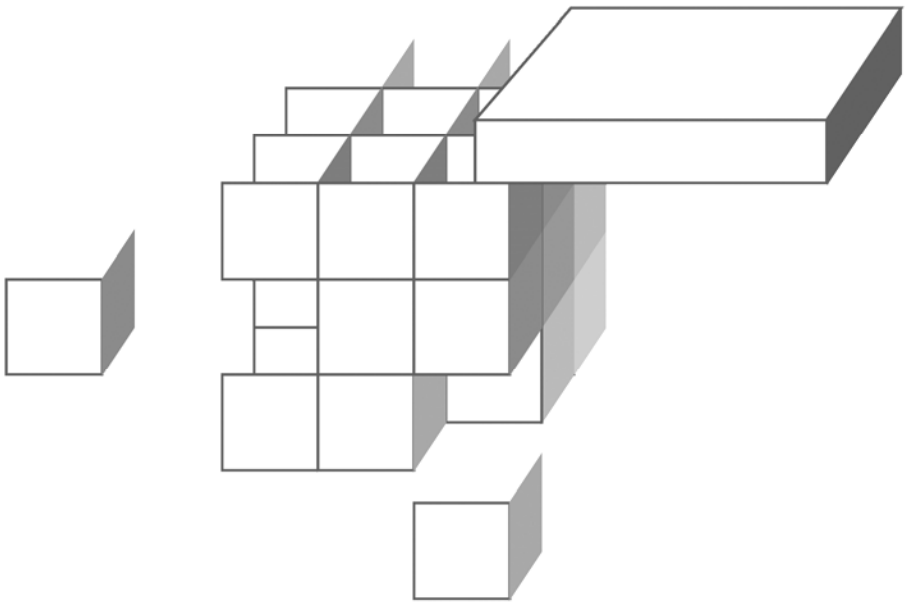
# CHAPTER 4
# STRUCTURAL PATTERNS



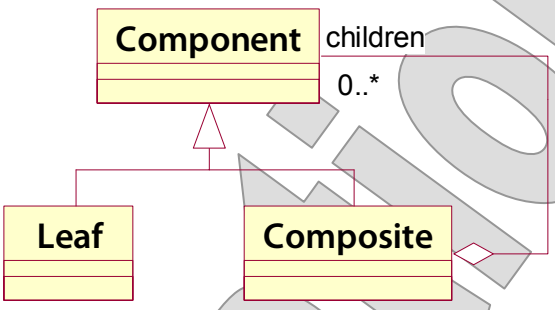
## OBJECTIVES

*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

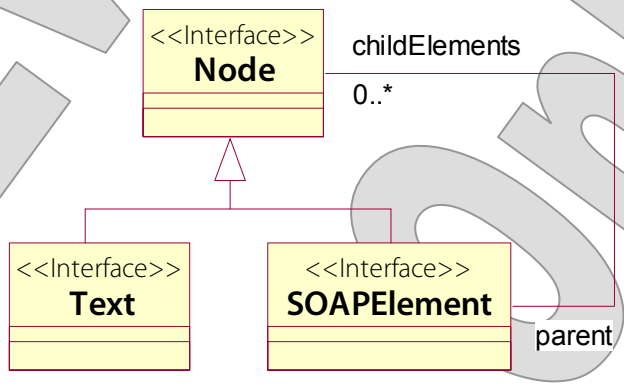
- **The Composite pattern defines a means of creating composite objects or parent-child hierarchies of arbitrary depth.**




  - **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

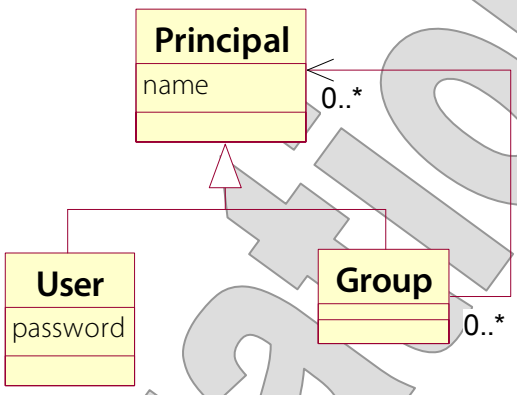
- **Examples of Composite include:**
  - **File/folder systems** – here the parent is associated with the child, but not really composed of its children
  - **JFC and JavaFX window hierarchies** – here we have a clearer concept of true composition – a part/whole relationship
- **XML offers some interesting 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:

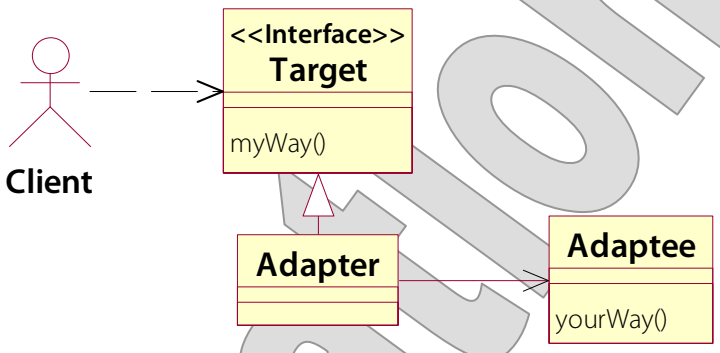
```
Testing Administrators ... authorized.
Testing gameshowhost ... found, but not authorized.
Testing wprovost ... authorized.
```

# The Adapter Pattern

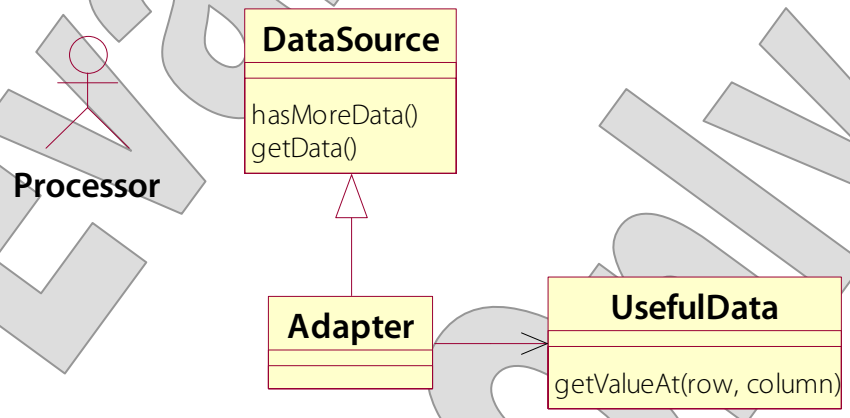
- **The Adapter pattern addresses a very general problem: getting two unlike objects to work together.**
- **Adapter is about getting one object to be able to use another.**
  - We assume that we can't or **shouldn't modify** either of the things, but instead must find a way to get them talking to each 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."



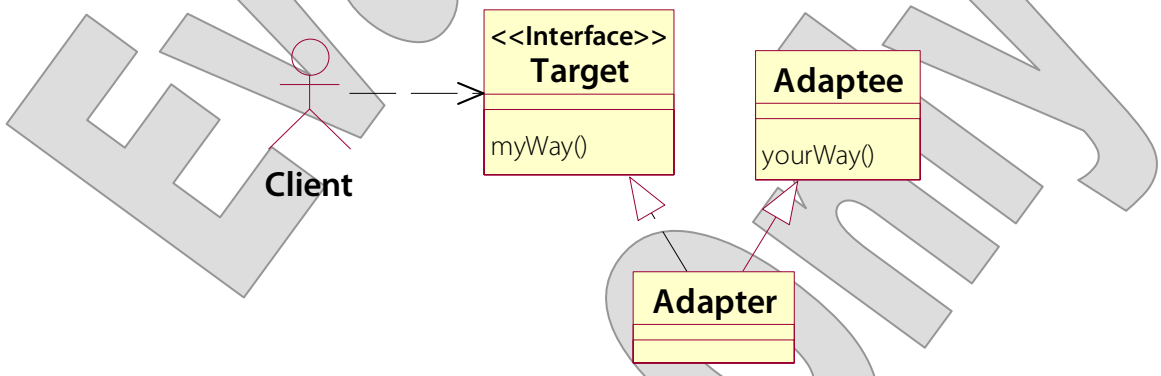

  - Calls on the previously-understood **Target** interface or class will be handled in a way that brings the **Adaptee** into play by private delegation.
  - Our **DataSource**-vs.-**UsefulData** problem is resolved with an adapter, as in:




  - 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:**

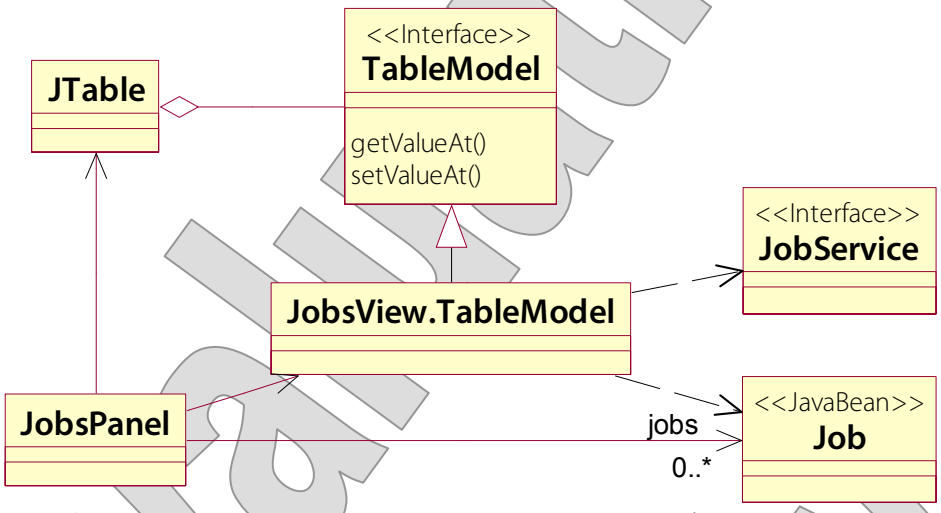



  - In C++ this is generally workable, because we have **multiple implementation inheritance** in that language.
  - In Java, the strategy is limited to situations in which the Target is an **interface**; but fortunately that is a fairly common case.

# Adapting the Jobs Table

EXAMPLE

- Consider **HR_Step6**, and specifically the way in which the **JobsView** uses a **JTable**.
  - **JTable** relies on a **TableModel** interface as an abstraction of data management
  - **JobsView** implements this interface in an inner class that extends the helper implementation **AbstractTableModel**:




- **The alternative – and really a warning sign – would have been for JobsView to use the default table model and to populate it with jobs information manually.**
  - This approach would be surprisingly similar to the cloned-list solution we just discussed, with the same adverse effects.
  - Failure to "Adapt" often springs from basic unfamiliarity with the API one is using: this weaker strategy of cloning data into a default table model is surprisingly common, and usually the sign of a lack of deep experience with JFC.

## Adapting the Jobs Table

EXAMPLE

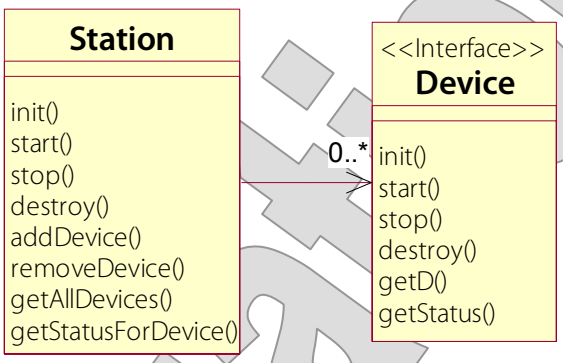
– See **src/cc/hr/gui/JobsView.java**.

```
  table = new JTable (new TableModel ());
  // ... as in our inner-class TableModel

protected class TableModel
  extends AbstractTableModel
{
  public TableModel ()
  {
    ...
      jobs = service.getAllJobs ();
    ...
  }

  public  int getRowCount ()
  {
    return jobs.size ();
  }

  public Object getValueAt (int row, int col)
  {
    Job job = jobs.get (row);
    if (col == 0)
      return job.getName ();
    if (col == 1)
      return job.getMinimumSalary ();
    if (col == 2)
      return job.getMaximumSalary ();
    ...
  }

  ...
}
```

# 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:



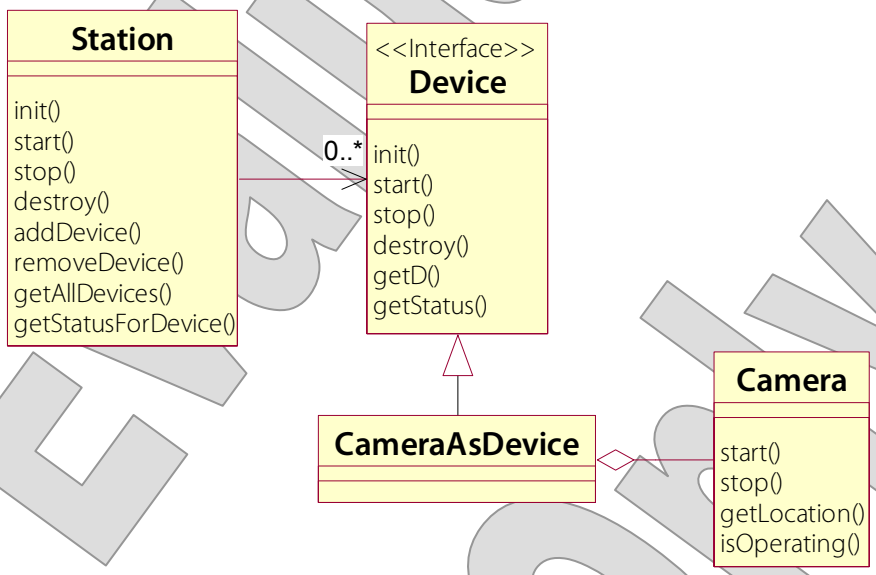

- The **Device** interface is our Target:

```
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 implement **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

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

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

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

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

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

4. 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

5. 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

6. 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)
    );
}
```

7. 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.

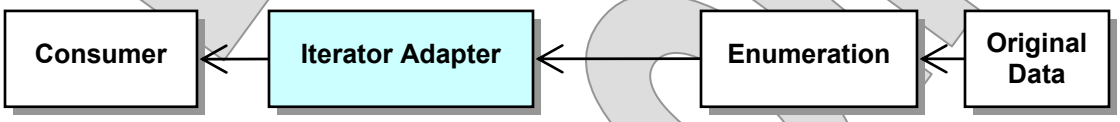
# Keeping It Flowing

- 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**

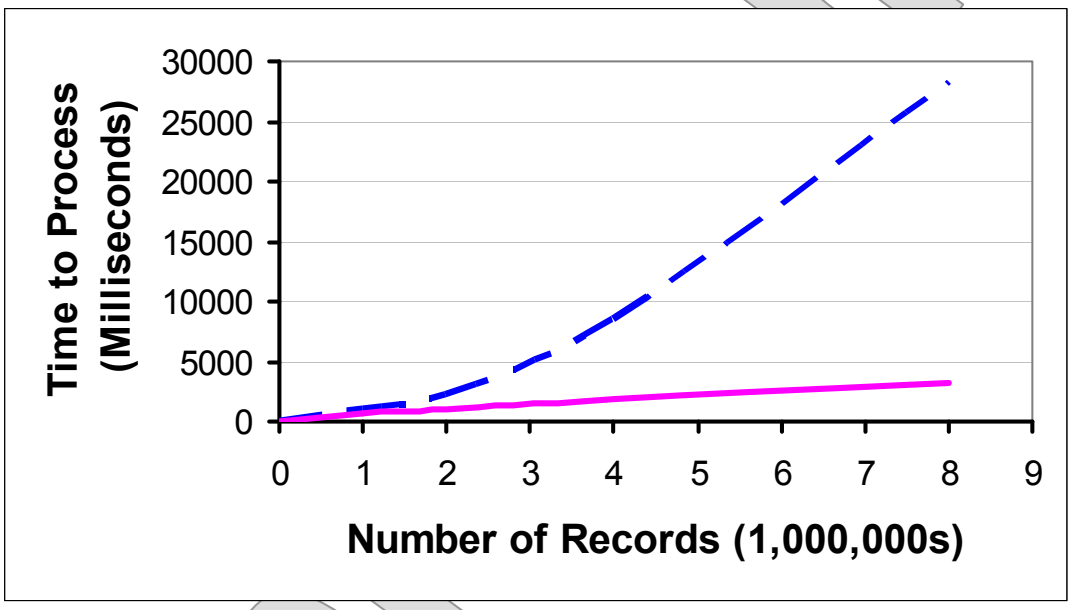
In this lab you will refactor an application that merges existing inventory files from multiple sites into a single inventory file. The "before picture" application works, but it is inefficient and doesn't scale well at all. You'll see the warning sign that the application takes the shortest route to 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

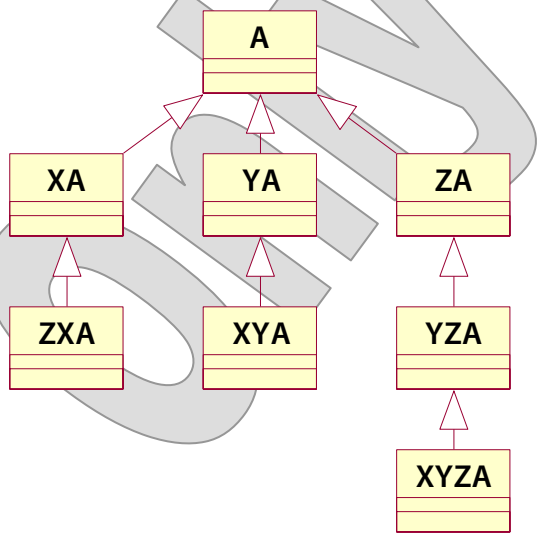
- **Here again are the metrics from the previous lab, in the form of a chart of processing time vs. the size of the merged inventory:**




  - 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**.

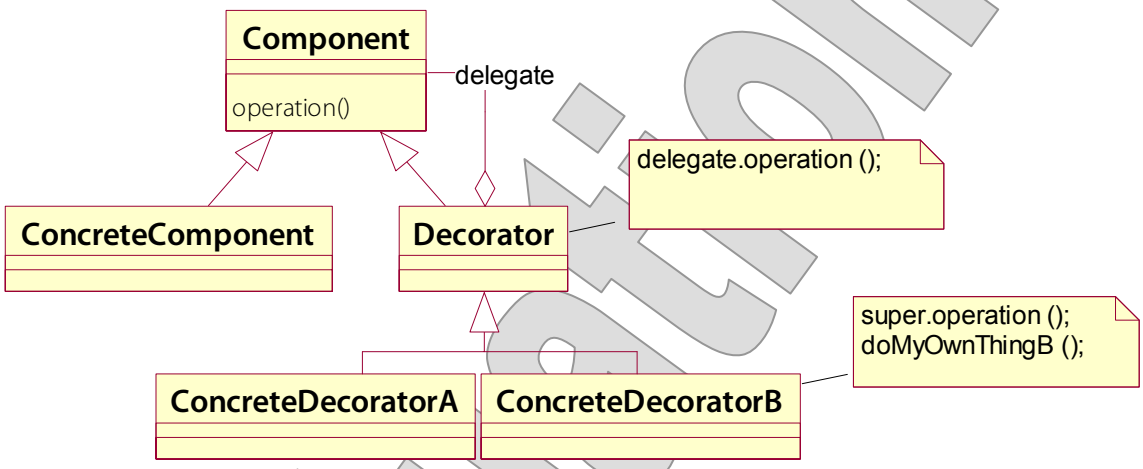
# The Decorator Pattern

- 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.

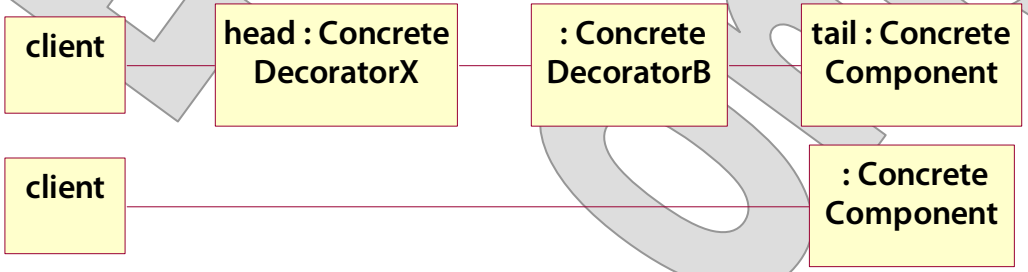


# The Decorator Pattern

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



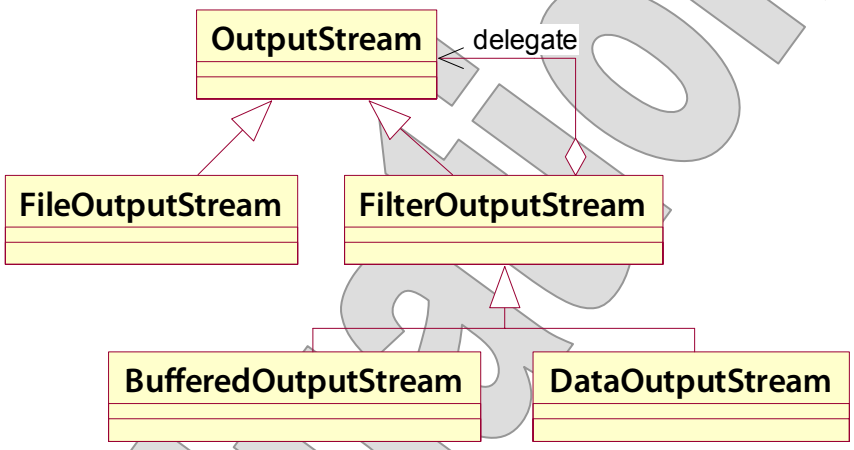

  - There is a single derivation of the base type called the **Decorator**; it adds only the ability to create **chains** of objects, each said to **decorate** the next one in the chain.
  - Subtypes of **Decorator** define the independent decorations, and can then be mixed and matched arbitrarily.
  -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



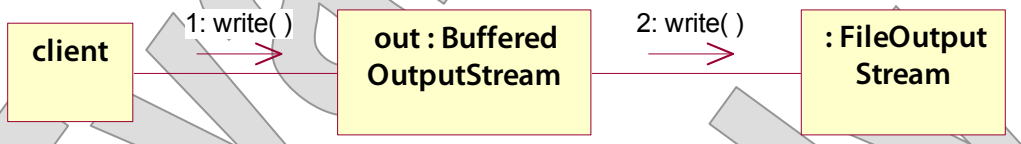

  - This is much more maintainable and flexible than an inheritance-based approach.

# 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.

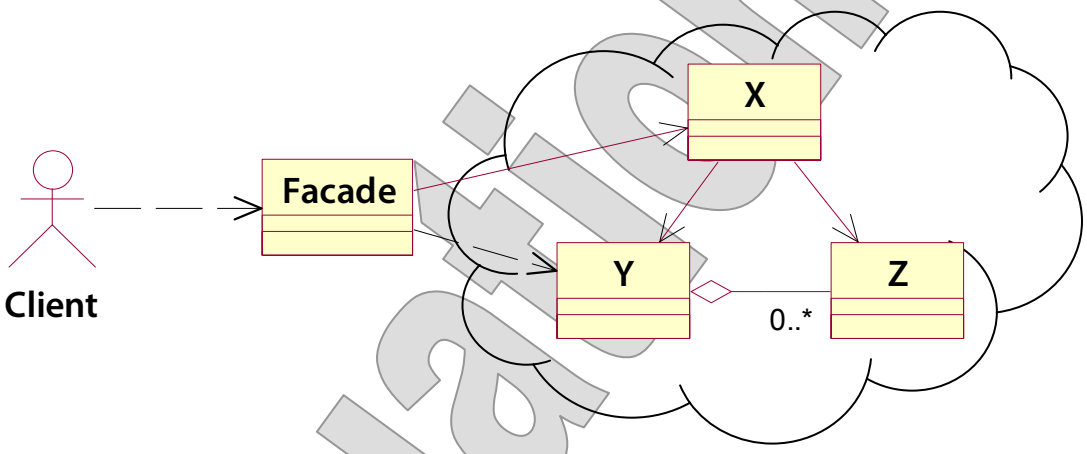
Detailed instructions are found at the end of the chapter.

# 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.

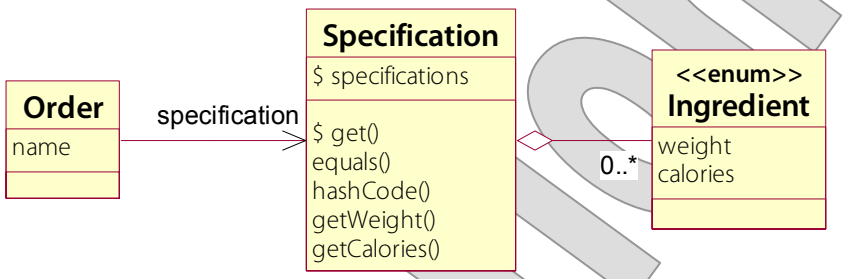
# The Flyweight Pattern

- **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 *

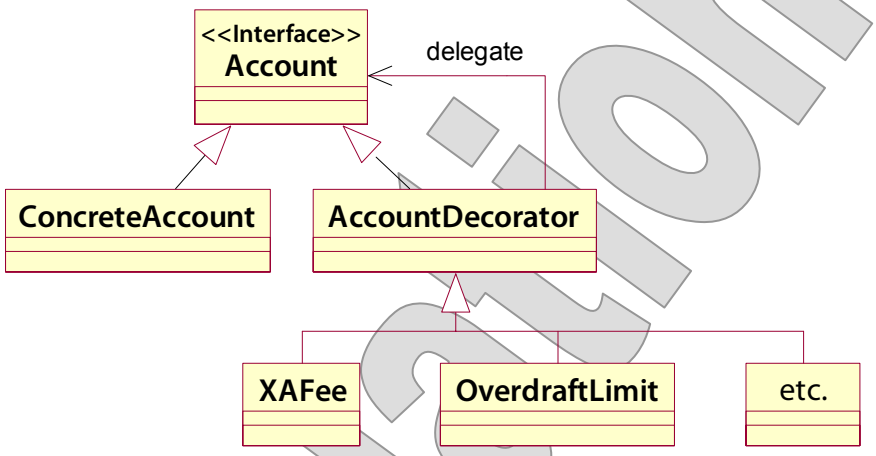
**Instructions:**

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.

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**.

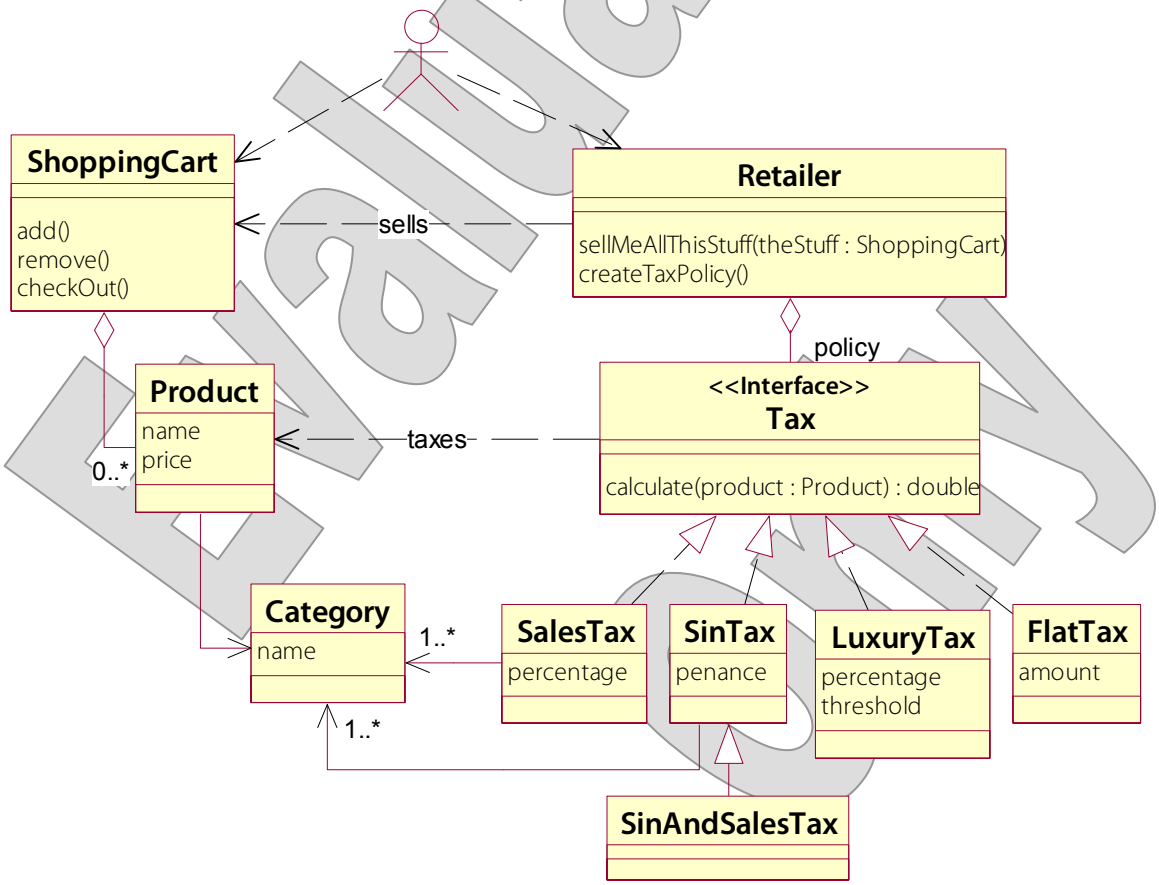
# Structural Refactoring

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: