Root cause analysis of some faults in Design Patterns

Ralph Johnson
University of Illinois at Urbana-Champaign
rjohnson.uiuc@gmail.com

Design Patterns: Elements of Reusable Object-Oriented Design by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, 1995.

Released in October 1994 at OOPSLA.

Jolt Award, Dr. Dobbs Journal award, SIGPLAN Programming Languages award, Aito Dahl-Nygard Prize, SIGSOFT Software Engineering award

- The Design Patterns Smalltalk Companion, Alpert, Brown and Woolf
- Heads First Design Patterns, Freeman, Freeman, Bates and Sierra.
- Domain Driven Design, Eric Evans
- Patterns of Enterprise Application Architecture, Fowler et. al.

- We could have done better.
- But we did pretty well!

Why Design Patterns gives bad advice

- Times have changed
  - There are better patterns now
  - We know how to describe patterns better now
- Should have known better in 1994

Categories of Design Patterns

- Structural
- Behavioral
- Creational

- Core
- Peripheral
- Compound
- Creational

Creational Patterns

- Factory Method
- *Factory Object*
  - Abstract Factory
  - Builder
  - Prototype
- Singleton

Singleton

- What if you want to make sure that a class has only one instance?

- One possibility is global variables. Another is using static member functions.

- Best solution: store single instance in static member variable.

Singleton in Java

```
public class Singleton
{
    private static Singleton soleInstance = null;

    protected Singleton() {}

    public static Singleton getInstance() {
        if (soleInstance == null)
            soleInstance = new SIngleton();
        return soleInstance;
};
```

Real problem with Singleton

- Singleton encapsulates global state
  - Increases coupling
  - Makes testing harder
  - Makes components less reusable

Alternatives to Singleton

- If object is not a singleton, how do you refer to it?
  - Client object has a reference to it
  - Reference is passed as method argument
- How does client object get reference?
  - Initialized by constructor
  - Factory ensures it is initialized correctly

Core patterns

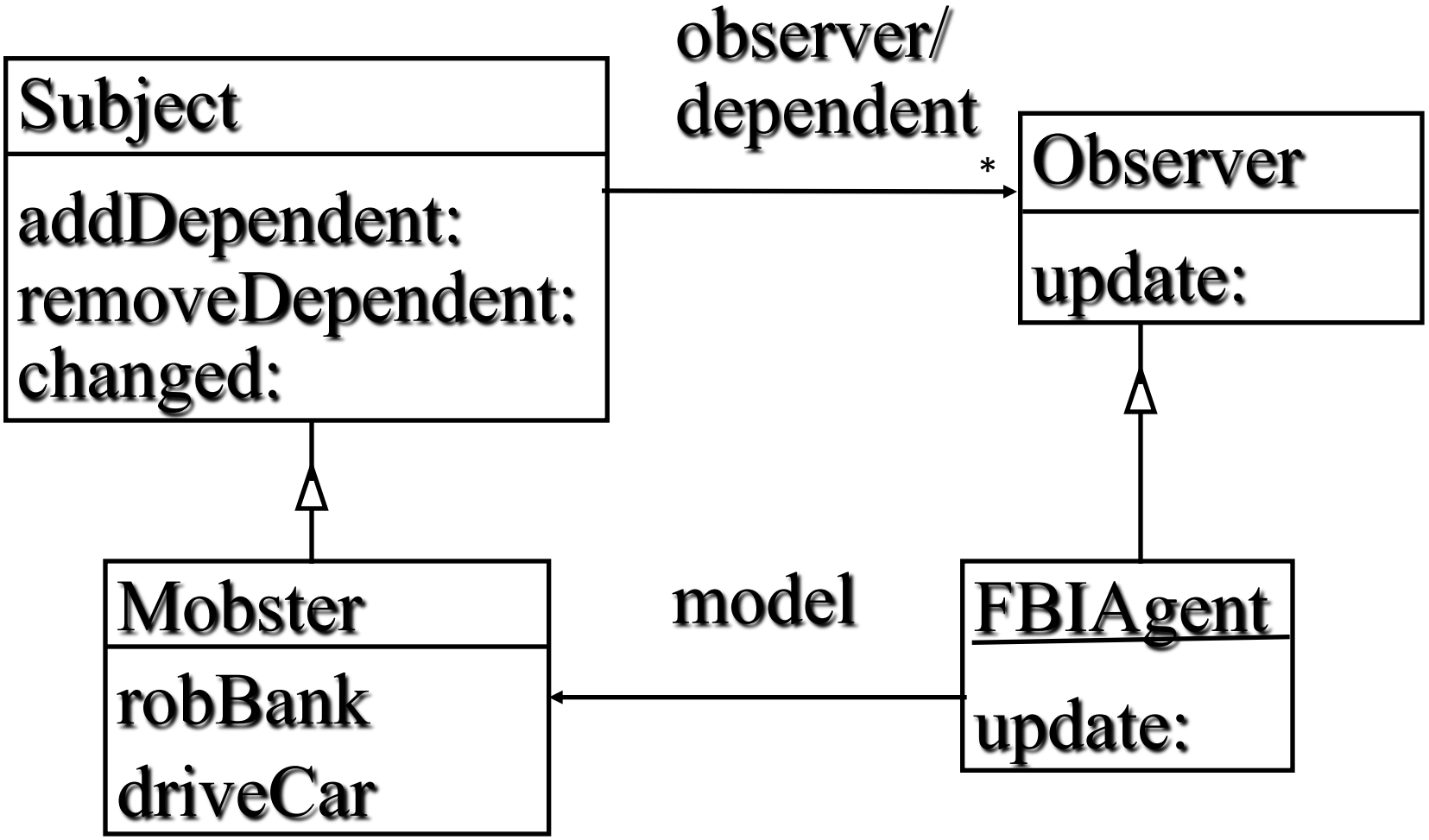
- Composite
- Strategy
- Decorator
- State
- Iterator
- Observer

- Mediator
- Façade
- Proxy
- Command
- Template Method
- Adapter

Observer

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Observer Pattern



Observer Pattern:

- Registration
  - *subject addDependent: observer*
- Notification
  - *self changed. self changed: #value*
- Update
  - define *update: aSymbol*

The book says …

```
class Subject;
class Observer {
public:
    virtual ~Observer();
    virtual void Update(Subject* theChangedSubject) = 0;
protected:
    Observer();
};
```

Listening instead of Observing

- Argument to “update” is an Event
- Many kinds of EventListeners, each with their own interface
- An EventListener interface can have more than one update method

Different Kinds of Listeners

- ActionListener
  - actionPerformed(ActionEvent)
- ComponentListener
  - componentResized(ComponentEvent)
  - componentMoved(ComponentEvent)
  - componentShown(ComponentEvent)
  - componentHidden(ComponentEvent)
- PropertyChangeListener
  - propertyChange(PropertyChangeEvent)

From the book …

Avoiding observer-specific update protocols: the push and pull models. Implementations of the Observer pattern often have the subject broadcast additional information about the change. The subject passes this information as an argument to Update. The amount of information may vary widely.

…

The pull model emphasizes the subject's ignorance of its observers, whereas the push model assumes subjects know something about their observers' needs. The push model might make observers less reusable, because Subject classes make assumptions about Observer classes that might not always be true.

Reality …

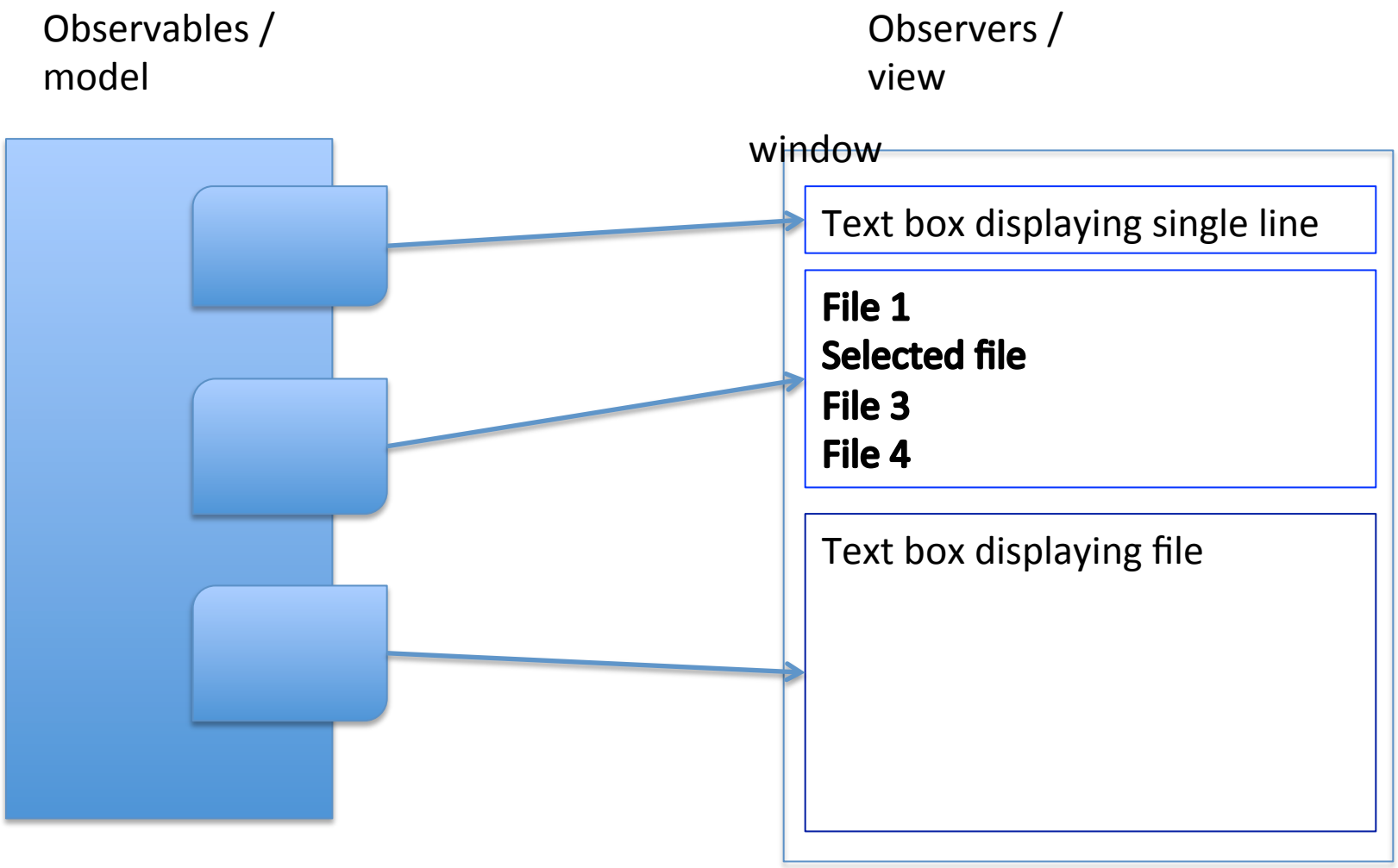
- Use the push model to make observers *more* reusable, by having specialized observer interfaces for particular kinds of changes.

- Example: observe a string or a number

- Result: a library of both subjects and observers

Data Binding

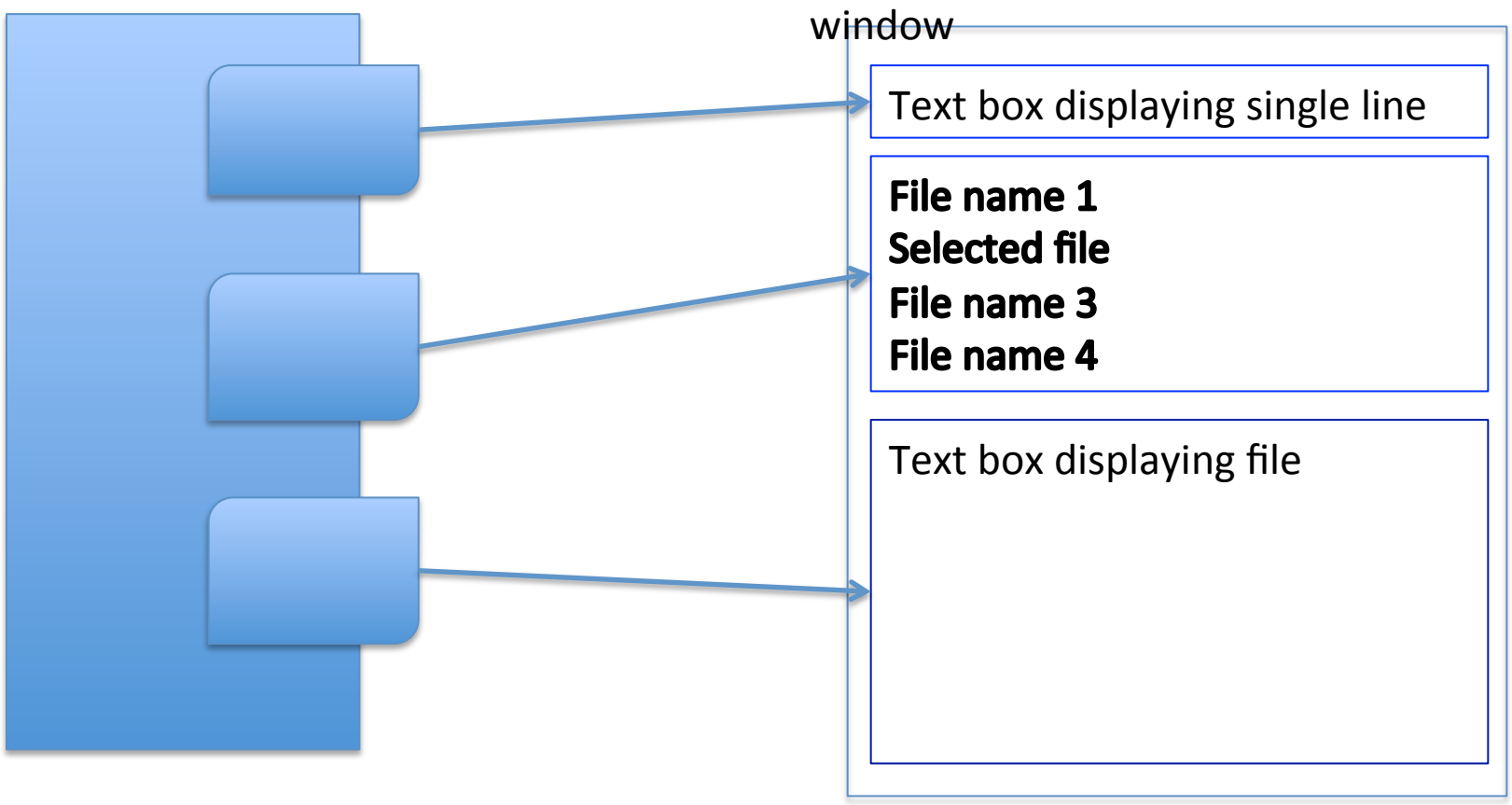
- Implement the model as a composite model.
- Primitives are properties, lists, etc.
- Model does not “assign to variable”, but “sets value of property”.


Observables /
model
Observers /
view
window
Text box displaying single line
File 1
Selected file
File 3
File 4
Text box displaying file

Consequences of “data binding”

- Don’t have to invoke update events explicitly
- Observers can be reused
- Main benefit comes once there is a library of reusable observers and observables.
- Requires a set of standard Observer/Listener interfaces.

When a new file name is selected, the file is read and its text is displayed in the lower text box.



Use of Mediator with Observer

- Reusable observers might handle 95% of events. What about the other 5%?
- Have a mediator handle all the “special” events.
- Typically the mediator is either the top-most model or a high-level view.

The book says …

Related patterns: Mediator

By encapsulating complex updates, the ChangeManager acts as a mediator between subjects and observers.

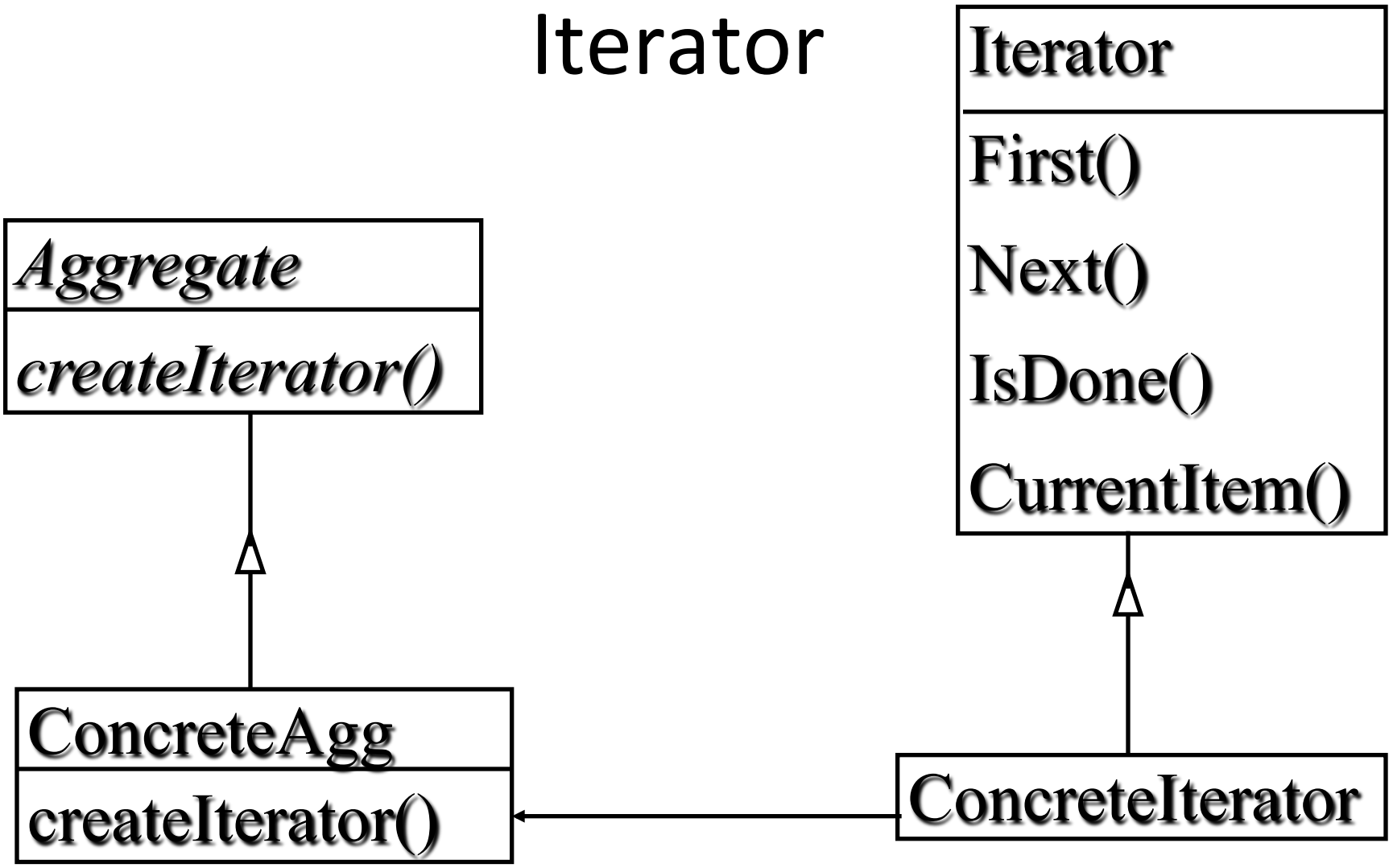
Use of Adapter with Observer

Java: user inner class to make a listener that listens to a particular event source.

Two buttons -> two listeners -> two inner classes

```
redButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        performRedAction();
    }
}
```

Iterator


Iterator
First()
Next()
IsDone()
CurrentItem()
Aggregate
createIterator()
ConcreteAgg
createIterator()
ConcreteIterator

Java

Iterator

- next()
- hasNext()
- remove()

Reader

- read(char cbuf[]) returns number read or -1

Java 8 Streams

External vs. Internal Iterator

Smalltalk: iterate over a collection without making a separate object by using an internal iterator.

```
'This is a string' do: [:c | c isVowel ifTrue:
    [c printOn: Transcript]]
```

An external iterator

```
aStream := ReadStream on: 'This is a string'.
[aStream atEnd] whileFalse: [c := aStream next.
c isVowel ifTrue: [c printOn: Transcript]]
```

Smalltalk Collection protocol

do:	iterate: apply closure to each elem
select:	return elements that match
reject:	return elements that don't match
collect:	return new collection of elements that are result of applying closure
inject:into:	reduce elements to a single element

Iterator in Groovy

```
[1,2,3,4,5,6,7].each {it -> println “$it,   ${it*it}”}
[1,2,3,4,5,6,7].collect {it -> it*it }
[1,2,3,4,5,6,7].findAll {it -> it <= 5}
[1,2,3,4,5,6,7].reduce(0) {sum, it -> sum+it}
```

You can define these methods on streams, not just on collections.

Iterator in JavaScript

- [1,4,9].forEach(x => consol.log(x))
- [1,4,9].map(x => x*x)
- [1,4,9].filter(x => x > 3)
- [1,4,9].reduce((sum,x => sum + x), 0)

- “EventStream has all the Array functions because an EventStream is like an Array, only laid out in time instead of memory.”

Streams in Java 8

forEach

map

filter

reduce

collect

max

skip

…

What the book did not say …

- A Stream is a kind of iterator
- You *can* define map, filter and reduce on your iterators. If your language has closures, you *should*.

Why collection operations?

- Makes programming simpler and easier
  - Eliminates off-by-one errors
  - Programs are shorter
  - Looks more like straight-line code
- Easier to parallelize
  - SIMD model

Reactive programming: Iterator meets Observer

- Think of a “subject” as an EventStream. Observer is a function on EventStreams.

- Eliminates need to keep state to synchronize different event streams
- Makes error handling easier
- Reduces race conditions, memory leaks

Asynchronous streams

- concatAll – takes a list of lists of T and returns a single list of T by concatenating them
- takeUntil(stopStream) will keep returning elements from the receiver until it gets an element from the stopStream

Mouse Drags Collection

```
Var getElementDrags = elmt =>
    elmt.mouseDowns.
        map(mouseDown =>
            document.mouseMoves.
                takeUntil(document.mouseUps)).
        concatAll();
getElementDrags(image).forEach(pos =>
image.position = pos)
```

- For more information, search for “reactive extensions”
- Libraries for many languages.
- Originated at Microsoft
- Netflix Javascript talks – Asynch JavaScript with Reactive Extensions
- https://www.youtube.com/watch?v=XRYN2xt11Ek

How Observer should change

- Listener
  - pass events as argument of Update
  - Multiple observer interfaces, each specialized
- Data binding (observer class library)
- Use of mediator, adapter
- Simplify complex observer logic by treating event streams as iterators and using collection operators like map and filter

Changes for Iterator

- Other operations on iterator
  - map, filter, reduce
- Streams are iterators

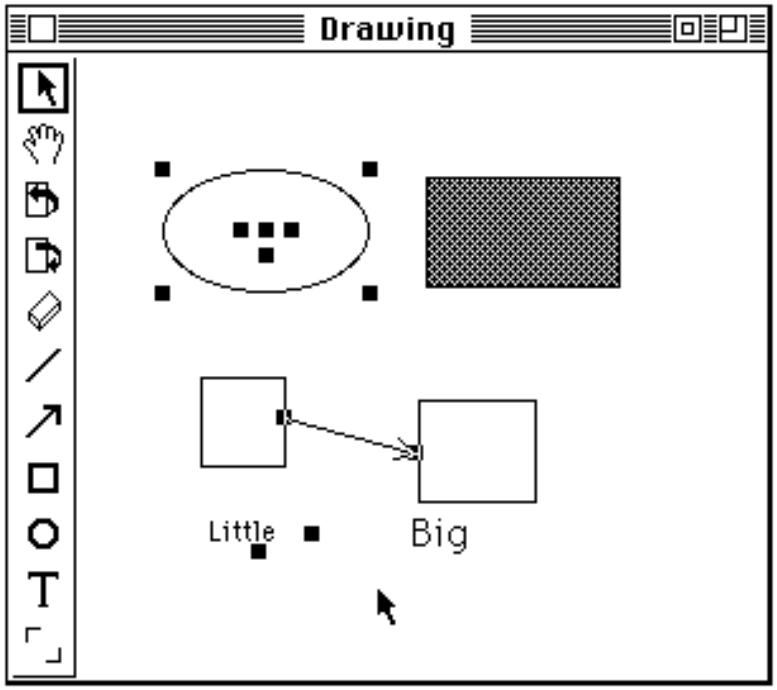
Some faults

- Too much emphasis on patterns as end-points instead of steps in the design process
- Conformity to patterns is not measure of goodness

Factory Method

- Don't call constructor directly.
- Make a separate method to create object.
- Advantages:
  - can change class of product in subclass
  - can produce easier to read functions
- Disadvantages:
  - more methods
  - parallel class hierarchies

HotDraw example



Factory Object

- Problem with factory method -- have to create subclass to parameterize.
- Often end up with parallel class hierarchies.
- Example: subclass of Tool for each figure you want to create
- Alternative: parameterize CreationTool with object that creates figure
- (Note: Factory Object is generalization of Abstract Factory,Builder, and Prototype. It is not in the book.)

Applicability

- Use factory objects:
  - when system creates them automatically
  - when more than one class needs to have product specified
  - when most subclasses only specialize to override factory method

Prototype

Making a class hierarchy of factories seems wasteful.

The parameters of an object can be as important as its class.

Solution:

- Use any object as a factory by copying it to make a new instance.

Advantages

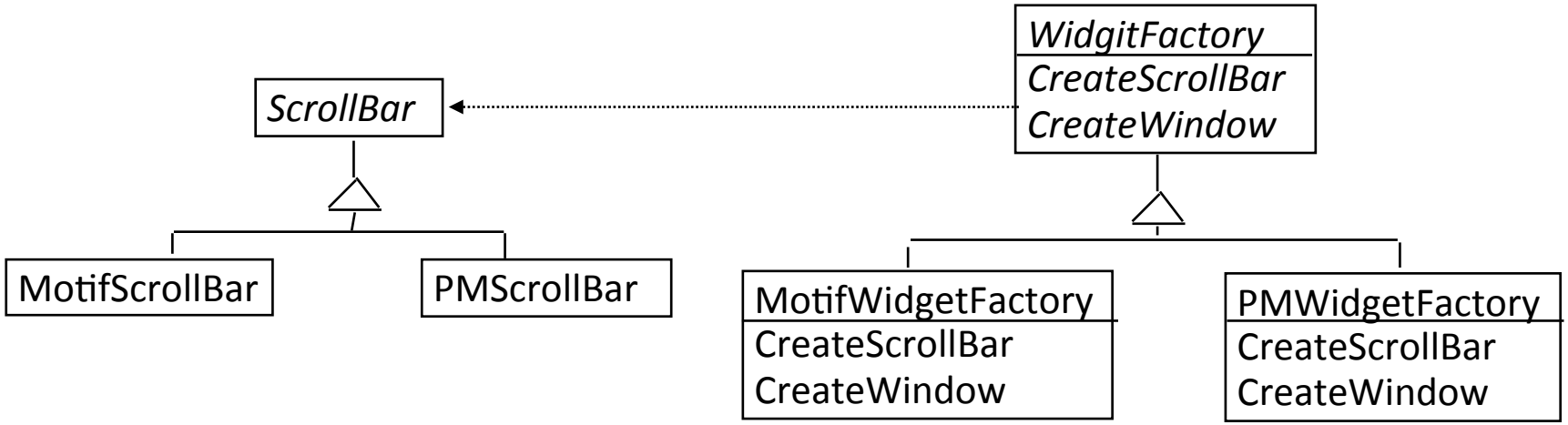
- Don't need new factory hierarchy.
- Can make new "class" by parameterizing an object

Disadvantages

- Requires robust copying

Abstract Factory

- Sometimes a group of products are related -- if you change one, you might need to change them all.
- Producer should be decoupled from products
- Solution:
  - Make a single object that can make any of the products.



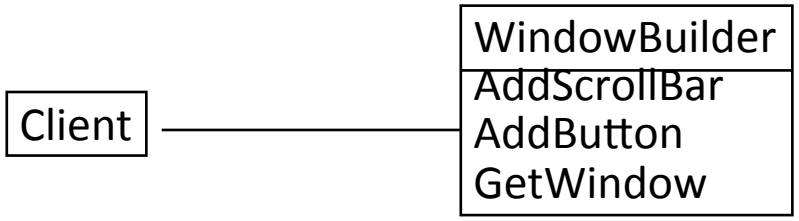
How to create Abstract Factory

- Start with a class/module that creates product
  - Lots of calls to constructors
- Convert calls to constructors to factory methods
- Create a factory class, move f.m. into it.
- Producer is parameterized by factory object
- Now producer calls methods on factory instead of methods on itself

Builder

- Complex objects require a lot of work to make.

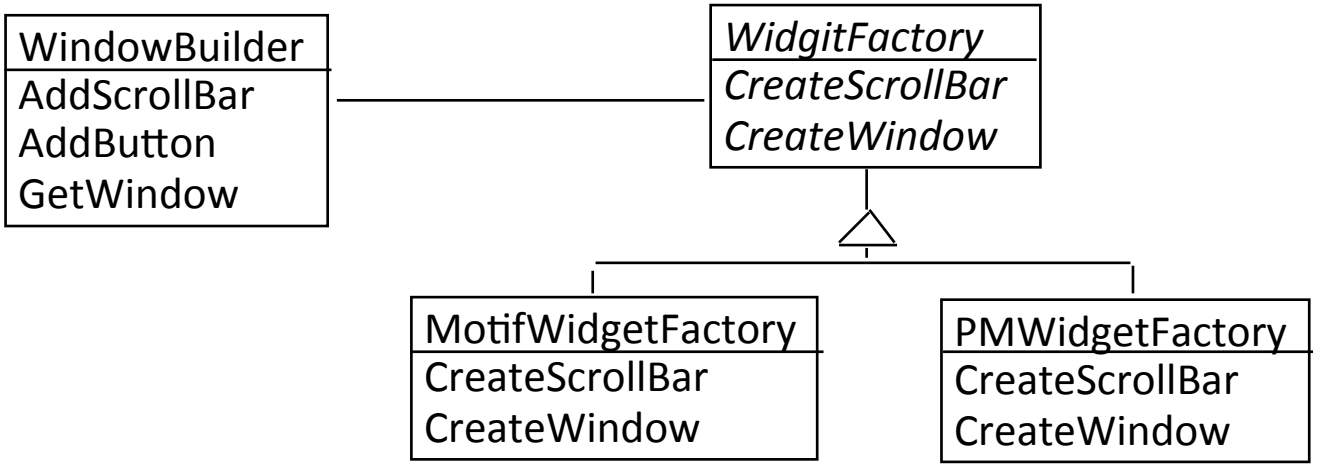
- Solution:
  - Factory keeps track of partly built product.
  - Client specifies product by performing series of operations on factory.
  - Client asks for product when it is finished



Implementing Builder

Builder can make components using

- Factory Method
- Abstract Factory, or
- Prototype



Summary of Factory Patterns

- Factory method – make it easy to change class of product
- Abstract factory -- use when there is a set of related products
- Builder -- use when product is complex
- Prototype -- use when Factory Method is awkward and when classes are not objects, or when you want to specify new "classes" by composition

Dependency injection

- Remove dependencies between classes.
  - References should be to interfaces.
  - Use factories to produce other objects.
  - Dependencies should be “injected” by the constructor
  - Concrete classes are determined by a “script” that creates instances and makes connections between them

What does this mean?

- Classes should only import interfaces, never concrete classes?
  - What about Date or ArrayList?
- Classes should never call constructors unless they are a factory?
  - What if you never want to change the class being constructed?

Dependency injection & modules

- Modules should be units of
  - Reuse
  - Testing
  - Release
  - …
- Minimize dependence between modules
- Dependence inside a module is not so bad

Modules

- Some classes are meant to be used together
  - Abstract classes that make up a framework
  - Strategies depend on their context
  - States depend on their context
  - Façade hides particular classes inside module
  - Concrete subclasses of Abstract Factory depend on concrete classes of products

Modules

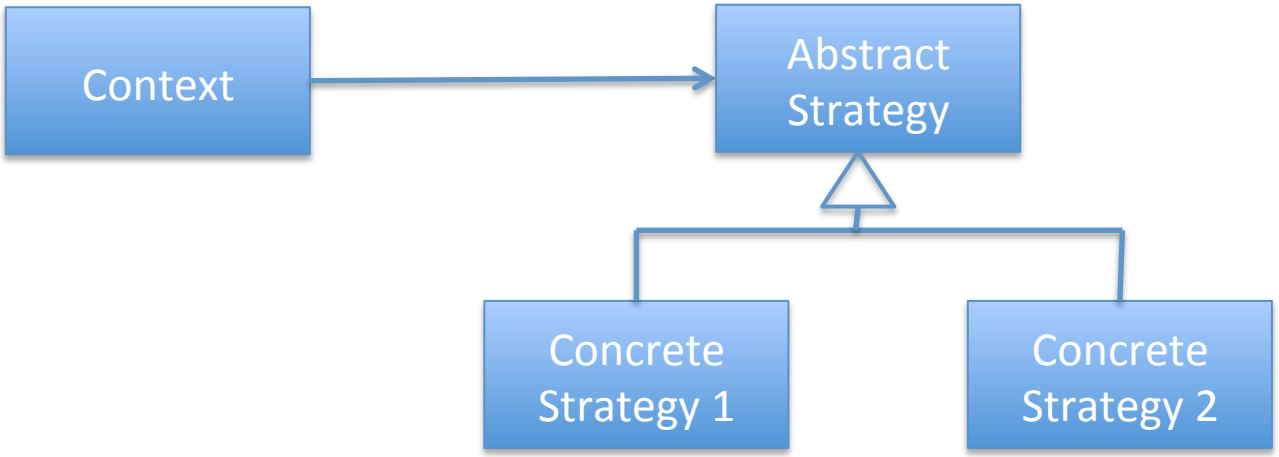
- A module is an example of a “bounded context” from Eric Evan’s *Domain Driven Design.*
- To understand how modules are related to the organization of your developers, read his patterns about relationships between bounded contexts.

Creational

- Abstract factory
- Factory method
- Prototype
- Builder
- Singleton
- *Dependency Injection*

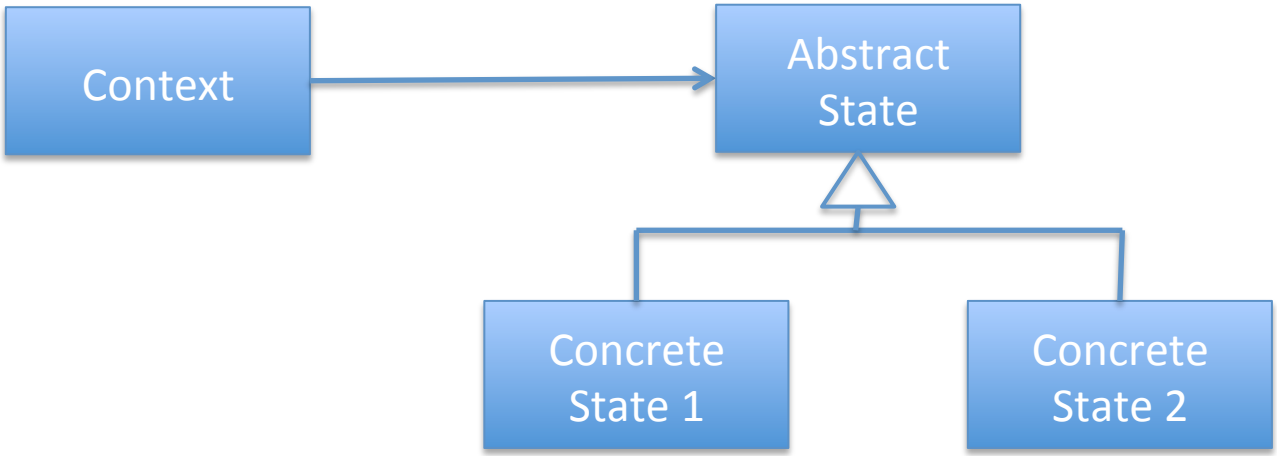
Strategy

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.



State

- Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.



State and Strategy

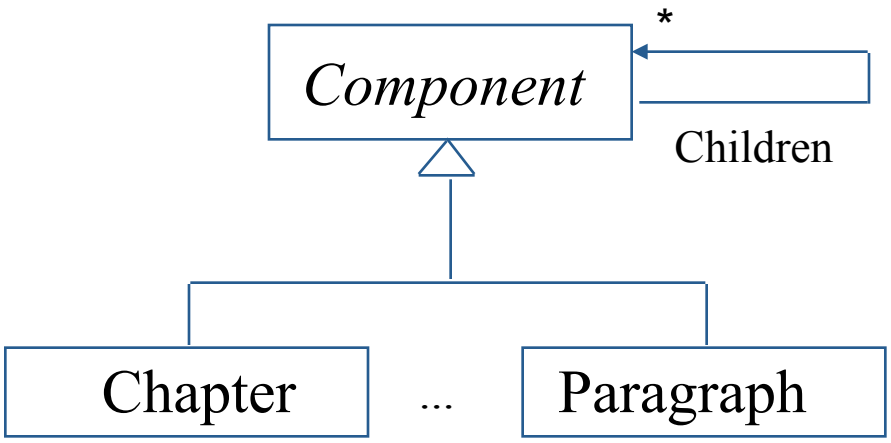
- State
  - One method for each kind of event
  - Methods don't call each other
  - State has no state
  - State changes state
- Strategy
  - One public method
  - Lots of private methods
  - State of the algorithm
  - User of context chooses strategy
- Strategy can be replaced by closure. State can't.

State and Strategy

- Pattern is more than structure diagram
- Many variations of patterns
- Goal is to make a good design, not to make it like the book
- Even when design is different from the book, it is useful to compare with “standard”.

Composite

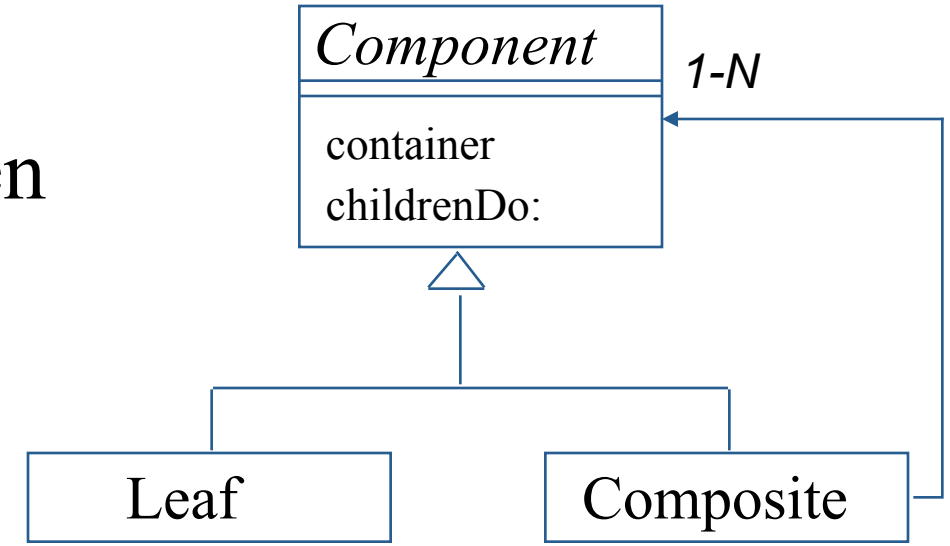
- Idea: make abstract "component" class.
- Alternative 1: every component has a (possibly empty) set of components.
- Problem: many components have no components.



Composite Pattern

Composite and Component have the exact same interface.

- enumerating children
- childrenDo: for Component does nothing
- only Composite adds removes children.



Design Patterns says:

An important issue in the Composite pattern is which classes declare (add and remove)

Defining the child management in the Composite gives you safety because any attempt to add or remove objects from leaves will be caught at compile-time in a statically typed language like C++. But you lose transparency, because leaves and composites have different interfaces.

The only way to provide transparency is to define default Add and Remove operations in Component

Current best solution

- Put Add and Remove in Composite
- Use type-safe down-cast when you want to add or remove a component from a Component.

Summary

New patterns compete with old ones and change how old ones are used.

Issues such as concurrency can change how patterns are implemented and used.

Patterns are not always end-points of refactoring, but can also be intermediate steps.

The point is not patterns, but good design. Patterns can help us make/learn design.

- rjohnson.uiuc@gmail.com