Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapter 3:

Basing Software Development on Reusable Technology

3.1 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

3.3 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**

Shape Drawing Framework

```
// Framework
interface Shape {
    void draw();
}
class ShapeDrawer {
    static void drawAll(Shape[] shapes) {
        for (Shape s : shapes) {
            s.draw();
        }
    }
}
// Application 1: Circle
class Circle implements Shape {
    public void draw() {
        System.out.println("Drawing a Circle");
    }
}
// Application 2: Rectangle
class Rectangle implements Shape {
    public void draw() {
        System.out.println("Drawing a Rectangle");
    }
}
public class ShapeDemo {
    public static void main(String[] args) {
        Shape[] shapes = { new Circle(), new Rectangle() };
        ShapeDrawer.drawAll(shapes);
    }
}
```

Message Sending Framework

```
// Framework
interface MessageSender {
    void send(String message);
}

class SenderUtil {
    static void sendAll(MessageSender[] senders, String message) {
        for (MessageSender s : senders) {
            s.send(message);
        }
    }
}

// Application 1: Email
class EmailSender implements MessageSender {
    public void send(String message) {
        System.out.println("Sending Email: " + message);
    }
}

// Application 2: SMS
class SMSSender implements MessageSender {
    public void send(String message) {
        System.out.println("Sending SMS: " + message);
    }
}

public class MessageDemo {
    public static void main(String[] args) {
        MessageSender[] senders = { new EmailSender(), new SMSSender() };
        SenderUtil.sendAll(senders, "Hello Framework!");
    }
}
```

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

Framework is intrinsically incomplete (slots).

- MiniBatchFramework calls load() and process() but doesn't implement them.
- These are required slots: The framework can't do useful work without an application supplying them.

```
protected abstract List<String> load();
protected abstract List<String> process(List<String>);
```

Some functionality is optional (hooks/extension points).

- onBeforeProcess(), onAfterProcess() default to no-op.
- save(result) has a default implementation (print to console), but apps may override it.

```
@Override protected void onBeforeProcess() { /* custom */ }
@Override protected void save(List<String> result) { /* custom */ }
```

Developers use services provided by the framework (API).

- The API here is the set of services exposed via FrameworkServices: log() and version().

```
api().log().info("message");
api().version();
```

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 and there are often many Interfaces

Example:

- A framework for payroll management
- A framework for frequent buyer clubs
- A framework for university registration
- A framework for e-commerce web sites

1. A Framework for Payroll Management

- Purpose: Automate salary calculation, tax deductions, and payment processing.
- What the framework provides (API): Classes like Employee, PayrollProcessor, TaxCalculator. Methods like calculatePay(), generatePayslip().
- Slots (abstract classes/interfaces): Employee might be abstract because different employees (hourly, salaried) calculate pay differently.
- Hooks (optional): A method like applyBonus() could be overridden if a company wants custom bonus logic.

Example use: A company plugs in its own tax rules and employee types.

2. A Framework for Frequent Buyer Clubs

- Purpose: Manage loyalty programs, reward points, and discounts.
- What the framework provides: Classes like Member, RewardCalculator, DiscountPolicy. API methods like addPoints(), redeemPoints().
- Slots: RewardCalculator could be an interface so businesses can define their own point rules.
- Hooks: Optional method to send promotional emails after a purchase.

Example use: A retail store customizes how points are earned and redeemed.

3.4 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

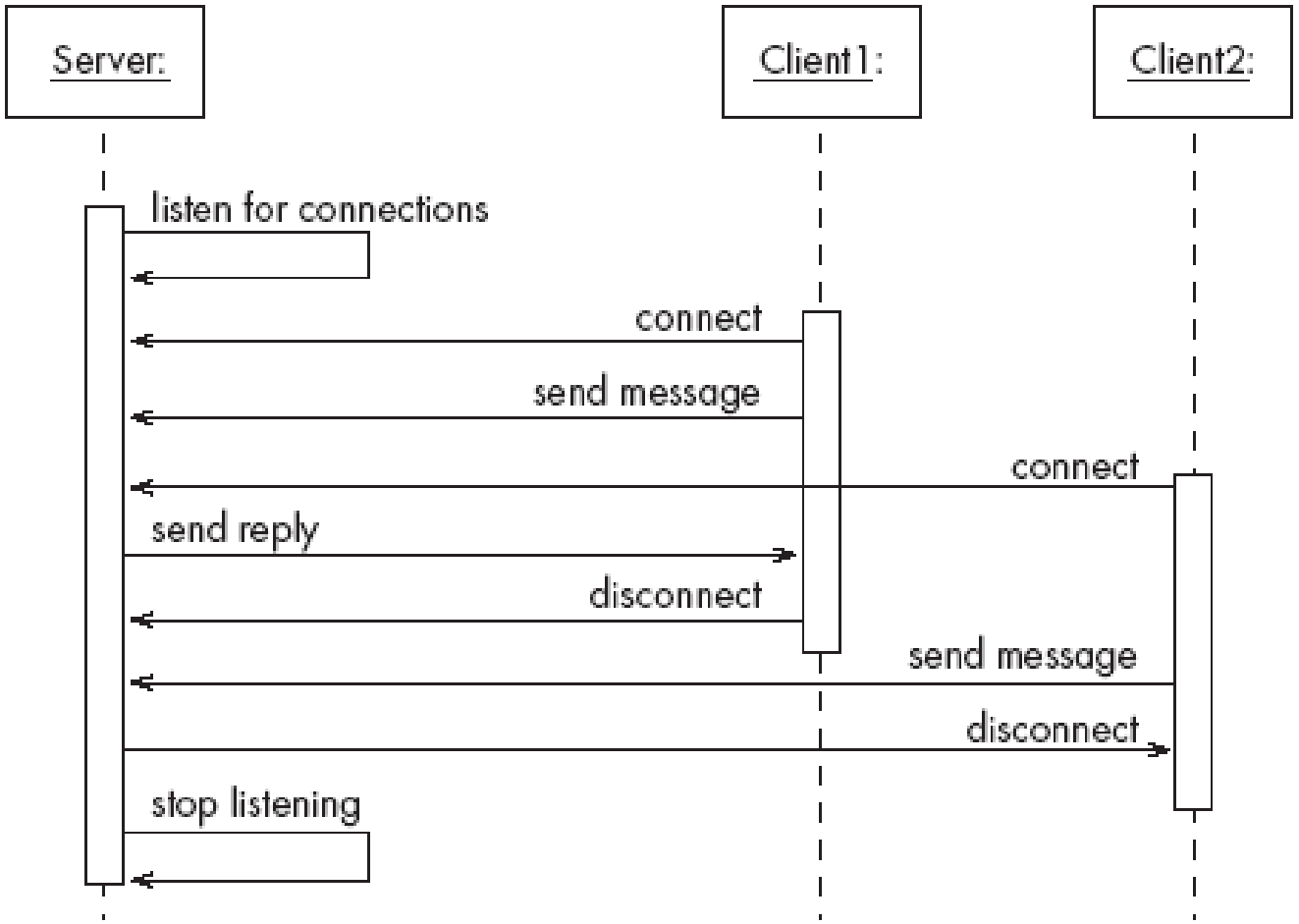
Client

- A program that accesses a server (or several servers) *to obtain services*
- A server may be accessed by many clients simultaneously

Example of client-server systems

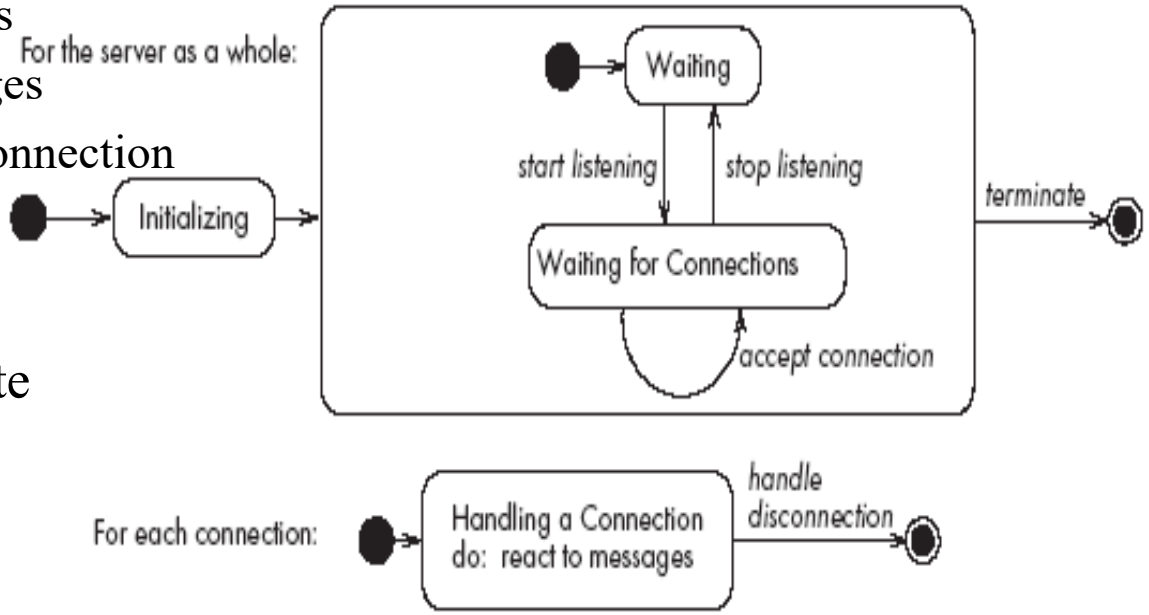
- The World Wide Web
- Email
- Network File System
- Transaction Processing System
- Remote Display System
- Communication System
- Database System

A server program communicating with two client programs



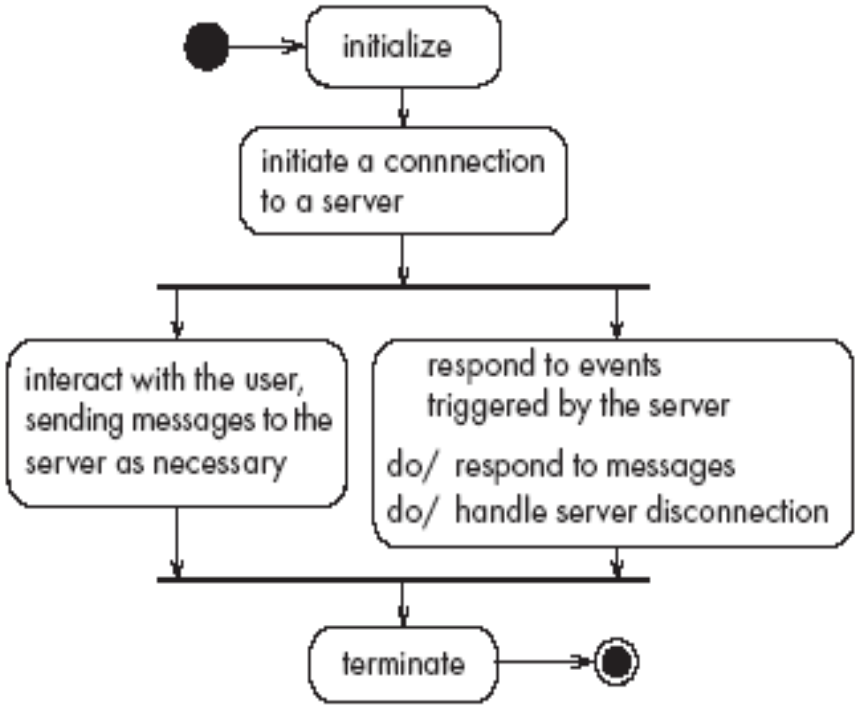
Activities of a server

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

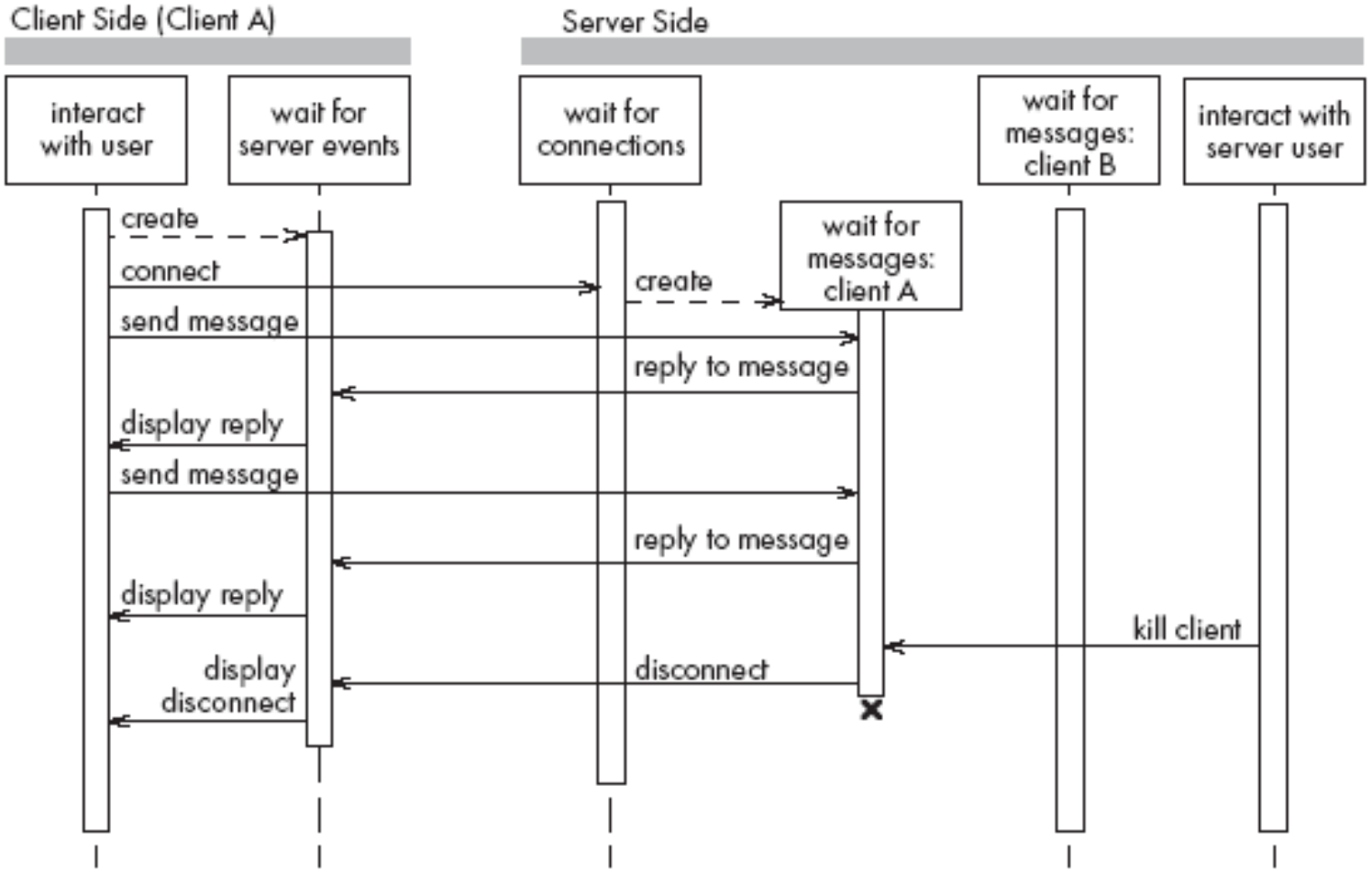


Activities of a client

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



Threads in a client-server system



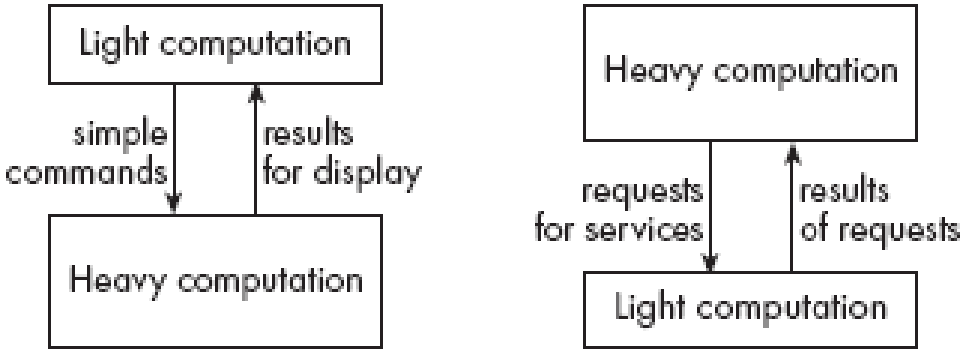
Thin- versus fat-client systems

***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



Communications 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

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

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*
- There is a choice about where to keep data:
  - —All the *data can be kept centrally* at the server
  - —D*ata can be distributed* among many different 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

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**

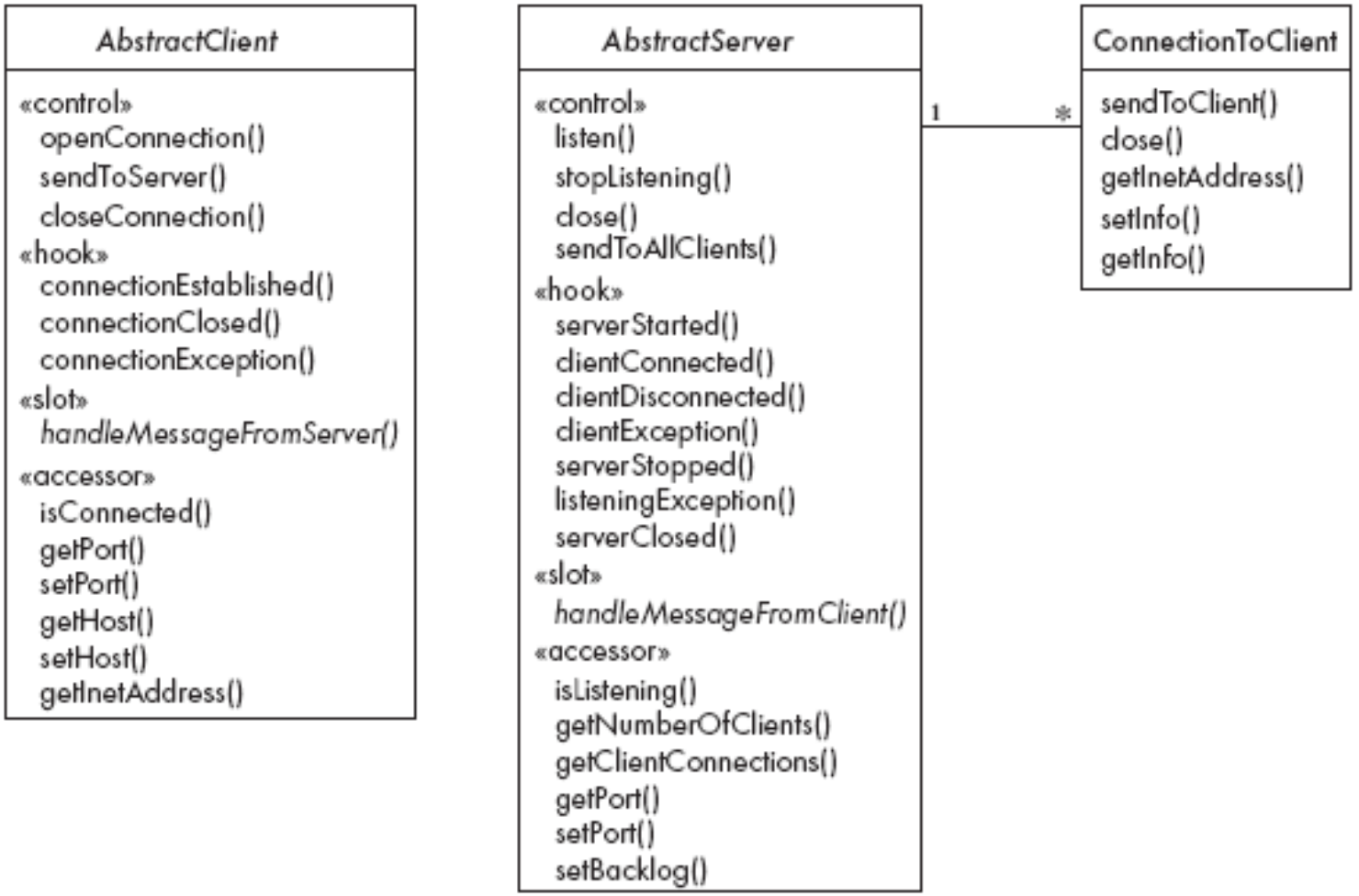
```
output.writeObject(msg);
msg = input.readObject();
```

Raw InputStream/OutputStream → You read/write bytes. You must define your own message format (framing) and interpret bytes yourself. You are responsible for Framing: How do you know where one message ends and the next begins? (length prefix, delimiter, fixed size, etc.)

DataInputStream/DataOutputStream → You read/write Java primitive types (int, double, etc.) in a portable binary format (big-endian). Still your protocol, but type-safe for primitives.

ObjectInputStream/ObjectOutputStream → You read/write full Java objects via serialization (includes class metadata, object graphs). Java-specific.

3.6 The Object Client-Server Framework (OCSF)



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

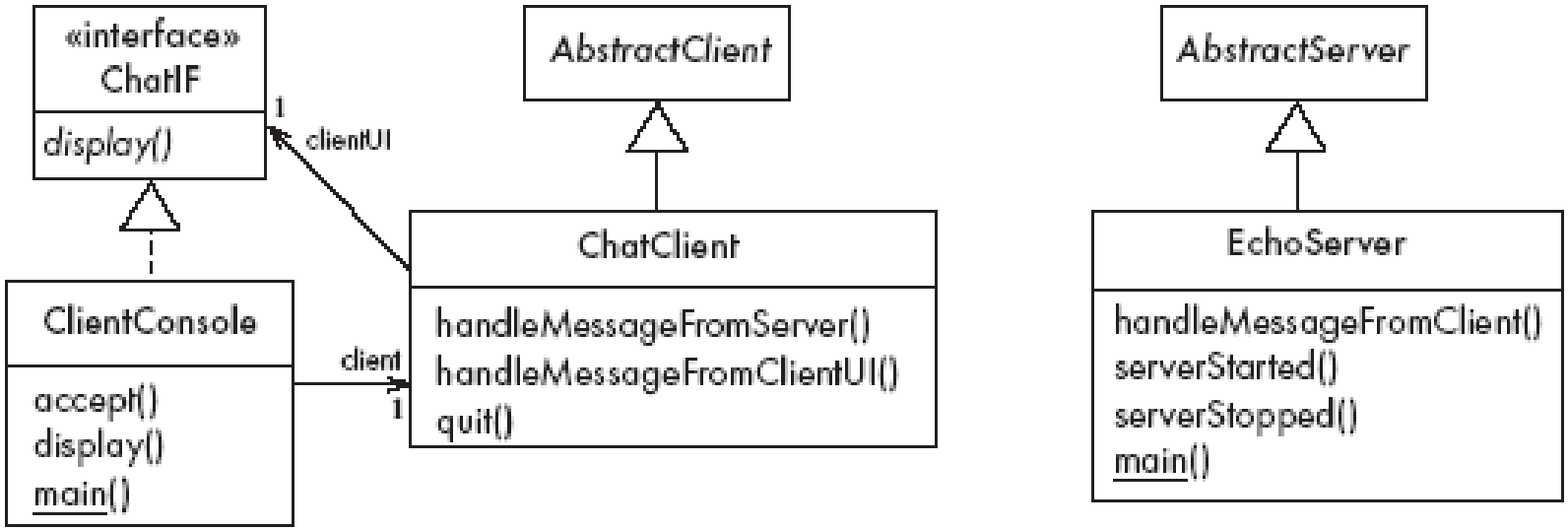
Using AbstractServer and ConnectionToClient

- 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

3.9 An Instant Messaging Application: SimpleChat



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

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**

3.10 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*