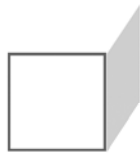
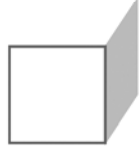
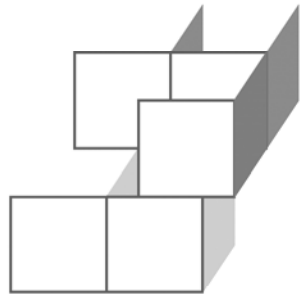




CHAPTER 3
BEHAVIORAL PATTERNS



OBJECTIVES

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

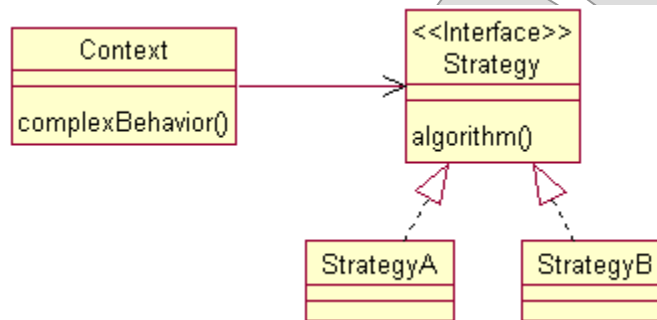
- Strategy
- Template Method
- Observer
- Model/View/Controller
- Command
- Chain of Responsibility

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

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

The Strategy Pattern

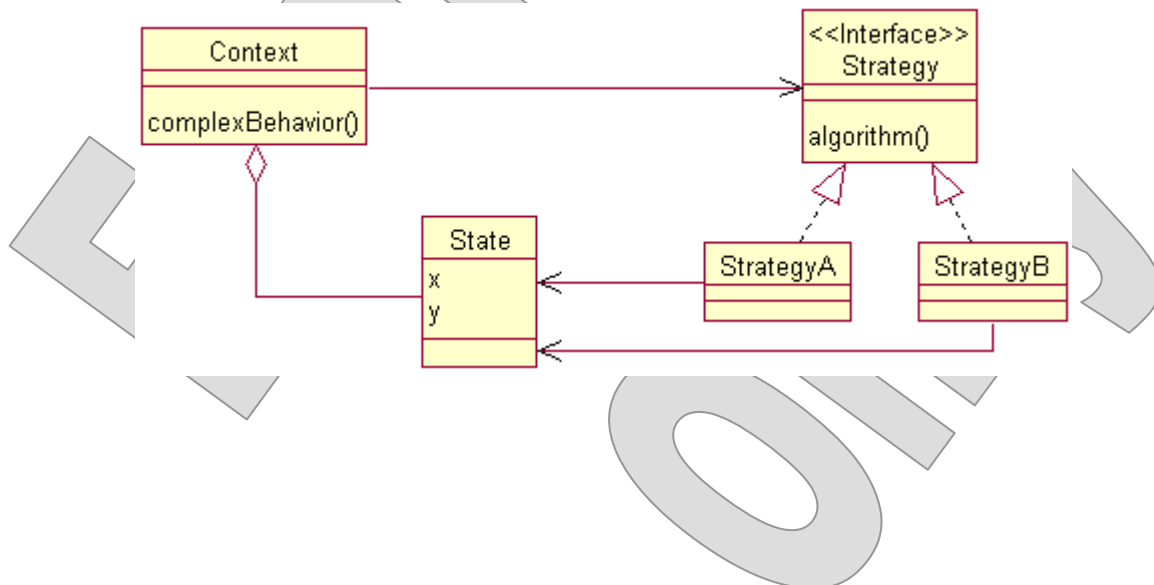
- The **Strategy** pattern addresses the problem of varying behavior required by what is initially considered a single encapsulation.



- 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

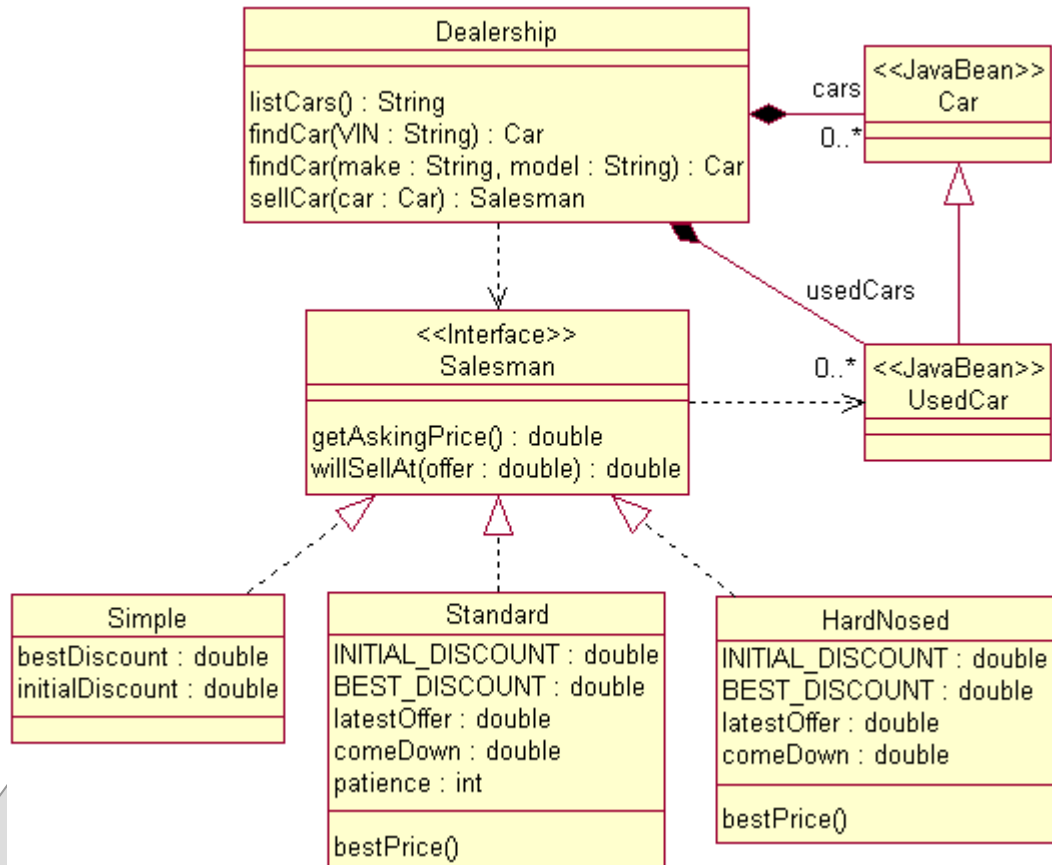
- Who controls the choice of strategy?
 - Can a client plug in a strategy of its choosing?
 - Does the context object decide for itself?
 - Hmm ... perhaps a **factory** makes the decision in assembling the system.
- How can the context object share useful state information with a Strategy delegate?
 - 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

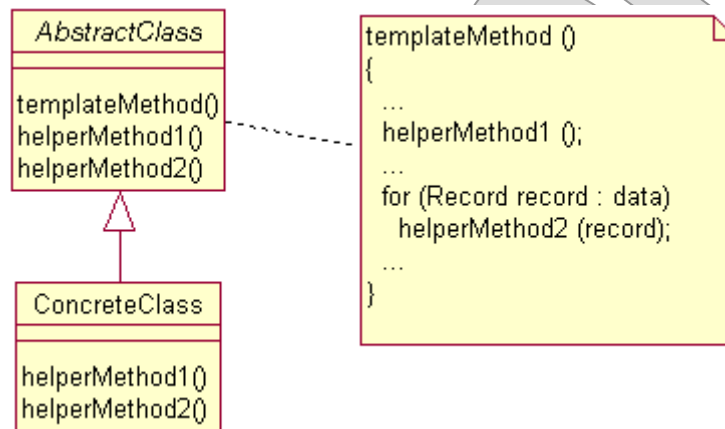
- In **Examples\Cars**, the act of selling a car is broken out of the **Dealership** class as a separate strategy.



- 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

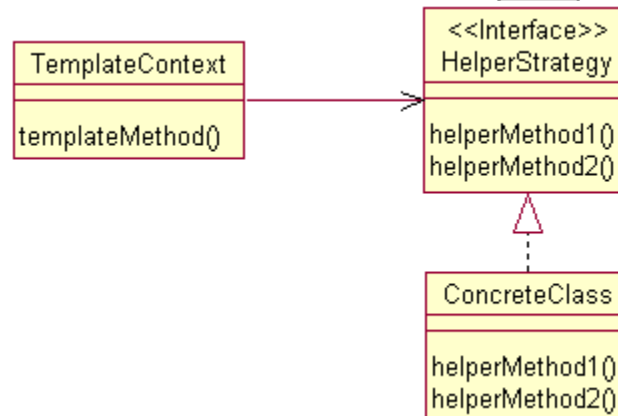
- Another simple pattern is Template Method, which addresses the need to specialize how some parts of a general and more complex task are carried out.



- 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

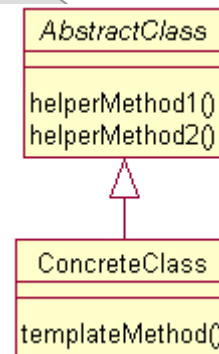
- Note that any Template Method solution could be refactored as a Strategy:



- 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.



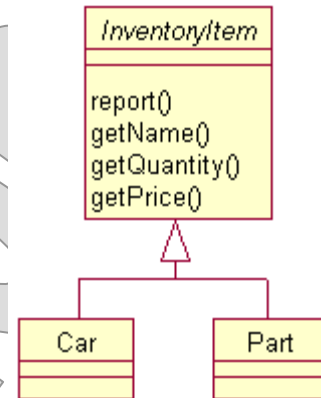
Report Template Method

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.

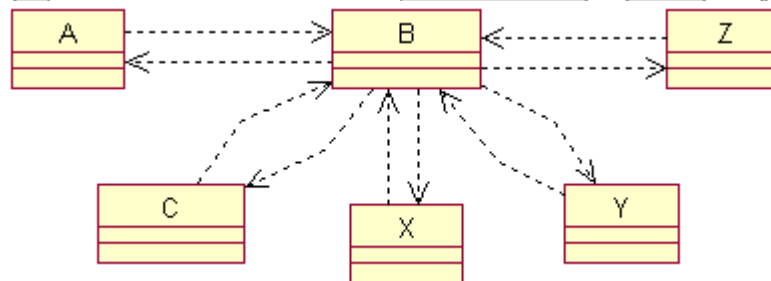
Evaluation Only

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.

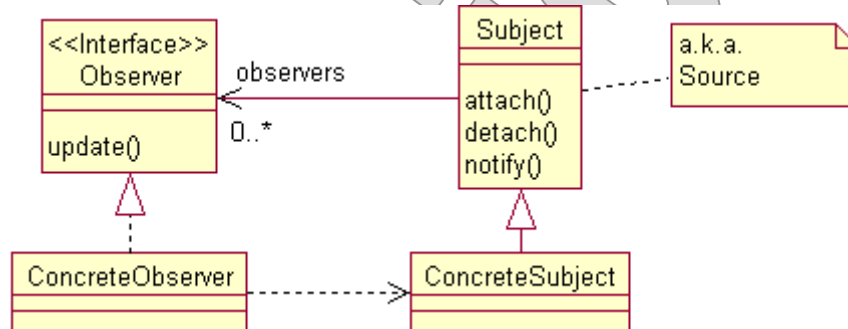
```
while (!download.isComplete ())
  progressBar.setValue (download.pctComplete ());
```

- 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);
    }

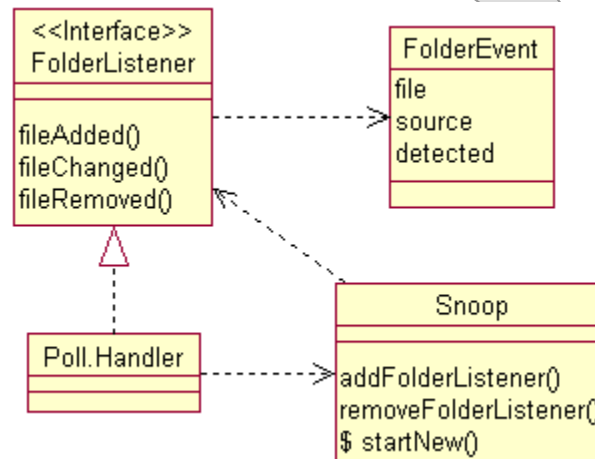
    for (Observer o : local)
        o.update ();
}

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 **FolderListeners**, 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.

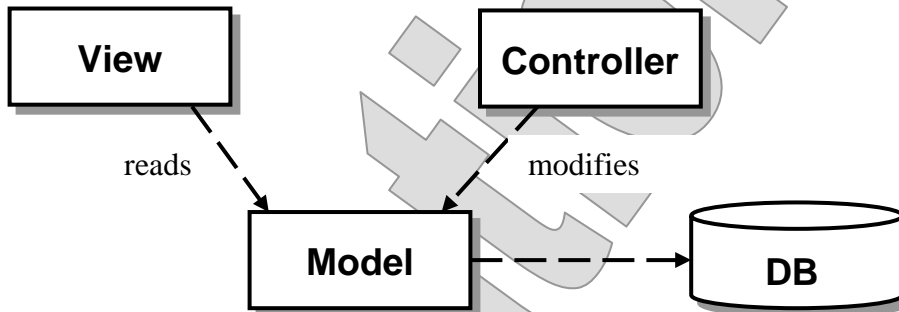
Evaluation Only

The Model/View/Controller Pattern

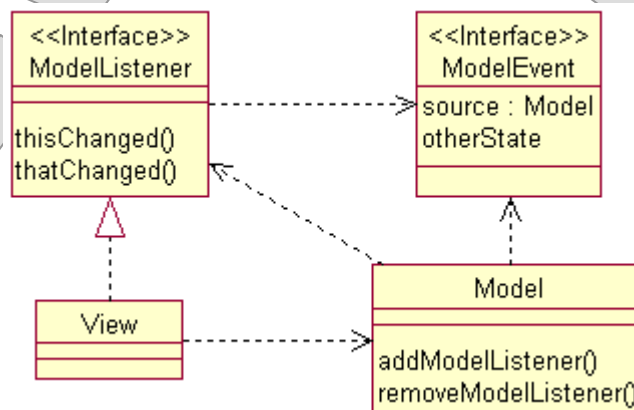
- The one pattern in this course that is not defined in the GoF text is **Model/View/Controller**, or **MVC**.
- MVC originated in the Smalltalk platform.
- It addresses the problem of organizing an application's state and behavior in a way that
 - **Centralizes the state** information
 - Defines **clear 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

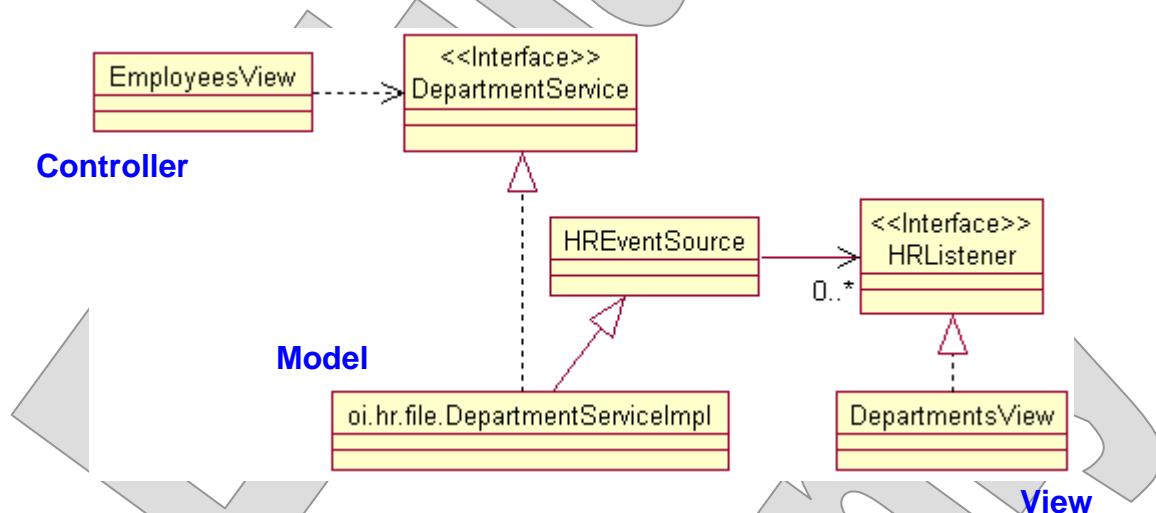
- Examples of MVC live at different levels of the Core API and common application architectures.
- Many JFC controls have internal MVC systems.
 - Consider **JList** and **ListSelectionModel**.
 - Some JFC controls reduce the pattern to a two-role version called the **model/view split**.
- In Web application 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);
    ...
}
```

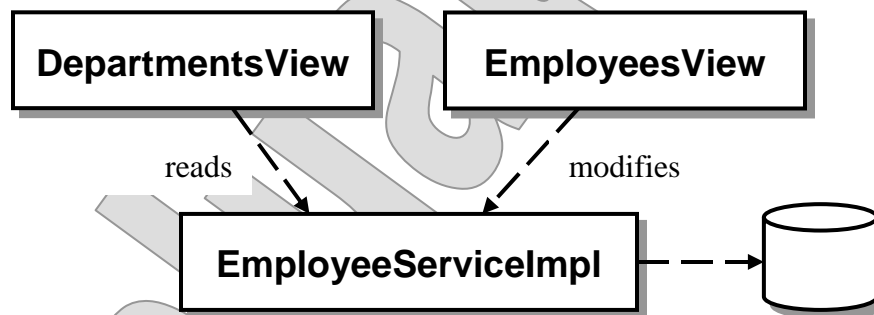


- The model consists of the service implementations, whose mutators fire HR events by calling one or more methods on registered **HRListeners**.
- The three view classes act as **controllers** when they call those mutators, and as **views** by implementing **HRListener** themselves.

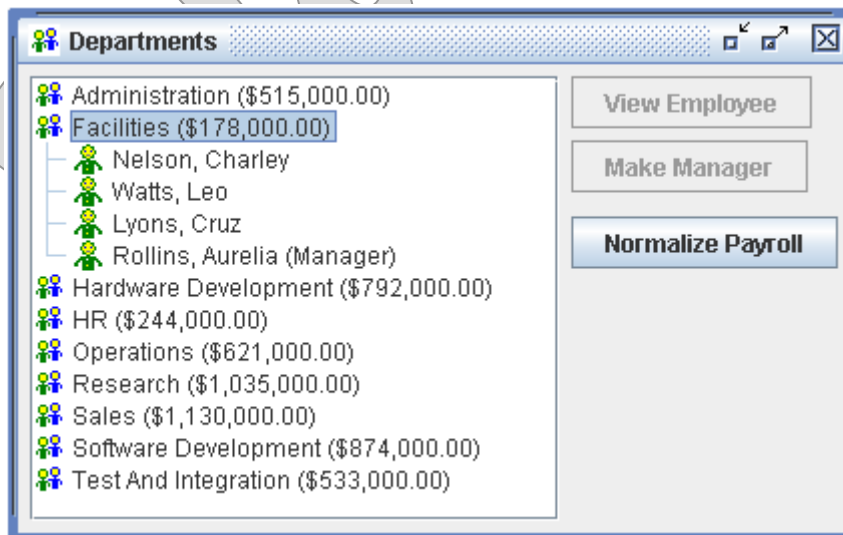
An MVC System

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.



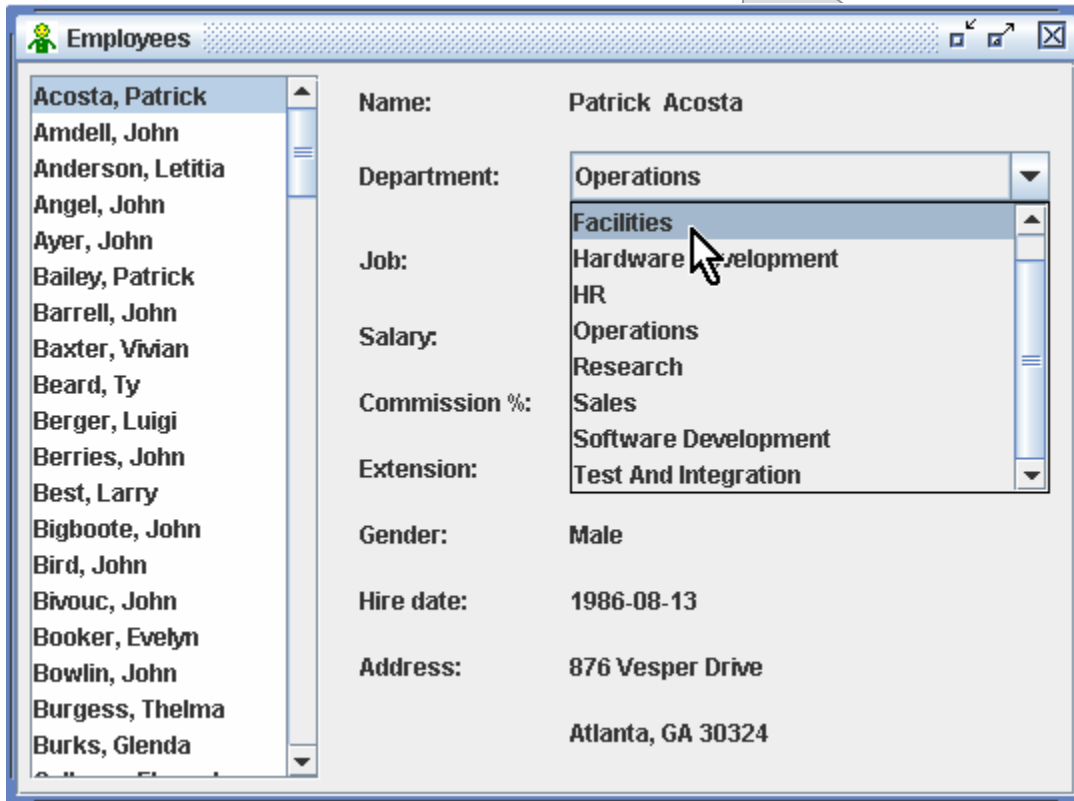
- Try it! **View|Departments** and open Facilities ...



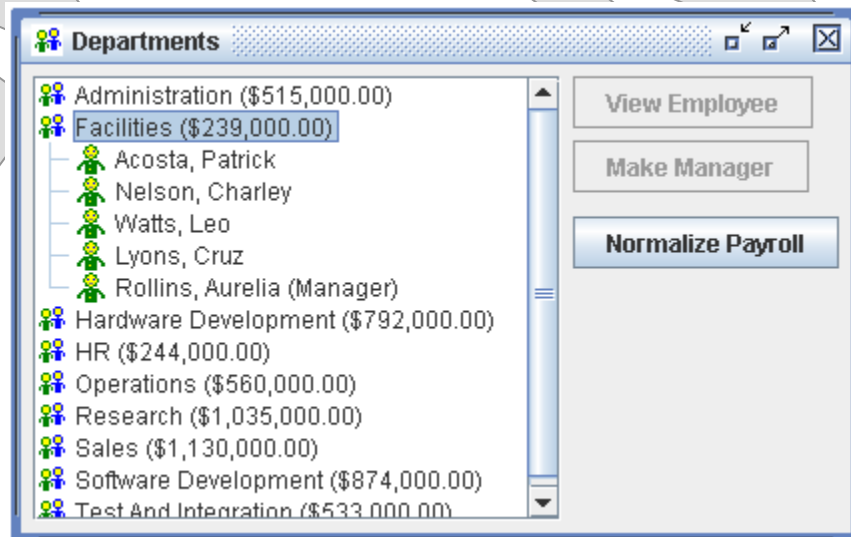
An MVC System

EXAMPLE

- 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 ());
}
```

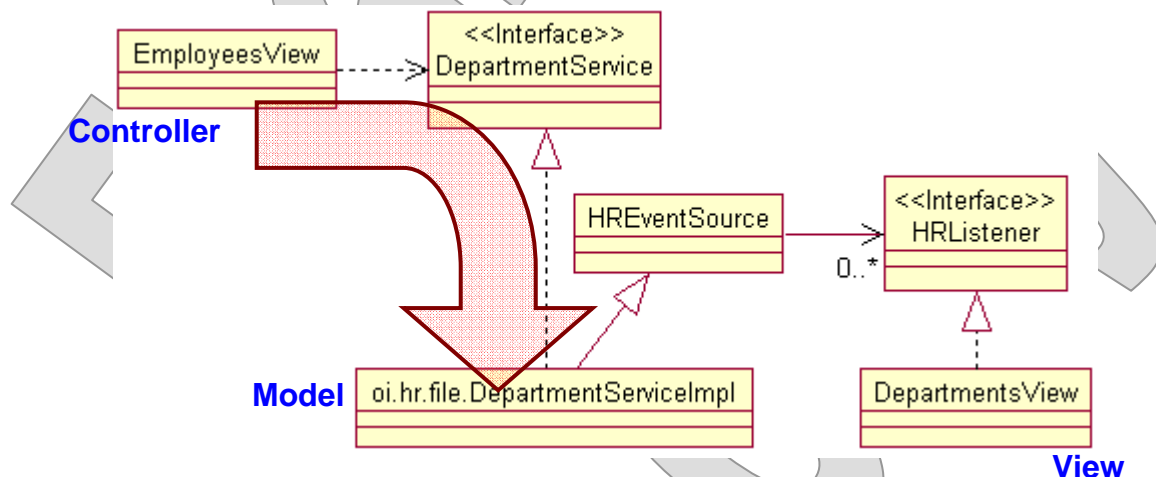
An MVC System

EXAMPLE

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.

An MVC System

EXAMPLE

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 (HRLListener 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));
}

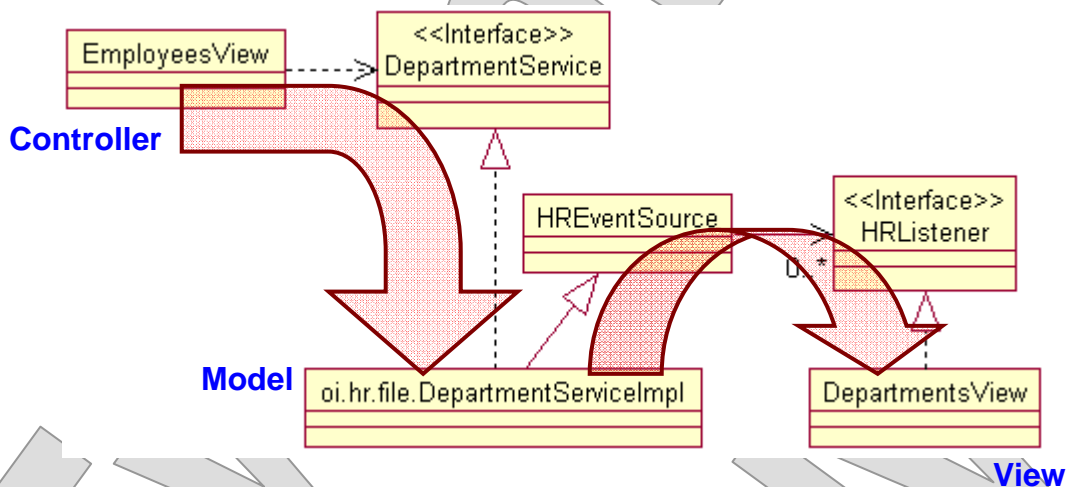
public class DepartmentsView
->protected class WindowHandler
->public @Override void internalFrameOpened (...)
{
    source.addHRLListener (mainPanel);
}
```

An MVC System

EXAMPLE

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.

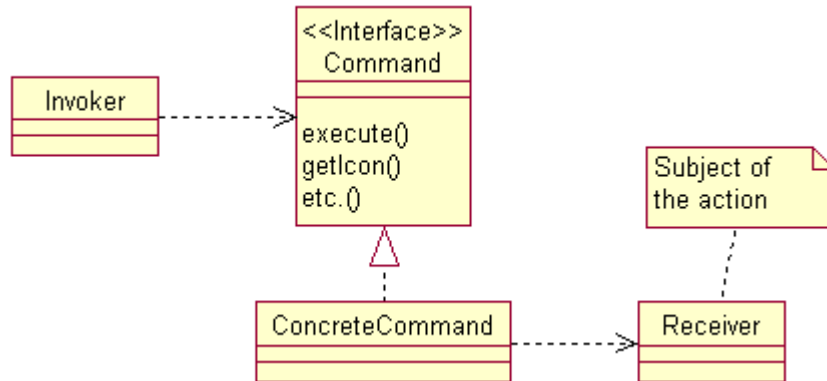
Evaluation Only

The Command Pattern

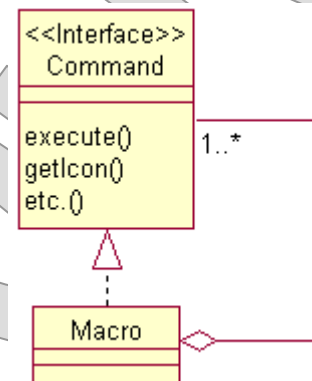
- The **Command** pattern encapsulates an **action** – often a user’s action – as a class unto itself.
- This is sometimes counter-intuitive: isn’t an action supposed to be just one method on a class?
- The idea of the Command pattern is to distinguish between an ordinary method call and an action, which 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:

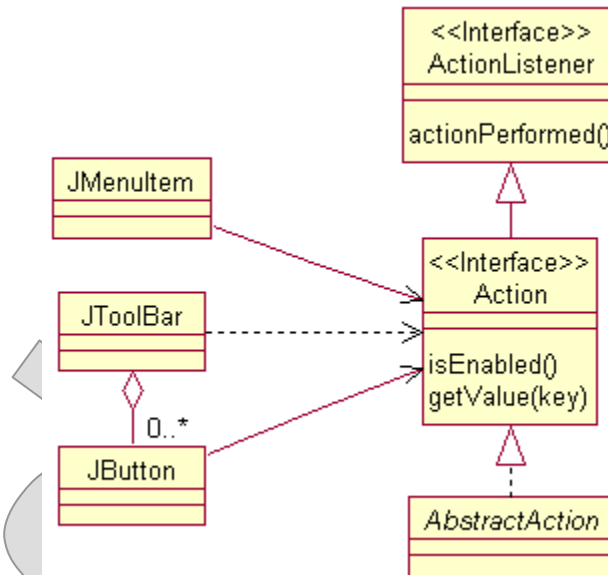


- The **Command** defines at least a method such as **execute** – which, if you like, is the core method that got a promotion!
- It will typically define additional standard attributes and methods suitable for its context: get an icon, undo/redo, etc.
- Then various **ConcreteCommands** will implement **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 **Actions**:



- **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:

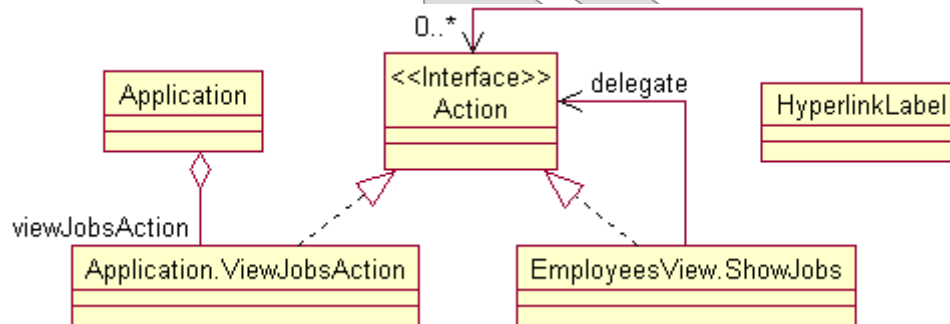
```
JMenu dataMenu = new JMenu ("Data");
dataMenu.setMnemonic ('D');
dataMenu.add (connectAction);
dataMenu.add (closeAction);
dataMenu.add (exitAction);
add (dataMenu);
```

- 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);
}
  
```

HR Commands

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);

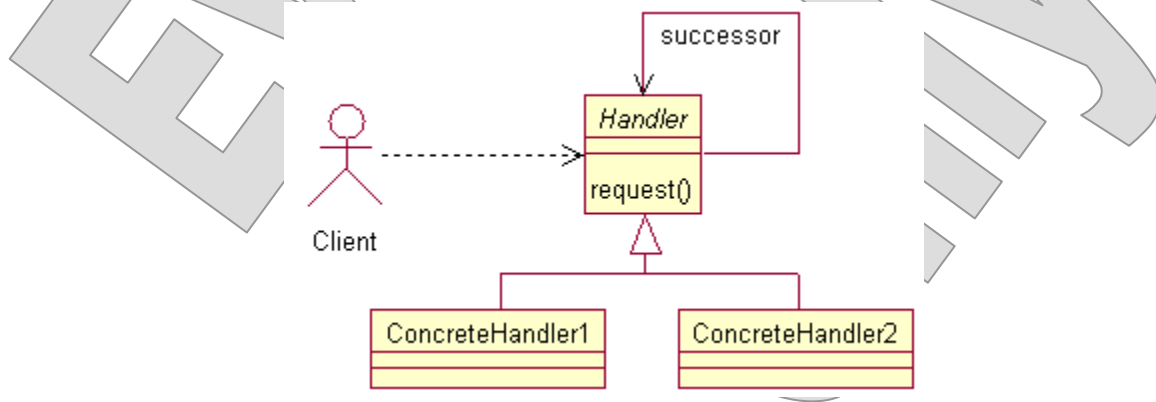
        if (selectedEmployee != null)
            ... // select this employee's job
    }

    private ActionListener delegate;
}
```

- The aggregate action is passed along to a **HyperlinkLabel** object that fires **ActionEvents**. 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.

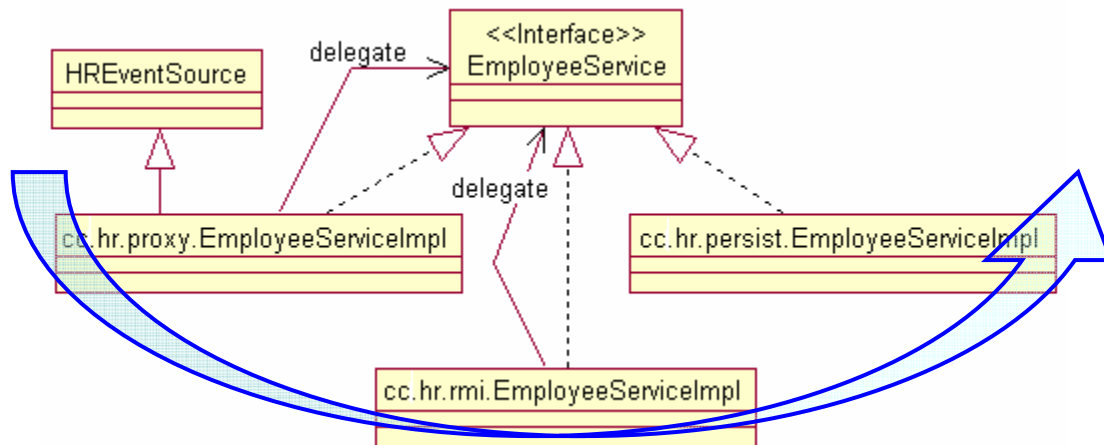


The Chain of Responsibility Pattern

- 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.

Evaluated Only

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.

Health Information Request

LAB 3A

Introduction

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.

Suggested Time: 45 minutes

Root Directory: Capstone\JavaPatterns

Directories: Labs\Lab3A (do your work here)
 Examples\Health\Step1 (backup copy of starter files)
 Examples\Health\Step2 (contains lab solution)

Packages: cc.health

Files: cc\health\InfoRequest.java
 cc\health\LabResultsRequest.java

Instructions

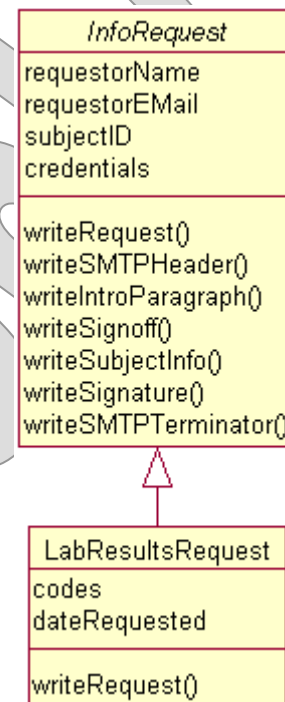
- Review the starter code and the design shown at right. Observe that the base class **InfoRequest** offers an abstract method **writeRequest** and a set of helper functions, such as **writeSMTPHeader** and **writeSignature**. The derived class **LabResultsRequest** implements the abstract method by writing some specific information, interlaced with generic information and functionality provided by the helper functions:

```
public void writeRequest (OutputStream out)
{
    PrintWriter writer = new PrintWriter
        (new OutputStreamWriter (out));

    writeSMTPHeader (writer);

    StringWriter capture = new StringWriter ();
    PrintWriter buffer = new PrintWriter (capture);

    writeIntroParagraph (buffer);
    ...
}
```



2. Give the application a quick test – certainly it works just fine:

build

run LabResultsRequest

MAIL FROM:<request@healthcare_r_us.com>

RCPT TO:demento@whatsit.com

DATA

We hereby request the following patient data,
with credentials and authorizations as shown below.

Lab tests were ordered by this office on 10/10/2005
If you have any questions, please contact our office.

Thank you,
Dr. Demento

Subject ID: Will Provost

Credentials: quicksand

Information requested: Lab test results

Test codes:

7033

0085X

21-889

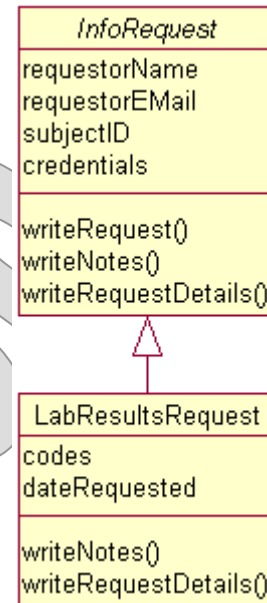
BEGIN_SIGNATURE

302c214418dfcc4d666afb5914da23e6c8cad93684a72f921440a37954ae87686bcccde9
ce33f3533a6249dad

END_SIGNATURE

3. This code exhibits the warning sign for the Template Method pattern. Yes, the derived class gets the advantage of calling helper methods, but it still has to implement the overarching logic of the complex task – which will be the same for all subclasses, and hence should ideally be captured in the base class along with the code in the existing helper methods. How will you refactor this code? You might want to draw up a quick design diagram before proceeding, and/or write up some pseudocode that represents the new arrangement of base- and derived-class methods.

4. We'll proceed to implement a refactored design as shown at right. Now, the **writeRequest** method is implemented as a Template Method in the base class, and this method absorbs what were the helper functions. (There is no longer any need for derived classes to call these helpers – the whole point of Template Method is for the base class to coordinate the task and to call derived-class overrides of virtual methods to complete the process. Put another way, the derived class now answers questions, rather than asking them.)
5. In **InfoRequest.java**, declare two new abstract methods, **writeNotes** and **writeRequestDetails**. Each method should be **protected**, **abstract**, **void**, and take a **Writer** as a parameter.
6. Remove the **abstract** modifier from **writeRequest**. Open a method body. Over the next several steps you will assemble the implementation, from a combination of the old derived-class implementation and the old helper methods.
7. Declare and initialize a **writer** to which the email content will be produced. Get this code from **LabResultsRequest.writeRequest**, and import **java.io.OutputStreamWriter**.
8. Produce the SMTP header, using code from **writeSMTPHeader**.
9. Since much of the message body is digitally signed, you must create a second writer as a **buffer**. Use the derived-class method as a basis for this code, and you will need to import **java.io.StringWriter** at this point.
10. Now copy code from **writeIntroParagraph**, but change **writer** to **buffer** so that the content is captured in the buffer for signing later.
11. Now call **writeNotes**, passing the **buffer**.
12. Bring in code from **writeSignoff** and **writeSubjectInfo** – again, change **writer** to **buffer**.
13. Call **writeRequestDetails**.
14. Now capture the buffer content in a string **signedContent**, and produce that string to the main **writer**. Code for this can be drawn from **LabResultsRequest.writeRequest**.
15. Bring in code from **writeSignature** that will produce a digital signature based on **signedContent**.
16. Produce the final line of the message, as in **writeSMTPTerminator**.
17. Call **writer.flush**. This concludes the implementation of **writeRequest**.
18. Remove the old helper methods from the class.
19. In **LabResultsRequest**, implement **writeNotes** and **writeRequestDetails** using code from **writeRequest**.



20. Now remove the **writeRequest** implementation – by now, all the code here has been copied somewhere else!
21. You should now be able to rebuild and retest, and see the same output (the digital signature will vary from run to run).
22. So – how'd we do? What do you think: is the refactored code better than what we had? Consider how much code is packed into the base vs. the derived class. Also, as additional subtypes of **InfoRequest** are implemented in this fledgling application, how much initial development work have we saved ourselves? If a standard feature were to be added to the message structure after there were, say, 20 different request types, how much maintenance work have we saved? Conversely, what costs or other impacts do you see from applying the Template Method pattern?

Evaluation Only

Primes

LAB 3B

Introduction

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.

Suggested Time: 45-60 minutes

Root Directory: Capstone\JavaPatterns

Directories:

Labs\Lab3B	(do your work here)
Examples\Primes\Step1	(backup copy of starter files)
Examples\Primes\Step2	(intermediate answer)
Examples\Primes\Step3	(contains lab solution)
Examples\Primes\Step4	(answer with optional steps)

Packages: cc.primes

Files:

- cc\primes\FindPrimes.java
- cc\primes\CLI.java
- cc\primes\PrimesEvent.java (to be created)
- cc\primes\PrimesListener.java (to be created)
- cc\primes\StressTest.txt which becomes **StressTest.java**
- cc\primes\GUI.txt which becomes **GUI.java**

Instructions

1. Review the starter code, which really just involves two classes: **FindPrimes** is a component that runs a prime-finding algorithm, and **CLI** is a console application that uses this component and prints out the results, as in:

```
FindPrimes worker = new FindPrimes (Integer.parseInt (args[0]));
worker.find ();
for (long prime : worker.getPrimes ())
    System.out.print (" " + prime);
```

2. Give it a try:

```
build
run CLI 10
2 3 4 5 7
```

3. Try it again with larger ceilings, and notice as the application slows down that it prints all its results at the end of the run, not in a real-time stream. If we want to see the inner workings of the component, one naïve way to do it would be to have **FindPrimes** write to the console itself. This is a classic short-term solution that tends to require refactoring later: what if we want to pipe to a file or a GUI instead?
4. The stronger solution is to make the component observable. Start by creating an event class **PrimesEvent**. This can be a simple **JavaBean** with a single property, a **long** called **prime**.
5. Now create the observer interface **PrimesListener**. This has a single method **foundPrime**, which takes a **PrimesEvent** as its only parameter.
6. Now you'll make **FindPrimes** an source for **PrimesEvents**. First, add a private field **listeners** to the class: this is a **List** of **PrimesListener** references, and should be initialized to an empty **LinkedList**.
7. Add methods to register and remove listeners: **addPrimesListener** and **removePrimesListener** just add or remove the given **PrimesListener** to or from the **listeners** list.
8. Now, create a protected helper method called **firePrimesEvent**: void return type and taking a **long** as its parameter. This method will create a new **PrimesEvent** based on the given value, and then loop through **listeners**, calling **foundPrime** on each registered listener and passing the newly-created event object.
9. In **find**, near the bottom of the code, you'll see a call to **primes.add** a new number. Right after that, call your new helper method to fire the event based on the number, **candidate**.
10. The last piece of the puzzle is to make the console application listen for events, so it can write the values to the console in real time. Open **CLI.java** and add an inner class **Handler**, which implements **PrimesListener**. In your **foundPrime** method, just write the value out to the console – you can cut and paste the line of code for this from the **main** method.
11. Now change **main** to:
 - Register an instance of **Handler** as a listener with the **worker** object
 - Stop printing out results at the end of the process – this would just duplicate the output from the **Handler** at this point
12. Build and test, and you should see that you get the same apparent behavior for a few primes, but for large ceiling values you now see the primes as they are discovered, rather than in a big dump after a long pause. (This is the intermediate answer in **Step2**.)
13. Have you noticed something is wrong with the code as written so far? Give this a bit of thought before proceeding.

14. Rename the file **StressTest.txt** to a **.java** extension; this prepared class wouldn't have compiled at the start of the lab because it uses your new event and listener types. It stores a reference to a **FindPrimes** component, and then repeatedly and rapidly adds and removes itself as a handler for **PrimesEvents**.
15. Open **CLI.java** and add one line of code to attach the stress tester to the component while running the application:

```
FindPrimes worker = new FindPrimes (Integer.parseInt (args[0]));
worker.addPrimesListener (new Handler ());
new StressTest (worker).start ();
worker.find ();
```

16. Build and run the application again, with a fairly high ceiling. Exact behavior here is not entirely deterministic, since the OS will schedule threads differently each time you run it, but you should quickly see that there is a race condition in the way listeners are registered and removed:

```
build
run CLI 100000
2 3 4 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83
89Exception in thread "main" java.util.ConcurrentModificationException
  at java.util.LinkedList$Itr.checkForComodification(LinkedList.java:617)
  at java.util.LinkedList$Itr.next(LinkedList.java:552)
  at cc.primes.FindPrimes.firePrimesEvent(FindPrimes.java:104)
  at cc.primes.FindPrimes.find(FindPrimes.java:47)
  at cc.primes.CLI.main(CLI.java:22)
```

17. This is the classic pitfall of subjects that are not thread-safe in how they handle their observers. What do you need to do to fix this?
18. Open **FindPrimes.java** and make three fixes:

- Make both **addPrimesListener** and **removePrimesListener** **synchronized**
- Change **firePrimesEvent** to make a local copy of the **listeners** list and to call **foundPrime** on each reference in that local list – to do this, declare the local list object, initialized to **null**, then enter a code block that is **synchronized on this**, and in that block, assign your local list reference to a new **ArrayList**, passing **listeners** to its constructor to make the copy, and use your local list in the **for** loop

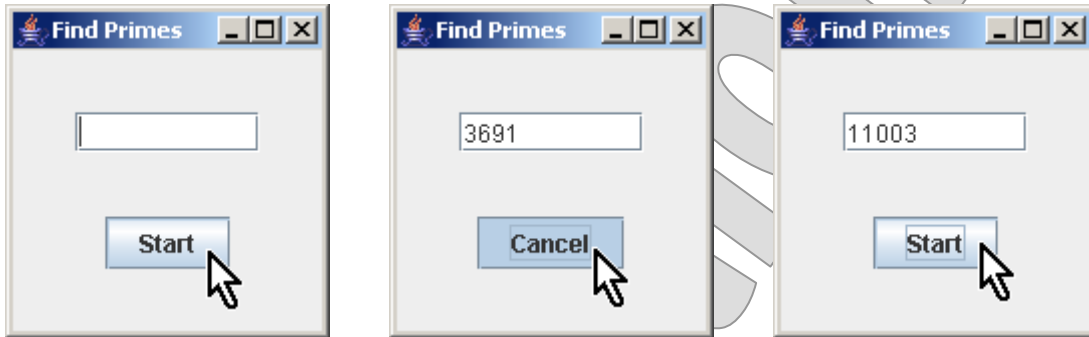
Now the management of listener references is thread-safe, because the local copy made in **findPrimes** can't be modified during the loop. The copy itself is made in a block of code that can't be interrupted by either the add or remove method, since they are also synchronized (implicitly, on **this**).

19. Test again, and you should now be unable to produce the race-condition crash you saw earlier. (This is the final answer in **Step3**.)

Optional Steps

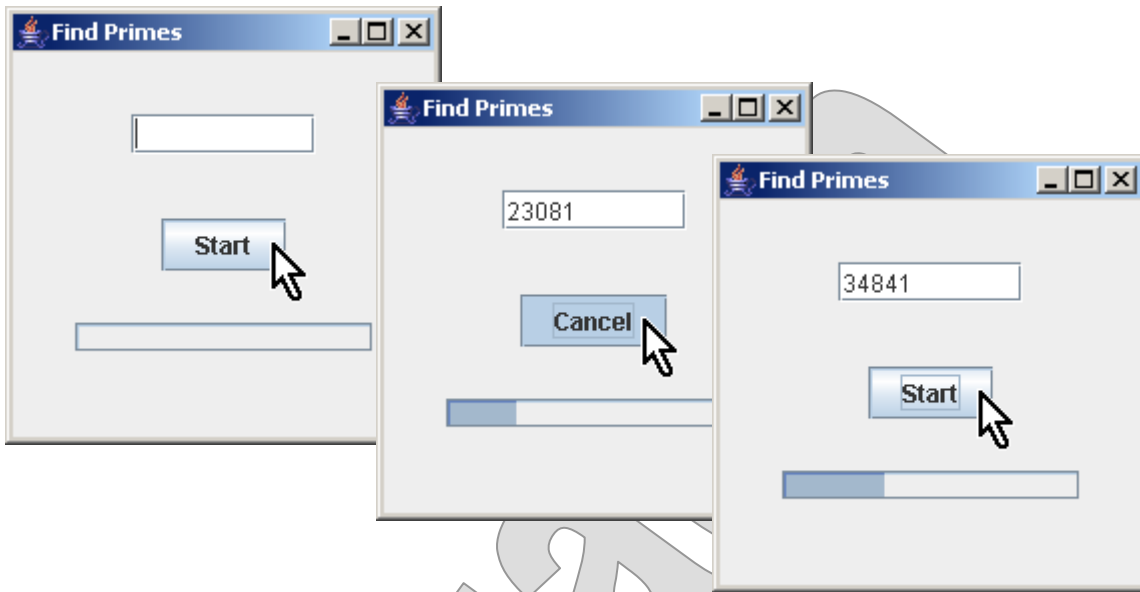
20. Rename **GUI.txt** to **GUI.java**, build and test. This application has the ceiling hard-coded to 100,000; click **Start** to kick off a run of prime-finding and see that you can stop and restart by clicking the button repeatedly.

```
build
run GUI
```



21. Create a new class **ProgressBar** which extends `javax.swing.JProgressBar` and implements **PrimesListener**.
22. Write a constructor that takes a **FindPrimes** object as a parameter. Set the maximum value of the progress bar to the ceiling value for the object – the method is **getHowHigh**. Set the starting value of the progress bar to zero, and then add **this** as a listener on the **FindPrimes** object.
23. Implement **foundPrime** to set the progress bar value to the latest prime number.
24. Open **GUI.java** and add a field **progress**, initialized to an instance of your new class. Pass **worker** to the **ProgressBar** constructor.
25. In the constructor, change the size of the **GridLayout** from two rows to three.
26. Also in the constructor, add code to place the **progress** control as the third row – use the four lines of code that create the **pnStart** panel as a template for this, creating a **pnProgress** instead and adding **progress** to it.

27. Build and test:



(This is the optional answer in **Step 4**.)

MVC in the HR Application

LAB 3C
Optional

Introduction

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).

Suggested Time: 45-75 minutes.

Root Directory: Capstone\JavaPatterns

Directories:

Labs\Lab3C	(do your work here)
Examples\HR\Step1	(backup of starter files)
Examples\HR\Step2	(intermediate answer)
Examples\HR\Step3	(intermediate answer)
Examples\HR\Step4	(contains lab solution)

Files: cc\hr\gui\EmployeesView.java

1.

Instructions

- Build and run the starter application. As in the initial example, select **File|Open** to open the data file. Select **View|Employees** to see the full list of employees. Choose a few and notice that some have “exceptional” salaries – they are out of bounds for their own job descriptions, and these are shown in red with the guidelines next to them.
- Select any employee with an exceptional salary. Click over the **Job:** label to bring up the jobs view with that employee’s job selected in the table. (You may have to scroll to see the job listing.) Now change the job’s salary boundaries to accommodate the employee in question – for instance if the employee’s salary is \$60,000 and the current range for the job is \$40,000-\$55,000, make the upper bound \$60,000. You can do this by double-clicking on the number you want to change, typing in the new number, and hitting ENTER.
- Now look at the employee screen again – no change! This is the MVC failure, or incompleteness really, that we need to correct. Close the application.
- Open **EmployeesView.java** and find the inner class **EmployeesPanel**. Make this class implement the **HRListener** interface.
- Stub out all of the **HRListener** methods, so that the class will still compile. Add a line to each method that writes a diagnostic line to the console, as in “employeeChanged() called.”. This will assure us that we have indeed plugged in to the application’s MVC system correctly.

6. Note that the **EmployeesView** constructor accepts an **HREventSource** reference, but currently doesn't do anything with it. (The **Application** prepares this source reference, so that it's anonymous to us; in fact it is a demultiplexer connected to all three service objects, so that we can attach the panel as an **HRListener** to just this one object and be sure we get events from all services.)
7. Since the view classes are managed as singletons by the application (no need for two employee windows, for example), but opened and closed possibly many times, we need to be able to respond to window open/close events, and add/remove our main panel as an **HRListener** accordingly. For this purpose, create an inner class called **WindowHandler** that extends **javax.swing.event.InternalFrameAdapter** – you might want to look over this class and the interface it implements in the Core API documentation.
8. Give the class a constructor that accepts an **HREventSource** reference and saves it off in a private field called **source**.
9. Override two methods: **internalFrameOpened** and **internalFrameClosed**. In one method, add **mainPanel** (which is the instance of **EmployeesPanel**, held by the enclosing class) to **source** as an **HRListener**; in the other, remove it again.
10. In the **EmployeesView** constructor, create a **WindowHandler** instance, passing the given **source** object to it, and add it to **this** as an **InternalFrameListener**.
11. So what do we have? The main employees view, an internal frame window, uses an inner class to listen for opening/closing events related to itself. In response it connects its main panel, which implements **HRListener**, to an event source that it was given at construction by the application itself. It trusts then that any news about the model will be sent to the panel via one of the listener methods.
12. Build and test at this point. You should see that any time data is changed by your actions in the application (including the scenario we ran at the beginning of the lab), your event handlers are called and write diagnostics to the console. If not, double-check your wiring: did you connect the **WindowHandler**, and does it correctly connect the panel as an HR listener? The compiler should have caught anything more basic than this; consult your instructor if you can't see the problem here.
13. When you have proven that events are flowing to your skeleton methods, close the application and implement the methods properly – as described in the following steps. You may want to start by removing all your diagnostic lines. All methods not mentioned below are no-ops.
14. In **wholeDepartmentChanged**, you must check to see if the **selectedEmployee** is a member of the affected department. **selectedEmployee** is maintained by the panel in response to list selection and other stimuli; note that it can be **null**! Note also that the **equals** method is implemented for all the domain objects, and that this is the correct way to check if one department is the same as another one.
15. If the employee is indeed in the changed department, then call the helper method **refreshSelectedEmployee**, which triggers an update of all GUI controls to the (possibly changed) values in the current employee record.

16. Implement **employeeReassigned** and **employeeChanged** the same way, except that your test will be to see if the **selectedEmployee** is the affected employee itself.
17. You must implement **jobChanged** as well, in case the change involves a salary-range shift that would make **selectedEmployee**'s salary exceptional when it wasn't, or not exceptional when it was before.
18. Build and test again, and you should now see proper updating in the employee view, so that all three views are in perfect synchronization over any changes to the model. (This is the intermediate answer in the **Step2** directory.)

Optional Steps

19. That's getting **EmployeeView** to play nicely as an MVC view. Now let's add some capabilities to it as a controller, and see if the other windows can keep up with us. In this optional section we'll make the employee's salary editable, instead of a static label.
20. Start by finding the **lblSalary** control and refactoring it to be a **JTextField** called **txSalary**. This will involve several replacements through the file; use an IDE, a text editor's global search, and/or rely on the compiler to flunk existing code when the field itself is changed from one thing to another.
21. In **updateDetailFields**, simplify the format used for representing the salary, removing the currency symbol, the justification and comma-separator tokens (- and ,) from the formatting string. The idea here is to allow easy parsing of the value if and when the user edits it.
22. Create an inner class **SalaryHandler** that implements **ActionListener**, and add it as a listener to **txSalary**. The **actionPerformed** method will be called when the user hits ENTER while input focus is on the salary text field.
23. Implement **actionPerformed** to call a helper method **updateSalary** – also a member of **SalaryHandler**. In this helper method, check that **selectedEmployee** is not **null**; if it is, return immediately.
24. Enter a **try/catch** against **NumberFormatException**, and in the **try** block get the text from **txSalary** and parse it as a **double**. If the value you get is not equal to the employee's current salary (that is, only if the user actually changed something), call **eService.updateSalary**, passing **selectedEmployee** and the new value.
25. In the **catch** block, use **ErrorMessage.show** to post an error message box to the user, and then call **updateDetailFields** to reset the salary value, canceling the edit. (See the **ErrorMessage** class in the GUI package; it's a utility to simplify error handling throughout the presentation tier.)
26. Build and test, and see that you can edit salaries; that they “stick” when you edit them – you can go to a different employee record, come back, and your change is still there – and that other views hear about the changes. Test this last assertion by changing an employee's salary so it goes out of range, that is becomes exceptional. The departments view shows this in its tree by a similar red color-coding; you should find that if you make this sort of change while the employee is in view in the departments window, the rendering of that employee in the tree will change automatically, again thanks to MVC. (This is the intermediate answer in the **Step3** directory.)

27. Finally – and in most cases you will not have time to complete this final, challenge step – add more event handling to allow the format of the representation to change from full currency format to a simpler, editable format when the user enters or leaves the text field. Detailed instructions for this are not included here; generally it means making the **SalaryHandler** a listener for **FocusEvents** as well as **ActionEvents**, and resetting the representation in the text control when focus is gained and lost. You can review the final answer, which includes these event handlers, in the **Step4** directory.

Evaluation Only

Design Exercises: Behavioral Refactoring

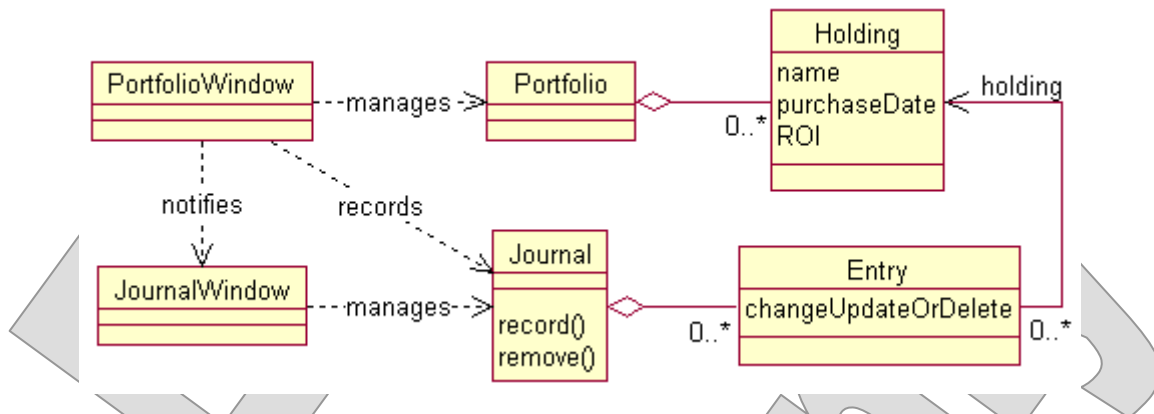
Introduction

In this lab 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.

Suggested Time: 30-45 minutes.

Exercise 1 – Auditing

Consider the following design for a piece of a financial analysis system. The user manages one or more **Portfolios**, which are composed of **Holdings**, and can buy and sell these. For compliance with SEC regulations, the application captures all the user’s activities in a **Journal**, which in turn comprises **Entry** objects. When the user buys or sells, the new **Entry** holds a reference to the affected **Holding** object, along with a boolean that indicates which type of action was taken.



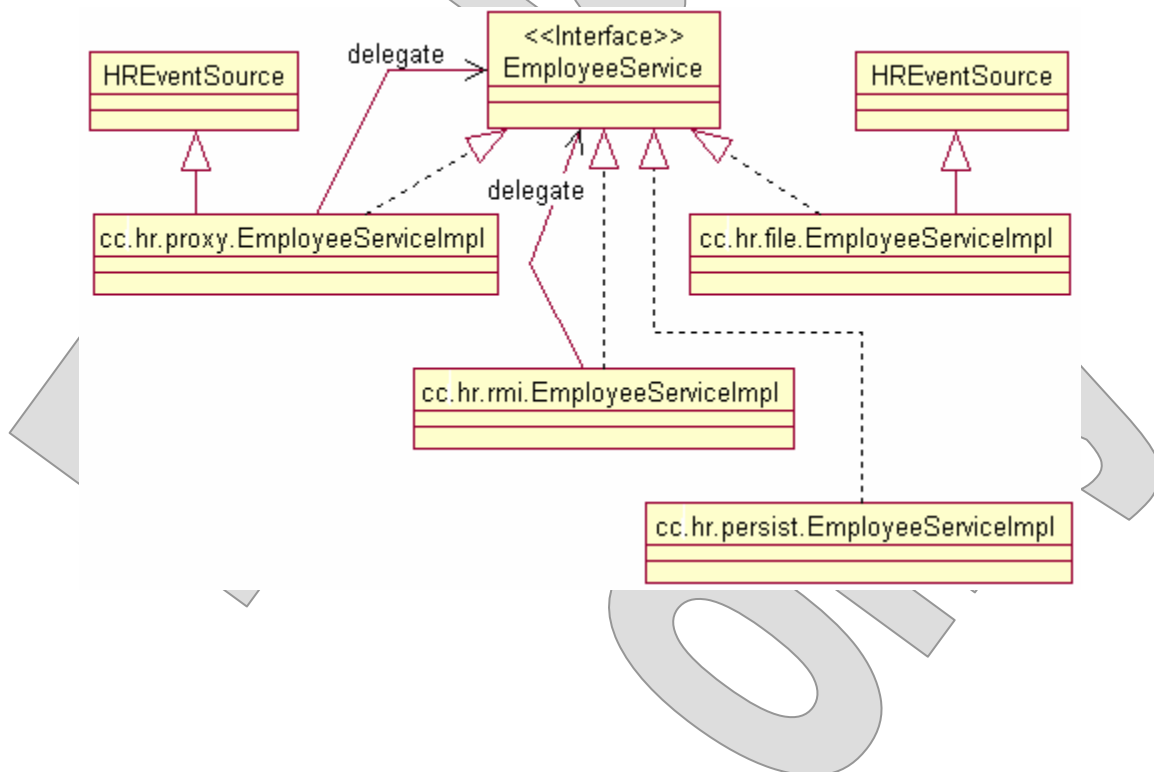
The **PortfolioWindow** allows the user to manage a **Portfolio**, and when the user acts, this window object also **records** in the **Journal**. Separately, a **JournalWindow** shows the current Journal state as a list or table, in reverse-chronological order. The user can also clear records from the journal – a possible breach of trust with the SEC, but we’ll allow it! **PortfolioWindow** notifies **JournalWindow** whenever it updates the **Journal**, which allows **JournalWindow** to refresh its presentation.

Well ... what do you think? What do you like, what don’t you like, and most importantly how can design patterns be applied to this system? What patterns do you see in play already? What patterns should be in force but aren’t, or aren’t being implemented cleanly? There is one in particular that could benefit this system ... Analyze and recommend modifications to the design.

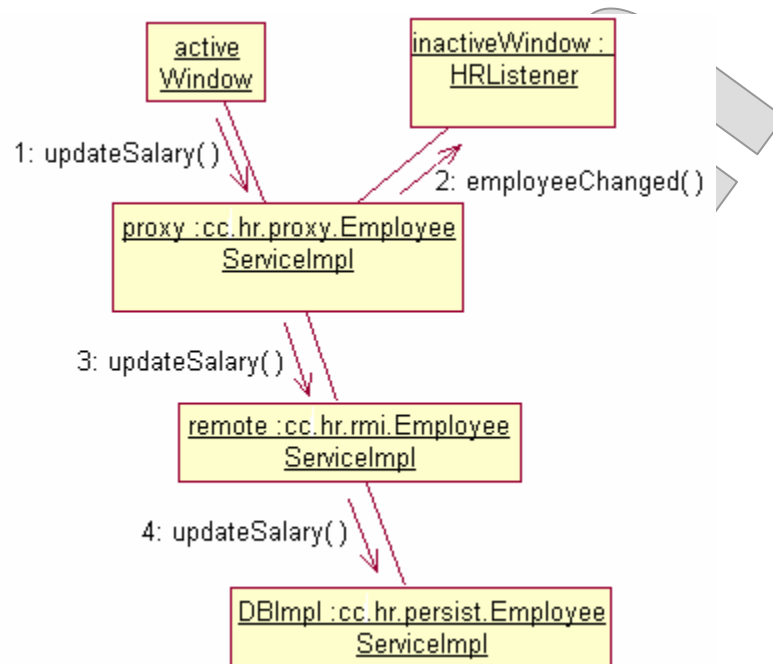
Exercise 2 – Human Resources

Consider another feature of the HR application: a capability to save off downloaded records in a local file, and then use that as a data source in future sessions. This is actually implemented in **Examples\HR\Step1-6**, first for local use, then for use by the RMI server, not the client. But we could include both files and databases: the GUI design would offer both a File menu and a Database menu, such that the application could externalize the database to a local file, using Java Serialization as the persistence mechanism. This might then support a disconnected user, who could make updates and submit them for integration into the database later.

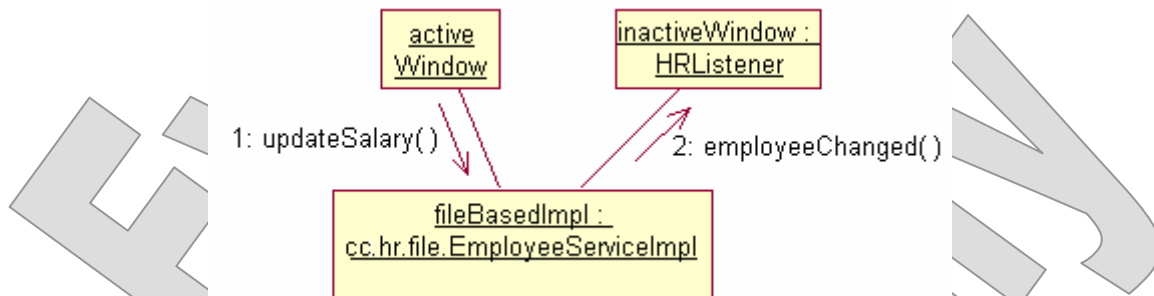
This requires another set of service implementations, that can manage persistent objects stored in a file. A design is shown below that uses the various services as they are currently implemented, but connects them in a new way. The idea is that the **cc.hr.file** implementations can be instantiated locally, manage the local data, and fire HR events via the **HREventSource** interface, which the various windows will handle as they do now via the remote proxies in **cc.hr.proxy**. The **cc.hr.file** and **cc.hr.persist** objects don't delegate the way the others do:



Here are the possible connection paths from the application, shown in UML collaboration diagrams (see Appendix B for a summary of this notation). First, the remote connection to the database services that already exists:



... and the simpler connectivity to the proposed file-based services:



But there's something a bit unsatisfying about the way the file-based services (package **cc.hr.file**) and the local proxies to remote services (package **cc.hr.proxy**) both fire HR events. There's an obvious redundancy there, and why would file-based services fire events while the JDBC-based services (**cc.hr.persist**) don't?

Recommend a refactoring that would result in a cleaner, more maintainable and extensible design.

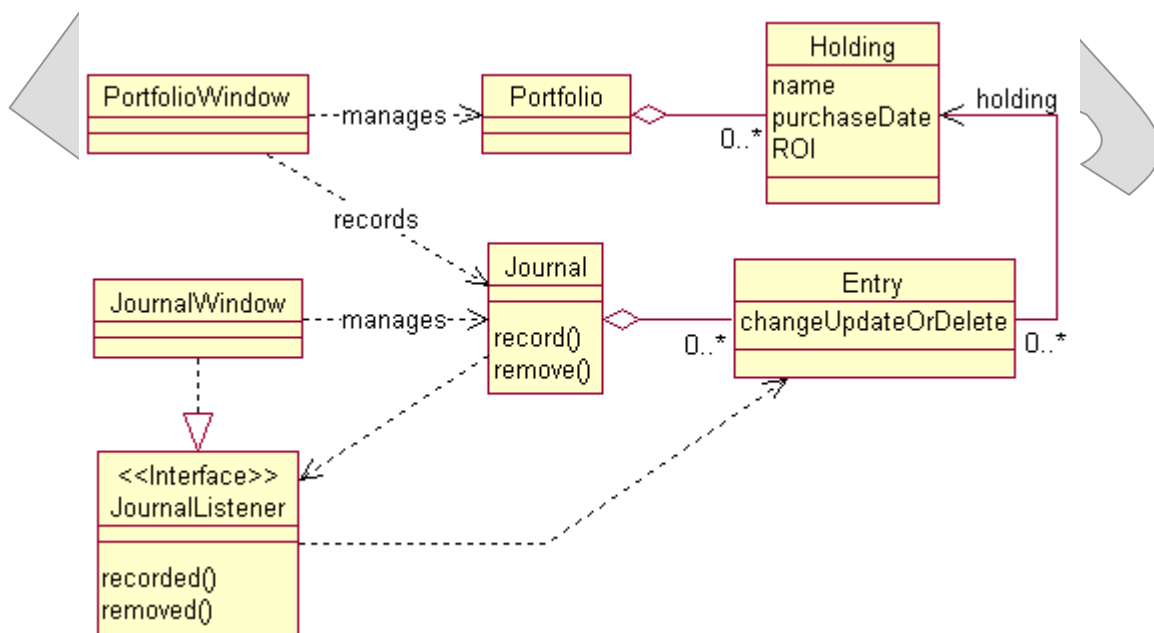
Design Exercises: Answers and Discussion

Exercise 1 – Auditing

What we like: a clear separation of presentation and logic, such that there are window classes that act on separate data objects such as **Portfolio** and **Journal**. We might call this second layer of classes a “model.”

What we don't like: if that's a model, where are the views and controllers? In the use case described, the **PortfolioWindow** acts as a controller on both the **Portfolio** and the **Journal**; **JournalWindow** is both a controller and view on the **Journal**. So far, so good, but why is a controller interacting directly with a view? That is, why does **PortfolioWindow** have to “manually” notify **JournalWindow** every time it changes **Journal**? The MVC pattern says this shouldn't happen. Why is this a bad thing? What if other controllers and views proliferate in this design? How many other controllers might appear over journals? How many other ways of seeing journal information might be defined? We have a maintenance problem wherein the total number of notifications that flow from controllers to views is the product of number of controllers and number of views – that's not going to scale well, and it results in multiple maintenance points and almost always in buggy applications.

The cleanest solution is to define an observer interface like **JournalListener**, and to let **Journal** fire events when it changes. **JournalWindow** subscribes, and **PortfolioWindow** is relieved of responsibility for, and dependency on, **JournalWindow**.



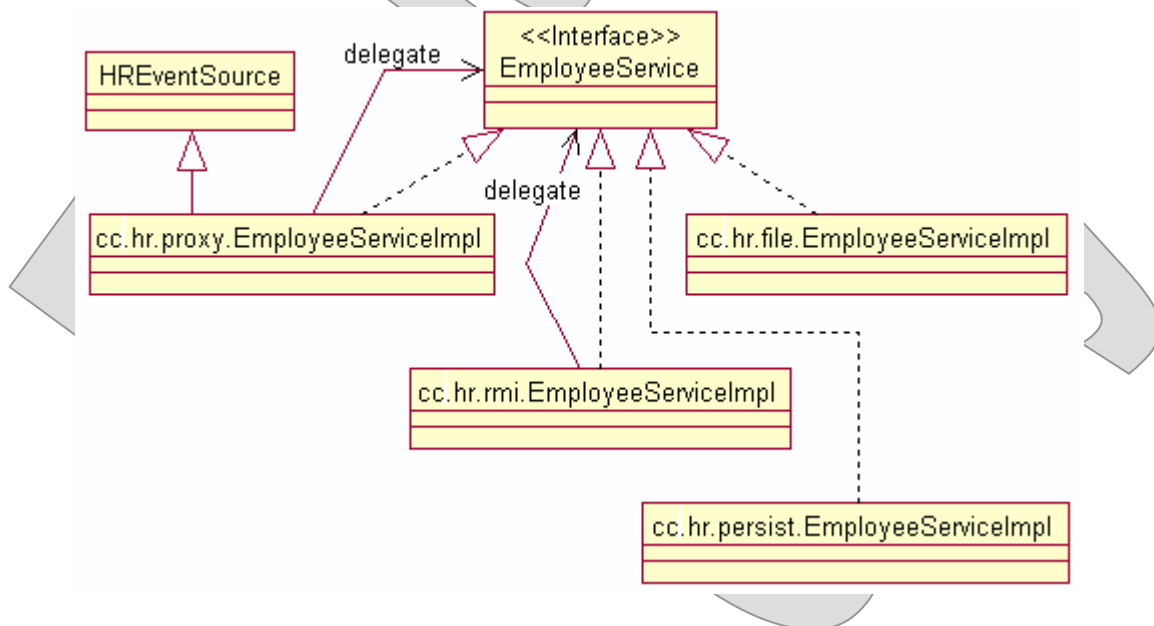
Exercise 2 – Human Resources

The key pattern here is Chain of Responsibility, by which each handler in the chain is given a single, severable responsibility. Viewed through this lens, the problem with the proposed **cc.hr.file** implementations is that they were taking on two responsibilities: persistence and firing model events.

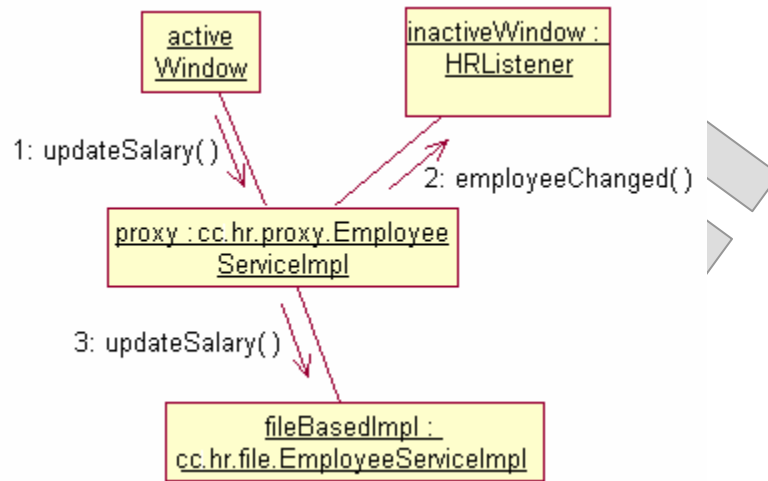
At the same time, the **cc.hr.proxy** services seem to have remote connectivity as one of their responsibilities, but really they are just delegating to some “next” service – it just happens that they are currently wired via a remote connection. (That’s the beauty of location transparency via RMI – or EJB.)

The problem description said that **cc.hr.file** objects don’t delegate, which is true, and correct as far as it goes. But if they don’t delegate, why are they doing the job of the **cc.hr.proxy** objects as well as handling persistence chores? The cleanest solution is actually to relieve the **cc.hr.file** services from responsibility for event firing, and to create a longer chain for the file-based scenario: wire up a proxy to each file based service, which now will be a local connection. Responsibilities are neatly decoupled, and notice that the decision to host either persistent implementation (file or DB) is independent of the choice of local or remote connectivity; we have total flexibility over both choices.

The resulting interface/class design would be:



... and the resulting object chain to support file-based records would look like this:



What's the advantage of this refactoring? Mostly, it's in reduction of maintenance points: if the **HRListener** had to change for some reason – say we add another method for a different sort of notification – how many code changes would we have to make? We're subclassing **HREventSource**, so we've captured the firing behavior in one place, but that isn't always possible; often the add/remove methods and notification code have to be written into each event source. And even as it is, there is duplicate code in the "before" picture, as both the **cc.hr.file** and **cc.hr.proxy** implementations have to get a clone of the listeners list and fire off the notifications in various mutator methods. It's always good to collapse multiple points of maintenance to a single point.