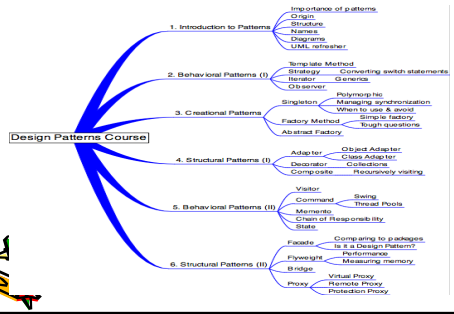
Design patterns

Observer,Strategi, Composite,Template
(Chap 5, 6)

The Pattern Concept

- Each pattern has
  - a short *name*
  - a brief description of the *context*
  - a lengthy description of the *problem*
  - a prescription for the *solution*

Desing patterns

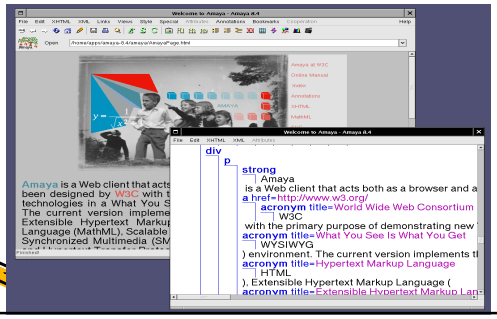


Multiple view

Some programs have multiple editable views

- editing one view updates the other
- updates seem instantaneous

Multiple view

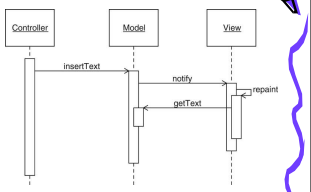


Model/View/Controller

- Model: data structure, no visual representation
- Views: visual representations
- Controllers: user interaction

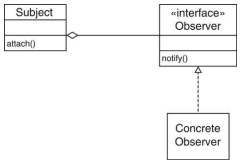
Model/View/Controller

- Controllers update model
- Model tells views that data has changed
- Views redraw themselves



Observer Pattern

- An object, called the subject, is source of events
- One or more observer objects want to be notified when such an event occurs.
- **Solution**
- Define an observer interface type. All concrete observers implement it.
- The subject maintains a collection of observers.
- The subject supplies methods for attaching and detaching observers.
- Whenever an event occurs, the subject notifies all observers.



Java Observable / Observers

```
class MyObservable extends
Observable {

public void drink() {
      name = "java kofee" ;
      setChanged();
      notifyObservers();
}
 public String getName() {
       return name;
      }
 private String drinkname = "java";
}
```

```
class Person implements Observer {

 public Person(String name, String says) {
   this.name = name;
   this.says = says;
 }
 public void update (Observable thing, Object o) {
   System.out.println("It's " + thing.getName() + "
+  name + ": " + says);
  }

 private String name;
 private String says;
}
```

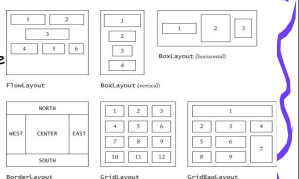
Put it work together

```
public class MainClass {
    public static void main(String[] args) {

   MyObservable man = new MyObservable();
      Observer[] people = {
          new Person("A", "a"),
          new Person("B", "b"),
          new Person("C", "c"),
   }
   for (Observer observer : crowd) {
       man. addObserver(observer);
      }
      man. drink() ;
    }
   }
```

Layout Managers

- User interfaces made up of *components*
- Components placed in *containers*
- Container needs to arrange components
- Swing doesn't use hard-coded pixel coordinates
- Advantages:
  - Can switch "look and feel"
- Layout manager controls arrangement



Layout Managers How it works

Set layout manager

```
JPanel keyPanel = new JPanel();
keyPanel.setLayout(new GridLayout(4, 3));
```

- Add components

```
for (int i = 0; i < 12; i++)
   keyPanel.add(button[i]);
```

containerObject. setLayout()

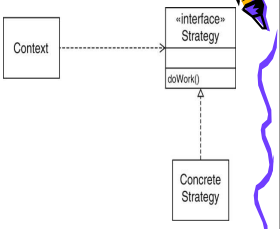
Set layout manager

```
.....setLayout( LayoutManager thelayout){
thelayout.preferredLayoutSize(... );
thelayout.layoutContainer(....);
.............
}
```

- The strategy pattern applies whenever you want to allow the client to supply an algoritm

Strategy Pattern

Context
- A class can benefit from different variants for an algorithm
- Clients sometimes want to replace standard algorithms with custom versions
- **Solution**
- Define an interface type that is an abstraction for the algorithm
- Clients can supply strategy objects
- Whenever the algorithm needs to be executed, the context class calls the appropriate methods of the strategy object

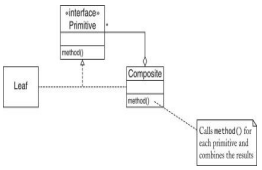


Containers and Components

- Containers collect GUI components
- Sometimes, want to add a container to another container
- Container should *be* a component
- Composite design pattern
- Composite method typically invoke component methods
- E.g. Container.getPreferredSize invokes getPreferredSize of components

Composite Pattern

- **Context**
- Primitive objects can be combined to composite objects
- Clients treat a composite object as a primitive object
- **Solution**
- Define an interface type that is an abstraction for the primitive objects
- Composite object collects primitive objects
- Composite and primitive classes implement same interface type.
- When implementing a method from the interface type, the composite class applies the method to its primitive objects and combines the results



How to Recognize Patterns

- Look at the *intent* of the pattern
- Remember common uses (e.g. STRATEGY for layout managers, Observers for updating views)
- Implement examples for comon patterns