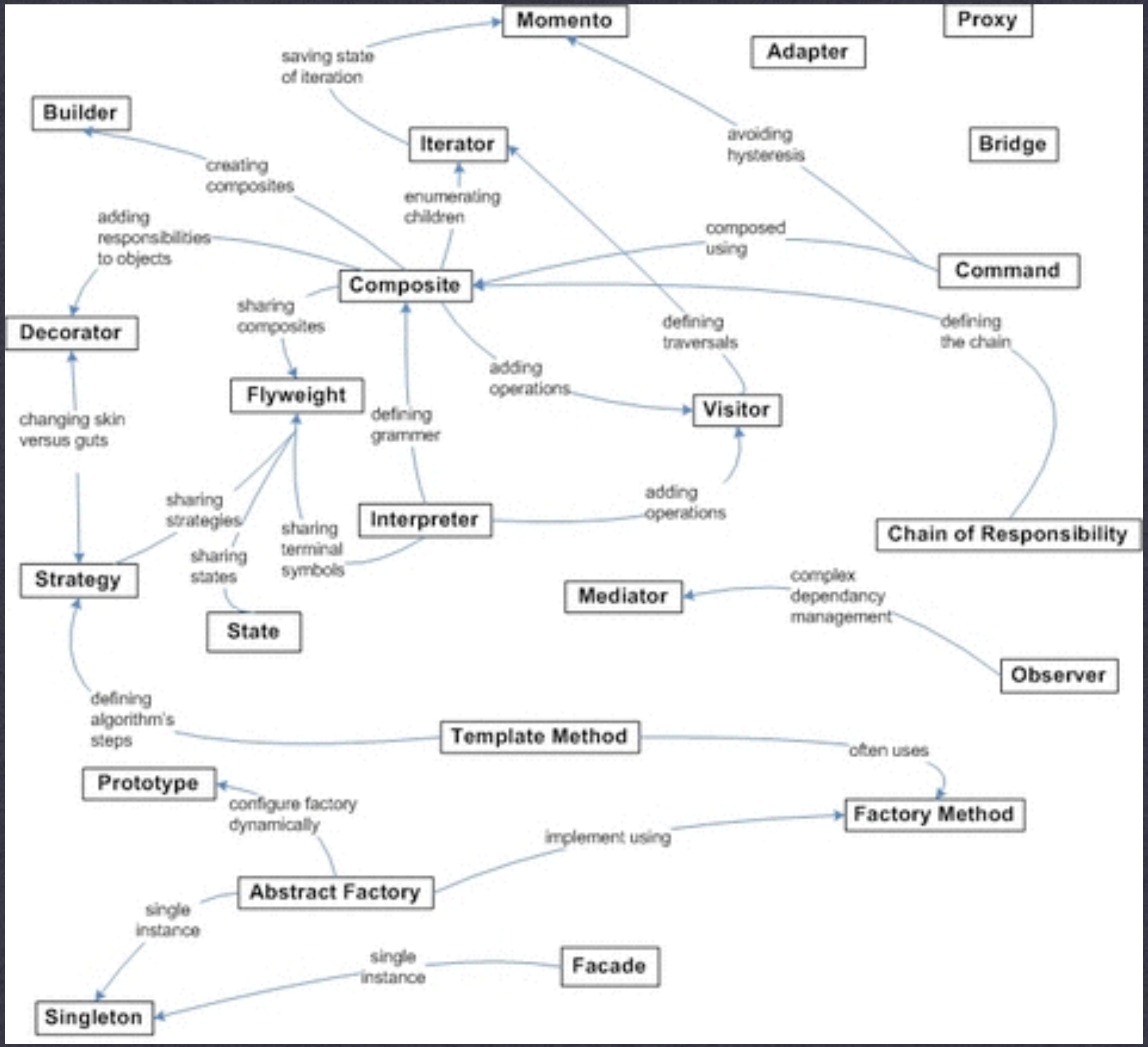
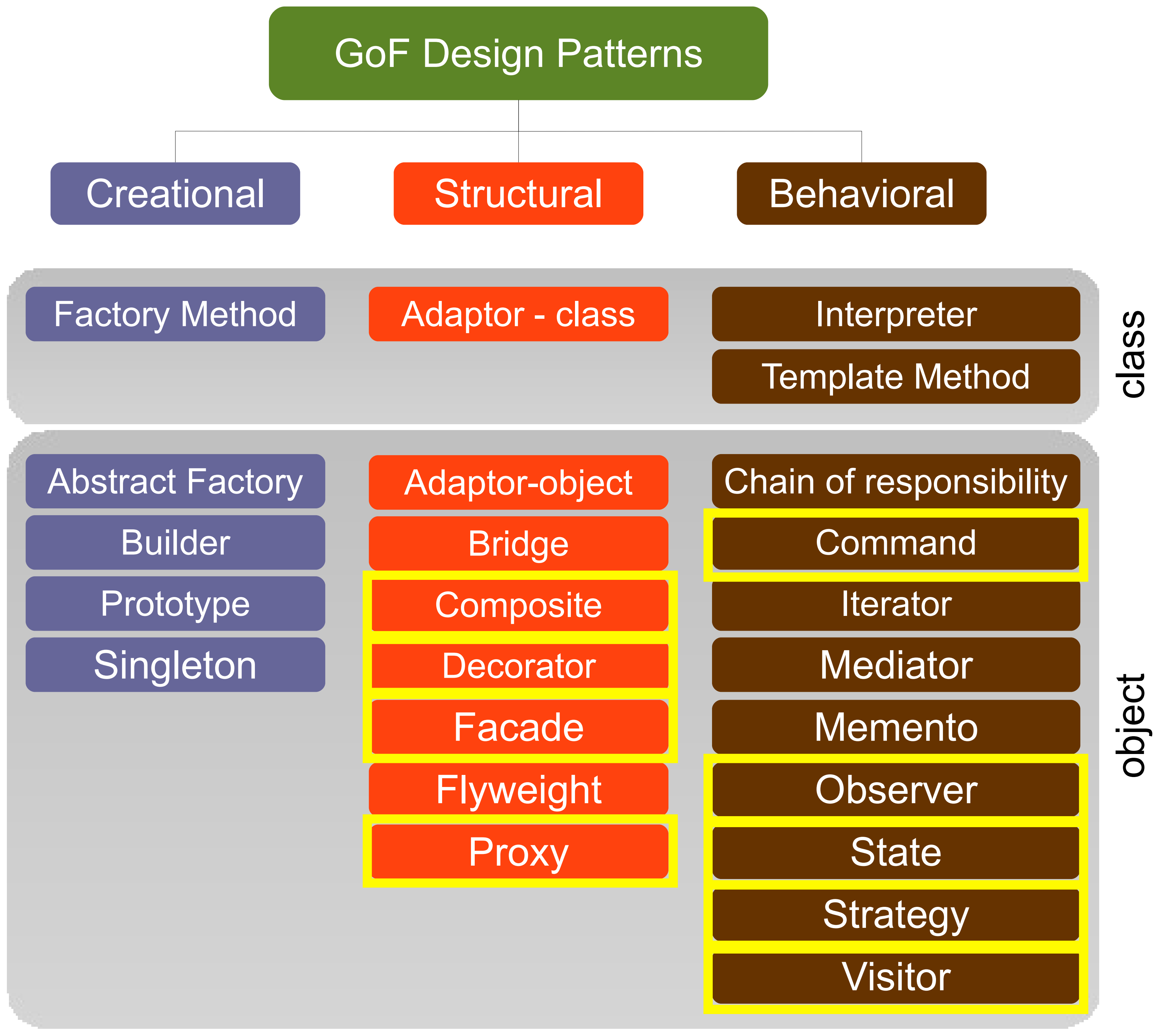


Material and some slide content from:
- Head First Design Patterns Book
- GoF Design Patterns Book

Design Patterns

Reid Holmes

GoF design patterns



Pattern vocabulary

- Shared vocabulary
  - communicate qualities
  - reduce verbosity
  - focus on design
  - increase understanding

Observer example

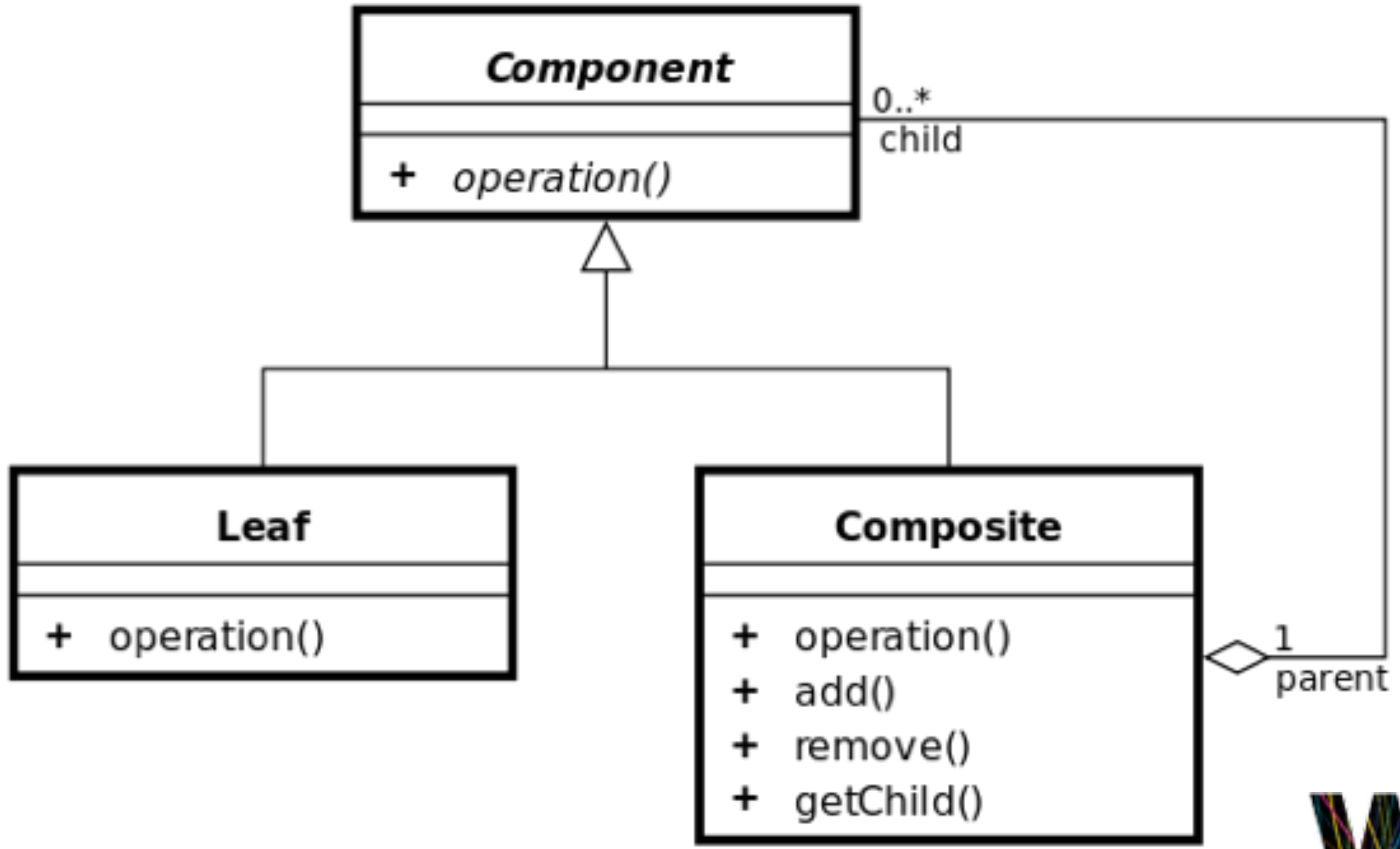
- Weather data example (similar to Eclipse example)
  - WeatherData
    - temp, humidity, pressure
    - calls newData() whenever something changes
      - bad: update views directly from here
  - WeatherViews
    - Current View
    - Forecast View
    - Stats View

Composite

- Intent: “Enable a group of objects to be treated as single object”
- Motivation: Differentiating between interior and leaf nodes in tree-structured data data increases system complexity.
- Applicability:
  - If you notice you are treating groups and individual of objects the same way
  - Can also be used when primitives and objects need to be treated identically

Composite

- Participants:
  - Component: base class
  - Leaf: individual leaf node
  - Composite: node that maintains a list of children nodes



Composite

- Implementation:
  - 1) Composite maintains a list of child elements and methods to maintain the children.
  - 2) Composite object applies overridden methods from the component across all child methods.
- Known uses:
- Related to: Decorators are often used along with the Composite pattern to augment objects while grouping them.

Facade

- Intent: “Provide a unified, higher-level, interface to a whole module making it easier to use.”
- Motivation: Composing classes into subsystems reduces complexity. Using a Facade minimizes the communication dependencies between subsystems.
- Applicability:
  - When you want a simple interface to a complex subsystem.
  - There are many dependencies between clients and a subsystem.
  - You want to layer your subsystems.

Facade

- Participants:
  - Facade
  - Subsystem classes
- Collaborations:
  - Clients interact subsystem via Facade.
- Consequences:
  - Shields clients from subsystem components.
  - Promotes weak coupling. (strong within subsystem, weak between them)
  - Doesn't prevent access to subsystem classes.

Facade

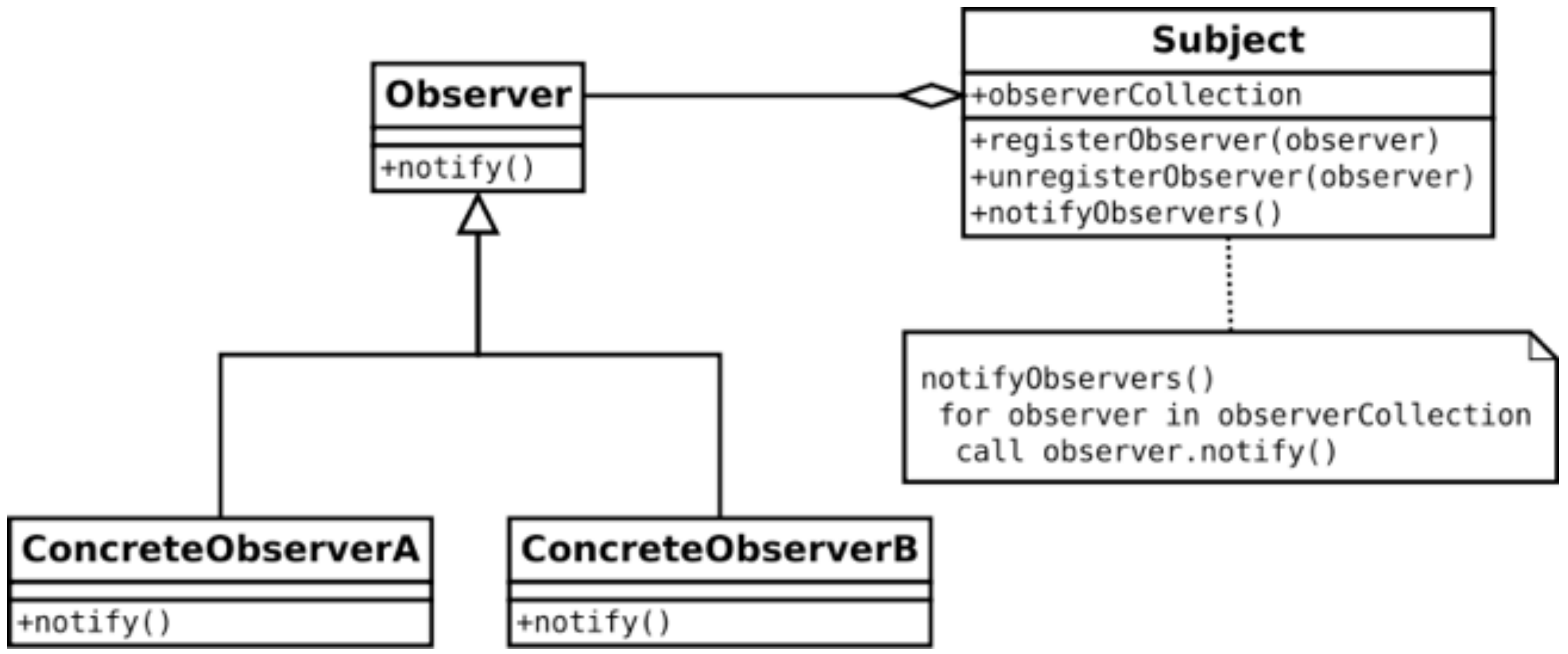
- Implementation:
  - 1) Analyze client / subsystem tangling.
  - 2) Create interface. Abstract factories can also be used to add further decoupling.
- Known uses: Varied.
- Related to: Abstract Factory can be used with Facade to create subsystem objects. Facades are frequently Singletons. Abstracts functionality similar to Mediator but does not concentrate on communication.

Observer

- Intent: Define a one-to-many relationship between objects so that when an object changes state its dependents are updated automatically
- Motivation: To maintain consistency between multiple different objects without tightly coupling them
- Applicability:
  - When you want to compartmentalize modifications to two dependent objects
  - When you want to publish updates but not couple classes

Observer

- Structure:



```
notifyObservers()
 for observer in observerCollection
  call observer.notify()
```

- Participants:
  - Subject: tracks observers and fires updates
  - Observer: subscribes/unsubscribes to subjects, receives updates

Observer

- Collaborations
  - Subjects call observer's update method when they change
  - Subjects can forward data (push) or just send blank update notifications (pull)
- Consequences:
  - Reduce coupling between subject & observer
  - Support broadcast communication
  - Can result in expensive updates

Observer

- Implementation:
  1. Subjects track observers (abstract class helpful)
  2. Caching updates
  3. Push vs. pull
- Related to:
  - Employed by MVC & MVP.

GWT example

```
Window.addResizeHandler(new ResizeHandler() {
        @Override
        public void onResize(ResizeEvent event) {
           if (event.getWidth() > event.getHeight()) {
              setPortrait(false);
           } else {
              setPortrait(true);
           }
        }
     });
```

Command

- Intent: “Encapsulate requests enabling clients to log / undo them as required.”
- Motivation: In situations where you need to be able to make requests to objects without knowing anything about the request itself or the receiver of the request, the command pattern enables you to pass requests as objects.
- Applicability:
  - Parameterize requests.
  - Specify, queue, and log actions.
  - Support undo.
  - Model high-level operations on primitive operations.

Command

- Structure

- Participants:
  - Command / ConcreteCommand
  - Client
  - Invoker
  - Receiver

Command

- Collaborations:
  - Client creates ConcreteCommand and specifies receiver.
  - Invoker stores ConcreteCommand object.
  - Invoker requests execute on Command; stores state for undoing prior to execute (if undoable).
  - Concrete invokes operations on its receiver to perform request.
- Consequences:
  - Decouples the invoker from the object that knows how to perform an action.
  - Commands are first-class objects.
  - Commands can be assembled into composite.
  - Adding new commands is easy.

Command

- Implementation:
  - 1) How smart should a command be?
  - 2) Support undo/redo.
  - 3) Avoiding error accumulation in the undo process.
- Related to: Composite commands can be created; the Memento pattern can store undo state. Commands often use Prototype when they need to be stored for undo/redo.

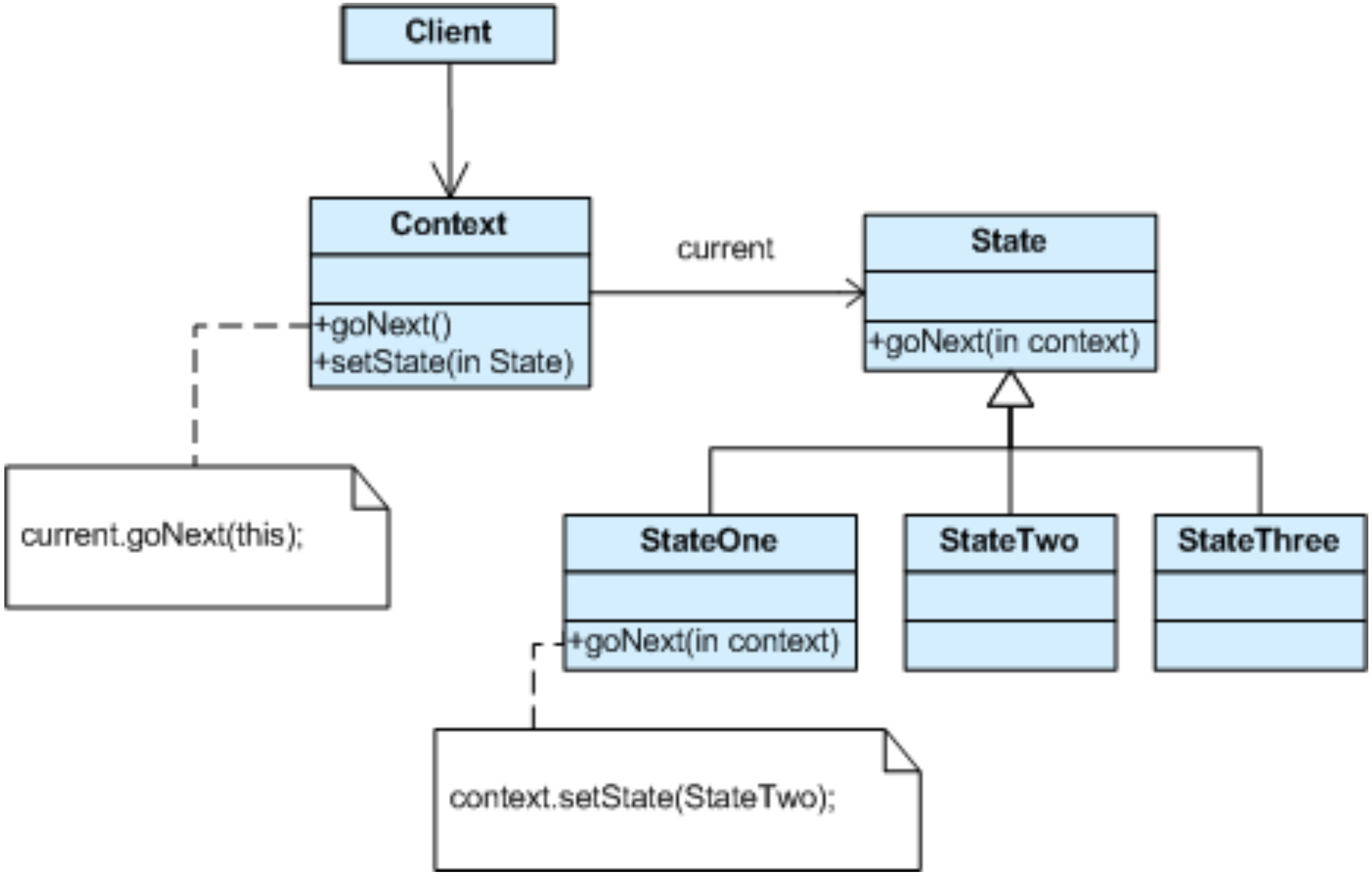
State

- Intent: “Allow an object to alter its behaviour when its internal state changes; the object will appear to change its class.”
- Motivation: Systems often have a limited set of discrete states and will behave differently depending on its current state. The state pattern enables control of state-dependent operations.
- Applicability:
  - Systems where runtime behaviour is dependent on the state of the system.
  - When decisions are based on many flags that are in effect checking state before deciding which operation to perform.

State

- Participants:
  - Context: maintain State reference; delegates behaviour through composition to State class.
  - IState: define abstract operations
  - ConcreteState: implement IState, update Context as required.
- Consequences:
  - Localizes state-specific behaviour.
  - Makes state-transitions explicit.

State


Client
Context
+goNext()
+setState(in State)
current
State
+goNext(in context)
StateOne
+goNext(in context)
StateTwo
StateThree
current.goNext(this);
context.setState(StateTwo);

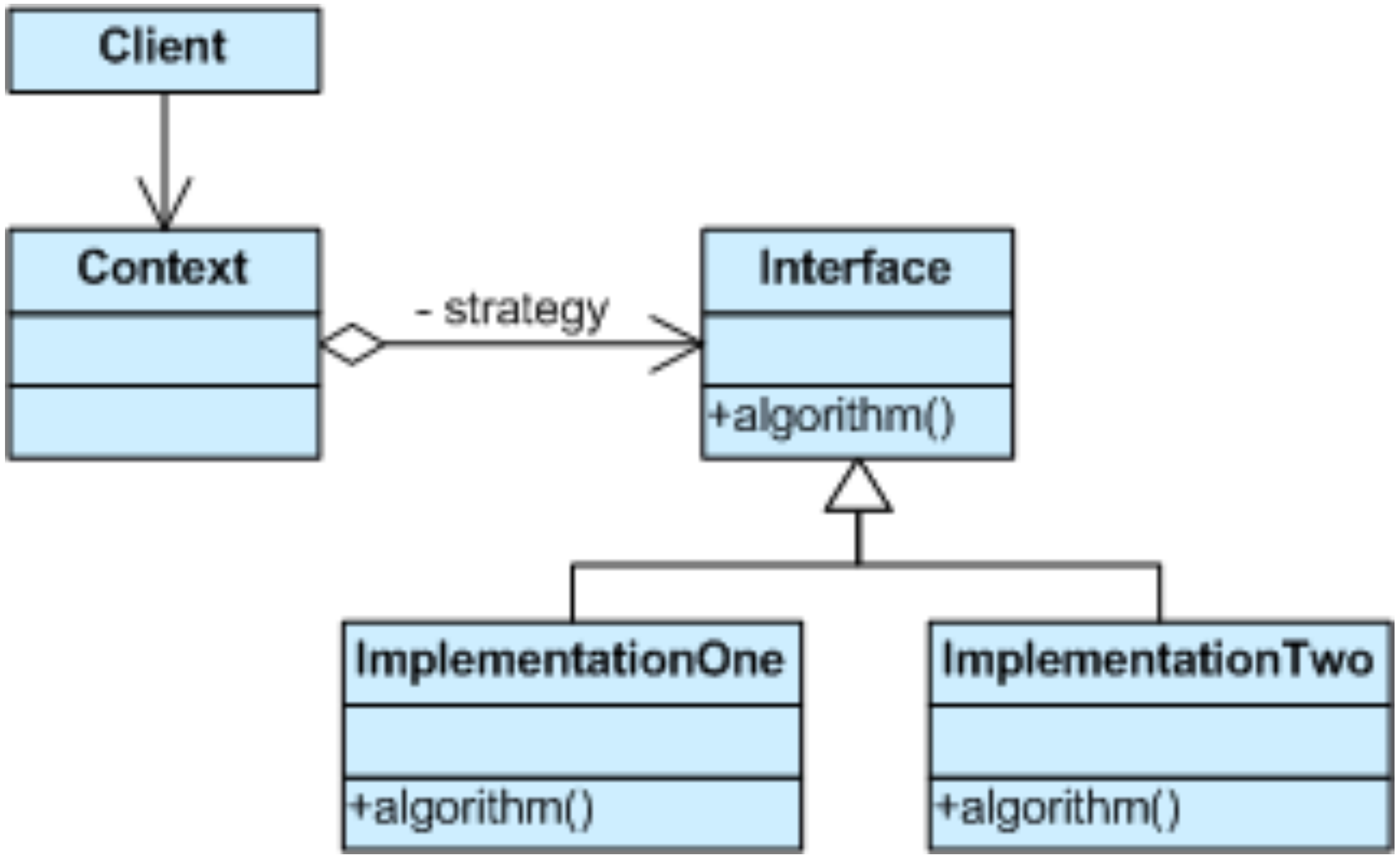
State

- Implementation:
  - 1) Context maintains a reference to a State.
  - 2) Concrete states implement state and have a reference to the context. On state-transition operations, the current state object can update the State reference in the context.
  - Related to: Flyweight objects are often used for sharing State objects. Strategy looks structurally similar, although strategies do not change their context method, nor are they typically used as dynamically as States.

Strategy

- Intent: “Define a family of algorithms that can be easily interchanged with each other”
- Motivation: Support the open/closed principle by abstracting algorithms behind an interface; clients use the interface while subclasses provide the functionality and can be easily interchanged.
- Applicability:
  - When you want to be able to replace a behaviour at runtime (strategy reference can be dynamically altered).
  - When you want to have a family of behaviours that might not be applicable for the client class.

Strategy


Client
Context
- strategy
Interface
+algorithm()
ImplementationOne
+algorithm()
ImplementationTwo
+algorithm()

Strategy

- Participants:
  - Context contains reference to chosen strategy and invokes algorithm.
  - Strategy interface declares algorithm structure.
  - ConcreteStrategy implements algorithm. Context does not use any methods not defined in the Strategy interface.
- Consequences:
  - Context uses the interface, not the concrete class.
  - Concrete classes can be easily exchanged.

Strategy

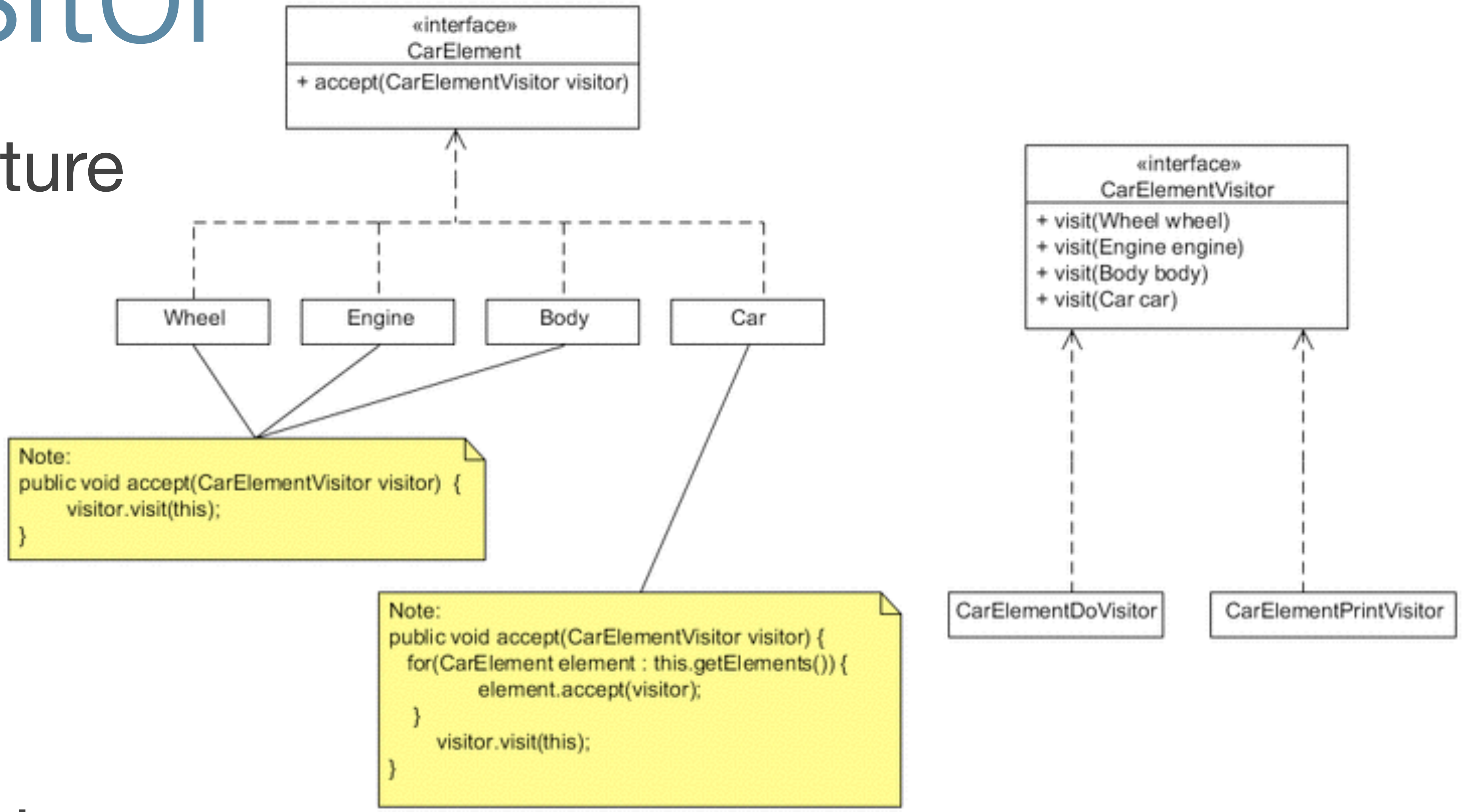
- Implementation:
  - 1) Create strategy interface and reference in client.
  - 2) Interact only through interface.
  - Related to: Abstract Factory uses inheritance to achieve a subset of what Strategy does though a composition-based behaviour. Decorators are similar but focus on extending the functionality of a base class.

Visitor

- Intent: “Represent operations to be performed on classes of elements.”
- Motivation: Consider a large tree of objects that you want to perform an analysis on; this could require changing many objects. Visitors enable these
- Applicability:
  - When you have a large object structure you want to traverse.
  - When you have many different operations you want to perform but don’t want to pollute the objects.
  - The objects rarely change but the operations may.

Visitor

- Structure



```
public void accept(CarElementVisitor visitor)  {
        visitor.visit(this);
}
```

Note:

```
public void accept(CarElementVisitor visitor) {
   for(CarElement element : this.getElements()) {
            element.accept(visitor);
   }
      visitor.visit(this);
}
```

- Participants:
    - Visitor / ConcreteVisitor
    - Element / ConcreteElement
    - ObjectStructure

Visitor

- Collaborations:
  - Client creates the ConcreteVisitor that traverses the object structure.
  - The visited object calls its corresponding visitor method on the ConcreteVisitor.
- Consequences:
  - Adding new operations is easy.
  - Visitor gathers related operations (and separates unrelated ones).
  - Adding ConcreteElement classes is hard.
  - Accumulating state.
  - Negative: Breaking encapsulation. (visitor may need access to internal state)

Visitor

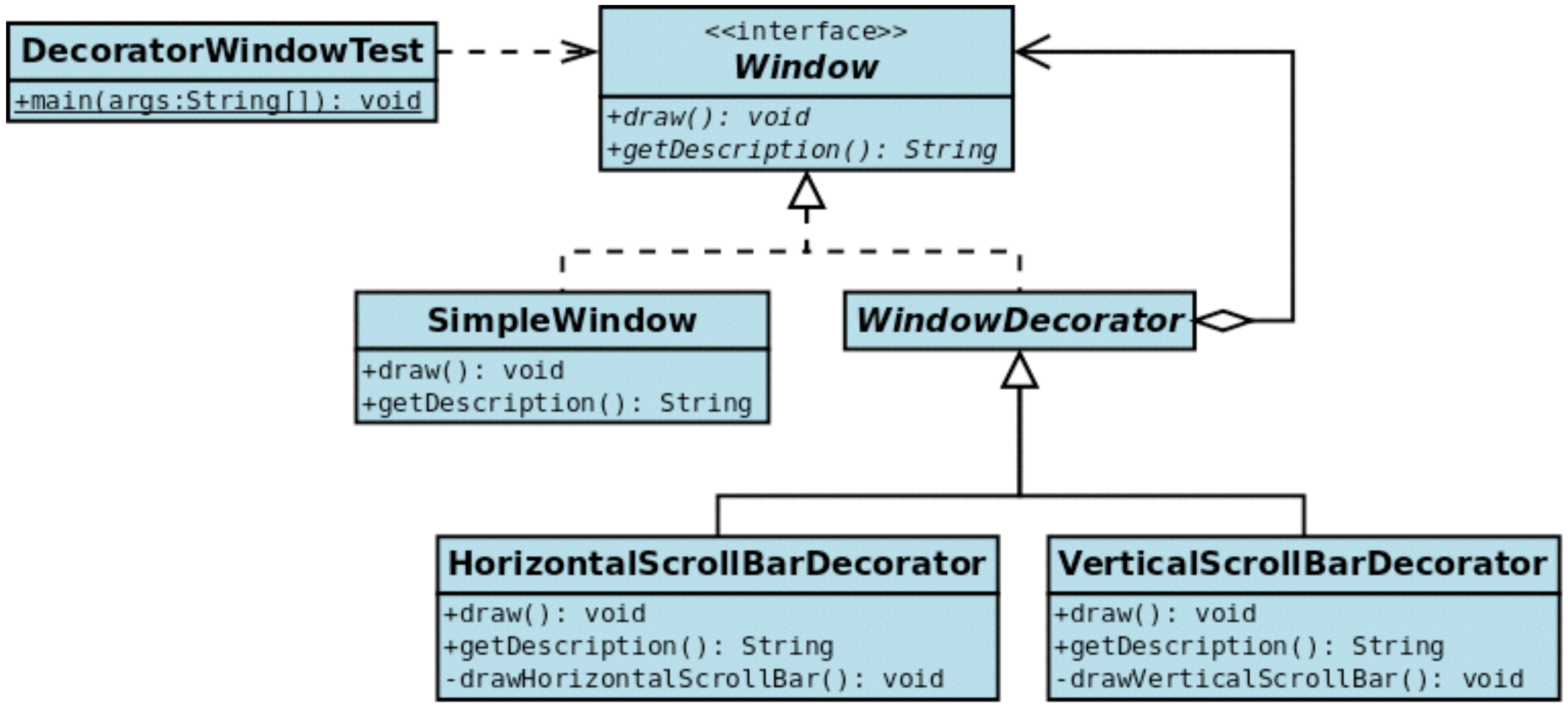
- Implementation:
  - 1) Double dispatch.
  - 2) Who traverses structure?
- Related to: Good at visiting Composite structures.

Decorator

- Intent: "Dynamically add additional responsibilities to structures."
- Motivation: Sometimes we want to add new responsibilities to individual objects, not the whole class. Can enclose existing objects with another object.
- Applicability:
  - Add responsibilities dynamically and transparently.
  - Remove responsibilities dynamically.
  - When subclassing is impractical.

Decorator

- Structure


DecoratorWindowTest
+main(args:String[]): void
<<interface>>
Window
+draw(): void
+getDescription(): String
SimpleWindow
+draw(): void
+getDescription(): String
WindowDecorator
HorizontalScrollBarDecorator
+draw(): void
+getDescription(): String
-drawHorizontalScrollBar(): void
VerticalScrollBarDecorator
+draw(): void
+getDescription(): String
-drawVerticalScrollBar(): void

- Participants:
  - Component / concrete component
  - Decorator / concrete decorator

Decorator (code ex)

```
// the Window interface
interface Window {
    public void draw(); // draws the Window
    public String getDescription();
}

// implementation of a simple Window
class SimpleWindow implements Window {
    public void draw() {
        // draw window
    }

    public String getDescription() {
        return "simple window";
    }
}
// abstract decorator class
abstract class WindowDecorator implements Window {
    protected Window decoratedWindow;

    public WindowDecorator (Window decoratedWindow) {
        this.decoratedWindow = decoratedWindow;
    }
   public void draw() {
        decoratedWindow.draw();
    }
}
```

```
// adds vertical scrollbar functionality
class VerticalScrollBarDecorator extends WindowDecorator {
    public VerticalScrollBarDecorator (Window decoratedWindow) {
        super(decoratedWindow);
    }
    public void draw() {
        drawVerticalScrollBar();
        super.draw();
    }
    private void drawVerticalScrollBar() { .. }
    public String getDescription() {
        return decoratedWindow.getDescription() +" and vert sb";
    }
}
// adds horizontal scrollbar functionality
class HorizontalScrollBarDecorator extends WindowDecorator {
    public HorizontalScrollBarDecorator (Window decoratedWindow) {
        super(decoratedWindow);
    }
    public void draw() {
        drawHorizontalScrollBar();
        super.draw();
    }
    private void drawHorizontalScrollBar() { .. }
    public String getDescription() {
        return decoratedWindow.getDescription() + "and horiz sb";
    }
}
```

```
 public class DecoratedWindowTest {
     public static void main(String[] args) {
         Window decoratedWindow = new HorizontalScrollBarDecorator (
                 new VerticalScrollBarDecorator(new SimpleWindow()));
         // print the Window's description
         System.out.println(decoratedWindow.getDescription());}}
```

Decorator

- Collaborations
  - Decorators forward requests to component object.
- Consequences:
  - More flexible.
    - (than static inheritance; arbitrary nesting possible)
  - Avoids feature-laden classes.
    - (KISS and add functionality as needed.)
  - Warn: Decorator & component are not identical.
    - (equality can be thrown off because decorator != decorated)
  - Negative: Many of little objects.
    - (Lots of small, similar-looking classes differentiated by how they are connected. hard to understand and debug.)

Decorator

- Implementation:
  - 1) Interface conformance. (decorator interface required)
  - 2) Abstract decorator not needed if only one decoration is required.
  - 2) Keep component classes lightweight. (too heavyweight can overwhelm decorators
  - 3) Changing a skin instead of changing the guts. (if component is heavy, consider strategy instead)
- Related to: Decorators are a kind of single-node Composite. Decorators can change the skin, Strategy pattern can change the guts.

Decorator Example

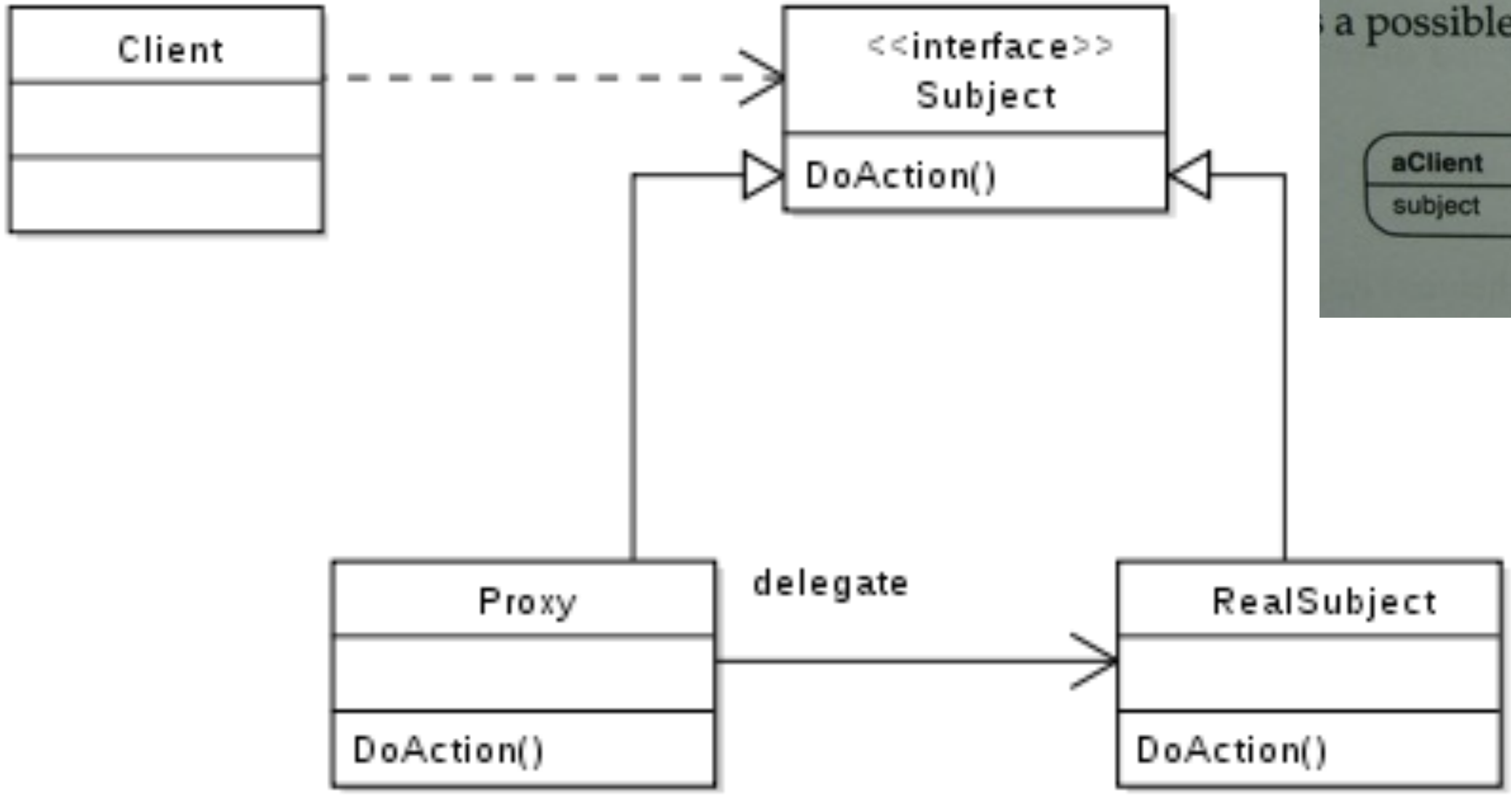
- Starbucks example
  - Beverages: house, dark, decaf, espresso
  - Toppings: whip, milk, soy, mocha,...
- Show Crazy class diagram (state explosion)
  - -> sucky to update
- KEY: classes open for extension but closed for modification
- KEY: decorators mirror type of their decorating obj
- BAD: lots of little objects floating around

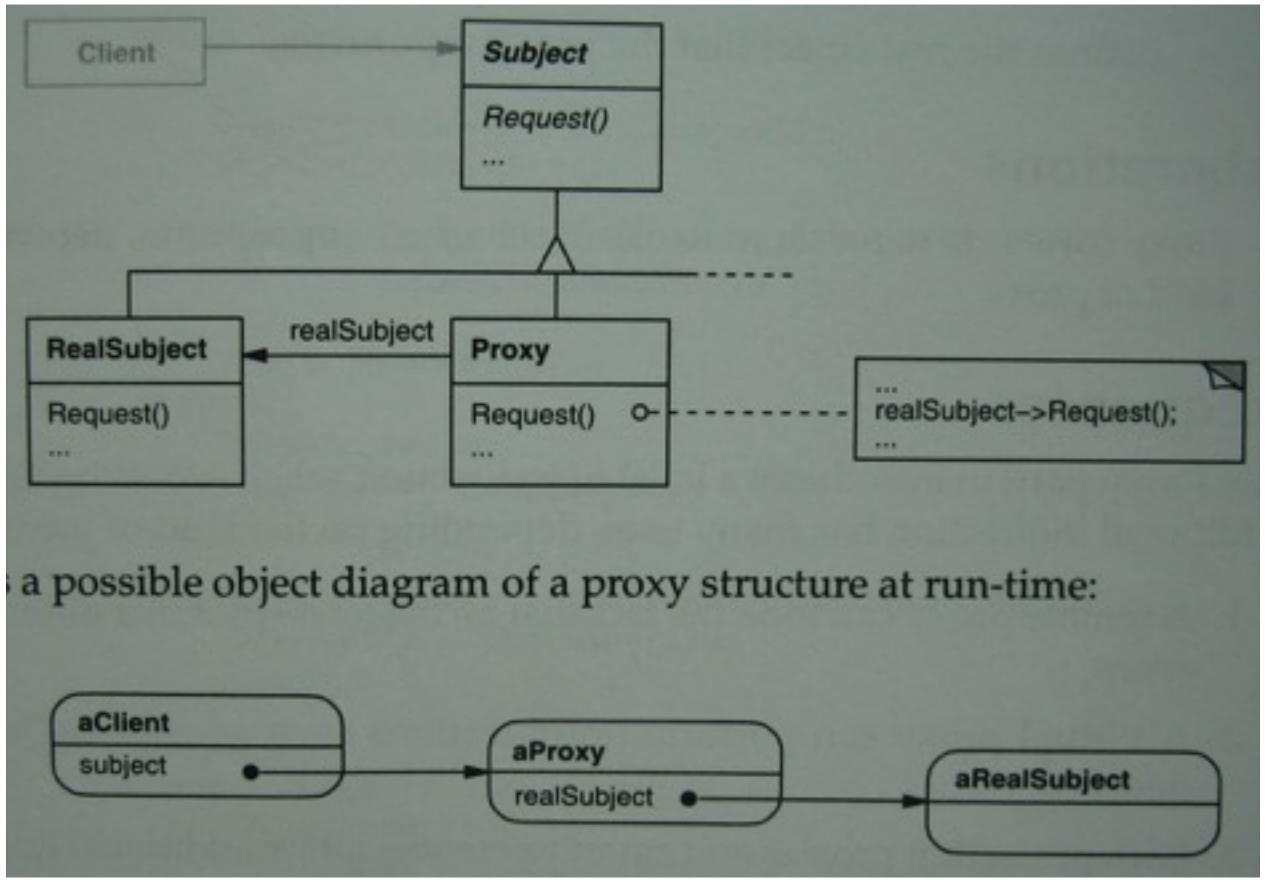
Proxy

- Intent: "Provide a placeholder to control access to another object."
- Motivation: One reason to control access is cost: consider an object that is expensive to populate entirely but cheap to partially populate. (e.g., remote object, large file from disk, etc.)
- Applicability: (When a more versatile reference is needed.)
  - **Remote**: Hide the fact that an object is remote.
  - **Virtual**: Create expensive objects on demand.
  - **Protection**: Protect access to objects.
  - **Smart reference**: Performs additional actions.

Proxy

- Structure





- Participants:
  - Proxy
  - Subject / RealSubject

Proxy

- Collaborations:
  - Client interacts with proxy.
  - Proxy forwards req. to RealSubject as required.
- Consequences:
  - **Remote**: location hidden.
  - **Virtual**: Optimizations can be applied.
  - **Protection**: Housekeeping / auth can occur.

Proxy

- Implementation:
  - 1) Create common interface for Proxy/RealSubject.
  - 2) Reference Proxy in Client.
  - 3) Proxy forwards requests as necessary.
- Known uses: Image manipulation / RPC.
- Related to: Similar to Adapter and Decorator. Decorators add responsibilities while proxies serve as mediators.