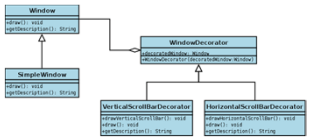


Patterns in GUI programming

Cristian Bogdan
cristi@kth.se

Pattern illustration: Composite

- Problem: working with complex, hierarchical whole-part structures
- AWT solution
  - java.awt.Container extends java.awt.Component
  - Container contains many components (including other Containers!)
  - Painting a Container involves painting the Components it contains (including other Containers…)
- Android solution
  - android.widget.ViewGroup extends android.Widget.View
  - ViewGroup contains many views (including other ViewGroups!)
  - Painting a ViewGroup involves painting the Views it contains (including other ViewGroups…)
- Solutions are not identical but follow same pattern

Pattern illustration: Observer

- Problem: maintain one-to-many dependency via notification
- Solution 1: Observable and Observer
  - Observable: addObserver(), removeObserver()
  - Observer: udpate(…, Object event)
- Solution2: JComponent and XXXListener
  - component: addXXXListener(), removeXXXListener()
  - listener: xxxHappened(XXXEvent e) (e.g. keyPressed())
- Solutions are not identical but follow same pattern
- Reusing Observable-Observer *classes* for GUI events would not have been practical
- Reusing / applying the Observer *pattern* is much easier

OO: No silver bullet

- Fred Brooks (1986)
- It is difficult to get a OO design (classes and their relations) right from the beginning
- *Refactoring* is standard practice by now, re-structure the code without changing overall behavior as we learn more
- In fact, any domain can be modeled in a large number of ways, all correct, depending on our priorities, concerns (see frameworks later)
- Class is maybe not the suitable abstraction
- Pattern: a reoccurring problem and an OO design that addresses it

Patterns in GUI programming

- GUI was a pioneering area for Object-Oriented Programming
- Consequently many principles and techniques originated from GUI programming
- Including many OOP design patterns
- Lecture objectives:
  - visit a number of patterns and exemplify their usage in AWT/Swing
  - re-visit MVC and hierarchical MVC and put them on the pattern map

Design Patterns (Designmönster)

- Emerge from architect Christopher Alexander work in the 1970s
- Alexander tried to find patterns in how one tried to solve re-occurring problems in architecture practice and wrote a book where he describes the patterns and typical solutions
  - Alexander, C. et al. (1977). **A Pattern Language**. Oxford University Press
  - see also **Notes on the Synthesis of Form**, 1964
- The idea was adopted in systemdesign in the mid 1990s
- Best known book Gamma et al. (1995). **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley (Gang of Four, GOF)

Alexander pattern: Window on Two Sides of Every Room

- This pattern does not prescribe the size of the windows, or the distance between them, their height from the floor, what frames etc. The improved quality comes from the incontestable fact that a room with two windows is better then a room with a single window.

Alexander pattern: A Place to Wait

- Dedicated waiting (in doctor's reception, in an airport etc) is not a enjoyable experience (unpredictability). Solution: create a situation which makes waiting positive — fuse in other activities (magazines, coffee, etc), something which draws people in, not just waiting, or, the opposite, make waiting place to draw people into reverie, quiet, positive silence

Design patterns

- Documenting design patterns led to the invention of the Wiki

- Name and general type
- Intent
- Also Known As
- Motivation

  - problem addressed, set of forces
- Applicability
- Structure (UML-diagram)

Design patterns (cont)

- Participants (description of all classes that participate)
- Collaborations (how class instances work together)
- Consequences (of using the pattern)
- Implementation
- Sample code
- Known uses
- Related patterns

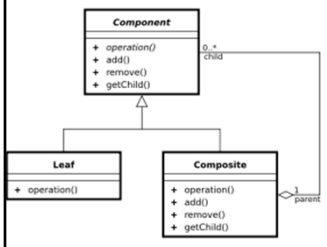
Pattern Language

- Patterns in a domain form a small vocabulary of well-known terms
- Thus practitioners in the domain can communicate easier if they know the patterns
- "You use this *Observer* and combine here with a *Template Method*"…
- So well-defined patterns will create a language for that practice

Composite

- Used to represent part-whole hierarchies
- Tree-like structures
- Individual objects and compositions can be treated uniformly
- Example: JComponent can contain JComponent's
  - A JFrame can contain a JMenu and two JPanels
  - Each JPanel can contain simple components (e.g. JButtons) but also other JPanels…

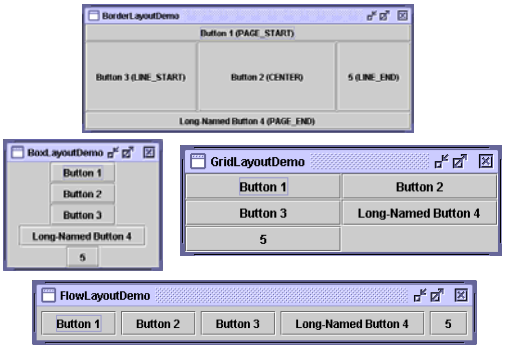
Composite



- Component: java.awt.Component, android.view.View
- Leaf example: Button, JButton..., android TextView...
- Composite superclass: java.awt.Container, android.view.ViewGroup
- Composite examples: Panel, Frame, android AbsoluteLayout, FrameLayout, LinearLayout...
- operation() example: paint(Graphics g)
- JComponent is a subclass of Container (not Component!), so all Swing components are composites, thus they can contain other components
  - e.g. Jtable, JMenuBar use this, JButton doesn't always

Typically Composite's operation() iterates through the children and invokes their operation() (e.g. paint(), layout, event propagation)

LayoutManager as Strategy



Strategy examples

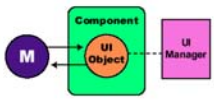
- Laying out Components in a Container. Container just keeps track of which are its components, does not know/care how they are positioned
- Pluggable look and feel. The application only knows the types and position of components, not their actual look. That is delegated to a painting strategy
- In the MVC paradigm, if a View has multiple controllers, they can be regarded as Strategies, as they are different ways to interpret events that occur on the View. The view itself is not concerned with interpreting the events.
  - Drawing tool, one controller Strategy per tool

Strategy

- Define a family of algorithms
- Encapsulate each in an object
- Use them interchangeably
- Algorithms vary independently from their clients
  - Andorid did not separate layouts as strategies! Instead they included the layout behavior in specific kinds of composites

UI Manager as Strategy

- The drawing is delegated to the UI Object, which calls a UI manager so that we are able to change look and feel by just changing the manager

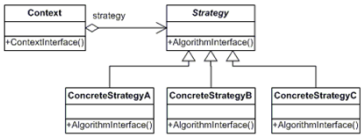


UIManager as Strategy

- Plugging another look-and-feel strategy for the overall application
- We don't pass an object but a strategy name

```
// Install the look and feel
try {
        UIManager.setLookAndFeel(
            "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
}
catch(Exception e) {
}
```

Strategy

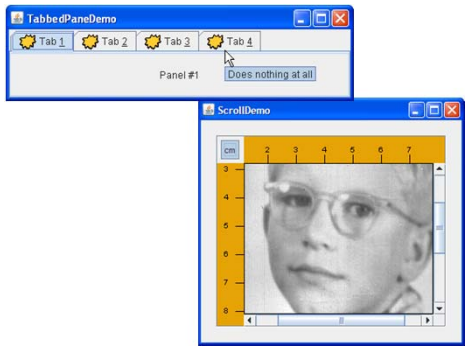


•Strategy example: java.awt.LayoutManager

•Concrete strategy examples: FlowLayout, BorderLayout, etc

•Context (client) example: java.awt.Container, invokes the strategy when it needs to position (lay out) its contained Components

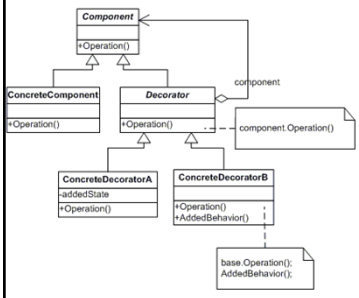
Decorator



Decorator

- Attach additional behavior to an object dynamically
- Unlike subclassing, it adds behavior at runtime
- Examples:
  - Add scrollbars to any component, to be able to show it in less space than it would take. We do not know/care about drawing the component (ScrollPane)
  - Draw tabs above/near any component, to be able to switch to another component (TabbedPane)

Decorator



Example:

- Component: JComponent
- ConcreteDecorator: ScrollPane
- •Operation(): paint()
- •AddedBehavior():
  - •draw scrollbars
  - •Change parameters for the decorated paint() so that the decorated component is painted at the position given by the scrollbars

Other Decorators in Java

- Decorating InputStream and OutputStream
- FileInputStream "knows" how to read bytes
- BufferedInputStream doesn't know how to read bytes, but it knows how to buffer what it reads
  - Added behavior: buffering, flushing
  - Added state: buffer, buffer size
- To do the actual byte reading, BufferedInputStream will always need to invoke read() on the decorated stream (FileInputStream, SocketInputStream, etc).
- All BufferedInputStream needs to know is that the decorated object is an InputStream
- Other InputStream decorators: DataInputStream, PushbackInputStream, etc etc

Android TabHost as template/decorator

- TabHost expects a TabWidget and a FrameLayout inside it, with specific IDs
- You can place the TabWidget and FrameLayout as you wish
- TabHost will add the "tabbed" user interaction behavior to them

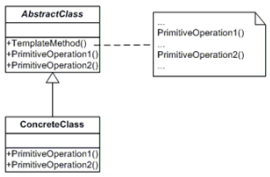
Template Method: JComponent paint()

- A JComponent needs to layout and paint its contents
- Generic pseudo-code:
  - Ask the Layout Manager to position all the JComponents in the container (invoke layout())
  - For all components
    - set background and foreground colors
    - invoke component paint()

Template Method

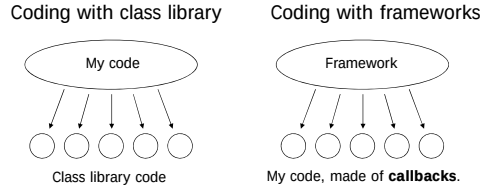
- Define the skeleton of an algorithm
- Leave some algorithm steps to subclasses
- Subclasses can thus redefine algorithm steps without changing the algorithm structure

Template Method



- Primitive operations in the JComponent paint example: layout(), paint()
- The AWT/Swing template method invokes also related classes, not just subclasses
- Template Method is at the core of frameworks

Frameworks



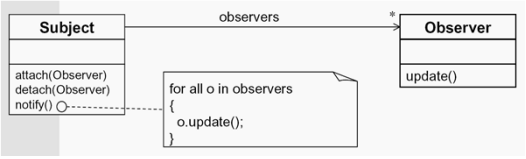
•Fundament: we implement the callbacks, but never call them. We wait for the framework to call them (Hollywood principle)

•The framework invokes the callbacks when they are required by its Template Methods

Observer

- Define a one-to-many dependency between objects
- so that when one object changes state, all its dependents are notified and updated automatically
- Example: in Model-View Controller, both Model-View and View-Controller relationships are Observer relationships

Observer



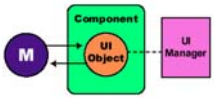
- Subject: java.util.Observable
- Observer: java.util.Observer
- Subject: java.awt.Component
- Observer: java.awt.event.*Listener
  - more specialized observer
  - with several finer-grained methods

Re-visiting MVC

- MVC is a combination of more specific patterns (Observer, Strategy)
- So MVC is an *Architectural* pattern
- Many patterns have this kind of "typical combinations" with other patterns
- Swing applies MVC in a hierarchical way
  - Besides the "basic" application model, Swing defines models for its components
  - These models will be populated with some processing and selection of data from the basic model
  - In between the "basic" and "component-level" models, one can define other intermediate models

Separable Model Architecture

- Each component manages view/control.
- Each component has a model associated
- Hierarchical MVC
- http://www.javaworld.com/javaworld/jw-07-2000/jw-0721-hmvc_p.html
- Presentation-Abstraction-Control (PAC)



Synchronizing Swing model with abstract model

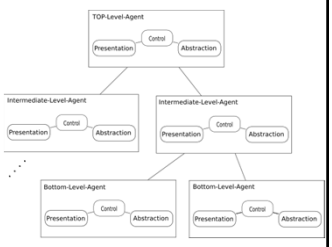
- Approach 1: Listener on the Swing model changes abstract model (e.g. DocumentListener)
- Problems when the Swing model changes
  - Mutation: Swing model notifies listener, listener changes Abstract model, Abstract model notifies views, Swing model must be changed but cannot be changed during event propagation.
    - Solution: SwingUtilities.invokeLater(Runnable)
  - Infinite loop/false change: Swing model notifies listener, listener changes Abstract model, Abstract model , notifies views, Swing model changes, produces another event, listener gets the event, abstract model is changed…
    - Solution: don't call setChanged() in the abstract model unless it really changes

Synchronizing Swing model with abstract model

- Approach 2: Implement/subclass the Swing model
- Override its mutation operations to change the abstract model when the Swing model really changes
- E.g. extend PlainDocument instead of implementing DocumentListener
  - Document is a pretty complex interface, so we better extend a class that implements it
- The custom model can replace the default model of the Swing component by calling setModel()
- Overriding the "observable" is thus sometimes preferable to the Observer pattern

Presentation-Abstraction-Control

- The application is a hierarchy of cooperating *agents*
- Presentation: "visible" agent behavior
- Abstraction: data model and operations
- Control:
  - connect P and A
  - connect to other agents



- Top-level agent: system's functional core
- Bottom-level: user interfaces

Presentation-Abstraction-Control

- Can be used in complex systems where agents are heavily concurrent
- E.g. Robots with their various subsystems
- Leads to more rapid interface initialization
  - Presentation is not so heavily dependent on Abstraction like View is on Model in MVC
- In this conceptualization
  - Swing components like JTable correspond to bottom-level agents
  - Swing components' model is the Abstraction, their graphical view is the Presentation
  - The MVC overall Model corresponds to the top-level agent
  - Intermediate models can easily be accommodated

Bridge

- separate abstraction from implementation so they can vary independently
- old AWT 1.0 had a native *peer* (implementation) for each component (abstraction)
- the peer was different on each platform, drawing and event treatment was done in native code
- Swing does away with this and draws everything in Java
- SWT (used by Eclipse) brings back the native peers in a more efficient way

Abstract Factory

- Provide an interface to create families of related objects
- without specifying their concrete class
- AWT 1.0 used AF to initialize the native peers
  - The interface provided methods for creating Button peer, List peer, Label peer…
- Swing uses AF to create the UI objects that do the look-and-feel specific drawing

Other patterns in Swing

- **Command**: Encapsulate a request as an object,
  - thereby letting you parameterize clients with different requests
  - queue or log requests,
  - and support undoable operations
  - Supported via javax.swing.Action
- **Adapter**: Convert the interface of a class into another interface clients expect
  - See the AWT/Swing event adapters

Other GUI-related patterns

- **Flyweight**: Use sharing to support large numbers of fine-grained objects efficiently
  - E.g. a graphical word processor will not render each occurrence of a letter every time
- **Memento**: capture and externalize an object's internal state
  - so that the object can be restored to this state later.
  - without violating encapsulation
  - E.g. `java.io.Serializable` and the `transient` keyword

Pattern types

- Creational
  - Abstract Factory
- Structural
  - Composite, Decorator, Bridge, Flyweight
- Behavioral
  - Strategy, Template Method, Observer, Command, Memento
- Architectural
  - MVC, PAC