Frameworks: Reusable Subsystems

The OCSF

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 similar designs:**

- similar patterns of interaction among the components

***Method*: Identify the common design elements and develop software that implements these design elements in a reusable way.**

Frameworks Architecture

A framework is intrinsically *incomplete*

- ***Slots:*** Certain classes or methods are used by the framework, but are missing

- ***Hooks:*** Some functionality is optional
  - Allowance is made for developer to provide it

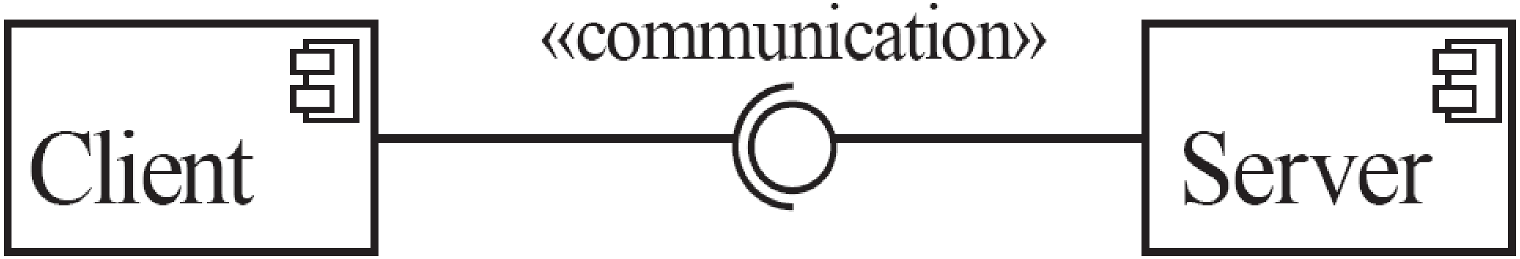
- Developers use the *services* that the framework provides
  - Taken together the services are called the Application Program Interface (***API***)

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: The Client-Server Architecture



Example - client-server systems:

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

© Lethbridge/Laganière 2005

Example: 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

Advantages of client-server systems

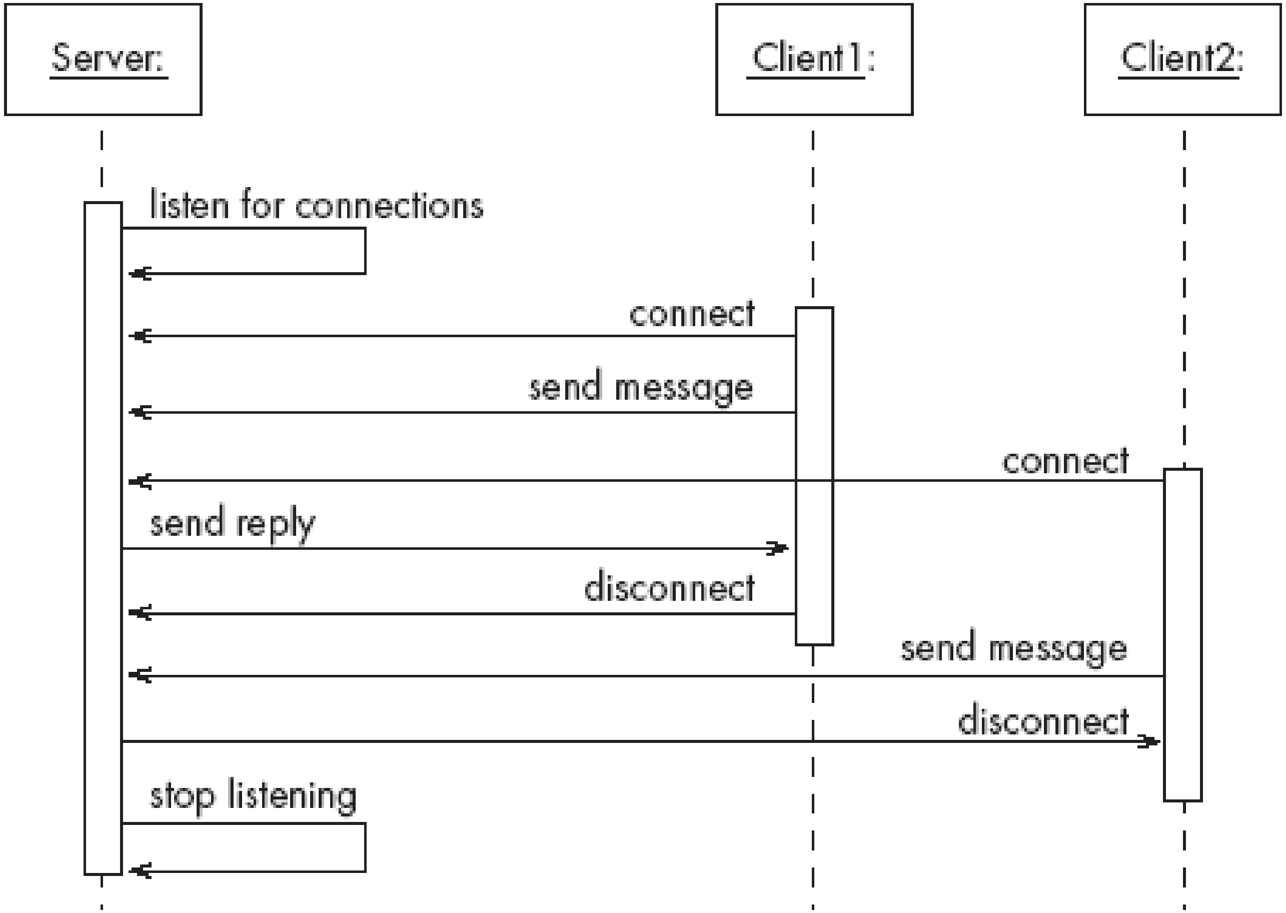
- The work can be *distributed* among different machines
- The clients can access the server's functionality from a *distance*
- The client and server can be *designed separately*
- They can both be *simpler*
- All the *data can be kept centrally* at the server
- Conversely, *data can be distributed* 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

© Lethbridge/Laganière 2005

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

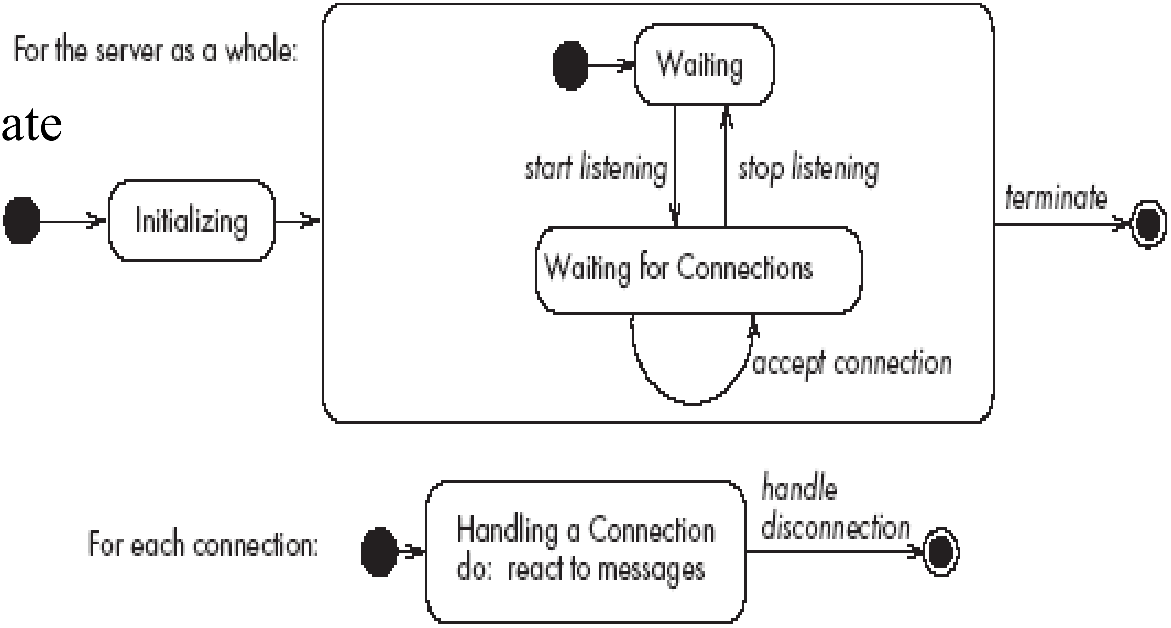
A server program communicating with two client programs



© Lethbridge/Laganière 2005

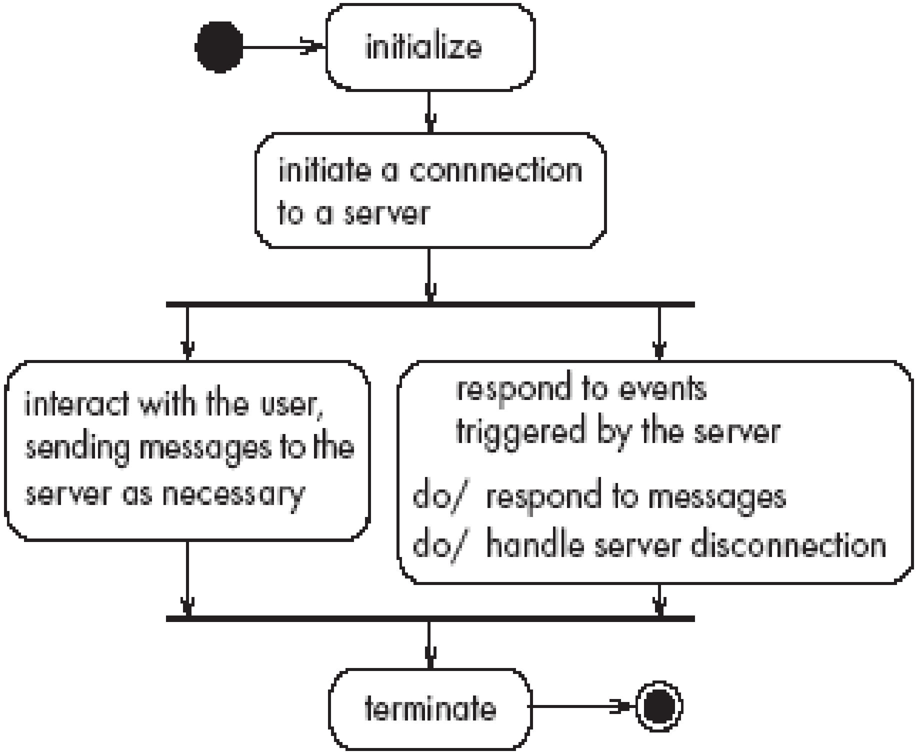
Activities of a server

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



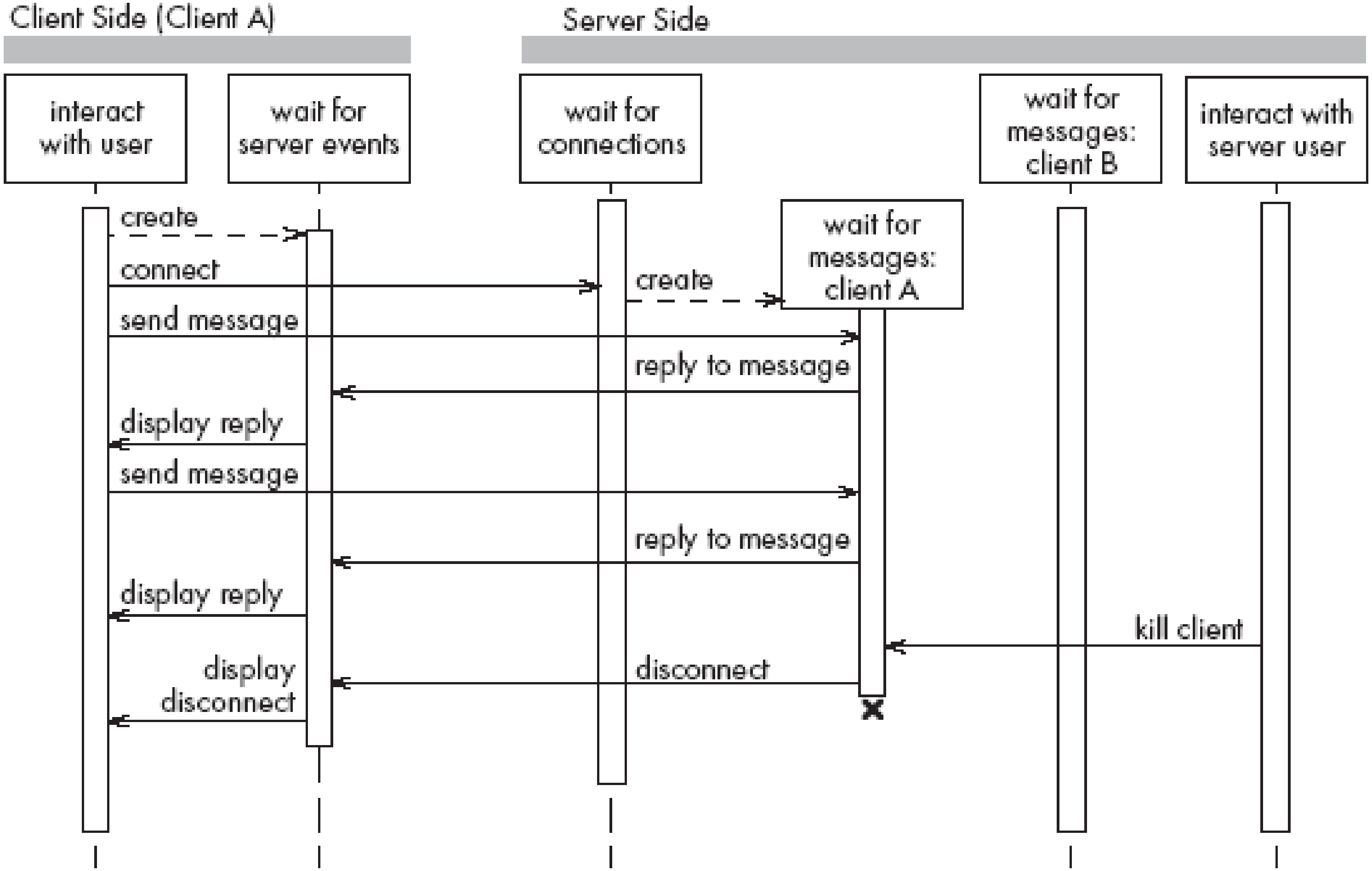
Activities of a client

1. Initializes itself
2. Initiates a connection
3. Sends messages
4. Handles the following types of events originating from the server
   - responds to messages
   - 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

© Lethbridge/Laganière 2005

Threads in a client-server system



© Lethbridge/Laganière 2005

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

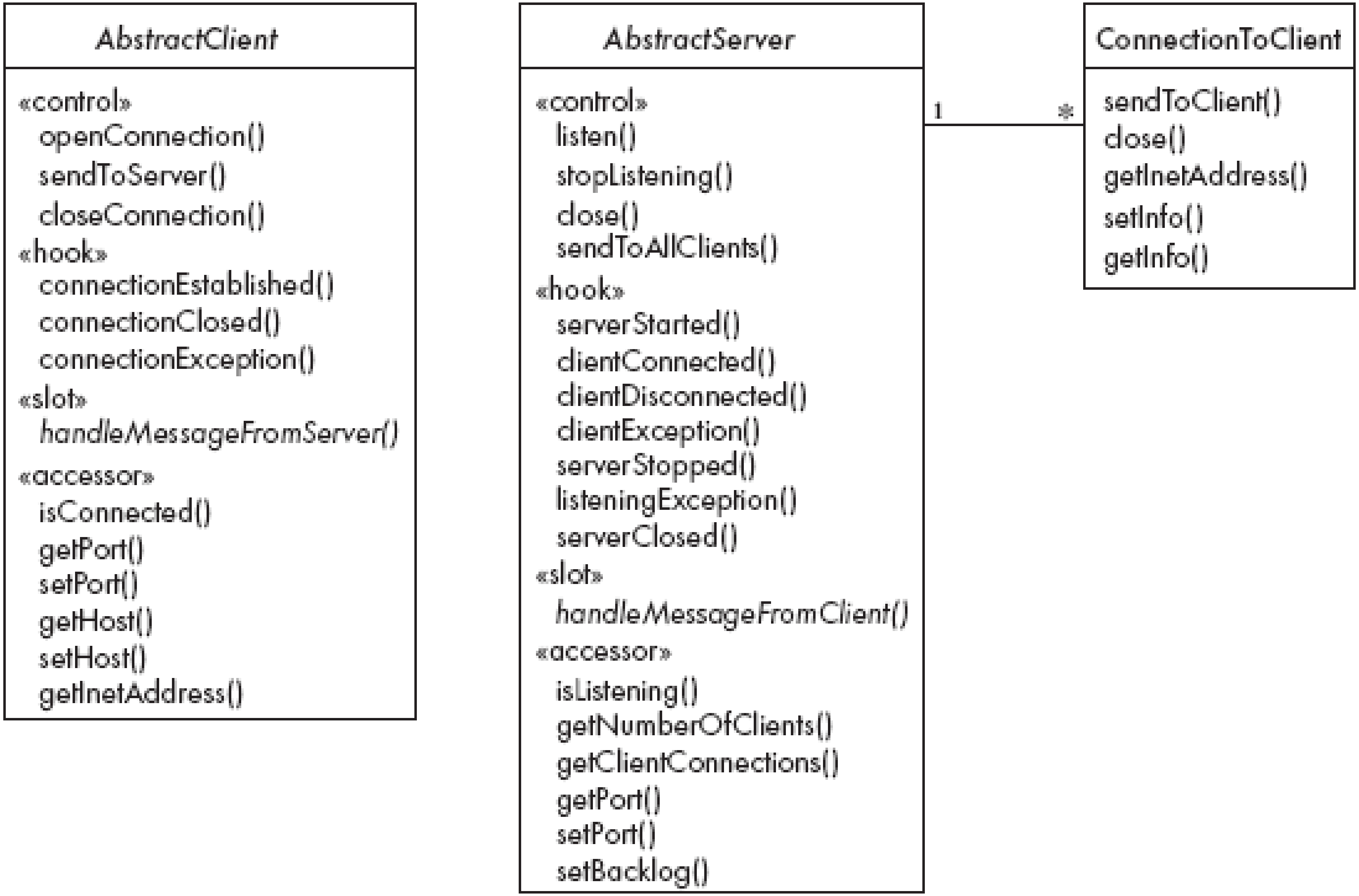
A host has an *IP address* and a *host name*

- 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

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*

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()

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)

3.7 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

3.8 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

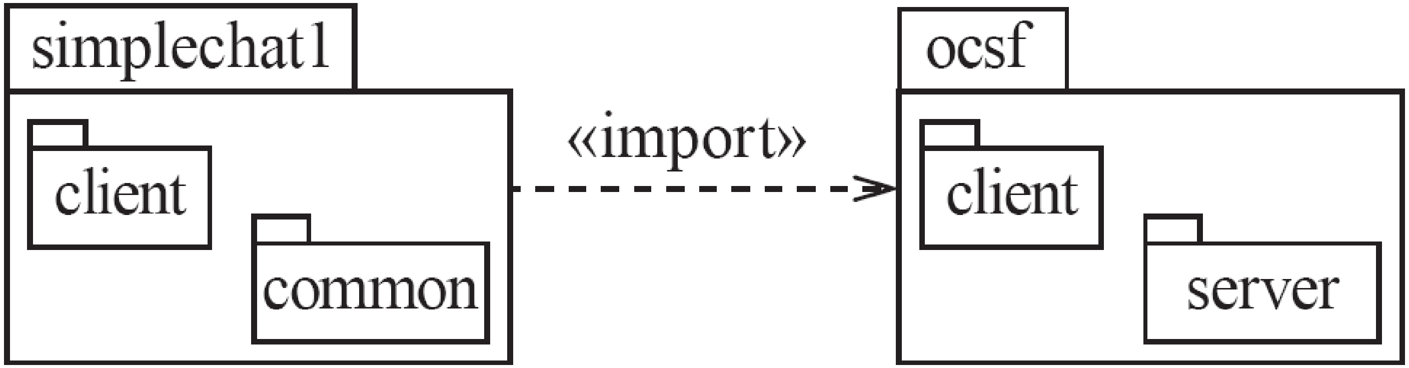
Using AbstractServer and ConnectionToClient

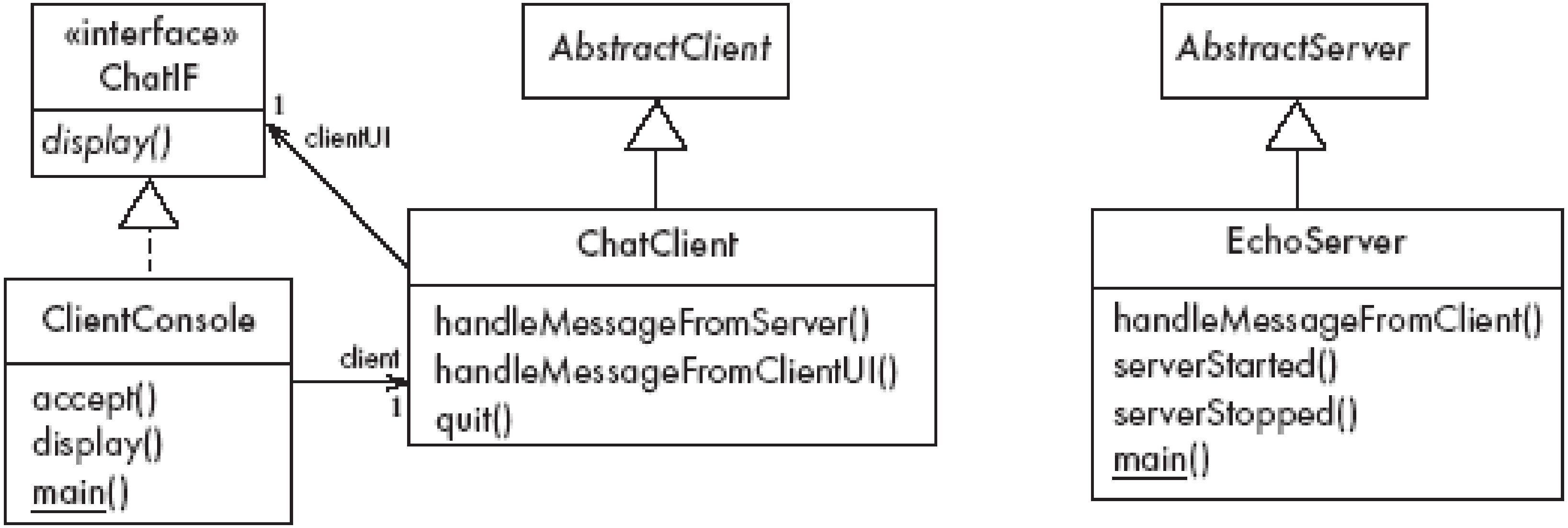
- Create a subclass of **AbstractServer**
- Implement the slot method **handleMessageFromClient**
- Write code that:
  - Creates an instance of the subclass of **AbstractServer**
  - Calls the **listen** method
  - Sends messages to clients, using:
    - the **getClientConnections** and **sendToClient** 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:

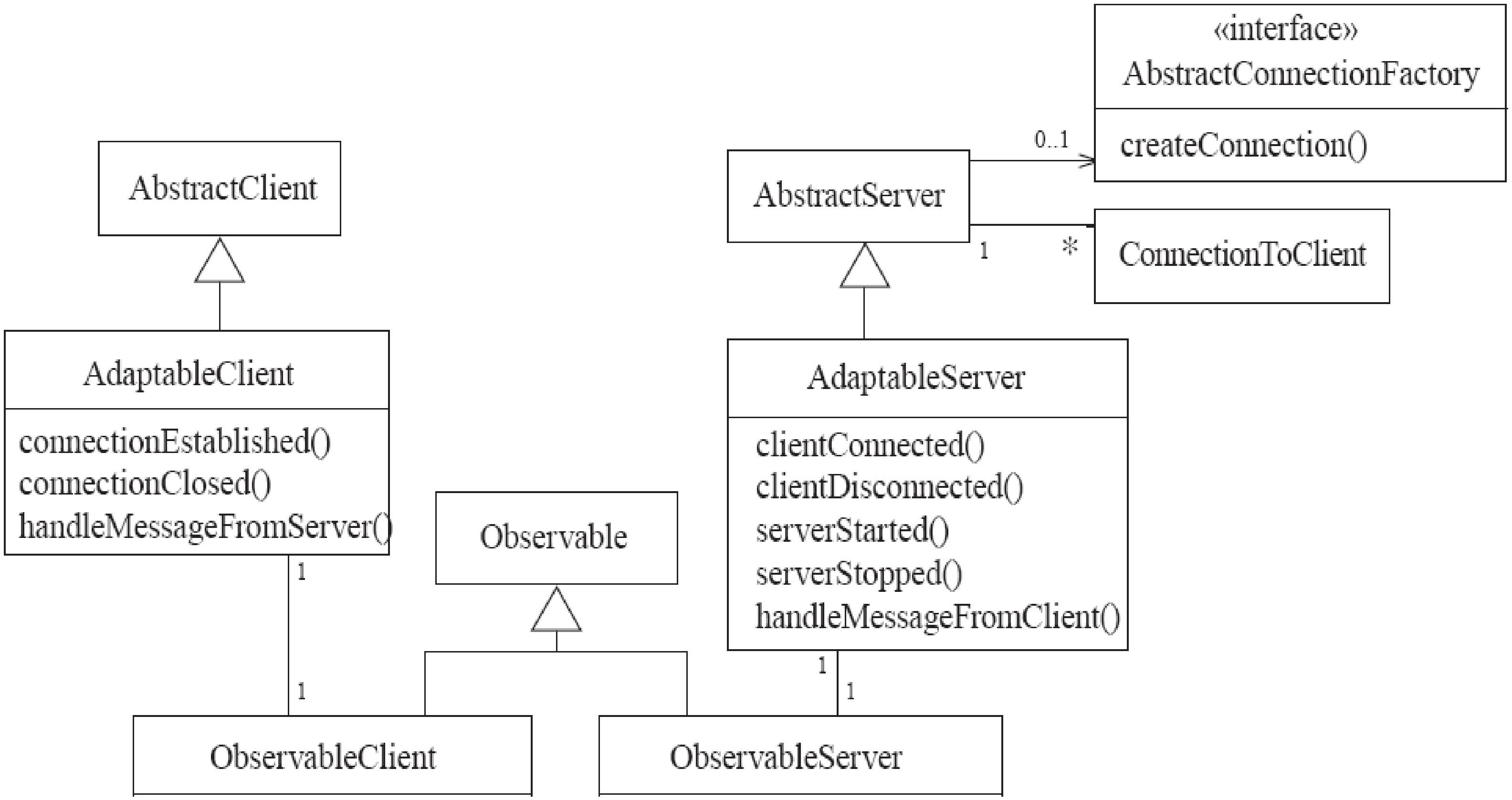
- **ChatClient**
  - —A subclass of **AbstractClient**
  - —Overrides **handleMessageFromServer**
    - This calls the **display** method of the user interface
- **ClientConsole**
  - —User interface class that implements the interface **ChatIF**
    - Hence implements **display** which outputs to the 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

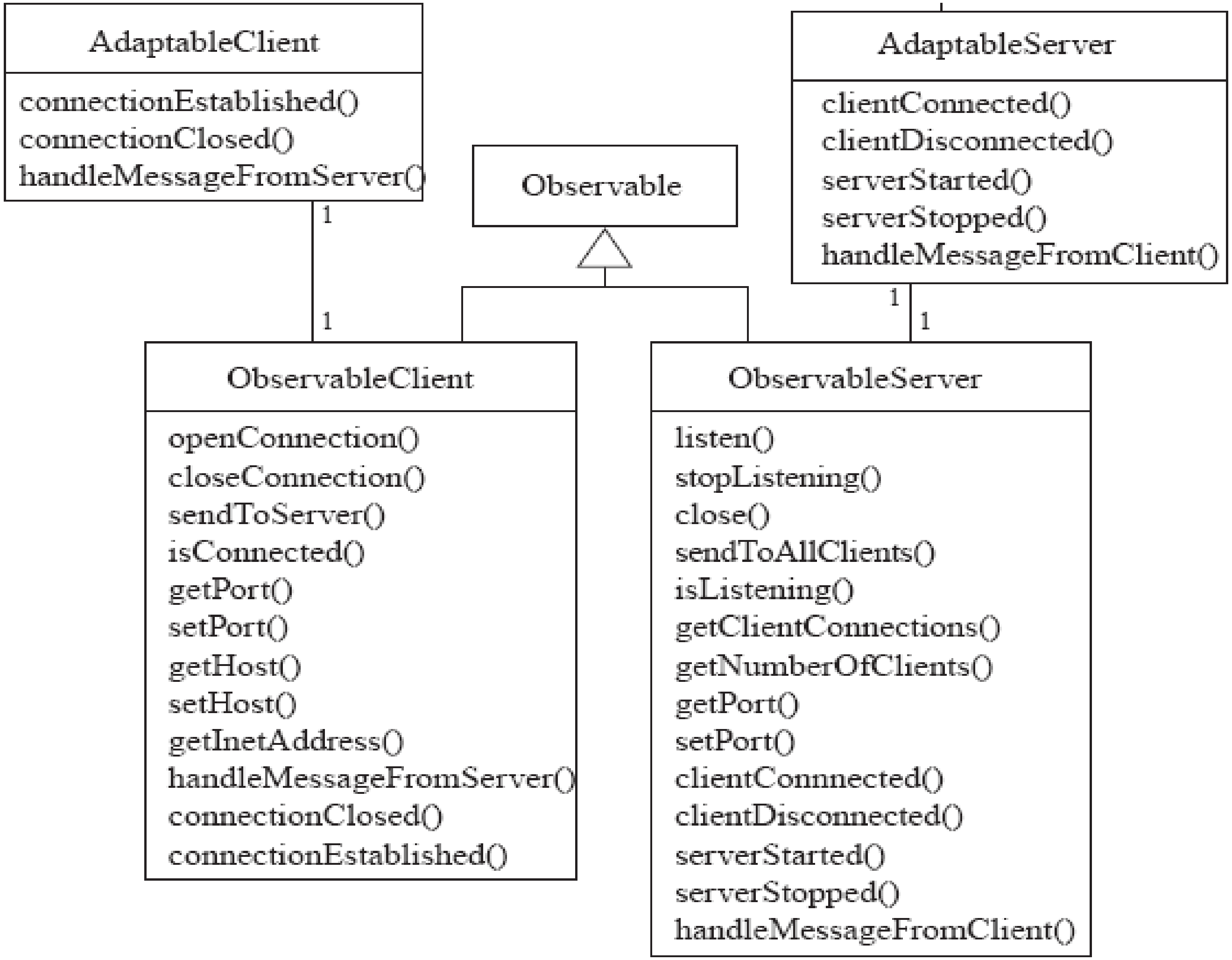
```
public void handleMessageFromClientUI(String message)
{
    try
    {
       sendToServer(message);
    }
    catch(IOException e)
    {
        clientUI.display (
           "Could not send message. " +
           "Terminating client.");
       quit();
    }
}

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

6.14 Detailed Example: The Observable layer of OCSF



The Observable layer of OCSF (continued)


AdaptableClient
connectionEstablished()
connectionClosed()
handleMessageFromServer()
Observable
AdaptableServer
clientConnected()
clientDisconnected()
serverStarted()
serverStopped()
handleMessageFromClient()
1
1
1
1
ObservableClient
openConnection()
closeConnection()
sendToServer()
isConnected()
getPort()
setPort()
getHost()
setHost()
getInetAddress()
handleMessageFromServer()
connectionClosed()
connectionEstablished()
ObservableServer
listen()
stopListening()
close()
sendToAllClients()
isListening()
getClientConnections()
getNumberOfClients()
getPort()
setPort()
clientConnnected()
clientDisconnected()
serverStarted()
serverStopped()
handleMessageFromClient()