Decorator Pattern

Decorator pattern description

- Intent
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Also known as “Wrapper”
- Motivation
  - Sometimes we want to add responsibilities to individual objects, not to an entire class. A graphical user interface toolkit, for example, should let you add properties like borders or behaviors like scrolling to any user interface component.

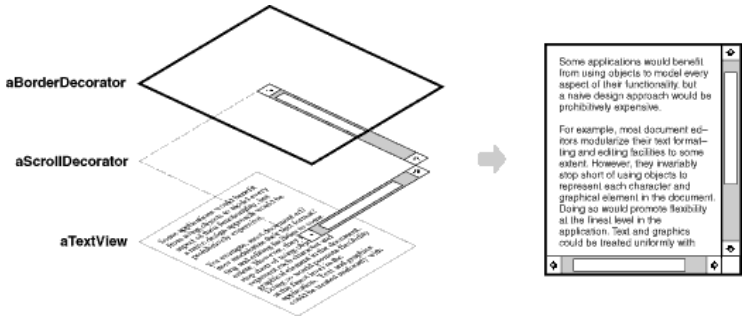
Scope: add responsabilities to individual objects

- One way to add responsibilities is with inheritance. Inheriting a border from another class puts a border around every subclass instance. This is inflexible, however, because the choice of border is made statically. A client can't control how and when to decorate the component with a border.

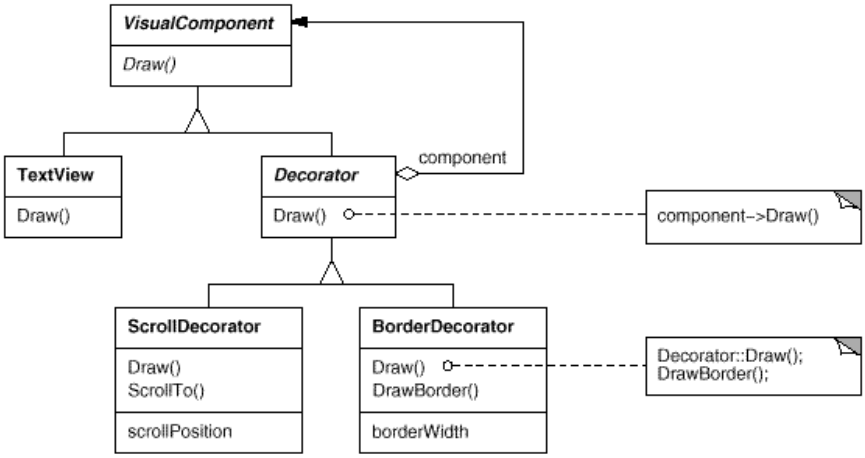
- A more flexible approach is to enclose the component in another object that adds the border. The enclosing object is called a **decorator.** The decorator conforms to the interface of the component it decorates so that its presence is transparent to the component's clients. The decorator forwards requests to the component and may perform additional actions (such as drawing a border) before or after forwarding.

Motivation for Decorator Pattern

- Suppose we have a TextView object that displays text in a window. TextView has no scroll bars by default, because we might not always need them. When we do, we can use a ScrollDecorator to add them. If we also want to add a thick black border around the TextView, we can use a BorderDecorator to add this as well.



Decorator Pattern structure



Decorator pattern: applicability

- Use the Decorator pattern for:
  - add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
  - responsibilities that can be withdrawn.
  - when extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class may be hidden or unavailable for subclassing.

Decorator Pattern: participants

- Component (VisualComponent)
  - defines the interface for objects that can have responsibilities added to them dynamically
- ConcreteComponent (TextView)
  - defines an object to which additional responsibilities can be attached
- Decorator
  - maintains a reference to a Component object and defines an interface that conforms to Component's interface
- ConcreteDecorator (BorderDecorator, ScrollDecorator)
  - adds responsibilities to the component

Decorator Pattern: consequences (1/2)

- *More flexibility than static inheritance.*
  - The Decorator pattern provides a more flexible way to add responsibilities to objects than can be had with static (multiple) inheritance. With decorators, responsibilities can be added and removed at run-time simply by attaching and detaching them. In contrast, inheritance requires creating a new class for each additional responsibility and this gives rise to many classes and increases the complexity of a system.
- *Avoids feature-laden classes high up in the hierarchy.*
  - Decorator offers a pay-as-you-go approach to adding responsibilities. Instead of trying to support all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects. Functionality can be composed from simple pieces. As a result, an application needn't pay for features it doesn't use.

Decorator Pattern: consequences (2/2)

- *A decorator and its component aren't identical.*
  - A decorated component is not identical to the component itself. Hence you shouldn't rely on object identity when you use decorators.
- *Lots of little objects.*
  - A design that uses Decorator often results in systems composed of lots of little objects that all look alike. The objects differ only in the way they are interconnected, not in their class or in the value of their variables.

Decorator Pattern: sample code (1/4)

```
class VisualComponent {
    public: VisualComponent();
    virtual void Draw();
    virtual void Resize();
    // ...
}
```

Decorator Pattern: sample code (2/4)

```
class Decorator : public VisualComponent {
    public: Decorator(VisualComponent*);
    virtual void Draw();
    virtual void Resize();
    // ...
    private: VisualComponent* _component;

    void Decorator::Draw () {
        _component->Draw();
    }
    void Decorator::Resize () {
        _component->Resize();
    }
}
```

Decorator Pattern: sample code (3/4)

```
class BorderDecorator : public Decorator {
    public: BorderDecorator(VisualComponent*, int borderWidth);

    virtual void Draw();

    private: void DrawBorder(int);
    private: int _width;

    void BorderDecorator::Draw () {
        Decorator::Draw();
        DrawBorder(_width);
    }
}
```

Decorator Pattern: sample code (4/4)

```
void Window::SetContents (VisualComponent* contents){
    // ...
}

Window* window = new Window;
TextView* textView = new TextView;

window->SetContents(textView);

window->SetContents(
    new BorderDecorator(
        new ScrollDecorator(textView), 1);
```

Conclusion

- What Makes it a Pattern? A Pattern must:
  - Solve a problem and be useful
  - Have a context and can describe where the solution can be used
  - Recur in relevant situations
  - Provide sufficient understanding to tailor the solution
  - Have a name and be referenced consistently

Benefits of Design Patterns

- Design patterns enable large-scale reuse of software architectures and also help document systems
- Patterns explicitly capture expert knowledge and design tradeoffs and make it more widely available
- Patterns help improve developer communication
- Pattern names form a common vocabulary
- Patterns help ease the transition to OO technology

Drawbacks to Design Patterns

- Patterns do not lead to direct code reuse
- Patterns are deceptively simple
- Teams may suffer from pattern overload
- Patterns are validated by experience and discussion rather than by automated testing
- Integrating patterns into a software development process is a human-intensive activity

Suggestions for Effective Use

- Do not consider everything as a pattern
  - Instead, develop strategic domain patterns and reuse existing tactical patterns
- Institutionalize rewards for developing patterns
- Directly involve pattern authors with application developers and domain experts
- Clearly document when patterns apply and do not apply
- Manage expectations carefully

Further readings and References

- Other Design Patterns:
  - A Creational Pattern: "Abstract Factory"
  - A Behavioral Pattern: "Observer"
- Main reference:
  - E.Gamma et al. "*Design Patterns: Elements of Reusable Object-Oriented Software*", Addison-Wesley, 1995.
- Other references:
  - Sommerville, "Software Engineering", section 7.2
  - Davide Di Ruscio, "Programmazione Java – Design Patterns", Tecnologie dei Linguaggi di Programmazione