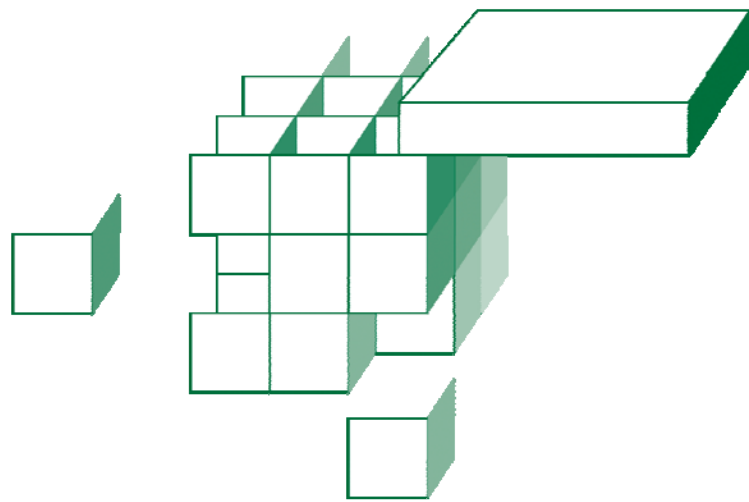


# CHAPTER 4

## STRUCTURAL PATTERNS



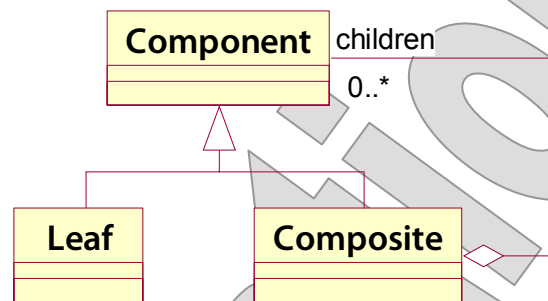
## OBJECTIVES

*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

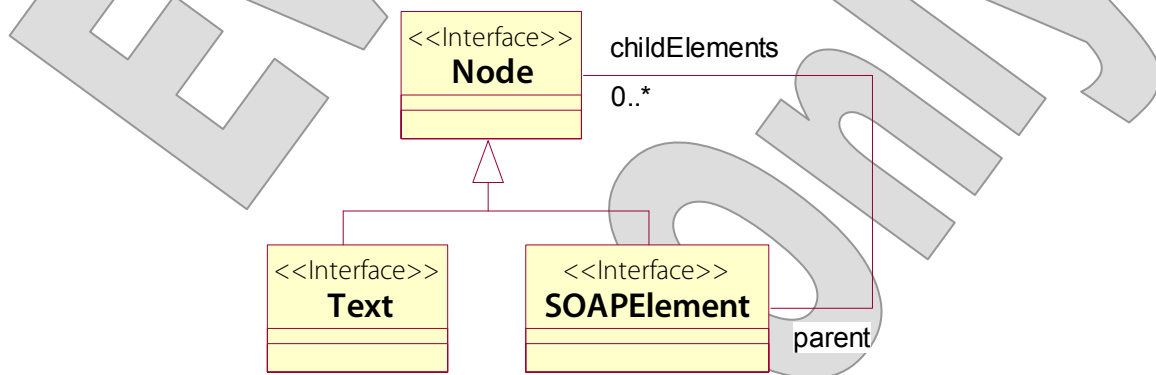
- The **Composite** pattern defines a means of creating composite objects or parent-child hierarchies of arbitrary depth.



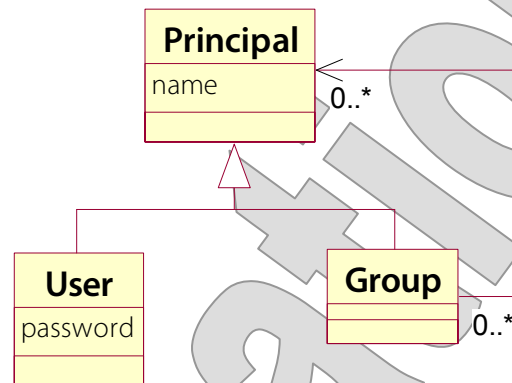
- **Component**, a rather GUI-centric term, might be considered more generally as **Node**.
  - **Composite** is simply a node with children that are of the node type: that is it both extends and collects the node type.
  - **Leaf** is a node with no children. In some expressions of this pattern, **Leaf** 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 **Users** differently from its collection of other **Groups**.
    - **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**.



- 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 **Principals** – which of course could be **User** instances or other **Groups**
- 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.

- 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`.

- 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.

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.");
}
```



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.
```

- 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.

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;
}
```

- 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);
}
```

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.
```

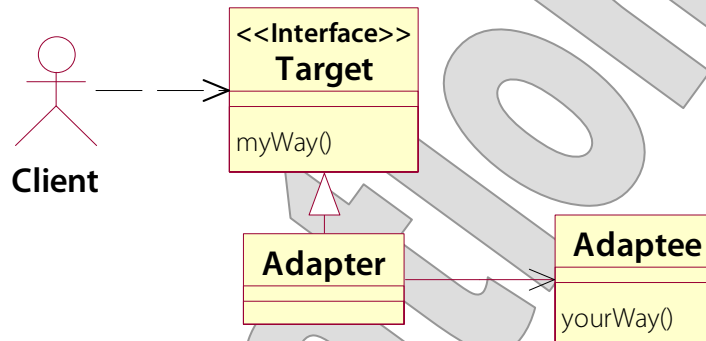
# The Adapter Pattern

---

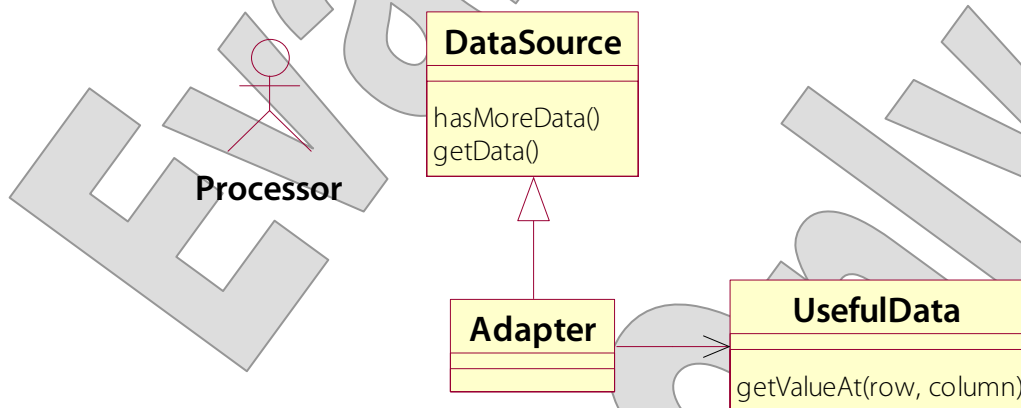
- The **Adapter** pattern addresses a very general problem: getting two unlike objects to work together.
- **Adapter** is about getting one object to be able to use another.
  - We assume that we can't or **shouldn't modify** either of the things, but instead must find a way to get them talking to each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



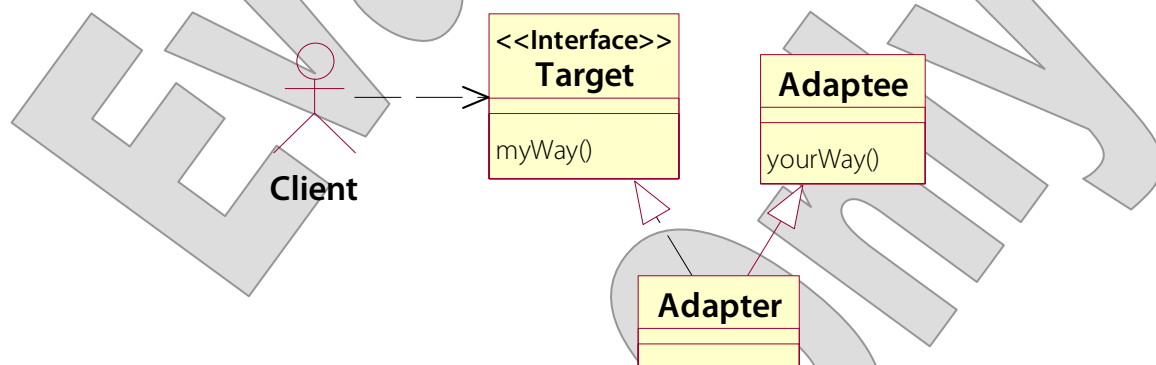
- Calls on the previously-understood **Target** interface or class will be handled in a way that brings the **Adaptee** into play by private delegation.
- Our **DataSource**-vs.-**UsefulData** problem is resolved with an adapter, as in:



- 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

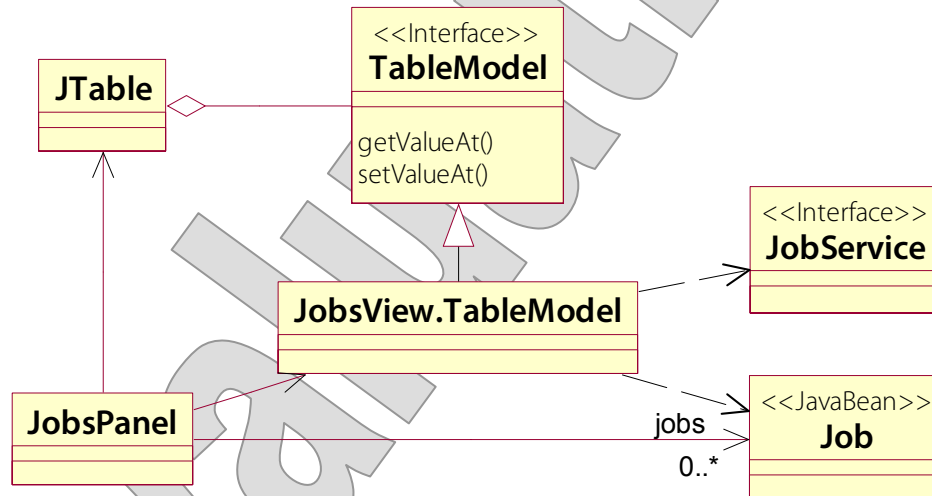
- Adapter examples abound in the Core API:
  - The **java.util.Collections** utility can convert from one collection type to another – **Enumerations** to **Lists**, etc.; can create **unmodifiable** or **thread-safe wrappers** over existing collections; and a bit later we'll talk about adaptations with **Streams**.
  - 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.



- 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.

– See `src/cc/hr/gui/JobsView.java`.

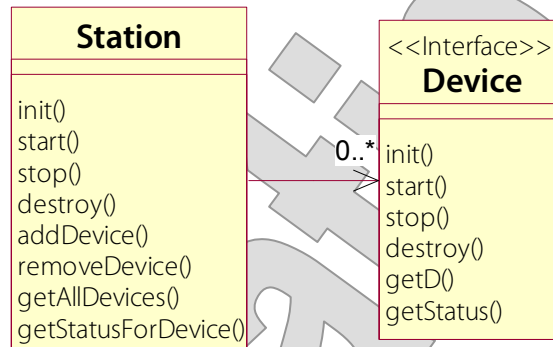
```
table = new JTable (new TableModel ());
// ... as in our inner-class TableModel

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 ();
        ...
    }
    ...
}
```

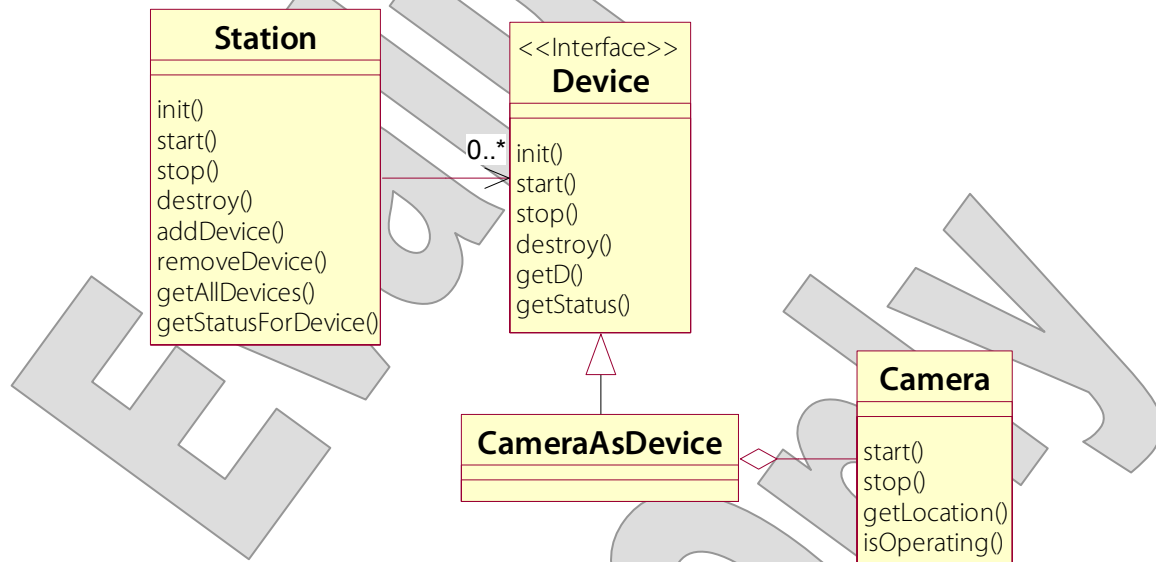
- In **Station\_Step1** is the kernel of a management, monitoring, and control system in which a **Station** can keep track of any number of **Devices**:



- **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 ();
}
```

- 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.

## 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 **Devices** 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
    (
        );
}
```

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;
}
```

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 ()
{
}
```

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**.



**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

---

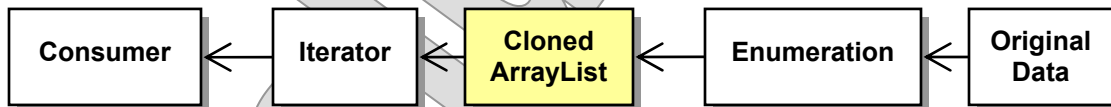
- By this point in the evolution of the Java language, there are many ways in which one might choose to accept large volumes of data in a method, or to return data from a method.
  - Going way back, we have the **Enumeration**.
  - For a while after that the best practice was to use **Iterators** over 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:

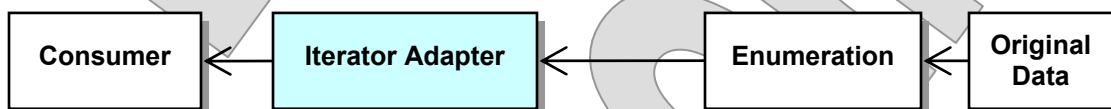
```
List<X> temp = new ArrayList<X> ();  
while (myEnum.hasMoreElements ())  
    temp.add (myEnum.nextElement ());  
processThis (temp.iterator ());
```

- 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 ());
```

**Suggested time: 45 minutes**

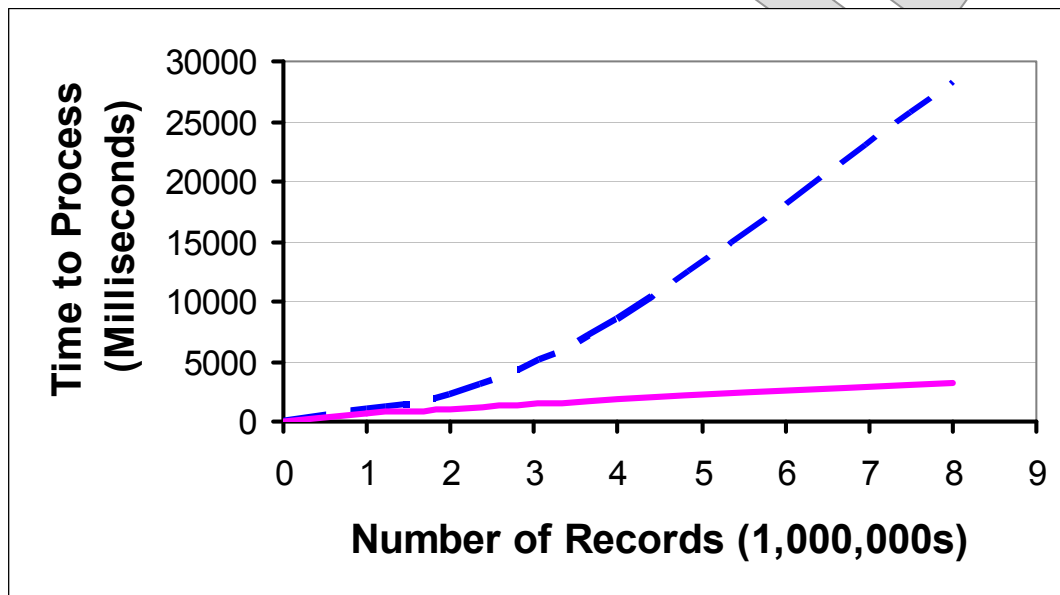
In this lab you will refactor an application that merges existing inventory files from multiple sites into a single inventory file. The “before picture” application works, but it is inefficient and doesn’t scale well at all. You’ll see the warning sign that the application takes the shortest route to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

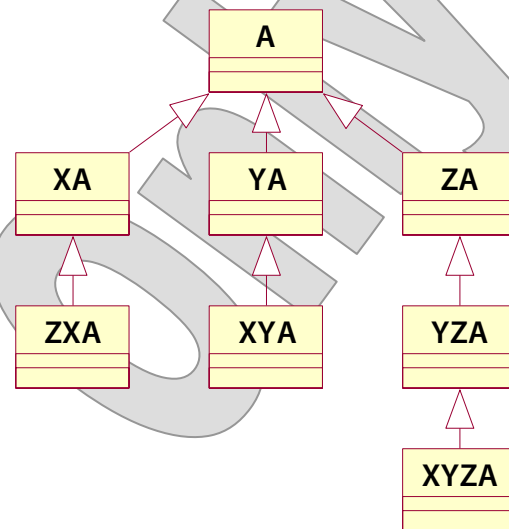


- The dashed line shows the performance when we loaded a list with all the records, and then sorted it in place before passing it along to the signature component.
- The solid line shows the performance of your adapter.
- So, first off, your adapter is much faster, which is great.
- More compelling is 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

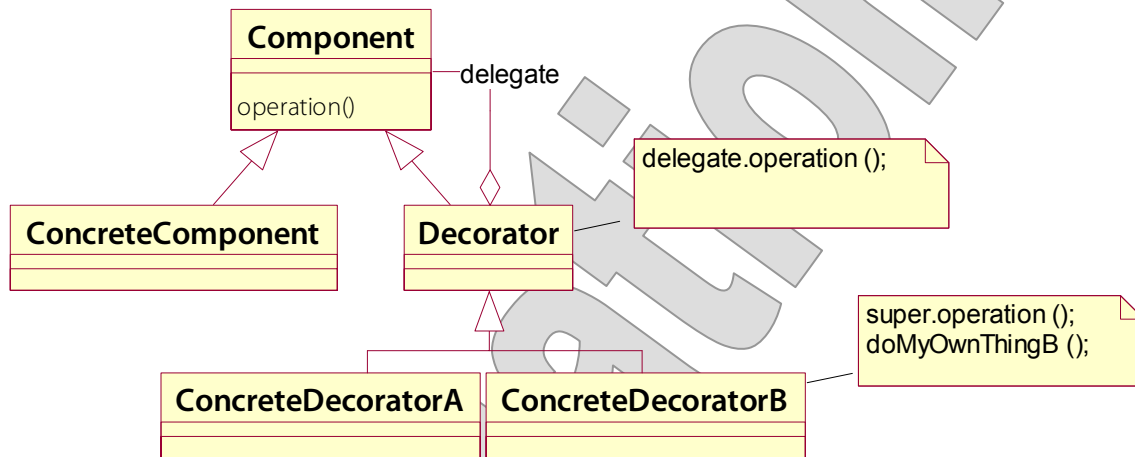
- The **Decorator** pattern offers an alternative to inheritance for providing **progressive specialization**.
- The problem occurs where one or more classes should offer the same behavior, but then each one should also specialize by adding its own behaviors.
- On the surface, this seems like a perfect 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.

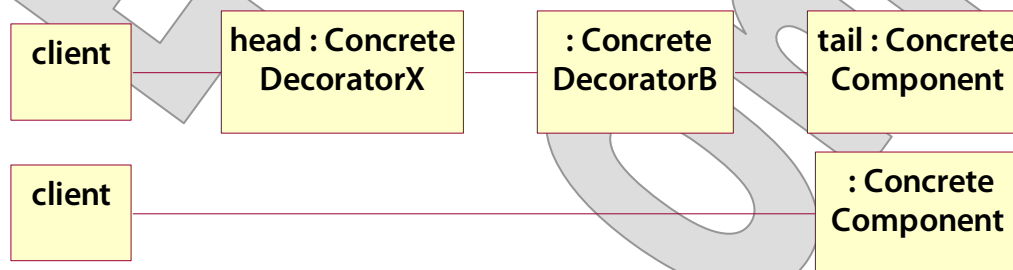


# The Decorator Pattern

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



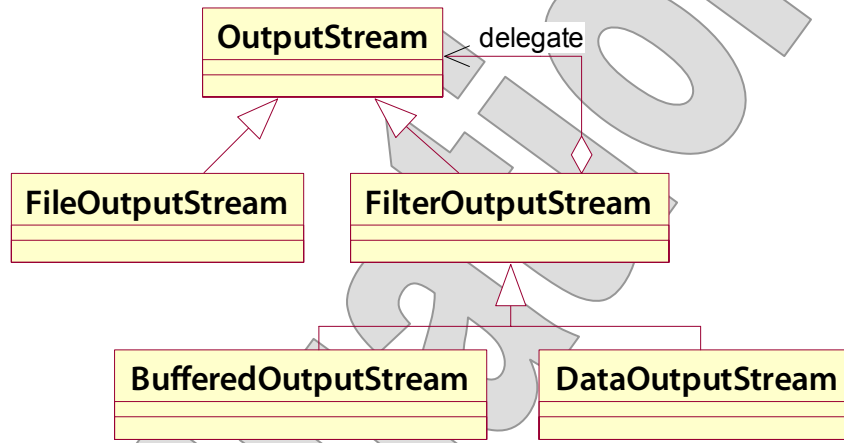
- There is a single derivation of the base type called the **Decorator**; it adds only the ability to create **chains** of objects, each said to **decorate** the next one in the chain.
- Subtypes of **Decorator** define the independent decorations, and can then be mixed and matched arbitrarily.
- The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



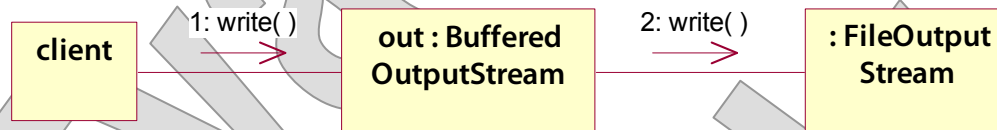
- This is much more maintainable and flexible than an inheritance-based approach.

# 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.”



**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

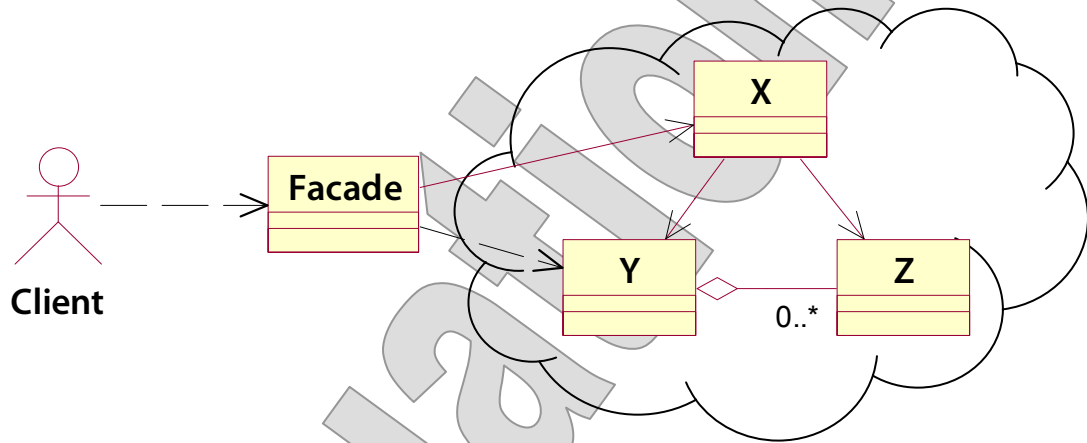
---

- 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.

Evaluation Only

# The Façade Pattern

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



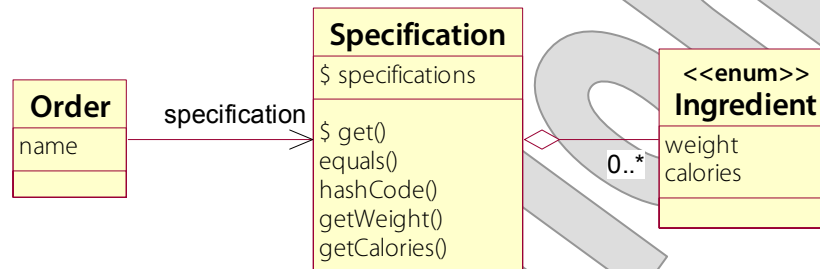
- This class will offer a simplified interface that hides the inner workings while giving the caller good high-level ability to direct the activity of the subsystem.
- There are many examples in the Core API ...
  - The **Java Sockets API** uses a complex set of classes to model URLs, addresses, network 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.

- 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;
    }
}
```

- 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);
        }
    }
}
```

- ... 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.

- 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.



**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**!