# CS122 Lecture: Reuse, Components, API's and Frameworks

Last revised April 20, 2022

Objectives:

1. To introduce the basic concept of re-use
2. To introduce the notion of component-based software engineering
3. To introduce the notion of a framework

*Materials:*

1. Projectables
2. Ability to demo man command
3. Video to show: https://www.mulesoft.com/resources/api/what-is-an-api
4. Javadoc for ocsf package
5. Demo of framework based client-server chat and address book systems to run
   a) Executable of servers on laptop
   b) Executable of clients on laptop and Canvas
   c) Directions for running
6. Handout of methods of AbstractClient labeled as slot, service, hook

## I. **Re-use**

A. One idea that has been prominent in software engineering is the idea of re-use. One textbook defines this idea as follows: "Reuse refers to using components of one product to facilitate the development of a different product with different functionality" (Schach 2008 p. 216).

1. The idea is actually quite old - some of the initial work goes back over 40 years.

2. But reuse has been of particular interest in connection with object-orientation.

B. Re-use is desirable for several reasons

1. By re-using previously developed components, rather than developing new components from scratch, it becomes possible to develop a solution to a new problem more quickly and inexpensively.

2. Further, a reused component is likely to be more reliable than one developed from scratch, since it has already been tested in one or more previous uses.

3. Developing components with re-use in mind typically results in components that are better designed, documented, and tested.

C. Re-use can occur at many levels

1. Portions of code may be reusable. In particular, in OO it is possible to develop reusable classes.

   The standard Java library (or similar libraries for other languages) are a good example of this, but it is also possible to re-use code more specific to a particular problem area.

2. But so are things like portions of a design or documentation, etc.

3. As one important example: design patterns are reusable. Indeed, the subtitle of the "Gang of Four" book is "Elements of Reusable Object-Oriented software".

D. It has been estimated that only about 15% of a typical software product serves a truly original purpose. Thus, up to 85% of a piece of software may be constructed from re-used components - though in practice 40% seems to be an achievable upper bound. (Schach 217)

E. If reuse is a good idea, why isn't it practiced more often?

1. It is more expensive to develop a reusable component than it is to develop something that is intended to be used in just one place - in terms of careful design, documentation, and testing.

2. The "Not Invented Here" syndrome. (If someone else developed it, it can't possibly be as good as something I could produce)

3. Related to this is the reluctance of developers to trust work done by others - a fear that something that is reused might introduce a fault into the product.

4. There is also the problem of maintaining libraries of reusable components in such a way as to make it easy for someone to find what they're looking for.

F. Actually, there may be some dangers in re-use if not done with great care.

Read Schach §8.3.2 (attached)

### 8.3.2 European Space Agency

On June 4, 1996, the European Space Agency launched the Ariane 5 rocket for the first time. As a consequence of a software fault, the rocket crashed about 37 seconds after liftoff. The cost of the rocket and payload was about $500 million [Jézéquel and Meyer, 1997].

The primary cause of the failure was an attempt to convert a 64-bit integer into a 16-bit unsigned integer. The number being converted was larger than $2^{16}$, so an Ada **exception** (run-time failure) occurred. Unfortunately, there was no explicit exception handler in the code to deal with this exception, so the software crashed. This caused the onboard computers to crash which, in turn, caused the Ariane 5 rocket to crash.

Ironically, the conversion that caused the failure was unnecessary. Certain computations are performed before liftoff to align the inertial reference system. These computations should stop 9 seconds before liftoff. However, if there is a subsequent hold in the countdown, resetting the inertial reference system after the countdown has recommenced can take several hours. To prevent that happening, the computations continue for 50 seconds after the start of flight mode, that is, well into the flight (notwithstanding that, once liftoff has occurred, there is no way to align the inertial reference system). This futile continuation of the alignment process caused the failure.

The European Space Agency uses a careful software development process that incorporates an effective software quality assurance component. Then, why was there no exception handler in the Ada code to handle the possibility of such an overflow? To prevent overloading the computer, conversions that could not possibly result in overflow were left unprotected. The code in question was 10 years old. It had been reused, unchanged and without any further testing, from the software controlling the Ariane 4 rocket (the precursor of the Ariane 5). Mathematical analysis had proven that the computation in question was totally safe for the Ariane 4. However, the analysis was performed on the basis of certain assumptions that were true for the Ariane 4 but not for the Ariane 5. Therefore, the analysis no longer was valid, and the code needed the protection of an exception handler to cater to the possibility of an overflow. Were it not for the performance constraint, there surely would have been exception handlers throughout the Ariane 5 Ada code. Alternatively, the use of the **assert pragma** both during testing and after the product had been installed (Section 6.5.3) could have prevented the Ariane 5 crash if the relevant module had included an assertion that the number to be converted was smaller than $2^{16}$ [Jézéquel and Meyer, 1997].

The major lesson of this reuse experience is that software developed in one context must be retested when reused in another context. That is, a reused software module does not need to be retested by itself, but it must be retested after it has been integrated into the product in which it is reused. Another lesson is that it is unwise to rely exclusively on the results of mathematical proofs, as discussed in Section 6.5.2.

## II. **Components**

A. An idea closely connect to reuse is the notion of a reusable component.

Perhaps the idea can be illustrated by looking at some examples of the notion of a component from outside the realm of software engineering.

1. Standard hardware fasteners

a) If you go to any hardware store in the country, you'll find a variety of screws and other fasteners.

b) Though there is great variety, there is also a definite structure. For example, machine screws

(1) come in many lengths

(2)can be made of a variety of different materials (steel, brass, nylon)

(3)Can have several different head styles (flathead, roundhead, oval heat) and types of slot (straight, Phillips, Allen etc.)

(4)But they all have one of several standard thread types - e.g. 6-32. Moreover, a 6-32 nut will fit any 6-32 screw regardless of length, material, or head style.

c) When a manufacturer designs a product, it normally uses one of the standard designs instead of creating a custom design just for that product.

2. A home stereo system

a) My stereo system at home consists of a number of component parts, purchased over a period of several years.

(1) An LG TV

(2)A Sony CD changer

(3)A Sony receiver/amplifier

(4)A Jinto DVD player

(5)A pair of Bose loudspeakers

(6)Several Jensen loudspeakers

b) Though these components were purchased over a period of many years and come from five different manufacturers, they all work together perfectly. In fact, even the cables that connect them are all (except for the speakers) one of two types of cable - standard HiFi cables with RCA plugs on both ends or HDMI cables.

Why is this possible?

ASK

This is possible because of well-established standards in the industry concerning connectors, signal levels, impedances etc.

B. There is an approach to software engineering - called component based software engineering - that focusses on developing systems by combining components. The complete system must then include some "glue" code that holds the components together (analogous to the cables used in a stereo system) - but the bulk of the functionality is provided by the components.

Example: PROJECT Example of a Component-based System from Wikipedia

C. While the full vision of software developed by interconnecting “standard” components is far from reality, there are a number of places where this sort of approach is used in practice.

1. Standard libraries (the Java Class library , Microsoft .NET, libraries of scientific computing functions, etc.)

2. Libraries supporting specific applications (Java EE, etc.)

3. Mobile apps that use functionality provided by one or more API's implemented on one or more servers.

4. Plugins for web-browsers and various software packages

## III. **API's**

A. The capabilities of a re-usable software component or set of components are typically specified through what is known as an API - Application Programming Interface.

B. The idea of an API originated with operating systems. The capabilities furnished by an OS to applications running on it would be specified by its API.

1. The standard API for Unix-like systems is specified by an IEEE specification known as POSIX (Portable Operating System Interface). One important part of POSIX is a set of specifications for system routines that can be used by application programs.

   a) Example: man getuid

   b) Many Unix-like systems support other API's as well

2. Likewise, there is a Microsoft Windows API specified by Microsoft.

3. Mac OSX implements both the POSIX API and Berkeley Unix APIs and also offers a large number of application-oriented API's..

C. Today, <u>many</u> information sources provide access to information through API's they furnish, which allow applications to access their information in a clearly defined way.

The following video does a good job of explaining the concept:

PLAY: https://www.mulesoft.com/resources/api/what-is-an-api

D. The use of API's is discussed much more fully in the Internet Programming and Mobile Devices courses as well as the Senior Project

## IV. **Frameworks**

A. One interesting approach to re-use centers on the notion of a framework. A framework is a skeleton of a particular type of application that can be converted into a complete application by adding a few key elements of code.

B. Perhaps the best way to understand the concept is to look at an example. A book that was once used for a previous version of this course included an example framework for creating a simple client-server application.

1. Using this framework, it is possible to create a simple client-server application by writing a minimum amount of code.

   a) DEMO:

   Run both Chat server and client on my system and demo. Then ask one or more other students to run client (from Canvas) on their systems and demo - give each directions Handout

   Note: to run client `java -jar ChatClient.jar` *`screen-name ip`*
   ip can be left blank for localhost

   b) How much code do you think is needed to accomplish this?

   ASK

   PROJECT Code for Server and Client - Focus on code found in `handleMessageFromServer()`, `handleMessageFromClient()`.

   Note that the complex networking code is all found in the abstract base classes re-used from the framework, so these classes only need to deal with the application-specific matters

```
/* SimpleChatServer.java */

package frameworkdemo.server;
import ocsf.server.AbstractServer;
import ocsf.server.ConnectionToClient;

/** Server side of a very simple client-server chat system - demonstrating the
 *  use of the ocsf framework.  Simplified from demonstration class EchoServer
 *  written by Timothy Lethbridge and Robert Langaniere
 *
 * @author Russell C. Bjork
 */
public class SimpleChatServer extends AbstractServer
{
  /**
   * Constructs an instance of the simple server
   */
  public SimpleChatServer()
  {
    super(PORT);
  }

  /**
   * This method handles any messages received from the client.
   *
   * @param msg The message received from the client.
   * @param client The connection from which the message originated.
   */
  public void handleMessageFromClient(Object msg, ConnectionToClient client)
  {
    sendToAllClients(msg);
  }

  /** Main program
   *
   *  @exception any exception thrown while listening is propagated
   */
  public static void main(String[] args) throws Exception
  {
    new SimpleChatServer().listen();
  }

  /*
   * Port used by this application
   */
  public static final int PORT = 5555;
}
```

```
/* SimpleChatClient.java */

package frameworkdemo.client;

import java.io.*;
import ocsf.client.AbstractClient;

/** Client side of a very simple client-server chat system - demonstrating the
 *  use of the ocsf framework.  Simplified from demonstration classes ChatClient
 *  and ClientConsole written by Timothy Lethbridge and Robert Langaniere
 *
 * @author Russell C. Bjork
 */
public class SimpleChatClient extends AbstractClient
{
  /**
   * Constructs an instance of the simple client.
   *
   * @paran name the name to use in messages from this client
   * @param host The server to connect to.
   * @param IOException if the connection cannot be opened
   */

  public SimpleChatClient(String name, String host) throws IOException
  {
    super(host, PORT);
    clientName = name;
    reader = new BufferedReader(new InputStreamReader(System.in));

    // Open a connection to the server.  This will start a thread to handle
    // messages from the server

    openConnection();

    // Set up a separate thread to handle input from the user

    new Thread() {
        public void run()
        {
            while(true)
            {
                handleInputFromUser();
            }
        }
    }.start();
  }
```

```
  /**
   * This method handles all data that comes in from the server.
   *
   * @param msg The message from the server.
   */
  public void handleMessageFromServer(Object msg)
  {
    System.out.println(msg.toString());
  }

  /**
   * This method waits reads a message from the user.  Once it is
   * read, it sends it to the server with the client name appended.
   */
  public void handleInputFromUser()
  {
    try
    {
       String message = reader.readLine();
       sendToServer(clientName + ": " + message);
    }
    catch (Exception e)
    {
      System.err.println
        ("Unexpected error while reading from console! " + e);
    }
  }

  /** Main program
   *
   * @param args[0] The name to use in messages from this client - ? will be
   *                used if omitted
   * @param args[1] The host to connect to - localhost will be used if omitted
   * @exception any exception thrown while connecting is propagated
   */
  public static void main(String[] args) throws Exception
  {
      new SimpleChatClient(args.length < 1 ? "?" : args[0],
                           args.length < 2 ? "localhost" : args[1]);
  }

  /*
   * Port used by this application
   */
  public static final int PORT = 5555;

  // Name to be used in messages from this client

  private String clientName;

  // Reader to be used in getting input from the user

  BufferedReader reader;
}
```

2. The framework consists of two packages - `client` and `server` - used to develop the programs running on the client and server system, respectively.

   a) The main class in the client package is an abstract class known as `AbstractClient`. As a minimum, to create a client application, one creates a concrete subclass of this class which implements just one method: `handleMessageFromServer()`.

   b) The main class in the server package is an abstract class known as `AbstractServer`. As a minimum, to create a server application, one creates a concrete subclass of this class which implements just one method: `handleMessageFromClient()`.

   c) Show Javadoc for oscf

3. This framework can be used to create many other kinds of simple client-server application.

   a) DEMO: Run both AddressBook server and client on my system and demo with Anthony and Zelda

      Note: to run client `java -jar AddressBookClient.jar` *`ip`*
      ip can be left blank for localhost

      (1)Note that this version is much simpler than the one used in Lab 3 - it just supports lookup of last name given first name.

      (2)Then ask one or more other students to run client (from Canvas) on their systems and demo - give each directions Handout

   b) The Client code in this case is virtually identical to the chat client - in fact, it is a bit simpler since it doesn't append a screen name to messages send to the server

PROJECT: Code for AddressBook server - note how similar it is to ChatServer!

```
/* SimpleAddressBookServer.java */

package frameworkdemo.server;
import java.util.TreeMap;
import ocsf.server.AbstractServer;
import ocsf.server.ConnectionToClient;

/** Server side of a very simple client-server address book system -
 *  demonstrating the use of the ocsf framework.  Simplified from
 *  demonstration class EchoServer  written by Timothy Lethbridge and
 *  Robert Langaniere
 *
 * @author Russell C. Bjork
 */
public class SimpleAddressBookServer extends AbstractServer
{
  /**
   * Constructs an instance of the simple server
   */
  public SimpleAddressBookServer()
  {
    super(PORT);
    people = new TreeMap<String, String>();
    people.put("Anthony", "Aardvark");
    people.put("Boris", "Buffalo");
    people.put("Charlene", "Cat");
  }

  /**
   * This method handles an inquiry from a client.
   *
   * @param inquiry The inquiry received from the client.
   * @param client The connection from which the message originated.
   */
  public void handleMessageFromClient(Object inquiry,
                                      ConnectionToClient client)
  {
    String result = people.get(inquiry);
    if (result == null)
        result = "Unknown";
    try
    {
        client.sendToClient(result);
    }
    catch(Exception e)
    {
        System.err.println(e);
    }
  }
```

```
  /** Main program
   *
   *  @exception any exception thrown while listening is propagated
   */
  public static void main(String[] args) throws Exception
  {
    new SimpleAddressBookServer().listen();
  }

  /*
   * Port used by this application
   */
  public static final int PORT = 5556;

  private TreeMap<String, String> people;
}
```

c) (Of course, a database might be used for storing the address book and the response to an inquiry would be more meaningful - this example is just to illustrate the power of the framework)

C. Framework code includes three kinds of methods that are relevant to a person using a framework. We will illustrate each using the server class of the chat system.

HANDOUT: Annotated Javadoc for class `AbstractClient`

1. Slot methods

   A slot method is one that is typically left as abstract in the framework (and thus is a method of an abstract class). To use the framework, one must supply a concrete implementation of the method, which does whatever the specific application requires.

   Example: the method `handleMessageFromServer()` in class `SimpleChatClient`. PROJECT AGAIN

```
/**
 * This method handles all data that comes in from the server.
 *
 * @param msg The message from the server.
 */
public void handleMessageFromServer(Object msg)) {
  System.out.println(msg.toString());
}
```

2. Service methods - the methods provided by the framework that the user can use in coding the slot methods.

   Example: `sendToServer()` in `handleInputFromUser` in `SimpleChatClient`. PROJECT AGAIN

```
/**
 * This method waits reads a message from the user.  Once it is
 * read, it sends it to the server with the client name appended.
 */
public void handleInputFromUser()
{
   try
   {
      String message = reader.readLine();
      sendToServer(clientName + ": " + message);
   }
   catch (Exception e)
   {
        System.err.println
            ("Unexpected error while reading from console! " + e);
   }
 }
```

   (Note that most of the other service methods are not actually needed by `SimpleChatClient`)

3. Hook methods - methods that allow the user to add additional functionality if appropriate for a specific context. These are methods that are typically implemented as null methods (i.e. a body consisting only of {} or returning some default value in the framework. But the user can override such a method to add application-specific functionality where needed.

   Example: the ocsf server framework class `AbstractClient` provides a hook method called `connectionClosed()`, which is called whenever a client connection is closed. The default implementation is empty, but a concrete implementation can override this if it wants to do something special when a client connection is closed. (Not needed in `SimpleChatClient`)

D. Basing an application on a suitable framework - rather than developing everything from scratch - can not only save a lot of effort, but also can result in more robust code, for two reasons:

   1. The framework developers have (hopefully) produced code that is more carefully designed and thoroughly tested than what the application developer might produce for just a single application.

   2. By application developer can concentrate on the core functionality of the application, without having to become enmeshed in the details of functionality (like - in the case of our example - network communication) which is not central to the application.

E. However, developing a quality reusable framework can be much more expensive than developing single-application code, because of the need for generality.

   1. Example: note hook methods of `AbstractClient` in the javadoc. The designer of a framework needs to anticipate the things an application might want to do and provide hooks for them, as opposed to simply thinking about the needs of a single application.

   2. However, purchasing a suitable general framework from a vendor is often much cheaper than developing one's own application-specific code.