CSCI E-97
Lecture 10

November 6, 2014

Outline

- Assignment 4
- More behavioral patterns
  - Mediator
  - Observer
  - State
  - Strategy

Assignment 4

- Authentication.csv file will be posted with assignment 4 materials
- Review gradesheet
- Peer design reviews should be complete by Tuesday (11/10)
- Assignment 4 is due Thursday (11/20) at 11:59 pm
- Questions

Behavioral Patterns

- Behavioral Patterns
  - Command
  - Iterator
  - **Mediator**
  - **Observer**
  - **State**
  - **Strategy**
  - Template method
  - Visitor

Behavioral Patterns

- Behavioral Patterns are concerned with proper allocation of services to objects in a system
- This covers both the algorithms that must be executed in a program as well as inter-object communication
- Behavioral *class* patterns, such as the Template Pattern, use inheritance to distribute behavior between classes
- Behavioral *object* patterns, such as the Command Pattern, use object composition to distribute behavior between objects

Example Behavioral Design Questions

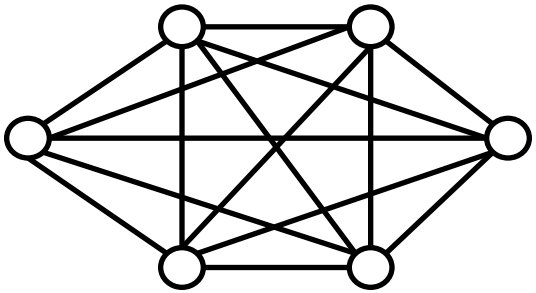
- How can I simplify the interactions among a group of collaborating objects (see the Mediator Pattern)
- How can I create a set of callback objects that simplifies the way the system responds to external requests (see the Command Pattern)
- How can I standardize the way that a client traverses a complex object without exposing the internal structure (see the Iterator Pattern, covered in Module 1)

Example Behavioral Design Questions – 2

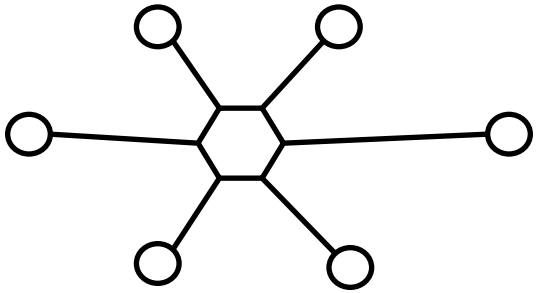
- How can I design an object so that it behaves differently as it transitions through a set of state changes (see the State Pattern)
- How can I manage a whole family of like algorithms, each of which can be applied in the same context (see the Strategy Pattern)

The Mediator Pattern

- The Problem: We want to be able to encapsulate the way that a group of objects interacts so that the objects in the group don't have to know about each other explicitly. This reduces an *nxn* communication problem to an *n+n* problem

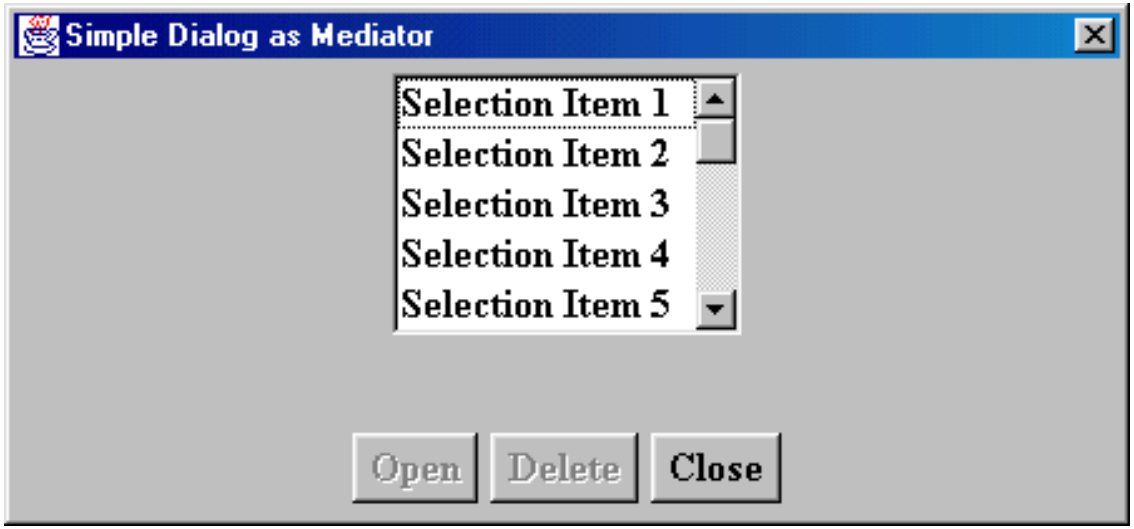


nxn communication



n+n communication

Example – A Simple Dialog

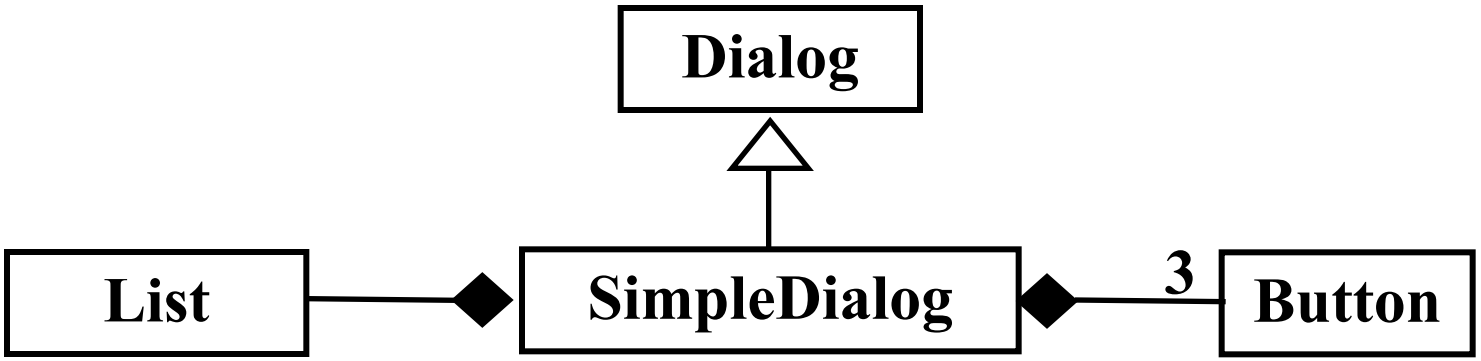


- Before any item is selected, the Open and Delete buttons are disabled
- Selecting an item enables the Open and Delete buttons
  - And so on…

Implementing the Dialog

- Assume we want to use existing classes such as `Dialog`, `Button`, and `java.awt.List` to implement the dialog
- In a typical windowing system, we would build a class, say `SimpleDialog`, as the container for the `List` and `Button` objects
- The question now becomes how to manage the interactions needed among the `List` and the three `Button` objects in the instance of `SimpleDialog`

A Class Diagram for `SimpleDialog`



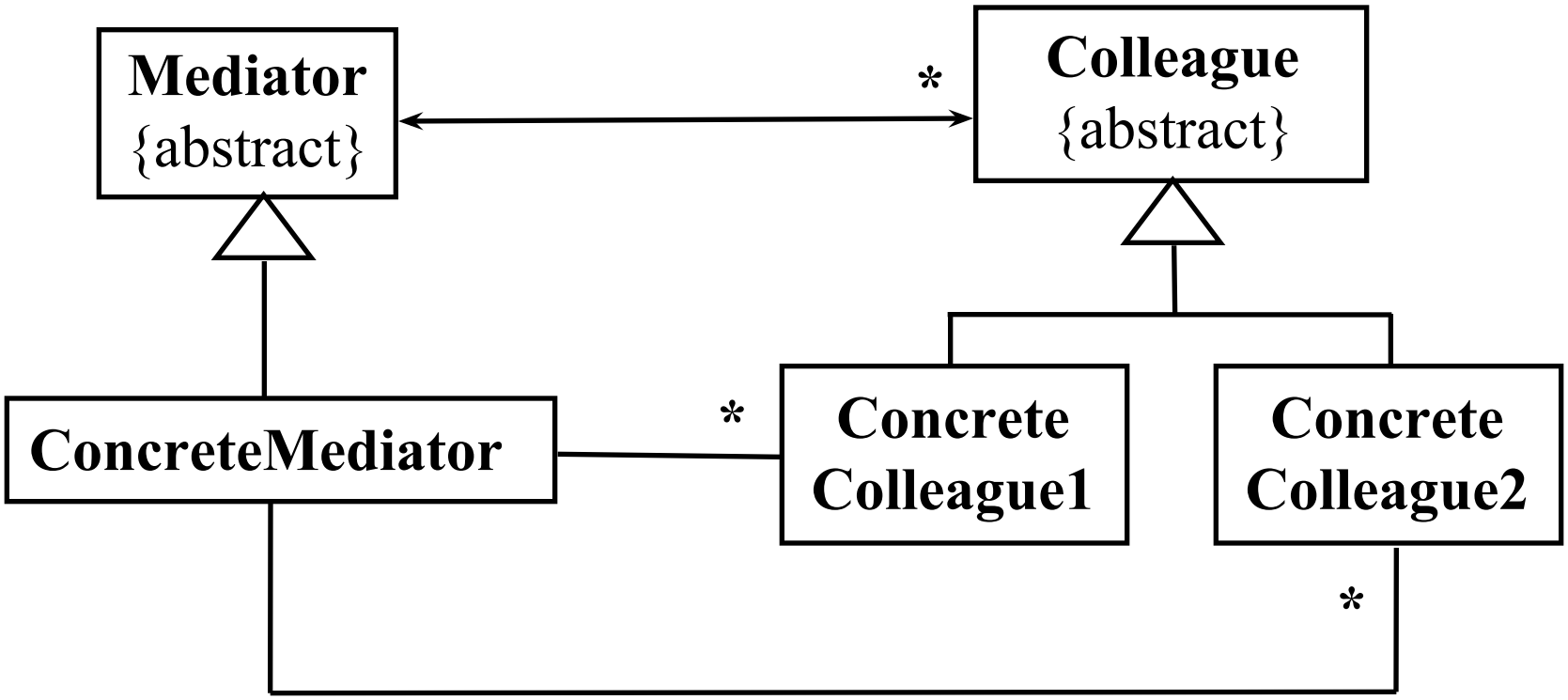
Choices for Implementing the Interaction

- We can create a new class whose sole responsibility is to perform the mediation role
- We can assign the mediation role to an existing object, such as the `SimpleDialog` or the object that creates the `SimpleDialog`
- We can create subclasses of the standard windowing classes that are specialized to manage the interaction

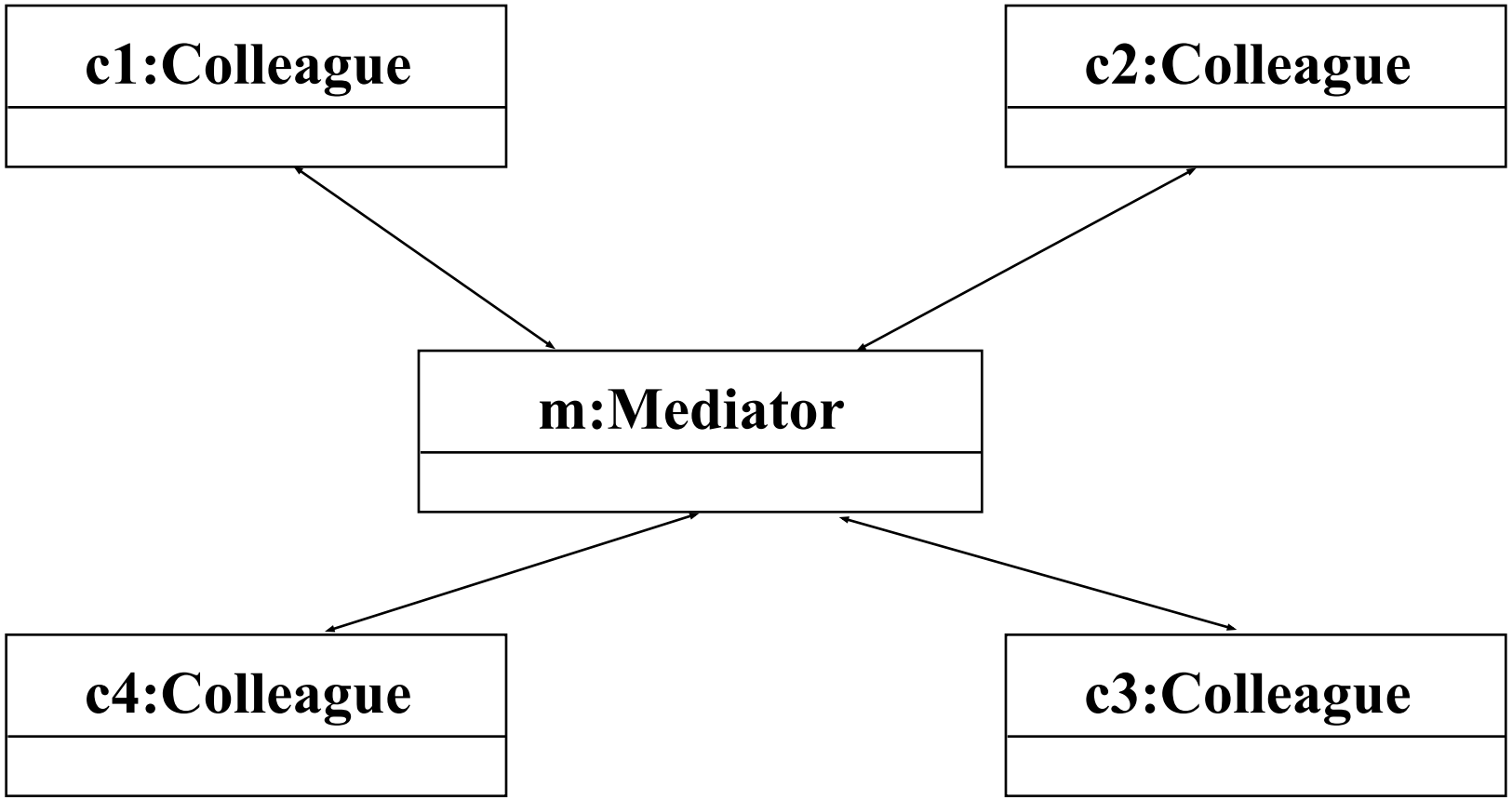
Using the Dialog As the Mediator

- The most appropriate choice is to assign the mediation functionality to the `SimpleDialog` class
- When an item is selected, the `List` object could notify the `SimpleDialog` object, which in turn could enable the `Button` objects
- Similarly, when an Open or Delete `Button` is selected, the `SimpleDialog` instance would be notified and then execute an action on the selected item

The Generic Model for the Mediator



An Instance Diagram for the `Mediator`



Sample Code – The `SimpleDialog` Class

```
// Somewhere in the initialization
// of the SimpleDialog object, it sets
// up notifications from the Button
// objects and Listbox object by adding
// itself as a listener for their events

 myCloseButton.addActionListener(this);
 myOpenButton.addActionListener(this);
 myDeleteButton.addActionListener(this);
 myListBox.addItemListener(this);
```

Sample Code – The `SimpleDialog` Class – 2

```
// itemStateChanged is the callback to handle an
// event generated by selecting an item
public void itemStateChanged(ItemEvent ie) {
  setCurrentSelection((Integer)(ie.getItem()));
  myOpenButton.setEnabled(true);
  myDeleteButton.setEnabled(true);
}
// The actionPerformed() callback for events on
// the three buttons is not shown
```

When to Use the Mediator Pattern

- When objects interact in an unstructured, complex way
- When you want to make the colleagues reusable in situations where not all the objects are present
- When the behavior that's distributed among several classes should be customizable without a lot of subclassing

Tradeoffs and Consequences

- Applying the Mediator Pattern limits the need to subclass all the different kinds of `Colleagues` – only the `Mediator` class needs to be subclassed
- The `Mediator` object centralizes control, which could be a drawback
- The `Mediator` object decouples `Colleagues` and simplifies inter-object protocols
- Taken to an extreme, the `Mediator` object takes on all the complexity of the interaction and becomes a bottleneck to change

Tradeoffs and Consequences – 2

- The `Mediator` need not be an abstract class
- In the example given, we might rely on an internal windowing system event to notify the `Mediator` of an action in the `Colleague` objects
- If we are more interested in data-change messages between the `Colleague` object and the `Mediator`, we can use the Observer Pattern defined next

Mediator Pattern Applications

User Interface

- Coordinating interaction between UI components

Air Traffic Control System

- AirCraft coordinate through central system

Ad Broker

- Mediates request for ads with ad providers.

Auction Service

- Mediates buyers with sellers

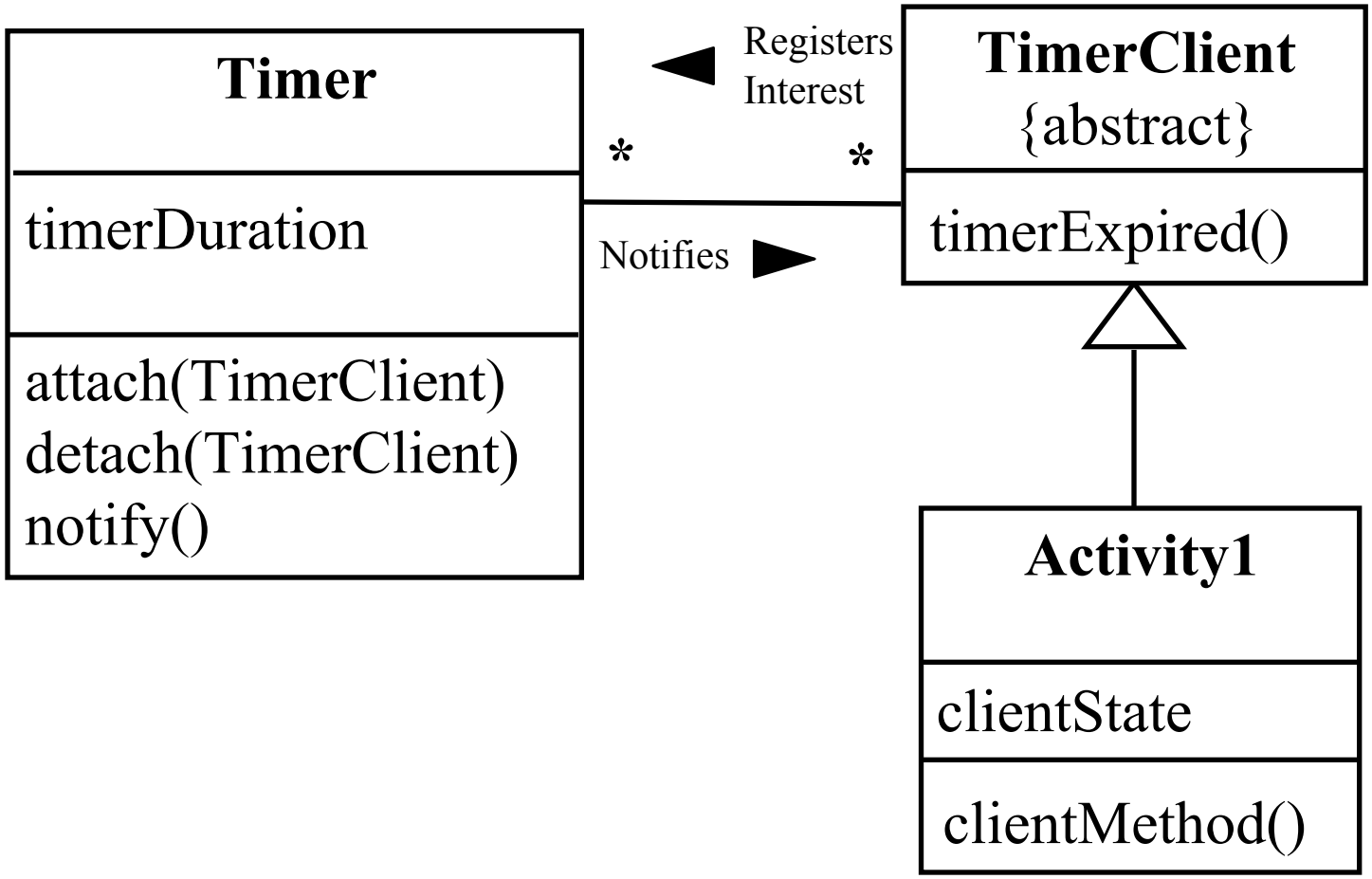
911 System

- Directs rescue teams to people who need help

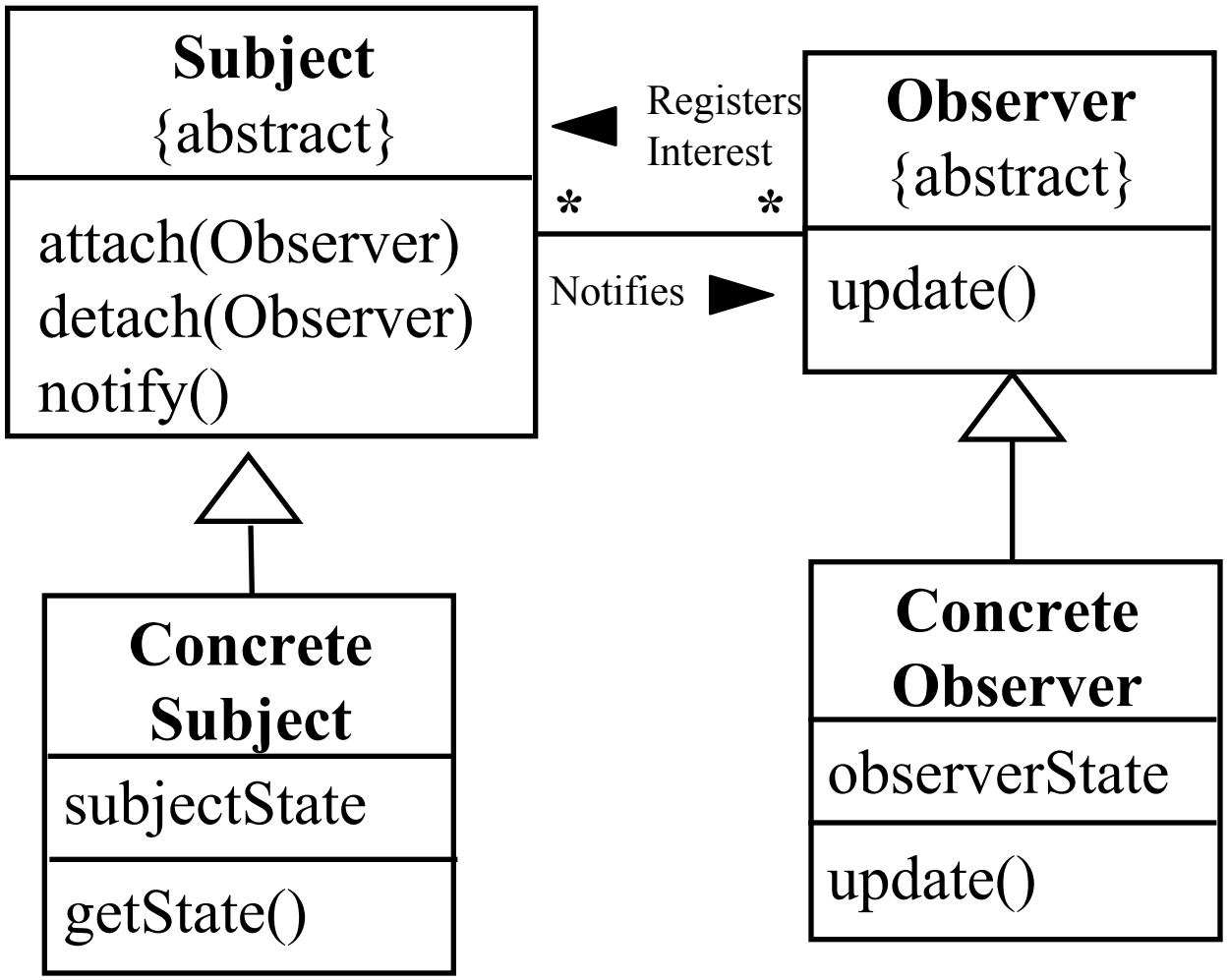
The Observer Pattern

- The Problem: We want to allow a one-to-many dependency among objects so that when one object changes, the others are notified

Example – A Timer



The Generic Model for the Observer Pattern



Sample Code – The `TimerClient` Class

```
public class TimerClient {
    public void timerExpired(){
        // do what you gotta do
    }
    // other methods and data
}
```

Sample Code – The `Timer` Class

```
class Timer {
    private long timerDuration;
    private LinkedList timerListeners;

    public Timer(long duration){
        timerDuration = duration;
    }
    public void attach(TimerClient tClient) {
        timerListeners.add(tClient);
    }
    public void detach(TimerClient tClient) {
        timerListeners.remove(tClient);
    }
```

Sample Code – The `Timer` Class – 2

```
  public void notifyListeners(){
    Iterator listenerIterator =
                        timerListeners.iterator();
    TimerClient tc;
    while (listenerIterator.hasNext()) {
      tc = (TimerClient)(listenerIterator.next());
      tc.timerExpired();
    }
}// end class Timer
```

When to Use the Observer Pattern

- When you want a level of decoupling so that the changed object need not be aware at compile time as to who the Observers are

Tradeoffs and Consequences

- The `Subject` class can define more than one kind of data-change event, in which case the `Observer` will attach itself to one or more particular data-change events as needed
- A third object (perhaps a `Mediator`) could attach an `Observer` to a `Subject`
- The `update()` method on `Observer` can be defined to contain more specific information about what data has changed, or the identity of the `Subject`

Tradeoffs and Consequences – 2

- Internally, the `Subject` class can keep a list of all `Observers` and traverse that list whenever it needs to notify all `Observers` of data-change events
- If a `Subject` is to be shared among multiple `Observers`, some protocol needs to be established so that the `Subject` is not deleted while it has listeners

A Second Example of the Use of Observer

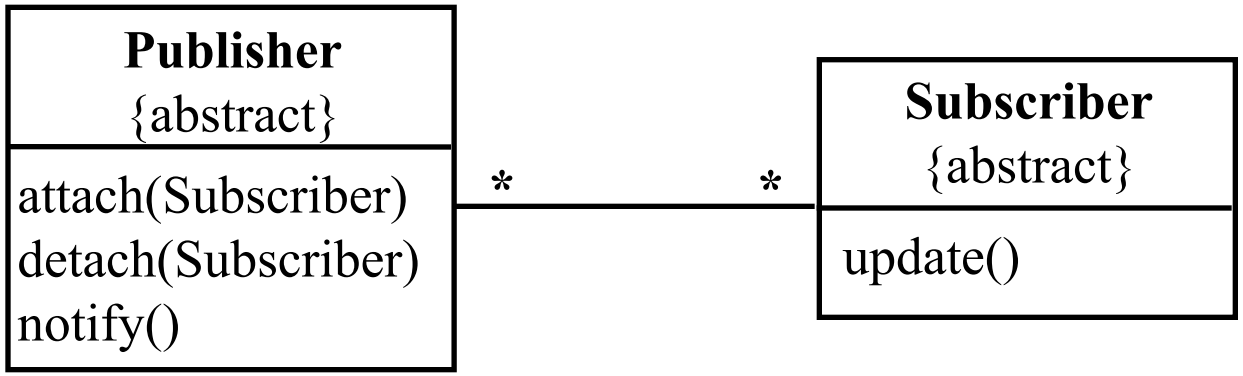
- The Sudoku Advisor is an example given for the Command Pattern
- It also takes advantage of the Observer Pattern
- The `SudokuClient` is an observer of the `SudokuPuzzleData` object. It receives notifications when
    a) A new legal value has been added to the puzzle data
    b) The puzzle has been solved
    c) A cell for which the collection of possible legal plays is empty
- The `SudokuClient` should provide a separate method for each of these notifications.

Event-Driven Systems

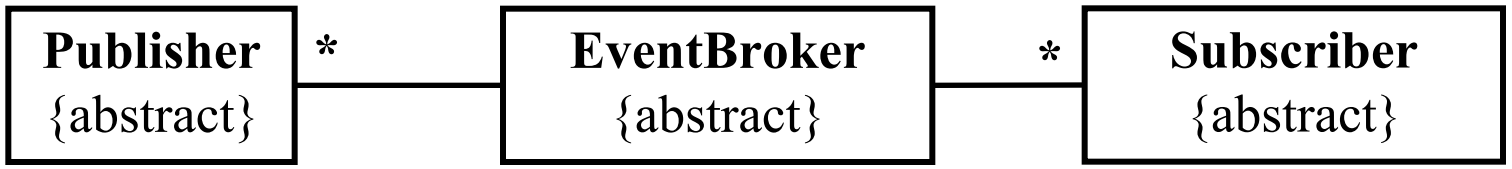
- Some systems are based on a concept of asynchronous event notification
- One example would be a network monitoring application where various network devices generate events related to their status. In that case there are multiple systems generating events and multiple systems receiving them
- In this case one can define a specialized 'Event Broker' to handle the event notification

An Event Broker Model

- Applying Observer directly requires Publishers and Subscribers to find out about each other



- Instead we can put a specialized object between them to handle event notification



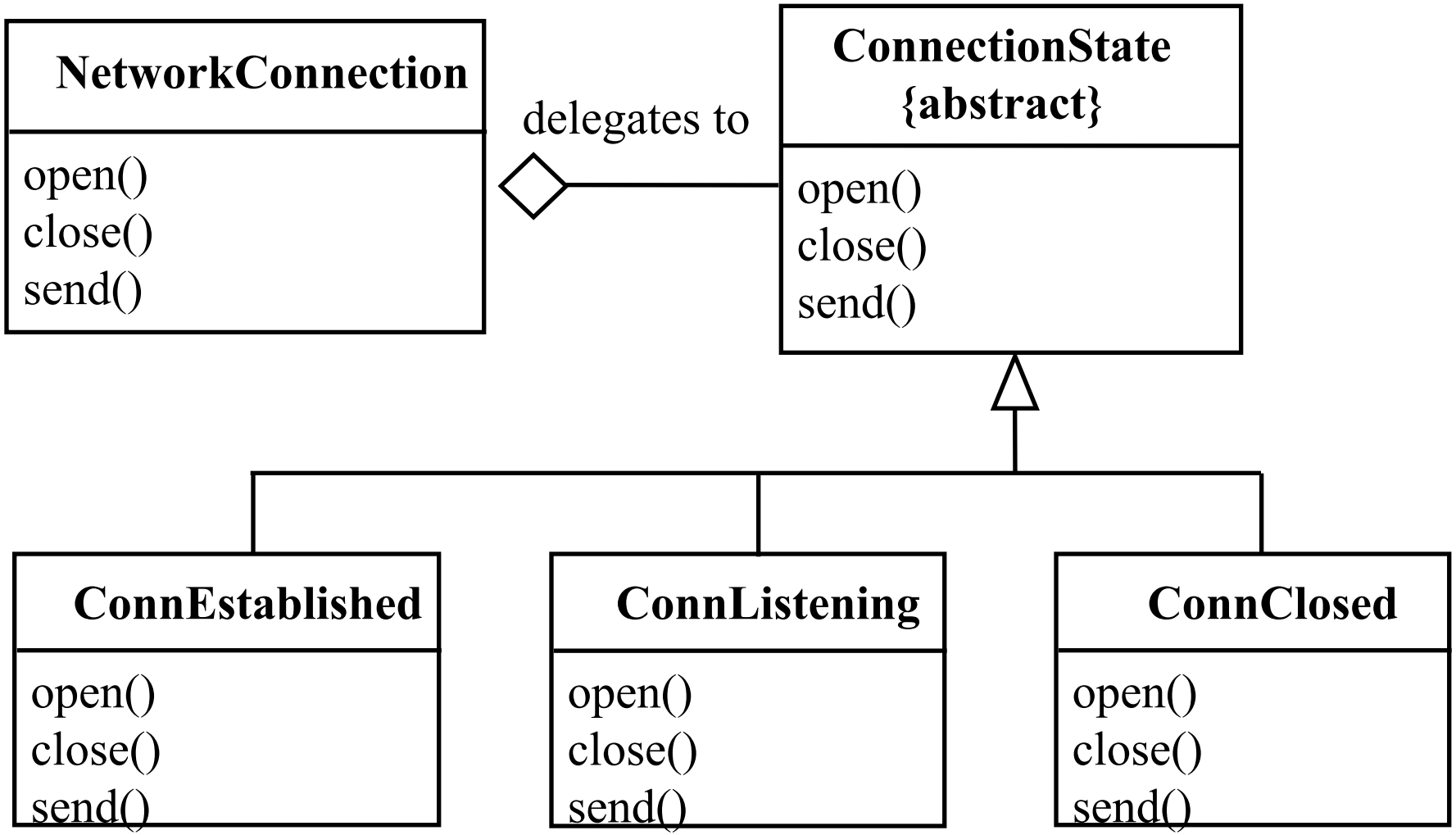
The State Pattern

- The Problem: We want to allow an object to alter its behavior when its internal state changes (as if the object had changed its class)
- Use the State Pattern when an object can have different behavior based on its current state, there are many possible states, and you want to to avoid many state-dependent switch statements

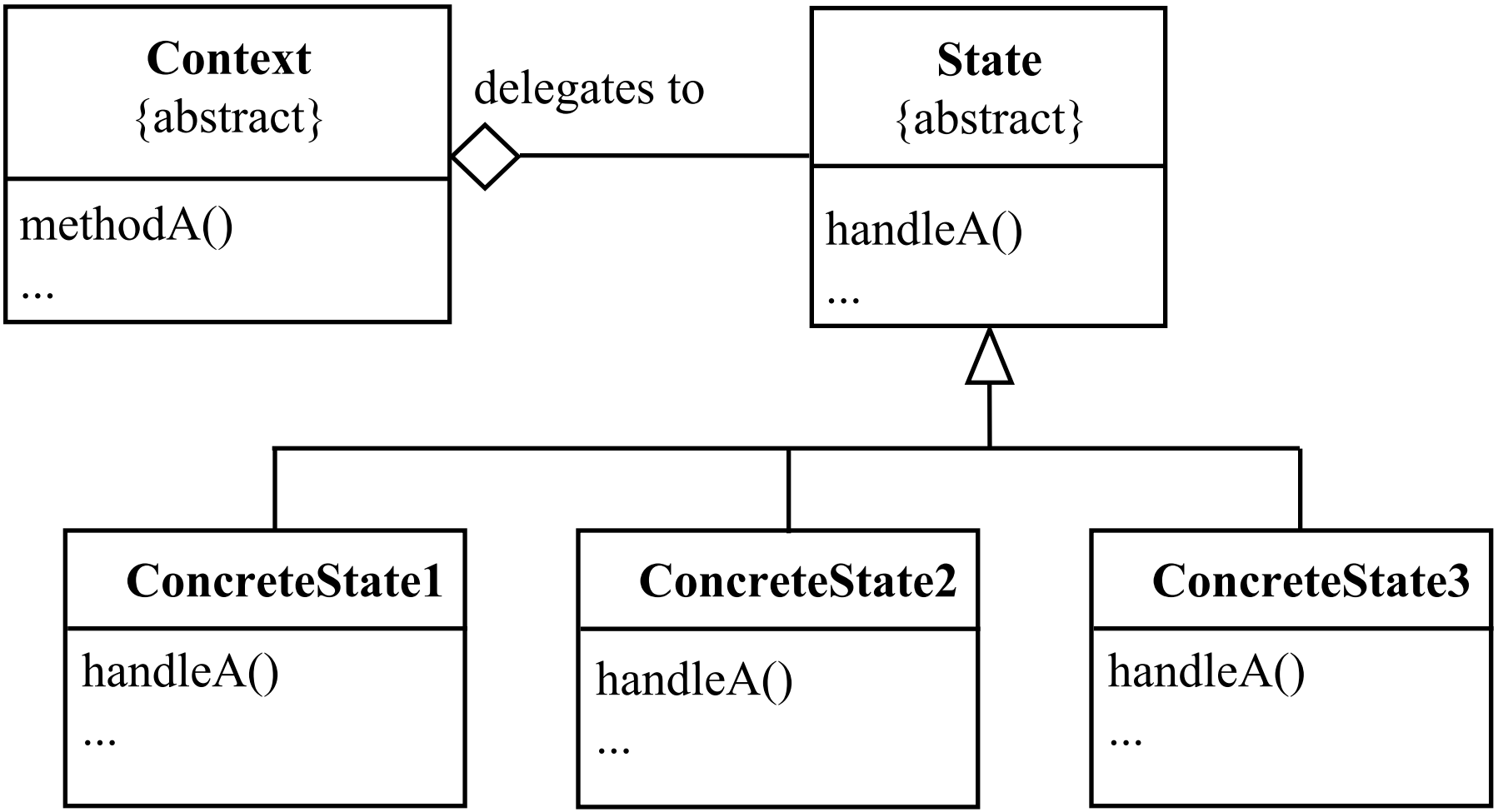
Example – A Network Transport Connection

- A network transport connection can be in one of several states (established, closed, listening, sending,…) based on the actions that have occurred
- Let NetworkConnection be a class that represents the connection
- An instance of NetworkConnection must respond differently depending on the state it is in

The Network Transport Connection Model



The Generic Model for the State Pattern



Sample Code – Class `NetworkConnection`

```
public class NetworkConnection {
  public void open() {
    // delegate this one to the
    // current state object
    currentState.open(this);
  }
  public void close(){} // ditto
  public void send(){}  // ditto
  // changeState has package scope
  void changeState(ConnectionState newState) {}
  private ConnectionState currentState =
                ConnClosed.instance();
}
```

Sample Code – Class `ConnectionState`

```
public class ConnectionState {
  public void open(NetworkConnection connection){}
  public void close(NetworkConnection connection){}
  public void send(NetworkConnection connection){}
  // ...
  protected void changeState(
        NetworkConnection connection,
        ConnectionState newState) {
    connection.changeState(newState);
  }
}
```

Sample Code – Class `ConnClosed`

```
// Subclasses of ConnectionState implement
// the 'state-dependent' behavior
// Each is also 'stateless' and is implemented
// as a Singleton object
class ConnClosed extends ConnectionState {

  public static ConnectionState instance(){
    if (connInstance == null) {
        connInstance = new ConnClosed();
    }
    return connInstance;
  }
```

Sample Code – Class `ConnClosed – 2`

```
public void open(NetworkConnection connection){
    // do a handshake
    // ...
    // then change state
    changeState(connection,
                ConnEstablished.instance());
  }
  // other methods are implemented similarly
  public void close(NetworkConnection connection){}
  public void send(NetworkConnection connection){}
  // ...
  private static ConnectionState connInstance = null;
}
```

When to Use the State Pattern

- Use the State Pattern when an object must respond differently to the same request based on the state it is in

Tradeoffs and Consequences

- The `State` class hierarchy allows you to encapsulate the varying behavior of methods on the `Context` object without writing state-dependent switch statements
- The `Context` object appears to change state, although internally it's just switching instances of subclasses of the `State` class
- The cost of the flexibility comes at the expense of one additional level of indirection on each request. This can be faster than relying on a table-driven approach

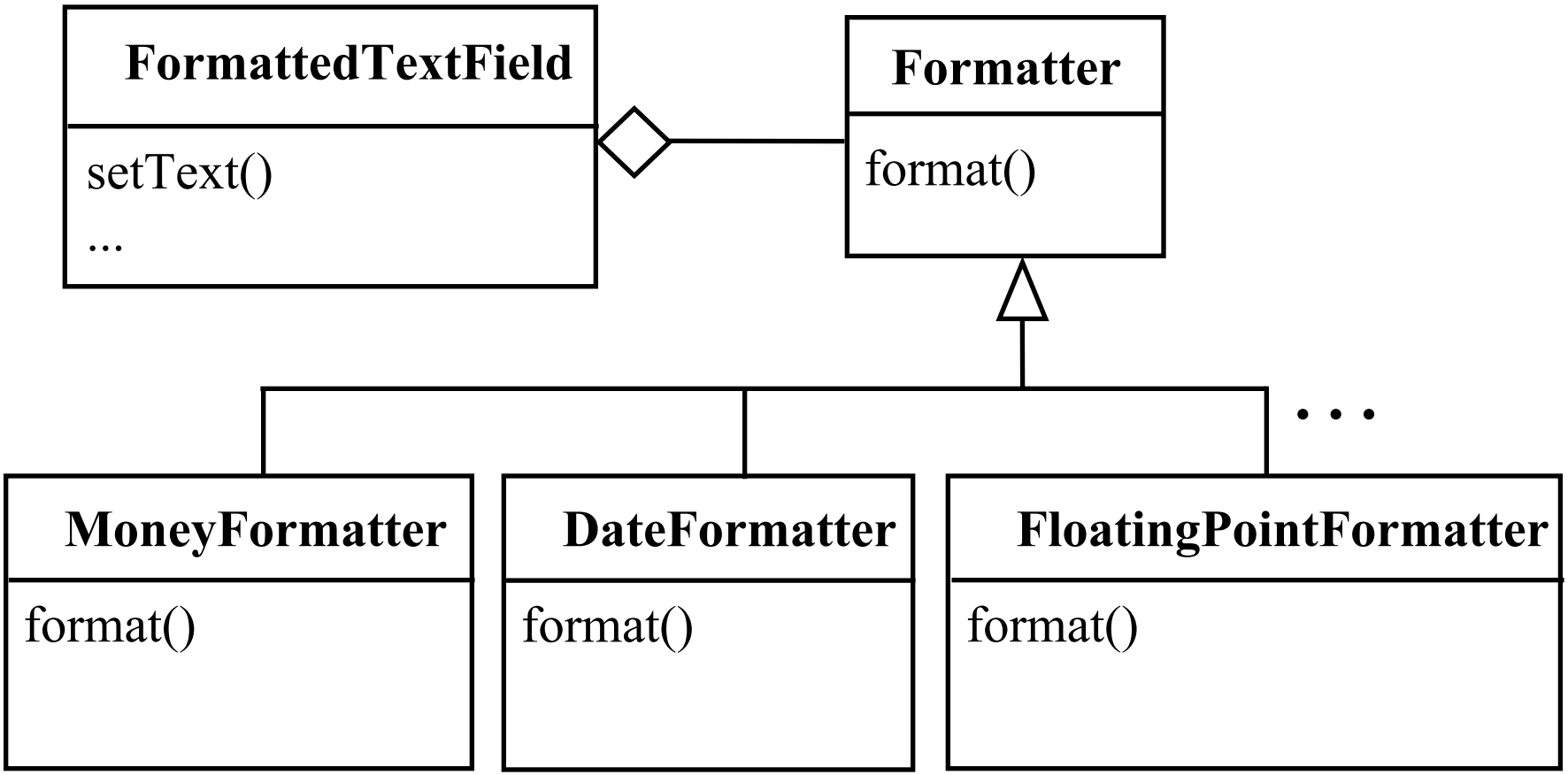
The Strategy Pattern

- The Problem: We want to define and use a family of algorithms as if they were interchangeable

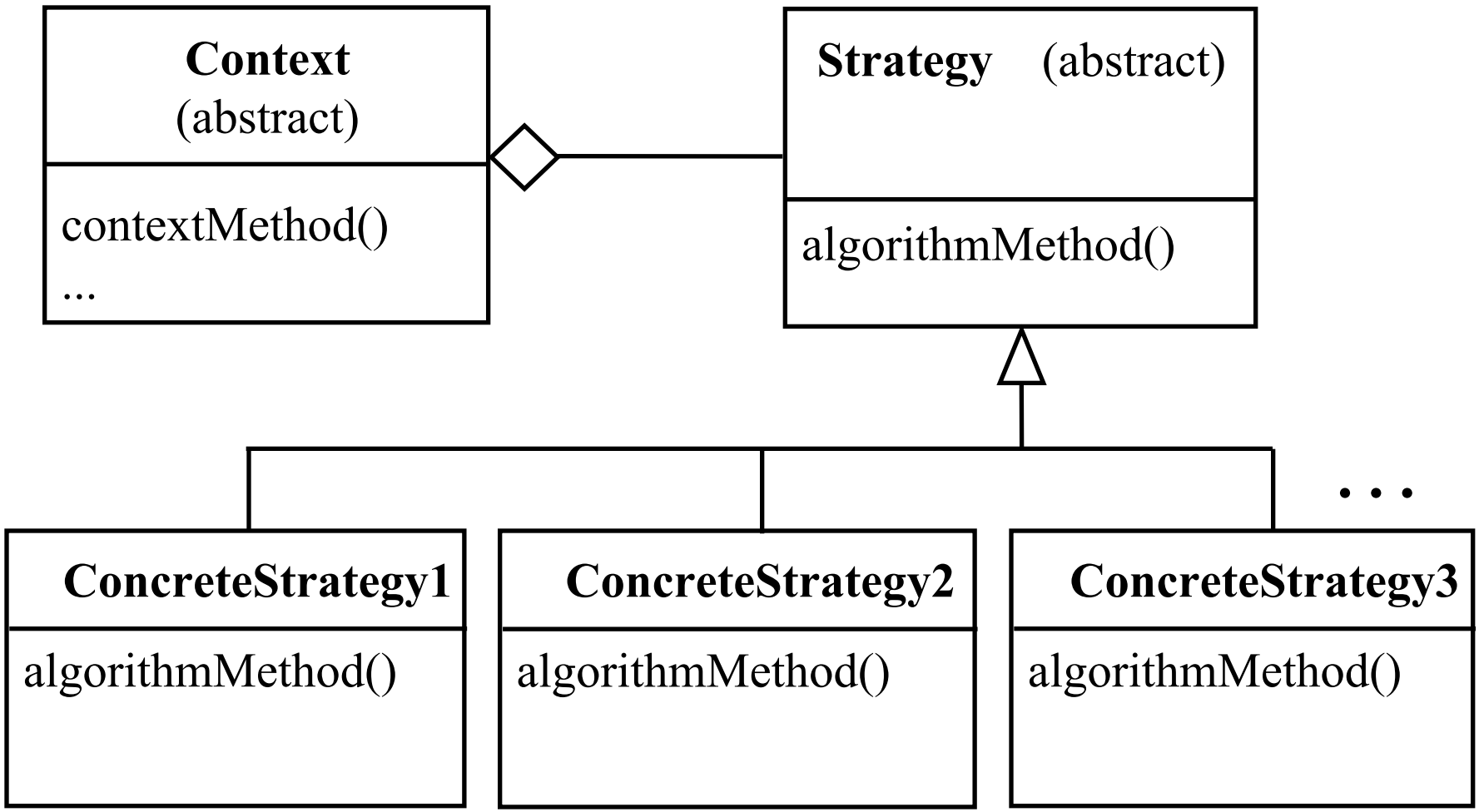
Example – Formatting Fields for Display

- In displaying text in fields on a screen, we often have multiple options for formatting the data – for instance, we need to represent text such as dates, money, floating point numbers, and so on, in multiple formats
- We could create a hierarchy of `TextField` classes for the different formatting options
- It's simpler to encapsulate each formatting algorithm in its own formatting class and have a `TextField` object own a formatting object

Example – Formatting Fields for Display



The Generic Model for the Strategy Pattern



Sample Code – The Base `Formatter` Class

```
package cscie97.formatter;

public class Formatter {
  public String format(String str) throws FormattingException
  {
    // this just returns what it's given
    return str;
  }
}
```

Sample Code – Class `FormattedTextField`

```
public class FormattedTextField extends JTextField
    implements ActionListener {
    public FormattedTextField() {
    }

    protected String originalData;
    private Formatter formatterObj;


    public FormattedTextField(Formatter f) {
        formatterObj = f;
        originalData = "";
        super.addActionListener(this);
    }
```

FormattedTextField – 2

```
public FormattedTextField(Formatter f, String str) throws
     FormattingException {
     formatterObj = f;
     super.setText(formatterObj.format(str));
     super.addActionListener(this);
     originalData = str;
 }
 public void setText(String str) {
     try {
         super.setText(formatterObj.format(str));
         originalData = str;
     }
     catch (FormattingException fe) {
         /**
          * there's nothing to see here
          */
     }
 }
```

FormattedTextField – 3

```
/**
 * For this one, we have to handle the exception here.
 * If the try block succeeds, we set the text to the newly
 * formatted string, and set the original data to that string
 * If the try block fails, we revert to originalData
 */
 public void actionPerformed(ActionEvent e) {
        String newString = "";
        String input = getText();
        try {
            newString = formatterObj.format(input);
            super.setText(newString);
            originalData = input;
        }
        catch (FormattingException fe) {
             super.setText(originalData);
 }
} // end class FormattedTextField
```

DateFormatter – 1

```
package cscie97.formatter;

import java.text.*;
import java.util.Date;

class DateFormatter
  extends Formatter {
  public DateFormatter() {
  }

  public DateFormatter(String formatString) {
    formatPattern = formatString;
  }
 // the formatPattern String is like "yyyymmdd"
  private String formatPattern;
```

DateFormatter – 2

```
  public String format(String str) throws FormattingException {
    SimpleDateFormat formatter
      = new SimpleDateFormat(formatPattern);
    DateFormat df =
      DateFormat.getDateInstance(DateFormat.SHORT);
    String contents = str;
    String dateString = "";
    try {
      Date datetime = df.parse(contents);
      dateString = formatter.format(datetime);
    }
    catch (ParseException pe) {
      throw new FormattingException("bad date format", str,
   "mm/dd/yyyy");
    }
    return dateString;
  }
} // end DateFormatter
```

Sample Code – Class `PercentFormatter`

```
package cscie97.formatter;
public class PercentFormatter extends Formatter {
    public PercentFormatter() { }
    public String format(String str) throws FormattingException {
      String returnString = str;
    // check if input is really a number
    try {
      Double classValue = new Double(returnString);
      double value = classValue.doubleValue() * 100;
      returnString =
        (new Double(value)).toString() + "%";
    }
    catch (NumberFormatException nfe) {
     throw new FormattingException("invalid number format", str,"
   n.m");
    }
    return returnString;
  }
}
```

Sample Code – Applying the Formatters – 1

```
package cscie97.formatter;
import java.awt.*;
import javax.swing.*;

public class FormatTest
    extends JFrame {
    public FormatTest() { }

    public static void main(String args[]) {
        /**
         * The following code fragment shows how we can
         * add three FormattedTextFields, each with a
         * different Formatter object
         */
        FormatTest ft = new FormatTest();
        Container contentPane = ft.getContentPane();
        contentPane.setLayout(new GridLayout(3, 2, 5, 5));
```

Sample Code – Applying the Formatters – 2

```
// First add a field with a HalloweenFormatter
    Formatter testFormatter = new HalloweenFormatter();
    FormattedTextField fTF1 = new FormattedTextField(testFormatter);
    fTF1.setText("abcd");
    contentPane.add(new Label("Halloween format"));
    contentPane.add(fTF1);

    // Next add a field with a DateFormatter
    Formatter testDateFormatter =
        new DateFormatter("yyyy/MM/dd G 'at' hh:mm:ss a zzz");
    try {
        FormattedTextField fTF2 = new FormattedTextField(
        testDateFormatter, "10/31/2009");
        contentPane.add(new Label("Date format"));
        contentPane.add(fTF2);
    catch (FormattingException fe){
      // do something useful for the user
    }
```

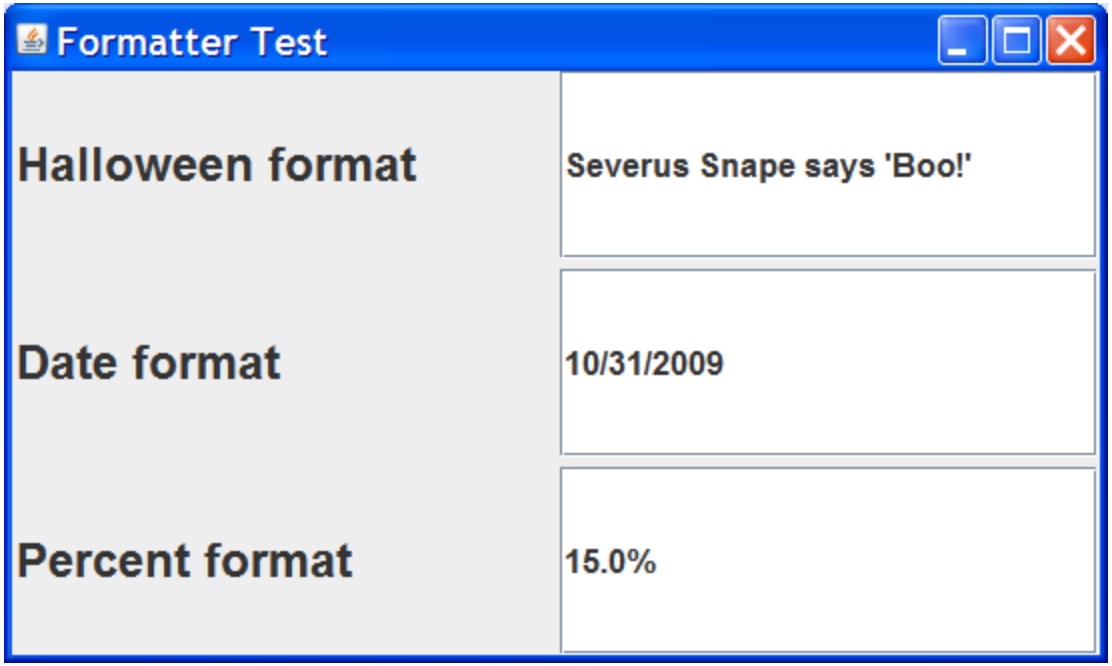
Sample Code – Applying the Formatters – 3

```
  try {
      // Finally add a field with a PercentFormatter
      FormattedTextField fTF3 = new FormattedTextField(
        new PercentFormatter(), ".15");
      contentPane.add(new Label("Percent format"));
      contentPane.add(fTF3);
    }

    catch (FormattingException fe) {
      // do something useful for the user
    }

    ft.setSize(400, 200);
    ft.show();
  }
}
```

The Resulting Screen



When to Use the Strategy Pattern

- Use the Strategy Pattern when
  - You have many variants of the same algorithm
  - You want to encapsulate the algorithm-specific data in the algorithm, not the structure to which the algorithm applies

Tradeoffs and Consequences

- The Strategy Pattern encapsulates each algorithm in its own class
- The algorithm is invoked by a client via the same method name no matter what subclass encapsulates an algorithm. This allows the algorithm to vary independently of the clients that use it
- Clients typically create a `Strategy` object and pass it to the object containing the contextual data on which it operates

Comparing Command, State, and Strategy

- All three of these patterns encapsulate algorithms
- Command can be used to pass data and an algorithm around together for execution at an appropriate time. Execution is typically done by a generic run-time component
- State is used to provide different implementations of the same object (ironically, via stateless classes)
- Strategy allows you to substitute algorithms that solve the same problem with one another

Comparing Command, State, and Strategy – 2

- The thing that distinguishes these patterns is the concept that is varying
- For Command, the objects vary according to the data and the methods acting on that data
- For State, the implementation is typically stateless, and the methods vary according to the 'logical state' of the objects
- For Strategy, it's the set of algorithms that solve the same problem on the same data with different approaches