Design Patterns in Java

Will Provost

Version 17

JavaPatterns. Design Patterns in Java
Version 17

A publication of Capstone Courseware, LLC.
877-227-2477
www.capstonecourseware.com

Published in the United States.

This book is printed on 100% recycled paper.

Course Overview

Prerequisites

- **This course is intended for experienced Java programmers; specifically, we expect that the student will be comfortable with the following concepts and skills:**
  - Java as a **procedural language**: data types, flow control, writing algorithms
  - Java as an **object-oriented language**: classes, inheritance, polymorphism, interfaces and abstract classes, exception handling
  - The **functional style** of Java programming, new to Java 8: functional interfaces, lambda expressions, method references
  - Use of the **Collections API**, and therefore some familiarity with Java **generic types**
- **Discussions are pitched to those who've already confronted many of the issues that motivate design patterns.**
  - Patterns reflect best practices acquired by the software community through experimentation.
  - Experienced coders will grasp the importance of a given pattern much more easily than those who've not seen first hand the ill effects of poorer practices.
- **Those with lighter Java experience – and who perhaps are more focused on analysis and design – may still find the course quite interesting and useful.**
  - It may take a bit more discussion and more examples before the usefulness of patterns "sinks in."

Labs

- **The course relies on hands-on experience in various topics and techniques.**
- **All lab code is written to build and run according to the standards of the Java Platform, Standard Edition 17.**
- **Lab exercises are deployed as standard Java projects, all under a common root folder.**
  - You may have gotten these files as a ZIP from your instructor.
  - Or you may have downloaded or cloned a Git repository.
- **Throughout the coursebook we'll refer to projects and other paths relative to that root – wherever you've located it on your local system – for example `Inventory/Step1`.**
  - You can import these projects into your IDE of choice.

Table of Contents

CHAPTER 0

REFACTORING

OBJECTIVES

After completing "Refactoring," you will be able to:

- Identify various sorts of warning signs in your Java code, and the associated outcomes to be avoided.
- Apply better practices as appropriate to clean up observed problems, through code refactoring.
- Design and implement systems that preserve single points of maintenance for application logic and critical information.

A Place For Everything …

- **Before we start to address ourselves to the question of design patterns, in this initial chapter we'll try to establish a foundation of a few good object-oriented practices.**
- **We'll consider several techniques, with a common element: they are all about factoring code properly.**
  - In OO development, there is always **one right place** for a given piece of code, large or small.
  - The term **analysis** means "to break down" a problem into smaller problems, thus arriving at **indivisible concepts**.
  - In OO **implementation**, we find that a larger system ultimately comprises indivisible **factors**: the computation of a price, the retrieval of a piece of data from storage, or the validation of input received from a client or a user.
- **So to factor code is to put each thing in its one right place.**
- **To refactor is to move things from other places – often multiple other places! – to their single, right places.**
  - Refactoring is a **natural part** of iterative development – it's not necessarily a sign that something's gone off the tracks.
  - We often do a job in the **simplest way** first, to see it done.
  - Then we tidy up, re-organize, and **eliminate duplication**.
- **The refactoring techniques we discuss in this chapter each establish a certain relationship between pieces of code that allows those pieces to be singular instead of plural.**

Warning Signs

- **For just about any good practice, there is a negative space – that we might call bad practice, if we were in a mood to be blunt.**
  - You'll also occasionally hear the accusatory term **anti-pattern** – usually implying that something that's been considered good or common practice is actually bad.
- **We'll think more in terms of warning signs.**
  - A **warning sign** is some characteristic of an implementation that we can recognize as indicating a common problem, with bad outcomes for the resulting code, class, or code base.
- **For one very simple example ...**
  - If you see **non-descriptive identifiers** for variables, fields, methods, etc. – such as **ae**, **d**, **d2**, or **x_** – that's a warning sign.
  - The danger is that code becomes **unreadable**.
  - That can make it **unmaintainable**: someone, maybe even the original author, is going to make a mistake because they don't understand what is what.
- **We'll identify some warning signs throughout the chapter, each to be followed by a better practice that will help you to avoid the associated pitfall.**

Design Patterns

- Some of the techniques described in this chapter relate, more or less closely, to design patterns that we discuss later in the course.
- Some don't really relate to any specific pattern but are just good elements of object-oriented practice.
- Where a pattern is involved, we will often identify it.
- But we'll steer clear of any detailed pattern discussions, and leave those for later chapters.

Error Handling and Logging

- **Though neither is exactly a matter of refactoring, error handling and logging practices deserve a quick mention.**
- **Java has standard means of throwing and catching exceptions.**
  - Exactly **where** and **how** you catch exceptions is a matter of design.
  - Regardless, it's critical that you **report** exceptions – to someone, somewhere, somehow.
  - Any piece of code that encounters a checked exception must either **throw** it along to the prior caller or **catch** it – and if you catch it, you take responsibility for it.
  - Handle it, effect a recovery, and/or **log the exception**.
  - Never "swallow" an exception – i.e. catch it without reporting it. This makes for some really maddening debugging sessions!
- **There is also a standard logging API – or you can use a third-party library such as Log4j.**
  - As a habit, **every class** should have its own logger.
  - Log **exceptions** at **warning** or **severe/error** levels.
  - Also log useful information on the progress of a component at **info** level, and configuration details at **config** level.
  - **Lower levels** (debug, fine/finer/finest, trace, etc.) have their uses, too – and be sure not to promote **high-frequency** entries to higher levels, because this can make it hard to filter the log.

Tools

- **Our primary tools for hands-on exercises in this course are:**
  - The **Java 17** developer's kit
  - An integrated development environment (IDE) of your choice
- **The lab software supports two modes of operation:**
  - Integrated building, deployment, and testing in your IDE
  - **Command-line** builds using prepared scripts
- **We'll provide instructions for each approach separately on the next few pages.**
- **Most students will prefer to use the IDE for hands-on exercises, and where there are differences in practice we will lean in the direction of using the IDE.**

Using a Command Shell

- **To compile and run your code projects from DOS, PowerShell, bash, or similar, you'll just need to be sure that your JDK is set up correctly: a JAVA_HOME environment variable and the bin folder in your executable path.**
  - For Example, for Windows:

```
set JAVA_HOME=c:\Java17
PATH="%JAVA_HOME%\bin";...
```

  - Your JDK location will be specific to your machine: this is often found under **c:\Program Files\Java\...**
- **Note that environment changes in an open console don't affect the operating system as a whole.**
  - This can be a good thing, especially if you have another Java environment set up for work. Anything you do in a DOS or PowerShell window is isolated.
  - But remember that any new consoles will need the same setup – or you can spawn one from another and inherit the environment settings, for instance with **start** in DOS.

Using an IDE

- **You can open any of the Java projects (look for a folder with an src subfolder) in your IDE of choice.**
- **Some IDEs will call this "opening" a project, and some will call it "creating" a project on the existing code.**
- **Most IDEs are set up to build open projects automatically, so the "build" step mentioned in some instructions in the course will not be an explicit action on your part.**
- **Then you can run a given Java class as an application.**
  - The user interface for this varies by IDE.
  - This boils down to invoking the **java** launcher in order to run the **main** method in that class.

Warning Sign: Magic Numbers

- **It's often easiest, at first, to write everything in one place.**
- **This can lead to, among other things, writing special literal values into your logic.**
  - A **magic number** such as the expected size of an array or a defined threshold value:

```
int[] slots = new int[12];
for (int s = 0; s < 12; ++s)
  doSomethingWith (slots[s]);
```

  - **Literal strings** to represent concepts, options, modes, etc.:

```
if (mode == "summary")
  writeSummaryPage ();
else
  ...
```

- **There are actually a number of dangers here:**
  - Magic numbers are not **self-descriptive**: out of context, later or elsewhere in the code ... why 12? What's 12 mean?
  - The same literal value is often written multiple times, and this means **multiple points of maintenance**, which in turn leads to a maintenance-induced failure when the values **drift apart**.
  - Magic values are often **not type-safe**: the meaning of most strings of this sort is clear, but what if a well-meaning caller passes a **mode** of "Summary" instead of "summary"?

Static-Final Fields

- **There are at least two ways to refactor such code.**
- **For a variety of magic values, you can declare static fields of the appropriate type.**

```
private static final int NUMBER_OF_SLOTS = 12;
public static final String MODE_SUMMARY = "summary";
```

  - Use whatever visibility modifier makes sense; above, the **NUMBER_OF_SLOTS** may only be relevant to the implementation, while **MODE_SUMMARY** perhaps is a value that can be passed in by client code, or returned to it.

- **Advantages are numerous:**
  - You can use this symbol in a **compiler-checkable** way. So if you were to mistype 132 for 12, the compiler can't help you; but it will know you've goofed if you type **NUMBER_OF_SLOBS**.
  - You can refer to the field from multiple places, making the definition of the field a **single point of maintenance**: more slots? Change 12 to 15 in **one place**, not twenty-nine.
  - The code is **self-descriptive**: it's become obvious why the **slots** array is dimensioned the way it is.

Enumerated Types

- **Where a value must always be one of a set, it's better to use an enumerated type:**

```
public enum Mode { SUMMARY, DETAIL, UNKNOWN };
```

- **This is more type-safe.**
  - With a static final **string**, still the compiler can't save errant code from ignoring the symbol and supplying its own string literal.
  - Only the use (direct or indirect) of one of the symbols in an **enum** can possibly satisfy the compiler when that enum is expected.
- **It also helps to group related values.**
  - For example the **java.util.Calendar** class (which pre-dates support for the **enum** in Java) defines a lot of **int**s:

```
public static final int JANUARY = 0;
public static final int FEBRUARY = 1;
...
public static final int HOUR = 10;
...
public static final int MINUTE = 12;
```

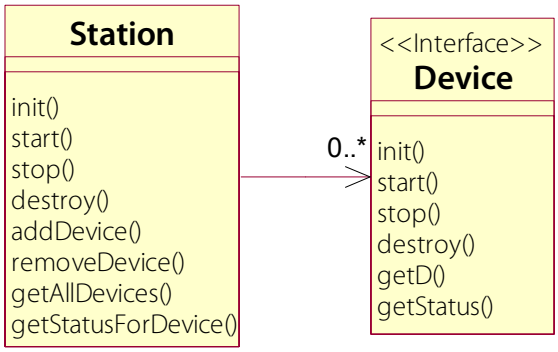
  - It would be easy to pass a literal integer as the month argument in a method call, hoping for the best – or even to pass **HOUR** and mistakenly be saying **NOVEMBER**!
  - The more modern **java.time** API includes, for example, separate enumerations for a **Month** and **ChronoUnit**:

```
public enum Month { JANUARY, FEBRUARY, ... }
public enum ChronoUnit { ... HOUR, MINUTE ... }
```

Enumerating Status Options

DEMO

- **We'll look at an application that exhibits exactly this warning sign, by using strings to represent possible values for the operating status of a hardware device in a monitoring, management, and control (MMC) application.**



  - A **Device** represents a hardware device, with a focus on its lifecycle and status.
  - The **Station** represents one physical location, at which there will often be many **Device**s, which it manages and on which it can report.

Enumerating Status Options

DEMO

- Do your work in **Station_Step0**.

1. See **src/cc/monitor/Device.java**:

```
public interface Device
{
  public String getID ();
  public String getStatus ();
  public void init (Station station);
  public void destroy ();
  public void start ();
  public void stop ();
}
```

2. In **src/cc/monitor/Station.java**, see the two business methods at the bottom – and note the literal strings in the first one:

```
  public Stream<Device> getDevicesInTrouble ()
  {
    return devices.values ().stream ()
      .filter (d ->
        d.getStatus ().equals ("warning") ||
        d.getStatus ().equals ("failed"));
  }

  public String getStatusForDevice (String ID)
  {
    if (!devices.containsKey (ID))
      throw new IllegalArgumentException
        ("No such device: " + ID);

    return devices.get (ID).getStatus ();
  }
```

Enumerating Status Options

DEMO

3. A test application **src/cc/monitor/ShowErrors.java** creates a set of devices, using the default **DeviceImpl** class, which can be configured with an ID and a fixed status, and it adds them to the station:

```
public static void main (String[] args)
{
  Station station = new Station ();
  station.addDevice
    (new DeviceImpl ("Device1", "running"));
  station.addDevice
    (new DeviceImpl ("Device2", "warning"));
  station.addDevice
    (new DeviceImpl ("Device3", "idle"));
  station.addDevice
    (new DeviceImpl ("Device4", "stopped"));
  station.addDevice
    (new DeviceImpl ("Device5", "fail"));
```

4. It then asks the station for a list of "devices in trouble:"

```
  System.out.println ("Devices in trouble:");
  station.getDevicesInTrouble ().forEach
    (d -> System.out.println (d.getID ()));
}
```

5. See the problem?

Enumerating Status Options

DEMO

6. Run the application now.
   - From the command line, use the prepared **compile** and **run** scripts, working in the project directory – as shown below:

```
compile
run ShowErrors
```

   - In Eclipse, right-click the class in the Project Explorer view and choose to **Run As | Java application**; or, with the source file open, hit **Ctrl-F11** to launch.
   - Either way, see the console output:

```
Devices in trouble:
Device2
```

   - No – there should be two devices listed here, at least by the looks of the code – what does "fail" mean, if not that your in trouble?

- **We're just seeing the eventual failure of a system that's designed to use strings – about the most open type in terms of possible values – to capture a small set of possible values such as running, idle, warning, and ... hmm, "failed"? "failure"? "fail"?**
  - What do we call it, anyway?
  - We wouldn't have to guess if there were a pre-defined **identifier**.

Enumerating Status Options

DEMO

7. Define a new enumerated type **Status**. We'll make it a top-level type, so create **src/cc/monitor/Status.java**:

```
public enum Status { RUNNING, IDLE, DIAGNOSTIC,
  STOPPED, WARNING, FAILED, UNKNOWN }
```

8. Refactor **Device** to use this type instead of a plain string:

```
public Status getStatus ();
```

9. This will force a number of changes in the rest of the application – start with **DeviceImpl**:

```
public class DeviceImpl
  implements Device
{
  private String ID;
  private Status status;

  public DeviceImpl (String ID, Status status)
  {
    this.ID = ID;
    this.status = status;
  }

  public Status getStatus ()
  {
    return status;
  }
```

Enumerating Status Options

DEMO

10. Only the business methods at the bottom of **Station** are affected:

```
public Stream<Device> getDevicesInTrouble ()
{
  return devices.values ().stream ()
    .filter (d ->
      d.getStatus () == Status.WARNING ||
      d.getStatus () == Status.FAILED);
}

public Status getStatusForDevice (String ID)
{
  if (!devices.containsKey (ID))
    throw new IllegalArgumentException
      ("No such device: " + ID);

  return devices.get (ID).getStatus ();
}
```

11. Finally, change the test class to use the enumerated values – and, not to put too fine a point on it, but notice that you can't get away with **Status.FAIL** ...

```
... new DeviceImpl ("Device1", Status.RUNNING));
... new DeviceImpl ("Device2", Status.WARNING));
... new DeviceImpl ("Device3", Status.IDLE));
... new DeviceImpl ("Device4", Status.STOPPED));
... new DeviceImpl ("Device5", Status.FAILED));
```

12. Test the updated application and see correct results:

```
Devices in trouble:
Device2
Device5
```

- **This demo continues a little later in the chapter.**

Externalization

- **Where the correct value for an option, threshold, or other "magic number" may not be permanent, or can't be known until runtime, consider externalizing the information.**
  - Some values vary **over time**: a maximum capacity, a range of supported written languages, etc.
  - Some values vary **over space**: deployments of the same application at different physical locations may require different threshold values, legal age requirements, resource locations, and so on.
- **To externalize a value is simply to put it somewhere outside of the Java source code – and not necessarily outside the scope of the deployed application, by the way.**
- **There are many techniques for this – XML files, databases, dependency-injection systems, initialization parameters for Java-EE components – but the grand-daddy of them all is the system property.**
  - Initialize a static field or final variable by calling the **getProperty** method on the **System** class utility:
  - It's fine to have a **default value**, too:

```
public static final int LOCALE =
  System.getProperty ("com.me.locale", "en-US");
```

  - Be sure to **log at config level** the results of your initialization:

```
myLogger.config ("Locale set to " + LOCALE);
```

Finding Resources

DEMO

- A component that loads and saves the user and group records in a security realm will usually need some sort of pointer to persistent storage, and this will often vary by deployed location.
- We'll work in **UserDB_Step0** and see that the starter component looks to a hard-coded file path – another "magic value."
- We'll fix this by loading a system property.
  - The completed demo is in **UserDB_Step1**.

1. Observe the multiple uses of the same literal string in **src/cc/user/Persistence.java** ...
   - In **saveUserDB**:

```
public static void saveUserDB ()
{
  try ( ObjectOutputStream out =
    new ObjectOutputStream
      (new FileOutputStream ("Realm.ser")); )
  {
    out.writeObject (root);
  }
  catch (Exception ex)
  {
    System.out.println ("Couldn't write ...");
    ex.printStackTrace ();
  }
}
```

Finding Resources

DEMO

– In **loadUserDB**:

```
public static Group loadUserDB ()
{
  ...
  File realmFile = new File ("Realm.ser");
  if (realmFile.exists ())
  {
    try ( ObjectInputStream in =
            new ObjectInputStream
      (new FileInputStream ("Realm.ser")); )
    {
      root = (Group) in.readObject ();
    }
    catch (Exception ex)
    {
    }
  }
  ...
```

2. So, clearly this should be refactored to a single symbol, and probably a private static field is appropriate. Let's start there:

```
private static final String FILENAME =
  "Realm.ser";
```

3. Define a helper method to return this value – which we might not bother to do, except that we're going to add some code in a moment to load this value from a system property. But, for now ...

```
private static File getRealmFile ()
{
  return new File (FILENAME);
}
```

Finding Resources

DEMO

4. Replace all the uses of "Realm.ser" with calls to the helper method – and take better advantage of the **realmFile** variable already defined in **loadUserDB**:

```
public static void saveUserDB ()
{
  try ( ObjectOutputStream out =
    new ObjectOutputStream
      (new FileOutputStream (getRealmFile ())); )
  ...

public static Group loadUserDB ()
{
  ...
  File realmFile = getRealmFile ();
  if (realmFile.exists ())
  {
    try ( ObjectInputStream in =
      new ObjectInputStream
        (new FileInputStream (realmFile)); )
  ...
```

5. Now, enhance the helper method to load a configured value as an override, if available:

```
private static File getRealmFile ()
{
  String filename = System.getProperty
    ("cc.user.realmFilename", FILENAME);
  return new File (filename);
}
```

Finding Resources

DEMO

6. Test by running the **cc.user.Load** application, which calls **loadUserDB** and then **saveUserDB** with one additional user.
   - Run it once ...

```
/
  Administrators
    administrator
    wprovost
  gameshowhost
  carseller
```

   - Run it again ...

```
/
  Administrators
    administrator
    wprovost
  gameshowhost
  carseller
  extra
```

   - But, what file did the component use?
   - We can tell by looking at the project directory ... you'll see a new file **Realm.ser** (though you may need to refresh the project as you are working in Eclipse). So it used the default.
   - But in a more complex case, we might be out of luck.

Finding Resources

DEMO

7. Add a logger to the class:

```
private static final Logger LOG =
  Logger.getLogger (Persistence.class.getName ());
```

8. Use it in the helper method – we'll use **INFO**-level logging, just to make it easier to see in a quick test, but then you'd demote this to **CONFIG** level for ongoing use.

```
private static File getRealmFile ()
{
  String filename = System.getProperty
    ("cc.user.realmFilename", FILENAME);
  LOG.info ("Realm filename: " + filename);
  return new File (filename);
}
```

9. While we're at it, use the logger to report exceptions caught in the persistence methods – which in one case was dumping to standard output and in the other (ouch!) was not doing anything at all:

```
catch (Exception ex)
{
  LOG.log (Level.WARNING,
    "Couldn't save realm file.", ex);
}

catch (Exception ex)
{
  LOG.log (Level.WARNING,
    "Couldn't load realm file.", ex);
}
```

Finding Resources

DEMO

10. Test again, first by running normally. You'll again see the "enhanced" realm with the user "extra" – and you'll see confirmation of where it all comes from:

```
cc.user.Persistence getRealmFile
INFO: Realm filename: Realm.ser
/
  Administrators
    administrator
    wprovost
  gameshowhost
  carseller
  extra
```

11. Now try configuring a different location.

   – From the command line, run the following (on one line):

```
java -classpath build
  -Dcc.user.realmFilename=OtherFile.ser
    cc.user.Load
```

   – From Eclipse, edit the existing run configuration and set the property in the "VM arguments" area (on the Arguments tab):

```
-Dcc.user.realmFilename=OtherFile.ser
```

   – See that the component observes the change:

```
cc.user.Persistence getRealmFile
INFO: Realm filename: OtherFile.ser
/
  Administrators
    administrator
    wprovost
  gameshowhost
  carseller
```

Warning Sign: The Dreaded switch/case

- **The switch/case construct is a legacy from the C language, carried into a number of OO languages, including Java.**
- **But to be "oriented to objects" generally means that we encapsulate different behaviors, and let different runtime types of object influence the choice between them.**
- **So switch/case is often a warning sign of a failure to encapsulate – as are some related code structures:**
  - Testing with **if** or the **ternary operator** for equivalence to one of **multiple values**
  - Chains of **if / else if** logic where the conditions are all about testing for specific values
- **The concern is the same as it always is with a failure to encapsulate logic – even small fragments of logic – and that is that the decision-making in that switch/case will proliferate to many places in the code base, instead of being captured once.**

Stateful and Behavioral Enumerations

- **It is not as widely understood as it should be that the Java `enum` is more than a simple naming-and-scoping trick.**
- **Every `enum` is actually a specialized Java class.**
  - It extends **`Enum<E extends Enum<E>>`**, whence it gets all those nice methods: **`valueOf`**, **`ordinal`**, **`name`**, etc.
  - It can be **further defined** with its own **state** and **behavior**.
- **Stateful enumerations can help to**
  - **Assign additional values** to each enumerated value
  - Classify or **qualify a subset**
- **Behavioral enumerations are less common, but can be a great way to encapsulate logic to be run in those different "cases" that otherwise have to live in external `switch/case` passages.**

Enumerating Status Options

DEMO

- Continuing in **Station_Step0**, we'll tighten up one last feature of the application.
- So far, the **Station** class checks for each of two status values on a device as a way of classifying it as "in trouble:"

```
public Stream<Device> getDevicesInTrouble ()
{
  return devices.values ().stream ()
    .filter (d ->
      d.getStatus () == Status.WARNING ||
      d.getStatus () == Status.FAILED);
  }
```

  - But, shouldn't this be native to the concept of **Status**?
  - If we leave it this way, other clients of the **Status** class will have to **duplicate** these criteria, or come up with their own.

Enumerating Status Options

DEMO

1. In **src/cc/monitor/Status.java**, expand the enumerated type to define a field **error**:

```
public enum Status
{
  RUNNING, IDLE, DIAGNOSTIC, STOPPED,
    WARNING, FAILED, UNKNOWN;

  private boolean error;
```

2. Add constructors to initialize this or leave it at a default of **false**:

```
  private Status ()
  {
    this (false);
  }

  private Status (boolean error)
  {
    this.error = error;
  }
```

3. Report the value of the field:

```
  public boolean isError ()
  {
    return error;
  }
}
```

Enumerating Status Options

DEMO

4. Define which status values are considered error conditions by providing arguments to their constructors:

```
public enum Status
{
  RUNNING, IDLE, DIAGNOSTIC, STOPPED,
    WARNING(true), FAILED(true), UNKNOWN;
  ...
}
```

   – The rest will get the default value of **false**, defined in the no-argument constructor.

5. Now you can simply adjust **getDevicesInTrouble** to check to see if the status "is an error"

```
public Stream<Device> getDevicesInTrouble ()
{
  return devices.values ().stream ()
      .filter (d -> d.getStatus ().isError ());
}
```

   – ... and so could anyone else, and be sure to get consistent results.

6. Re-run the application if you like, to be sure the output is consistent with the previous test.

- **The completed demo is in Station_Step1.**

Encapsulating Transaction Logic

EXAMPLE

- **Bank_Flyweight** uses enumerated types liberally to encapsulate certain business rules for a bank.
- In **src/cc/bank/Transaction.java**, the transaction class defines a nested enumeration for transaction type, and that **enum** sets an associated fee per transaction type:

```
public class Transaction
{
  public enum Type
  {
    WITHDRAW      (2.5),
    DEPOSIT       (1.5),
    CHECK          (.75),
    POST_INTEREST (0),
    ASSESS_FEE    (0);

    private double fee;

    Type (double fee)
    {
      this.fee = fee;
    }

    public double getFee ()
    {
      return fee;
    }
  };
  ...
```

Encapsulating Transaction Logic

EXAMPLE

- **src/cc/bank/Account.java** defines a nested type **Standing**, to express the state of an account:

```
public class Account
{
  public enum Standing
  {
    NO_FEES   (1000.0),
    GOOD         (0.0),
    OVERDRAWN (-100.0),
    FROZEN      (null);

    private Double threshold;

    Standing (Double threshold)
    {
      this.threshold = threshold;
    }

    public Double getThreshold ()
    {
      return threshold;
    }
```

Encapsulating Transaction Logic

EXAMPLE

– The **Standing** itself can determine the correct fee – in concert with the **Type** enumeration, in fact:

```
public Transaction getFee (Type xaType)
{
  if (this == FROZEN && xaType != Type.DEPOSIT)
    throw new IllegalStateException ("frozen");

  if (this == NO_FEES ||
      xaType == Type.ASSESS_FEE)
    return null;

  double fee = xaType.getFee ();
  if (this == OVERDRAWN &&
        xaType != Type.DEPOSIT)
    fee *= 2.5;

  return new Transaction
    (Type.ASSESS_FEE, fee);
}
```

– It can even assign a standing to an existing account:

```
public static Standing determineStanding
  (Account account)
{
  double balance = account.getBalance ();
  for (Standing candidate : values ())
    if (candidate.getThreshold () != null &&
        balance >= candidate.getThreshold ())
      return candidate;

  return FROZEN;
}
}
```

Warning Sign: Over-Encapsulating

- **Sometimes as a design issue, and sometimes as a sort of mission creep when coding a class, it can be all too easy to take on more than one responsibility.**
- **A class by its nature should be a "minimal and complete" encapsulation of a single concept.**
- **But some sorts of functionality are tempting to implement within a class even though they expand its role:**
  - **Persistence** – "loadThyself" and "saveThyself" methods
  - **Validation** – "validateThyself"
  - **Presentation logic** or **user-oriented facts** – user-friendly names and symbols, console output, etc.
- **Often we over-encapsulate because it seems that we're in the best position to implement a given feature.**
  - Who knows the state of the class better than the author?
  - What method would be better positioned to work on that state than a member method – that's encapsulation, right?
- **The danger is that the class becomes brittle: unusable except in a specific anticipated context.**
  - What if the expected persistence form is unavailable?
  - What if different output from a validation process is needed, or partial validation is required?
  - What if the presentation style changes, or instead of a human user we're serving a remote piece of software?

Separation of Concerns

- **OO analysis and design is informed by the notion of the separation of concerns: that a given business concept, as encapsulated in a class, need not and should not attempt to encompass other concepts or other features.**
  - Perhaps it's easy to see that, say, a **Person** and that person's **Address** are two different concepts.
  - But, along a different axis if you will, it may be more tempting to include in **Person** the idea of persistence, or validation, etc.
- **Industry experience shows that these are features best handled externally – and often by generic, dynamic frameworks.**
  - And this is often the answer to the question of "who better ...?"
  - These external frameworks can use a variety of advanced techniques to work on classes that they've never seen before – especially by working with the **Reflection API**.

Observable Classes

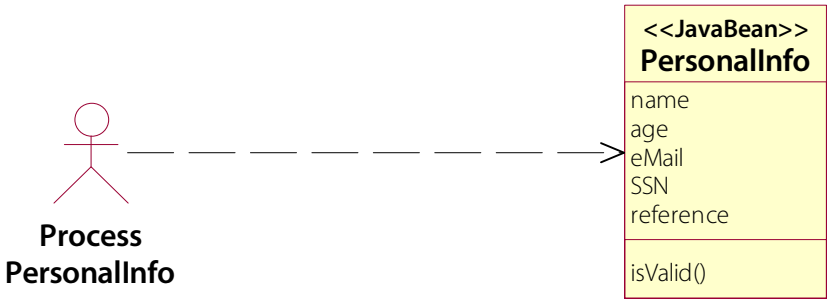
- Sometimes the trap is that we look at a class and think, "who else could implement this?" – and this in turn is an interesting question only because the function in question depends on **private** state elements.
- And often the answer, instead of moving ahead and absorbing a new responsibility, is to make the class **observable**.
  - This doesn't mean making the fields **public**!
  - It just means making sure that there is an **accessor method** – or "getter method" – for each element of state.
- This can open up some sorts of functionality – such as validation – to external classes.
- It's also quite helpful for unit testing.
- Some features, to be implemented, will require not just observability, but **mutability** – and you may not want to have a simple "setter method" for every state element, as that can compromise encapsulation and data hiding.
- Many such tools – persistence frameworks are a good example – work by Java Reflection as a way of (hopefully) getting the best of both worlds.

Validation Logic

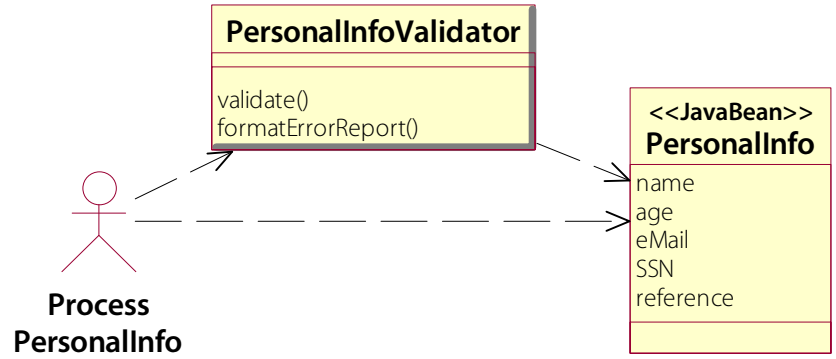
LAB 0

Suggested time: 30 minutes

In this lab you will refactor a JavaBean that validates itself ...



...by building an external validator:



In the process you will improve on the validation logic and especially the error reporting, which in the "before picture" is quite weak.

Detailed instructions are found at the end of the chapter.

Java-EE Bean Validation

EXAMPLE

- In **Validation_Step3** there is a non-working, alternate implementation of validation for the **PersonalInfo** class.
- This approach relies on a Java-EE standard known as **Bean Validation**, which takes an annotation-driven approach.
  - So far we've seen a starter that implemented both validation **constraints** and **error handling internally**.
  - And you built a new version that handled both of these concerns **externally**.
- Bean Validation splits these concerns.
  - **Constraints** are best encapsulated in the class itself, and can be expressed by **annotations** on fields or methods.
  - **Handling** is best handled elsewhere, and a general-purpose validator can enforce a class' stated constraints on any instance of the class, dynamically, and prepare an error report.
  - This is a complex data structure, not that far from the one that you built in the lab, and the caller can decide what to do with it – in much the same way as your client application is now empowered to process your map as it likes.

Java-EE Bean Validation

EXAMPLE

- See **src/cc/info/PersonalInfo.java**, and note that all fields now bear Bean-Validation annotations that express validation constraints for them:

```
@Min
(
  value=18,
  message="Age must be at least 18"
)
@Max
(
  value=120,
  message="Age must be no greater than 120"
)
private int age;

@NotNull
@Pattern
(
  regexp="\\w+([\\.-]?\\w+)...",
  message="Invalid e-mail address"
)
private String eMail;
```

Java-EE Bean Validation

EXAMPLE

- The client now can aim the validator at the target object, pull the trigger, and process any errors as it likes:

```
public static void printReport
  (String label, PersonalInfo info)
{
  System.out.println (label + ":");

  Set<ConstraintViolation<Object>> violations =
    validator.validate (info);
  if (violations.size () == 0)
    System.out.println ("  ... succeeded.");
  else
    for (ConstraintViolation<Object>
        violation : violations)
      System.out.println ("  " +
        violation.getMessage () + ".");

  System.out.println ();
}
```

- To make this example build and run, we'd need the Bean-Validation API and implementation JARs.
- It's beyond our scope to pursue Bean Validation in depth.

Un-Tangling Your Code

- Often the hardest factoring problems involve the smallest chunks of code – especially, where a general, complex process involves sub-processes that should be carried out in specific ways.
- How to achieve single points of maintenance for the common elements of a tightly-integrated algorithm?

```
void doTheUsualThing1()
{
  MyClass result = ...
  preProcess1(result);
  start(result);
  int x = calculate();
  result.x = refine1(x);
  finish(result);
  postProcess1(result);
}
```

```
void doTheUsualThing2()
{
  MyClass result = ...
  preProcess2(result);
  start(result);
  int x = calculate();
  result.x = refine2(x);
  finish(result);
  // no post-processing
}
```

  - We're using method calls as a shorthand for the actual code of a method such as **doTheUsualThing1** – and the implied helper methods already give an idea of how code might be cleaned up.
- Design patterns can help here, and the two that are most apt are the Strategy and Template Method patterns.
- We'll defer in-depth study of these two, but, in brief ...
  - **Strategy** breaks out the specific parts to **helper objects**, which then can use inheritance in their type models.
  - **Template method** defines the specific parts as **abstract methods**, and lets **subclasses** define varying behaviors.

Delegation Instead of Inheritance

- **As powerful as OO inheritance is, it is also easy to over-use.**
- **Sometimes it's better to delegate to a common object than to extend that object's class.**
  - It is often **more work**, and that's part of what makes inheritance so tempting.
  - But it gives **finer control** and more **flexibility**.
- **Strategy is one example of pattern that favors delegation.**
- **Another is Decorator, which can un-block your inheritance hierarchy when you find yourself defining more and more subclasses to encapsulate a variety of overlapping features.**
  - We'll consider Decorator later in the course, as well.

Factories and Dependency Injection

- **One of the toughest things to manage in OO systems is the choices you make about what types of objects to create.**
  - At their best, OO designs use **dependency on interfaces** to preserve adaptability to complex requirements, and to future requirement changes.
  - But in code, somewhere you have to decide what class comes after the **new** keyword.
  - **Hard-coding** the implementation type gives rise to a **dependency on implementation**, which is not always desirable because it binds two implementation types such that one can't live without the other.
- **The family of Factory patterns makes a good start at a solution.**
  - The ultimate effect is to **centralize** type information that is coded into the application, making it more maintainable.
  - Some factories may also **externalize** type information, for example by loading the name of an implementation type as a system property, and then using the Reflection API to instantiate it.
- **Dependency-injection frameworks go farther and bring a more complete benefit in configurability and adaptability.**
  - Potentially **all type information** can be externalized – along with non-type information, such as the magic numbers, resource locations, and other values we've managed in earlier exercises.
  - The whole process of **constructing the system** can be managed from outside the source code of domain classes.

SUMMARY

- **When we question just about any implementation practice in Java – apart from matters of coding style, anyway – we do so because it appears to result in multiple points of maintenance for the same logic or for the same value.**
  - **Magic numbers**
  - Use of **switch/case** and similar code structures
  - **Over-encapsulating** and thus assuming multiple responsibilities (a/k/a addressing multiple concerns) in one class
  - **Failure to encapsulate**, especially where complex algorithms make re-use of small fragments of code a challenge
  - **Over-inheriting** where delegation may be a cleaner solution
- **In many such cases we can identify a design pattern that applies to the problem at hand, and apply it – and that's the core subject of this course and what we'll proceed to study from here.**
- **But even absent any familiarity with patterns, simply keeping the above issues in mind while writing or reviewing your code, and applying more fundamental code-refactoring techniques, can bring significant benefit to the "health" of your code base.**

Validation Logic

LAB 0

In this lab you will refactor a JavaBean that validates itself, by building an external validator. In the process you will improve on the validation logic and especially the error reporting, which in the "before picture" is quite weak.

Lab project: **Validation_Step1**

Answer project(s): **Validation_Step2** (final)

Files: * to be created
src/cc/info/PersonalInfo.java
src/cc/info/ProcessPersonalInfo.java
src/cc/info/PersonalInfoValidator.java *

Instructions:

1. Review **PersonalInfo.java** and see that it is a plain-vanilla JavaBean with five properties, all with getters and setters; and that it offers a method **isValid** which can be used to check that the state of a given instance adheres to certain rules.
2. Review and run the **ProcessPersonalInfo** class, which creates a "good" and a "bad" instance of **PersonalInfo**, and then validates each and reports what it finds.

```
Good PersonalInfo:
Object is valid.

Bad PersonalInfo:
Must include at least first and last name
Age must be at least 18
Invalid e-mail address
Invalid SSN
Please keep reference to 40 characters or less
Object is not valid.
```

Now, what's good and what's not so good here? Good is that the validation is reasonably strict (maybe too strict on for example the regular expression for a person's name) and cleanly implemented; and the error messages are clear.

Not so good? It is inflexible against a number of possibly shifting requirements. What if we want to validate differently for different contexts? Imagine a multi-page workflow in which only some of the values had been submitted at a certain time, and we might want to validate those, without failing because others aren't there yet.

Also, it turns out that the error messages are being written directly to the console by the **isValid** method. What if we want them to go into an HTTP response, or a log? What if we want to process errors programmatically? They're simply not available to the caller.

Validation Logic LAB 0

3. Create a new class **PersonalInfoValidator**.
4. Give it a public, static method **validate** that takes a reference **info** to a **PersonalInfo** object. For the moment, have it return **boolean**, so we can most easily carry over the existing validation logic.
5. Copy the implementation of the **isValid** method from **PersonalInfo.java**, and paste into place here.
6. Now, you'll immediately see a number of errors. First, replace all references to fields with the corresponding accessor method, as called on **info**.
7. In **ProcessPersonalInfo.printReport**, replace the call to **info.isValid** with a call to **PersonalInfoValidator.validate**, passing **info**.
8. Test and see that you get the same results as you did before. That is the basic job of separating out concerns done; and now that we have the validator as a separate class, we can start to improve the validation logic in ways that would probably not be worth the time or trouble in individual target classes ...
9. Change the return type of **validate** to be a map whose keys are strings and whose values are lists of strings. This is a typical data structure for reporting validation errors: the keys are the names of fields (or perhaps even dot-separated "paths" to fields on sub-objects) and the values are lists of error messages – so that we can assign messages to specific fields, and also have more than one per field if appropriate.
10. Instead of printing to the console, we're going to want to add error messages to this map. We'll do this enough that it will be worth providing some helper code. At the top of the method, define a local class **Errors** that extends **TreeMap<String,List<String>>**.
11. Give this class a method **add** that takes two strings: a **key** and a **message**.
12. In the method body, check if the **key** is already in the map. If not, put a new **ArrayList<String>** at that key in the map.
13. Now, regardless, **get** the list for that **key**, and **add** the **message** to that list.
14. In **validate** itself, declare a variable **errors**, of type **Errors**, and initialize it to a new instance of the local class.
15. Now, replace each conditional block in which you write to **System.out** and set **result** to **false**, with a call to **errors.add**, passing a string representation of the name of the property that failed (e.g. "name", "eMail") and the appropriate error message.
16. Remove the declaration of the **result** variable.
17. Instead of returning this value, return **errors**.

Validation Logic LAB 0

18. Now we can process errors in whatever way we want, which is great. But the existing application just wants what it had before, which is console output.

    Add a static method **formatErrorReport** to the validator class, taking a map from string to list of string (the same as the return type from **validate**) and returning a string.

19. In this method, loop over all keys in the map, and print a heading for each.
20. Then, inside that loop and after printing the heading, loop over all the error messages for that key, and print the message, indented slightly.
21. Change **ProcessPersonalInfo.printReport** so that it not only calls **validate**, but then passes the results to **formatErrorReport**.
22. Test again and see your formatted report in the console.

    So now the caller that wanted that simple console output can have it, easily; but other callers can choose how to handle errors: process the formatted report differently? process the raw map entries themselves? Whatever is appropriate.

    And partial validation is possible in a few ways – either build a different validator, or add logic to this validator to support "phases" or "groups" of fields and then have the client indicate what to validate and what to leave alone.

Chapter 1

Recognizing and Applying Patterns

OBJECTIVES

After completing "Recognizing and Applying Patterns," you will be able to:

- **Describe the usefulness of design patterns in OOAD, and Java software design in particular.**
- **Explain how pattern-based thinking can become part of:**
  - An iterative design process
  - A refactoring process
- **Distinguish between problem-specific design considerations and the more abstract thinking that leads to pattern recognition and application.**

Design Patterns

- **Patterns provide a means of identifying and cataloguing common design problems and their solutions.**
- **The seminal work is the "Gang of Four" text[1], to which we'll make frequent reference.**
- **Why do we strive to think in terms of patterns?**
  - If OOAD is about reusing **artifacts**, such as classes, patterns are about reusing **experience and ideas** – even if the code from an earlier application of a pattern is not reusable for a later application of the same pattern.
  - With patterns we **recognize problems** and **arrive at solutions** more quickly.
  - Patterns capture **lessons learned** by the software-development community – sometimes learned the hard way, by getting a design wrong and observing the ill effects.
  - Patterns help to define a **shared vocabulary**.
  - *Head First Design Patterns* makes an excellent point, as well: that applying patterns helps one "stay 'in the design' longer."[2]

[1] Gamma, Helm, Johnson, Vlissides, *Design Patterns*, Addison-Wesley.
[2] Freeman, Bates, Sierra, Robson, *Head First Design Patterns*, O'Reilly Press.

Pattern Catalogs

- **There is no design-patterns standards body.**
- **Patterns are often defined formally, but they are shared informally, though textbooks, whitepapers, and simple word of mouth.**
- **There are attempts – more and less successful, more or less popular – to gather design patterns into comprehensive catalogs.**
  - The **Gang-of-Four** text was the first widely successful pattern catalog, defining 23 fundamental patterns.
  - These apply well to all object-oriented languages.
  - Sun Microsystems spearheaded efforts to define a catalog of **Java EE patterns**.[3] Many of these have been incorporated, over the years, in Java-EE standards and open-source frameworks.
  - There are many textbooks on the subject, each with a proposed catalog, and a great deal of overlap.
  - Developer organizations such as The Server Side have online catalogs that intersect all of the above.

[3] Alur, Crupi, and Malks, *Core J2EE Patterns: Best Practices and Design Strategies*, Sun Microsystems.

Defining a Pattern

- **Most catalogs define a pattern by at least these attributes:**
  - **Name** – and there can be confusion when multiple "recognizers" choose different names, but the goal is to develop the name as a unique key
  - **Problem** – how do we recognize that this pattern is in play?
  - **Solution** – how do we solve the problem, in general?
  - **Strategies** – specific approaches to implementing the solution – these often are variations on the solution as theme
  - **Consequences** – what good and bad things obtain when this pattern is put to use
  - **Related patterns** – some patterns are implemented in terms of others, or are similar to others but with critical differences, or are often found in conjunction with others
- **The GoF text goes much deeper, as do many others.**
- **Though this course is not a pattern catalog, but rather a tutorial, we will touch on many of these aspects of patterns as we discuss them.**

Unified Modeling Language

- **Patterns are naturally geared towards object-oriented analysis and design (OOAD).**
- **Problems and solutions are described in OO terms.**
  - They are so much about **structure and relationships** that learning them from the perspective of implementation code is difficult and seldom leads to complete understanding.
- **We use the Unified Modeling Language, or UML.**
  - UML is a **graphical language** – a notation for designing, documenting, or commenting on OO software.
  - UML is a **consolidation** of several precursor notations. It is managed as a standard by the **OMG**.
  - We will primarily use the UML **class diagram** as a way of describing static structures.
  - To a lesser extent we'll use UML **interaction diagrams** to show actual instances and their interactions.
- **See Appendix A for a UML primer and reference.**
- **So that UML can be a tool for understanding patterns – and not a veil thrown between the student and the pattern structure – we will connect simple UML diagrams to code and other practical facts.**

Seeing Patterns

- **How and when do patterns come into play?**
- **The classic OOAD process is iterative and involves at least these steps:**
  - Requirements
  - Analysis
  - Design
  - Implementation
- **The trick is that none of these steps is truly discrete.**
  - Software development is a creative process, and creative people will naturally think ahead.
  - Often the analyst can anticipate the design, or the designer can envision the implementation. The challenge is to consider the impact of decisions on later steps, without jumping to conclusions.
- **Pattern recognition can come into play anywhere in the process, though it is most appropriate to the design phase.**
  - Fluency in patterns can help analysts and designers anticipate later stages of decision-making with greater confidence.
  - This is part of how patterns help to stay in the design phase later in a given iteration, which saves implementation effort, and increases the reliability of that effort.

Warning Signs

- **Patterns are also used heavily in refactoring existing systems.**
  - Something about existing software may be observably **dysfunctional**: poor performance, buggy behavior, etc.
  - Or it may be working well as it is, but certain parts of the design are starting to show their weaknesses: it has become clear that they **won't scale well**.
  - The term **scalability** may be familiar in dealing with large software systems, and it generally describes how well a system performs against an increasing load.
  - That load might be a high volume of persistent data, or a high frequency of client requests for a web application; for our purposes it is **size of the application** itself – perhaps measured by the number of classes in the code base.
- **Patterns can facilitate re-thinking the existing design.**
- **For many patterns, there are warning signs – indications in code or design artifacts that a given pattern would improve that part of the system.**
- **We'll highlight some of these warning signs as we study individual patterns.**

Pitfalls

- **Regardless of how a pattern is recognized and applied, it may pose some tricky problems when it comes time to express or implement the pattern.**
  - We sometimes use the term "to express" to describe the application of a pattern to a **design**; contrast this with "to implement", which means reducing to practice through coding in a programming language.
- **We call these pitfalls; by contrast to warning signs they obtain only once a pattern is recognized and applied.**
  - Sometimes in solving one problem, we create another
- **Sometimes the recognition of a pitfall will avoid wasted effort and recoding.**
- **Sometimes it will lead to code or design refactoring in itself: i.e. the pattern was the right one, but it was executed poorly.**
- **We'll mention a few pitfalls for individual patterns.**

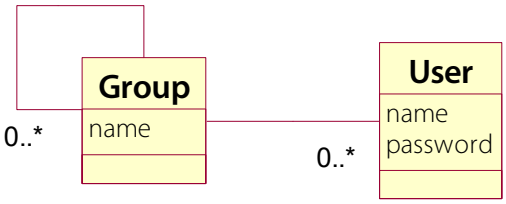
A Design Exercise

- **Let's consider a microcosmic OOAD iteration, with small, simple requirements, and see how patterns can apply to that process.**
- **We'll design the object model for a simple user database that meets the following requirements:**
  - **Users** can be authenticated based on user **name and password**, and then can be authorized to do things (by some external security manager – we won't try to model this externality).
  - **Groups** have **names** and are **collections of users and/or other groups**, and can also be authorized.
- **How would you tackle this problem?**
  - You may want to take a few minutes individually to sketch out a design that meets the above requirements.
  - Use UML if you're comfortable with it, or another blocks-and-arrows notation for diagramming, or perhaps write pseudo-code – or combine approaches.
- **What do you observe, most generally, about the problem, and possibly about your solution to it?**
  - Can you draw analogies from users and groups to other problems or concepts?
  - What aspect of the users-and-groups problem is analogous to those other problems?

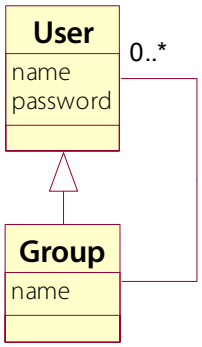
Unsatisfying Solutions

- **You modeled at least two classes, User and Group.**
- **You may have considered a few design options that don't solve the problem as elegantly as possible:**

1. **Group** as a collection of **User**s. But **Group** would also have to have a separate collection of **Group**s, which seems clunky:

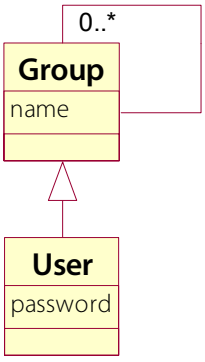


2. We want **polymorphism**, yes? So, should one type specialize (extend) the other? How about **Group** as a subclass of **User**?



   – This doesn't quite hit the nail on the head: why does **Group** inherit a **password** field? And there are probably **User**-specific methods as well, which wouldn't apply to **Group**. So it's a **false specialization**.

3. What about going the other way? This is less intuitive, but has been attempted: **User** as a **Group** that has no children, but adds a password.

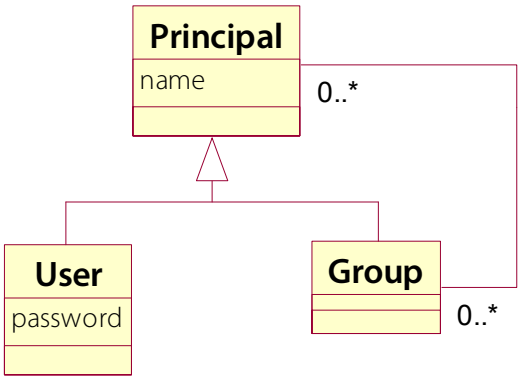


   – This is really no better than solution #2: now it's **User** that's inheriting things that aren't really applicable.

   – **Hmm …**

A Better Solution

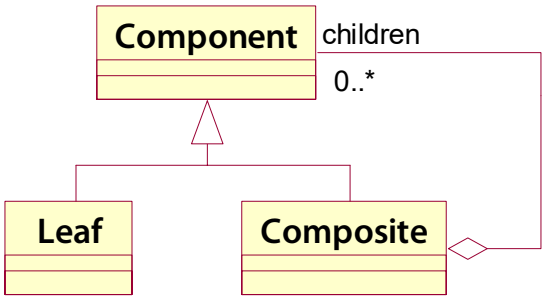
- **The scales fall from our eyes when we realize that what we really need is a third concept in our design: a common base type such as Principal generalizes users and groups.**



  - This allows each subtype to capture what is unique to it: users have credentials such as **password**, and groups have membership.
  - This allows each subtype to capture what is unique to it: users have credentials such as **password**, and groups have membership.
  - But it also provides a target type for group membership, giving us the polymorphism we need.
- **An implementation of this design is found in UserDB_Step1, and we'll return to consider this example in a later chapter.**
- **How does this prompt us to think about patterns?**

The Pattern in Play

- **What is common to this problem and several others is the need to model hierarchies of information.**
  - It is not just group having users; groups can have other groups as members, and so on, to an arbitrary depth.
- **This common aspect of a problem leads to a commonly known, high-quality solution …**
- **That's a pattern, folks! The Composite pattern, to be more specific:**



Studying Patterns

- **We'll now proceed to cover patterns by their GoF categories.**
- **Therefore, each of the next three chapters covers:**
  - **Creational** patterns
  - **Behavioral** patterns
  - **Structural** patterns – including Composite, which will cover in depth at that time
- **All of these patterns are applicable to most or all modern high-level programming languages, but especially object-oriented languages such as Java.**
- **We'll consider design implications and specific implementation issues for the Java language, and many of our labs will focus on these.**

An Unbalanced Approach

- **Rather than distributing our time evenly over the Gang-of-Four patterns, we will cover some more deeply than others.**
  - Some patterns are simply more important!
  - Also, it seems better to investigate a few patterns in depth, taking more time with those few than we could afford to take on every pattern that we want to cover.
- **So we'll invest in a few patterns with more discussion, design exercises, and coding labs.**
- **Other patterns we'll cover with significant discussion but no hands-on exercise.**
- **Finally, some patterns will not be covered – usually for one of two reasons:**
  - The pattern is not heavily practiced.
  - The pattern has been so thoroughly absorbed into the Java Core and extension APIs that it is second-nature to all Java design – this for example is the thinking on Iterator and Proxy.

Tools

- **Our primary tools for hands-on exercises in this course are:**
  - The **Java SE 8** developer's kit
  - The **Eclipse** integrated development environment (IDE)
- **These tools should all be installed on your system, along with the lab software.**
- **The lab software supports two modes of operation:**
  - Integrated building, deployment, and testing, using **Eclipse**
  - **Command-line** builds using prepared scripts
- **We'll provide instructions for each approach separately on the next few pages.**
- **Most students will prefer to use Eclipse for hands-on exercises, and where there are differences in practice we will lean in the direction of using the IDE.**

Environment: Ant and the Command Line

- **To work from the Windows/DOS command line, start by finding the script Capstone\JavaPatterns\Admin\SetEnvironment.bat:**

```
set JAVA_HOME=c:\Java8
PATH="%JAVA_HOME%\bin";c:\Windows\System32;%~dp0
```

  - Edit this file now only if your JDK is installed somewhere other than **c:\Java8**.
  - Once you've made any necessary tweaks, run the batch file:

```
SetEnvironment
```

  - Be sure to run this in any DOS console that you use; environment variables defined in one console are not exported to the whole OS.
  - Or, just use the **start** command from one console to get another with the same environment.

- **You should be able to invoke the Java compiler – which is called javac – from any working directory in a command console, and get output substantially like what's shown below:**

```
javac -version
javac 1.8.0_nn
```

  - Oracle releases periodic **updates** of the JRE and JDK, and the update number will show as ***nn*** in the listing above.
  - The differences between updates under a given edition should not be significant for our purposes in this course.

Environment: Ant and the Command Line

- **For Mac, Linux, and other systems, you can use the script Capstone/JavaPatterns/Admin/SetEnvironment:**

```
export JAVA_HOME=$(/usr/libexec/java_home)
PATH=$PATH:$JAVA_HOME/bin
  :$HOME/Capstone/JavaPatterns/Admin
```

  - You may want to edit this file, for example if your default Java home is not the JDK 8 used in this course.
  - The path will need attention, too, if you installed somewhere other than your **$HOME** directory.
  - Be sure to **source** the script so it affects your current shell.
- **Or, set environment by your own preferred method (profile, .bashrc, etc.) – just assure that the following environment variables are set globally:**
  - **JAVA_HOME** must be set to the root of your JDK 8.
  - The **executable path** must include the Java **bin** directory and **Capstone/JavaPatterns/Admin**.
- **You should be able to invoke the Java compiler as described on the previous page, and see similar results.**

Environment: Eclipse

- **When starting Eclipse, you can select the prepared workspace for this course module at:**

`Capstone/JavaPatterns`

- **You'll see a tree of working sets and projects in the Project Explorer – one project for each exercise in the course.**
  - There is a resource project **Docs**.
  - Most projects start out **closed**, and **build automatically** is set, so it's good to keep the number of open projects to a minimum.

Functional Programming

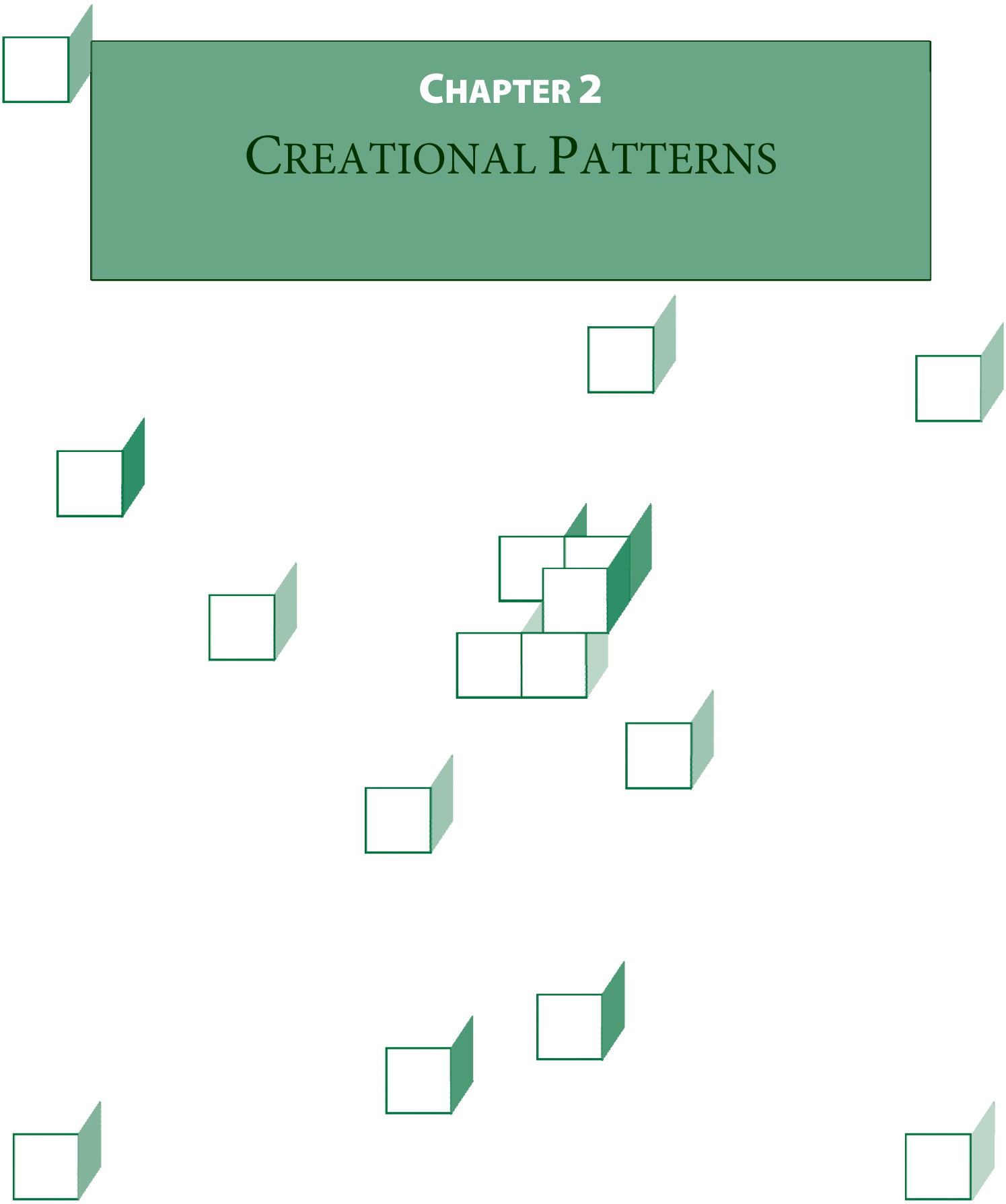
- **Java 8 brings a functional programming style to the language.**
  - The **functional interface** is any interface that has exactly one method.
  - This is important because only functional interfaces can be implemented in the more code-efficient styles of **lambda expressions** and **method references**.
- **Functional programming style has its own impacts on the expression of certain design patterns in Java.**
  - Essentially, any pattern that calls for the use of interfaces might be applied in such a way as to use functional interfaces.
  - Strategy, Observer, Factory Method, and Adapter are a few interesting candidates, all of which we'll see in this course.
  - We'll take a harder look at functional programming specifically with the **Observer** pattern.
- **Java 8 also introduces the Stream and related classes, providing a compelling new alternative to traditional use of the Collections API that supports deferred processing and parallelism.**
  - Collections of course are essential to most Java coding, and inherent to the application of many patterns in Java.
  - Builder, Adapter, Iterator, and Visitor can be affected, among others.
  - We'll explore **Adapter** specifically, by way of a code exercise.

SUMMARY

- Patterns capture past design experience.
- They help the developer "get it right the first time," even when confronting a new problem or application.
- Patterns promote reuse, by gathering what is common to a class of solutions, giving it a name, and promoting it as part of a shared design vocabulary.
- Some patterns can be recognized by warning signs in existing designs or code – "what not to do."
- Then, use of some patterns can lead to new problems, which we call "pitfalls" and try to avoid in the ways we apply patterns to real requirements.

Chapter 2

Creational Patterns



OBJECTIVES

After completing "Creational Patterns," you will be able to:

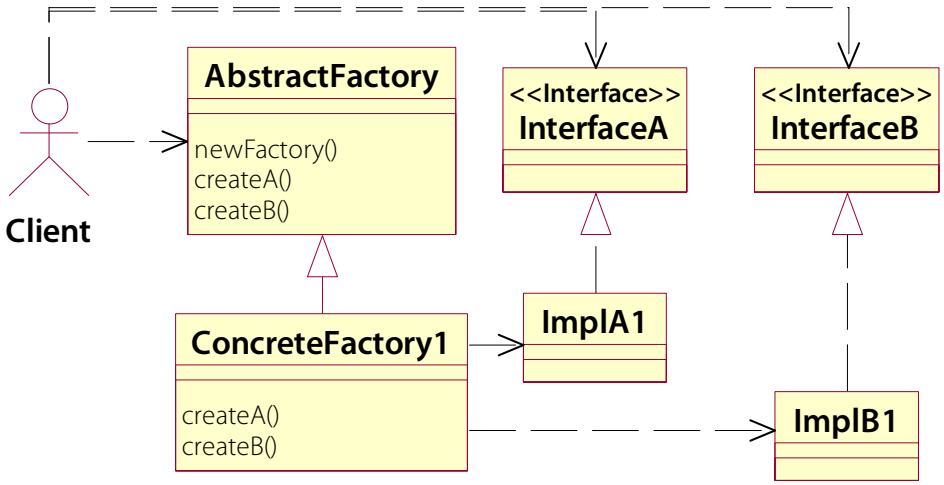
- **After completing this unit you will be able to recognize and apply the following patterns in designing Java software:**
  - Abstract Factory
  - Factory Method
  - Singleton
- **Gang-of-four creational patterns not explicitly covered in this course are:**
  - Builder
  - Prototype

Factory Patterns

- **There are several patterns that address the problem of object creation and are loosely considered as one Factory pattern.**
  - In the GoF text, there are three distinct but related patterns, Abstract Factory, Factory Method, and Prototype.
- **The motivating problem is quite simple: we want to get and to keep control over object creation.**
- **Most OO languages provide a simple means of instantiating a class – in Java there is just one way, using new.**
- **Often we want to put one class in control of the creation of instances of another class.**
  - We might want to manage a fixed **pool of objects**, to keep the total count down and allow object recycling.
  - We might want to **hide the actual implementation** of an expected interface or class from its user – for instance to allow pluggable implementations of a standard API.
  - We might want to assure **coherence** in the choices of implementations of related interfaces.
- **If we allow the caller to use new to instantiate objects directly, we lose this control, and can't implement the above policies reliably.**

Factory Patterns

- **One solution is the Abstract Factory: define a separate class whose responsibility is to create objects from a family of target classes:**



- **The simpler Factory Method pattern just calls for a method in a given class, rather than a separate class full of such methods.**
  - Factory Method is often more appropriate when there is only **one abstract target type**.
- **Factory implementations are among the most varied and hybridized among all patterns.**
  - This is not a one-size-fits-all pattern, if there is such a thing.
  - Especially, there may or may not be an **interface/implementation split** in every case – for target types or for the factory
  - An "Abstract Factory" with no factory interface really devolves to a class with several Factory Methods.

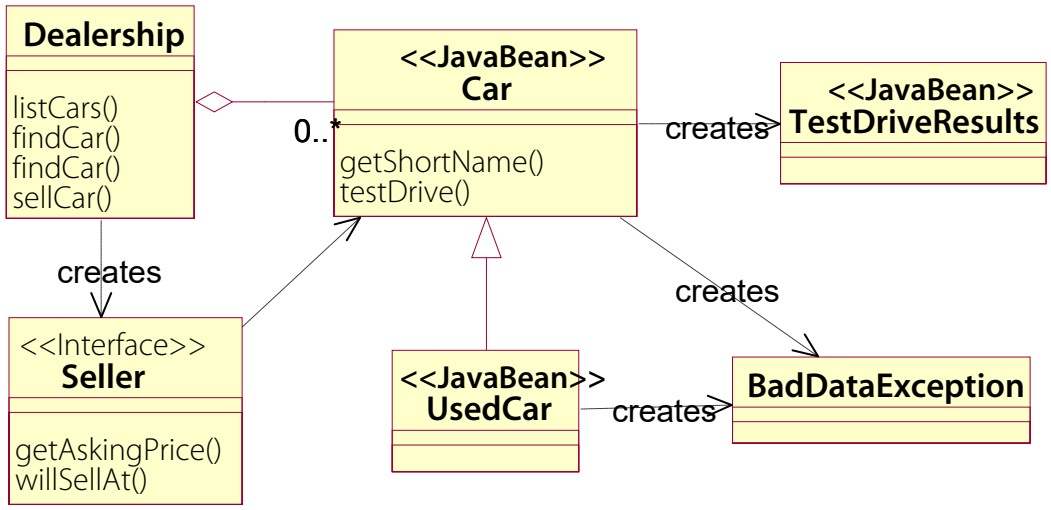
Factory Patterns

- **Key strategies to implement Abstract Factory:**
  - **Hide the constructors** of target classes.
  - Thus callers outside the desired scope can be forced to use your factory classes and methods to create objects, which gives you a critical "hook."
  - Though it may seem to be overreaching, don't make the factory classes directly instantiable either!
  - Use a Factory Method there, so that the **AbstractFactory** can control the choice of **ConcreteFactory** to begin the process.
- **Rigorous implementation of factory patterns can be surprisingly complicated.**
  - The trick to enforcing use of factories is to make the target class' constructor unavailable to the client, but available to the factory.
  - Thus the factory (method or class) is in some way **privileged**.
- **In Java, the common options for privilege are based on visibility constraints.**
  - A **private** constructor necessitates a static factory method on the target class itself – this is rarely useful except in implementing Singleton or Flyweight patterns.
  - Factories and targets often share a **package**, and this can become an important constraint in package design.

Car Dealership Factories

EXAMPLE

- The Car Dealership application, in the **Cars** project, exhibits several variations on the factory theme.



  - When a caller wants to buy a car, it calls **sellCar** – and receives a newly-created **Seller**, which then carries out the process of negotiating a price.
  - In a smaller way, the **Car** creates a **TestDriveResults** object as a carrier for multiple return values.

Car Dealership Factories

EXAMPLE

- In **src/cc/cars/Dealership.java**, take a look at the code that creates the seller.

```
public Seller sellCar (Car car)
{
  Seller seller = null;
```

  - First, it allows a type to be configured explicitly; this is especially useful for testing. Otherwise, it chooses at random:

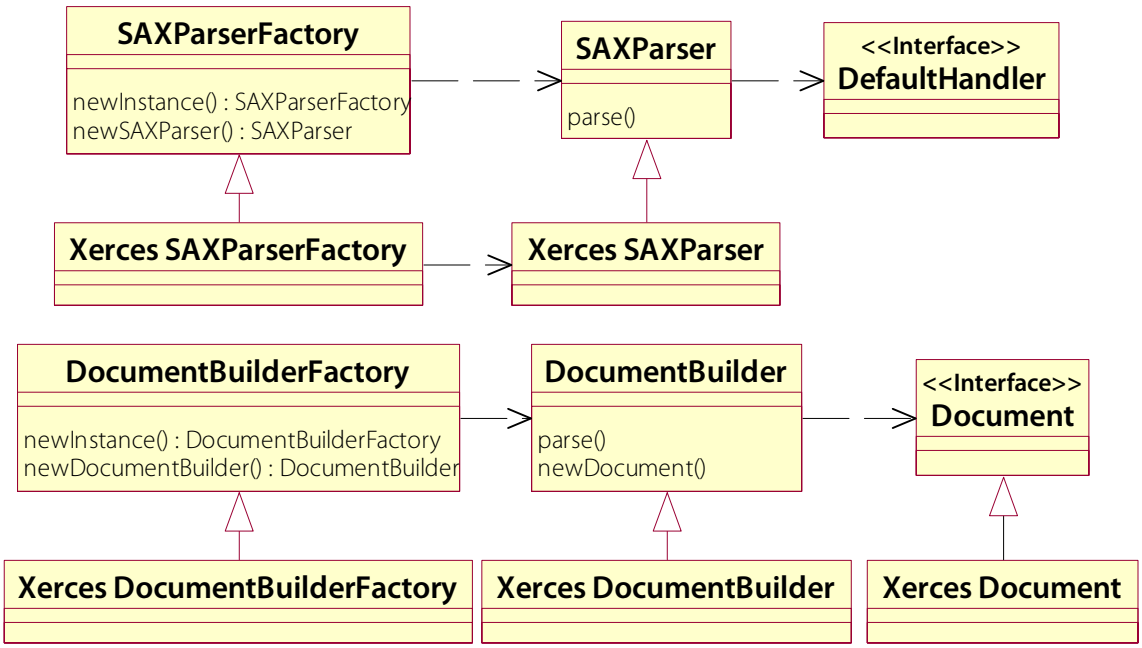
```
String type =
  System.getProperty("cc.cars.Seller.type");

if (type == null) { /* choose at random */ }
else LOG.config ("Configuration ... " + type);
```

  - It then uses Reflection to instantiate and configure the seller – and this makes it adaptable to new implementations as they may be created in the future:

```
try
{
  Class<?> sellerClass =
    Class.forName (type);
  Class<?>[] params = { Car.class };
  Constructor<?> ctor =
    sellerClass.getConstructor (params);
  Object[] args = { car };
  Object obj = ctor.newInstance (args);
  seller = (Seller) obj;
}
catch (ReflectiveOperationException ex)
  { /* logging */ }
...
return seller;
}
```

Other Factory Examples

- Other examples of factory patterns include:
  - Most modern Java and Java EE APIs, especially in the XML and Web-services areas
  - The Java SE **logging API**
  - Distributed object systems such as **RMI** and **CORBA**
- Here's an overview diagram of the parser factories in the JAXP – see API docs for **javax.xml.parsers**:



  - A bit more on this factory expression later.

APIs and Providers

- **Another challenge of the Abstract Factory and Factory Method patterns is that they threaten to introduce dependencies from abstract factory classes to concrete subclasses.**
  - The method that creates a concrete factory is implemented on the abstract factory: a **chicken-and-egg problem**.
  - For example, in the preceding lab, **Factory.createFactory** must be written to choose a specific concrete factory class.
- **Often the practical solution involves some external choice of concrete factory, read and observed by the abstract factory. Common strategies use:**
  - A **system property** for a Java SE application
  - A **deployment setting** in a Java EE application
- **These strategies allow for maximum decoupling between client code and the implementation.**
- **This is sometimes called a provider architecture.**

APIs and Providers

- **Many Java SE APIs offer a provider architecture which is much like a formal expression of Abstract Factory, but with additional logic to discover a default choice of concrete factory at runtime.**
- **JAXP factories apply a heuristic when they are loaded:**
  - They will observe **system property** settings, if found.
  - Failing that they will observe a standard called the **Service API**, discovering declarations in the **manifest of a provider's JAR** file, essentially stating that the JAR includes a JAXP implementation.
  - It's **first-come, first-served**: by rearranging the runtime class path, a different parser can be designated as the default.
- **Many other APIs follow some variant on this provider theme, including:**
  - **JDBC**, or Java Database Connectivity
  - **JCA**, or Java Cryptography API
  - **JavaIDL**, which is the API for implementing CORBA objects in Java
- **Where either client code or an abstract factory does hard-code a choice of concrete factory, it's best to do so via the Reflection API, thus avoiding a compile-time dependency on a particular provider.**
- **In a moment we'll see a double-duty example that shows both this reflection-based technique for a Factory Method, and the Singleton pattern as well.**

The Singleton Pattern

- **The Singleton pattern addresses the need for there to be one and only one instance of a class.**
  - This is often a requirement related to **global** objects in a system: a **manager** of other high-level objects, or a **registry** for drivers, or a **dictionary** to translate certain terms
- **A warning sign for this pattern is the use of new to create certain sorts of objects:**
  - Objects that clearly **relate to the VM** – e.g. display metrics
  - **Heavyweight** objects – that use a lot of memory or perform slow tasks like remote requests
  - Objects that **spawn threads** – this may or may not be a problem, but it's worth considering how object proliferation might result in an unwanted proliferation of threads
  - **Registries or repositories** – forgetting to enforce singleton behavior here can lead to an unwanted partitioning of what should be a global object space
- **These warning signs may argue for a Singleton, or for a pool of objects (see Flyweight in a later chapter), but they clearly indicate the need to control object creation to some degree.**

The Singleton Pattern

- **The solution is fairly simple, and might be seen as a degenerate case of the Factory Method:**
  - **Hide the constructor** by making it **private**
  - Define a **static field** that is the single instance
  - Create an apparent **factory method** that simply returns the static instance
- **Singleton is easy enough, but one common confusion is worth considering, and that is between Singleton and a simple class utility.**
  - A class utility is a class with only static fields and methods.
  - It is not technically a singleton, because there are zero instances, rather than one.
- **Use Singleton when an object, not a class, is required:**
  - When the object's type might vary by **subclassing** – polymorphism can apply from one run of an application to the next, or one deployment of a component to the next
  - When you might want to **defer object creation** until an instance is requested by a client
  - When later versions of a design might call for **multiple instances** rather than one – the factory method is essential in this case, as it provides exactly this flexibility

A Singleton Factory

EXAMPLE

- In **HTTP**, the **SocketFactory** is a natural singleton.
- In fact, most factories are natural Singletons: no need for more than one! and yet we may want to allow
  - Deferred creation
  - External configuration of the ultimate factory type
- See **src/cc/sockets/SocketFactory.java**:

```
public class SocketFactory
{
  public static final String FACTORY_CLASS_PROPERTY
    = "cc.sockets.SocketFactory.type";
```

  - Here's the one and only instance, and a **protected** constructor so as to allow subclasses to be instantiated:

```
  private static SocketFactory instance;
  ...
  protected SocketFactory ()
  {
  }
```

  - The factory method, which by default creates an ordinary **java.net.Socket** for the given host and port:

```
  public Socket createSocket
      (String host, int port)
    throws IOException
  {
    return new Socket (host, port);
  }
```

A Singleton Factory

EXAMPLE

– And the factory-for-the-factory, which assures a single instance, and when creating the instance allows external configuration of a subclass in favor of this class as the type to instantiate:

```
public static SocketFactory getInstance ()
{
  if (instance != null)
    return instance;

  String factoryClass = System.getProperty
    (FACTORY_CLASS_PROPERTY,
      SocketFactory.class.getName ());
  try
  {
    instance = (SocketFactory)
      Class.forName (factoryClass).newInstance();
    return instance;
  }
  catch (ReflectiveOperationException ex)
  {
    LOG.log (Level.SEVERE,
      "Couldn't instantiate ... " + factoryClass,
      ex);
  }

  return null;
}
}
```

- A caller might use the factory thus:

```
Socket socket = SocketFactory.getInstance ()
  .createSocket ("somehost", 8008);
```

– See actual usage in **src/cc/sockets/HttpSocketClient.java**.

A Singleton Factory

EXAMPLE

- An alternate implementation provides a mock object, primarily for unit-testing – see **test/cc/sockets/MocketFactory.java**:

```
public class MocketFactory
  extends SocketFactory
{
  public static Socket mocket;

  @Override
  public Socket createSocket
      (String host, int port)
    throws IOException
  {
    return new Mocket ();
  }

  public static void install ()
  {
    System.setProperty (FACTORY_CLASS_PROPERTY,
      MocketFactory.class.getName ());
  }
}
```

- So, a unit test can work with clients classes that use **SocketFactory**, such as by the usage shown on the previous page, and be sure that they will wind up using its configured mocks – simply by calling **MocketFactory.install** in the test set-up.
  - See **test/cc/sockets/HttpSocketClientTest.java**.

A Class Utility

EXAMPLE

- By contrast, the persistence class in **UserDB_Step1** is a class utility, not a singleton. See **src/cc/user/Persistence.java**:

```
public class Persistence
{
  private static final String FILENAME =
    "Realm.ser";
  private static final Logger LOG =
    Logger.getLogger
      (Persistence.class.getName ());

  private static Group root;
  ...
  public static void saveUserDB ()
  {
    ...
  }

  public static Group loadUserDB ()
  {
    ...
  }
}
```

Other Singleton Examples

- A couple of anti-examples – these are actually class utilities:
  - JDBC **DriverManager**
  - The **java.util.Collections** utility

Design Exercise: A Pattern Hunt!

Suggested time: 30 minutes

In this exercise you will go hunting for patterns. Later we will undertake some more traditional design tasks, but right now we're most interested in pattern recognition. Load the Java Core API documentation (this should already be installed on your machine), and find as many Factory and Singleton expressions as you can. For each, identify the package and/or class(es), and which pattern is in play (Abstract Factory, Factory Method, Singleton, or a combination).

Train Schedule

LAB 2

Suggested time: 45-60 minutes

In this lab you will refactor an existing application, using factory patterns to control object creation and to discourage incorrect mixing of objects from different implementation families.

Detailed instructions are found at the end of the chapter.

Reading a Property

EXAMPLE

- In **Trains_Step5** there is a final enhancement to the HTML-producer application from the lab.
  - See **src/cc/html/Factory.java** – **createFactory** now reads a system property; attempts to load a class by the given name; and calls that class' **register** method.

```
public static Factory createFactory ()
{
  try
  {
    String implClassName = System.getProperty
      ("cc.html.Factory.implClass",
        "cc.html.table.TableFactory");
    Class implClass =
      Class.forName (implClassName);
    Class[] params = {};
    java.lang.reflect.Method method =
      implClass.getMethod ("register", params);
    Object[] args = {};
    method.invoke (null, args);
  }
  catch (Exception ex)
  {
    cc.html.table.TableFactory.register ();
  }

  return FACTORY;
}
```

  - There is still a dependency in that we identify a default concrete-factory class – but it is no longer a compile-time dependency.
  - We could remove it, but the resulting application would handle configuration failures less gracefully.

Reading a Property

EXAMPLE

- Test this version as follows:
  - From the command line, run the **ProducePage** application while setting the necessary system property:

```
java –classpath build -Dcc.html.Factory.implClass=
  cc.html.list.ListFactory cc.client.ProducePage
```

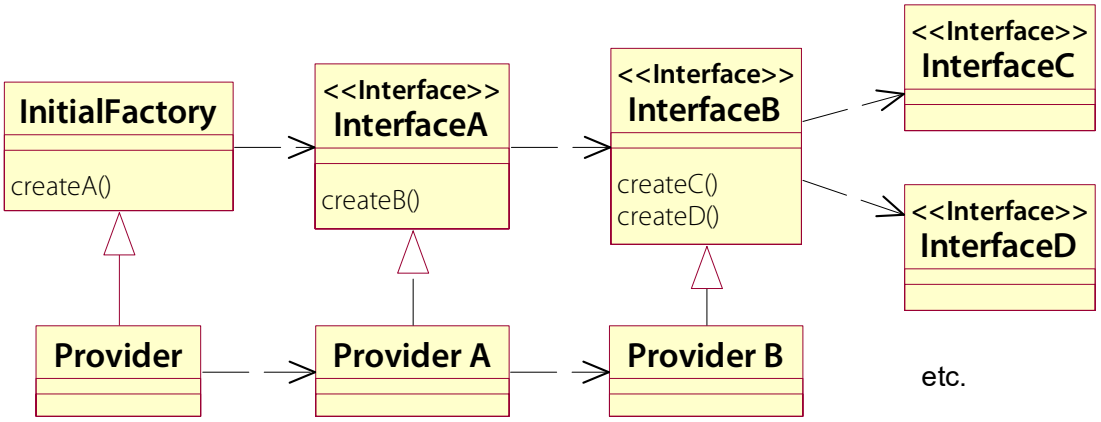
  - From Eclipse, define the property in the "VM Arguments" section of the Run configuration for this class:???

```
-Dcc.html.Factory.implClass=
  cc.html.list.ListFactory
```

- You'll see the list view.
- Try it with the **TableFactory** class as the value of the property, and see the table view as a result.

Cascading Factories

- **Another variation on factory patterns applies nicely when the family of target objects falls naturally into a hierarchy or chain of objects, with a natural "top" or "head" object.**
- **In this case factory methods might be strewn about the interface model, with each interface offering a factory method for the interface(s) below it:**



- **This has the significant advantage that the client code makes just one choice of factory and from that point forward all object-creation control is hidden under the interface model.**
  - Thus compared to a textbook Abstract Factory implementation, it is cleaner to implement.
  - No special privileges, like the one held by Abstract Factory in addressing the Concrete Factory types, is required.
  - Family coherence is nevertheless enforced.

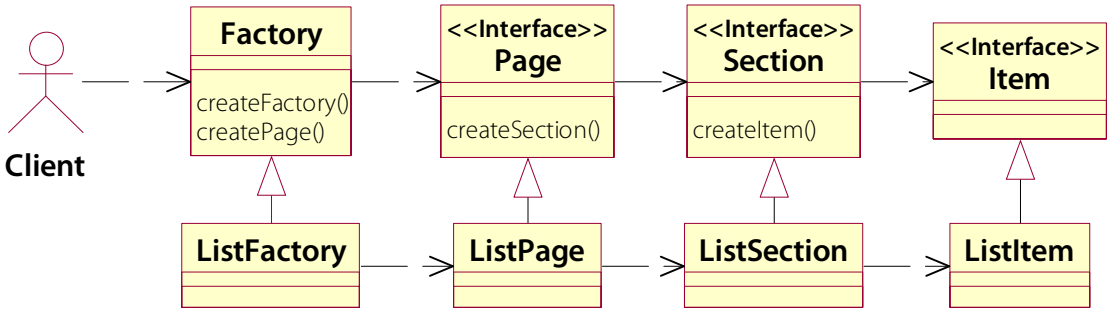
Cascading Factories

- **Cascading factory methods remove the need for extra-strong enforcement of the use of factories.**
- **This approach is increasingly favored in Java APIs:**
  - A **JDBC Connection** creates a **Statement** and that object produces **ResultSet**s
  - An **SAAJ SOAPEnvelope** can create a **SOAPHeader**, which can create a **SOAPHeaderElement**, etc. (Though later versions of SAAJ allow either this approach or a master **SOAPFactory**, the whole API also follows a provider pattern.)
- **A related pattern is Builder, which defines something like an Abstract Factory, but with an interesting algorithm for creating a complex structure of objects, and not just objects drawn willy-nilly from a family.**

Cascading Factories

EXAMPLE

- In **Trains_Cascade**, a redesigned application allows anyone to choose their page provider, but from there the creation of sections and items is under the control of that chosen implementation.



  - Thus incorrect mixing of presentation styles is prevented, without the complication of the **register** method and the stronger relationship between abstract and concrete factories.
  - Client code now looks like **src/cc/client/ProduceList.java** – letting the abstract factory choose the default implementation:

```
Factory factory = cc.html.Factory.createFactory ();
Page page = factory.createPage ();
Section body = page.createSection ();
Item item = body.createItem ();
```

  - … or like **ProduceTable**, choosing one's own starter factory:

```
Factory factory = new cc.html.table.TableFactory();
Page page = factory.createPage ();
Section body = page.createSection ();
Item item = body.createItem ();
```

Spring and Other Dependency-Injection Tools

- **In the context of creational patterns, the Spring framework deserves a mention.**
- **Spring is a broad and ambitious framework for development in Java EE; but its first claim to fame is its "lightweight container."**
  - The Spring container acts as a **super-factory**, able to instantiate any Java class as directed by **external configuration**.
  - That configuration is traditionally in the form of an **XML file**, but increasingly is done through **annotations** in Java.
  - The container can also configure the "beans" it creates, and control their **scopes** – the default scope, in fact, being **singleton**.
  - The container can also **resolve dependencies** between configured beans, again based on directives in the configuration.
  - This is a process known as **dependency injection**, and it offers even more power to applications wishing to configure their components flexibly – and to take advantage of other, non-creational patterns that involve runtime type choices.
- **The framework also makes use of creational and other patterns, throughout its own design.**
- **Other dependency-injection systems offer similar advantages – including some that are standard to Java EE:**
  - Component-to-component injection via annotations such as **@EJB** and **@WebServiceRef**.
  - The **Contexts and Dependency Injection** standard, or **CDI**, which extends dependency injection capabilities to any Java class.

SUMMARY

- **Factory behavior in particular is difficult to pin down in discrete patterns.**
  - All patterns are somewhat vague, by their nature.
  - Factory patterns are unusually so, and there is great hybridization of the known patterns.
  - The key feature of any factory is **control of object creation**. This can be effected by a wide variety of strategies, as we've seen.
- **Singleton, by contrast, is a well-solved problem.**
  - There are variants, as when one class as the factory holds a single instance of the target type, rather than the target type controlling its own instance pool.

Train Schedule

LAB 2

In this lab you will refactor an existing application, using factory patterns to control object creation and to discourage incorrect mixing of objects from different implementation families.

Lab project: **Examples/Trains/Step1**

Answer project(s): **Examples/Trains/Step2** (intermediate)
Examples/Trains/Step3 (intermediate)
Examples/Trains/Step4 (final)

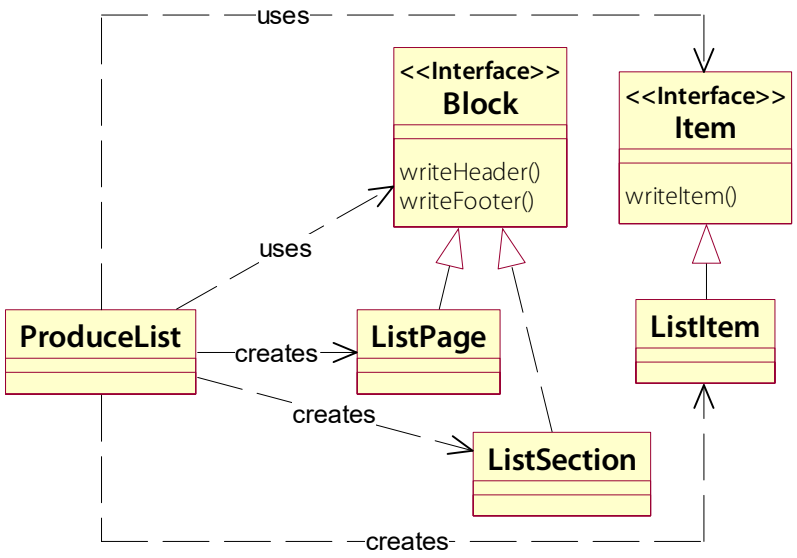
Files: * to be created

src/cc/html/Block.java
src/cc/html/Item.java
src/cc/client/ProduceList.java
src/cc/client/ProduceTable.java
src/cc/client/MakeAMess.java
src/cc/html/Factory.java *
src/cc/html/list/ListFactory.java *
src/cc/html/list/ListItem.java
src/cc/html/list/ListPage.java
src/cc/html/list/ListSection.java
src/cc/html/table/TableFactory.java *
src/cc/html/table/TableItem.java
src/cc/html/table/TablePage.java
src/cc/html/table/TableSection.java
src/cc/client/ProducePage.java

Train Schedule LAB 2

Instructions:

1. Review the starter code. This application produces HTML pages listing trains and their schedules, in at least two possible formats. One format is based on a bulleted list, and the other on a table.



2. Note especially that the client code is directly responsible for object creation – for instance, in **ProduceList.java**:

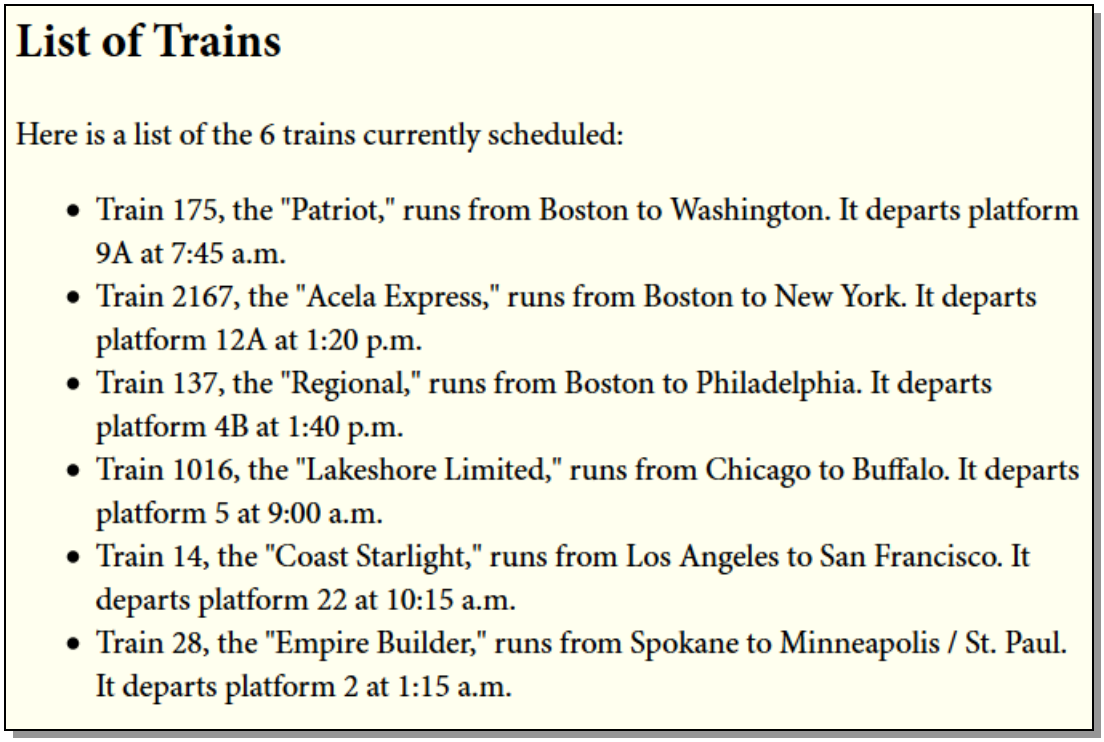
```
Block pageProducer = new cc.html.list.ListPage ();
Block bodyProducer = new cc.html.list.ListSection ();
Item itemProducer = new cc.html.list.ListItem ();
```

Train Schedule LAB 2

3. Test the starter code, running each of the client applications and observing their output in a browser. Run from the command line as follows ...

```
compile
run ProduceList
```

... and open the newly-created **Page.html** in a browser; or just **cc.client.ProduceList** as a Java application from Eclipse, refresh the project, and open **Page.html** using Open With | Web Browser.



List of Trains

Here is a list of the 6 trains currently scheduled:

- Train 175, the "Patriot," runs from Boston to Washington. It departs platform 9A at 7:45 a.m.
- Train 2167, the "Acela Express," runs from Boston to New York. It departs platform 12A at 1:20 p.m.
- Train 137, the "Regional," runs from Boston to Philadelphia. It departs platform 4B at 1:40 p.m.
- Train 1016, the "Lakeshore Limited," runs from Chicago to Buffalo. It departs platform 5 at 9:00 a.m.
- Train 14, the "Coast Starlight," runs from Los Angeles to San Francisco. It departs platform 22 at 10:15 a.m.
- Train 28, the "Empire Builder," runs from Spokane to Minneapolis / St. Paul. It departs platform 2 at 1:15 a.m.

Train Schedule LAB 2

Then try **cc.client.ProduceTable**:

Here is a table of the 6 trains currently scheduled:

Number	Name	Origin	Destination	Departure	Platform
175	Patriot	Boston	Washington	7:45 a.m.	9A
2167	Acela Express	Boston	New York	1:20 p.m.	12A
137	Regional	Boston	Philadelphia	1:40 p.m.	4B
1016	Lakeshore Limited	Chicago	Buffalo	9:00 a.m.	5
14	Coast Starlight	Los Angeles	San Francisco	10:15 a.m.	22
28	Empire Builder	Spokane	Minneapolis / St. Paul	1:15 a.m.	2

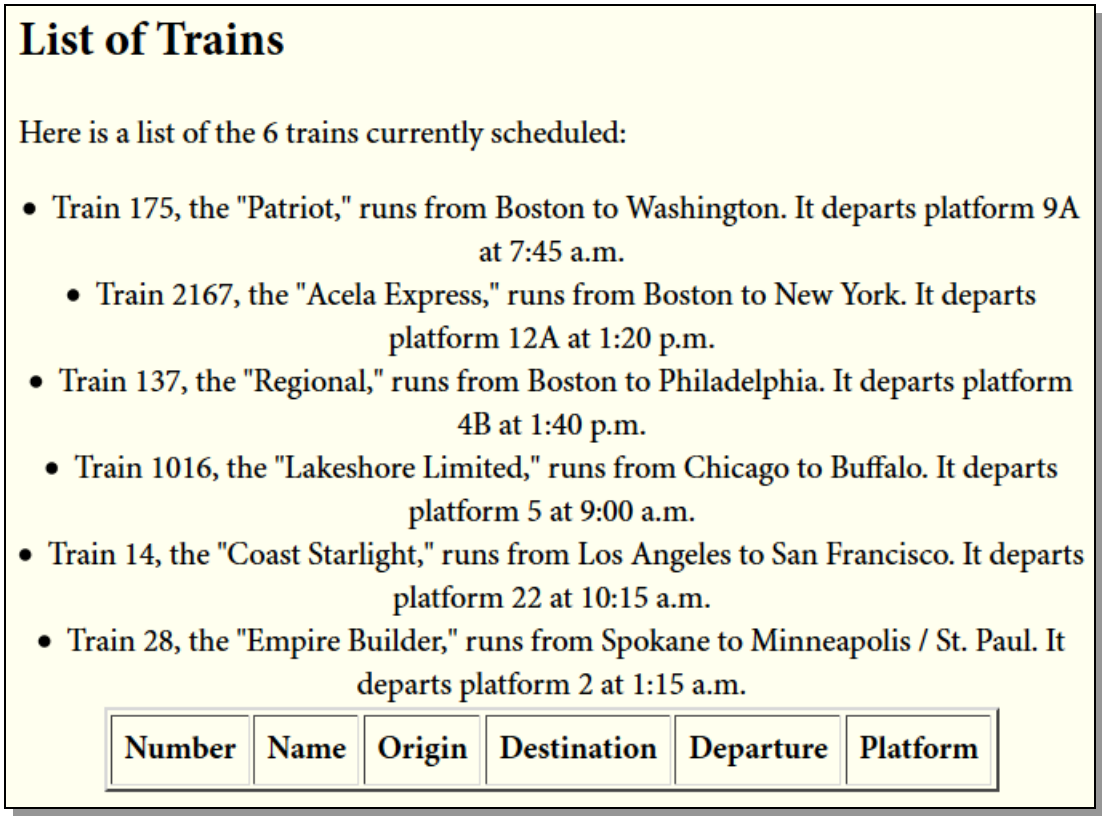
Train Schedule LAB 2

4. Because the client classes can create their own **Block** and **Item** objects, they are in a position to mix and match them arbitrarily. An egregious case of this is **MakeAMess.java**:

```
Block pageProducer = new cc.html.list.ListPage ();
Block bodyProducer = new cc.html.table.TableSection ();
Item itemProducer = new cc.html.list.ListItem ();
```

5. Test this out and find that the browser can't resolve the conflicts between list and table styles:

```
run MakeAMess
```



List of Trains

Here is a list of the 6 trains currently scheduled:

- Train 175, the "Patriot," runs from Boston to Washington. It departs platform 9A at 7:45 a.m.
- Train 2167, the "Acela Express," runs from Boston to New York. It departs platform 12A at 1:20 p.m.
- Train 137, the "Regional," runs from Boston to Philadelphia. It departs platform 4B at 1:40 p.m.
- Train 1016, the "Lakeshore Limited," runs from Chicago to Buffalo. It departs platform 5 at 9:00 a.m.
- Train 14, the "Coast Starlight," runs from Los Angeles to San Francisco. It departs platform 22 at 10:15 a.m.
- Train 28, the "Empire Builder," runs from Spokane to Minneapolis / St. Paul. It departs platform 2 at 1:15 a.m.

Number	Name	Origin	Destination	Departure	Platform

6. Now, this is the mistake of the client programmer – but ideally it shouldn't be possible! You will now implement factories to first discourage this sort of usage, and then to prevent it entirely.

Train Schedule LAB 2

7. Create a new interface **cc.html.Factory**. This will be your Abstract Factory interface; give it one factory method for each of the three types in the family: a **createPageProducer**, a **createSectionProducer**, and a **createItemProducer**. The first two methods should return type **Block**, and the last can return **Item**.
8. Let's focus on the **cc.html.list** package first. Create a concrete factory class **cc.html.list.ListFactory** that implements the abstract factory interface. Each method will return a new instance of the corresponding target type – for example:

```
public Block createPageProducer ()
{
    return new ListPage ();
}
```

9. For each of the target types **ListPage**, **ListSection**, and **ListItem**, remove the **public** modifier from the default constructor. This will force client code to use the factory class.
10. You will now see errors in two of the client classes, because they are trying to use the now package-visible constructors for list producer types. Change these classes to create new **cc.html.list.ListFactory** objects, and to derive their producer objects using the factory methods.
11. Once you clean up the build, you should be able to run all the client code as before, with the original output.
12. Make the same changes for the **cc.html.table** package that you did for **cc.html.list**, and adjust client classes accordingly.

    This is the intermediate answer in **Trains_Step2**. What have we accomplished thus far? There is a better structure to using factories to create producer objects, and that's a step in the right direction. But the factory implementation itself is chosen by the client code – hence we don't have a true abstract factory yet. As a result, we still haven't made a class like **MakeAMess** impossible, because the client can decide to create two different factories and mix objects created by one factory with objects created by the other.
13. Make **cc.html.Factory** an abstract class instead of an **interface**. Add a method **createFactory** that, for the moment, always creates an **cc.html.list.ListFactory**.
14. Create a new client class **cc.client.ProducePage**, using **ProduceList** as a template. Change the code that creates the factory object to rely on **Factory.createFactory** instead of explicitly creating a list factory. This represents the best practice we're trying to encourage in all client classes: let the abstract factory decide what's the best implementation family. Test your new class, and you should get the list presentation.

    This is the intermediate answer in **Step3**.

Train Schedule LAB 2

Optional Steps

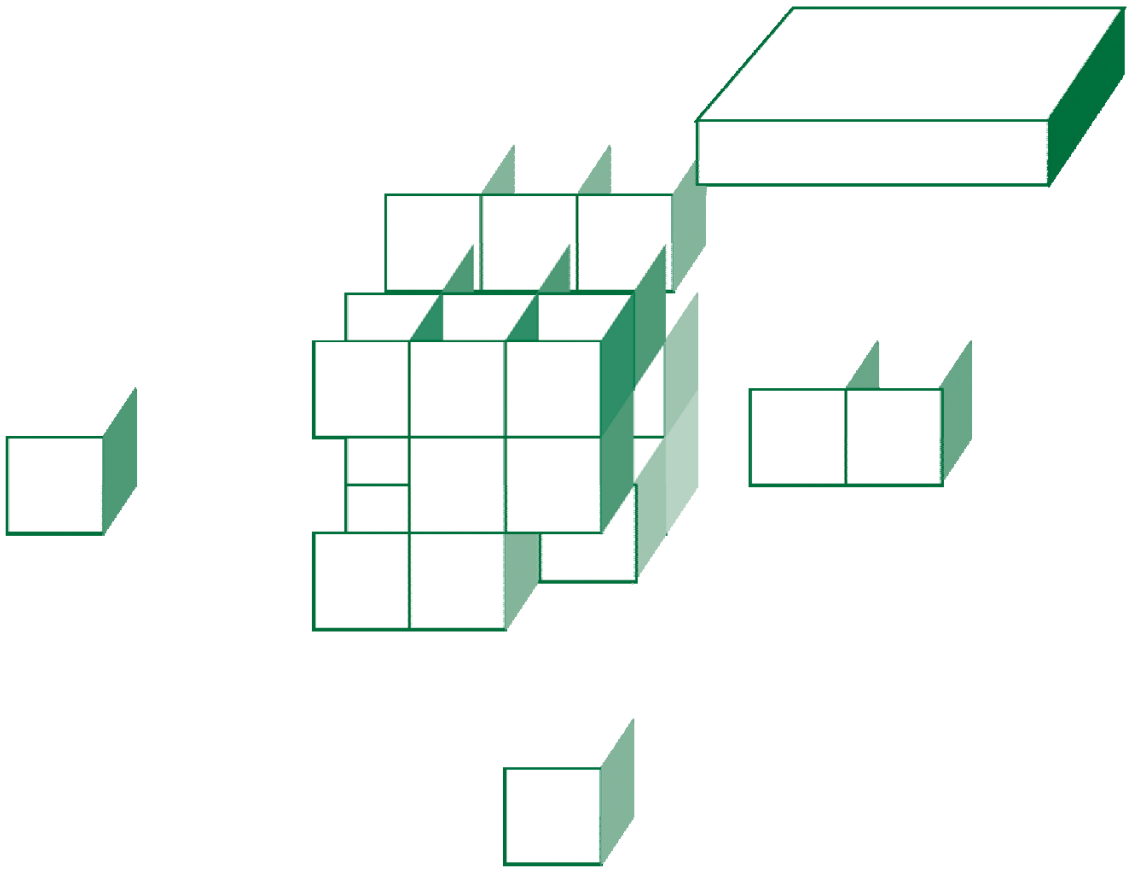
15. Now we're going to get tough on those existing client classes – even the ones that are currently well-behaved. Make the constructors of **ListFactory** and **TableFactory** package-visible – in fact to do this you'll have to add explicit default constructors.
16. Now try a build, and see that **Factory.createFactory** is now the only way to work with the system. **ProduceList**, **ProduceTable** and **MakeAMess** are all broken. In fact, to make them work with the new system would be to make them identical to **ProducePage**. So, just delete all three source files, and we'll work only with **ProducePage** the rest of the way.
17. Ah, but there is a remaining problem: now **Factory** itself won't compile! When we hide the concrete-factory constructors from the client classes, we also hide them from the abstract factory. This is the sort of visibility tangle that can ensnare factory implementations. Some other OO languages have "friend" relationships such that a single class can be given privileged access to another's private members. Java doesn't have that; the closest we could get would be to request this ability as a privileged action from the resident security manager – let's assume that's farther than we're prepared to go! Then, how to establish a privileged relationship and solve this problem?
18. Here's one possible trick: let a concrete factory register itself as the default implementation. On **Factory**, define a **protected** static **FACTORY** field, of type **Factory**.
19. To **ListFactory**, add a static method **register** that sets **FACTORY** to a new instance of **ListFactory**. Do the same thing for **TableFactory**.
20. Change **Factory.createFactory** to (a) call **ListFactory.register** (which unlike the constructor is public), and (b) return **FACTORY**.
21. Test at this point. You should see that everything now compiles cleanly and runs correctly. And what have we gained? The abstract factory is in sole control of the concrete factory choice: it can choose a class to register and then return the results. No other class could do both things; if another object did register a different factory, its choice would just be overridden when it called **createFactory**.

    This is the final answer in **Step4**.

    What's still a bit unsatisfying is that we have to hard-code the factory implementation type in **Factory.createFactory**. There shouldn't be a dependency on any subclass, if at all possible. How can we get dynamic decision-making, without breaking the pattern as we've implemented it so far? If you have time and interest, and are familiar with Java Reflection, you might experiment with an externally-configurable solution. We will review such a solution after the lab.

Chapter 3

Behavioral Patterns



OBJECTIVES

After completing "Behavioral Patterns," you will be able to:

- **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

Un-Tangling Your Code

- **In Chapter 0 we talked about the common problem of complex, tightly-integrated code "hardening" into an indivisible unit, when logically it really is divisible.**
  - Shouldn't we find a way to re-use the common parts of the two methods shown (in pseudo-code) below?

```
void doTheUsualThing1()
{
  MyClass result = ...
  preProcess1(result);
  start(result);
  int x = calculate();
  result.x = refine1(x);
  finish(result);
  postProcess1(result);
}
```

```
void doTheUsualThing2()
{
  MyClass result = ...
  preProcess2(result);
  start(result);
  int x = calculate();
  result.x = refine2(x);
  finish(result);
  // no post-processing
}
```

  - Of course! but at first it seems like a pretty sticky job.

Warning Sign: Letting Subclasses Dictate

- **The easier way to use base classes is to "call up:" that is, to call their methods from subclass methods.**
  - This is often an appropriate design, especially for classes that you might consider to be **utilities** or **frameworks**.
  - It works best when the **overall algorithm** is best encapsulated by the subclass, such that it can use **helper methods** in the base class as it sees fit.
- **This can be a warning sign, though, if used to implement an algorithm that is known at the base-class level.**
- **In our example from the previous page, we'd see it as a warning if the methods highlighted below were superclass-defined helpers:**

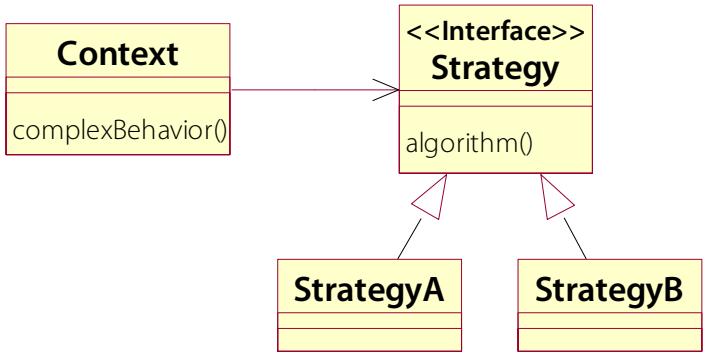
```
void doTheUsualThing1()
{
  MyClass result = ...
  preProcess1(result);
  start(result);
  int x = calculate();
  result.x = refine1(x);
  finish(result);
  postProcess1(result);
}
```

```
void doTheUsualThing2()
{
  MyClass result = ...
  preProcess2(result);
  start(result);
  int x = calculate();
  result.x = refine2(x);
  finish(result);

}
```

- **Why is this bad?**
  - It leaves the **correct execution** of the overall algorithm in the hands of each **subclass**.
  - There is **no single encapsulation** of the protocol that says "thou shalt first call **start**, and then **calculate**, and then proceed to **finish**." Likewise the pre- and post-processors are vulnerable.

The Strategy Pattern

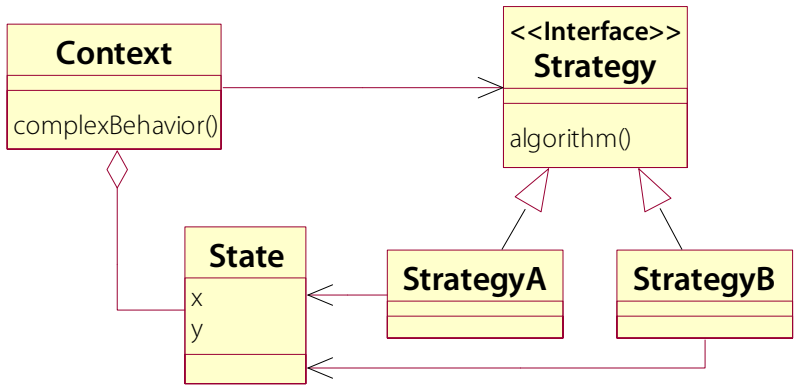
- **The Strategy pattern addresses the problem of varying behavior required by what is initially considered a single encapsulation.**



- **Break a certain behavior out into a separate Strategy when ...**
  - It may **vary independently** of the Context
  - It might be **reusable** in other Contexts (and this is one important tool in "un-tangling" closely-integrated code, as we discussed toward the end of Chapter 0)
- **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

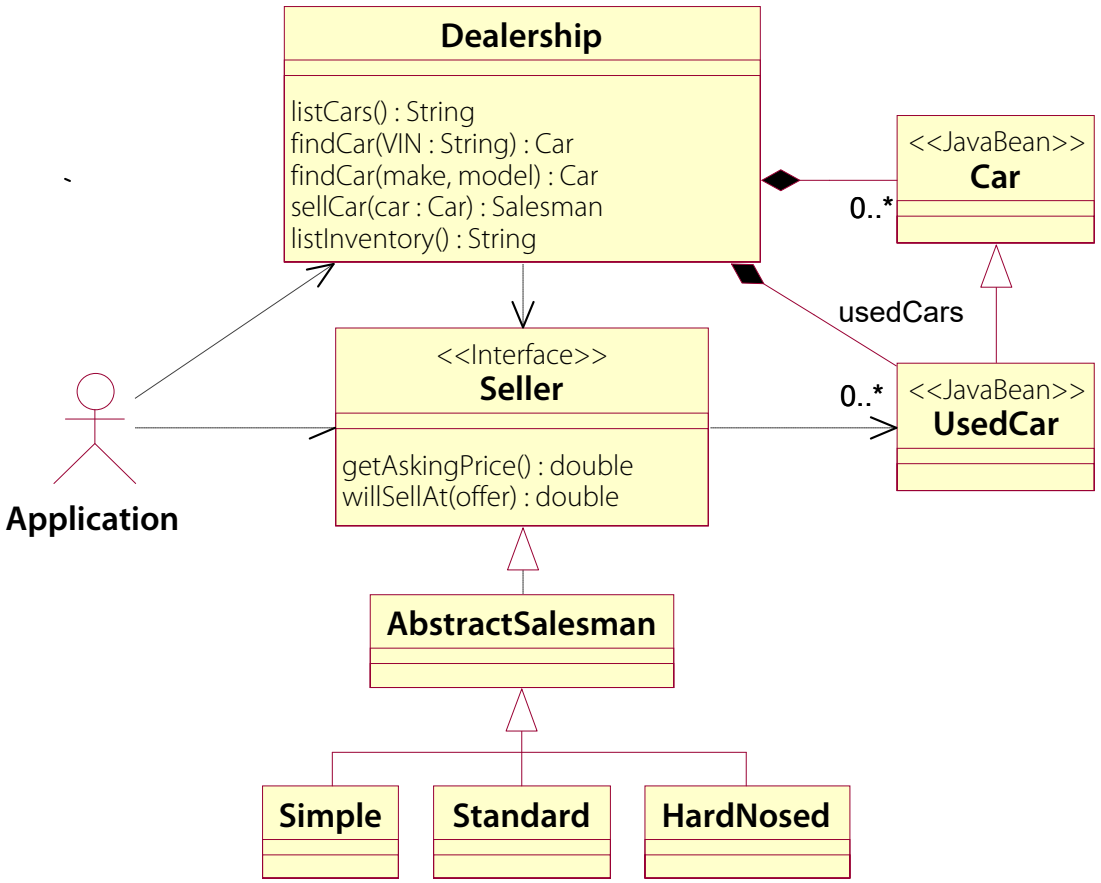
- **Who controls the choice of strategy?**
  - Can a **client** plug in a strategy of its choosing?
  - Does the context object decide for itself? Probably not, as that compromises at least one of the advantages of Strategy, which is adaptability based on swapping in new implementations.
  - Hmm … perhaps a **factory** makes the decision in assembling the system – or, here is a likely role for **dependency injection**.
- **How can the context object share useful state information with a Strategy delegate?**
  - This might require **parameters** to the methods on the Strategy interface.
  - Or the strategy might be **stateful**, and be 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

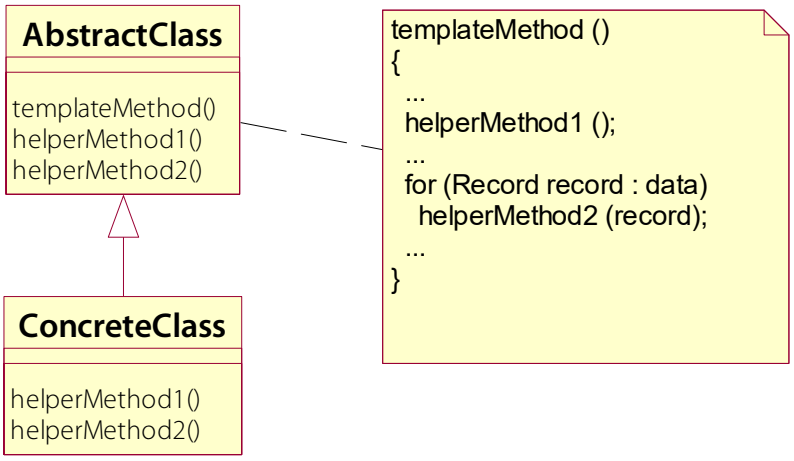
- In **Cars**, the act of selling a car is broken out of the **Dealership** class as a separate strategy.



  - The **Seller** interface defines the semantics of the strategy.
  - Various salesmen take different approaches to the negotiation, and will arrive at different results: sale or no sale, and different prices.
  - The choice of seller is made by a **factory method**, as we discussed in the previous chapter.
  - Note that this is also an example of sharing a state object – the **Car** – with the strategy, by passing it as a parameter at creation time.

The Template Method Pattern

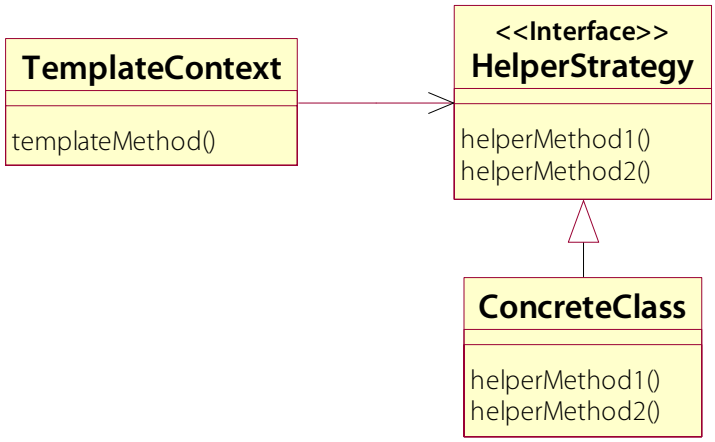
- **Another simple pattern is Template Method, which addresses the need to specialize how some parts of a general and more complex task are carried out.**



- **Define a Template Method when ...**
  - A base class **encapsulates** significant knowledge about a complex business process ...
  - ... but needs to **parameterize** parts of the process in order to allow it to be reused in multiple situations.
- **Some good examples of Template Method:**
  - Implementing **network protocols**, as most protocols require precision in how messages are formatted, when they are sent, timing, error handling, etc. – and yet you will want to send different messages, or process requests variously.
  - The **Servlets** standard defines a template method to handle HTTP, and lets subclasses implement helpers such as **doGet** and **doPost**.
  - **Printing and parsing**: reports, data files, etc. – for similar reasons.

The Template Method Pattern

- Note that any Template Method solution could be refactored as one or more Strategies:



- The choice between the two can be 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.

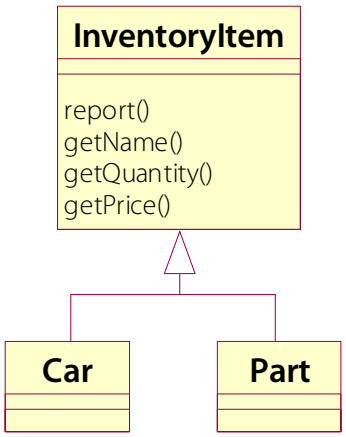
The Template Method Pattern

- **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 **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**.

All Un-Tangled! by Strategy

- Let's consider how we might apply either Strategy or Template method to the hypothetical "tangled" code we discussed earlier.
- A Strategy solution would involve the definition of a new interface – let's call it **Processor**:

```
public interface Processor
{
  public void preProcess (Result result);
  public int refine (int calculation);
  public void preProcess (Result result);
}
```

- Then we could define a single implementation of the general process, and delegate to a provided (or pre-configured) strategy to apply any particulars:

```
public void doTheUsualThing (Processor proc)
{
  MyClass result = ...
  proc.preProcess (result);
  start (result);
  int x = calculate ();
  result.x = proc.refine (x);
  finish (result);
  proc.postProcess (result);
}
```

- We might also define multiple strategies – if perhaps only some of the specific methods were logically related.
  - **preProcess** and **postProcess** clearly belong together.
  - But **refine** might be a separate Strategy altogether, and we might want to allow these to be configured or chosen independently.

All Un-Tangled! by Template Method

- Via Template Method, we'd incorporate the methods defined on the **Processor** interface into the class that defines the template method **doTheUsualThing**.

```
protected abstract void preProcess (Result result);
protected abstract  int refine (int calculation);
protected abstract  void preProcess (Result result);
```

- Then we'd call them on "this" as part of the template method, and it would be up to subclasses to define them:

```
public void doTheUsualThing ()
{
  MyClass result = ...
  preProcess (result);
  start (result);
  int x = calculate ();
  result.x = refine (x);
  finish (result);
  postProcess (result);
}
```

- We might further modify this to share state more easily, by making **result** a protected field on the base class.
- Considering the earlier statement that Template Method varies part of an algorithm, while Strategy varies the whole, we might give the nod to Template Method for this example as stated.
- Or, consider a hybrid solution: depending on the details of the processing being done, perhaps pre/post processing is appropriate for a subclass, while "refining" can be seen as an external strategy.

Health Information Request

LAB 3A

Suggested time: 45 minutes

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.

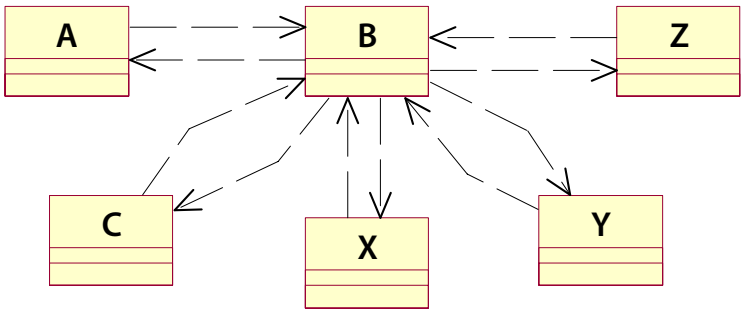
Detailed instructions are found at the end of the chapter.

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, both of which are really a warning signs for this pattern:
  - A could call B repeatedly, in a **polling loop**. This is inefficient and can cost the whole application – and others in the same process, others on the same host – in available processing power.

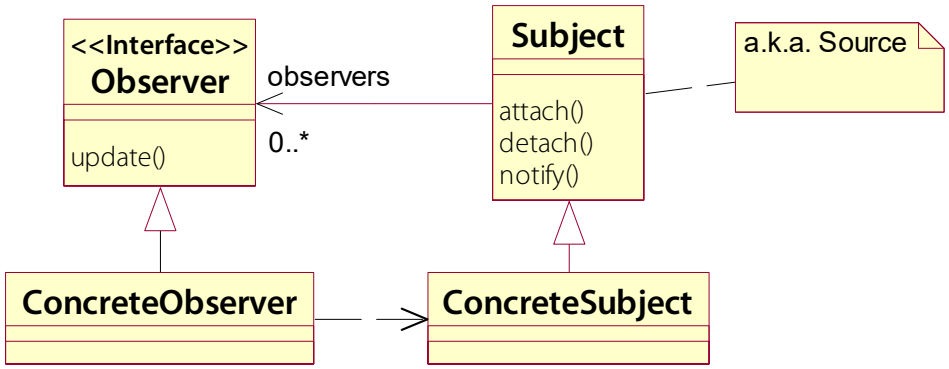
```
while (!download.isComplete ())
  progressBar.setValue (download.pctComplete ());
```

  - 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 Java SE and EE:**
  - JFC and JavaFX **event handling** and **image and resource loading**
  - **User preferences** – one can listen for changes
  - Various **lifecycle listeners** in Java EE standards – for example a servlet can be notified when a new user session is created, or a persistence service can get a call whenever an entity is removed.
- **JFC and JavaFX 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);
```

  - Thus it is a **multicast** model – supporting any number of listeners. You will also encounter **unicast** event models.

The Observer Pattern

- **There are a few important pitfalls to recognize in building a clean Observer implementation. The first 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.
  - But 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> ();
```

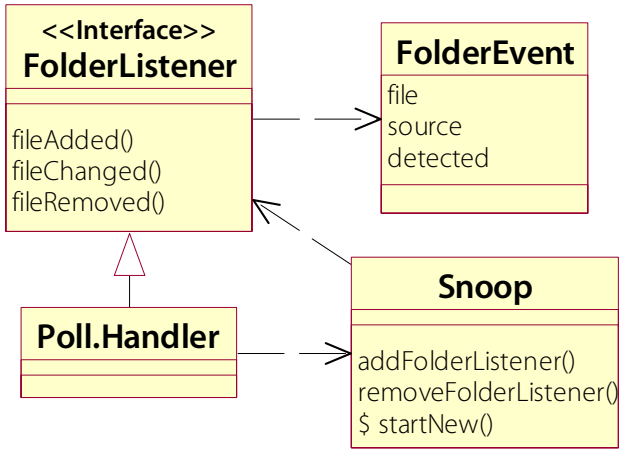
The Observer Pattern

- **Another pitfall: don't expect reliable sequencing, or even serialization of calls to multiple observers!**
  - Chain of Responsibility is a better fit where order is important.
- **Finally, always remember to write the code to remove a listener.**
- **This is one of those mistakes that can elude your early testing, because you'll run a quick main method or JUnit test, see that you're getting events or notifications as you want them, and then shut everything down.**
- **But a longer-running application, or an enterprise component that lives on a running server for a long time, can suffer.**
  - If you add a handler, do some business, and then walk away ... your handler will stick around!
  - Thanks to Java reference counting, even after the rest of your component has been garbage-collected, the registered handler will **outlive its usefulness**, hanging out in memory.
  - And it will **continue to receive calls**. What happens next depends on implementation details, but it won't be good.
  - An eventual **NullPointerException** is the most likely outcome; subtler wastes of resources or spurious program output is probably in second place.
- **Always find an appropriate place in your component's lifecycle in which to clean up what you've created.**
- **It's also a great practice to make proper cleanup part of the testable responsibilities of a component.**

Observing a Folder

EXAMPLE

- In **Polling_Classic**, a component 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.
  - These sorts of conversions between "pull" models (polling on a timer) and "push" models (a/k/a Observers) are common.
  - **Snoop** takes this polling behavior off the client's hands, acting as a Subject for observation.
  - 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.

Observing a Folder

EXAMPLE

- **src/cc/files/Snoop.java** acts as a as a multicast source for **FolderEvent**s, with a **listeners** list and synchronized **add/removeFolderListener** methods.
  - There is one helper method **fireXXXEvent** for each of the three listener methods – for example here is **fireAddEvent**:

```
private void fireAddedEvent (File file)
{
  FolderEvent ev = new FolderEvent
    (file, path.getName (),
      System.currentTimeMillis ());
  for (FolderListener recipient :
      getRecipients ())
    recipient.fileAdded (ev);
}
```

  - The **start** method sets a timer that will call **captureImage** once every second:

```
public void start ()
{
  timer = new Timer ();
  timer.schedule (new TimerTask ()
    {
      public void run  ()
      {
        captureImage (true);
      }
    }, INTERVAL, INTERVAL);
}
```

Observing a Folder

EXAMPLE

– That method compares a previous image with the current one, as retrieved from the file system, and generates events wherever there are discrepancies:

```
private void captureImage (boolean compare)
{
  Map<String,Long> newContents =
    new HashMap<String,Long> ();
  synchronized (contents)
  {
    for (File file : path.listFiles ())
    {
      if (compare)
      {
        if (!contents.containsKey
            (file.getName ()))
          fireAddedEvent (file);
        else if (contents.get (file.getName ()) <
                 file.lastModified ())
          fireChangedEvent (file);
      }

      contents.remove (file.getName ());
      newContents.put
        (file.getName (), file.lastModified ());
    }

    if (compare)
      for (String name : contents.keySet ())
        fireRemovedEvent (new File (name));
  }

  contents = newContents;
}
```

Observing a Folder

EXAMPLE

- In **src/cc/files/Poll.java**, the **main** method creates a **Snoop** for each of the folders identified through command-line arguments:

```
public static void main (String[] args)
{
  ...
  FolderListener handler = new Handler ();
  try
  {
    for (String arg : args)
    {
      System.out.println ("Polling directory " +
        new File (arg).getCanonicalPath () ...);
      Snoop.startNew (arg, handler);
    }
  }
  catch (IOException ex) { ... }
}
```

- The nested class **Handler** implements each listener method, just printing a line to the console for each:

```
private static class Handler
  implements FolderListener
{
  public void fileAdded (FolderEvent ev)
  {
    System.out.println ("File added to " +
      ev.getSource () + ": " +
      ev.getFile ().getName ());
  }

  public void fileChanged (FolderEvent ev) {...}
  public void fileRemoved (FolderEvent ev) {...}
}
```

Functional Interfaces as Observers

- Functional programming, using new features in Java 8, is especially powerful for any sort of callback method.
- Going back to JFC and GUI programming ... consider the **ActionListener**, the most basic of the many event-listener interfaces, which is used for button clicks, hitting "Enter" or "Return" in a text field – any sort of "action" by the user.

```
public interface ActionListener
{
  public void actionPerformed (ActionEvent ev);
}
```

- Though not originally conceived as a functional interface, it nevertheless acts as one, and so a client can register an event handler for a button, for example using a method reference:

```
JButton OK = new JButton ("OK");
OK.addActionListener (this::closeDialog);
```

  - Or, using a lambda expression:

```
someTextField.addActionListener
  (ev -> { otherTextField.setText
    ("prefix" + someTextField.getText ()) } );
```

Breaking Up the Listener

- **Where traditionally we might have defined a multi-method Observer interface for a particular system, it's interesting to consider using a set of single-method Observers, instead.**
  - In fact, it gets easier, if you become familiar with the range of built-in functional interfaces defined as part of the Stream API.
  - Most observer methods are ultimately just **Consumer**s of a given event type:

```
public interface Consumer<T>
{
  public void accept (T toConsume);
  public default Consumer<T> andThen
    (Consumer<? super T> after);
}
```

- **So you could dispense with the traditional listener interface, and just register one or more Consumer<YourEventType>s.**
- **It can mean more housekeeping on the event source.**
- **The win is seen on the other side, where handlers often can be implemented quite easily, inline.**

Capturing a Reference

- One trick to this, though, is that in order to un-register an observer, you'll need a reference to that observer as an object.
- Lambda expressions and method references are most often passed directly to methods that use them – and as such it's impossible to un-register them later.
- But it is possible to capture an object reference to a method reference or lambda expression; in fact it's quite simple.
- So, if you want to add a listener, but then be able to remove it later ...
  - Define the listener and capture a reference to it:

```
Consumer<MyEvent> handler1 = ev -> doSomething ();
Consumer<MyEvent> handler2 = this::foo;
```

  - Register it:

```
eventSource.addMyListener (handler1);
eventSource.addMyListener (handler2);
```

  - Un-register it:

```
eventSource.removeMyListener (handler1);
eventSource.removeMyListener (handler2);
```

Polling with Functional Interfaces

EXAMPLE

- In **Polling_Functional**, the Polling application is refactored to favor functional interfaces.
- The **FolderListener** interface is gone, though the event type **FolderEvent** is still useful.
- In **src/cc/files/Snoop.java**, the event-source implementation moves from a single list of listeners to a map, with the key being an event type and the value being the list of listeners for that type of event:

```
public enum EventType { ADD, CHANGE, REMOVE };

private
  Map<EventType,List<Consumer<FolderEvent>>>
    listeners = new TreeMap<> ();

public synchronized void addListener
  (EventType eventType,
    Consumer<FolderEvent> listener)
{
  listeners.get (eventType).add (listener);
}

public synchronized void removeListener
  (EventType eventType,
    Consumer<FolderEvent> listener)
{
  listeners.get (eventType).remove (listener);
}
```

- So this is a more involved system for managing listeners – but note that it would also scale up to any number of event types, with very little additional code.

Polling with Functional Interfaces

EXAMPLE

- Instead of three strongly-typed helper methods to fire events, we have a single helper that takes the event type as a parameter.
  - This method in turn relies on a helper to **clone** the specific list of listeners, so as to avoid race conditions:

```
private synchronized List<Consumer<FolderEvent>>
  clone (EventType eventType)
{
  return new ArrayList<Consumer<FolderEvent>>
    (listeners.get (eventType));
}

private void fireEvent
  (File file, EventType eventType)
{
  FolderEvent ev = new FolderEvent
    (file, path.getName (),
      System.currentTimeMillis ());
  for (Consumer<FolderEvent> recipient :
      clone (eventType))
    recipient.accept (ev);
}
```

Polling with Functional Interfaces

EXAMPLE

- **captureImage** now uses the modified style of helper method:

```
private void captureImage (boolean compare)
{
  Map<String,Long> newContents =
    new HashMap<String,Long> ();
  synchronized (contents)
  {
    for (File file : path.listFiles ())
    {
      if (compare)
      {
        if (!contents.containsKey
            (file.getName ()))
          fireEvent (file, EventType.ADD);
        else if (contents.get (file.getName ()) <
                 file.lastModified ())
          fireEvent (file, EventType.CHANGE);
      }

      contents.remove (file.getName ());
      newContents.put
        (file.getName (), file.lastModified ());
    }

    if (compare)
      for (String name : contents.keySet ())
        fireEvent (new File (name),
          EventType.REMOVE);
  }

  contents = newContents;
}
```

Polling with Functional Interfaces

EXAMPLE

- **src/cc/files/Poll.java** is greatly reduced, because it is able to use a handful of lambda expressions in lieu of the nested handler class.

```
public class Poll
{
  public static void log
    (FolderEvent ev, String effect)
  {
    System.out.println ("File " + effect + " " +
      ev.getSource () + ": " +
      ev.getFile ().getName ());
  }

  public static void main (String[] args)
  {
    ...
    try
    {
      for (String arg : toProcess)
      {
        System.out.println ("Polling directory " +
          new File (arg).getCanonicalPath () ...);
        Snoop.startNew (arg,
          ev -> log (ev, "added to"),
          ev -> log (ev, "changed in"),
          ev -> log (ev, "removed from"));
      }
    }
    catch (IOException ex) { ... }
  }
}
```

- Also, it's now possible for a client to register listeners selectively: maybe only for one or two of the available event types, if it has no interest in the others.

Primes

LAB 3B

Suggested time: 45-60 minutes

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.

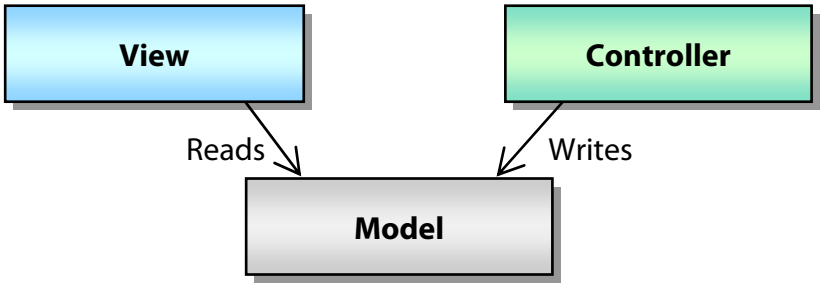
Detailed instructions are found at the end of the chapter.

The Model/View/Controller Pattern

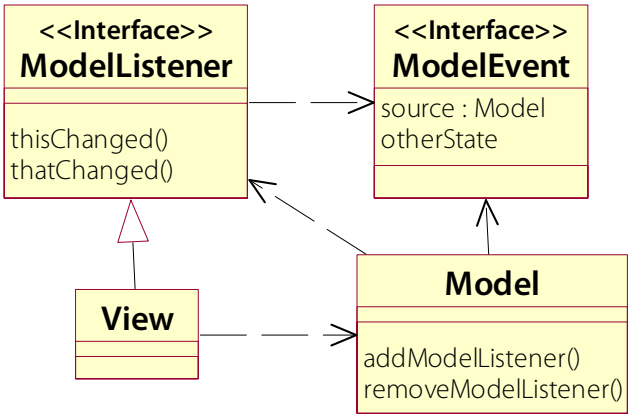
- **The one pattern in this course that is not defined in the GoF text is Model/View/Controller, or MVC.**
- **MVC originated in the Smalltalk platform.**
- **It addresses the problem of organizing an application's state and behavior in a way that**
  - **Centralizes the state** information
  - Defines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

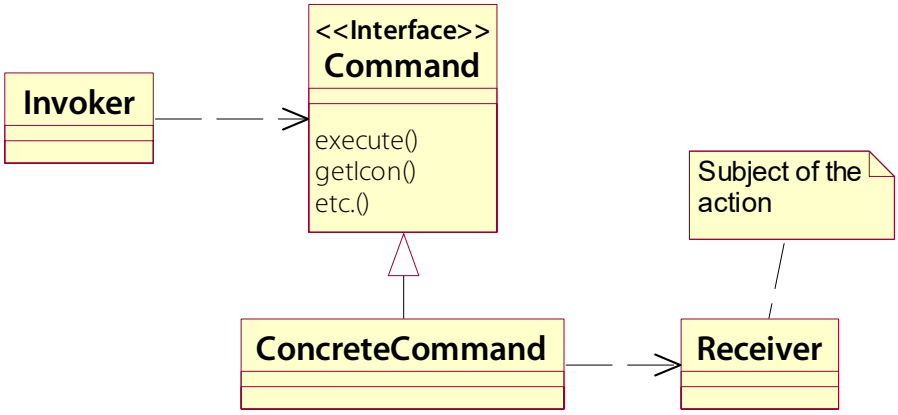


The Command Pattern

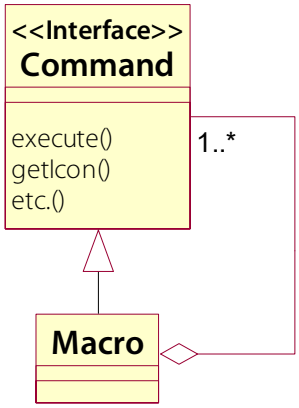
- **The Command pattern encapsulates an action – often a user's action – as a class unto itself.**
- **This is sometimes counter-intuitive: isn't an action supposed to be just one method on a class?**
- **The idea of the Command pattern is to distinguish between an ordinary method call and an action,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

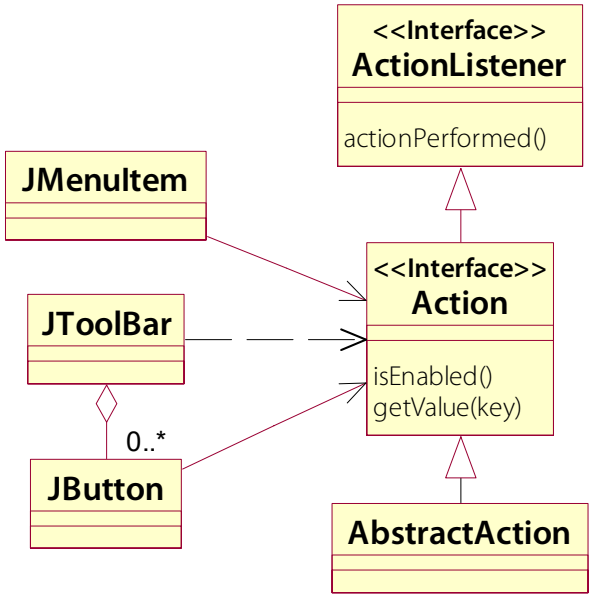


  - The **Command** defines at least a method such as **execute** – which, if you like, is the core method that got a promotion!
  - It will typically define additional standard attributes and methods suitable for its context: get an icon, undo/redo, etc.
  - Then various **ConcreteCommand**s will implement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

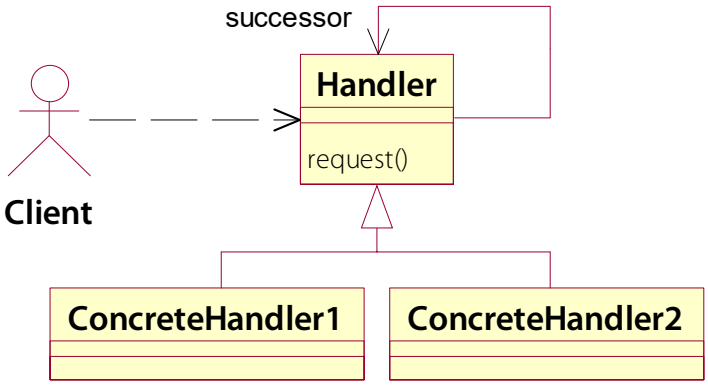
The Command Pattern

- **In the enterprise space, the command pattern can also be mapped to remote requests from clients or peers, and here too it can interact closely with MVC.**
- **Spring MVC captures form input as the current command, and implicitly automates certain handling when a command class is used as a parameter on a request-handling method.**
  - A web application then can be analyzed as the sum of the commands that the user might invoke – each triggering an MVC request-handling cycle.
- **The REST, or representational state transfer, approach to web services emphasizes the encapsulation of commands.**
  - As web applications offer links and submit buttons to the user, a web service can offer hyperlinks to the client, pointing the way forward in the use of the service.
  - There are even emerging standards by which these links can be expressed, using **JavaScript Object Notation** (**JSON**) to encode URL, HTTP method, and other information.

```
"_links":
{
  "self": { "href": "/dept/123" },
  "manager": { "href": "/dept/123/manager" },
  "employees":
  {
    "href": "/dept/123/emps/{ord}",
    "templated": true
  }
}
```

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.

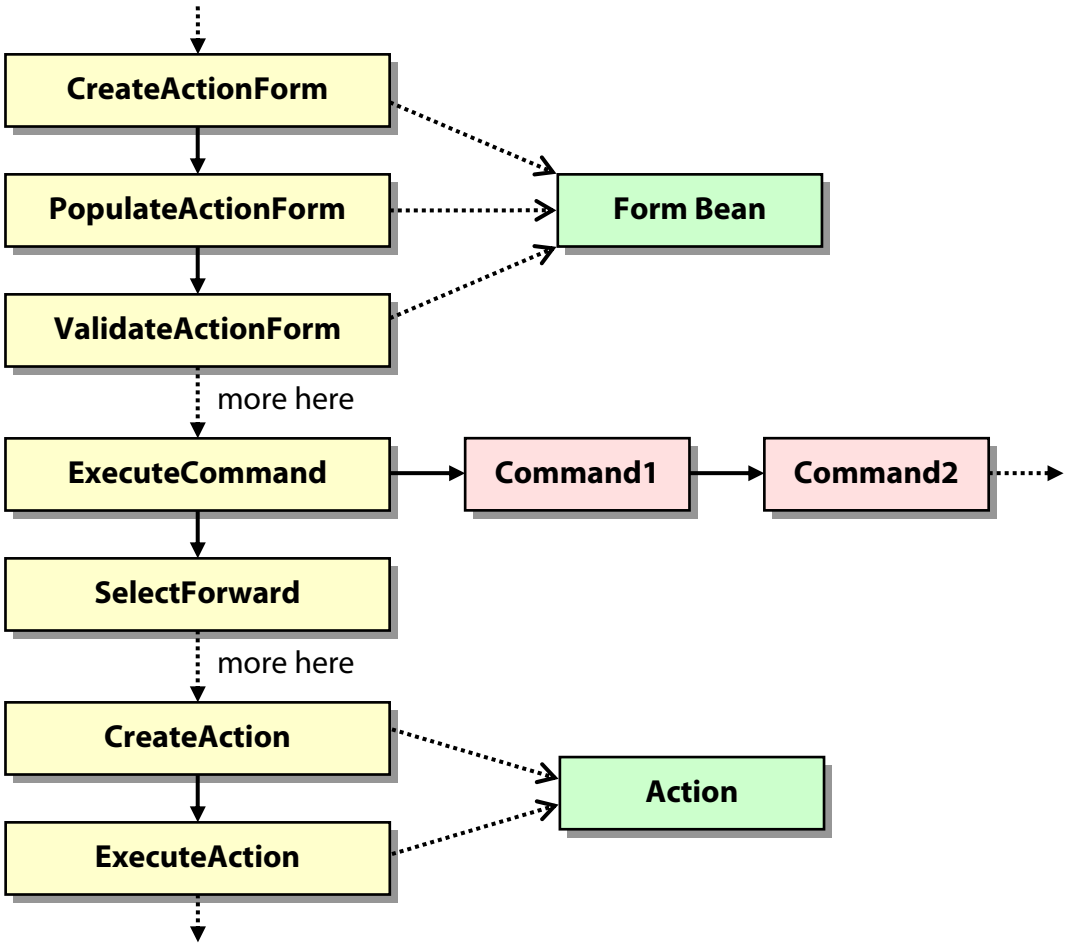


The Chain of Responsibility Pattern

- **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 Java EE as the Intercepting Filter pattern.

The Chain of Responsibility Pattern

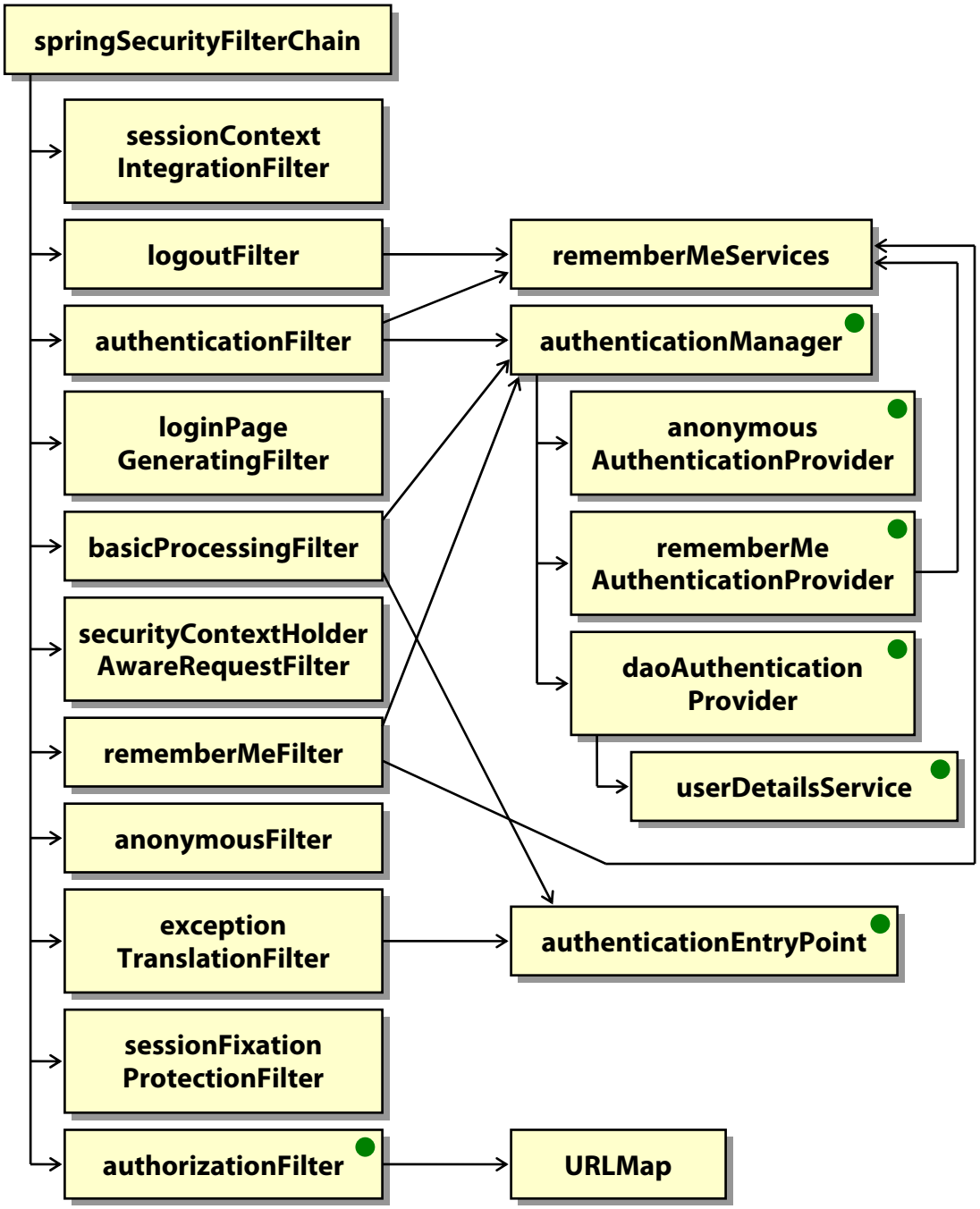
- Struts 1.3+ breaks its request-handling infrastructure into a highly pluggable/overridable chain of responsibility system known as the **ComposableRequestProcessor**:



  - Default processing is factored into a chain of objects, each of which gets a chance to see the request and carry out some part of the total framework behavior.
  - Inserting new handlers or removing default ones is straightforward.

The Chain of Responsibility Pattern

- Spring Security operates fundamentally as a filter chain, intercepting incoming HTTP requests and running them through various sorts of processing as configured.



Behavioral Refactoring

LAB 3C

Suggested time: 30-45 minutes

In this exercise you will analyze a "before picture" 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 found 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.

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

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.

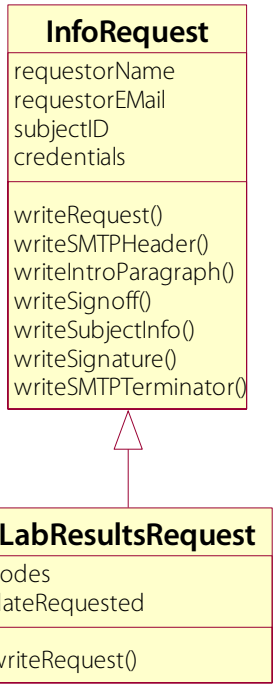
Lab project: **Health_Step1**

Answer project(s): **Health_Step2**

Files: * to be created
src/cc/health/InfoRequest.java
src/cc/health/LabResultsRequest.java

Instructions:

1. 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);
    ...
```

Health Information Request LAB 3A

2. Run the **buildKeyStore** script to put a keystore in place that will support the digital signature operations; you only need to do this once for the lab exercise. (If working in Eclipse, you can just double-click **buildKeyStore.bat**.)

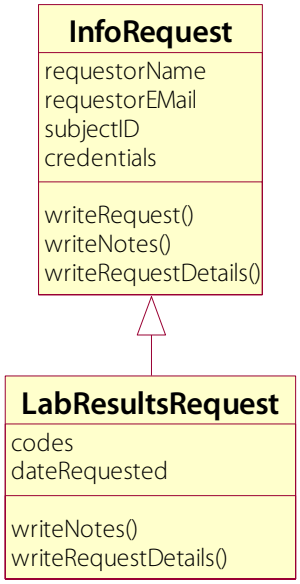
   You'll see a new file **KeyStore**.

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

```
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
302c214418dfcc4d666afb5914da23e6c8cad93684a72f921440a37954ae87686bcccde
9ce33f3533a6249dad
END_SIGNATURE

.
```

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.

Health Information Request LAB 3A



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**.

Health Information Request LAB 3A

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 retest, and see the same output.

    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?

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.

Lab project: **Examples/Primes/Step1**

Answer project(s): **Examples/Primes/Step2** (intermediate)
Examples/Primes/Step3 (intermediate)
Examples/Primes/Step4 (final)

Files: * to be created
src/cc/primes/FindPrimes.java
src/cc/primes/CLI.java
src/cc/primes/PrimesEvent.java *
src/cc/primes/PrimesListener.java *
src/cc/primes/StressTest.txt which becomes **StressTest.java**
src/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
    (args.length == 0 ? 1000 : Integer.parseInt (args[0]));
worker.find ();
for (long prime : worker.getPrimes ())
    System.out.print (" " + prime);
```

2. Give it a try:

```
run CLI
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191
193 197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283
293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397 401
409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 503 509
521 523 541 547 557 563 569 571 577 587 593 599 601 607 613 617 619 631
641 643 647 653 659 661 673 677 683 691 701 709 719 727 733 739 743 751
757 761 769 773 787 797 809 811 821 823 827 829 839 853 857 859 863 877
881 883 887 907 911 ...
```

Primes LAB 3B

3. Try it again with larger ceilings – pass 10000 or 100000 as a program argument – 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. But this is a classic example of a short-term solution that tends to require refactoring later: what if we want to pipe to a file or a GUI instead?

   (If working in Eclipse, you'll also notice a quirk at this point. At first it will seem that you have no console output, and you may be tempted to start the debugger to see what's going on. Just right-click the Console view, choose Properties, and make it a fixed-width console. All the output will appear! However, you may also find that you can't leave the console set this way, or you'll get spurious exceptions from the IDE as you try to test. If that happens, either switch back and forth as you go, or just plan to run your tests from the command line for this lab. Switching the **print** call to **println** should do the trick, too, but of course then you'll get a very long vertical list of text.)

   Anyway, the stronger solution is to make the **FindPrimes** component observable. Your listener will have a single method, so you can use **Consumer**, as the second Polling example does.

   What's more, in this case even your event type is unnecessary. Polling wants to communicate three separate values and so encapsulates them in an event bean. You'll only need to communicate the prime number, which is just a long.

4. Open **FindPrimes.java**, and add a private field **listeners** to the class: this is a **List** of **Consumer<Long>** references, and should be initialized to an empty **LinkedList**.
5. Add methods to register and remove listeners: **addPrimesListener** just adds the given **Consumer<Long>** to **listeners**, and **removePrimesListener** removes from the list.
6. Now, create a protected helper method called **firePrimesEvent**: void return type and taking a **long** as its parameter. This method will loop through **listeners**, calling **accept** on each and passing the given prime number.
7. Near the bottom of the **find** method, you'll see a call to **primes.add** a new number. Right after that (still inside the **Candidate** loop), call your new helper method to fire the event based on the number, **candidate**.
8. 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 remove the loop at the end of the **main** method that prints out all the primes.
9. Add a call to **worker.addPrimesListener**, just before calling **worker.find**. For a listener, pass a lambda expression that prints a space and the number to the console, just as you were doing before.

Primes LAB 3B

10. 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 **Primes_Step2**.

11. Have you noticed something is wrong with the code as written so far? Give this a bit of thought before proceeding.

12. 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 **PrimesEvent**s.

13. 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 ();
```

14. Build and run the application again. 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:

```
run CLI
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$ListItr.checkForComodification
  at java.util.LinkedList$ListItr.next
  at cc.primes.FindPrimes.firePrimesEvent
  at cc.primes.FindPrimes.find
  at cc.primes.CLI.main
```

15. This is the classic Observer pitfall of a Subject that is not thread-safe in how it handles its observers. What do you need to do to fix this?

Primes LAB 3B

16. Open **FindPrimes.java** and make two 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**).

17. Test again, and see that – even with a higher ceiling – you won't encounter race conditions or get the fail-fast **ConcurrentModificationException** as before.

    This is the intermediate answer in **Primes_Step3**.

18. 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. This class listens for primes events, as **CLI** does, and reacts by filling the text field and by writing to the console.

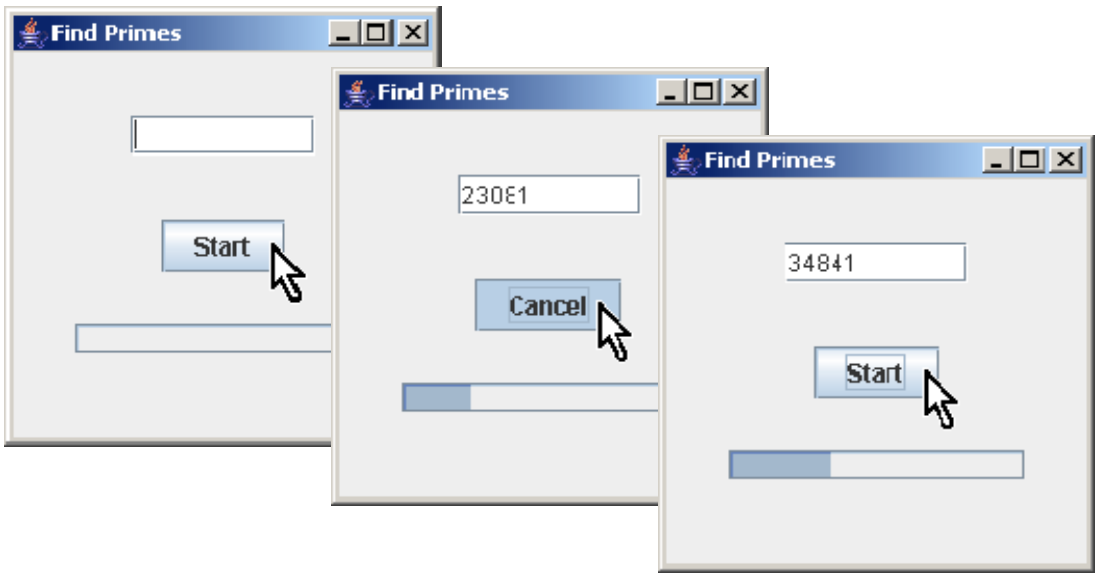
```
run GUI
```

Also notice, if you look at the source code, that **GUI**, while still using lambda expressions to implement event handling, captures references to the resulting objects in fields **updateHandler** and **consoleHandler**. This makes it possible to remove these listeners when the window is closed. This is an important bit of cleanup, even if in your testing it won't matter, because the lifecycle of the **GUI** class is aligned with the lifecycle of the JVM. Consider what might happen if this class were used in a larger application, and could be started and stopped, or created, closed, and re-created.

Primes LAB 3B

Optional Steps

19. Create a new class **ProgressBar** which extends **javax.swing.JProgressBar**.
20. Write a constructor that takes a **FindPrimes** object as a parameter called **worker**. 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.
21. Now give the class a field of type **Consumer<Long>**, and initialize it to a lambda expression that calls **setValue**, passing the results of a call to **intValue** on the given prime number. This will trigger an update to the visual presentation of the progress bar.
22. In the constructor, add this object as an even handler by passing it to **worker.addPrimesListener**.
23. Now, remember that clean-up pitfall ... just as **GUI**is tidying things up by removing its two listeners, we should be doing the same – or at least putting **GUI** in a position to do so. Add a public method **close** to the class, and have it un-register your listener. (You could even make your class **AutoCloseable**, if you like, although in GUI programming the ability to use try-with-resources doesn't come up as often.)
24. Open **GUI.java** and add a field **progress** that is of type **ProgressBar**.
25. In the constructor, change the size of the **GridLayout** from two rows to three.
26. A bit further down, after the code that builds the grid row based on **pnStart**, initialize **progress** to a new **ProgressBar**, passing **worker**. Then add code to place the **progress** control as the third row – use the four preceding 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 final answer in **Primes_Step4**.



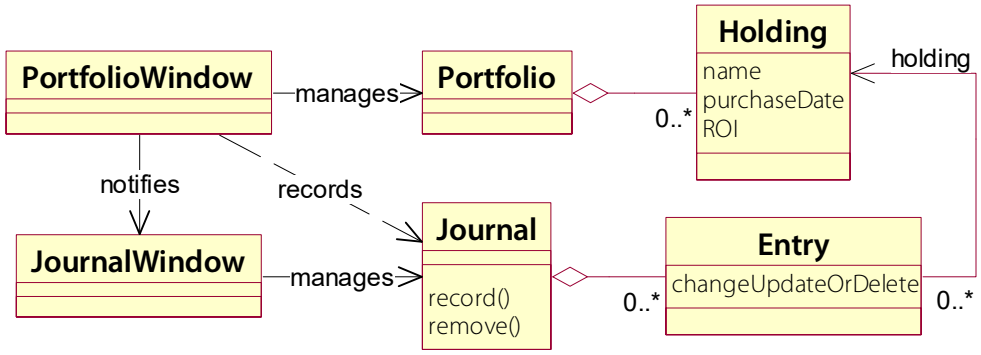
Behavioral Refactoring

LAB 3C

In this exercise you will analyze a "before picture" 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.

Current Design

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 **record**s 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.

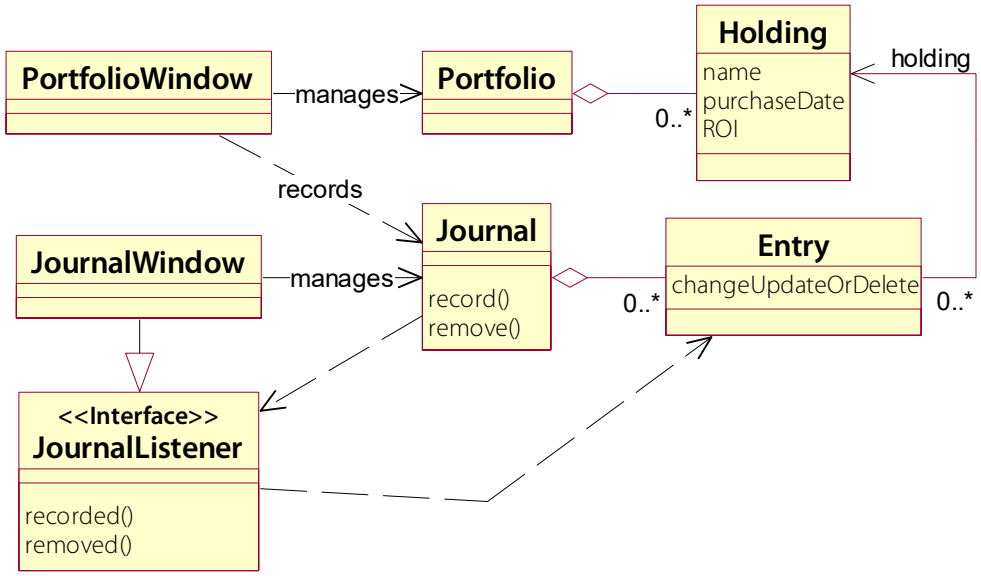
Behavioral Refactoring LAB 3C

Analysis and Improved Design

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."

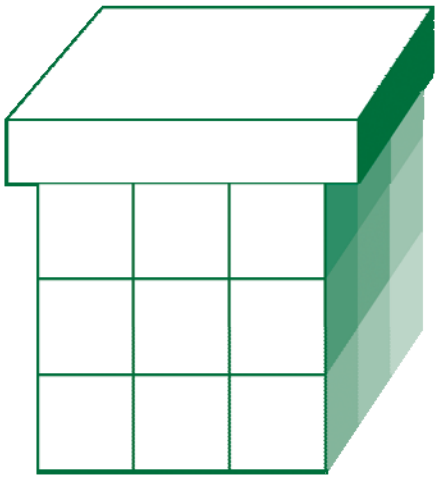
What we don't like: if that's a model, where are the views and controllers? In the use case described, the **PortfolioWindow** acts as a controller on both the **Portfolio** and the **Journal**; **JournalWindow** is both a controller and view on the **Journal**. So far, so good, but why is a controller 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**.



Chapter 4

Structural Patterns



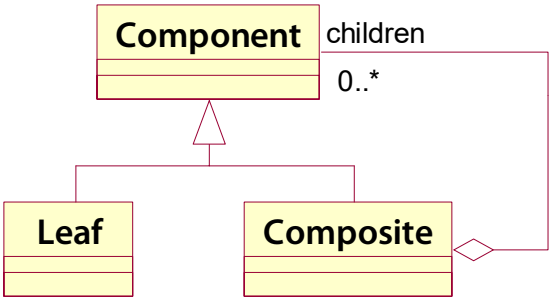
OBJECTIVES

After completing "Structural Patterns," you will be able to:

- **After completing this unit you will be able to recognize and apply the following patterns in designing Java software:**
  - Composite
  - Adapter
  - Decorator
  - Façade
  - Flyweight
- **Gang-of-four structural patterns not explicitly covered in this course are:**
  - Bridge
  - Proxy

The Composite Pattern

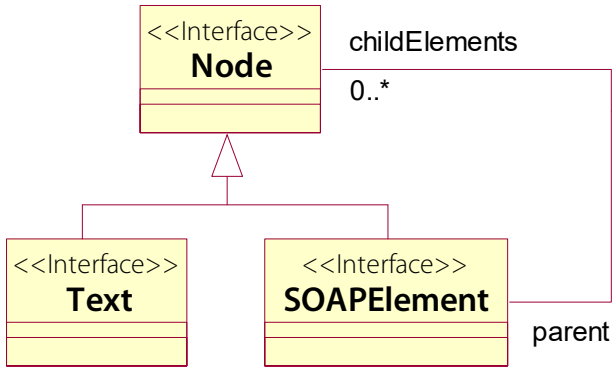
- **The Composite pattern defines a means of creating composite objects or parent-child hierarchies of arbitrary depth.**



  - **Component**, a rather GUI-centric term, might be considered more generally as **Node**.
  - **Composite** is simply a node with children that are of the node type: that is it both extends and collects the node type.
  - **Leaf** is a node with no children. In some expressions of this pattern, **Leaf** is not needed, as there is nothing "special" about being childless that would require a subclass.
- **We observed some Composite warning signs and pitfalls in our exercise designing users and groups:**
  - **Non-polymorphic compositions** that, for example, treat a **Group**'s collection of **User**s differently from its collection of other **Group**s.
  - **Two-class compositions** where the leaf or the composite is the base of the system, and the other class "wastes" inherited features from the base class – for instance **Group** extending **User** and having no use for **password**.

The Composite Pattern

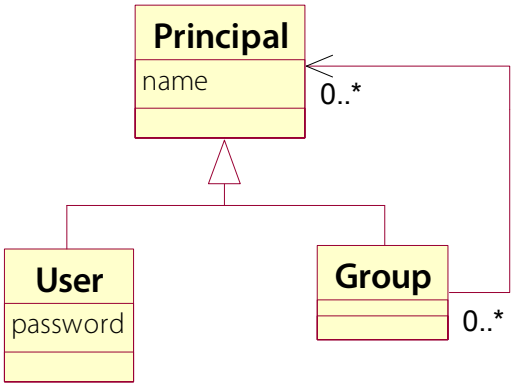
- **Examples of Composite include:**
  - **File/folder systems** – here the parent is associated with the child, but not really composed of its children
  - **JFC and JavaFX window hierarchies** – here we have a clearer concept of true composition – a part/whole relationship
- **XML offers some interesting comparisons:**
  - The DOM has **Node** and **Element** types, which given the nature of XML would be good candidates for component and composite roles. But **Node** offers access to children! even though many node types can't have them.
  - Note that the DOM is a language-neutral API defined by the W3C and mapped strictly into Java for the JAXP.
  - By contrast, the more recent (and Java-specific) SAAJ defines a **Node** as a proper component type, **SOAPElement** as the composite, and one leaf types **Text**.



Users and Groups

DEMO

- The user/group composition we discussed earlier is reduced to practice in **UserDB_Step1**.



- There are three classes expressing the design shown above – all by plain-vanilla JavaBeans standards:
  - **Principal** defines the one shared property, **name**.
  - **User** extends this with its specific **password** property (again, there's more to this design, surely, but we're keeping it simple). It doesn't depend on **Group**.
  - **Group** extends **Principal** and implements the composition, by also keeping a collection of **Principal**s – which of course could be **User** instances or other **Group**s
- Two classes exercise this composite domain model
  - **Persistence** creates a small in-memory user database, or loads from a file if available, and saves to the file when prompted.
  - **Load** offers a **main** method that calls **Persistence.loadUserDB** and then its own recursive **print** method to write the whole database to the console.

Users and Groups

DEMO

- In **src/cc/user/Load.java**, notice that the **print** method must use metadata and downcasting to effect a recursive algorithm:

```
public static void print
  (Principal subject, String indent)
{
  System.out.println
    (indent + subject.getName ());
  if (subject instanceof Group)
    for (Principal member :
        ((Group) subject).getMembers ())
      print (member, indent + "  ");
}
```

  - This may seem clunky or downright incorrect, but it's necessary in the full expression of Composite.
  - Some implementations avoid this, but the tradeoff is essentially a degeneration of the pattern: collapsing the component and composite types into one.
  - The alternatives in this case would be to resort to one of the solutions we dismissed in Chapter 1: make **Group** collect **User** and other **Groups** separately, or to make **User** a subtype of **Group** – a group with no members.
  - There is a bit of a phobia about **instanceof** in some circles, and, no doubt, one can rely too heavily on type information.
  - But it's in the language for a reason! and it's better to recognize a real need to tell two types apart than to pretend that they're one type and then have to call a method such as **isGroup** or **hasMembers** as a poor substitute for **instanceof**.

Users and Groups

DEMO

- **Let's add some more client functionality to the project, as a way of pushing on the design.**
- **First, let's say that we want to be able to walk the tree, looking for a principal by name – for example, to authenticate a prospective user by a given username and password.**

1. Open **src/cc/user/Principal.java**, and add a **find** method. At this level, all we can do is say whether we're the principal you want, or not:

```
public Principal find (String name)
{
  return name.equals (this.name) ? this : null;
}
```

2. Override this method in **src/cc/user/Group.java**:

```
@Override
public Principal find (String name)
{
  Principal result = super.find (name);
  Iterator<Principal> iterator =
    members.iterator ();
  while (result == null && iterator.hasNext ())
    result = iterator.next ().find (name);

  return result;
}
```

   – So we still allow that we may be the desired node; but if not we walk the list of children – which will result in a recursive navigation through the whole sub-tree.

Users and Groups

DEMO

3. Open **src/cc/users/Authenticate.java**, and see that a client application for the new method is just about ready to go.
4. In **findAndAuthenticate**, remove the placeholder value for the **found** variable, and un-comment the actual call to **realm.find**:

```
public static void findAndAuthenticate
  (String username, String password)
{
  System.out.print ("Testing " + username ...);

  Principal found = null;//TODO
    realm.find (username);
  if (found != null)
  {
    if (found instanceof User)
    {
      if (((User) found).getPassword ()
          .equals (password))
        System.out.println ("authenticated.");
      else
        System.out.println ("found, but not...");
    }
    else
      System.out.println ("found, but not...");
  }
  else
    System.out.println ("not found.");
}
```

Users and Groups

DEMO

5. The **main** method just tries this process out for different candidates:

```
public static void main (String[] args)
{
  findAndAuthenticate
    ("georgiaf", "razzledoozle");
  findAndAuthenticate ("Administrators", "");
  findAndAuthenticate
    ("gameshowhost", "comeondown");
  findAndAuthenticate
    ("gameshowhost", "tsohwohsemag");
}
```

6. Run the class now, and see that we're able to find all node types, know which is which, and authenticate against **User**.

```
Testing georgiaf ... not found.
Testing Administrators ... found, but not a User.
Testing gameshowhost ... authenticated.
Testing gameshowhost ... found, but not
authenticated.
```

Users and Groups

DEMO

- **Now, what if we were to want to authorize a User, once found, based on their identity?**
  - We could authorize by **access control list**, granting certain privileges to users by name.
  - We could take a **role-based** approach. Roles and groups are not exactly the same things, but often groups are used as if they were roles – sort of a hybrid of ACL and role-based styles.
- **Let's add authorization that supports either approach: the client will pass in one or more names, and we'll consider a user authorized if the user's own name, or any of the user's group names, are found in that list of roles.**
- **Note that this will require that we be able to walk up the tree.**
  - In the current implementation, the relationship between **Group** and **Principal** is **unidirectional**: i.e. you can navigate to children, but you can't navigate to parents.
  - You can see this, subtly, in the earlier UML diagram – the arrowhead indicates a unidirectional relationship.
  - And this is typical – one of those features that you may initially think excessive, trying to keep your encapsulations minimal, but that just about always wind up being required as you elaborate your design.

Users and Groups

DEMO

7. In **Principal.java**, add a **parent** property:

```
private Group parent;

public String getName ()
{
  return name;
}

public Group getParent ()
{
  return parent;
}
```

8. Now add the authorization method:

```
public boolean isAuthorizedAs (String... roles)
{
  for (String role : roles)
    if (role.equals (name))
      return true;

  if (parent != null)
    return parent.isAuthorizedAs (roles);

  return false;
}
```

Users and Groups

DEMO

- **Now we need to make sure that the parent is initialized when we build groups.**
  - This management of the bidirectional relationship can be one of the more challenging parts of a Composite design, because it's so easy to let **parent** in one place and **children** in another fall out of synch.

9. In **Group.java**, update each of the methods that modify the **members** list to be sure that they also set **parent** on the added/removed object:

```
public void addMember (Principal member)
{
  for (Principal existingMember : members)
    if (member.getName ().equals
        (existingMember.getName ()))
      throw new IllegalArgumentException ("...");

  members.add (member);
  member.setParent (this);
}

public void removeMember (Principal member)
{
  members.remove (member);
  member.setParent (null);
}
```

Users and Groups

DEMO

10. Again we have a client-in-waiting: open **src/cc/user/Authorize.java** and un-comment the bulk of the **findAndAuthorize** method:

```
public static void findAndAuthorize
  (String username, String... roles)
{
  System.out.print ("Testing " + username + ...);
  //TODO
  /*
  Principal found = realm.find (username);
  if (found != null)
  {
    if (found.isAuthorizedAs (roles))
      System.out.println ("authorized.");
    else
      System.out.println ("found, but not ...");
  }
  else
    System.out.println ("not found.");
  */
}
```

11. The **main** method again tests out a few candidates:

```
public static void main (String[] args)
{
  findAndAuthorize ("Administrators", "...");
  findAndAuthorize ("gameshowhost", "...");
  findAndAuthorize ("wprovost", "...");
}
```

12. Test and see that you can walk up the tree and match names to roles:

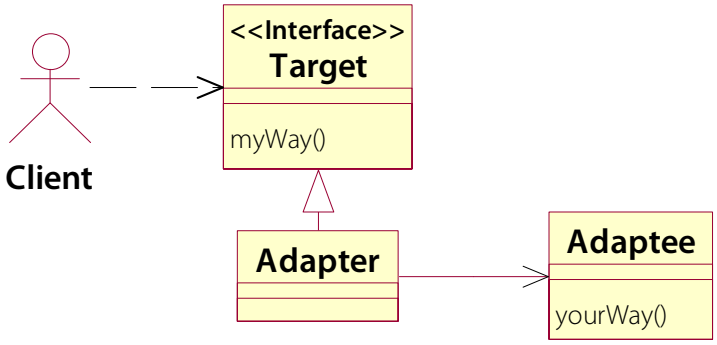
```
Testing Administrators ... authorized.
Testing gameshowhost ... found, but not authorized.
Testing wprovost ... authorized.
```

The Adapter Pattern

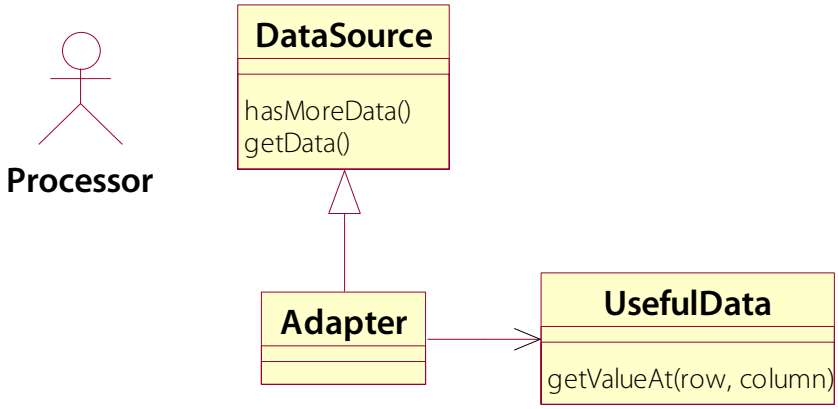
- **The Adapter pattern addresses a very general problem: getting two unlike objects to work together.**
- **Adapter is about getting one object to be able to use another.**
  - We assume that we can't or **shouldn't modify** either of the things, but instead must find a way to get them talking to each other.
  - Imagine an existing class, method, or larger system that has already been written to use a certain type of delegate object.
  - Call this delegate type **A**.
  - Then assume we have an object or objects of type **B** that we'd like to plug into this system.
  - The problem, then, is getting **B** to look and act like **A**.
- **Consider a hypothetical processing component that abstracts its DataSource.**
  - A standard **DataSource** implementation is part of the component.
  - We have a class **UsefulData** from another system that can provide data that would be useful to this component, but there's a misfit between it and the processor's expectations – i.e. **DataSource**.
  - We can't change either class – how then to integrate them?
- **A related pattern, Mediator, is distinguished from Adapter in that it is concerned with mediating a two-way conversation between unlike communicants – translating their protocols, if you will.**

The Adapter Pattern

- The solution is to subclass A – called the **Target** – and create an **Adapter** that makes B "fit the mold."



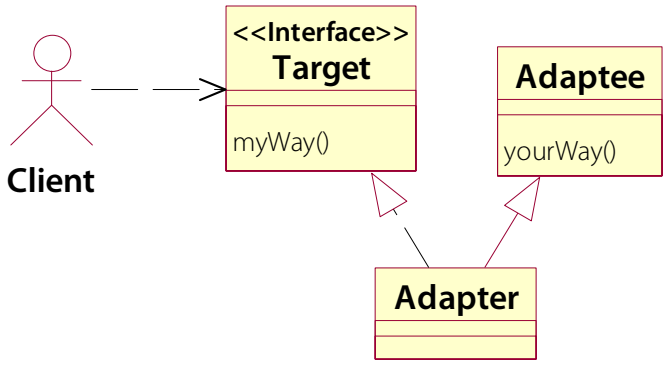
  - Calls on the previously-understood **Target** interface or class will be handled in a way that brings the **Adaptee** into play by private delegation.
  - Our **DataSource**-vs.-**UsefulData** problem is resolved with an adapter, as in:



  - Strictly speaking, this is a specific application strategy known as the **Object Adapter**.
  - There is a **Class Adapter** that allows one instance instead of two, and no delegation; but it requires multiple inheritance.

The Adapter Pattern

- **Adapter examples abound in the Core API:**
  - The **java.util.Collections** utility can convert from one collection type to another – **Enumeration**s to **List**s, etc.; can create **unmodifiable** or **thread-safe wrappers** over existing collections; and a bit later we'll talk about adaptations with **Stream**s.
  - A **JDBC Driver** is a kind of adapter.
  - The **java.io streams API** provides several adaptation points, the most obvious being **InputStream** and **OutputStream**.
  - **JFC table and tree models** break out interfaces for providing data, managing row, column, or node selection, rendering cells or nodes, and editing cells. All of these abstractions are **adaptable** to an application's own model.
- **Returning to the Class Adapter strategy, it's worth considering if the Adapter might extend the Adaptee:**

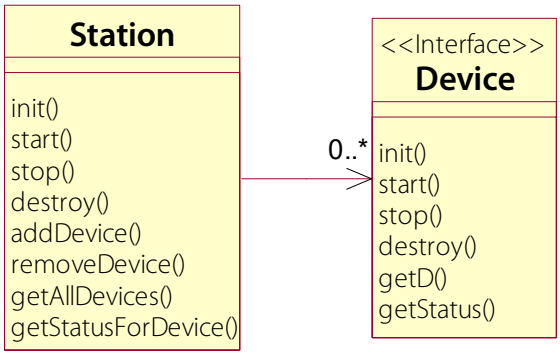


  - In C++ this is generally workable, because we have **multiple implementation inheritance** in that language.
  - In Java, the strategy is limited to situations in which the Target is an **interface**; but fortunately that is a fairly common case.

Adapting Monitorable Devices

DEMO

- In **Station_Step1** is the kernel of a management, monitoring, and control system in which a **Station** can keep track of any number of **Device**s:



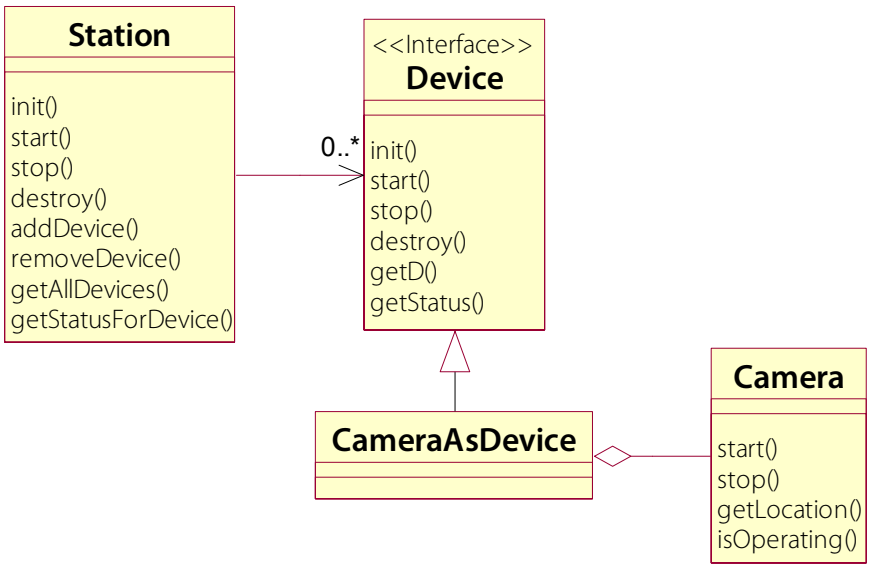
- The **Device** interface is our Target:

```
public interface Device
{
  public String getID ();
  public String getStatus ();
  public void init (Station station);
  public void destroy ();
  public void start ();
  public void stop ();
}
```

Adapting Monitorable Devices

DEMO

- There are also a number of devices – let's imagine that they are all from different manufacturers or other vendors – and each has certain lifecycle and monitoring methods of its own.
  - There's a **security camera**.
  - There are **elevators** ... so, let's guess we're in an office building!
  - There are **thermostats**.
- But none of these devices implements **Device**.
- To integrate them into our system, we will need adapters. We'll create the first one in this demonstration:



- You'll build two more in the upcoming lab exercise.
- For each, there will be different challenges; it all boils down to the basic problem of adapting unlike things, as there won't always be an obvious one-for-one correspondence between the semantics of one and the semantics of the other.

Adapting Monitorable Devices

DEMO

1. Review the test client in **src/cc/monitor/RunLifecycle.java**.
   - A helper method runs a target **Station** through a typical site lifecycle, adding any number of given **Device**s to it and then tracing execution throughout:

```
public static void exercise (Device... devices)
{
  Station station = new Station ();

  for (Device device : devices)
    station.addDevice (device);
  ...
  System.out.println ("Initializing...");
  station.init ();
  System.out.println ("...initialized.");
  ...
  System.out.println ("Starting...");
  station.start ();
  System.out.println ("...started.");
  ...
}
```

   - The **main** method is ready to call **exercise** – but of course is barred from doing so because **Camera** doesn't implement **Device**.

```
public static void main (String[] args)
{
  Camera cam1 = new Camera ("Basement");
  Camera cam2 = new Camera ("Lobby");
  ...
  exercise
    (
    );
}
```

Adapting Monitorable Devices

DEMO

1. Create the adapter class **cc.security.CameraAsDevice**, and make it implement **Device**:

```
public class CameraAsDevice
  implements Device
{
}
```

2. Give it a delegate **Camera**:

```
public class CameraAsDevice
  implements Device
{
  private Camera camera;

  public CameraAsDevice (Camera camera)
  {
    this.camera = camera;
  }
}
```

3. Implement **getID** and **getStatus** – determining in the latter method the operating status based on the **Camera**'s **operating** property:

```
  public String getID ()
  {
    return camera.getLocation () + " camera";
  }

  public Status getStatus ()
  {
    return camera.isOperating ()
      ? Status.RUNNING
      : Status.STOPPED;
  }
```

Adapting Monitorable Devices

DEMO

4. Now add the lifecycle methods.
   - The **Station** has the distinct concepts of **init/destroy** and **start/stop**. These could have very formal meanings, or more generally indicate that heavyweight initialization and cleanup might best be done in the outer pair of **init** and **destroy**, while lighter operations – maybe a camera ceasing to transmit images, even while it stays powered on? – belong in **start** and **stop**.
   - The **Camera** only offers **start** and **stop**, and let's say that these are most sensibly aligned with **start** and **stop** on **Device**:

```
public void init (Station station)
{
}

public void start ()
{
  camera.start ();
}

public void stop ()
{
  camera.stop ();
}

public void destroy ()
{
}
```

Adapting Monitorable Devices

DEMO

5. Now, in **RunLifecycle.java**, we can add the cameras to our station, using the adapters:

```
public static void main (String[] args)
{
  ...
  exercise
    (
      new CameraAsDevice (cam1),
      new CameraAsDevice (cam2)
    );
}
```

6. Test, and see that the camera's own diagnostic output meshes with that of the test application – how convenient!

```
Added 2 devices.
...
Starting station ...
  Basement camera started.
  Lobby camera started.
Station started.
Device status:
  Basement camera          RUNNING
  Lobby camera             RUNNING
Stopping station ...
  Basement camera stopped.
  Lobby camera stopped.
Station stopped.
...
Device status:
  Basement camera          STOPPED
  Lobby camera             STOPPED
Removed 2 devices.
```

- The completed demo is found in **Station_Step2**.

Adapting Elevator and Thermostat

LAB 4A

Suggested time: 45 minutes

In this lab you will build adapters for two more devices, each of which will require a new trick or two. Elevator will need to react to **init** and **destroy**, rather than **start** and **stop**, and in fact it will require that multiple methods be called for **init**. It also has its own status logic, based on whether it is currently serving a call; and, it has no natural ID, which means that your adapter will have to synthesize one.

Thermostat poses a different sort of challenge, in that it was not designed to be re-initialized or re-started: it's a one-shot object that implicitly starts when created, and then stops when closed. You need to be able to re-initialize and re-start, so you'll need your adapter to manage the repeated creation of the delegate object, rather than being handed a delegate that's there for as long as you need it as with the other two types of devices.

Detailed instructions are found at the end of the chapter.

Keeping It Flowing

- **By this point in the evolution of the Java language, there are many ways in which one might choose to accept large volumes of data in a method, or to return data from a method.**
  - Going way back, we have the **Enumeration**.
  - For a while after that the best practice was to use **Iterator**s over collections.
  - Then came **Iterable<E>** and the simplified **for** loop, and passing iterators around fell out of fashion. As of this writing the most common style is to accept and/or return a **List<E>**, **Set<E>**, or **Map<K,V>** as appropriate to the method semantics.
- **Various APIs of various ages use different techniques, and so it's a common problem in Java to adapt the data you have – or that you get from one API – to the requirements of another API.**
- **It can also be another one of those places in which we have to adapt between "push" and "pull" models: eager- vs. lazy-loading expectations, or immediate vs. deferred processing.**
- **The new kid on the block is the Stream<T>, introduced in Java 8; and this deserves some special consideration.**
  - Streams practice deferred processing – even when apparently modified through methods such as **filter**, **map**, and even **sort**. Everything happens at the last second, when a call comes to a "terminating method" such as **forEach** or **reduce**.
  - This makes them, potentially, much more **efficient**.
  - They can also support **parallel processing** on multiple cores.

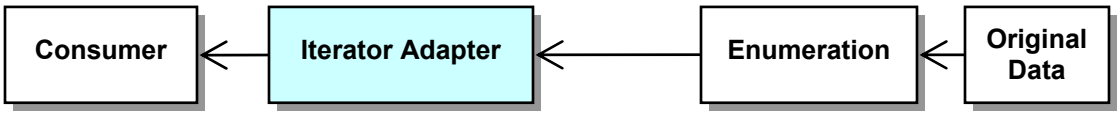
Keeping It Flowing

- Here's an Adapter warning sign: consider another example, in which a method you want to call expects an **Iterator** as a method parameter; but the data you have to offer is wrapped in an **Enumeration**.
- The easiest solution in this case might be to create a collection and to populated it with the data:

```
List<X> temp = new ArrayList<X> ();
while (myEnum.hasMoreElements ())
  temp.add (myEnum.nextElement ());
processThis (temp.iterator ());
```

  - This is clean enough, but it means cloning the data:

- Instead of preparing data in a required form, why not take advantage of the flexibility of **Iterator**, and adapt it to your data?
  - You could build your own **Iterator** that passes **next** to **nextElement** on the delegate enumeration.



  - Or, you could use the one provided in the Collections API! In fact it's a full **List** adapter, backed by an **Enumeration**:

```
processThis (Collections.list (myEnum).iterator());
```

Merging Large Inventories

LAB 4B

Suggested time: 45 minutes

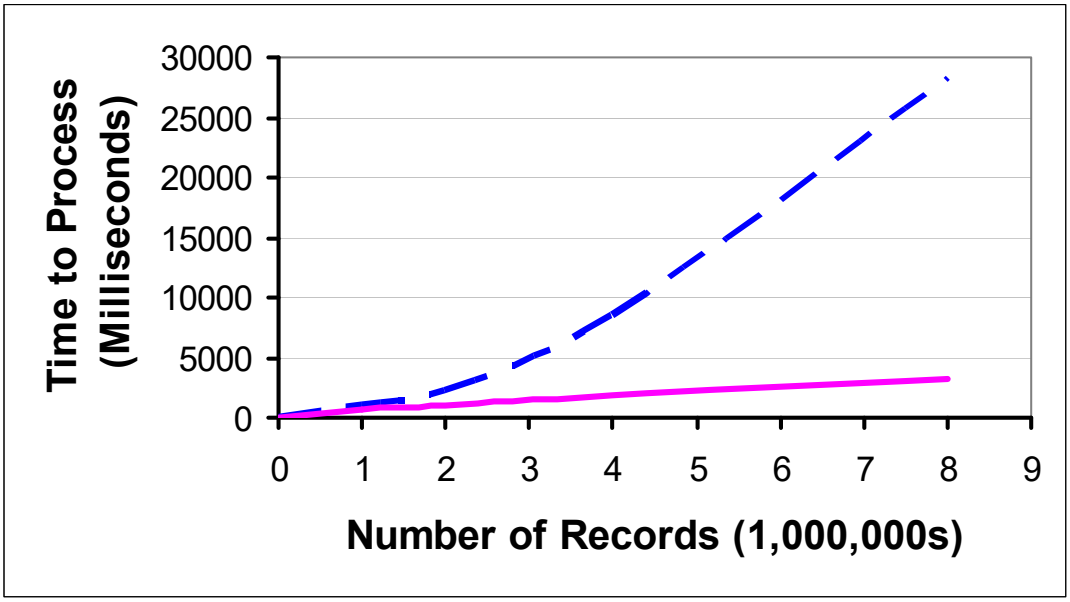
In this lab you will refactor an application that merges existing inventory files from multiple sites into a single inventory file. The "before picture" application works, but it is inefficient and doesn't scale well at all. You'll see the warning sign that the application takes the shortest route to meeting its requirements by pouring the contents of each site file, as loaded, into a single **List**, and then sorts that list in place.

You'll refactor by building an adapter that (a) defers processing by functioning as an **Iterable**, and (b) applies a more intelligent sort algorithm that takes advantage of its position as the initial reader of each site file, before they've all been thrown together.

Detailed instructions are found at the end of the chapter.

Cost of Intermediate Storage

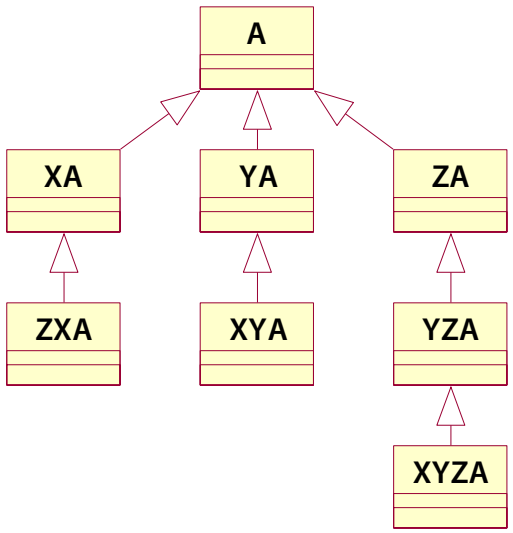
- **Here again are the metrics from the previous lab, in the form of a chart of processing time vs. the size of the merged inventory:**



  - The dashed line shows the performance when we loaded a list with all the records, and then sorted it in place before passing it along to the signature component.
  - The solid line shows the performance of your adapter.
- **So, first off, your adapter is much faster, which is great.**
- **More compelling is that the processing time seems to progress linearly against the size of the inventory.**
  - The best you can do with large volumes of data is to process iteratively and "forget" each record when you're done.
  - The Adapter has let you avoid the costs of creating new, standing data structures in memory, and instead preserve the deferred-processing model that you're given by **Files.lines**.

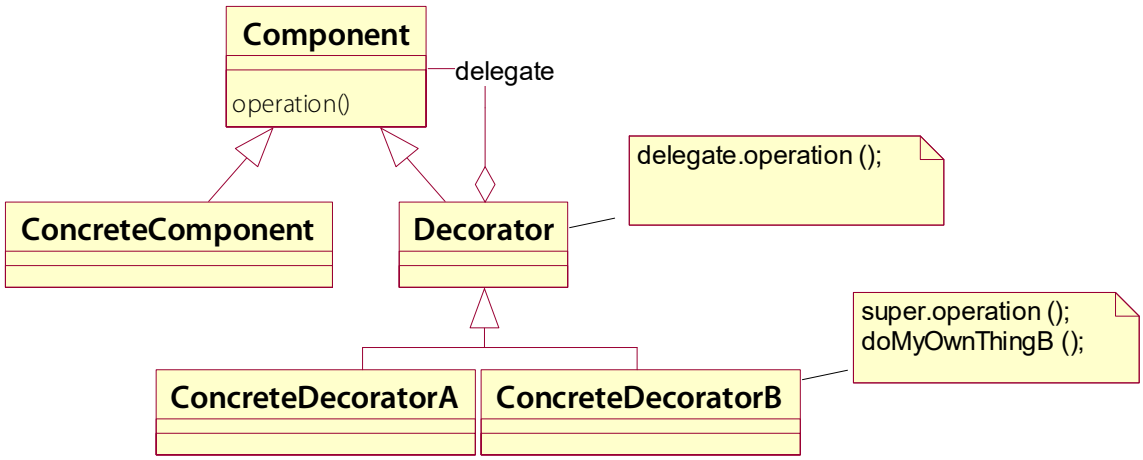
The Decorator Pattern

- The **Decorator** pattern offers an alternative to inheritance for providing **progressive specialization**.
- The problem occurs where one or more classes should offer the same behavior, but then each one should also specialize by adding its own behaviors.
- On the surface, this seems like a perfect job for inheritance! Problem solved, no?
  - Inheritance is only so flexible.
  - Especially, Decorator poses the problem of **several independent specializations** –lets say there's a base class that does A and we want specializations X, Y, and Z.
  - Decorator also addresses the issue of **mixing and matching** specializations – we want to be able to instantiate an object that does X+Y+A, and another that does only Z+A.
  - Inheritance in Java would not support this; or it might, but things quickly get ugly.
  - Even assuming that order is unimportant – that is that XYA and YXA are functional equivalents – the diagram at right shows what can happen.
  - Consider inheritance graphs like this one a clear warning sign for the Decorator pattern.

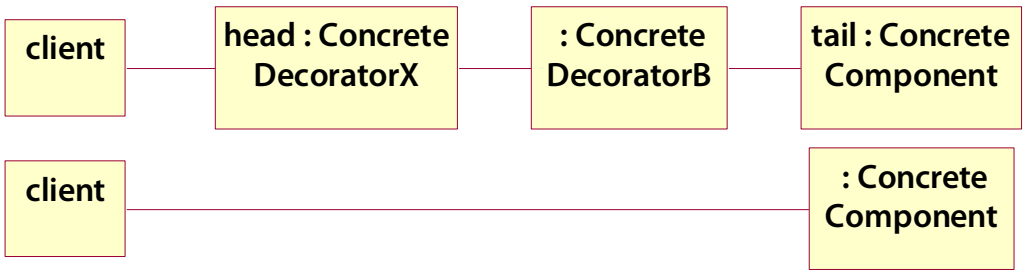


The Decorator Pattern

- The solution takes an approach based in **delegation** rather than inheritance:



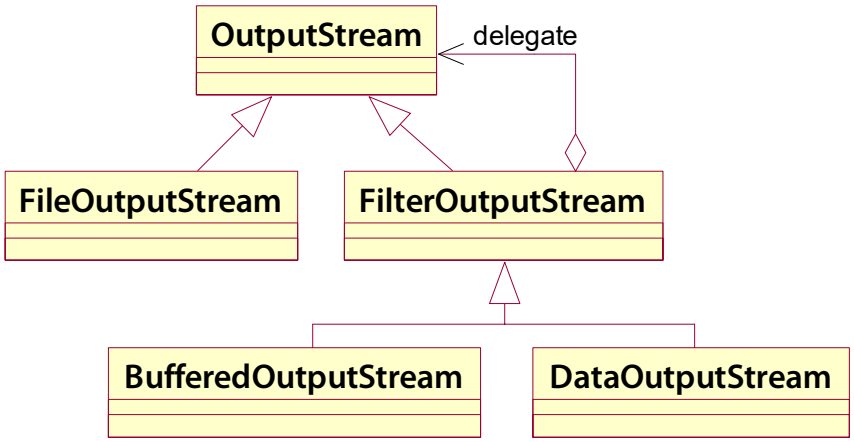
  - There is a single derivation of the base type called the **Decorator**; it adds only the ability to create **chains** of objects, each said to **decorate** the next one in the chain.
  - Subtypes of **Decorator** define the independent decorations, and can then be mixed and matched arbitrarily.
  - The following UML **object diagrams** illustrate the flexibility of a decorator system – each shows a possible chain:



  - This is much more maintainable and flexible than an inheritance-based approach.

The Decorator Pattern

- A classic Decorator example is the **Java streams** model: for instance, a **BufferedOutputStream** decorates a **FileOutputStream**:




```
OutputStream out = new BufferedOutputStream
  (new FileOutputStream ("MyFile.dat");
otherObject.writeToMyStream (out);
```

- We get polymorphism in **writeToMyStream** (which we assume takes an **OutputStream** argument), and mix-and-match flexibility for the caller.

- **Another example is JFC Borders; in fact it was GUI design that originated this pattern, hence the rather specific term "to decorate."**

A Bank's Account Products

LAB 4C

Suggested time: 30-45 minutes

In this lab you will refactor and then enhance an application that models bank accounts. The classic bank-account examples are inheritance-based: CheckingAccount as subclass of Account, etc. But, in the real world, banks are constantly rolling out new products, many of which are simply fresh combinations of familiar features: overdraft protection, different transaction or per-statement-cycle fee structures, minimum balances, etc. This is actually an encapsulation that is ripe for the Decorator pattern!

You will begin by refactoring the existing class into a base type, a concrete implementation, and a base decorator type. You will then implement several different decorators, and test them in various combinations. In the process you will see both the power of Decorator and some of its limitations.

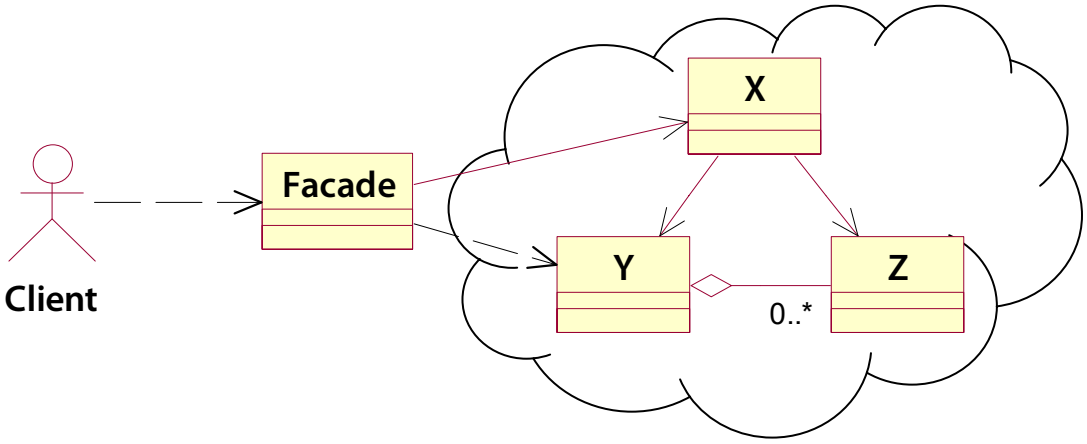
Detailed instructions are found at the end of the chapter.

The Façade Pattern

- **The Façade pattern helps to simplify the use of a federation of closely-related classes.**
  - Often a **subsystem** of classes is defined to do a complex job.
  - Invoking the processes of that subsystem shouldn't itself be a complex process, however, and we may want decoupling from the subsystem internals for its own sake.
  - We want to give the caller **clean, convenient** methods to call.

The Façade Pattern

- **The solution is to define a separate class just for the purpose of mediating conversation with the subsystem.**



  - This class will offer a simplified interface that hides the inner workings while giving the caller good high-level ability to direct the activity of the subsystem.
- **There are many examples in the Core API ...**
  - The **Java Sockets API** uses a complex set of classes to model URLs, addresses, network interfaces, and protocols, but most applications use **Socket** and **ServerSocket**.
  - **RMI** involves a bewildering nest of remote references, stubs, publishable objects, and registries, but most applications can be content with **LocateRegistry** and **Naming**.
- **... and in our applications ...**
  - HR defines a domain model of fine-grained encapsulations, but then controls them using a layer of coarse-grained services – see both domain classes and services in the **cc.hr** package.
  - The Car **Dealership** is a façade for the entire **cc.cars** package.

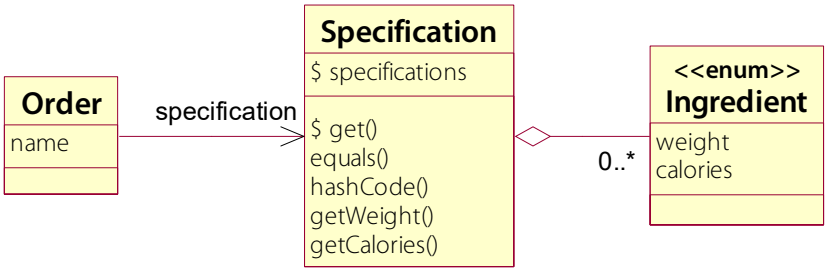
The Flyweight Pattern

- **The Flyweight pattern deals with types that can have only a finite number of possible states.**
  - These are often simple types that wrap **one or two values** and add some behavior: color choices, mode switches, status flags, or even printable characters in a word-processing application
  - Thus it is that much more tempting for client code to **create many** of these objects, and a high ratio of objects in memory to distinct objects in memory is bad for efficiency and performance.
- **For fixed pools, the Java enum implements the Flyweight pattern nicely, and gives it native language support.**
  - Consider **stateful** and **behavioral enums** where you need to encapsulate more than one value, or specific behavior.
- **The solution is to create a pool of objects. It's something like a Singleton expression, except that a Flyweight will only allow one instance to exist for each distinct state.**
  - Define **equivalence criteria** for the object type.
  - **Hide the constructor** to control object creation, and expose a **factory method** instead.
  - Keep a **cache of distinct objects** as you create them, and always return an existing object when found.
- **Flyweights may also be data-driven: a lookup table in a database might hold a fixed number of rows, each to be incarnated as one Java object of your Flyweight type.**

Ingredients and Pizzas

EXAMPLE

- In **Pizza** is a domain model for a pizza shop that includes both a fixed Flyweight and an open Flyweight.



  - Many thanks to Max Rahder for this contribution to the course!

- The fixed Flyweight in **src/cc/pizza/Ingredient.java** – a simple, stateful enumerated type:

```
public enum Ingredient
{
  onion (60, 80),
  pepper (80, 100),
  tomato (120, 200),
  broccoli (100, 200),
  mushroom (80, 100),
  pepperoni (120, 700),
  anchovies (80, 400),
  sausage (120, 800);

  public final int weight;
  public final int calories;

  private Ingredient (int weight, int calories)
  {
    this.weight = weight;
    this.calories = calories;
  }
}
```

Ingredients and Pizzas

EXAMPLE

- The pizza shop gets a lot of orders – but not that many distinct types of pizza. So **src/cc/pizza/Specification.java** captures distinct pizza specifications, as an open Flyweight:

```
public class Specification
{
  private static Set<Specification>
    specifications = new HashSet<> ();

  private Set<Ingredient> ingredients =
    new HashSet<> ();
```

  - We hide the constructor, which just populates the unique set of ingredients for this specification ...

```
  private Specification (Ingredient... ingredients)
  {
    super ();
    for (Ingredient i : ingredients)
    {
      this.ingredients.add (i);
    }
  }
```

Ingredients and Pizzas

EXAMPLE

- ... so that the factory method can first check the static set of all specifications, and possibly return one that suits the caller's requirements, before proceeding to create a new one if needed:

```
public static Specification get
  (Ingredient... ingredients)
{
  Specification newSpec =
    new Specification (ingredients);
  for (Specification spec : specifications)
    if (spec.equals (newSpec))
      return spec;

  specifications.add (newSpec);
  return newSpec;
}
```

- For this to work, we must define equivalence logic (and, not shown, we also provide a meaningful hash code):

```
@Override
public boolean equals (Object that)
{
  return ((Specification) this).ingredients
    .equals (((Specification) that).ingredients);
}
```

- Then an **Order** is just a pair of **Specification** and the customer's **name**, and we might have many of these while keeping the number of unique specifications down.

Ingredients and Pizzas

EXAMPLE

- Run the application class **OrderSomePizza** to see that, though we create many orders ...

```
public static void main(String[] args)
{
  orders.add (new Order ("Smith",
    Ingredient.mushroom));
  orders.add (new Order ("Jones",
    Ingredient.pepper, Ingredient.sausage));
  ...
```

- ... we implicitly create a smaller number of specifications:

```
10 orders were taken:

Weight(g) Calories  Ingredients
--------- --------  --------------------------
       80      100  mushroom
      200      900  sausage, pepper
      240     1500  sausage, pepperoni
      240     1500  sausage, pepperoni
      180      780  onion, pepperoni
       80      400  anchovies
      140      180  onion, pepper
      180      780  onion, pepperoni
      140      180  onion, pepper
       80      100  mushroom

There are 6 types of pizza for which
  we're storing data.
```

Structural Refactoring

LAB 4D

Suggested time: 30 minutes

In this exercise you will analyze a single "before picture" 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 found 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.

SUMMARY

- **These patterns are truly structural – the focus on organizing state elements and class relationships is immediately apparent.**
- **Since these patterns have to do with structure and class relationships, many of them are driven by language features that affect those relationships:**
  - **Polymorphism and Reflection** influence specific strategies under the Composite pattern, whose full expression often calls for **instanceof** testing.
  - **Package design** and the Façade pattern go hand in hand.
  - Decorator offers a more flexible but also more complicated means of extending an encapsulation. So it is an alternative to **inheritance** but also compromises some **Reflection** capability.
  - Attempts to apply Adapter are often foiled by fine points of **visibility** – such as not being able to read or to write a base-class field – and **inheritance** – especially where an otherwise ideal target class is defined to be **final**!

Adapting Elevator and Thermostat

LAB 4A

In this lab you will build adapters for two more devices, each of which will require a new trick or two. Elevator will need to react to **init** and **destroy**, rather than **start** and **stop**, and in fact it will require that multiple methods be called for **init**. It also has its own status logic, based on whether it is currently serving a call; and, it has no natural ID, which means that your adapter will have to synthesize one.

Thermostat poses a different sort of challenge, in that it was not designed to be re-initialized or re-started: it's a one-shot object that implicitly starts when created, and then stops when closed. You need to be able to re-initialize and re-start, so you'll need your adapter to manage the repeated creation of the delegate object, rather than being handed a delegate that's there for as long as you need it as with the other two types of devices.

Lab project: **Station_Step2**

Answer project(s): **Station_Step3** (intermediate)
Station_Step4 (final)

Files: * to be created
src/cc/transport/Elevator.java
src/cc/transport/ElevatorAsDevice.java *
src/cc/hvac/Thermostat.java
src/cc/hvac/ThermostatAsDevice.java *
src/cc/monitor/RunLifecycle.java

Instructions:

1. Review **Elevator.java** and see that it thinks of status in terms of whether it is currently **ready** (powered up and waiting in the lobby) and whether it has been **called**.
2. Create the new class **cc.transport.ElevatorAsDevice**, and make it implement **Device**.
3. Give it a delegate **elevator** and create a constructor that lets you accept a reference to the delegate – just as we did in **CameraAsDevice**.
4. You will have to model an ID for the delegate device, so add a field **ID** of type **int**. Add a parameter and logic to the constructor to accept this value on creation as well.
5. Implement **getID** to return a string of the form "Elevator *N*".
6. Implement **getStatus** to determine status first by checking the **ready** state of the **elevator**: if not **ready**, report that the device is **Status.STOPPED**. If **ready**, then check to see if it is also **called**: if so, report that it's **Status.RUNNING**, and if not report that it's **Status.IDLE**.
7. Implement **init** to call both **powerUp** and **goToLobby** on the delegate.

Adapting Elevator and Thermostat LAB 4A

8. Implement **destroy** to call **powerDown**.
9. Do nothing in your **start** and **stop** methods.
10. In **RunLifecycle.java**, you can now add the pre-configured **Elevator**s to the call to **exercise**, using your new adapter for each. Set IDs 1, 2, and 3 as you create the three adapter objects.
11. Run this class, and see that the elevators respond correctly throughout the test; this is the intermediate answer in **Station_Step3**.

```
Added 5 devices.

Initializing station ...
  Elevator powered up.
  Elevator sent to lobby.
  Elevator powered up.
  Elevator sent to lobby.
  Elevator powered up.
  Elevator sent to lobby.
Station initialized.

Starting station ...
  Basement camera started.
  Lobby camera started.
Station started.

Device status:
  Basement camera         RUNNING
  Lobby camera            RUNNING
  Elevator 1              IDLE
  Elevator 2              IDLE
  Elevator 3              RUNNING

Stopping station ...
  Basement camera stopped.
  Lobby camera stopped.
Station stopped.

Destroying station ...
  Elevator powered down.
  Elevator powered down.
  Elevator powered down.
Station destroyed.

Device status:
  Basement camera         STOPPED
  Lobby camera            STOPPED
  Elevator 1              STOPPED
  Elevator 2              STOPPED
  Elevator 3              STOPPED

Removed 5 devices.
```

Adapting Elevator and Thermostat LAB 4A

12. Now let's turn our attention to **Thermostat.java**. Again, this will be the trickiest of the three, because its lifecycle is radically different: it can't be reused once shut down.
13. Create the new class **cc.hvac.ThermostatAsDevice**, adapting the Target **Device**.
14. Give it a delegate **thermostat** – but don't initialize this in the constructor as we've done for the other two adapters; hold tight, and we'll do this elsewhere.
15. Define fields **ID**, **heatThreshold**, and **coolThreshold**, all to align with the same three fields on the Adaptee **Thermostat**. You won't need getter or setter methods – except that you need a **getID** method as part of your **Device** implementation, so you may as well fill that one in now.
16. Create a constructor that accepts these three values as parameters and initializes the three fields.
17. In the **start** method, create a new **Thermostat** and pass your three values for ID and heating/cooling thresholds. Set **thermostat** to refer to this new object.
18. In **stop**, call **thermostat.close**, and to set **thermostat** to **null**.
19. You'll do nothing in your **init** or **destroy** method for this one.
20. Implement **getStatus** to derive operating status, first by testing if we have a **thermostat** at all. That is, if **thermostat** is **null**, we know we are not between calls to **start** and **stop**, which means we should return **Status.STOPPED**.
21. Otherwise, call **thermostat.getFunction**, and translate **Thermostat.Function.IDLE** to **Status.IDLE**, and anything else to **Status.RUNNING**.
22. In **RunLifecycle.java**, you will have to instantiate your adapters instead of the **Thermostat** objects that are already defined – because your adapter will be responsible for creating the delegate **Thermostat**s as the lifecycle progresses. Replace the type **Thermostat** with **ThermostatAsDevice** in each of the two declarations, and you will then be able to add **therm1** and **therm2** directly to your call to **exercise**.

    Note that the existing calls to set the current temperature on each thermostat will have to wait a while – we don't actually have that implemented yet. Just comment these out, for now.
23. If you test now, you should see that your adapter is playing nicely with the others, and the test output includes tracing of the creation and closure of each **Thermostat** object.

    But, the status of each will never show as "IDLE" – because, as yet, we don't control the current temperature of the unit, and so it defaults to zero – and then makes the very sound decision that it should engage the heating function!

Adapting Elevator and Thermostat LAB 4A

24. Add a **temp** property to your adapter class, with getter and setter methods. Note that **setTemp** must check to see that **thermostat** is not **null**, before calling **setTemp** on it.
25. Since temperature can change at any point in the lifecycle, you should also update **start** to pass any **temp** already known to the adapter along to the newly-created **thermostat**.
26. Now you can un-comment those calls to **setTemp** in **RunLifecycle.main**.
27. Test again, and now you should see that the status of the "Server room" thermostat, once started, is "RUNNING"; the "Lobby" thermostat correctly remains idle, because we set it to a temperature in between the two thresholds. The parts of the program output for the thermostats are reproduced here:

```
Initializing station ...
  ...
Station initialized.

Starting station ...
  ...
  Lobby thermostat created.
  Server room thermostat created.
Station started.

Device status:
  ...
  Lobby thermostat        IDLE
  Server room thermostat  RUNNING

Stopping station ...
  ...
  Lobby thermostat closed.
  Server room thermostat closed.
Station stopped.

Destroying station ...
  ...
Station destroyed.

Device status:
  ...
  Lobby thermostat        STOPPED
  Server room thermostat  STOPPED
```

This is the final answer in **Station_Step4**.

28. To be certain that your strategy of creating and re-creating the delegate objects is working, you might try adding a second call to **exercise** to the **main** method, passing the same set of devices, or just **therm1** and **therm2**.

Merging Large Inventories

LAB 4B

In this lab you will refactor an application that merges existing inventory files from multiple sites into a single inventory file. The "before picture" application works, but it is inefficient and doesn't scale well at all. You'll see the warning sign that the application takes the shortest route to meeting its requirements by pouring the contents of each site file, as loaded, into a single **List**, and then sorts that list in place.

You'll refactor by building an adapter that (a) defers processing by functioning as an **Iterable**, and (b) applies a more intelligent sort algorithm that takes advantage of its position as the initial reader of each site file, before they've all been thrown together.

Lab project: **Inventory_Step1**

Answer project(s): **Inventory_Step2**

Files: * to be created
src/cc/inventory/GenerateInventories.java
src/cc/inventory/MergeInventories.java
src/cc/inventory/LabSorter.java *

Instructions:

1. In order to prepare data sets large enough to make for measurable differences in performance, this lab includes a Java class whose job is to generate sets of inventory files, for three sites. Run **cc.inventory.GenerateInventories** as a Java application. It will randomly strew a million inventory records – nothing fancy, just part number and quantity – into three files. Each file will hold its records in alphabetical order by the part "number" which is actually a six-letter string.
2. You can review the files, if you like: refresh the Eclipse project and look in the newly-created **inventory** folder.
3. Now, run **cc.inventory.MergeInventories**, and see that it times itself:

```
Processed 1000000 records in 0.942 seconds.
```

4. If you refresh and look in the **inventory** folder again, you'll see a fourth file, with all the records merged and still in alphabetical order.

Merging Large Inventories LAB 4B

5. Review **MergeInventories.java**. See the warning sign: it pours contents from each file, line-by-line, into a **List<String>**, and then sorts the list:

```
List<String> fullInventory = new ArrayList<> ();
for (int f = 0; f < FILENAMES.length; ++f)
    fullInventory.addAll
        (Files.lines (folder.resolve (FILENAMES[f]))
              .collect (Collectors.toList ()));

Collections.sort (fullInventory);

Signer signer = new Signer ("inventory/Complete");
int records = signer.sign (fullInventory);
```

   It doesn't do this arbitrarily, but because it also needs to affix a digital signature to the end of the file, certifying the contents that it's created. The **Signer** that it uses – we take this to be a third-party component, not subject to any refactoring on our part – accepts an **Iterable<String>** – a perfectly reasonable choice, even if the client code is not making the most of it.

6. Test the same process a few more times – but each time, increase the number of records that must be merged. You can pass record counts to **GenerateInventories** – just set the total number as a program argument. Try two million, four million, etc. Run **MergeInventories** on each new set of generated files: you'll see that the processing times increase disproportionately – logarithmic time? quadratic? but definitely not linear.

   Here are the results captured at the time of writing, on an i7 chip with 8meg of memory on hand:

```
Processed 1000000 records in 0.942 seconds.
Processed 2000000 records in 2.252 seconds.
Processed 4000000 records in 8.441 seconds.
Processed 8000000 records in 28.291 seconds.
```

   So, both theoretically and empirically ... it's not looking good. Let's see about improving the code design.

7. Create a new class **cc.util.ListSorter**, parameterized on any **Comparable** type. This turns out to be rather a mouthful, thanks to the definition of **Comparable**:

```
public class ListSorter<E extends Comparable<? super E>>
{
}
```

8. Declare a field **sources** that is a **List** of **Iterators**. The iterators will have to be over a wildcard extending your type **E**; this allows the client to pass iterators over collections of subtypes of **E**, so long as you don't do anything to modify the sources, which you won't need to do. So ...

```
private List<Iterator<? extends E>> sources;
```

Merging Large Inventories LAB 4B

9. Define a constructor that takes a similar list of iterators as a parameter, and initialize sources accordingly.
10. Make the class implement **Iterable<E>**. This will require that you implement the **iterator** method to provide an iterator over type **E**. So, first, you'll need to build a custom iterator ...
11. Define an inner class **SortedMultiIterator** that implements **Iterator<E>**.
12. Give it a **sources** field as well, exactly like the one on the outer class.
13. Also define a field **candidates**, of type **List<E>**, and a field **size**, of type **int**.

    Your sorting strategy will be to read the next element from each of any number of given iterators, and keep all of those "next elements" in a cache – this is the **candidates** list. Since the sources are all pre-sorted, whenever anyone asks our iterator for the next element, it is sure to find the appropriate element in that cache, and can select the right object by comparing just those elements. Each time an element is returned from the cache, you'll get the next element from the associated source – or you'll set that slot in the cache to **null**, indicating that the associated source is now exhausted. When all sources are exhausted, your iterator is done, too, and your **hasNext** method will return **false**.
14. Start in the constructor – which will take a list of iterators and use that to initialize the **sources** field.
15. Then, set **size** to the results of a call to **sources.size**.
16. Initialize **candidates** to a new **ArrayList<E>**, passing **size** to pre-allocate the underlying array and save the trouble of re-allocating later.
17. Run a loop over **sources**, and for each source **Iterator**, add the first element to **candidates** – or, just in case you're passed an empty iterator, add **null** if the source iterator's **hasNext** returns **false**.
18. Implement **hasNext** to return **true** if any of the elements in **candidates** is non-**null**, and false if they are all **null**.
19. Implement **next**, first by calling **hasNext** and returning **null** if it returns **false**.
20. Then, initialize a variable **result** of type **E** to **null.**
21. Set an **int** variable **winner** to -1.
22. Loop from zero to one less than **size**.
23. In the loop, get a reference **candidate** to the Nth element in **candidates**.

Merging Large Inventories LAB 4B

24. Check **candidate**: if it is not **null**, and if either **result** is **null** or **candidate** is less than **result** (use **candidate.compareTo**, which is guaranteed to be there since your type **E** is **Comparable**) then set **result** to **candidate**, and set **winner** to the loop index.

    So, you're checking every element in the cache (each of which is the "lowest" element in its source collection) to see which is the "lowest" according to the comparison logic for the given type **E**. Whoever is the "winner" will be the return value from the method, and you capture the winning index so that you can draw from the associated iterator to re-fill the cache.

25. After the loop, initialize a variable **source** to the result of a call to **sources.get**, passing **winner** as the index.

26. Now set the value at index **winner** in the **candidates** list to the **next** element in **source** – again, unless **source.hasNext** returns **false**, in which case you will set the replacement candidate to **null**.

27. Now, **return result**.

28. You have to implement **remove** on an iterator, but it's acceptable to refuse to remove anything: just throw an **UnsupportedOperationException**.

29. Now, you can implement **iterator** on the outer class, to return a new instance of your inner class, passing your **sources** list.

30. Now you'll refactor the client class to use your new sorting adapter. In the **main** method of **MergeInventories.java**, instead of declaring a list of strings as **fullInventory**, declare a list of iterators. Technically you'll have to use a wildcard that extends **String** – even though **String** is final! – to satisfy the type requirements of the **ListSorter**. Set this to a new **ArrayList<>**.

31. In the loop over the array of inventory filenames, instead of calling **collect** on each line of each file and passing that to a call to **fullInventory.addAll** ... call **iterator**, and pass the resulting iterator to **iterators.add**. Like this:

```
List<Iterator<? extends String>> iterators = new ArrayList<> ();
for (int f = 0; f < FILENAMES.length; ++f)
    iterators.add
        (Files.lines (folder.resolve (FILENAMES[f])).iterator ());
```

    So, instead of trying to flatten all the text lines of all the files into a list, you're now just building a list of iterators – each of which stands ready to read those lines from those files, but only once they're asked to pull the data. This is a big thing, because the old implementation forced all of the file data into memory, at once, before trying to process it in any way. Your new implementation will wait until it's time to do something with the information, and that will save a lot of memory – while your improved sorting algorithm will save a lot of time.

Merging Large Inventories LAB 4B

32. Now create a new **ListSorter<String>** called **sorter**, passing **iterators** to the constructor.
33. Now, you can pass **sorter** instead of **fullInventory** when you call **signer.sign**. The hope is that, between preserving the deferred-processing approach of the stream of text lines coming from the files, and a better sorting approach, you'll get a more efficient process. Let's see how you do ...
34. Test again, running a fresh **GenerateInventories** and then **MergeInventories** each time, with values starting at one million and increasing from there.

    You should see a marked improvement in performance – here are the metrics recorded at the time of writing:

```
Processed 1000000 records in 0.641 seconds.
Processed 2000000 records in 1.061 seconds.
Processed 4000000 records in 1.823 seconds.
Processed 8000000 records in 3.263 seconds.
```

    The final answer in **Inventory_Step2** has one refinement that lets you plug in a custom **Comparator<E>** to sort by different criteria than the "natural order."

A Bank's Account Products

LAB 4C

In this lab you will refactor and then enhance an application that models bank accounts. The classic bank-account examples are inheritance-based: CheckingAccount as subclass of Account, etc. But, in the real world, banks are constantly rolling out new products, many of which are simply fresh combinations of familiar features: overdraft protection, different transaction or per-statement-cycle fee structures, minimum balances, etc. This is actually an encapsulation that is ripe for the Decorator pattern!

You will begin by refactoring the existing class into a base type, a concrete implementation, and a base decorator type. You will then implement several different decorators, and test them in various combinations. In the process you will see both the power of Decorator and some of its limitations.

Lab project: **Bank_Step1**

Answer project(s): **Bank_Step2** (intermediate)
Bank_Step3 (final)

Files: * to be created
src/cc/bank/Account.java
src/cc/bank/Test.java
src/cc/bank/ConcreteAccount.java *
src/cc/bank/AccountDecorator.java *
Various decorator subtypes *

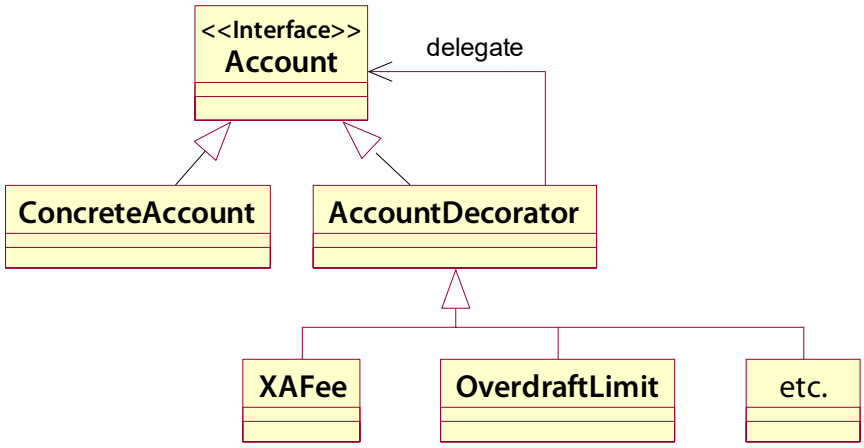
Instructions:

1. Review **Account.java**, which is pretty much the usual example, with a balance and deposit/withdraw methods, and then also the ability to trigger monthly maintenance and whatever that might entail.
2. Test to see how **Test.java** drives an instance of **Account**, calling each operation once and reporting the outcome. (The attempt to overdraft targets a final balance of negative-$50, so the withdrawal amount will vary by circumstance.)

```
Creating new Account with balance of $90.00
After deposit, balance is $140.00
After withdrawal, balance is $90.00
After maintenance, balance is $90.00
Can't overdraft the account: 140.0 > 90.0
After overdraft, balance is $90.00
```

A Bank's Account Products LAB 4C

Now let's reorganize this code to facilitate decoration instead of inheritance for new account types. Instead of the usual inheritance tree, you'll be developing a system of classes that looks like this:



3. First, sink **Account** into an implementation class **ConcreteAccount**, and convert **Account** itself to an interface.
4. Change the test code to instantiate and test a **ConcreteAccount**, and build and regression-test your code.
5. Create a second implementation of the interface, called **AccountDecorator**. This is the base implementation for all "concrete decorators" to come. Give it a field **delegate** which is a reference to any other **Account**.
6. Define a constructor that forces the client code to create a decorator by providing a reference to its delegate account, and store that reference in the **delegate** field.
7. Now implement all the interface methods as pass-throughs to the delegate: that is, call the same method on the delegate reference, passing any arguments and carrying any return value back to the caller.
8. Create a new class **cc.bank.XAFee** that subclasses **AccountDecorator**. The constructor should take a **delegate** reference, which it passes to the superclass constructor, and double-precision values for a **minimum** balance and a **fee** amount, which it will store as fields on the class.
9. Define a new method on this class **imposeTransactionFee**, which takes no parameters and returns nothing. Implement the method to deduct the **fee**, but only if the current balance in the account is below the **minimum**.

A Bank's Account Products LAB 4C

10. Override **deposit** and **withdraw** methods to call **imposeTransactionFee** before or after delegating the actual deposit or withdrawal. And notice that this provides an example of the subtle differences we can get by choosing when we delegate! The bank will do better over time if it (a) tries to impose fees before deposits but (b) after withdrawals, because it maximizes the chances that the minimum-balance requirement will not be met.
11. Add code to **Test.java** to create a second account object. This one should be an **XAFee** instance, with a **ConcreteAccount** instance as its delegate. For minimum balance and fee amount, use $100 and $1.50 – not realistic values but they will give more interesting test results than real-world values would, given the test logic.
12. Test and you should now see:

```
Creating new Account with balance of $90.00
After deposit, balance is $140.00
After withdrawal, balance is $90.00
After maintenance, balance is $90.00
Can't overdraft the account: 140.0 > 90.0
After overdraft, balance is $90.00

Creating new no-overdraft account with balance of $90.00
After deposit, balance is $138.50
After withdrawal, balance is $87.00
After maintenance, balance is $87.00
Can't overdraft the account: 137.0 > 87.0
After overdraft, balance is $87.00
```

One possible solution would be to loosen the encapsulation: provide a direct **setBalance** method and let the caller arbitrarily reset the balance at any time. Or maybe do this and make the method only package-visible?

Our approach will be to remove the overdraft prohibition to a new decorator class, and then let it vary by a new parameter which is the overdraft limit. So to get what we had before we'll have to decorate the core account type with an **OverdraftLimit** with the limit set to zero. Then we will also be able to set non-zero limits, which was the original goal. So ...

13. Create a new decorator subclass **OverdraftLimit**, and give it a constructor that takes a **delegate** reference and a **limit** value.
14. Override **withdraw** and move the logic for testing the withdrawal amount from **ConcreteAccount** to this class. And now instead of testing against the balance, test that (a) the balance is not already negative – can't overdraw twice – and (b) the proposed withdrawal wouldn't take us past the overdraft **limit**, whatever it is.
15. Add code to **Test.java** to build a chain with the **OverdraftLimit** delegating to the **XAFee** delegating to the **ConcreteAccount**. Set an overdraft limit of zero, to establish that overdraft is prohibited.

A Bank's Account Products LAB 4C

16. Test, and notice two things. First, your existing accounts now allow arbitrary overdraft! which you will probably want to tweak for any real account products. But, your newest account product refuses overdrafts, and if you try various values for the overdraft limit you'll see this is now adjustable.

```
Creating new ConcreteAccount with balance of $90.00
After deposit, balance is $140.00
After withdrawal, balance is $90.00
After maintenance, balance is $90.00
After overdraft, balance is $-50.00

Creating new transaction fee account with balance of $90.00
After deposit, balance is $138.50
After withdrawal, balance is $87.00
After maintenance, balance is $87.00
After overdraft, balance is $-51.50

Creating new no-overdraft account with balance of $90.00
After deposit, balance is $138.50
After withdrawal, balance is $87.00
After maintenance, balance is $87.00
Can't overdraft the account.
After overdraft, balance is $87.00
```

This is the intermediate answer code in **Step2**.

17. From here, you can implement and test other product features as required by the bank:

- A savings plan by which a certain amount will be transferred to a separate savings account each time a deposit is made -- as long as the deposit is big enough to support the savings deduction. (You don't need to model the separate account; just treat this as a deduction, and maybe write a line to the console saying we're saving $X.XX.)
- A flat monthly fee.
- Monthly interest at a given rate.
- A monthly savings transfer.

We leave this part of the lab largely to your experimentation: see on one hand how flexible the decorator approach makes the bank's product lineup – how easy it is to roll out a new product by re-combining existing feature implementations. And on the other hand you've observed some limitations, and you may find others as you explore the possibilities above.

One possible set of implementations is found as the final answer code in **Step3**.

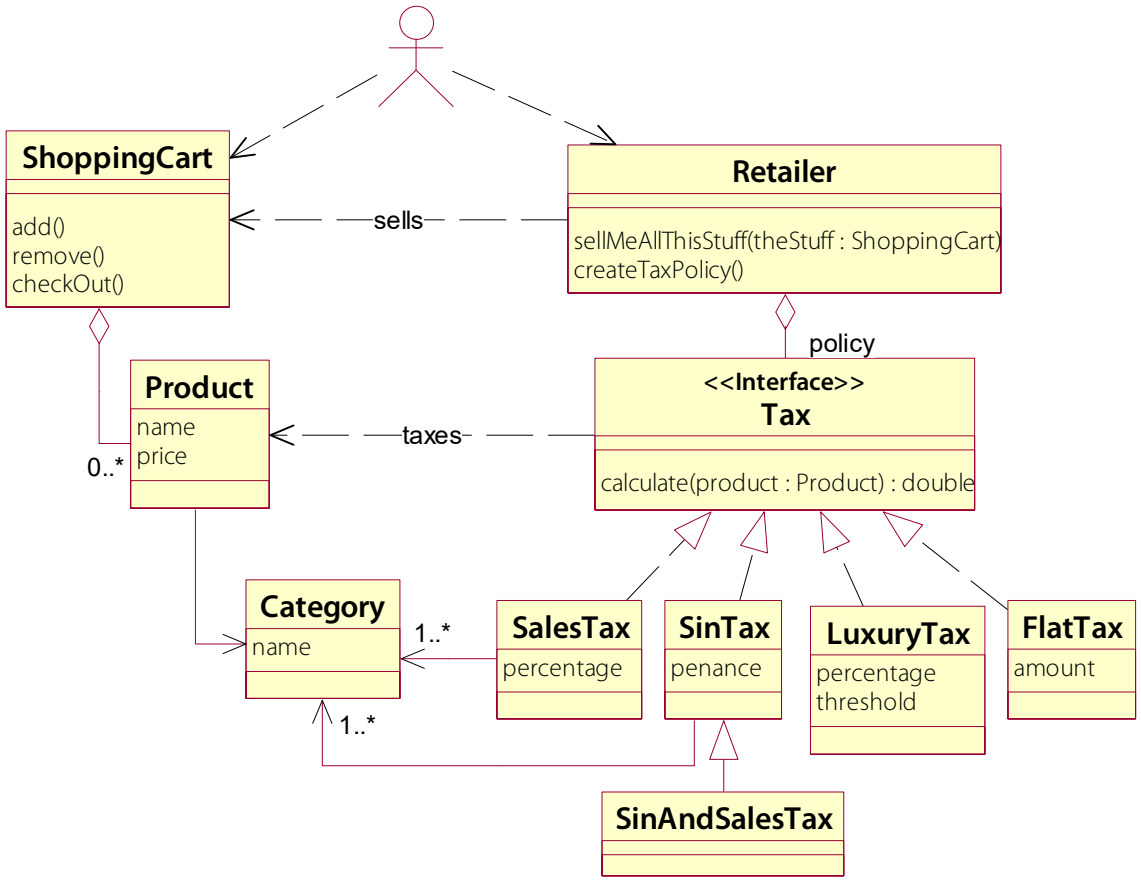
Structural Refactoring

LAB 4D

In this exercise you will analyze a single "before picture" 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.

Current Design

Consider the design for a web retailer, as diagrammed on the following page. The site offers a body of **Product**s, each of which is modeled very simply as having a name, a price, and a **Category**. The application knows of several categories, which may be hard-coded or perhaps loaded from a database at startup – things like "Food", "Clothing", "Alcohol", "Entertainment", and so on. The **ShoppingCart** is a transient object that can be filled with **Product**s, and when the user is ready he or she can "check out" by asking the **Retailer** to **sell[Them]AllThisStuff**.



Structural Refactoring LAB 4D

The **Retailer** (a Singleton, by the way), calculates the total price of the items, including the application of taxes. It keeps one or more instances of **Tax** implementations as helpers for this purpose, via another behavioral pattern called a Strategy: **Tax.calculate** will return the appropriate tax for a given **Product**. **Tax** implementations apply different procedures for this purpose: **SalesTax** is a fixed percentage rate; **LuxuryTax** is based on a price threshold while **SinTax** is based on the product **Category**; etc. **Retailer** uses a Factory Method to instantiate the appropriate implementation class for a particular sales locale.

But all is not well with this design! There are at least two possible enhancements, each based on one of the patterns we've studied in this chapter. Analyze the design and recommend refactorings. Use any notation you like, from pseudo-code to block-and-arrow diagrams with text notes to proper UML.

Structural Refactoring LAB 4D

Analysis and Improved Design

There are a two primary patterns we might put into play here. The simpler of these is the Flyweight pattern, which applies nicely to the **Category** class. Did you wonder why **Category** was a class at all? In the previous design, there's not much reason it couldn't be a string. But we note that **Category** can have only a certain finite number of instances; yet it can be used by many more **Product**s than there are possible values for the category name. This indicates the need for a Flyweight solution that controls the total number of **Category** instances. If the values are to be hard-coded, a Java-5.0 enum would be most appropriate, and easiest. For Java 1.x, or if the values are to be loaded from a file or database at startup, a full expression of the Flyweight pattern would have to be written out by hand, including a static initializer block to load the values.

The bigger change comes about when we recognize a problem with the **Tax** inheritance hierarchy. These different **Tax** policies are to be combined, and the total tax policy will be different for different deployments of the application – hence the use of a Factory pattern, as mentioned in the starter write-up. But the inheritance-based approach shown in the starter design is inflexible. The most glaring offender is the **SinAndSalesTax** extending **SinTax**. Why use inheritance to mix and match features when a delegation-based approach would be completely flexible?

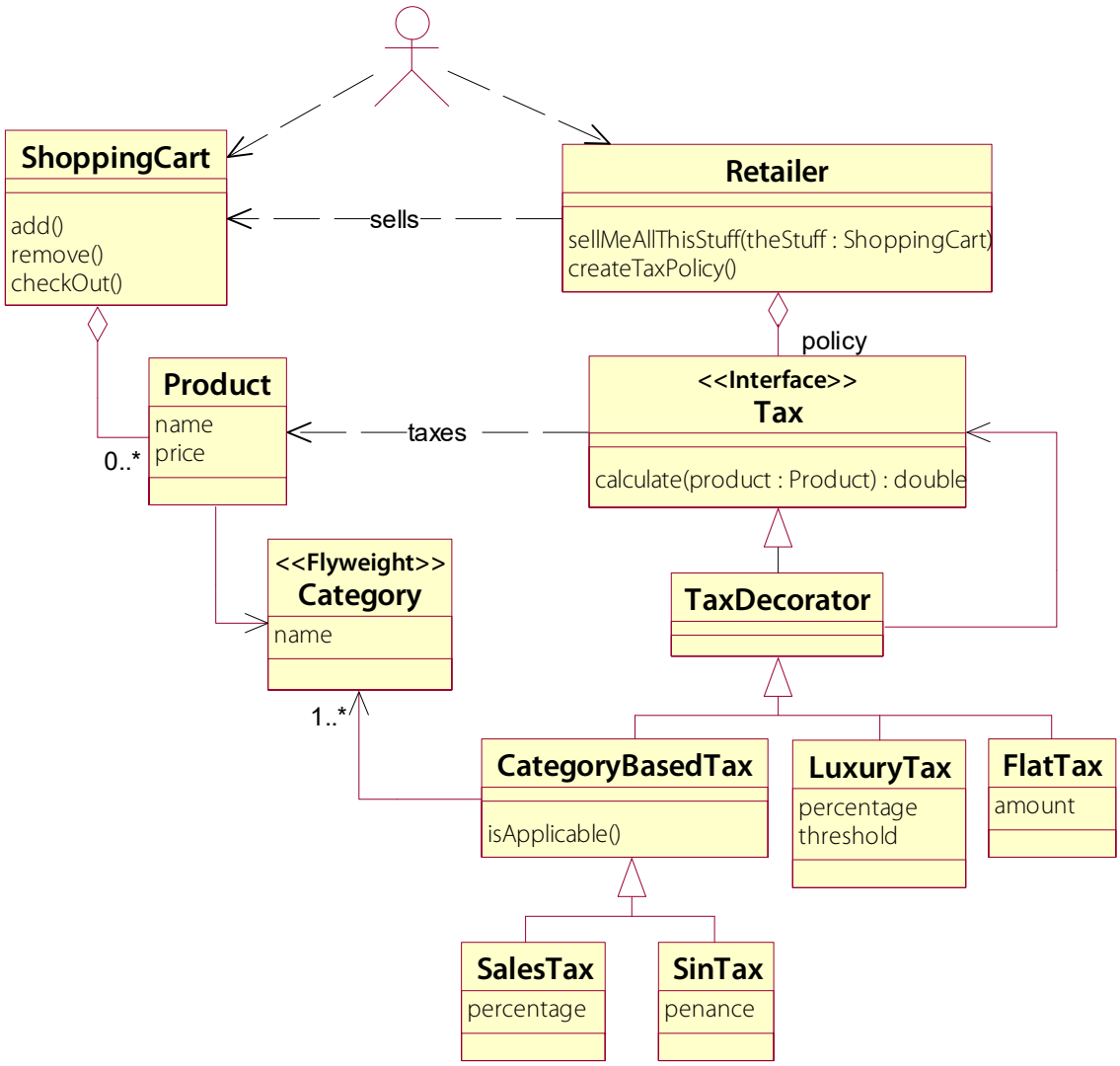
The strongest design here will apply the Decorator pattern to allow any tax implementation to delegate to others, so that a chain of policies can be aggregated together. The Factory Method **createTaxPolicy**, already found in the **Retailer**, can be responsible for assembling the chain, and then it can be used through the **Tax** interface with no further need for underlying type information or chain structure.

The **CategoryBasedTax** encapsulation is not specific to the Decorator expression, but does show that within this pattern, other more ordinary criteria can be applied to arrive at a clean classification system. Both sales and sin taxes will refer to some set of **Category** objects and will test a **Product** for inclusion in that **Category**, so this seems worth capturing in one base class under **TaxDecorator**.

Finally, note that this Decorator system has no "concrete components;" that is, all implementations are found as extensions of the decorator. Contrast Java Streams, which has many concrete implementations such as **FileOutputStream**; these can only exist at the end of the chain, while all these **Tax** objects can exist anywhere.

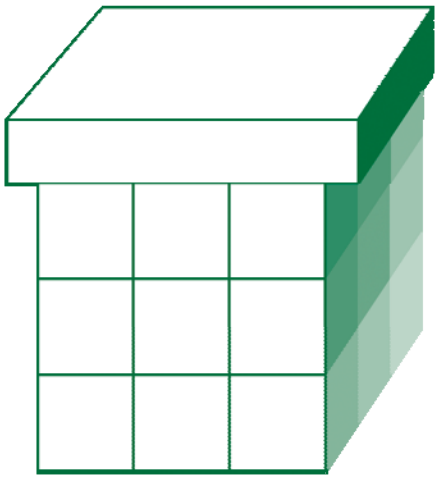
Structural Refactoring LAB 4D

The refactored system might look like this:



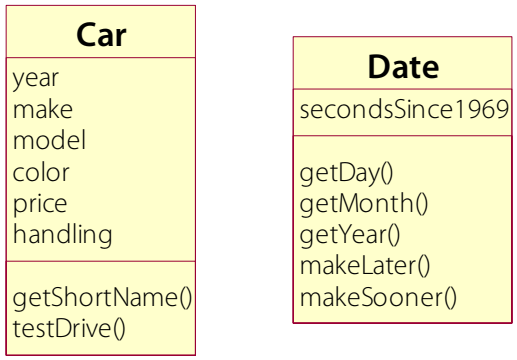
APPENDIX A

UML QUICK REFERENCE



UML Class/Interface Diagrams

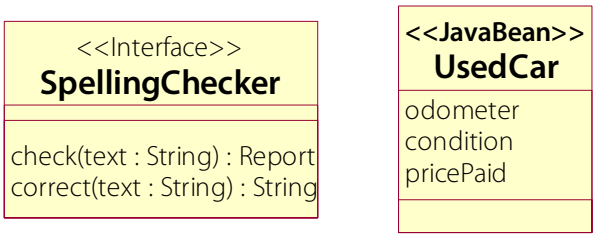
- **This appendix presents a summary of the Unified Modeling Language, or UML.**
  - Specifically, we will consider the most heavily used UML diagram type, the **class diagram**.
- **UML represents classes as shown below.**



  - The main rectangle comprises three **compartments**, for the class' **name**, **attributes**, and **operations**.
  - That is, the name, the state, and the behavior.
  - In Java, UML attributes are **fields**; operations are **methods**.
- **UML can express visibility of attributes and operations, and many other decorations are possible – static, transient, synchronized, etc.**
- **Though these concepts are important, for simplicity we'll mostly leave visibility notes out of our diagrams.**
  - The general rule will be that attributes are private and operations are public.

Stereotypes

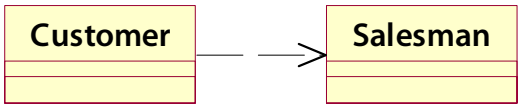
- **UML can model many OO and pseudo-OO systems, and the basic notation for a class might apply to:**
  - An actual **class**
  - An **interface**
  - A class with certain assumptions about it's makeup or usefulness, such as a **JavaBean**
  - A **struct**, whether implemented as a class or not
  - An **XML Schema complex type**
- **To denote the actual meaning of a class, use the device known as a UML stereotype:**



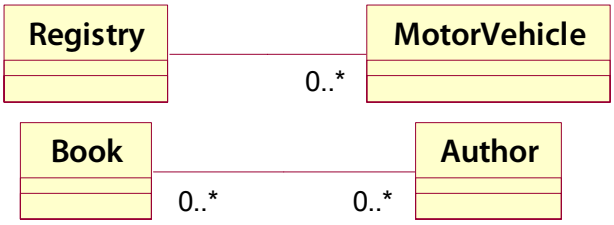
- **The stereotypes imply certain things about the classes to which they're applied:**
  - An <<Interface>> will have no state and only abstract methods – this is standard to UML.
  - A <<JavaBean>> stereotype might be used to imply that all attributes are actually JavaBeans **properties**, with accessor and mutator (get and set) methods.

Relationships

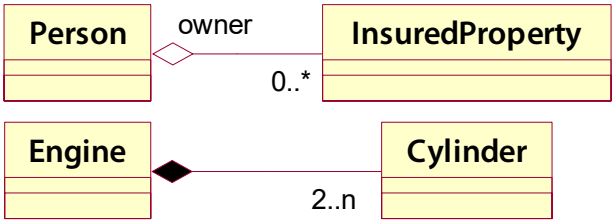
- **In order to fully describe a solution to a software problem, we must be able to show classes and the relationships between them.**
- **UML recognizes a few basic relationships, each of which suggests something about the interaction between classes as implemented and at runtime.**
  - If objects of one class use objects of another, the classes are related by **dependency** (a.k.a. **collaboration**), implying that one needs to know the semantics of the other:



  - A class that relates to another by a clear cardinal relationship – one-to-one, one-to-many, etc. – relates to that other class by **association**. Cardinality and **role** names can be attached to the simple solid line that connects associated classes.

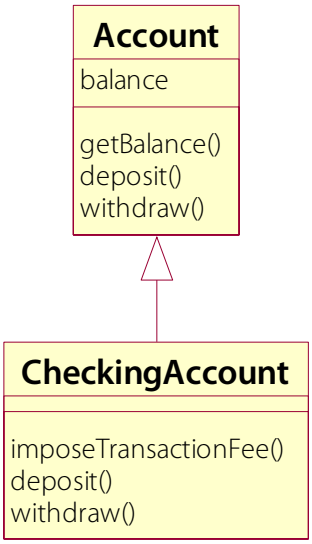


  - A more specific sort of association is the **aggregation**, which implies that one concept is really defined in terms of another. Some aggregations are also logical **compositions**.

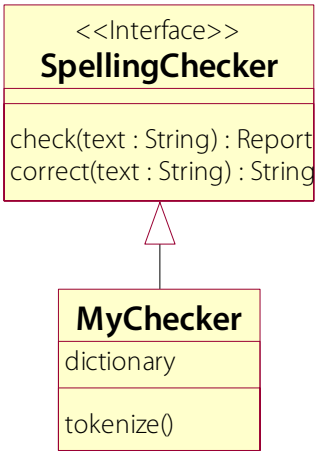


Specialization and Realization

- UML identifies the relationship between a class and its base class as **specialization**, and shows it as:



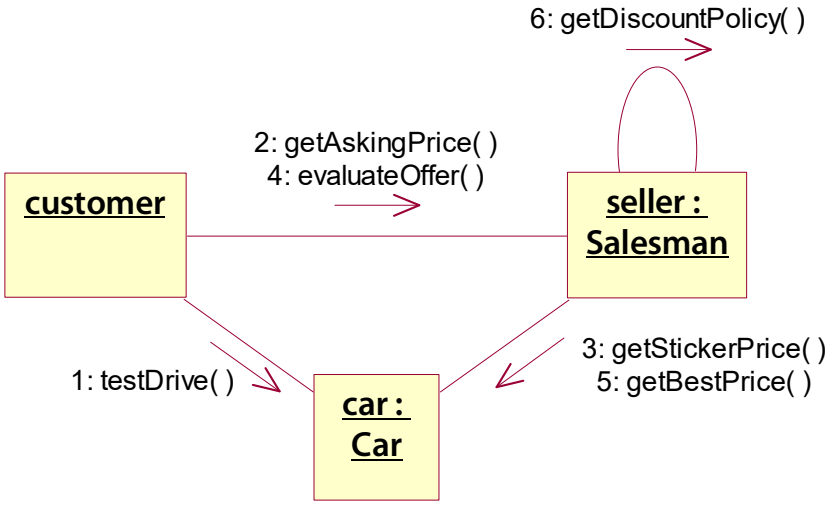
- As Java distinguishes between a class that **extends** another class and a class that **implements** an interface, UML distinguishes between specialization and **realization**:



  - UML interfaces can specialize other interfaces, just as Java interfaces **extend** other interfaces.

UML Interaction Diagrams

- To illustrate interactions between specific objects at runtime, UML includes two styles of **interaction diagram**:
- The **collaboration diagram** shows objects, their relationships, and messaging between them.
  - It includes sequence numbering to show how one message or method call triggers another.
- Here is a scenario in which a customer test-drives a car and then negotiates with a salesman to buy the car:



UML Interaction Diagrams

- The **sequence diagram** highlights interactions over time.
  - Time is represented by the vertical dimension of the diagram, and all object interactions are shown horizontally.
  - Sequence diagrams also clarify when a caller will block while the called object is carrying out its own calls.
- **Here's the same car-buying scenario, captured in a sequence diagram:**