Introduction to Design Patterns

Giuseppe Lipari
`http://retis.sssup.it`

Scuola Superiore Sant'Anna – Pisa

June 8, 2009

Motivation

- Good Object Oriented programming is not easy
  - Emphasis on design
- Errors may be expensive
  - Especially design errors!
- Need a lot of experience to improve the ability in OO design and programming
- Reuse experts' design
- Patterns $=$ documented experience

The source

- The design patterns idea was first proposed to the software community by the “Gang of four” [2]
  - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
  - Design patterns: elements of reusable object-oriented software
- They were inspired by a book on architecture design by Christopher Alexander [1]

  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 the same way twice.

Expected Benefits

- The idea of patterns has a general meaning and a general application: from architecture to software design!
  - One of the few examples in which software development has been inspired by other areas of engineering
- The expected benefits of applying well-know design structures
  - Finding the right code structure (which classes, their relationship)
  - Coded infrastructures!
  - A Common design jargon (factory, delegation, composite, etc.)
  - Consistent format

Object relationship

- Inheritance: Static and efficient, but exposes and couples modules
- Composition: Hides more from client and can change dynamically
- Gang of Four:

  You should favor composition over inheritance

- Dijkstra

  Most problems in computer science can be solved by adding another level of indirection

Designing for change

- Every software is subject to change
  - A good design makes changes less trouble
- Problems related to change:
  - The immediate cause of the degradation of the design is when requirements change in ways that the initial design did not anticipate
  - Often these changes need to be made quickly, and may be made by engineers who are not familiar with the original design philosophy
  - So, though the change to the design works, it somehow violates the original design. Bit by bit, as the changes continue to pour in, these violations accumulate until malignancy sets in
- The requirements document is the most volatile document in the project
  - If our designs are failing due to the constant rain of changing requirements, it is our designs that are at fault

The open/closed principle

- Bertrand Meyer said [3]:

  A class should be open for extension, but closed for modification

- In other words, (in an ideal world...) you should never need to change existing code or classes
  - except for bug-fixing and maintainance
- all new functionality can be added by adding new subclasses and overriding methods, or by reusing existing code through delegation

Design for change

- The Open-Closed principle
- Key issue: *prepare for change*
- Causes for re-design
  - Dependence on hardware or software platform
  - Dependence on representation or implementation
  - Algorithmic dependence
  - Tight coupling
  - Overuse of inheritance
  - Inability to alter classes easily

Categories

- **Creational:** Replace explicit creation problems, prevent platform dependencies
- **Structural:** Handle unchangeable classes, lower coupling and offer alternatives to inheritance
- **Behavioral:** Hide implementation, hides algorithms, allows easy and dynamic configuration of objects

Pattern of patterns

- Common approach in all patterns:
  - Encapsulate the varying aspect
  - Interfaces
  - Inheritance describes variants
  - Composition allows a dynamic choice between variants
- Criteria for success:
  - Open-Closed Principle
  - Single Choice Principle

Abstract factory

- A program must be able to choose one of several families of classes
- Example,
  - a program's GUI should run on several platforms
  - Each platform comes with its own set of GUI classes:
    - WinButton, WinScrollBar, WinWindow MotifButton, MotifScrollBar, MotifWindow pmButton, pmScrollBar, pmWindow
  - Inheritance:
    - Clearly, we can make all "button" classes derive from an abstract button that implements a virtual "draw" function
    - Then, we hold a pointer to button, and assign a specific button object, so that the correct draw() function is invoked each time
  - We probably need to dynamically create a lot of this objects
  - Problem: how can we simplify the creation of this objects?

Naive approach

- We keep a global variable (or object) that represents the current window manager and "look-and-feel" for all the objects
- Every time we create an object, we execute a switch/case on the global variable to see which object we must create

```
enum {WIN, MOTIF, PM, ...} lf;
...
// need to create a button
switch(lf) {
case WIN:   button = new WinButton(...);
            break;
case MOTIF: button = new MotifButton(...);
            break;
case PM:    button = new PmButton(...);
            ...
}
```

Problems with the naive approach

- What happens if we need to add a new look-and-feel?
  - We must change lot of code (for every creation, we must add a new case)
- How much code we must link?
  - Assuming that each look and feel is part of a different library, all libraries must be linked together
  - Large amount of code
- This solution is not compliant with the open/closed principle
  - Everytime we add a new look and feel we must change the code of existing functions/classes
- This solution *does not scale*

Requirements

- Uniform treatment of every button, window, etc.
  - Once you define the interface, you can easily use inheritance
- Uniform object creation
- Easy to switch between families
- Easy to add a family

Solution: Abstract factory

- Define a *factory* (i.e. a class whose sole responsibility is to create objects)

```
class WidgetFactory {
    Button* makeButton(args) = 0;
    Window* makeWindow(args) = 0;
    // other widgets...
};
```

- Define a concrete factory for each of the families

```
class WinWidgetFactory : public WidgetFactory {
    Button* makeButton(args) {
        return new WinButton(args);
    }
    Window* makeWindow(args) {
        return new WinWindow(args);
    }
};
```

Solution - cont.

- Select once which family to use:

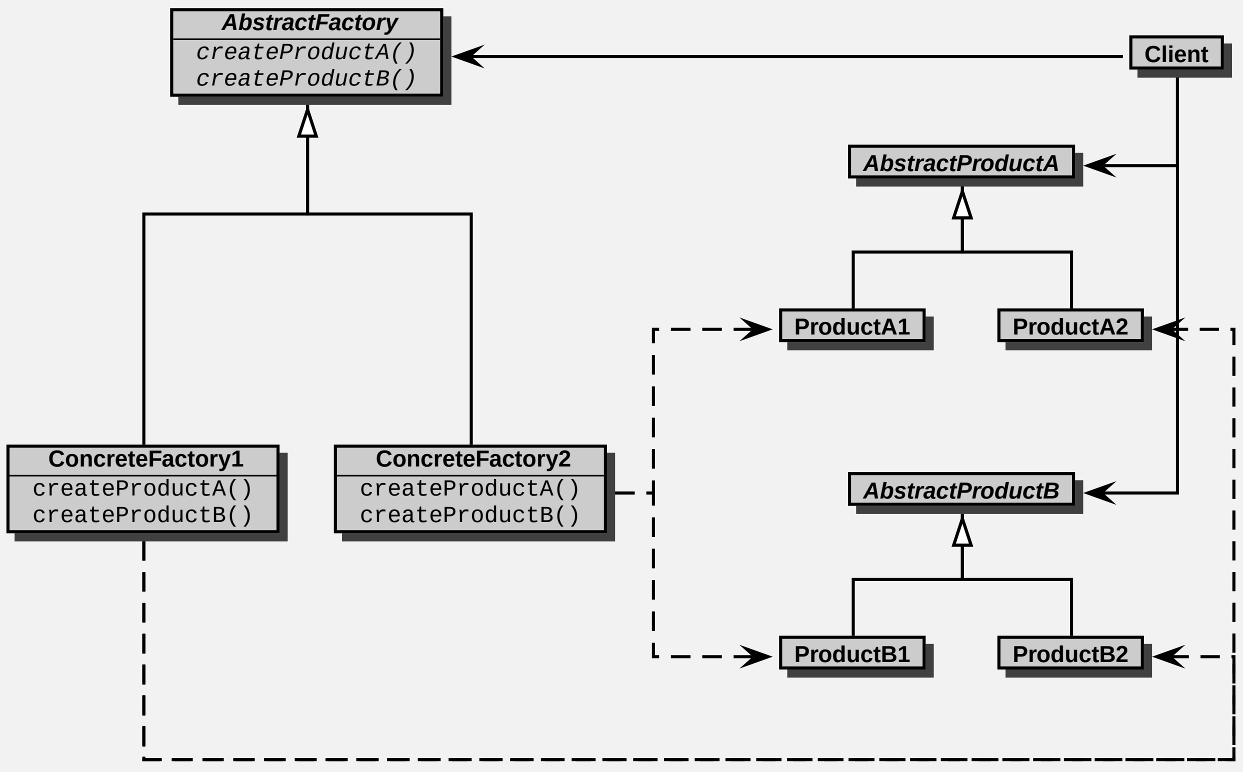
```
WidgetFactory* wf;
switch (lf) {
case WIN:    wf = new WinWidgetFactory();
             break;
case MOTIF:  wf = new MotifWidgetFactory();
             break;
...
}
```

- When creating objects in the code, don’t use “new” but call:

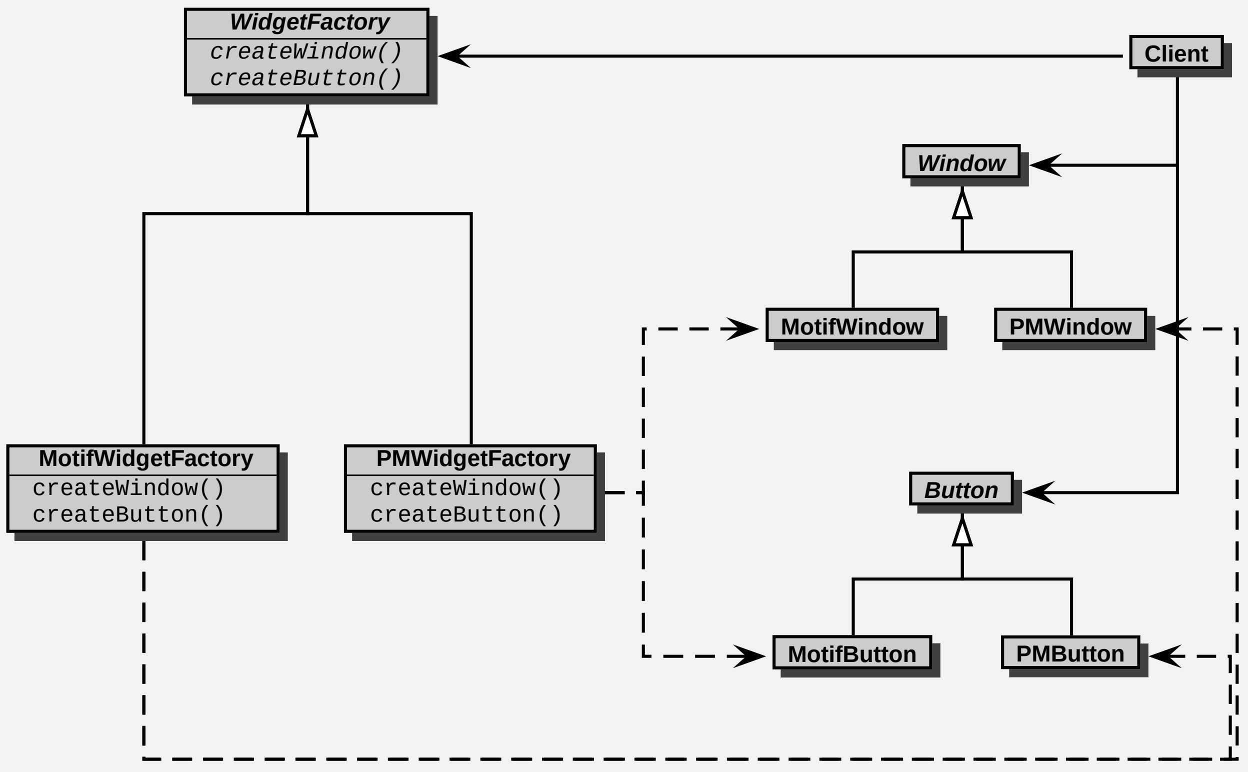
```
Button* b = wf->makeButton(args);
```

- Switch families – once in the code!
- Add a family – one new factory, no effect on existing code!

UML representation of the pattern



Pattern applied



Participants

- AbstractFactory (WidgetFactory)
  - declares an interface for operations that create abstract product objects.
- ConcreteFactory (MotifWidgetFactory, PMWidgetFactory)
  - implements the operations to create concrete product objects.
- AbstractProduct (Window, ScrollBar)
  - declares an interface for a type of product object.
- ConcreteProduct (MotifWindow, MotifScrollBar)
  - defines a product object to be created by the corresponding concrete factory.
  - implements the AbstractProduct interface.
- Client
  - uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Comments

- Pros:
  - *It makes exchanging product families easy*. It is easy to change the concrete factory that an application uses. It can use different product configurations simply by changing the concrete factory.
  - *It promotes consistency among products*. When product objects in a family are designed to work together, it's important that an application use objects from only one family at a time time.
    AbstractFactory makes this easy to enforce.
- Cons:
  - Not easy to extend the abstract factory's interface
- Other patterns:
  - Usually one factory per application, a perfect example of a singleton

Known uses

- Different operating systems (could be Button, could be File)
- Different look-and-feel standards
- Different communication protocols

Composite pattern

- We must write a complex program that has to treat several object in a hierachical way
  - Objects are composed togheter to create more complex objects
  - For example, a painting program treats shapes, that can be composed of more simple shapes (lines, squares, triangles, etc.)
  - Composite objects must be treated like simple ones
  - Another example: a word processor, which allows the user to compose a page consisting of letters, figures, and compositions of more elementary objects
- Requirements:
  - Treat simple and complex objects uniformly in code – move, erase, rotate and set color work on all
  - Some composite objects are defined statically (wheels), while others dynamically (user selection)
  - Composite objects can be made of other composite objects

Solution

- All simple objects inherit from a common interface, say Graphic:

```
class Graphic {
public:
    virtual void move(int x, int y) = 0;
    virtual void setColor(Color c) = 0;
    virtual void rotate(double angle) = 0;
};
```

- The classes Line, Circle and others inherit Graphic and add specific features (radius, length, etc.)

```
class CompositeGraphic
  : public Graphic,
    public list<Graphic>
{
    void rotate(double angle) {
        for (int i=0; i<count(); i++)
            item(i)->rotate();
    }
}
```

The solution – II

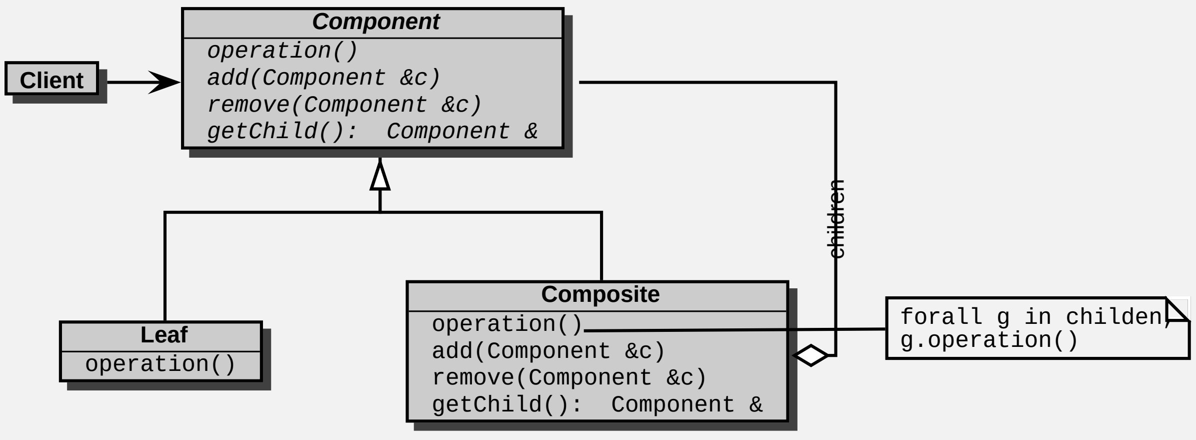
- Since a CompositeGraphic is a list, it had add(), remove() and count() methods
- Since it is also a Graphic, it has rotate(), move() and setColor() too
- Such operations on a composite object work using a “forall” loop
- Works even when a composite holds other composites – results in a tree-like data structure

The solution – III

- Example of creating a composite

```
CompositeGraphic *cg;
  cg = new CompositeGraphic();
  cg->add(new Line(0,0,100,100));
  cg->add(new Circle(50,50,100));
  cg->add(t); // dynamic text graphic
  cg->remove(2);
```

UML representation



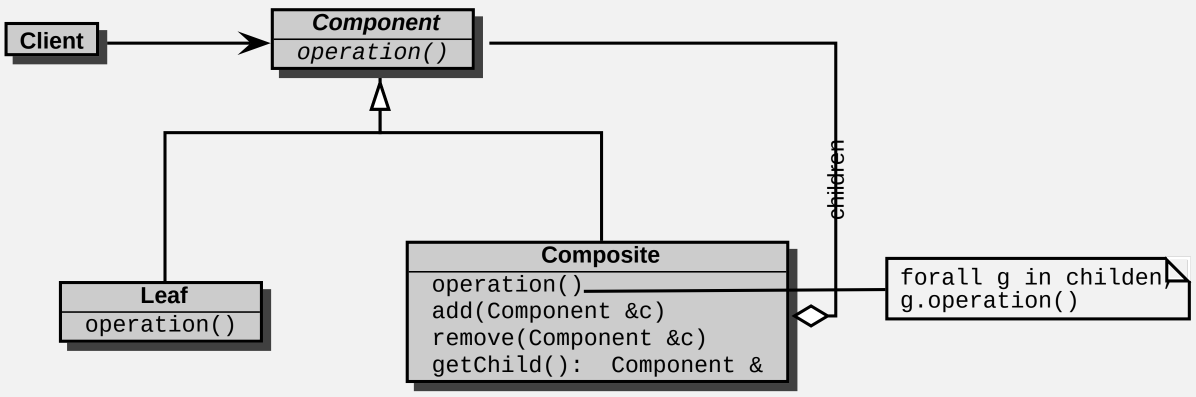
Participants

- Component (Graphic)
  - declares the interface for objects in the composition
  - implements default behavior for the interface common to all classes, as appropriate
  - declares an interface for accessing and managing its child components
  - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate
- Leaf (Rectangle, Line, Text, etc.)
  - represents leaf objects in the composition. A leaf has no children
  - defines behavior for primitive objects in the composition
- Composite (Picture)
  - defines behavior for components having children
  - stores child components
  - implements child-related operations in the Component interface
- Client
  - manipulates objects in the composition through the Component interface

Trade-off between transparency and safety

- Although the Composite class implements the Add and Remove operations for managing children, an important issue in the Composite pattern is which classes declare these operations in the Composite class hierarchy
- For transparency, define child management code at the root of the hierarchy. Thus, you can treat all components uniformly. It costs you safety, however, because clients may try to do meaningless things like add and remove objects from leaves
- For safety, define child management in the Composite class. Thus, any attempt to add or remove objects from leaves will be caught at compile-time. But you lose transparency, because leaves and composites have different interfaces

Composite for safety



Known uses

- In almost all O-O systems
- Document editing programs
- GUI (a form is a composite widget)
- Compiler parse trees (a function is composed of simpler statements or function calls, same for modules)
- Financial assets can be simple (stocks, options) or a composite portfolio

Bibliography

- Cristopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdhal-King, and Shlomo Angel.
  A pattern language.
  Oxford University Press, 1997.
- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
  Design patterns: elements of reusable object-oriented software.
  Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- Bertrand Meyer.
  Object-Oriented Software Construction.
  Prentice Hall, 1988.