**ECE155: Engineering Design with Embedded Systems** Winter 2015

# Lecture 8 — Software Design Patterns

*Jeff Zarnett*

## Software Design Patterns

Next, we go down a level from the architecture and communications structure and start looking at some common re-usable patterns for implementation within modules of the system. These are mostly relevant to object-oriented programming (OOP), but you should be familiar with this from ECE 150.

Here are some highlights from the Gang of Four Design Patterns [Car09, GHJV94], plus some that I've added. Many different design patterns can be used within the same system; Some may conflict with one another (for example, the Singleton isn't used for the same objects as the Template), but a large system could conceivably use all of them in different places.

### Creation Patterns

Creation patterns deal with, as the name suggests, creation of objects. How the objects are actually created is an implementation detail that is normally hidden from the caller. Creation patterns also serve the goal of composition: all of the creation logic should be kept together and not spread out over multiple classes.

**Factory.** The Factory pattern is one of the most common object-oriented design patterns. It is used for creating objects without exposing the instantiation logic and usually refers to the newly created object through a common interface. Or, in other words, create an object without specifying the exact class of the object to be created. Another use might be that setup is too complex for the constructor (knowledge is needed about the environment, etc). Constructors might be made private to force the use of the factory.

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

Example: Creating a new user. The `User` instantiation requires a lot of data not strictly available in the constructor of the `User` class, such as the user's group, permissions, default printer, et cetera. The solution is to have a `UserFactory` that creates a new `User`, populates its data fields as appropriate, and returns the `User`.

**Singleton.** In the Singleton design pattern, there is only one instance of a class and there is a global point of access to this instance. Creating new instances is not allowed, so the constructor will be private rather than public (and remember that in Java if you do not explicitly write a constructor, you get a default one!). Access to this class can be controlled through that single point in the system (return the instance if access is granted; return `null` if access is denied).

Use when: only one instance of the class should ever exist in the system.

Example: the software may have a `SecurityManager` class and there is only one instance of this class, accessed by `SecurityManager.getInstance()`.To check a permission, such as `Permissions.READ`, using the `SecurityManager`, the correct call is then:
`SecurityManager.getInstance().checkPermission(Permissions.READ)`.

**Template.** An object is used as the basis for making many copies; the copies can then be customized as necessary. The term comes from physical manufacturing processes where a template might be used to produce uniform car bodies (chassis) which are then customized according to the model of the car (sedan, coupe, SUV). Instead of having many lines of code where data is repeatedly set, define the object once and create copies from that object as are needed. This is also advantageous if it centralizes creation of the object, since it is necessary to update the logic only once.

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

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 the copy with the individual tax transactions for this month.

**Object Pool.** A set of already-initialized objects are kept on-hand ("in the pool"), ready to use, rather than allocating and destroying the objects on demand. When an object is needed, it is retrieved from the pool rather than allocated and returned, and when finished it is returned to the pool rather than destroyed. If the pool is empty, requesters will simply have to wait until an object is returned to the pool.

This also has the effect of limiting the amount of concurrent request processing. That is generally a good thing: attempting to do too many requests at once likely means degraded performance for all users of the system. We might limit the number of database connections to 5 to prevent overloading the database with requests.

Object pools are not necessarily of a permanently fixed size; sometimes the pool may grow or shrink depending on the available resources of the system and/or demand for the resources.

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

Example: a pool of workers to process requests for student transcripts. Request objects are put into a queue and there is a set of 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. When a particular student's transcript request is at the front of the queue, an available worker takes that request and fulfills it, returning the transcript information to the student. Then the worker returns to the pool, ready to accept the next transcript request.

## Structure Patterns

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

**Get-and-Set.** You should already be familiar with the idea of having get and set methods in your code, since that was likely emphasized in ECE 150. Rather than exposing the variables of a class to the outside, instead access to them is moderated through the use of methods that get and set those variables. This hides the implementation of the object.

Use when: pretty much always.

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

**Adapter.** An adapter converts the interface of one object into another; it allows classes to work together that otherwise would be incompatible. 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: if you have taken a trip to Europe, you may have noticed that the shape of the plugs for AC power is different there. Your device might accept the difference in voltage and frequency without any problem, but the prongs of the plug won't fit in. Solution: buy a travel adapter, that allows connection.

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

**Proxy.** A proxy is an intermediate object that mediates access to the "real" object. Instead of accessing the target directly, other classes access the proxy only. The proxy can do one or more things, such as check security or log some data.

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

Example: you want to log database queries. So in all places where the database access class was used, instead the callers should get the proxy which 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.

**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. This is not restricted to Lists, however, since we could use iterator on any collection: List, Queue, Stack, et cetera.

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

Example: A professor wants to check your programming assignment in which you had some choice about what data structure to implement your solution with. Rather than have a bunch of special-case code (if it's a List, do x, if it's a Queue, do y), use the iterator to go through each element in the collection.

**Memento.** In the Memento pattern, we capture a copy of the internal state (without revealing its internals) and provide a means for restoring the object to that state.

When to use: when it should be possible to save and restore state.

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

**Visitor.** The visitor pattern is a way of separating an algorithm from an object. It is a way to have a new operation on code that you cannot (or choose not to) modify. Since the class with the algorithm typically "visits" (examines) the elements of the structure, it is called a visitor.

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

Example: There is some third party code which you cannot modify, but you want to print a collection of elements out in a user-readable format to the screen. You can write a visitor that goes to each element and prints it to the screen in the right format.

**Listener.** A Listener is a class or 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, and when it “hears” such a change, it takes action. The action can be anything.

In addition, the listener needs to be registered: tell the system that your listener exists and what it is interested in listening for. 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.

Example: the user interface of programs often has a listener that listens for changes to the data being displayed. When the data is changed, the listener is executed, and updates the screen so the change is visible to the user.

## References

[Car09] Richard Carr. Gang of four design patterns, 2009. Online; accessed 21-December-2013.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.