Polytech Montpellier – IG4

Software Engineering & Design Principles

Frameworks

Frameworks

Building on the Experience of Others

- Software engineers should avoid re-developing software already developed
- Types of reuse:
  - Reuse of expertise
  - Reuse of standard designs and algorithms
  - Reuse of libraries of classes or procedures
  - Reuse of powerful commands built into languages and operating systems
  - Reuse of frameworks
  - Reuse of complete applications

Reusability and Reuse in SE

- *Reuse* and design for *reusability* should be part of the culture of software development organizations
- But there are problems to overcome:
  - Why take the extra time needed to develop something that will benefit *other* projects/customers?
  - Management may only reward the efforts of people who create the *visible 'final products'*.
  - Reusable software is often created in a hurry and *without enough attention to quality*.

A vicious cycle

- Developers tend not develop high quality reusable components, so there is often little to reuse
- To solve the problem, recognize that:
  - This vicious cycle costs money
  - *Investment* in reusable code is important
  - Attention to quality of reusable components is essential
    - So that potential reusers have confidence in them
    - The quality of a software product is only as good as its lowest-quality reusable component
  - Developing reusable components can often simplify design

Frameworks: Reusable Subsystems

- A *framework* is reusable software that implements a generic solution to a generalized problem.
  - It provides common facilities applicable to different application programs.

- Principle: applications that do different, but related, things tend to have quite similar designs

Frameworks to promote reuse

- A framework is intrinsically *incomplete*
  - Certain classes or methods are used by the framework, but are missing (*slots*)
  - Some functionality is optional
    - Allowance is made for developer to provide it (*hooks*)
  - Developers use the *services* that the framework provides
    - Taken together the services are called the Application Program Interface (*API*)

Object-oriented Frameworks

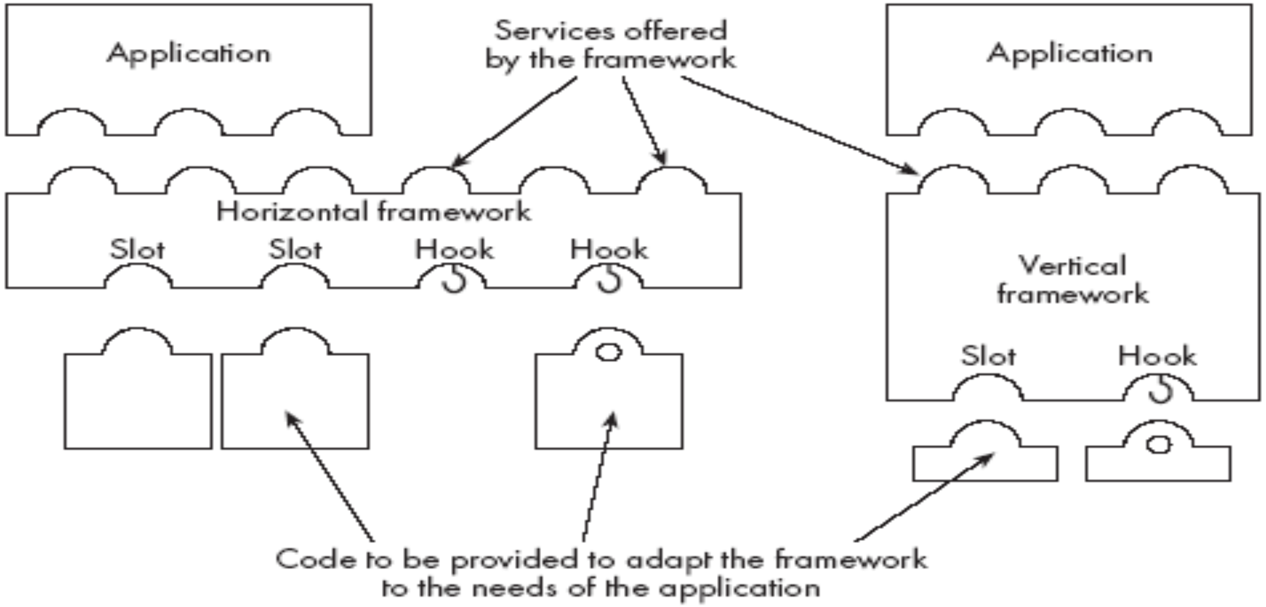
- In the object oriented paradigm, a framework is composed of a library of classes.
  - The API is defined by the set of all public methods of these classes.
  - Some of the classes will normally be abstract
- Example:
  - A framework for payroll management
  - A framework for frequent buyer clubs
  - A framework for university registration
  - A framework for e-commerce web sites

Frameworks and product lines

- A product line (or product family) is a set of products built on a common base of technology.
  - The various products in the product line have different features to satisfy different markets
  - The software technology common to all products is included in a framework
  - Each product is produced by filling the available hooks and slots
  - E.g. software products offering ‘demo’, ‘lite’ or ‘pro’ versions

Types of frameworks

- A *horizontal* framework provides general application facilities that a large number of applications can use
- A *vertical* framework (*application framework)* is more 'complete' but still needs some slots to be filled to adapt it to specific application needs


Application
Services offered by the framework
Application
Horizontal framework
Slot
Slot
Hook
Hook
Vertical framework
Slot
Hook
Code to be provided to adapt the framework to the needs of the application

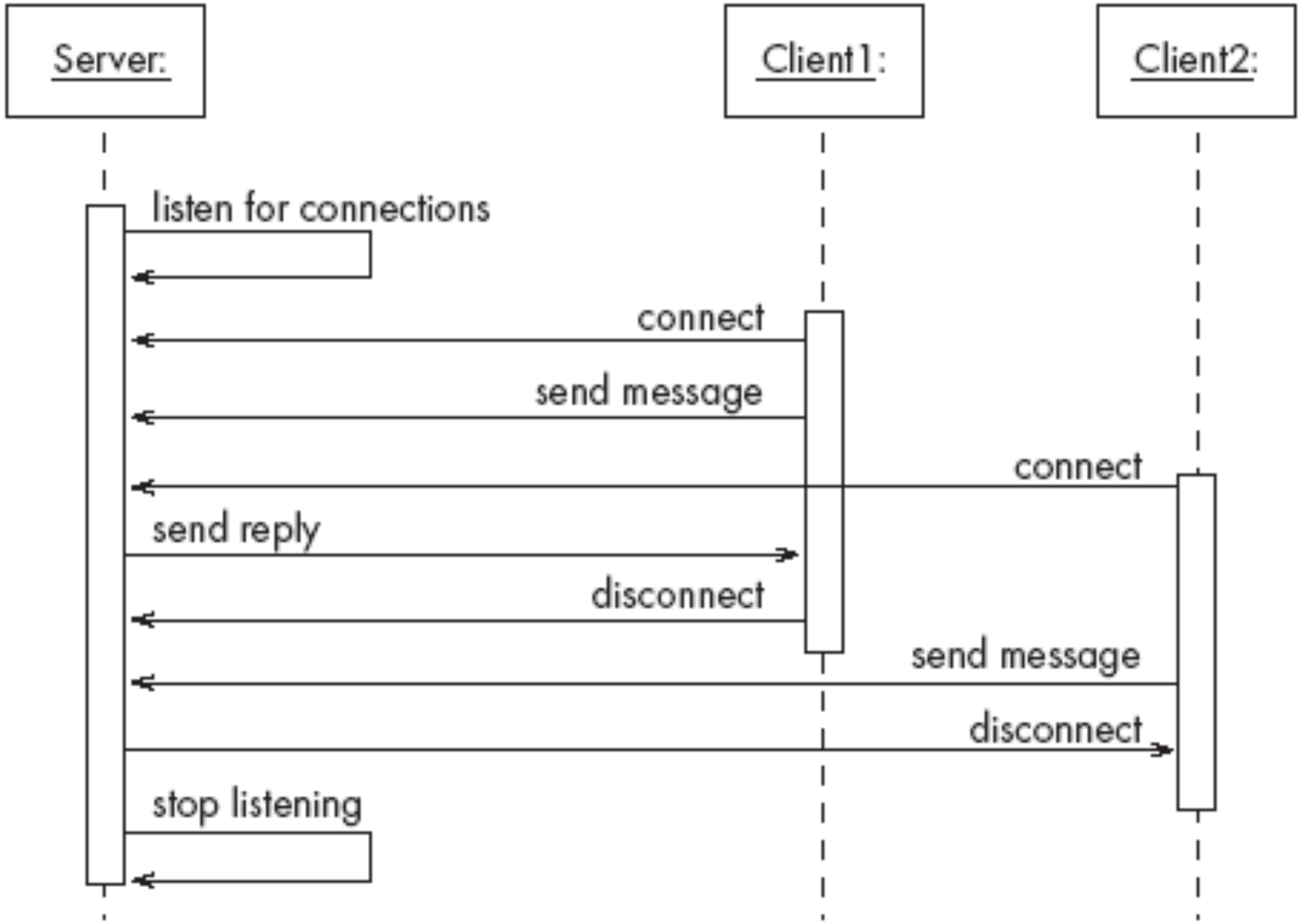
The Client-Server Architecture

- A distributed system is a system in which:
  - computations are performed by separate programs
  - normally running on separate pieces of hardware
  - that co-operate to perform the task of the system.
- Server:
  - A program that provides a service for other programs that connect to it using a communication channel
- Client
  - A program that accesses a server (or several servers) to obtain services
  - A server may be accessed by many clients simultaneously

Sequence of activities in a client-server system

1. The **server starts running**
2. The **server waits for clients** to connect. (*listening*)
3. Clients start running and perform operations
   - Some operations involve requests to the server
4. When a client attempts to connect, the **server accepts the connection** (if it is willing)
5. The **server waits for messages** to arrive from connected clients
6. When a message from a client arrives, the **server takes some action** in response, then resumes waiting
7. Clients and servers continue functioning in this manner until they decide to shut down or disconnect

A server communicating with two clients



Alternatives to the client server architecture

- Have a *single program* on one computer that does everything

- Have *no communication*
  - Each computer performs the work separately

- Have some mechanism other than client-server communication for exchanging information
  - E.g. one program writes to a database; the other reads from the database

Advantages of client-server systems

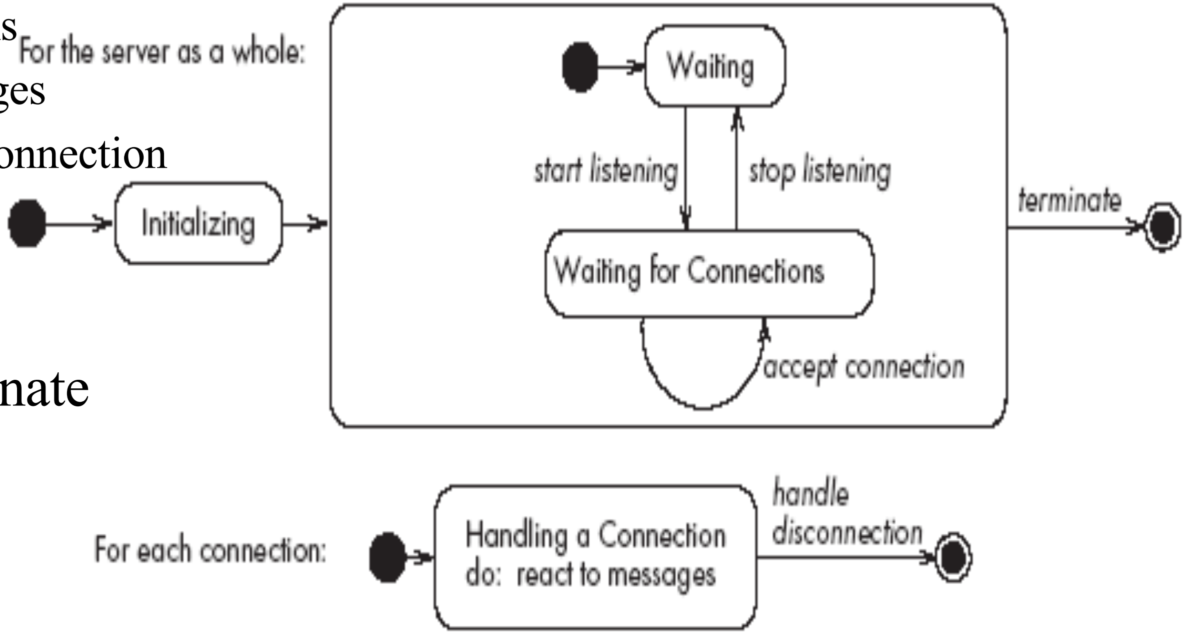
- The work can be distributed among different machines
- The clients can access the server’s functionality from a distance
- The client and server can be designed separately
- They can both be simpler
- All the data can be kept centrally at the server
- Conversely, data can be distributed among many different geographically-distributed clients or servers
- The server can be accessed simultaneously by many clients
- Competing clients can be written to communicate with the same server, and vice-versa

Example of client-server systems

- The World Wide Web
- Email
- Network File System
- Transaction Processing System
- Remote Display System
- Communication System
- Database System

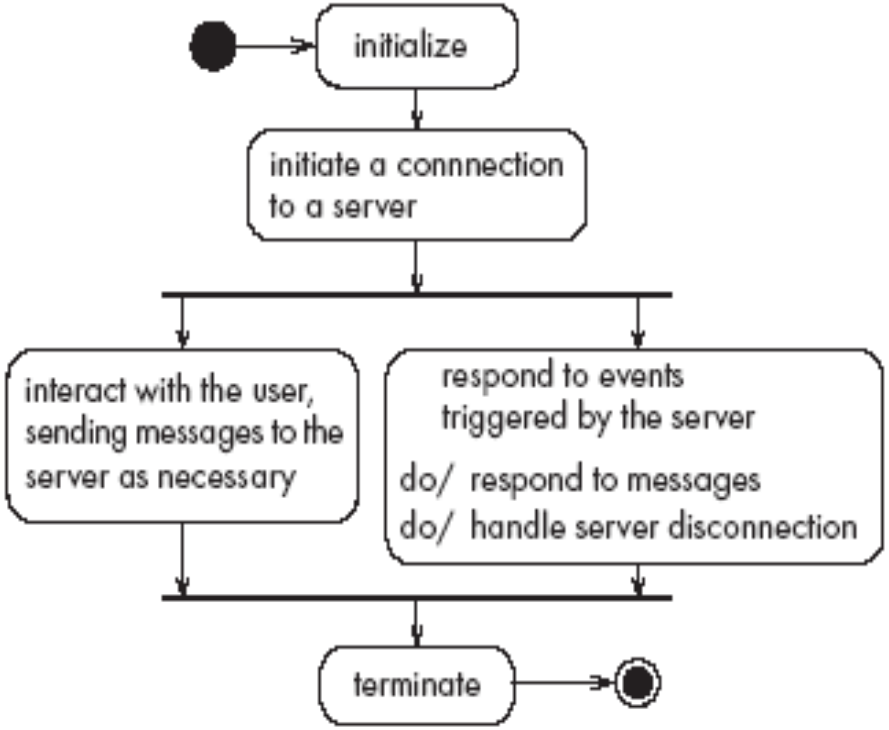
Activities of a server

1. Initializes itself
2. Starts listening for clients
3. Handles the following types of events originating from clients
    1. accepts connections
    2. responds to messages
    3. handles client disconnection
4. May stop listening
5. Must cleanly terminate

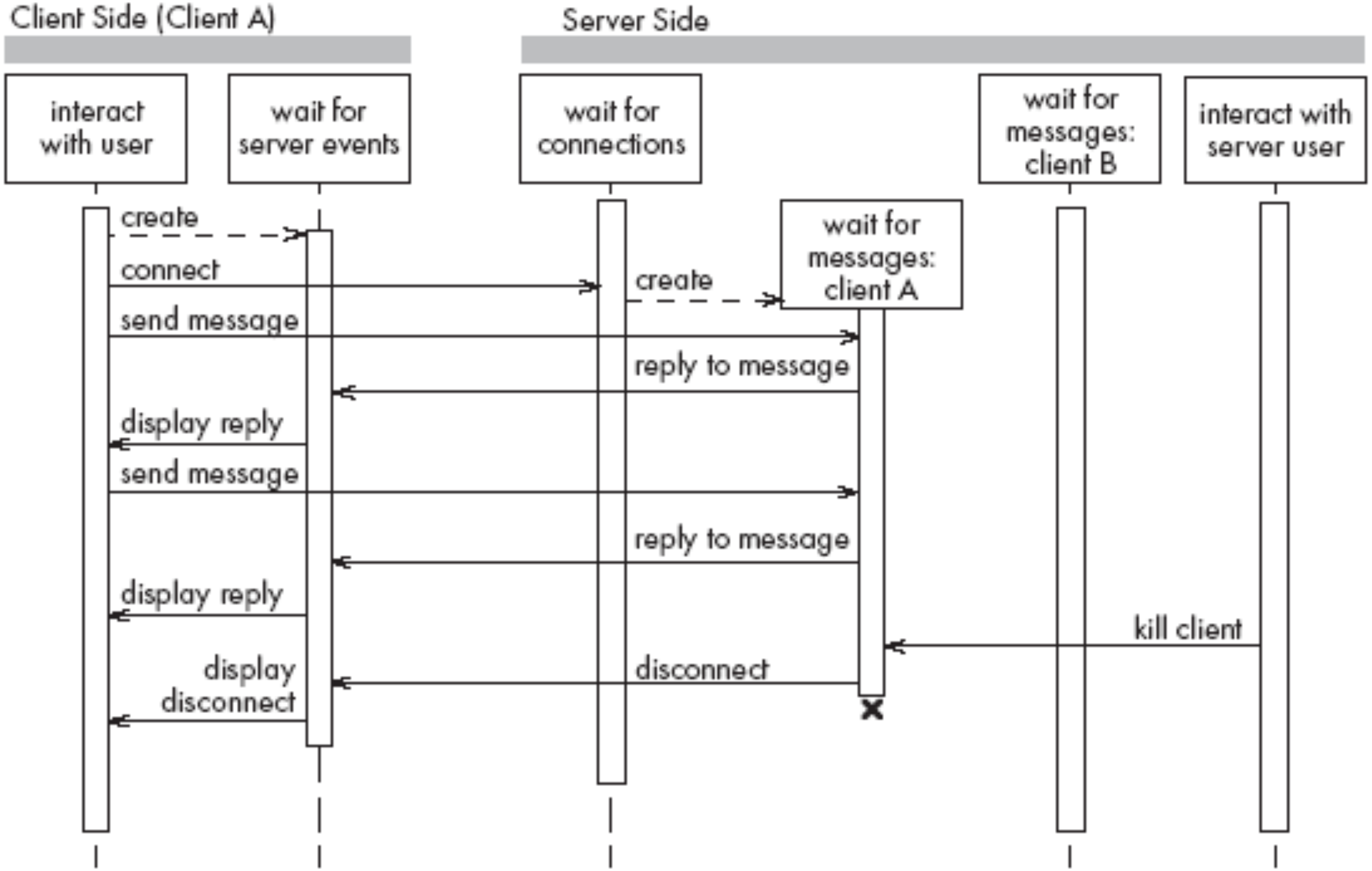


Activities of a client

1. Initializes itself
2. Initiates a connection
3. Sends messages
4. Handles the following types of events originating from the server
    1. responds to messages
    2. handles server disconnection
5. Must cleanly terminate

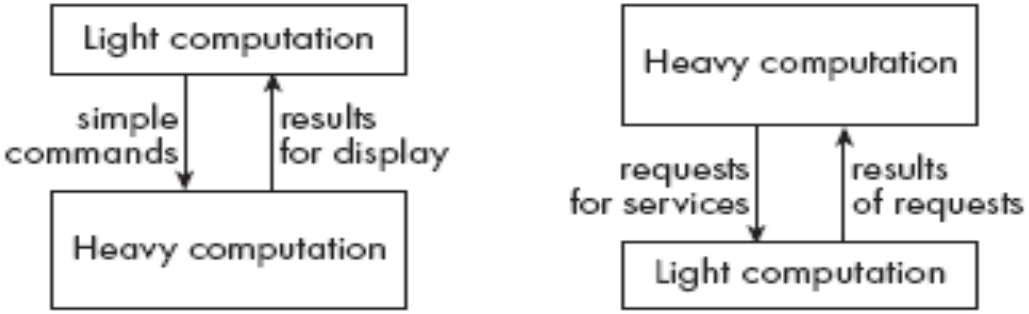

initialize
initiate a connnection to a server
interact with the user, sending messages to the server as necessary
respond to events triggered by the server
do/ respond to messages
do/ handle server disconnection
terminate

Threads in a client-server system



Thin-client vs. fat-client systems

- *Thin-client* system (a)
  - Client is made as small as possible
  - Most of the work is done in the server.
  - Client easy to download over the network
- *Fat-client* system (b)
  - As much work as possible is delegated to the clients.
  - Server can handle more clients



Communication protocols

- The messages the client sends to the server form a *language*.
  - The server has to be programmed to understand that language.
- The messages the server sends to the client also form a language.
  - The client has to be programmed to understand that language.
- When a client and server are communicating, they are in effect having a conversation using these two languages
- The two languages and the rules of the conversation, taken together, are called the *protocol*

Tasks to perform to develop client-server applications

- Design the primary work to be performed by both client and server
- Design how the work will be distributed
- Design the details of the set of messages that will be sent
- Design the mechanism for
  - Initializing
  - Handling connections
  - Sending and receiving messages
  - Terminating

Technology Needed to Build Client-Server Systems

- Internet Protocol (IP)
  - Route messages from one computer to another
  - Long messages are normally split up into small pieces
- Transmission Control Protocol (TCP)
  - Handles connections between two computers
  - Computers can then exchange many IP messages over a connection
  - Assures that the messages have been satisfactorily received
- A host has an IP address and a host name
  - Several servers can run on the same host.
  - Each server is identified by a port number (0 to 65535).
  - To initiate communication with a server, a client must know both the host name and the port number

Establishing a connection in Java

- The **`java.net`** package
  - Permits the creation of a TCP/IP connection between two applications
- Before a connection can be established, the server must start listening to one of the ports:

```
ServerSocket serverSocket = new
  ServerSocket(port);
Socket clientSocket = serverSocket.accept();
```

- For a client to connect to a server:

```
Socket clientSocket = new Socket(host, port);
```

Exchanging information in Java

- Each program uses an instance of
  - **`InputStream`** to receive messages from the other program
  - **`OutputStream`** to send messages to the other program
  - These are found in package **`java.io`**

```
output = clientSocket.getOutputStream();
input = clientSocket.getInputStream();
```

Sending and receiving messages

- without any filters (raw bytes)

```
output.write(msg);
msg = input.read();
```

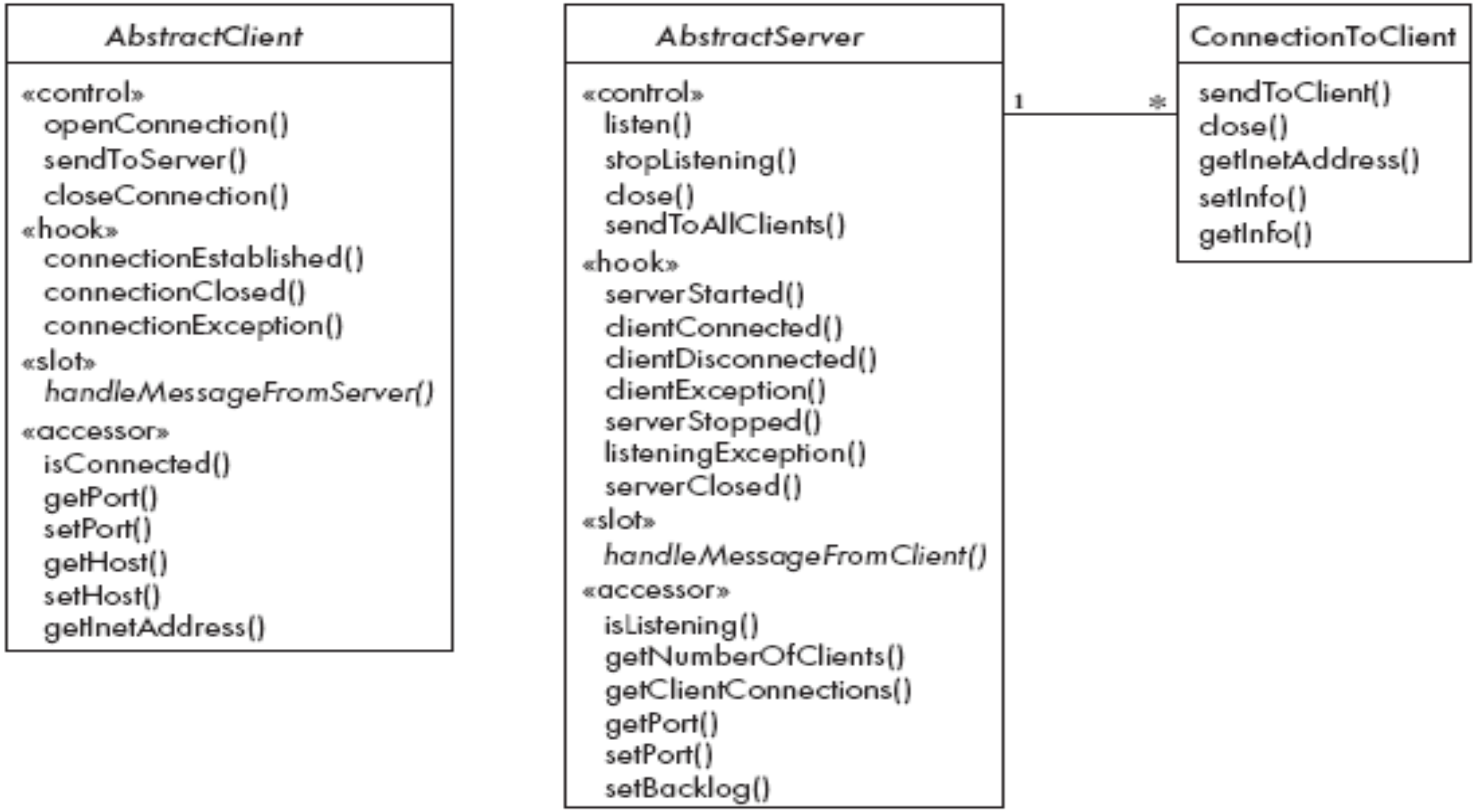
- or using **`DataInputStream`**/**`DataOutputStream`** filters

```
output.writeDouble(msg);
msg = input.readDouble();
```

- or using **`ObjectInputStream/ObjectOutputStream`** filters

```
output.writeObject(msg);
msg = input.readObject();
```

The Object Client-Server Framework (OCSF)



AbstractClient

«control»
openConnection()
sendToServer()
closeConnection()
«hook»
connectionEstablished()
connectionClosed()
connectionException()
«slot»
handleMessageFromServer()
«accessor»
isConnected()
getPort()
setPort()
getHost()
setHost()
getInetAddress()

AbstractServer

«control»
listen()
stopListening()
close()
sendToAllClients()
«hook»
serverStarted()
clientConnected()
clientDisconnected()
clientException()
serverStopped()
listeningException()
serverClosed()
«slot»
handleMessageFromClient()
«accessor»
isListening()
getNumberOfClients()
getClientConnections()
getPort()
setPort()
setBacklog()

1

*

ConnectionToClient

sendToClient()
close()
getInetAddress()
setInfo()
getInfo()

Using OCSF

- Software engineers using OCSF *never* modify its three classes
- They:
  - *Create subclasses* of the abstract classes in the framework
  - *Call public methods* that are provided by the framework
  - *Override* certain slot and hook methods (explicitly designed to be overridden)

The Client Side

- Consists of a single class: **`AbstractClient`**
  - *Must* be subclassed
    - Any subclass must provide an implementation for **`handleMessageFromServer`**
      - Takes appropriate action when a message is received from a server
  - Implements the **`Runnable`** interface
    - Has a **`run`** method which
      - Contains a loop that executes for the lifetime of the thread

The public interface of **`AbstractClient`**

- Controlling methods:
  - **`openConnection`**
  - **`closeConnection`**
  - **`sendToServer`**
- Accessing methods:
  - **`isConnected`**
  - **`getHost`**
  - **`setHost`**
  - **`getPort`**
  - **`setPort`**
  - **`getInetAddress`**

The callback methods of **`AbstractClient`**

- Methods that *may* be overridden:
  - **`connectionEstablished`**
  - **`connectionClosed`**
- Method that *must* be implemented:
  - **`handleMessageFromServer`**

Using **`AbstractClient`**

- Create a subclass of **`AbstractClient`**
- Implement **`handleMessageFromServer`** slot method
- Write code that:
  - Creates an instance of the new subclass
  - Calls **`openConnection`**
  - Sends messages to the server using the **`sendToServer`** service method
- Implement the **`connectionClosed`** callback
- Implement the **`connectionException`** callback

Internals of **`AbstractClient`**

- Instance variables:
  - A **`Socket`** which keeps all the information about the connection to the server
  - Two streams, an **`ObjectOutputStream`** and an **`ObjectInputStream`**
  - A **`Thread`** that runs using **`AbstractClient`**'s **`run`** method
  - Two variables storing the *host* and *port* of the server

The Server Side

- Two classes:
  - One for the thread which listens for new connections (**`AbstractServer`**)
  - One for the threads that handle the connections to clients (**`ConnectionToClient`**)

The public interface of **`AbstractServer`**

- Controlling methods:
  - **`listen`**
  - **`stopListening`**
  - **`close`**
  - **`sendToAllClients`**
- Accessing methods:
  - **`isListening`**
  - **`getClientConnections`**
  - **`getPort`**
  - **`setPort`**
  - **`setBacklog`**

The callback methods of **`AbstractServer`**

- Methods that *may* be overridden:
  - **`serverStarted`**
  - **`clientConnected`**
  - **`clientDisconnected`**
  - **`clientException`**
  - **`serverStopped`**
  - **`listeningException`**
  - **`serverClosed`**
- Method that *must* be implemented:
  - **`handleMessageFromClient`**

The public interface of **`ConnectionToClient`**

- Controlling methods:
  - **`sendToClient`**
  - **`close`**
- Accessing methods:
  - **`getInetAddress`**
  - **`setInfo`**
  - **`getInfo`**

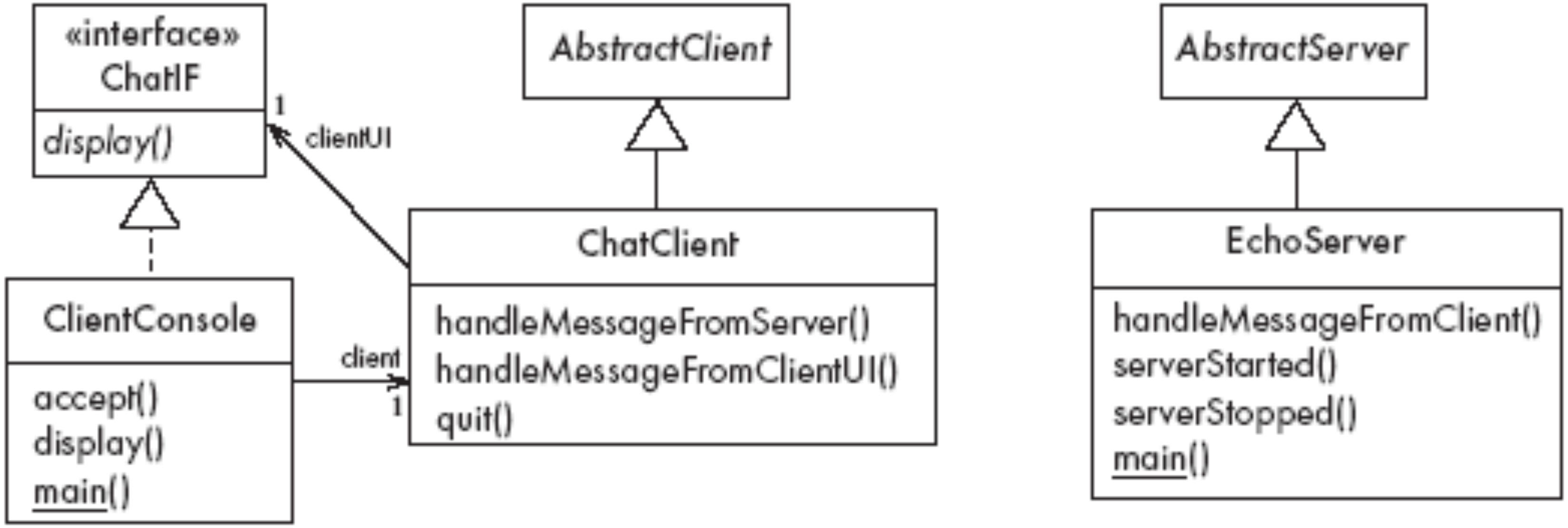
Using **AbstractServer** and **ConnectionToClient**

- Create a subclass of **AbstractServer**
- Implement the slot method **handleMessageFromClient**
- Write code that:
  - Creates an instance of the subclass of **AbstractServer**
  - Calls the **listen** method
  - Sends messages to clients, using:
    - the **getClientConnections** and **sendToClient** service methods
    - or **sendToAllClients**
- Implement one or more of the other callback methods

Internals of **`AbstractServer`** and **`ConnectionToClient`**

- The **`setInfo`** and **`getInfo`** methods make use of a Java class called **`HashMap`**
- Many methods in the server side are **`synchronized`**
- The collection of instances of **`ConnectionToClient`** is stored using a special class called **`ThreadGroup`**
- The server must pause from listening every 500ms to see if the **`stopListening`** method has been called
  - if not, then it resumes listening immediately

An Instant Messaging Application: SimpleChat



- **`ClientConsole`** can eventually be replaced by **`ClientGUI`**

The server

- **`EchoServer` is a subclass of `AbstractServer`**
  - The **`main`** method creates a new instance and starts it
    - It listens for clients and handles connections until the server is stopped
  - The three *callback* methods just print out a message to the user
    - **`handleMessageFromClient`**, **`serverStarted`** and **`serverStopped`**
  - The *slot* method **`handleMessageFromClient`** calls **`sendToAllClients`**
    - This echoes any messages

Key code in `EchoServer`

```
public void handleMessageFromClient
  (Object msg, ConnectionToClient client)
{
   System.out.println(
     "Message received: "
     + msg + " from " + client);
   this.sendToAllClients(msg);
}
```

The client

- When the client program starts, it creates instances of two classes:
  - **`ChatClient`**
    - A subclass of **`AbstractClient`**
    - Overrides **`handleMessageFromServer`**
      - This calls the display method of the user interface
  - **`ClientConsole`**
    - User interface class that implements the interface **`ChatIF`**
      - Hence implements **`display`** which outputs to the console
    - Accepts user input by calling **`accept`** in its **`run`** method
    - Sends all user input to the **`ChatClient`** by calling its **`handleMessageFromClientUI`**
      - This, in turn, calls **`sendToServer`**

Key code in `ChatClient`

```
public void handleMessageFromClientUI(String message)
{
  try
  {
    sendToServer(message);
  }
  catch(IOException e)
  {
    clientUI.display(
       "Could not send message. " + "Terminating client.");
    quit();
  }
}

public void handleMessageFromServer(Object msg)
{
  clientUI.display(msg.toString());
}
```

Risks when reusing technology

- Poor quality reusable components
  - Ensure that the developers of the reusable technology:
    - follow good software engineering practices
    - are willing to provide active support
- Compatibility not maintained
  - Avoid obscure features
  - Only re-use technology that others are also re-using

Risks when developing reusable technology

- Investment uncertainty
  - Plan the development of the reusable technology, just as if it was a product for a client
- The ‘not invented here syndrome’
  - Build confidence in the reusable technology by:
    - Guaranteeing support
    - Ensuring it is of high quality
    - Responding to the needs of its users

Risk when developing reusable technology – continued

- Competition
  - The reusable technology must be as useful and as high quality as possible
- Divergence (tendency of various groups to change technology in different ways)
  - Design it to be general enough, test it and review it in advance

Risks when adopting a client-server approach

- Security
  - Security is a big problem with no perfect solutions: consider the use of encryption, firewalls, ...
- Need for adaptive maintenance
  - Ensure that all software is forward and backward compatible with other versions of clients and servers