Design Patterns

David Talby

This Lecture

- Handle Synchronization & Events
  - Observer
- Simplify Complex Interactions
  - Mediator
- Change Behavior Dynamically
  - Strategy, State
- Undo, Macros and Versions
  - Command

10. Observer

- Define a one-to-many dependency between objects, so that changing one automatically updates others
- For example, a spreadsheet and several charts of it are open
- Changing data in a window should be immediately reflected in all

The Requirements

- Document and Chart classes must not know each other, for reuse
- Easily add new kinds of charts or other links
- A dynamic number of charts

The Solution

- Terminology
  - Subject and Observer
  - Publisher and Subscriber
  - Listeners
- Subjects attach and detach listeners, and notify of events
- Clients update themselves after receiving a notification

The Solution II

- Here's an abstract observer:

```
class Observer {
   void update() = 0;
}
```

- Concrete observers such as class *Chart* will inherit *Observer*

The Solution III

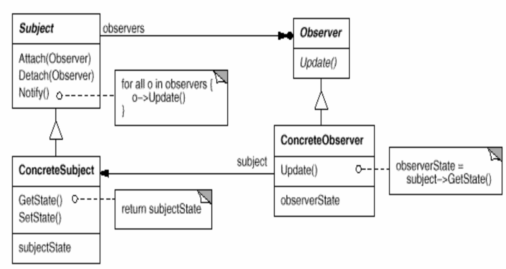
- Here's the (concrete!) subject:

```
class Subject {
   void attach(Observer *o)
        { observers.add(o); }
   void detach(Observer *o)
        { observers.remove(o); }
   void notify() {
     for i in observers do
        o->update();
   }
 protected:
   List observers;
}
```

The Solution IV

- Both subject and observer will usually inherit from other classes as well
- If multiple inheritance is not available, the observer must be a separate class that has a reference to the chart object and updates it
- Java has a special mechanism – *Inner classes* – to make this easier

The UML



The Fine Print

- Observing more than one subject
  - *Update* must include an extra argument to tell who is updating
- Observing only certain events
  - *Attach* must include an extra argument to tell which events interest this observer
- Observing small changes
  - *Update* includes arguments to tell what changed, for efficiency

The Fine Print II

- Who calls *Notify*?
  - Greedy – the subjects, on change
  - Lazy – the observers, on query
- Common errors
  - Forgetting to detach an object when it is destroyed
  - Calling *Notify* in an inconsistent state
- Java includes Observer as part of the standard libraries
  - In package *java.util*

Known Uses

- All frameworks of all kinds
  - MFC, COM, Java, EJB, MVC, …
- Handle user interface events
- Handle asynchronous messages

The Requirements

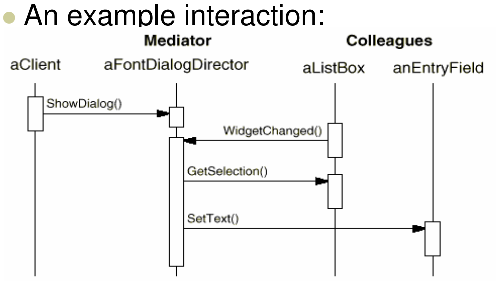
- A widget is a kind of *colleague*
- Colleague don’t know about the interactions they participate in
  - Can be reused for different dialogs
- Colleagues don’t know about others
  - Allow only *O(n)* connections
- Easy to change interactions

11. Mediator

- Encapsulate a complex interaction to preserve loose coupling
- Prevent many inter-connections between classes, which means that changing their behavior requires subclassing all of them
- For example, a dialog box includes many interactions of its widgets. How do we reuse the widgets?

The Solution II

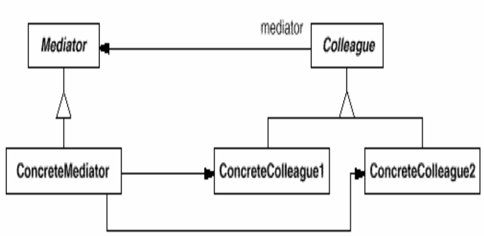
- An example interaction:



The Solution

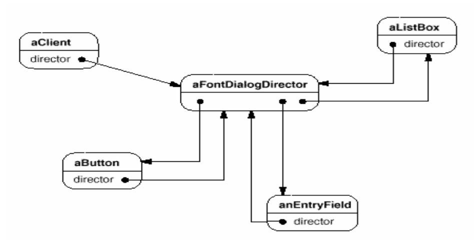
- All colleagues talk with a mediator
- The mediator knows all colleagues
- Whenever a colleague changes, it notifies its mediator
- The mediator codes the interaction logic, and calls operations on other colleagues

The UML



The Solution III

- Only *O(n)* connections:



Known Uses

- Widgets in a user interface
  - Delphi and VB "hide" this pattern
- Connectivity constraints in diagrams

The Fine Print

- The interaction logic (mediator) and colleagues can be reused separately and subclassed separately
- Protocols are simpler since n-to-1 relations replace n-to-m relations
- Abstract mediator class is unnecessary if there's only one mediator
- Observer or mediator?
  - One-to-many or many-to-many?
  - Should the logic be centralized?

The Requirements

- Algorithms are complex, would be havoc to have them inside the one *Document* class
- Switch algorithms dynamically
- Easy to add new algorithms

12. Strategy

- A program must switch between complex algorithms dynamically
- For example, a document editor has several rendering algorithms, with different time/beauty tradeoffs
- Word is a common example

The Solution II

- The document itself chooses the rendering algorithm:

```
class Document {
  render() {
    renderer->render(this);
  }
  setFastRendering() {
    renderer = new FastRenderer();
  }
  private: Renderer *renderer;
}
```

The Solution

- Define an abstract class that represents an algorithm:

```
class Renderer {
   void render(Document *d) = 0;
}
```

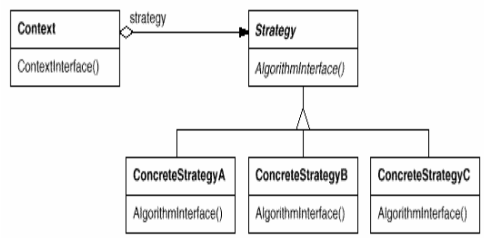
- Each specific algorithm will be a descendant class

```
FastRenderer, TexRenderer, …
```

The Fine Print

- Inheriting a strategy would deny a dynamic switch
- Some strategies may not use all information passed from Context
- Strategies can be stateless, and then they can be shared
- In some cases strategy objects are optional

The GoF UML



13. State

- Allow an object to alter its behavior when its internal state changes
- For example, most methods of a *TCPConnection* object behave in different ways when the connection is closed, established or listening
- How do we encapsulate the logic and data of every state?

Known Uses

- Document rendering programs
- Compiler code optimizations
- Different heuristic algorithms (games, portfolio selection)
- Different memory management schemes (Booch components)
- Validation of dialog boxes (optional strategies, Borland ObjectWindows)

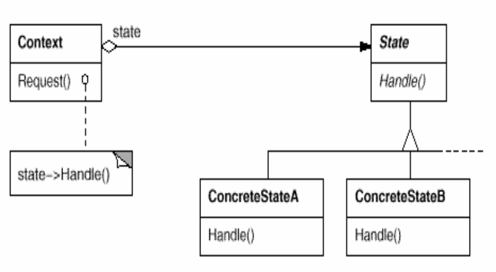
Pattern of Patterns

- Encapsulate the varying aspect
  - State of an object
- Interfaces
  - Let's have a *TCPState* interface that has all the state-sensitive methods
- Inheritance describes variants
  - *TCPEstablished*, *TCPListen* and *TCPClosed* implement the interface
- Composition allows a dynamic choice between variants

The Requirements

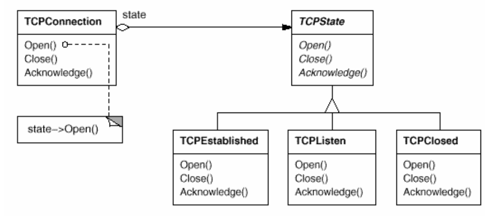
- A class has a state diagram, and many methods behave in wildly different ways in different states
- When in a state, only allocate memory for data of that state
- The logic of a specific state should be encapsulated

The UML



The Solution II

- A *TCPConnection* codes state transitions and refers to a *TCPState*



Known Uses

- Streams and connections
- Different tools on a drawing program
  - Select, Erase, Crop, Rotate, Add, …

The Fine Print

- In complex cases it is better to let states define transitions, by adding a *SetState* method to *Context*
- States may be created on-demand or on *Context*'s creation
- This pattern is really a workaround for the lack of *dynamic inheritance*
- State is very much like Strategy
  - State = Many (small) actions
  - Strategy = One (complex) action

The Requirements I

- Undo / redo at unlimited depth
- Only store relevant data for undo
- Easy to add commands

14. Command

- Encapsulate commands as objects
- We'll take the the uses one by one:
  - Undo/Redo
  - Macros
  - Queues and logs
  - Version control
  - Crash recovery
  - Message Grouping

The Solution

- Repesent a command as a class:

```
class Command
{
 public:
   virtual void execute() = 0;
   virtual void undo() = 0;
}
```

The Solution II

- Concrete commands hold undo data:

```
class DeleteLine : public Command {
   void execute() {
     line = document->getLine();
     document->removeLine();
   }
   void undo() {
     document->addLine(line);
   }
 private:
    Line line;
}
```

The Solution III

- Keep a list of executed commands:

```
Array<Command*> commands;
int i;
```

- When you click the ‘Undo’ button:

```
commands(i)->undo();
i--;
```

- When you click the ‘Redo’ button:

```
commands(i)->execute();
i++;
```

The Solution IV

- Whenever a command is activated:

```
commands.add(new_command);
i = commands.count();
```

- When you save a document:

```
document->save();
commands.clear();
i = 0;
```

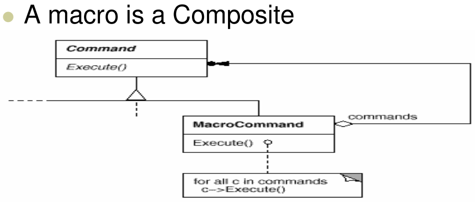
- The commands list may or may not be limited in size
- Only relevant undo data is kept

The Requirements II

- Macros are a series of commands
- Any command with any of its options may be used
- There are also *for* and *while* loops, *if* statements, calls to other macros...

The Solution

- A macro is a Composite



- *if*, *for*, *while* are Decorator Commands

The Solution

- Each *MenuItem* or *ToolbarItem* refers to its command object
- Just as it refers to an image
- The command can be configured
  - Less command classes
- Macros fit in the picture as well!

The Requirements III

- Commands are accessible from menus as well as toolbars
- A command may be available from more than one place
- We'd like to configure the menus and toolbars at runtime

The Solution

- The changes are exactly the list of commands since the last version was loaded
- In addition, a compaction algorithm is needed for commands that cancel each other
- Save = Serialize the compacted list
- Load = Read early version and call execute on command lists

The Requirements IV

- Keep multiple versions of a document
- When saving, only store the changes from the previous version

Summary

- Pattern of patterns
  - Encapsulate the varying aspect
  - Interfaces
  - Inheritance describes variants
  - Composition allows a dynamic choice between variants
- Design patterns are old, well known and thoroughly tested ideas
  - Over twenty years!

(More!) Known Uses

- Programs log commands to disk so they can be used in case of a crash
  - Works, since commands are small
  - Usually in a background thread
- Commands can be grouped and sent as one command to a server
  - Grouping for efficient communication
  - Grouping to define a transaction
  - Works even for user defined macros!