Design Patterns

James Brucker

Reusable Ideas

Developers **reuse** knowledge, experience, & code

Application Level

reuse a project design & code of a similar project

Design Level

apply known design principles and design patterns

Logic Level

apply *algorithms* to implement some behavior

Code Statement Level

use programming idioms for common tasks

A Programming Idiom

Task: process every element of an array...

Idiom:

1. initialize result
2. loop over the array
3. process each element of the array

```
// add the values of all the coupons we have sold...
Coupon [ ] coupons = ...
double total = 0;                          // initialize
for( int k=0; k<coupons.length; k++ ) {  // loop
    total += coupons[k].getTotal();  // process each one
}
```

An Algorithm

Task:

find the shortest path from node A to node B in a graph

Algorithm:

apply Dykstra's Shortest Path algorithm

Reusable Code

Requirement:

record activity & events in a file,
so we have a record of what the program has done
and any problems that occur.

Solution:

Use the open-source Log4J framework.

```
public class Purse {
   private static final Logger log =
            Logger.getLogger(Purse.class);
   public boolean insert(Money m) {
      if (m == null) log.error("argument is null");
      else log.info("inserting " + m);
```

Logger Output

Log File:

You control *where logging is output*, and *how much detail is recorded*. Config file: log4j.properties.

Example:

```
6:02:27 Purse insert INFO inserting 10 Baht
6:03:00 Purse insert INFO inserting 20 Baht
6:03:10 Purse insert ERROR argument is null
6:03:14 Purse withdraw INFO withdraw 10 Baht
```

Class and Method

message

Severity

Reusable Applications

Requirement:

Write a web application to manage appointments at a beauty shop.

Solution:

Modify the SpringFramework "JPetStore" sample application.

AppFuse has sample applications for many frameworks that you can use to start your project.

The sample apps have good design and use *best practices* from experienced developers.

What is a Design Pattern?

A *situation* that occurs over and over,

along with *forces* that motivate design of a *solution*,

leading to...

a *reusable pattern* for the design of a solution to similar problems.

Format for Describing a Pattern

Pattern Name: **Iterator**

Context

We need to access elements of a collection.

Motivation (Forces)

We want to access elements of a collection without the need to know the underlying structure of the collection.

Solution

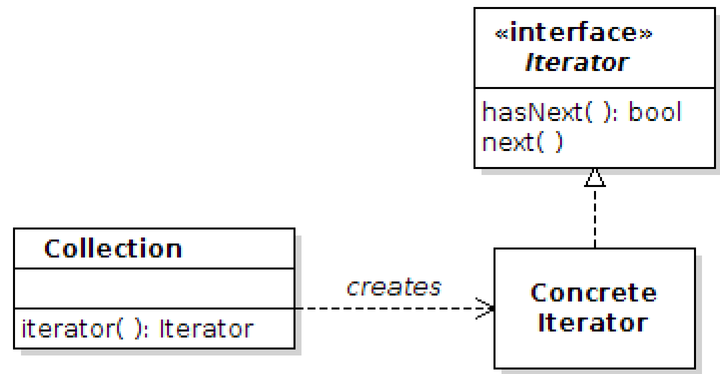
Each collection provides an iterator with methods to get the next element without exposing *how* it is done.

Consequences

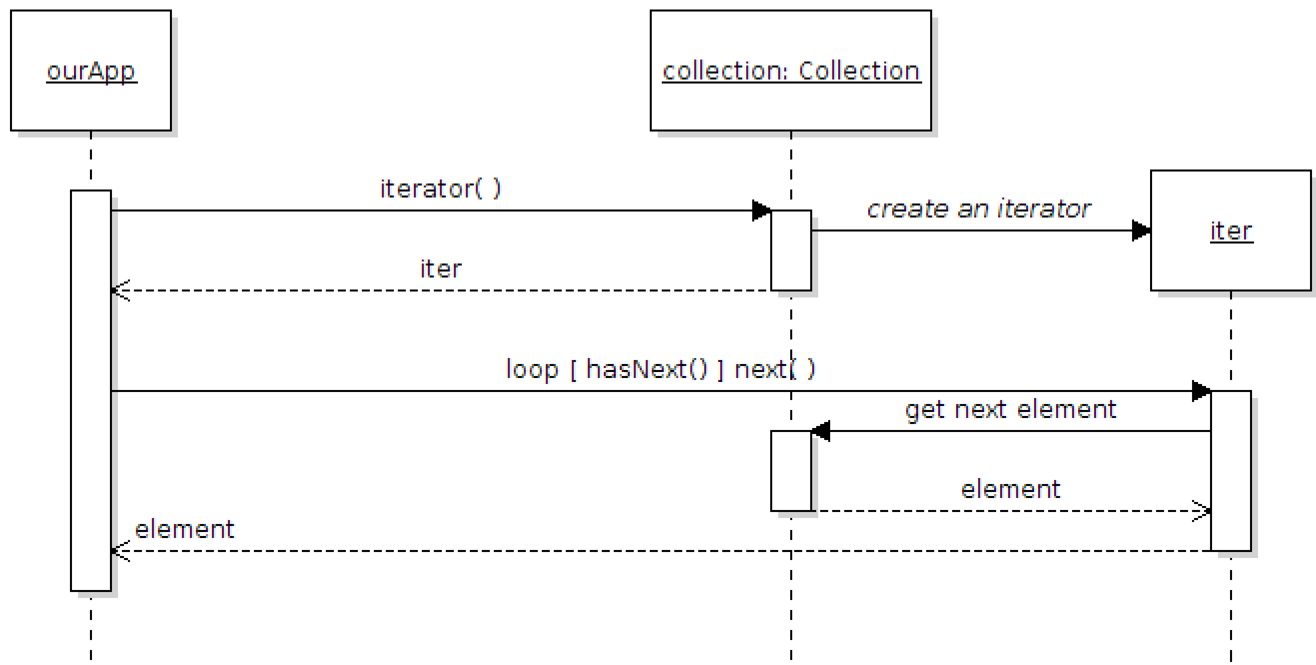
Application is not coupled to the collection. Collection type can be changed w/o changing the application.

Diagram of Iterator Pattern

Structure:



Behavior:



Example Code

A Purse has a collection of Coin objects.

We want to get the value of each coin in the collection.

```
// get the coins from the Purse
Collection<Coin> coins = purse.getCoins();
Iterator<Coin> iter = coins.iterator( );

while( iter.hasNext( ) ) {
    Coin c = iter.next( );
    sum = sum + c.getValue();
}
```

Getting data from JDBC ResultSet

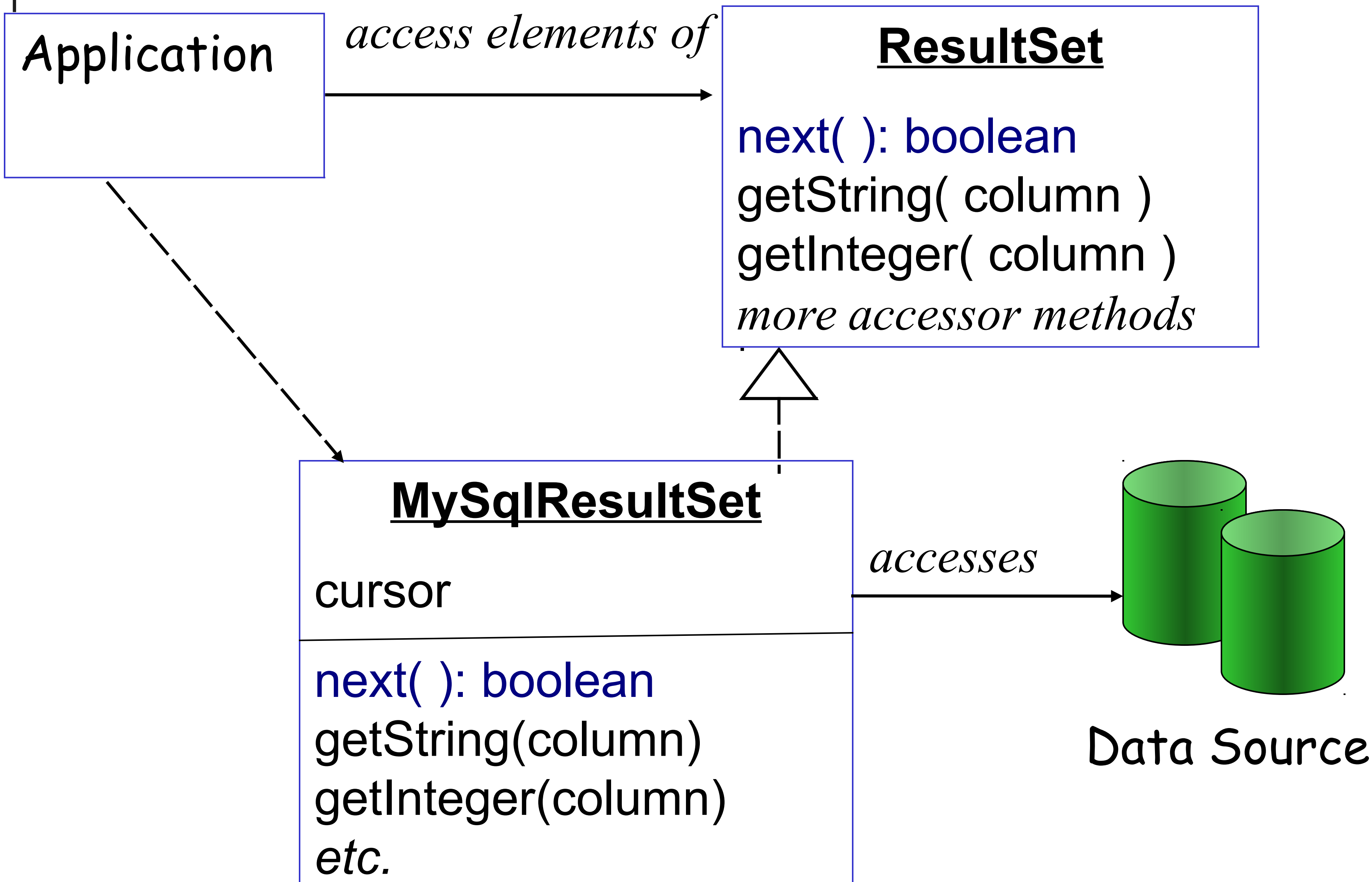
In JDBC, performing a query returns a ResultSet.

```
// connection is a connection to database
// Statement performs an action (command)
Statement stmt = connection.createStatement();

// get all rows from a table named "PEOPLE"
ResultSet rs = stmt.executeQuery(
                  "SELECT * from PEOPLE");

how can we get each row from ResultSet?
```

Iterator for JDBC ResultSet



ResultSet as Iterator

Get the name and birthday of each person in "PEOPLE".

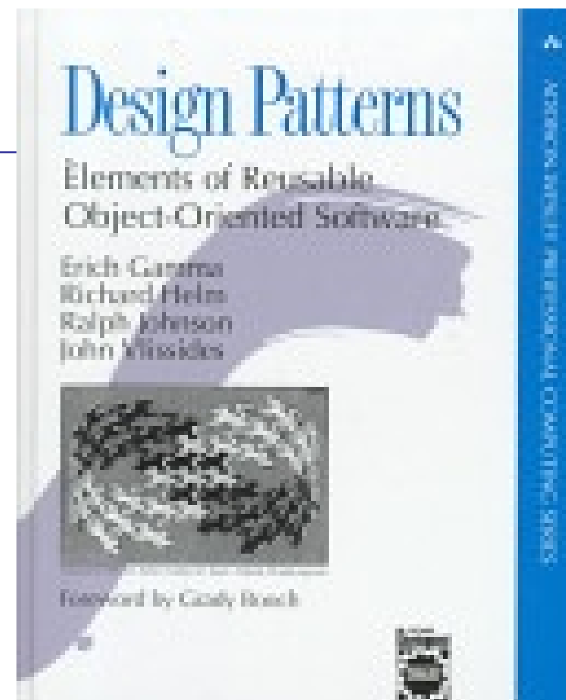
```
// get all rows from a table named "PEOPLE"
ResultSet rs = stmt.executeQuery(
                    "SELECT * from PEOPLE");
// how can we get each row from ResultSet?
rs.first( ); // move to first element
do {
   String name = rs.getString("name");
   Date birthday = rs.getDate("birthday");
   . . .
} while ( rs.next() );
```

Design Patterns - Gang of Four book

The "Gang of Four"

The first book to popularize the idea of software patterns:

Gamma, Helm, Johnson, Vlissides
Design Patterns: Elements of Reusable Object-Oriented Software. (1995)

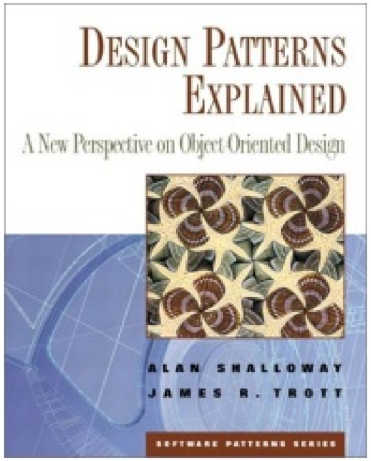


Other Good Design Patterns Books

Good for Java programmers

Design Patterns Explained, 2E (2004)
by Allan Shallow & James Trott

also wrote: *Pattern Oriented Design*.



Head First Design Patterns (2004)
by Eric & Elizabeth Freeman

Visual, memorable examples,
too simple code.

Structure of Patterns in Gang of Four book

Name of Pattern

Intent

what the pattern does.

Motivation

Why use this pattern? When to apply this pattern

Structure

Logical structure of the pattern. UML diagrams.

Participants and Collaborators

What are the elements of the pattern? What do they do?

Consequences

The benefits and disadvantages of using the pattern.

Singleton Pattern

Pattern Name: Singleton Pattern

Context

We want to ensure there is only **one instance of a class**. All parts of the application should share this instance.

Motivation (Forces)

Several objects need to access the same resource, or we want objects to share a resource that is "expensive". Many parts of the program need to access this shared resource.

Solution

Prevent direct instantiation by making the constructor private.

Provide a static accessor method that always returns the same instance of this class (same object).

Singleton Pattern

Single instance of this class

Static accessor for instance

1

Singleton

- instance: Singleton

<<constructor>>

- Singleton( )

+ getInstance( ) : Singleton

+ other methods for the object's behavior

Singleton Pattern

Singleton has 3 elements:

(1) private static attribute that is the only instance of this class

(2) constructor is private to prevent other classes from creating objects

(3) public static accessor returns the single instance of this class.

Singleton

- instance: Singleton

<<constructor>>

- Singleton( )

+ getInstance( ) : Singleton

Example of Singleton Pattern

A Store that has only one instance.

```
public class Store {
// (1) the single static instance
   private static Store theStore = null;
   private List<Transaction> transactions;
// (2) private constructor
   private Store( ) {
      transactions = new ArrayList<Transaction>( );
   }
// (3) static accessor method also creates singleton
public static Store getInstance() {
      if ( theStore == null ) theStore = new Store( );
      return theStore;
   }
```

lazy instantiation

Getting the Singleton object

How do other objects get the Store?

```
// in your application use:


Store store = Store.getInstance( );
```

Lazy Instantiation

Means that you create a resource only when it is needed.

This avoids creating something that may never be used.

```
// (3) static accessor method creates the singleton
public static Store getInstance() {
   if ( theStore == null ) theStore = new Store();
   return theStore;
}
```

The store instance is created the first time that getInstance( ) is called, but not before.

If getInstance is never called, no Store is created.

Lazy Instantiation of Loggers

Using Log4J you will see a lot of code like this:

```
// Create the logger for this class
private static Logger log = Logger.getLogger(...);
```

What if this class never logs any messages?

We wasted time and memory creating the logger.

So many apps use *lazy* instantiation:

```
// Don't create logger yet
private static Logger log = null;

private static Logger getLogger() {
   if (log == null) log = Logger.getLogger(...);
   return log;
}
```

Consequences of Using Singleton

Benefits

- control access to a single instance
- reduce name space pollution - better than using a global variable (in languages with global variables)
- permits a variable number of instances - you can modify the singleton to produce more than one instance, w/o changing other parts of application

Disadvantages

- Singleton cannot be subclassed, since the constructor is private and static getInstance() is not polymorphic.

Related patterns

- Factory Method

What Motivates the Patterns?

Why are the patterns considered "good" design?

What is "good"?

There must be some real-world needs
that result is some **principles** or **criteria**
for **good design**.

Software "Facts of Life"

Name some characteristics of software projects that are common sources of difficulty or failure.

- "intrinsic" characteristics, not "accidental" ones (like programming language or inexperienced developers)

1. Change
2. Complexity
3. Bugs

Software "Facts of Life"

Name some characteristics of software projects that are common sources of difficulty or failure.

- "intrinsic" characteristics, not "accidental" ones (like programming language or inexperienced developers)

1. CHANGE
2. COMPLEXITY OF PROBLEM
3. BUGS or ERRORS

Principles for Good Design

1. GRASP Principles (*Larman*)
2. "Gang of Four" principles (from Design Patterns book)
   - *A class should have only one reason to change*
   - *Prefer object composition (association) over inheritance*
   - *Program to an interface, not to an implementation*
   - *Separate what varies from what stays the same*
   - *Depend on abstractions. Don't depend on concretions.*
3. SOLID principles by Robert Martin
   - *Single Responsibility Principle*
   - *Open-closed principle: a class should be open for extension but closed for modification*
   - *Liskov Substitution principle - for valid subclass.*

Observer Pattern

Context:

An object (the *Subject*) is the source of interesting events. Other objects (*Observers*) want to know when an event occurs.

Solution:

(1) Subject provides a method for Observers to register themselves as interested in the event.

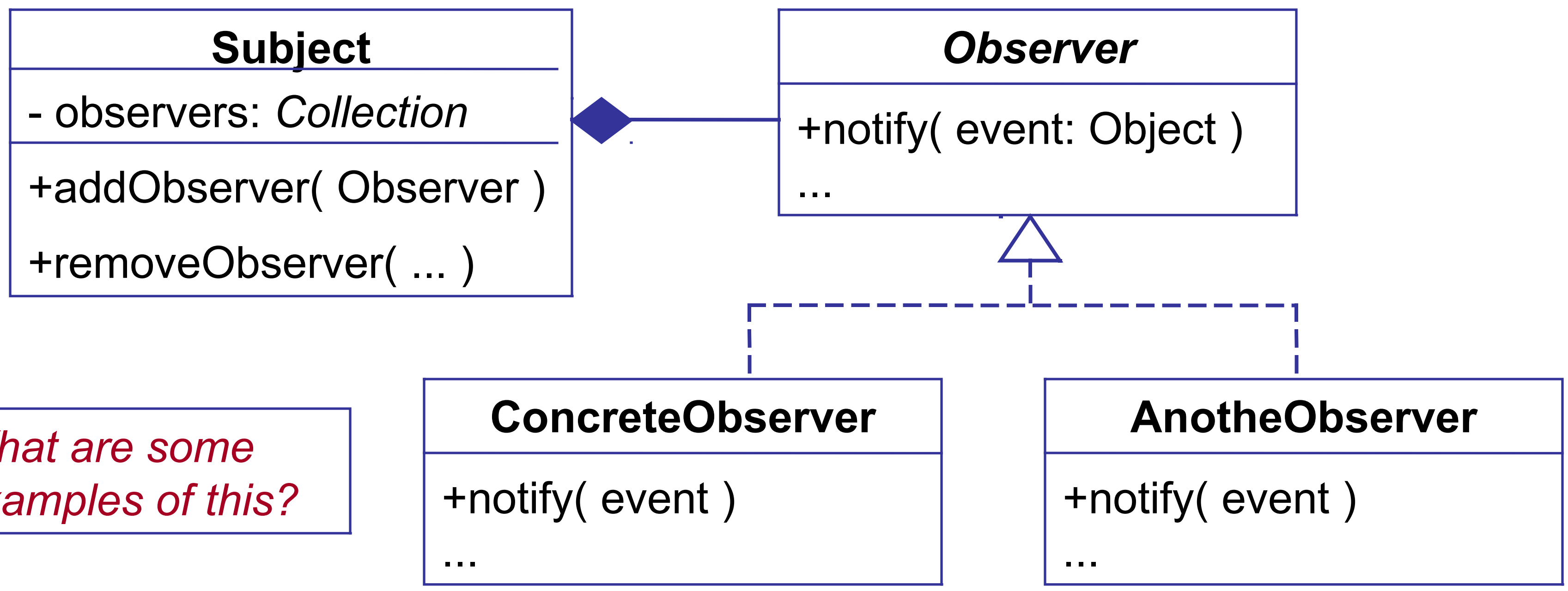
(2) Subject calls a known method (*notify*) of each Observer when event occurs.

Observer Pattern

Context: An object (the *Subject*) is the source of interesting events. Other objects (*Observers*) want to know when an event occurs.

Solution: (1) Subject provides a method for Observers to register themselves as interested in the event.

(2) Subject calls a known method (*notify*) of each Observer when event occurs.



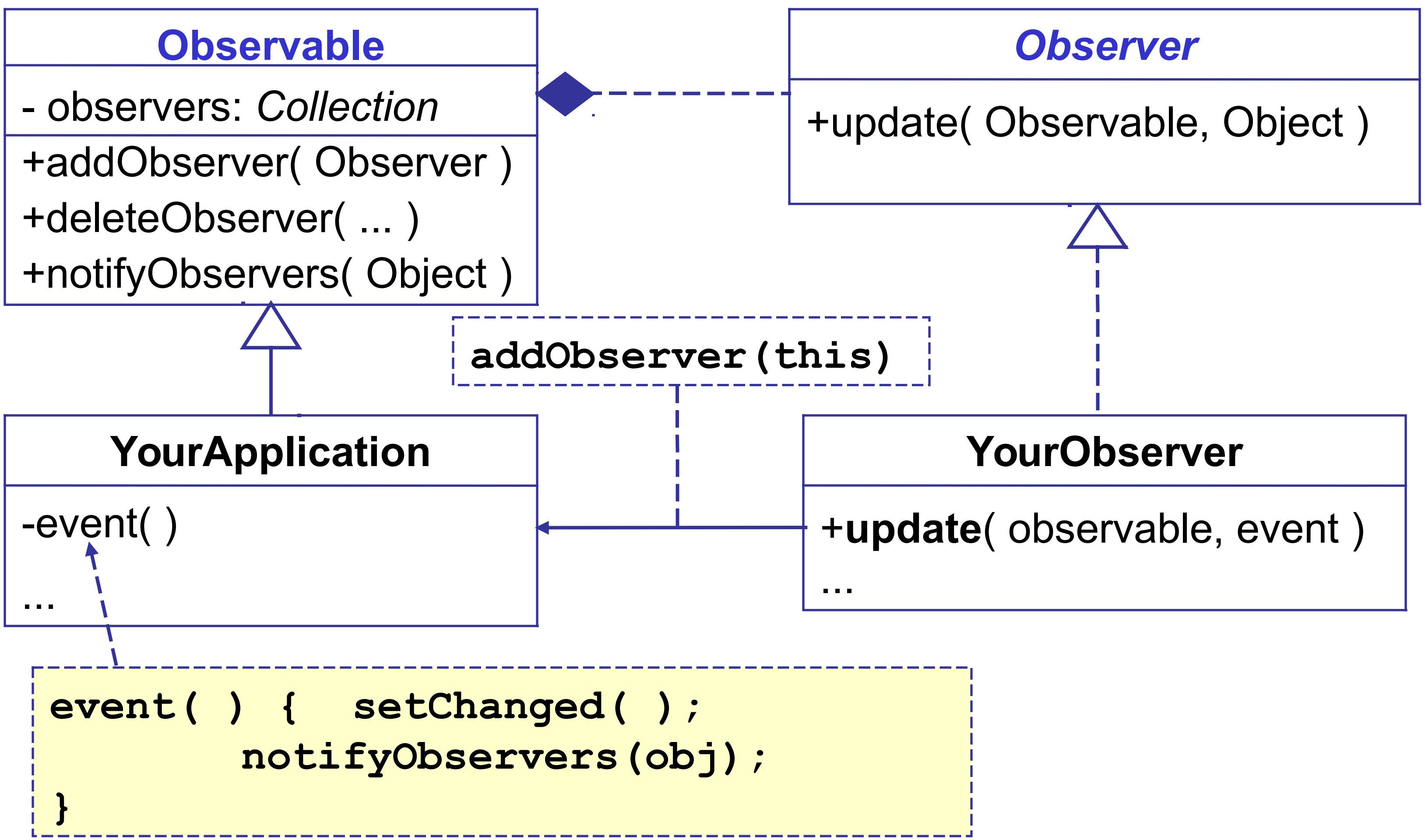
What are some examples of this?

Table for Identifying a Pattern

Name In Pattern	Name in Application: this is for a JButton
Subject *Observer* Concrete Observer	JButton *ActionListener* a class that implements *ActionListener*
addObserver(Observer)	addActionListener( )
notify(Event) [in the observer]	actionPerformed(ActionEvent)

Observer Pattern in Java

Java provides an **Observable** class and **Observer** interface that make it *easy* to use the Observer pattern..



Using the Observable class

(1) Declare that your class extends Observable

(2) When an event occurs, invoke setChanged() and notifyObservers( )

```
public class MySubject extends Observable
{
    Object myinfo;

  /** An event the observers want to know about */
  public void event( ) {
  doSomeWork( );
  // now notify the observers
  setChanged( );
  notifyObservers( ); // can include a parameter
  }
```

Writing an Observer

(3) Declare that observers *implement* the Observer interface.

```
public class MyObserver implements Observer {
  /* This method receives notification from the
   * subject (Observable) when something happens
   * @param message is value of parameter sent
   *   by subject in notifyObservers. May be null.
   */
  public void update( Observable subject,
                      Object message ) {
    info = ((MySubject)subject).getInfo( );
  ...
  ...
}
```

(4) update takes action using notification from the Subject.

Connecting Observer to Subject

Call addObserver( ) to add Observers to the subject.
You can have many Observers.

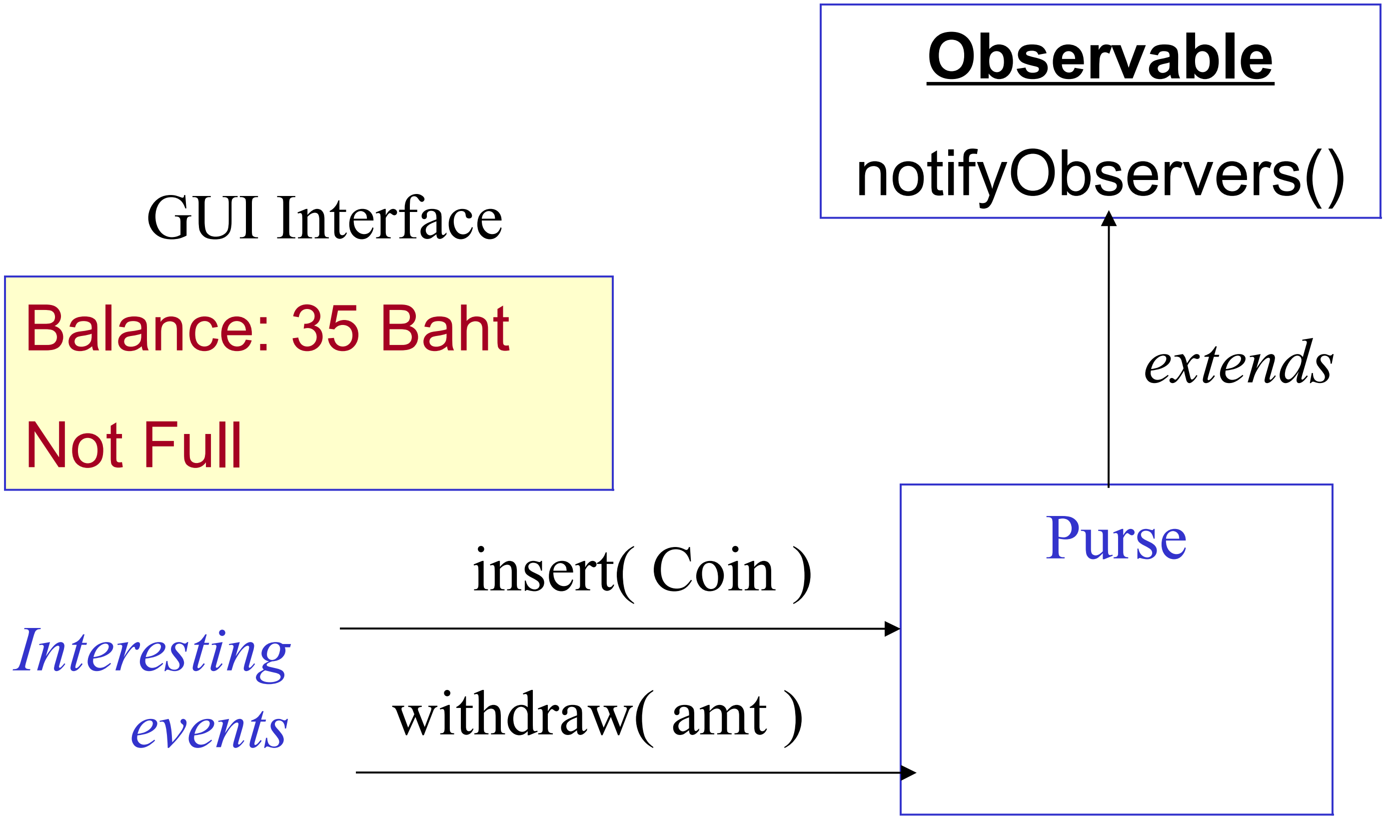
```
public static void main(String [] args) {
    Observable subject = new MySubject( );
    MyObserver observer = new MyObserver( );

    subject.addObserver( observer );

    subject.run( );

}
```

Example for Coin Purse



C# Delegates as Observers

- Delegate is a type in the C# type system.
- It describes a group of functions with same parameters.
- Delegate can act as a collection for observers.

```
/** define a delegate that accepts a string **/
public delegate void WriteTo( string msg );
```

```
/** create some delegates **/
WriteTo observers = new WriteTo( out.WriteLine );
observers += new WriteTo( button.setText );
observers += new WriteTo( textarea.append );
/** call all the observers at once! **/
observers("Wake Up!");
```

Design Patterns We Will Study

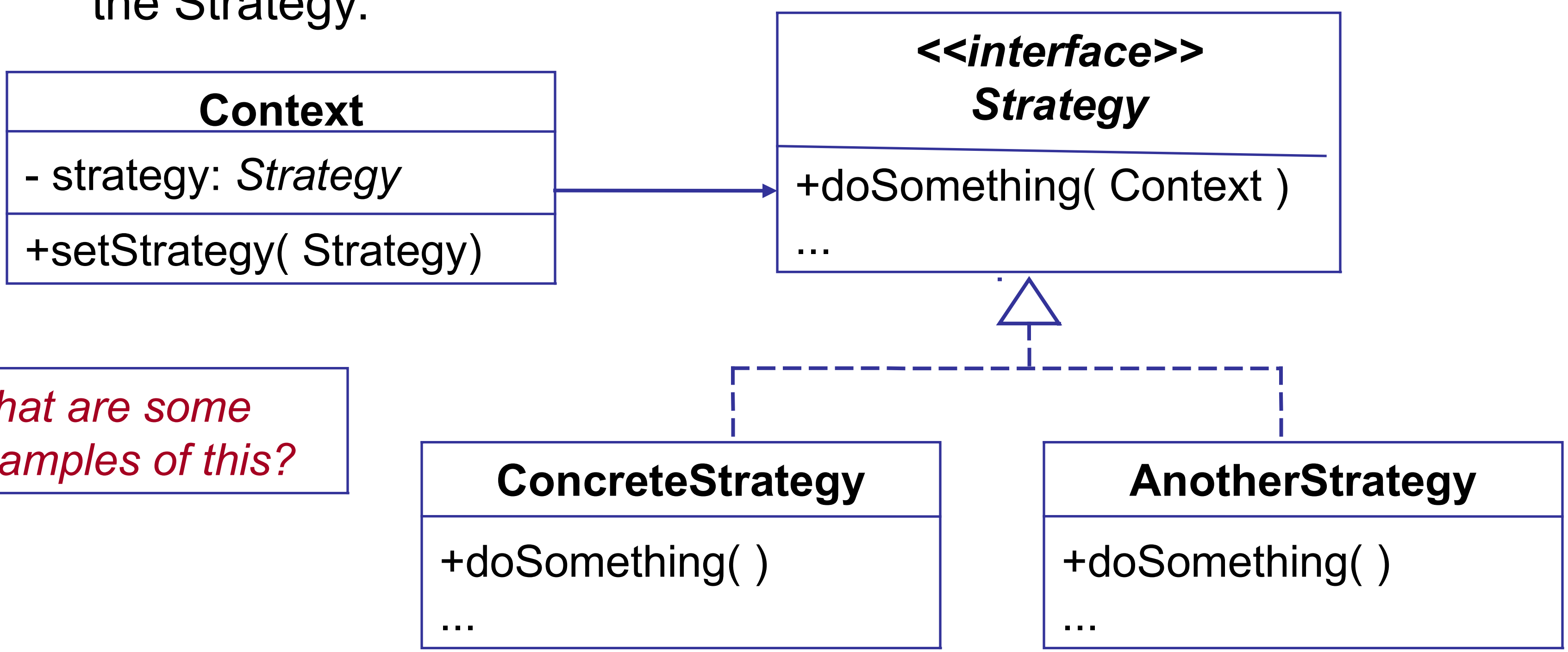
1. Iterator
2. Observer
3. Command
4. Strategy
5. State
6. Adapter
7. Factory Method
8. Decorator
9. MVC

Strategy Pattern

Context: A class requires some behavior, but there are many ways that this behavior can be implemented.

Solution: implement the behavior in a separate class, called the *Strategy*.

Create a Strategy interface to de-couple the context class from the Strategy.



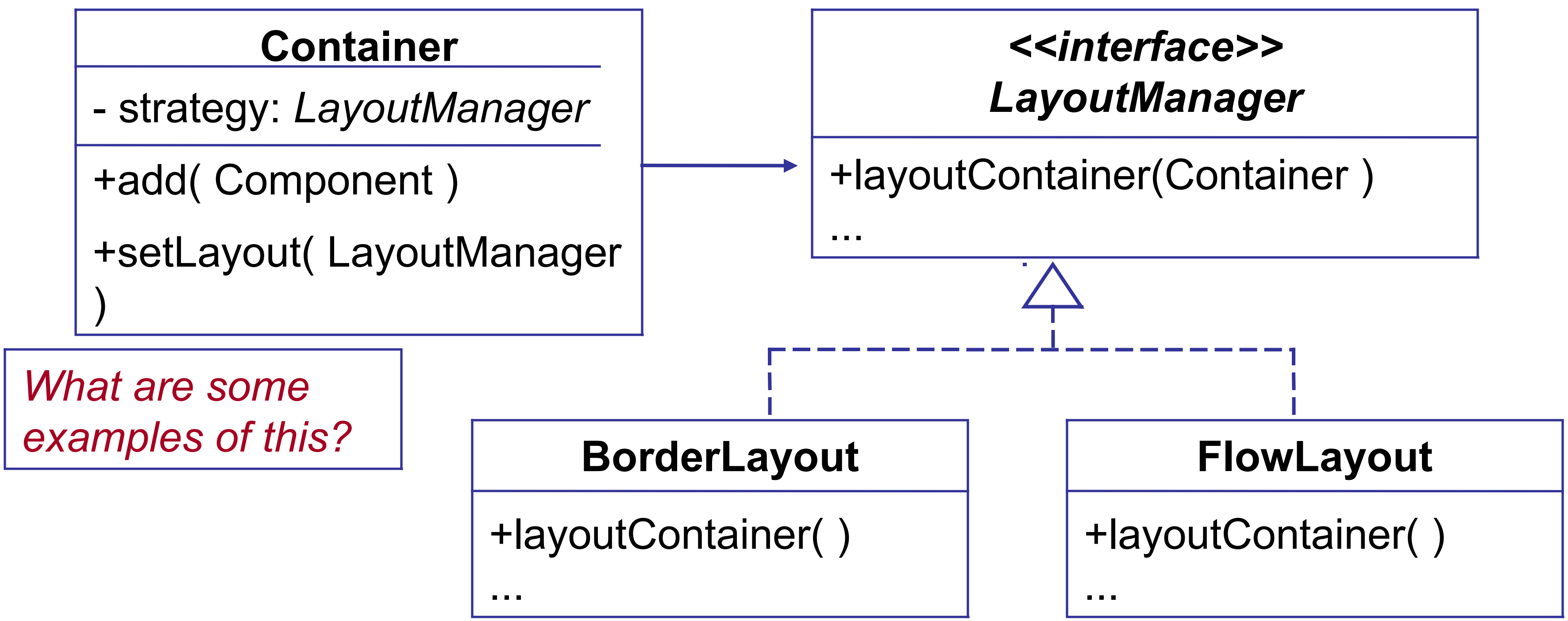
What are some examples of this?

Container uses Strategy Pattern

Context: Swing container.

Strategy: LayoutManager.

Create a Strategy interface to de-couple the context class from the Strategy.



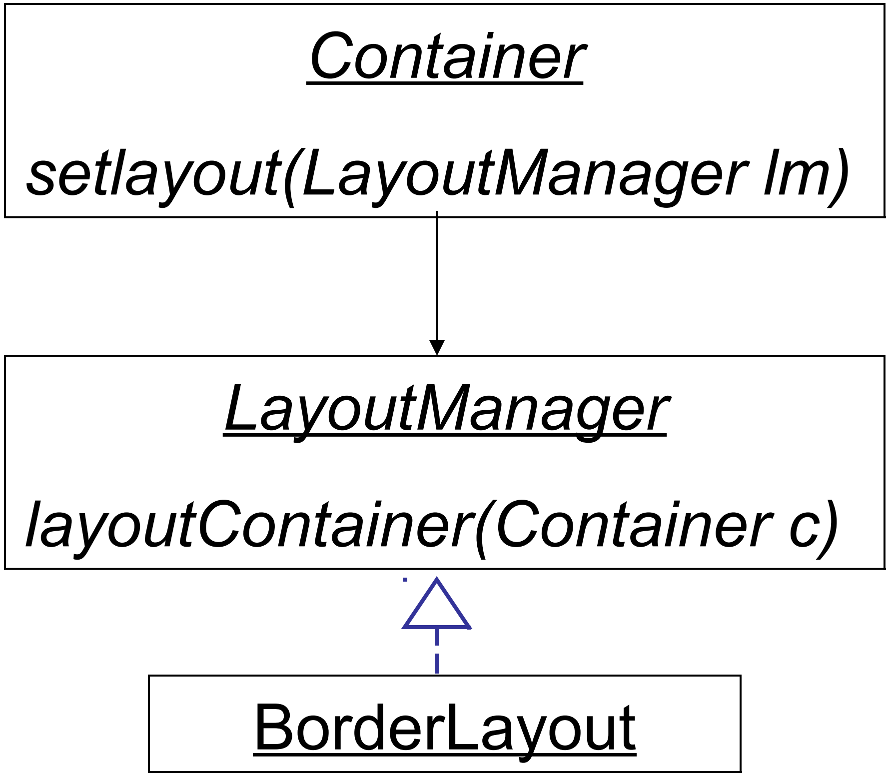
Strategy needs access to Context

To do its job, the Strategy usually needs a reference to the Context or some data of the Context.

Context: AWT/Swing Container (JPanel ...) contains components.

Strategy: A LayoutManager arranges and resizes components.

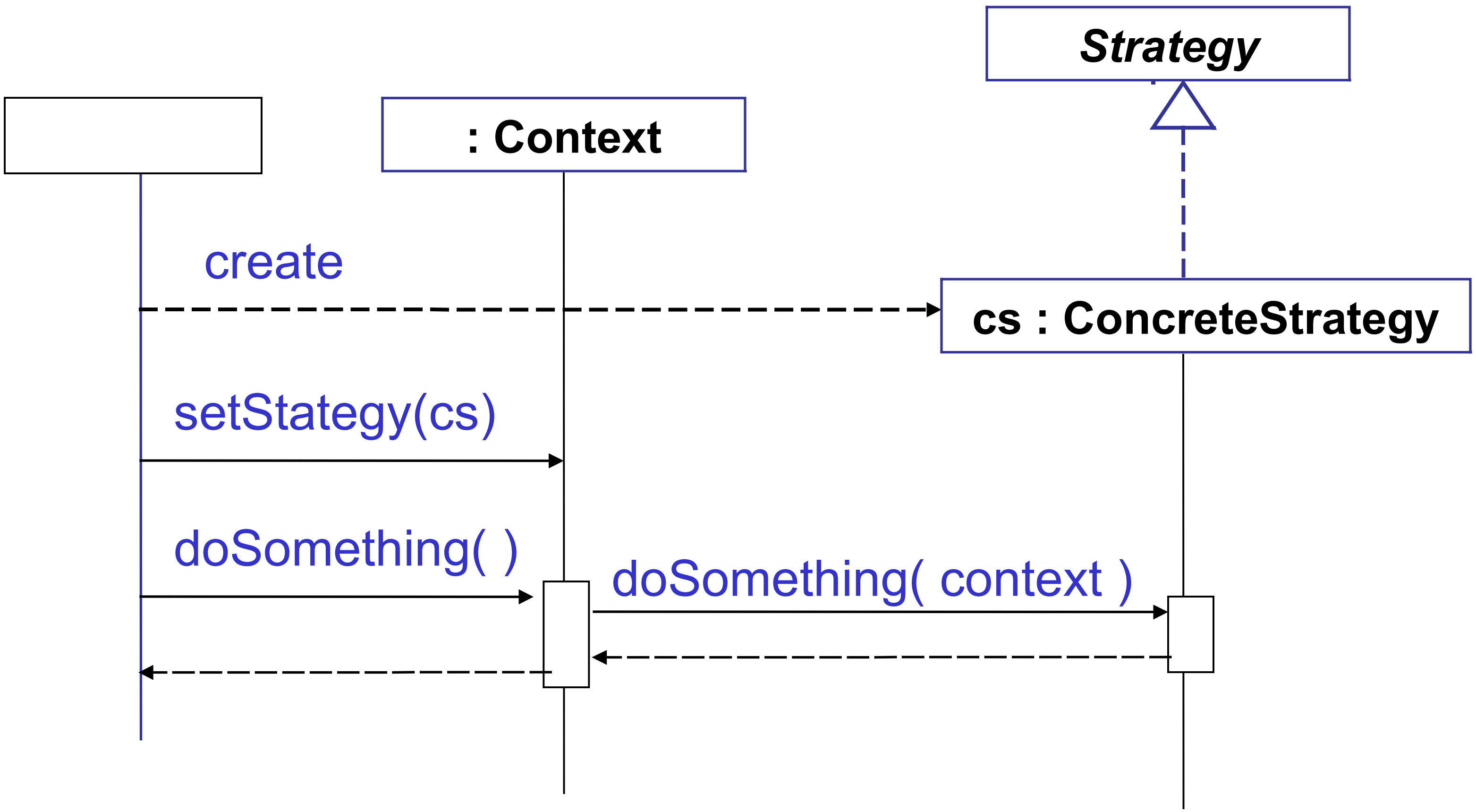
LayoutManger needs a reference to Container to get size and list of Components.



Using the Strategy Pattern

(1) The application creates a concrete strategy and assigns it to the context.

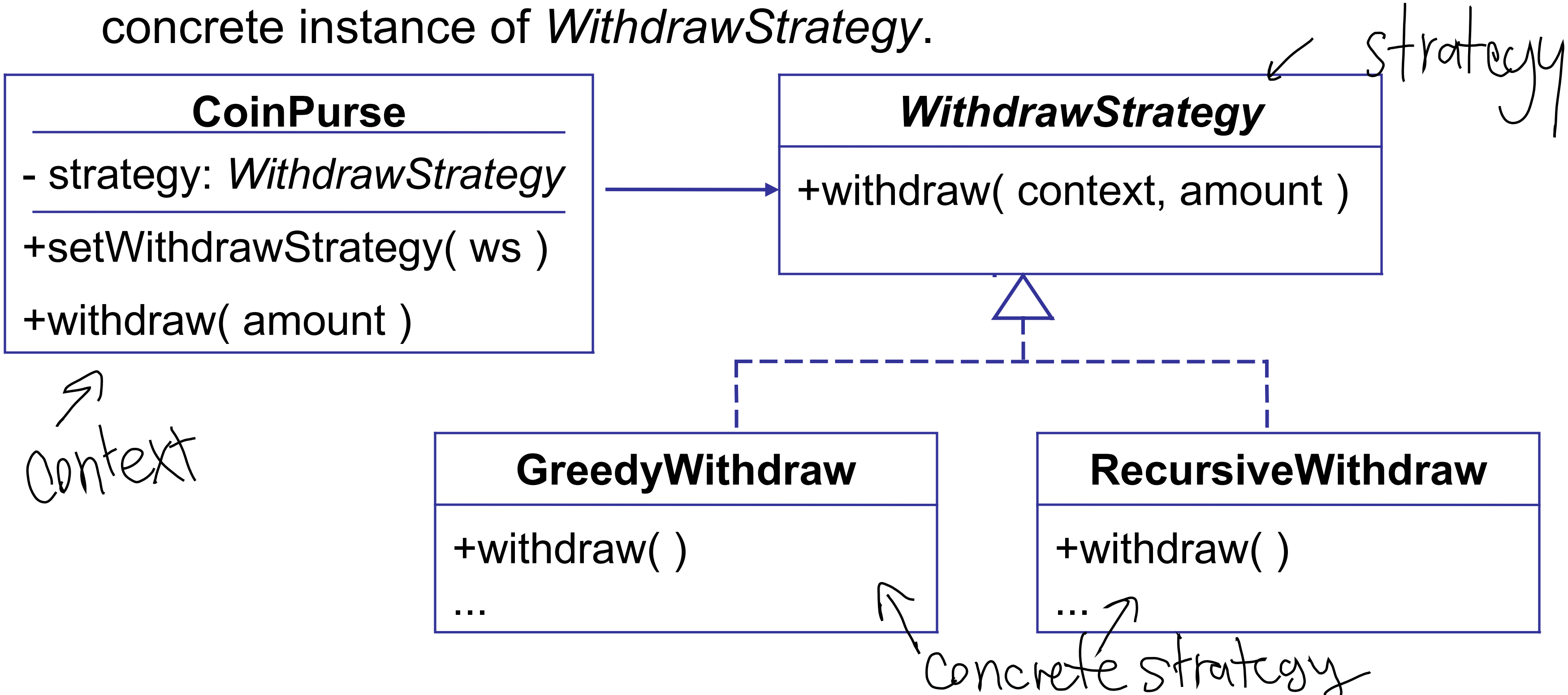
(2) The context delegates some work to the Strategy.



Strategy Pattern for Coin Purse

Context: A coin purse must decide what coins to withdraw; there are many ways to do this and we may want to change strategies.

Solution: Separate the withdraw( ) method from the Purse. Define a *WithdrawStrategy* interface for the withdraw operation, and modify the purse to *delegate* the withdraw operation to a concrete instance of *WithdrawStrategy*.



Principles seen in Strategy

Open-Closed Principle: Context can be *extended* with new strategies without modifying existing code

Single Responsibility: separate the "strategy" responsibility from the context using the strategy

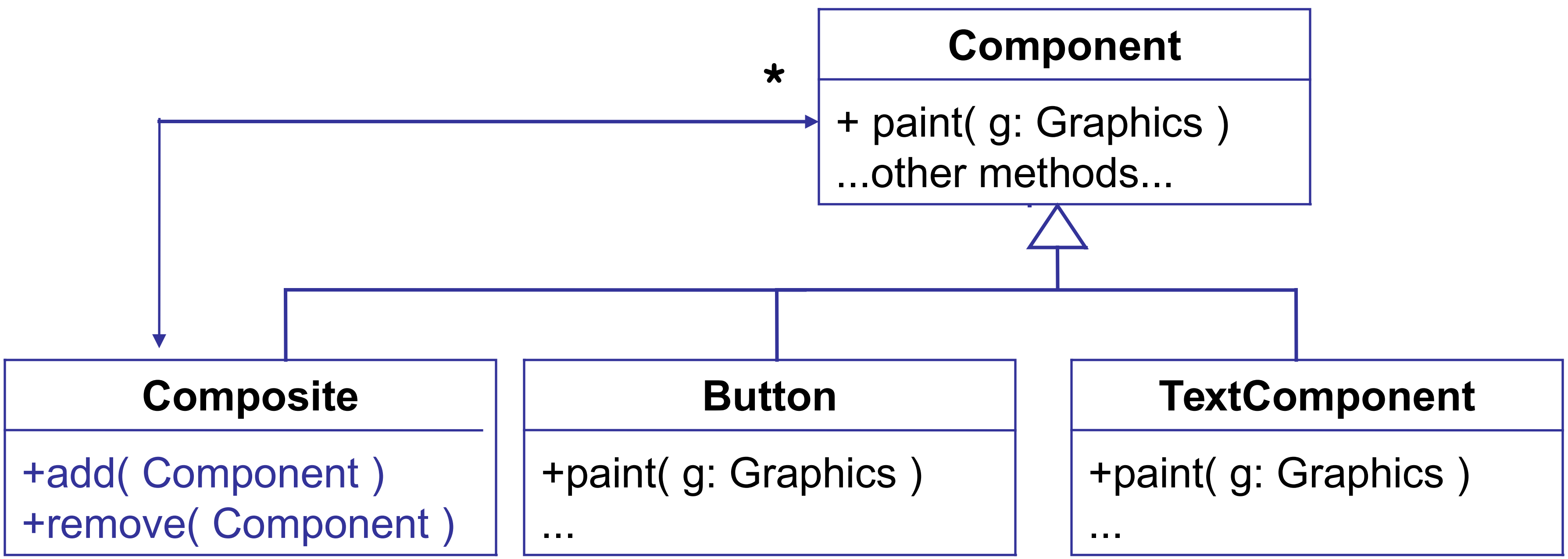
Separate that which varies from that which stays the same.

Prefer object composition over inheritance.

Composite Pattern

Context: We have a generic component that our application uses. We'd like to group together multiple components so that they look and behave like a single component.

Solution: Create a *composite* that implements the component interface and contains a collection of components. The compposite is responsible for managing components.



Situations (Context) not Patterns

Don't memorize pattern names.

Learn the situation and the goals (forces) that motivate the solution.

Adding New Behavior

Situation:

we want to add some new behavior to an existing class

Forces:

1. don't want to add more responsibility to the class
2. the behavior may apply to similar classes, too

Example:

Scrollbars

Changing the Interface

Situation:

we want to use a class in an application that requires interface A. But the class doesn't implement A.

Forces:

1. not appropriate to modify the existing class for the new application
2. we may have many classes we need to modify

Example:

change an Enumeration to look like an Iterator

Convenient Implementation

Situation:

some interfaces require implementing a *lot* of methods. But most of the methods aren't usually required.

Forces:

1. how can we make it *easier to implement interface?*
2. how to supply default implementations for methods?

Example:

MouseListener (6 methods), List (24 methods)

A Group of Objects act as One

Situation:

we want to be able to use a Group of objects in an application.

But the application can treat the whole group like a single object.

Forces:

1. need many objects in a framework that one allows us to insert one object.

Example:

KeyPad in a mobile phone app.

Creating Objects without Knowing Type

Situation:

- we are using a framework like OCSF.
- the framework needs to create objects.
- how can we change the type of object that the framework creates?

Forces:

- 1. want the framework to be extensible.
- 2. using "new" means coupling between the class and the framework.

Example:

- OCSF, JDBC DriverManager

Do Something *Later*

Situation:

we have a task that we want to run at a given time

Forces:

we don't want our "task" to be responsible for the schedule of when it gets run.

This problem occurs a lot, so let's write a reusable solution.

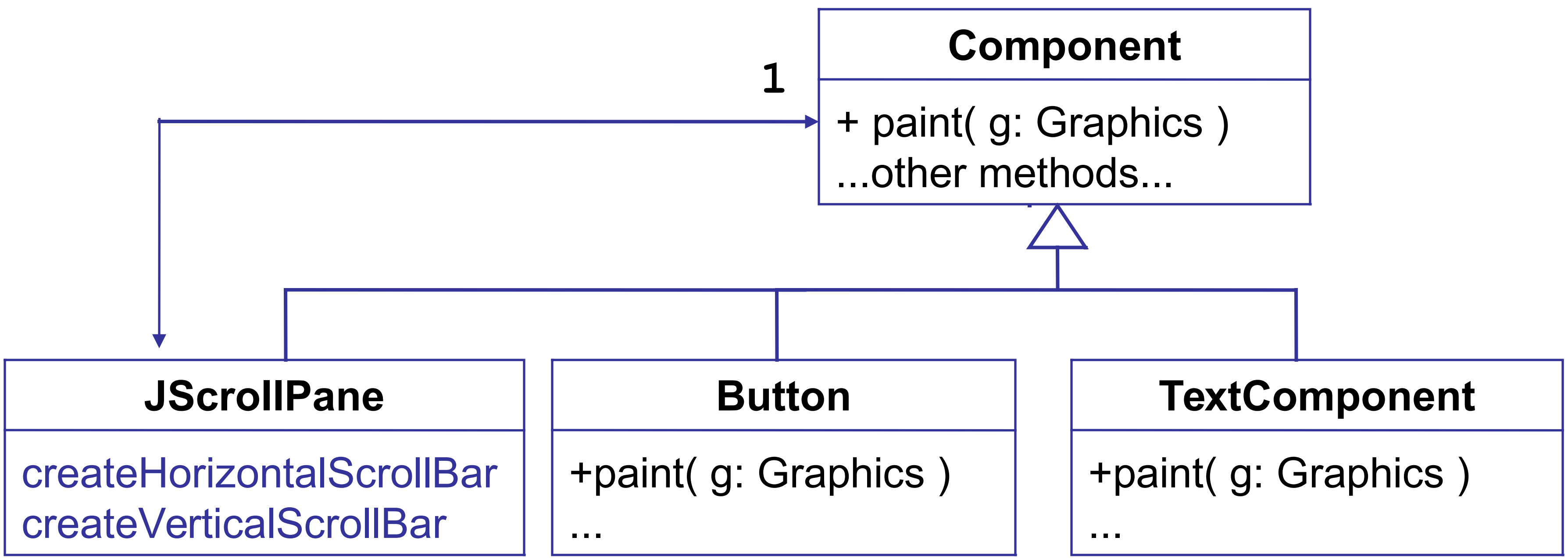
Example:

Decorator Pattern

Context: We want to *enhance* the behavior of a class, and there may be many (open-ended) ways of enhancing the class.

The *enhanced* class can be used the same as the base class.

Solution: Create an interface for the base class. The base class implements the interface. Create a *decorator* that implements the interface and wraps the plain class, "decorating" its behavior.



Decorator Example

Purpose: create a TextArea with scrollbars so that text will scroll when larger than the viewport.

```
// create TextArea with 5 rows, 40 columns
JTextArea textArea = new JTextArea( 5, 40 );
// decorate with JScrollPane to add scrollbars
JScrollPane pane = new JScrollPane( textArea );
pane.setVerticalScrollBarPolicy(
   JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED );

// Add the decorator to the contentpane.
// Don't add the textArea!
contentPane.add( pane );
```

Advantage of Using Decorators (1)

- We can write a behavior one time and apply it to many different kinds of objects.

Example: a JScrollPane can be applied to any component, not just JTextArea.

Advantage of Using Decorators (2)

- Improves the *cohesion* of objects, by not adding responsibility that isn't part of the object's main purpose.

Example: the purpose of a TextArea is to display text!

Not to manage scrolling.

Advantage of Using Decorators (3)

- New decorators can be added in the future, *extending* the behavior of the class.

Example: a *zoom decorator* to zoom a component.

Open-Closed Principle

A class should be open for extension but closed for modification.

Adapter Pattern

to be added.

Readers as Adapters

InputStream reads input as bytes.

```
int b = inputStream.read( );
```

InputStreamReader interprets the input as characters.

```
InputStreamReader reader
  = new InputStreamReader( inputStream );
char c = (char) reader.read( );
```

BufferedReader groups the characters into lines

```
BufferedReader bufReader
  = new BufferedReader( reader );
String line = bufReader.readLine( );
```

Adapter *wraps* a component

```
InputStream instream =
  new FileInputStream( "filename" );
InputStreamReader reader =
  new InputStreamReader( instream );
BufferedReader bufReader =
  new BufferedReader( reader );
String line = bufReader.readLine( );
```

BufferedReader (reads strings)

InputStreamReader (read chars)

InputStream (reads bytes)

Factory Method

Context:

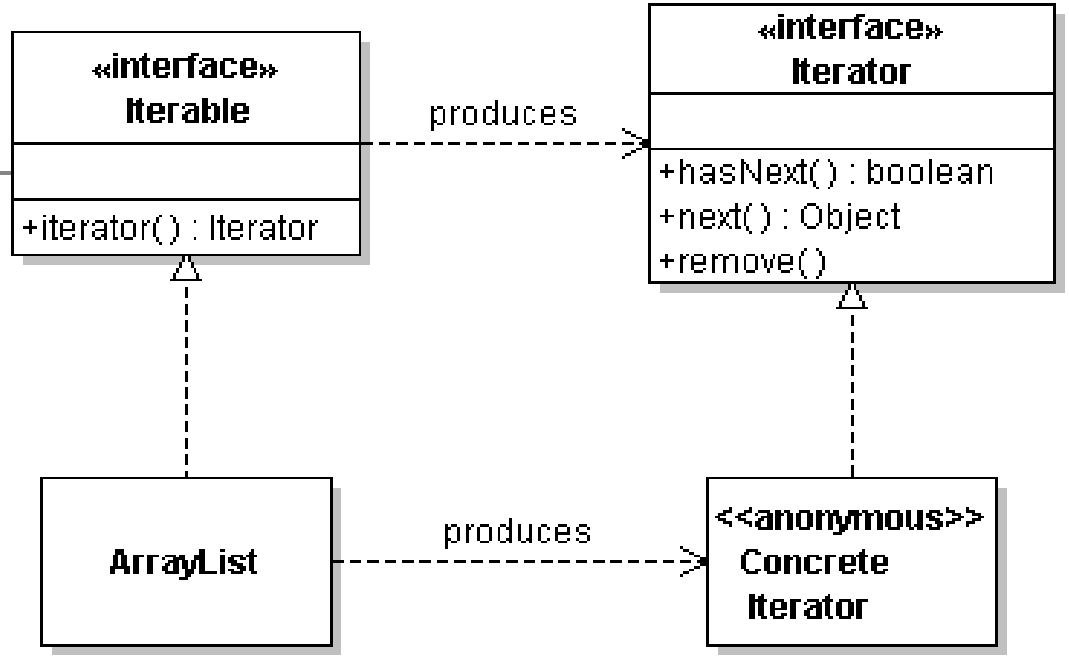
One class (the Factory) creates objects of another type (the Product). But, different classes need to create different implementations of the Product.

Example:

We need to access every item in a collection or other composite object, so we define an **iterator** that performs this job.

Every collection must create an iterator, but the implementation of the iterator depends on the type of collection.

Factory Method



Factory Interface	Iterable
Factory Method	iterator( )
Product	Iterator
Concrete Product	class implementing Iterator

Model-View-Controller (MVC) Pattern

- An *architectural pattern* used to help separate the external *view* of an application from the logic
  - The ***model*** contains the underlying classes containing the logic and state of the application
  - The ***view*** contains objects used to render the appearance of the data from the model in the user interface
  - The ***controller*** handles the user’s interaction with the view and the model

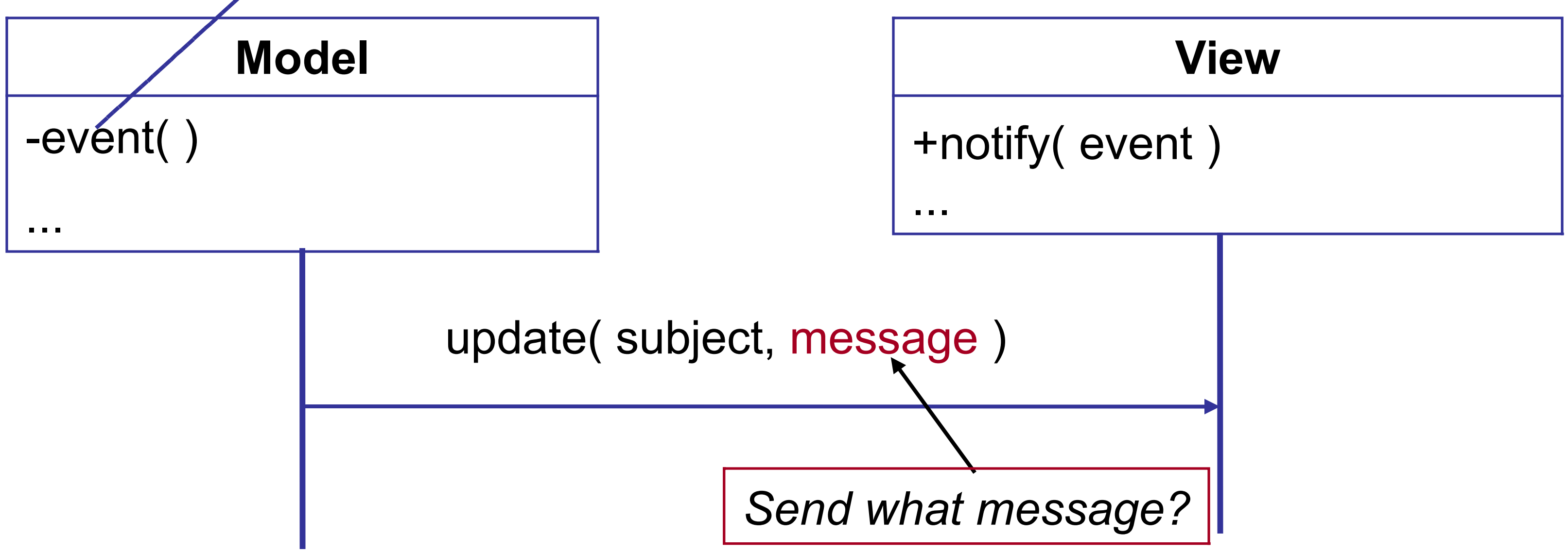
MVC Applications

- GUI Interfaces, like *Swing* (see *Core Java,* p.339-345)
- Editors: *Dreamweaver*

Model and Views

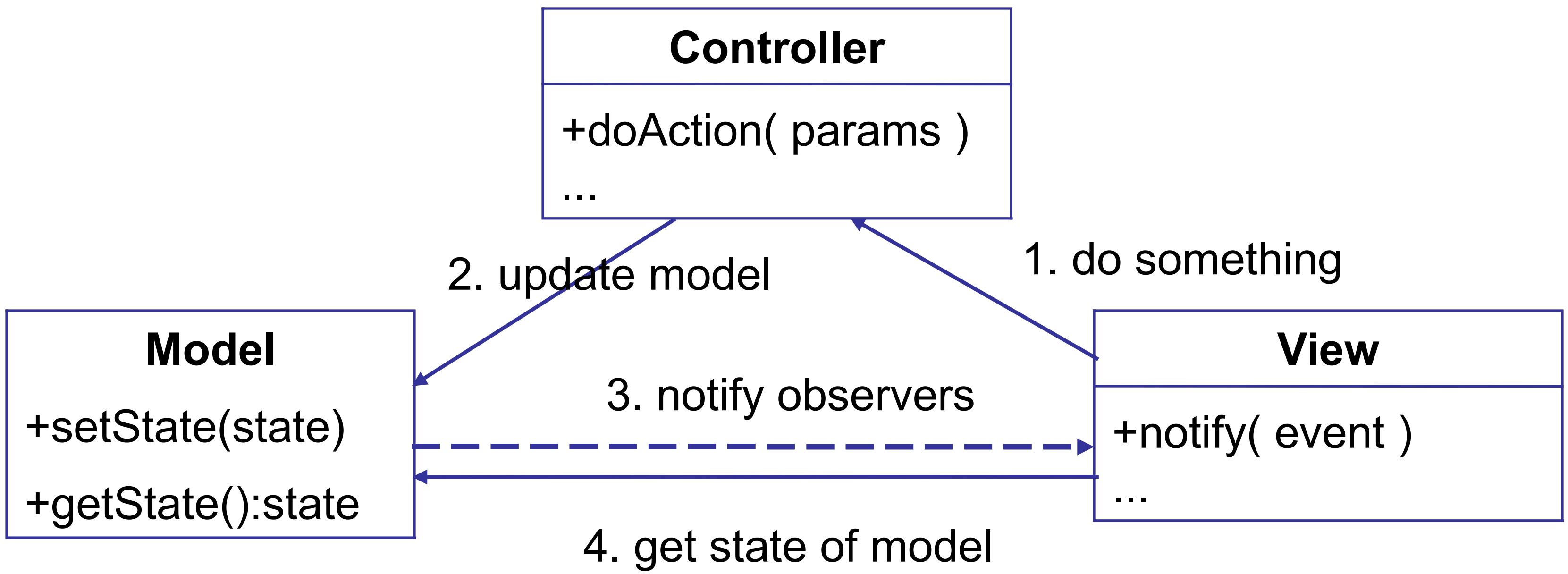
- **The model** and **view** relationship is often implemented as the Observable Pattern.
- This lets the application have multiple views.
- Reduces "coupling" between model and view.

```
event( ) {
    setChanged( );
    notifyObservers(message);
}
```

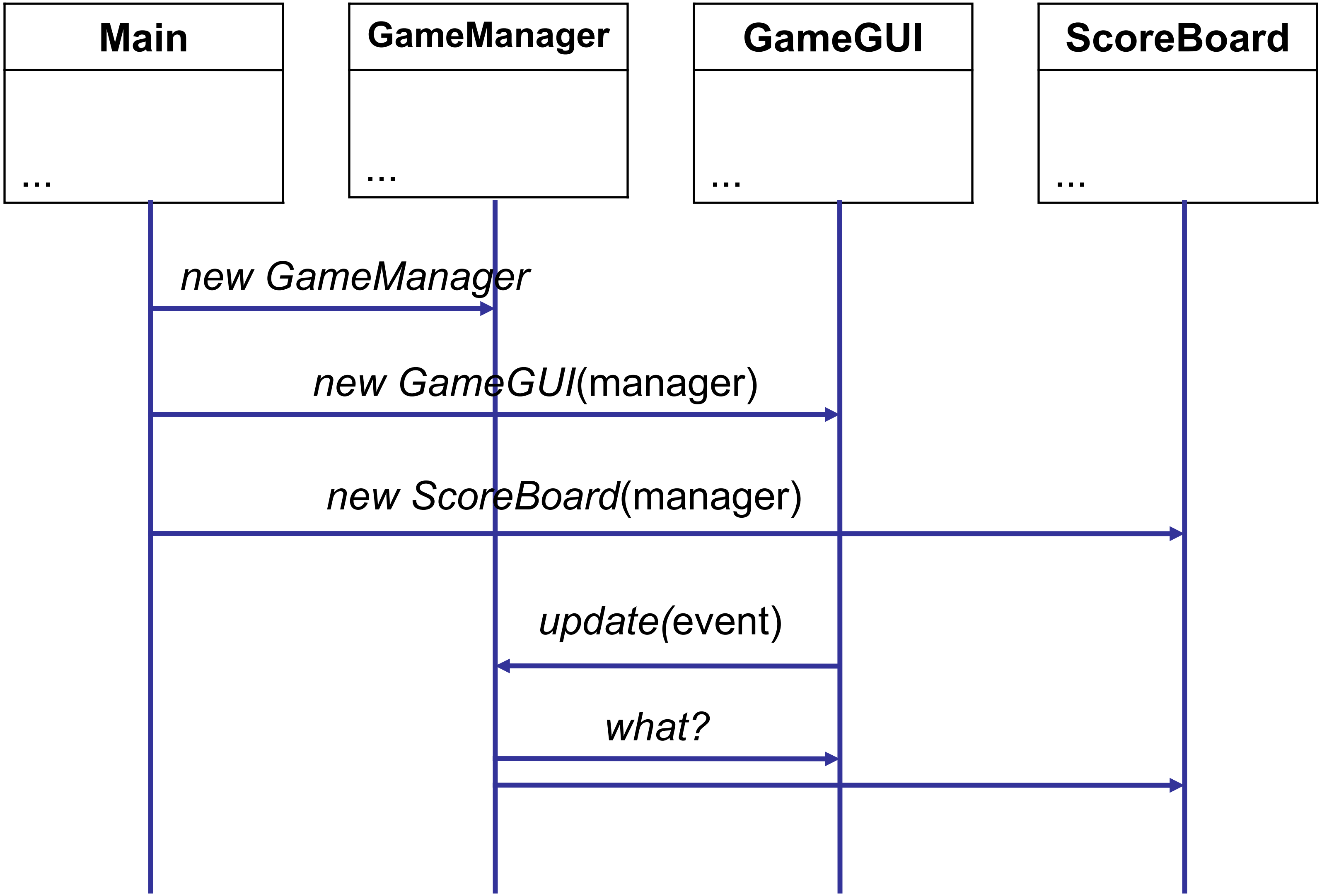


View and Controller

The view calls the controller class when it wants to do something.



Guessing Game with UI



Categories of Patterns

Creational - how to create objects

Structural - relationships between objects

Behavioral - how to implement some behavior