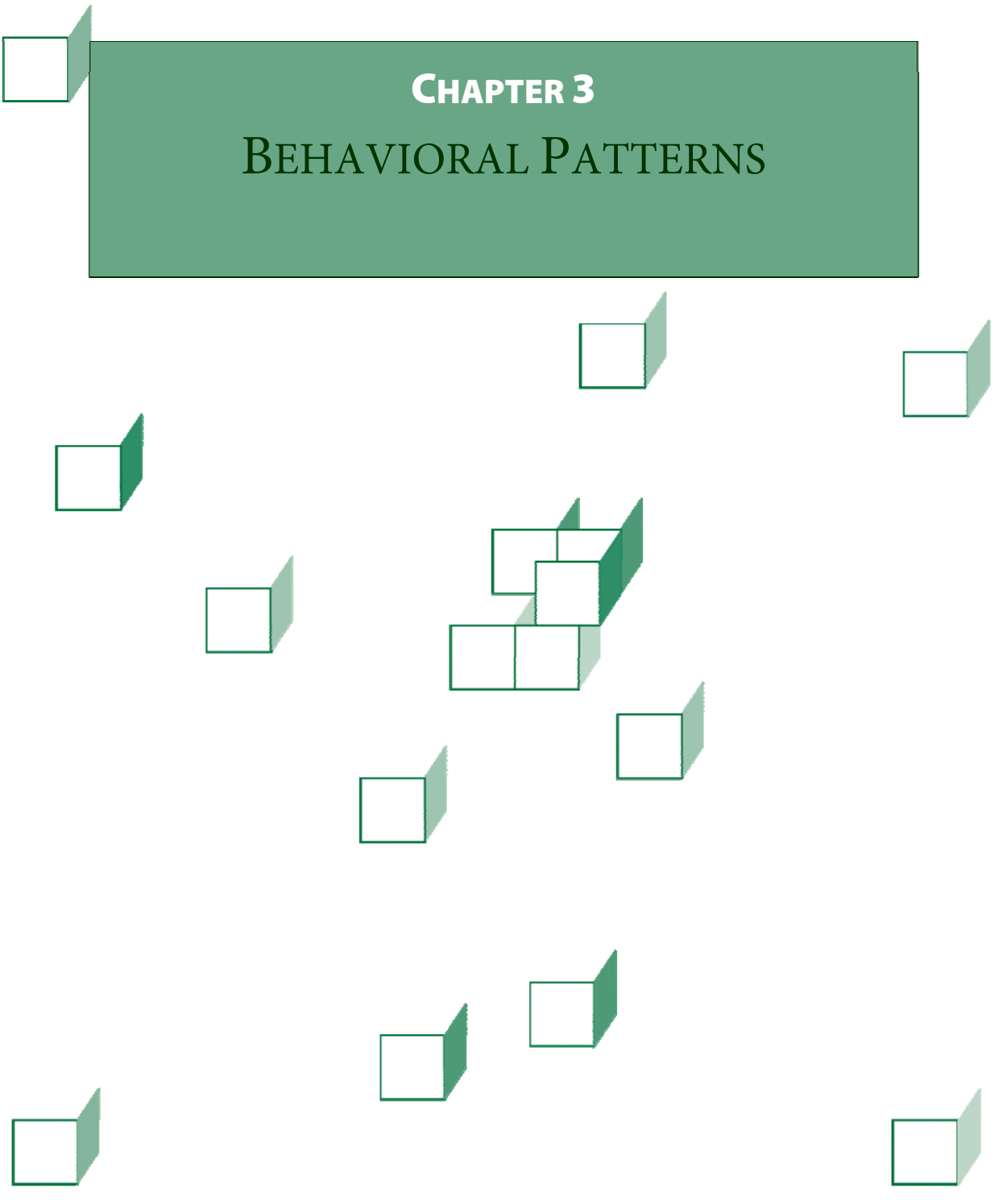
Chapter 3

Behavioral Patterns



OBJECTIVES

After completing this unit you will be able to recognize and apply the following patterns in designing Java software:

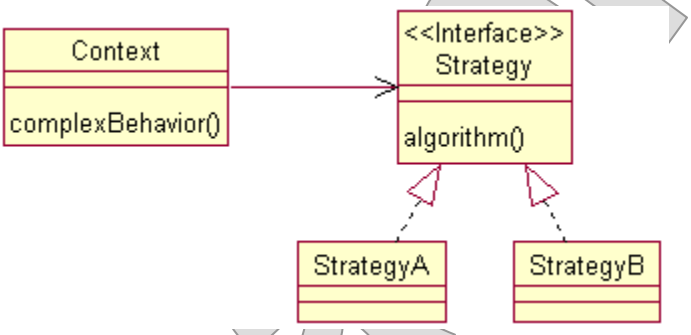
- Strategy
- Template Method
- Observer
- Model/View/Controller
- Command
- Chain of Responsibility

Gang-of-Four behavioral patterns not explicitly covered in this course are:

- Interpreter
- Iterator
- Mediator
- Memento
- State
- Visitor

The Strategy Pattern

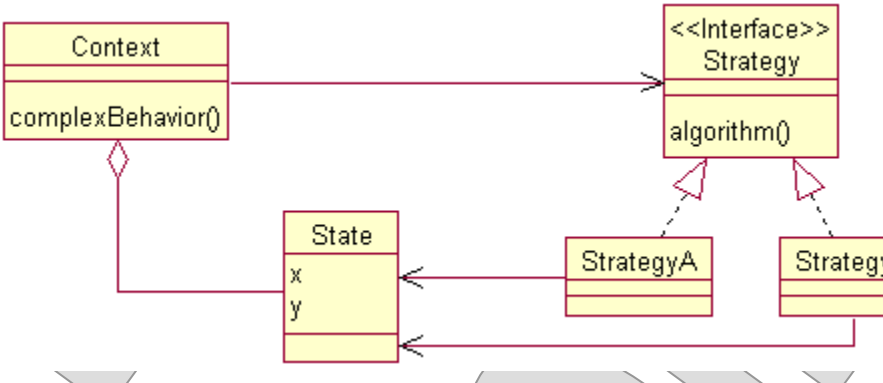
- The **Strategy** pattern addresses the problem of varying behavior required by what is initially considered a single encapsulation.



- Strategy is so small and simple that it may seem like just a part of ordinary OO design.
  - It is one example of a philosophy that **favors delegation over inheritance** for certain designs.
- But there are interesting questions – strategies for expressing Strategy, if you will.

The Strategy Pattern

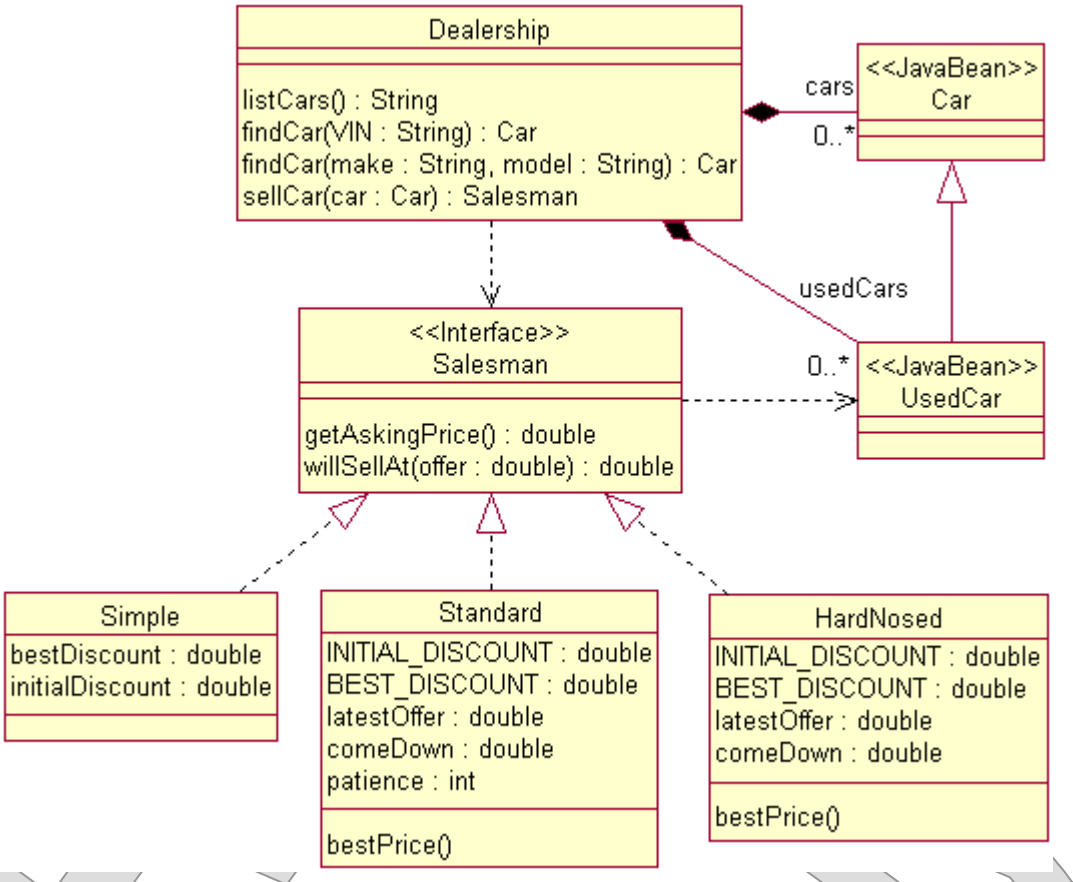
- **Who controls the choice of strategy?**
  - Can a client plug in a strategy of its choosing?
  - Does the context object decide for itself?
  - Hmm … perhaps a **factory** makes the decision in assembling the system.
- **How can the context object share useful state information with a Strategy delegate?**
  - This might require parameters to the algorithm(s).
  - Or the strategy might be stateful, informed of context state when it is created, or whenever context state changes.
  - Context and Strategy objects could share a state object:



Car Sales as a Strategy

EXAMPLE

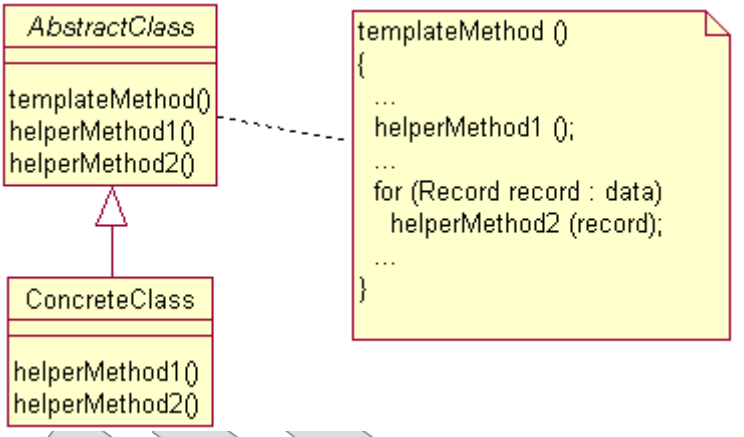
- In **Examples\Cars**, the act of selling a car is broken out of the **Dealership** class as a separate strategy.



  - The **Salesman** interface expresses the abstract strategy.
  - Various salesmen take different approaches to the negotiation, and will arrive at different results: sale or no sale, and different prices.
  - Note that this is an example of sharing a state object – the **Car** – with the strategy, but only by passing it as a parameter to the algorithm.

The Template Method Pattern

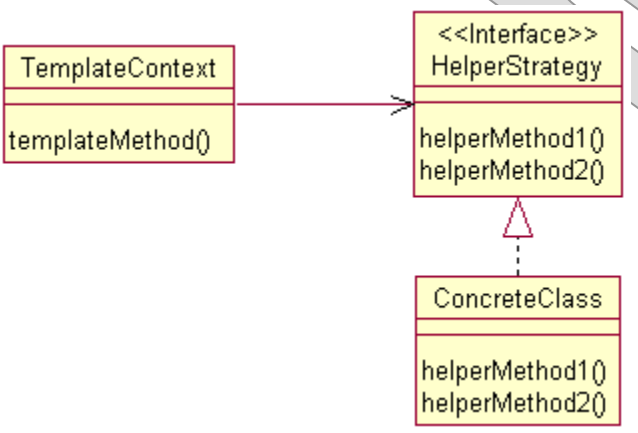
- **Another simple pattern is Template Method, which addresses the need to specialize how some parts of a general and more complex task are carried out.**



  - This is actually one of the most common reasons for creating an abstract class, because there are both general and special parts of the total implementation, and one must be implemented in terms of the other.

The Template Method Pattern

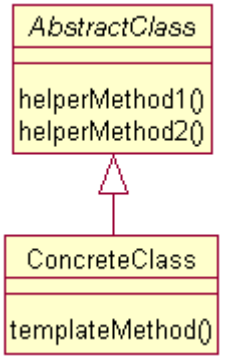
- **Note that any Template Method solution could be refactored as a Strategy:**



  - The choice between the two is a subtle thing.
  - Quoting the gang of four: "Template methods use inheritance to vary part of an algorithm. Strategy uses delegation to vary the entire algorithm."
  - Also, significant sharing of state during the task may argue for a Template Method solution, since inheritance allows for one object and there is no complexity to sharing state between methods of derived and base classes.

The Template Method Pattern

- **Another approach – and a warning sign for this pattern – is for the base class to provide helper methods for the generic parts of a complex algorithm.**
  - Such a class allows the derived class to provide the specialized implementation of the algorithm.
  - But it also requires the derived class to coordinate the task.
  - Where the task is sufficiently complex – even in how general and special parts are coordinated – a Template Method captures this coordination and saves the derived classes from having to duplicate top-level logic.

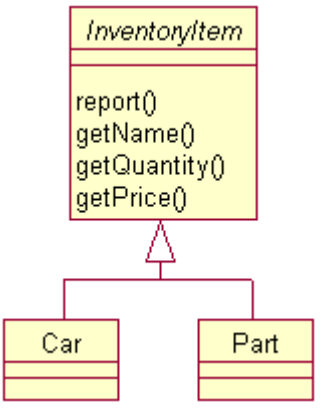


- **As to pitfalls: generally speaking, this is a pattern that is easily overused.**
  - Beware of **over-parameterizing** an algorithm, to the point that it's not really the same algorithm any longer, and a different solution is indicated. Coherent parts of a single, well-conceived task are one thing – a proliferation of abstract methods **preProcess**, **postProcess**, **thisHook**, **thatHook**, and **produceOptionalSpecialSection** is another!
  - Don't use abstract methods just to fetch derived-class state; use private fields and protected accessors and mutators on the base class, unless the state really must be stored or derived in different ways by different subtypes.

Report Template Method

EXAMPLE

- In **Examples\Cars**, the **InventoryItem** class defines a (very simple) template method **report.**
  - This is implemented in terms of helper methods **getName**, **getQuantity**, and **getPrice**:




```
public String report ()
{
  return String.format
    ("%-28s%8d    %,10.2f    %,10.2f%n",
      getName (), getQuantity (), getPrice (),
        getQuantity () * getPrice ());
}
```

- Derived classes provide the needed variations:
  - A **Part** has a simple name, quantity, and price.
  - A **Car** synthesizes a name from year, make and model, and its quantity is always one.
- Note that these are essentially state elements, but that it is sensible to implement them as abstract methods since they are derived differently by **Car** and **Part**.

Health Information Request

LAB 3A

In this lab you will refactor the code for a base/derived class pair that collaborate to write an email message requesting health information for a patient. The starter code exhibits clear warning signs for the Template Method pattern. You will rework the code to implement a Template Method.

Detailed instructions are contained in the Lab 3A write-up at the end of the chapter.

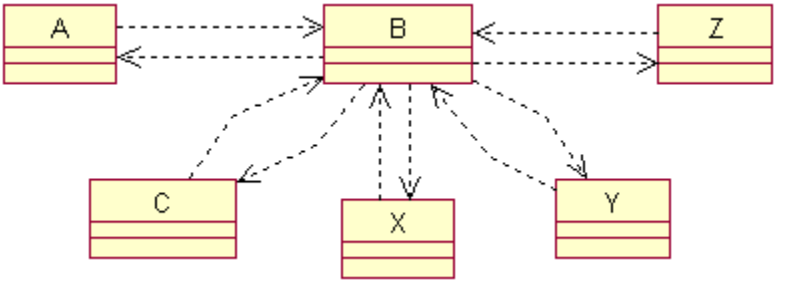
Suggested time: 45 minutes.

The Observer Pattern

- The **Observer** pattern addresses the problem of at least two objects, one of which wants to observe the activity of another.
- The term **activity** is important – it implies that the observed object is either busy doing something or having something done to it.
- The observing object cannot predict the timing of changes to the observed object's state.
- How to handle this unpredictability?
- There are at least two possible strategies, each of which is really a warning sign for this pattern:
  - A could call B repeatedly, in a **polling loop**. This is terribly inefficient and can cost the whole application – and others on the same host – in available processing power.

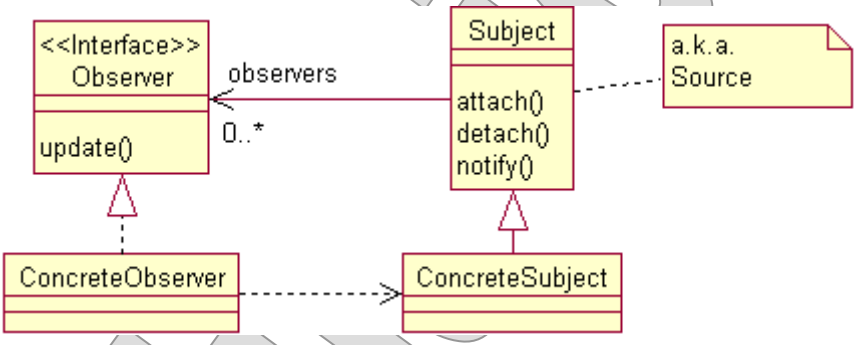
```
while (!download.isComplete ())
  progressBar.setValue (download.pctComplete ());
```

  - B could call A, but this develops a dependency of B on A, resulting in a **bidirectional dependency**, which scales badly as additional classes want to observe B's activity:



The Observer Pattern

- The solution lies in abstracting the two roles:
  - **Subject**, which can **attach** and **detach** observers – this was B
  - **Observer**, which has at least one method that can be called by the subject to advise of some interesting activity – was A



- This **decouples** the actual participants, **ConcreteSubject** and **ConcreteObserver**, which can interact with no mutual dependency.
  - That is, there is a dependency from observer to subject at the **concrete** level, but the dependency in the other direction is **abstracted** so that the concrete subject needn't know about any particular concrete observer.
  - It's also possible that another entity, at some higher level, registers one object as an observer on another; this further decouples the two concrete objects.
  - If there are many interested parties, they are each different **ConcreteObservers**, and the **ConcreteSubject** is interested only in the abstraction which is the **Observer** interface.

The Observer Pattern

- **Examples of Observer abound in the J2SE Core API:**
  - AWT/JFC **event handling** and **image and resource loading**
  - **SAX parsing** and error/warning notifications, and **DOM events** that can advise of changes to XML content in memory
  - **User preferences** – one can listen for changes
  - The **sound API**
- **AWT and JFC implement the Java event model, which is a variation on the classic Observer pattern.**
  - Observers are called **listeners** and are named by a convention **XXXListener**. Their methods are all of the form

```
public void somethingHappened (XXXEvent ev);
```

  - The information about the event itself is encapsulated in a class **XXXEvent**. Thus in the Java event model state information is always pushed to the listener.
  - Subjects are called **event sources**, and rather than implementing an interface type they are recognized by offering methods of two standard signatures

```
public void addXXXListener (XXXListener lstnr);
public void removeXXXListener (XXXListener lstnr);
```

- **A pitfall: don't expect reliable sequencing, or even serialization of calls to multiple observers!**
  - See Chain of Responsibility later in this chapter.

The Observer Pattern

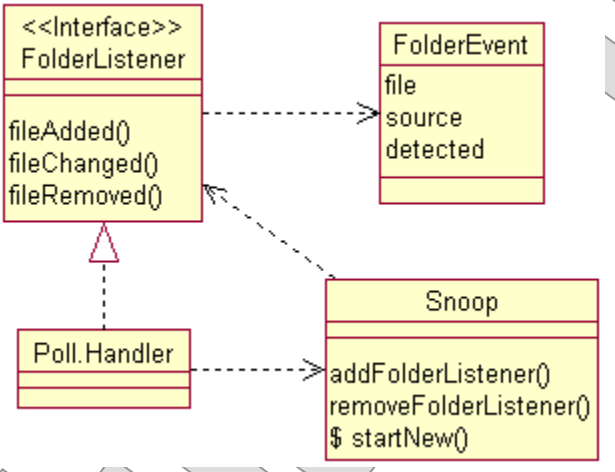
- **Another important pitfall in implementing Observer has to do with thread safety.**
  - Observers should act quickly when called; don't tie up the calling thread, which may have many heavy things to do and certainly is supposed to move along to notify other observers.
  - Subjects must guard against race conditions over their list of observers – such as when one thread is notifying observers and another is busy removing an observer from the list!

```
public synchronized void register (Observer o)
{
  observers.add (o);
}

public synchronized void remove (Observer o)
{
  observers.remove (o);
}

protected void notify ()
{
  List<Observer> local = null;
  synchronized (this)
  {
    Local = new ArrayList<Observer> (observers);
  }

  for (Observer o : local)
    o.update ();
}

private List<Observer> observers =
  new LinkedList<Observer> ();
```

An Observer

EXAMPLE

- In **Examples\Polling** is a component that can detect changes to the contents of a folder in the file system.



- The class **Snoop** actually polls the file system actively, which is the opposite of what an Observer would do.
  - But this is only necessary since we're dealing with the (non-OO, non-pattern-aware) native file system as a resource.
  - **Snoop** takes this polling behavior off the client's hands, and, while not an observer, is itself an observable subject.
  - It offers to register **FolderListener**s, and will notify them of additions, changes, or removals from the subject folder.
- **Poll** is the client class, which implements the observer interface in an inner class **Handler**.
  - Thus all the work is done by the component and the handler; the **main** method just wires up components to subject folders and its handler, and lets them all run.

Primes

LAB 3B

In this lab you will add an Observer system to a component that finds prime numbers. This will allow different listeners to provide updated output and progress indications. You will then confront threading issues in the Observer system, and fix them with appropriate synchronizations of code in the core component.

Detailed instructions are contained in the Lab 3B write-up at the end of the chapter.

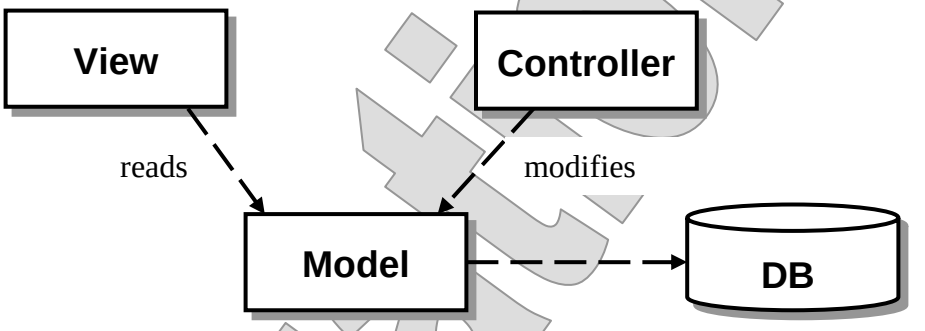
Suggested time: 45-60 minutes.

The Model/View/Controller Pattern

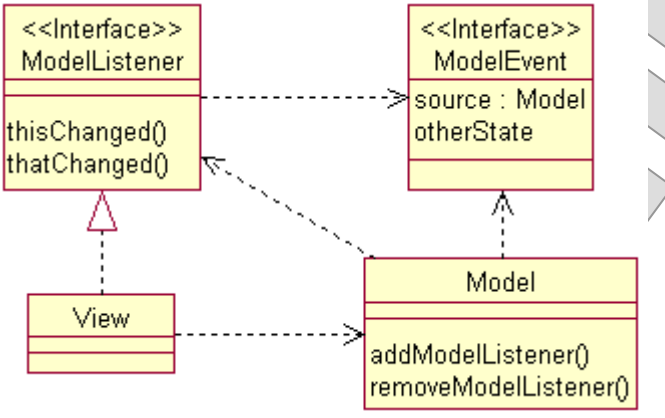
- **The one pattern in this course that is not defined in the GoF text is Model/View/Controller, or MVC.**
- **MVC originated in the Smalltalk platform.**
- **It addresses the problem of organizing an application's state and behavior in a way that**
  - **Centralizes the state** information
  - Defines **clear channels for changing** that information
  - Allows **multiple readers or views** of that information to stay **synchronized** to any changes
- **Warning signs can be subtle when reviewing source code, but are often obvious and all too familiar from a user or QA perspective:**
  - Multiple **views** of application state seem to **go stale** or fall out-of-date with changes made elsewhere. For instance the user adds an appointment to his or her book in a detail view, but a graphical calendar view doesn't show the new item. The classic user frustration is having to close and re-open a window, or otherwise "poke" or "shake" the application until it seems to catch up with itself.
  - The views stay in sync, but this is managed by code in **one GUI component** making a change to application state and then **explicitly notifying** other GUI components so that they can refresh their presentations. This can work cleanly but is not a scalable solution.

The Model/View/Controller Pattern

- The MVC solution is to recognize three roles **model**, **view**, and **controller**, to assign these roles to various classes, and to set constraints on their interactions:



  - The **model** encapsulates state information, and is a source for model **events**. It may manage state from a relational database, or transient state in memory, or both.
  - The **controller** does not hold state, but can act upon the model to change it.
  - The **view** does not hold state, but can show it by reading model information; it is also an **observer** on the model:



The Model/View/Controller Pattern

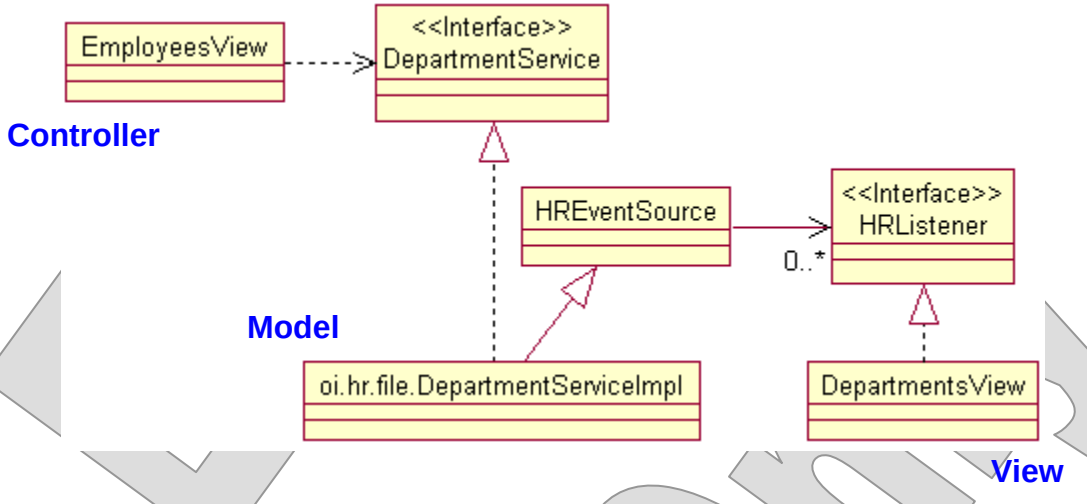
- **Examples of MVC live at different levels of the Core API and common application architectures.**
- **Many JFC controls have internal MVC systems.**
  - Consider **JList** and **ListSelectionModel**.
  - Some JFC controls reduce the pattern to a two-role version called the **model/view split**.
- **In Web application development, MVC appears at a larger scale:**
  - One component might be responsible for handling an incoming HTTP request and controlling the model.
  - Another might be responsible for shaping up the next view that the user sees as an HTML page, returned in the HTTP response.
- **In the HR case study, certain service implementations act as models, representing information to controllers and views through their APIs and also firing events so that views can stay in sync.**

An MVC System

EXAMPLE

- The application in **Examples\HR\Step1** implements an MVC system based on the **HRListener** interface:

```
public interface HRListener
{
  ...
  public void employeeReassigned
    (EmployeeDigest employee,
      DepartmentDigest newDepartment,
      DepartmentDigest oldDepartment);
  public void employeeChanged
    (EmployeeDigest employee);
  ...
}
```

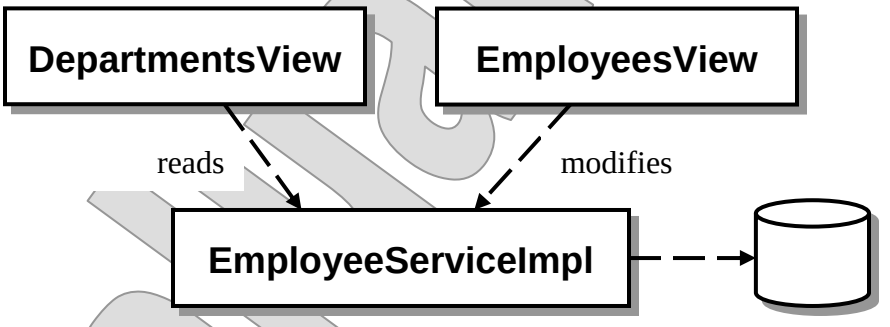


  - The model consists of the service implementations, whose mutators fire HR events by calling one or more methods on registered **HRListener**s.
  - The three view classes act as **controllers** when they call those mutators, and as **views** by implementing **HRListener** themselves.

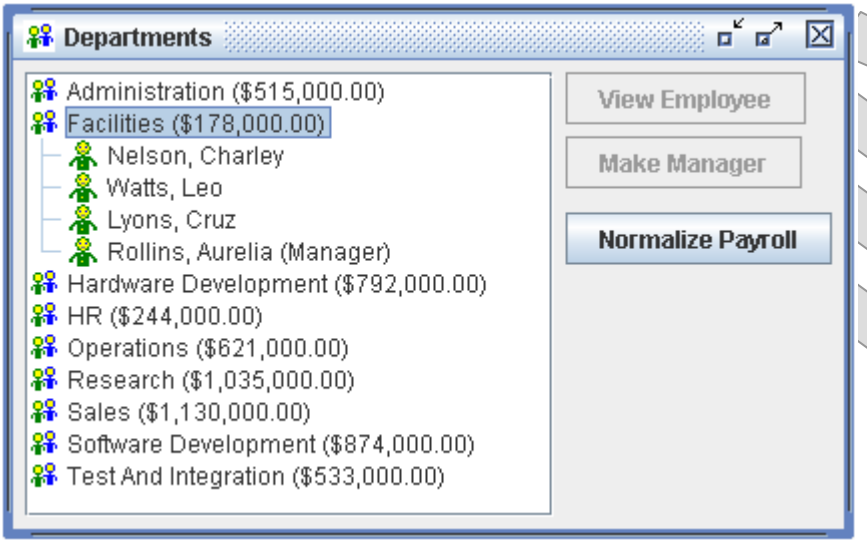
An MVC System

EXAMPLE

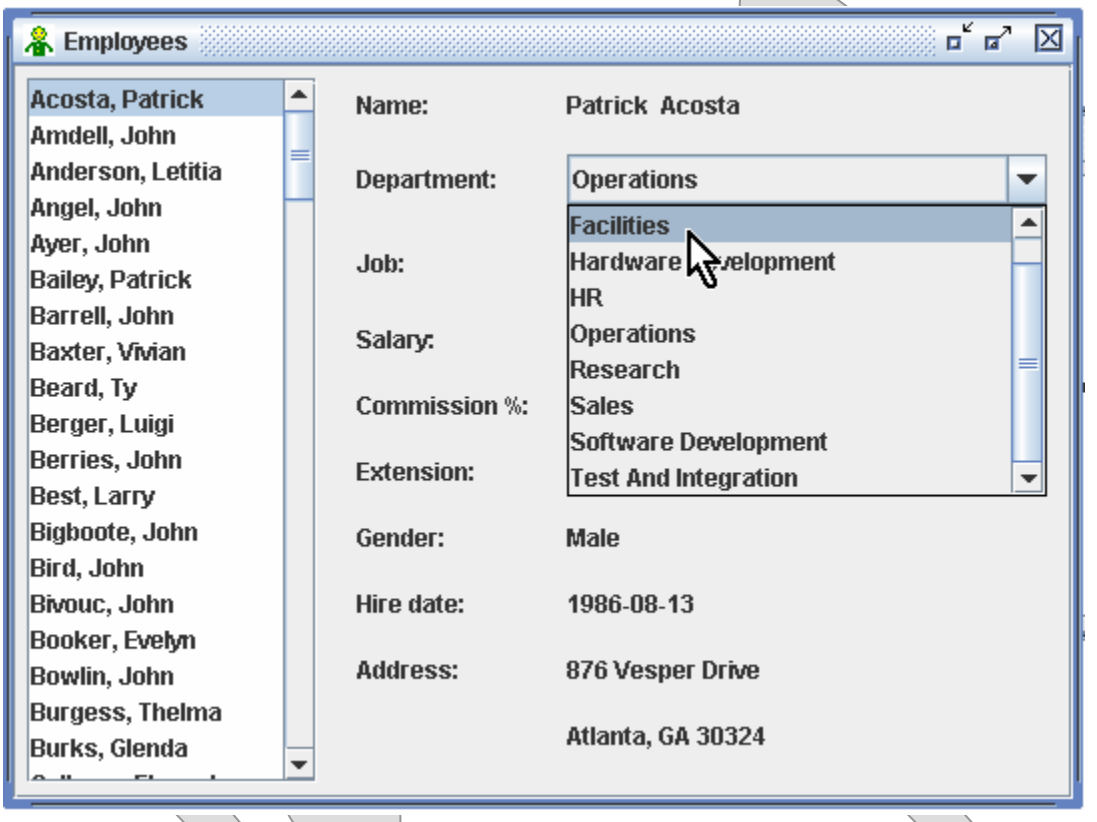
- This is how **DepartmentsView** automatically updates its tree when an employee is assigned to a new department from the **EmployeesView**.
  - MVC allows this state information to be centralized in the model, which in turn means that the controller (**EmployeesView** in this case) and view (**DepartmentsView**) needn't have an intimate relationship themselves.



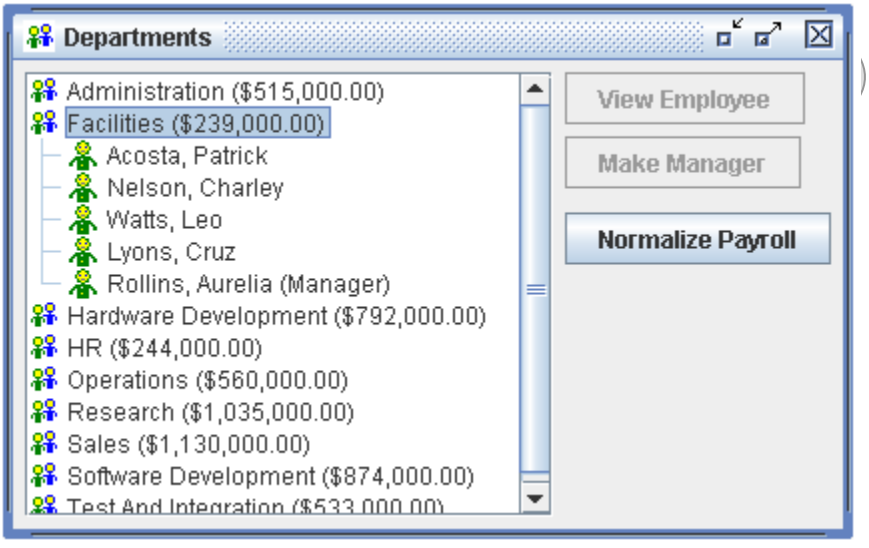
  - Try it! **View|Departments** and open Facilities …



– **View|Employees**, and assign the first employee to Facilities:



– You'll notice the change in the open Departments window:



An MVC System

EXAMPLE

- **Let's follow the processing path through source code, to see how this synchronization is achieved.**

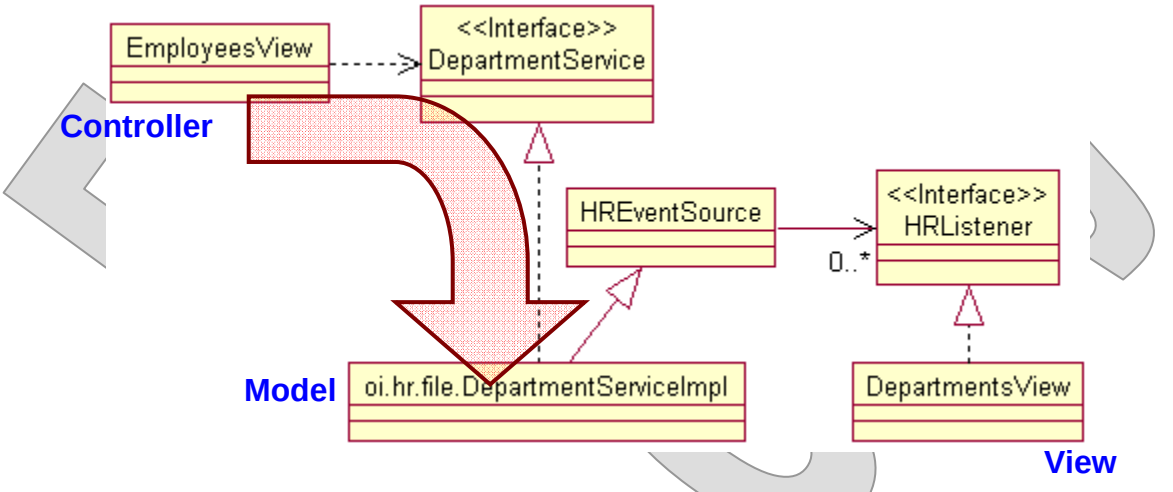
1. **cc.hr.gui.EmployeesView** holds an instance of an inner class **EmployeesPanel** as its only control. This class, in its constructor, connects an event handler to its combo-box of departments:

```
public class EmployeesView
->public static class EmployeesPanel
  ->public EmployeesPanel ()
    {
      cbDepartment.addItemListener
        (new DepartmentHandler ());
    }
```

2. Thus when the user chooses a new department for a selected employee in the Employees window, the method **itemStateChanged** in **DepartmentHandler** (itself an inner class of **EmployeesPanel**) does a bit of error checking, and then calls **DepartmentService.assignEmployee** to make the reassignment.

```
public class EmployeesView
->public static class EmployeesPanel
  ->protected class DepartmentHandler
    ->public void itemStateChanged (ItemEvent ev)
      {
        dService.assignEmployee
          (selectedDept, selectedEmployee);
      }
```

3. This then is the **controller** writing to the **model** – though the controller in this system is also an observer on a JFC control.



   – The warning-sign behavior here would have been an immediate call directly to the **DepartmentsView**, telling it to refresh itself.

4. The service implementation in play, which is **cc.hr.file.DepartmentServiceImpl**, carries out the changes to the model – but it also **fires a model event**:

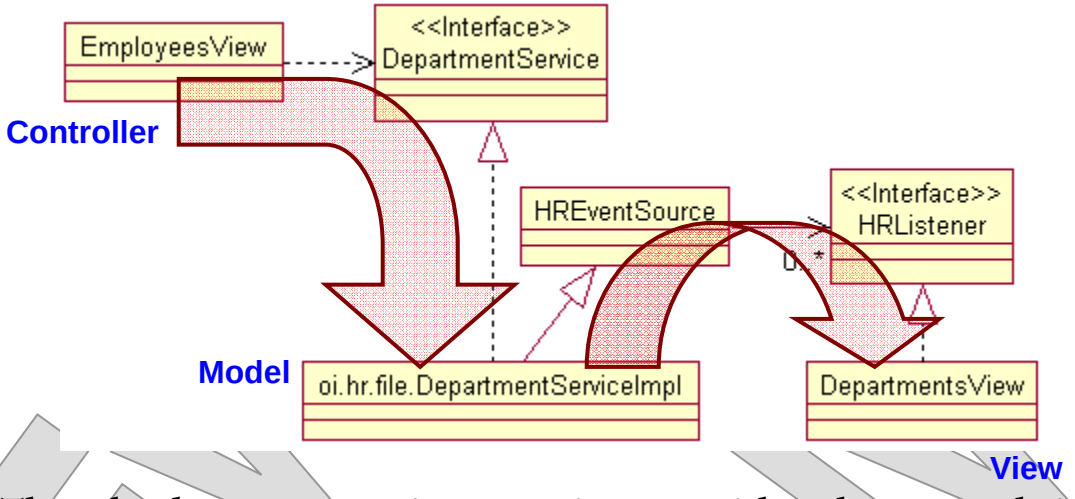
```
public class DepartmentServiceImpl
->public void assignEmployee (...)
  {
    ...
    pDept.addEmployee (pEmp);

    for (HRListener recipient : getListeners ())
      recipient.employeeReassigned
        (pEmp, pDept, oldDepartment);
  }
```

5. Who are the listeners for model events on the department service? Well, if there is a department window open at the moment, it will be one of those listeners: it takes a reference to the department service in its constructor, and hooks its own lifecycle methods to assure that it is attached as an observer for any model events. See **cc.hr.gui.DepartmentsView**:

```
public class DepartmentsView
->public DepartmentsView (...)
  {
    addInternalFrameListener
      (new WindowHandler (source));
  }

public class DepartmentsView
->protected class WindowHandler
  ->public @Override void internalFrameOpened (...)
    {
      source.addHRListener (mainPanel);
    }
```

6. So when the model fires an event, this view is listening! and immediately updates its display as necessary to reflect the change:

```
public class DepartmentsView
->public static class DepartmentsPanel
  ->public void employeeReassigned (...)
    // Removes the employee as a tree node from
    // his or her old department and adds to the
    // new department ...
```



- **Thus the departments view stays in sync with a change made in the employees view.**
- **The system is centralized such that all changes flow "down" to the model and back "up" to views that care about certain types of changes.**
- **However, the system is not complete!**
  - In the upcoming lab you'll observe a warning sign for MVC that indicates that one of the views has not yet subscribed for model events, and so is missing out on real-time changes.

MVC in the HR Application

LAB 3C

Optional

In this lab you will implement the **HRListener** interface for **EmployeesView**, and register it as a listener for HR events with the relevant service objects. This will connect it to the broader MVC system already in place for the other two views (and in which it is already acting as a controller as well).

Detailed instructions are contained in the Lab 3C write-up at the end of the chapter.

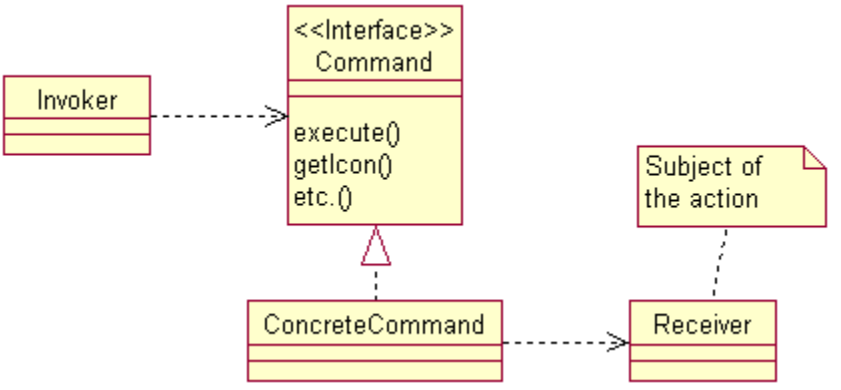
Suggested time: 45-75 minutes.

The Command Pattern

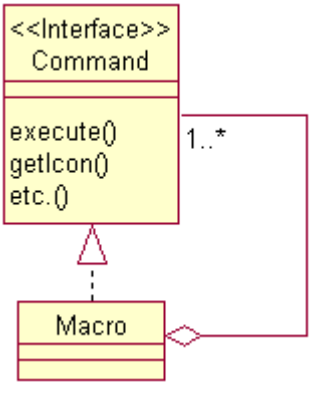
- **The Command pattern encapsulates an action – often a user's action – as a class unto itself.**
- **This is sometimes counter-intuitive: isn't an action supposed to be just one method on a class?**
- **The idea of the Command pattern is to distinguish between an ordinary method call and an action, which is easiest to understand in user-interface terms:**
  - An action is something a user might do.
  - It can be **done**, and also **undone**!
  - It can be **doable** – let's call that **enabled** – or not.
  - It can be represented in various graphical ways: by shorter and longer strings, tip text, and various icons.
  - Perhaps it can be **automated**.
- **A warning sign for Command is duplication of the above features in code that uses an event handler:**
  - A menu item, toolbar and button all coded with the same text, icon, and tooltip information.
- **To manage most of these features, we want the ability to encapsulate actions: to reference them, to pass them around, and possibly to collect them.**
  - You can't do that with a method! You need an object.

The Command Pattern

- The solution is to encapsulate an action as a class:

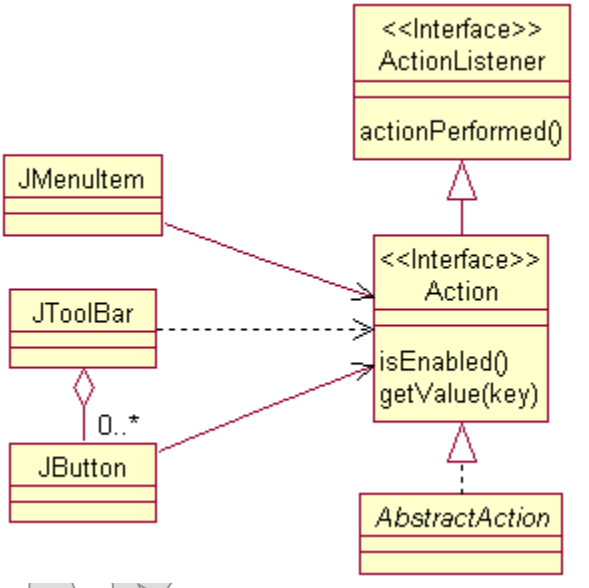


  - The **Command** defines at least a method such as **execute** – which, if you like, is the core method that got a promotion!
  - It will typically define additional standard attributes and methods suitable for its context: get an icon, undo/redo, etc.
  - Then various **ConcreteCommand**s will implement **execute** to carry out specific behavior, and the other standard attributes and methods to represent the action on a menu, toolbar, or button face; to live in a collection for undo/redo or macro scripting; etc.
  - A specialization of the pattern would be to combine it with Composite to support macros.



The Command Pattern

- **Examples are found especially in AWT and JFC:**
  - JFC defines the **Action** interface and **AbstractAction** class – **Action** here is **Command** from the GoF pattern definition. Menus, toolbars, buttons, etc., can all aggregate **Action**s:



  - **JTextComponent** organizes a great deal of its feature set as actions, and makes these actions publicly available so that they can be connected to a given application's UI: things like clipboard actions, select-all, clear, etc.
  - The HR class **cc.hr.gui.Application** organizes almost all of its code into actions; populates its menu with them, and also makes the actions available to its child windows. This provides a path for, say, an **EmployeesView** to trigger the opening of a **DepartmentsView**.
  - Jakarta Struts applies the Command pattern to HTML request/response cycles.

HR Commands

EXAMPLE

- **Examples\HR\Step6** introduces a client-server structure to the case study, with separate RMI server and client application.
  - See **cc\hr\Application.java** and the inner class **ConnectAction**:

```
private class ConnectAction
  extends AbstractAction
{
  public ConnectAction ()
  {
    super ("Connect");
    putValue (SHORT_DESCRIPTION,
      "Connect to HR server");
    putValue (MNEMONIC_KEY,
      new Integer (KeyEvent.VK_C));
  }

  public void actionPerformed (ActionEvent ev)
  {
    LoginDialog dlg =
      new LoginDialog (Application.this);
    dlg.setVisible (true);
    if (dlg.getHost () == null)
      return;
    ...
    connected = true;
    updateActionStates ();
  }
}
private ConnectAction connectAction =
  new ConnectAction ();
```

HR Commands

EXAMPLE

- **ConnectAction** extends **AbstractAction** to
  - Initialize a few properties in the constructor
  - Implement **actionPerformed** to connect to an RMI server
- Note that part of the action is to enable other actions, through the helper method **updateActionStates**:

```
public void updateActionStates ()
{
  viewDepartmentsAction.setEnabled (connected);
  viewEmployeesAction.setEnabled (connected);
  viewJobsAction.setEnabled (connected);
}
```

- An inner class **TopLevelMenu** builds the menu.
  - Note that almost all the work here is in creating menu items based entirely on **Action** objects:

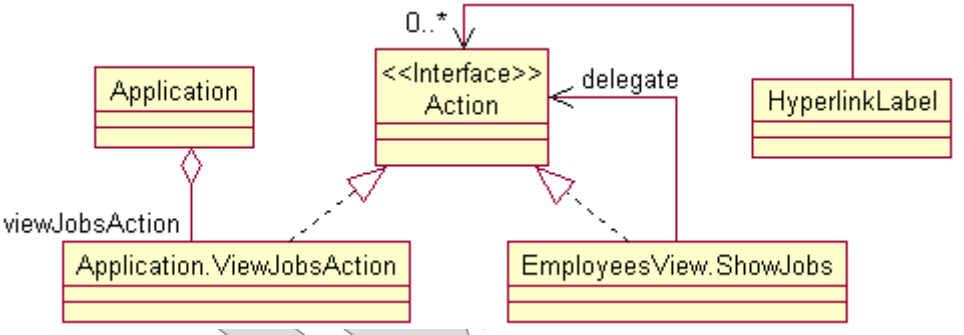
```
JMenu dataMenu = new JMenu ("Data");
dataMenu.setMnemonic ('D');
dataMenu.add (connectAction);
dataMenu.add (closeAction);
dataMenu.add (exitAction);
add (dataMenu);
```

- Build and test this step of the application, and observe the View menu items "lighting up" after the RMI connection is made.
  - Note that for this step to work you must have set up the database as described earlier in the chapter.

HR Commands

EXAMPLE

- **Application** has no toolbar, nor are there dialogs with buttons that could reuse these action objects.
  - In the future, there could be! and this would be a snap.
- It is also possible to re-use functionality packaged as an **Action**.
  - Reuse is not as direct as calling a method, but very nearly.



  - **Application** passes its **viewJobs** action (and the **viewDepartmentsAction**, for a similar purpose) when creating the **EmployeesView**:

```
public void actionPerformed (ActionEvent ev)
{
  if (employeesView == null ||
      !employeesView.isVisible ())
    addChildFrame
      (employeesView = new EmployeesView
        (employeeService, departmentService,
          jobService, viewDepartmentsAction,
          viewJobsAction, demux));
  else
    activateChildFrame (employeesView);
}
```

EXAMPLE

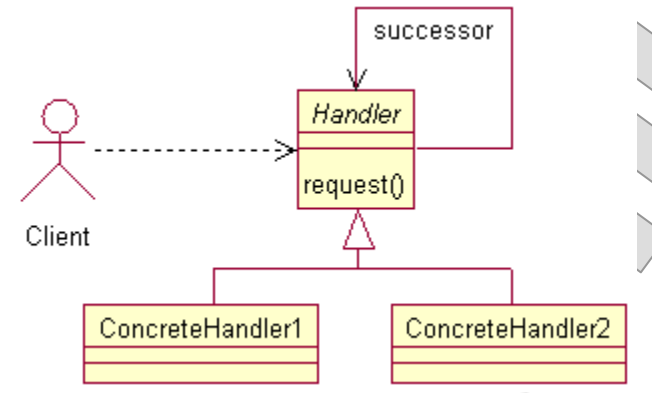
– This allows that window class to **aggregate** the action into an event handler of its own: **ShowJob** first delegates to the given action, to assure that a **JobsView** is active, and then assures that this employee's job is selected in the table.

```
protected class ShowJob
  implements ActionListener
{
  public ShowJob (ActionListener delegate)
  {
    this.delegate = delegate;
  }

  public void actionPerformed (ActionEvent ev)
  {
    delegate.actionPerformed (ev);

    if (selectedEmployee != null)
    ... // select this employee's job
  }

  private ActionListener delegate;
}
```

– The aggregate action is passed along to a **HyperlinkLabel** object that fires **ActionEvent**s. When the **Jobs:** label is clicked, **ShowJobs** is executed.

The Chain of Responsibility Pattern

- The **Chain of Responsibility** pattern allows multiple parties to get a look at a request, in sequence.
- The Observer pattern allows multiple observers, but there is no sense of synchronization between them.
  - The Observer pitfall is the Chain of Responsibility warning sign: don't expect subjects to coordinate notifications for your observing objects; take control yourself.
- Chain of Responsibility is distinct in two ways:
  - It is not necessarily a chain of observers, though it can be; in its basic form it deals with a simple **method invocation** or **request**, not the "inversion" of registering a callback interface.
  - It addresses situations in which it is important to **orchestrate** a response in which there are several players, each contributing some part of the response behavior or return value, or otherwise carrying out tasks related to the request.

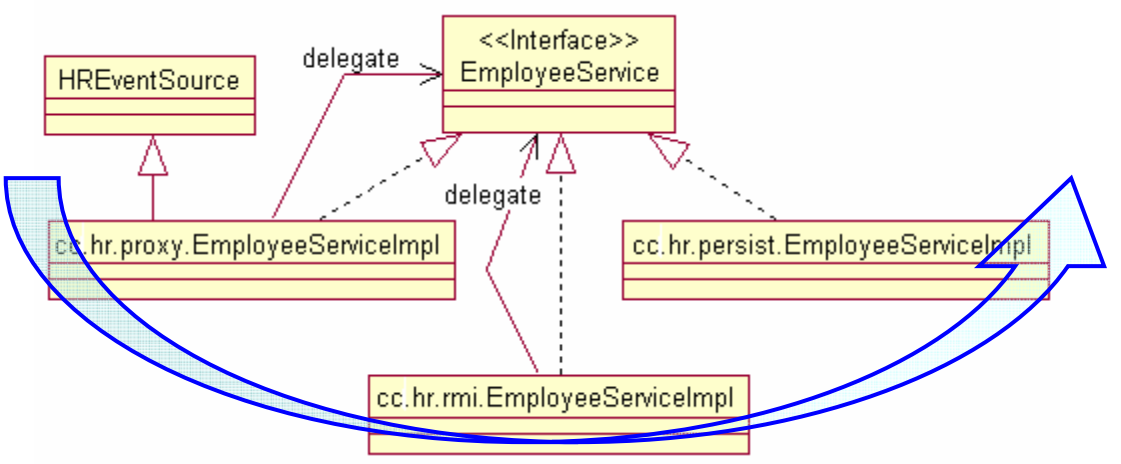


The Chain of Responsibility Pattern

- **We'll consider a few examples.**
- **In GUI event handling, there is a need to combine Observer structures with nested composition of application holding windows holding panels holding individual controls, etc.**
  - It can be important to give each level of the composition an opportunity to handle a user action.
  - For instance a text component gets first crack at a keystroke event, but then the panel or window that holds it might want a look. The application will want a look, for purposes of checking for keyboard accelerators.
- **Java Servlets defines a means of chaining filters to individual servlets.**
  - The decoupling here is nearly total, as the filters and servlet can be written in complete ignorance of each other.
  - They are connected into a system that will handle a given HTTP request **declaratively**, in the Web application's deployment descriptor.
  - We'll see in a later chapter that this is more explicitly defined for J2EE as the Intercepting Filter pattern.

The Chain of Responsibility Pattern

- **Finally, in multi-tier systems, service interfaces may be implemented by several, chained objects, each of which adds one feature to the total system and response.**
  - Consider the chained implementations of the HR application's **EmployeeService** from **Examples\HR\Step8**:



  - The **cc.hr.proxy** implementation fires model events back to various GUI observers, so that the MVC system isn't stretched out over the network.
  - The **cc.hr.rmi** object handles a remote request from the proxy; this is its sole responsibility.
  - The **cc.hr.persist** implementation serves the request using persistent objects based on relational data. (We'll look at this step more closely in a later chapter, as part of our study of Data Access Objects.)

Design Exercises: Behavioral Refactoring

In this exercise you will analyze several "before pictures" of existing software that could do with a dose of design patterns. This is primarily a pencil-and-paper exercise, focusing on refactoring existing designs to new ones.

Detailed instructions are contained in the Design Exercises write-up at the end of the chapter. Your instructor may recommend that you pursue these design exercises as a class, or perhaps in small groups, then to reconvene and discuss your solutions and alternatives.

Suggested time: 30-45 minutes.

SUMMARY

- **If the organizing principle of the previous chapter's patterns was control of object creation, the key features of behavioral patterns are decoupling by functionality and scalability of the design.**
  - Strategy and Template Method patterns address problems of organizing parts of complex tasks, going beyond what's offered by basic polymorphism in the Java language.
  - Observer and MVC have these benefits, too, but they also break code designs out of "traps" by which they would scale poorly as the application would grow in complexity.
  - Command offers a novel means of encapsulating things that aren't traditionally captured in a single class – GUI frameworks and application code alike can take advantage of these neat little packages of UI attributes and behavior.
  - Chain of Responsibility is perhaps the most directly and obviously committed to decomposition: every part of a complex process must be encapsulated separately, and those encapsulations can be combined in various ways at runtime.
- **Other GoF behavioral patterns not covered in this chapter also address these general concerns.**
  - Mediator, State, Visitor, etc. – surprisingly like Command, these can all be seen as strategies for encapsulating things that don't fit the traditional state-and-behavior OOAD model.