Lecture 8 — Software Design Patterns

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

November 14, 2015

Software Design Patterns

Some common re-usable implementation patterns.

Mostly relevant to OOP (but you know OOP).

Many patterns can appear in the same system.

Some conflict, but a large system could have some or all.

Basic Categories

We will examine three different categories:

1. Creation Patterns
2. Structure Patterns
3. Behaviour Patterns

Creation Patterns

Deal with creation of objects.

Hide creation logic as an implementation detail.

All of the creation logic should be kept together and not spread out over multiple classes.

Creation Patterns: Factory

Create objects without exposing instantiation logic.

Create an object without specifying the exact class.

Or: use when setup is too complex for the class constructor.

Creation Patterns: Factory

Use when: creating objects that require a lot of setuconfiguration, or to have flexibility in creating many different kinds of similar classes without having lots of duplicate code.

Creation Patterns: Factory

Example: Creating a new user.

The `User` instantiation requires data not available in the constructor of the `User` class, such as the user's group.

Solution: `UserFactory` that creates a new `User`, populates its data fields as appropriate, and returns the `User`.

Further examples:

```
public Rectangle createRectangle() {
  return new RectangleImpl();
}

public FileExample createFileExample() {
  FileExample file = new FileExample();
  file.setOwner(Environment.getInstance().getOwner());
  file.setCreationDate(CalendarDate.today());
}
```

Creation Patterns: Singleton

Only one instance of a class. Global point of access.

Access controlled through the single point of access.

Use when: only 1 instance of the class may ever exist.

Creation Patterns: Singleton

Example: program has a `SecurityManager`.

There is only one instance, accessed by `SecurityManager.getInstance()`.

To check `Permissions.READ`,
`SecurityManager.getInstance()`
`.checkPermission(Permissions.READ)`.

Code for the Security Manager Example:

```
public class SecurityManager() {
  private SecurityManager instance;

  private SecurityManager() {/* */}

  public SecurityManager getInstance() {
    if (instance == null) {
      instance = new SecurityManager();
    }
    return instance;
  }
}
```

Creation Patterns: Template

An object is used as the basis for making many copies; the copies can then be customized as necessary.

Instead of having many lines of code where data is repeatedly set, define the object once and create copies from that object.

Use when: similar objects will appear often in the system, or to avoid multiple creation routines for the same/similar objects.

Creation Patterns: Template

Example: every month, the company files tax information with the government.

Define a template with some fields filled in (e.g., tax ID number, name, address).

When filing the month's tax information, make a copy of the template, and fill it with the tax transactions for this month.

Creation Patterns: Object Pool

A set of already-initialized objects are kept on-hand ("in the pool"), ready to use.

Don't allocate and destroy the objects on demand.

Retrieve from the pool rather than allocate and returned.

When finished it is returned to the pool rather than destroyed.

If the pool is empty, requesters will simply have to wait.

Creation Patterns: Object Pool

Limits the amount of concurrent request processing.

Prevents overloading.

Object pools not necessarily permanently-fixed in size.

Use when: allocation and release of resources is expensive, or done very frequently.
(even an inexpensive allocation, done a million times...).

Creation Patterns: Object Pool

Example: pool of workers process student transcript requests.

Requests are put into a queue. 3 workers (running in separate threads) to take the requests and process them.

When all workers are currently busy with a request, other students have to wait until a worker becomes free.

Creation Patterns: Object Pool

When a particular student's transcript request is at the front of the queue, an available worker takes that request.

The worker fulfills it, returning the transcript information.

The worker returns to the pool, ready to accept a new request.

Structure Patterns

Structural patterns are used to design the relationships between entities of the system.

Structure Patterns: Get-and-Set

You should already be familiar with the idea (ECE 150).

Access to variables is moderated through the use of methods that get and set those variables.

This hides the implementation of the object.

Structure Patterns: Get-and-Set

Use when: pretty much always.

Example: … Do I really need to give an example?

Structure Patterns: Adapter

An adapter converts the interface of one object into another.

The adapter rarely does any work on its own; it mostly converts communication (requests and responses) from one form to another.

Real life analogy: power adapter.

Structure Patterns: Adapter

When to use: you have two objects or modules which are fundamentally capable of interacting, but have incompatible interfaces.

Example: The Linux Software known as WINE.

It allows (some) Windows programs to work on a Linux system.

A call to a Windows system function (such as “draw window”) is handled by WINE.

It converts that request to the equivalent Linux form and send the converted request to the Linux system.

Structure Patterns: Proxy

Intermediate object that mediates access to the “real” object.

Instead of accessing the target, other classes access the proxy.

The proxy can do one or more things, such as check security or log some data.

Structure Patterns: Proxy

When to use: when access to a target object should be controlled or monitored.

Structure Patterns: Proxy

Example: you want to log database queries.

Where the database access class was used, return the proxy.

Proxy logs the requests and then delegates the execution to the database access class.

Behaviour Patterns

We use behaviour patterns to model common communications between different objects in the system.

Behaviour Patterns: Iterator

An Iterator is used to access all of the elements of a collection, such as a List, without knowing what the internal structure is.

The List may be implemented as an array or linked list.

The Iterator is a common way of accessing the elements (with the `next()` method in Java) regardless of implementation.

Use iterator on any collection: List, Queue, Stack, etc.

Behaviour Patterns: Iterator

Use when: examining the elements of a collection, if the collection might be of more than one type.

Behaviour Patterns: Iterator

Example: Professor wants to check programming assignments.

Rather than have a bunch of special-case code, use the iterator to go through each element in the collection.

Behaviour Patterns: Memento

Capture a copy of the internal state and provide a means for restoring the object to that state.

When to use: when we need to save and restore state.

Behaviour Patterns: Memento

Example: Editing a document.

The user can make changes, but if they click on cancel then the object is reverted to the previous state.

Behaviour Patterns: Visitor

A way of separating an algorithm from an object.

Have a new operation on code that you cannot (or choose not to) modify.

The class with the algorithm “visits” (examines) the elements.

Behaviour Patterns: Visitor

Use when: defining a new operation without changing the elements on which it operates.

Behaviour Patterns: Visitor

Example: You want to print a collection of elements out in a user-readable format to the screen.

It is some third party code that you cannot modify.

You can write a visitor that goes to each element and prints it to the screen in the right format.

Behaviour Patterns: Listener

Method that executes in response to a change in the system.

It is called a listener because it “listens” for changes in the target object.

When it “hears” such a change, it takes action. The action can be anything.

Behaviour Patterns: Listener

Register listeners: tell the system you care about changes in something.

The system (or your code) will also have to indicate when there has been a change.

When to use: when an object needs to be notified of a change in another object.

Behaviour Patterns: Listener

Example: program user interface.

When the data is changed, the listener is executed, and updates the screen so the change is visible to the user.

Behaviour Patterns: Listener

In an upcoming lecture, we will talk about listener implementation in more detail.

You may have already started using listeners in your lab.