CS 580 Client-Server Programming
Spring Semester, 2005
Comments on Assignment 7
Contents

Inheritance Versus Composition

```
public class MessageReader extends UpToInputStream {
```

Verses

```
public class MessageReader {
    UpToInputStream in;

    public MessageReader(InputStream input) {
        in = new UpToInputStream( input);
    }
```

Inheritance

What should I use as a super class?

A has a B

Indicates that an instance variable of A is an instance of B

A is a B

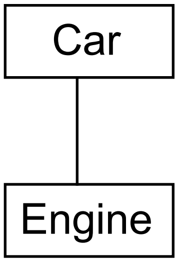
A is a type of B

Indicates that A is a subclass of B

A car has an engine, so car object contains an engine object

A BinarySearchTree has nodes, so it has instance variables left and right

Common Mistakes
Engine Subclass of Car



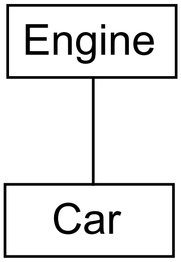
Car

Engine

Using a has-a relation for inheritance

- A car has-an engine
- An engine is not a type of car

Car subclass of Engine



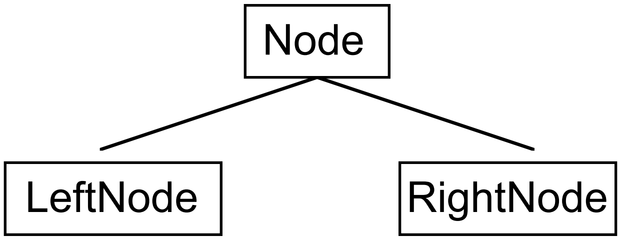
Engine

Car

“I need access to engine methods in the car class and now I have it.”

Roles Verses Classes

2.11 Be sure the abstractions you model are classes and not simply the roles objects play



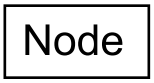
Node

LeftNode

RightNode

```
public class BinarySearchTree {
    LeftNode left;
    RightNode right;
```

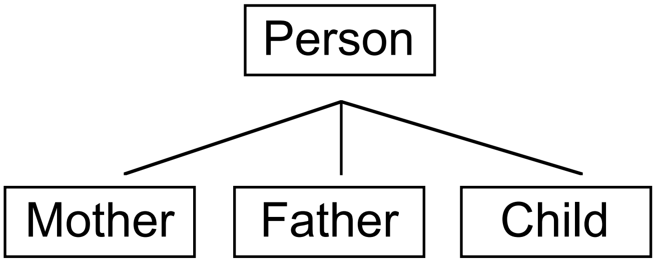
Verses



Node

```
public class BinarySearchTree {
    Node left;
    Node right;
```

More Roles



```
Mother mother = new Mother();
Father father = new Father();
etc.
```

Person

```
Person mother = new Person();
Person father = new Person();
etc.
```

Separate Accepting from Process Requests

```
while (listening) {
    connection = serverSocket.accept();
    out = new MessageWriter( blah);
    in = new MessageReader( blah );
    Message fromClient = in.readMessage();
    if (!fromClient.isHandShake() ) {
        code to end connection
    }
    else {
        code to process handShake;
        while (!(fromClient = in.readMessage()).isEndConnection() ) {
            lots more code
```

How do you test the above?
How do you introduce threads?

```
while (listening) {
    connection = serverSocket.accept();
    ClientHandler client = new ClientHandler( connection);
    client.start();
}
```

Extreme Programming (XP) & Planning

It is hard to know how to design new things

XP tells us to design and code for what we need now

The Simplest Possible Design

The right design at any given time is the one that

- Runs all the tests
- Has no duplicate logic
- State every intention important to the programmers
- Has the fewest possible classes and methods

This works if you:

- Write tests (first)
- Refactor your code as you add new functionality

Don't Hide Fields between Methods

A class can span many pages

Don't make your reader search pages to find field declarations

Place all fields either

- Before all methods
- After all methods

```
public class Foo {
    int firstField;
    public void bar() {
        blah:
    }
    int secondField:
    public void run() {
        blah;
    }
    float thirdField;
    etc.
}
```

Thread Priorities

Thread handling network code should have high priority

Code handling normal actives should be lower priority

Thread handling accept should have higher priority than that handling client connections

Don't Hide logic of Code

Keep code in a method at same level

```
public class Server {
    public void run() {
        while (isRunning) {
            client = socket.accept();
            client.setReceiveBufferSize (blah);
            client.setSoTimeout( balh);
            in = new TorrentReader(client.getInputStream());
            out = new TorrentWriter(client.getOutputStream());
            getHandshake();
        }
    }

private void getHandshake() {
    Message fromClient = in.readMessage();
    if (!fromClient.isHandshake() ) {
        blah
        blah
        blah
    }
    else {
        Message toClient = new Handshake();
        out.write( toClient);
        service();
    }
```

```
private void service() {
    now handled the clients request.

}
```

Some attempt to show logic

```
public class Server {
    public void run() {
        while (isRunning) {
            client = socket.accept();
            setNetworkParameters(client);
            setClientIOStreams(client);
            if (handshakeIsSuccessful())
                serviceClientRequest();
        }
    }
```

Naming Conventions

```
inReader.getMessage();
```

getXXX() returns a value
What is going on above?

Class Names

Classes are things - names normally are nouns

Subclasses names use adjectives to refine name

List	Component	InputStream
AbstractList	Button	FilterInputStream
ArrayList		

Verbs normally indicate actions or methods

```
public class CreateMetaData { }
```

Fields Verses Arguments

```
public class Server {
    private ServerSocket serverSocket;
    private Socket client;

    public void run() {
        client = serverSocket.accept();
        byte[] message = readMessage();
        blah
    }

private byte[ ] readMessage() {
    UpToFilterInputStream input =
        new UpToFilterInputStream(
            new BufferedInputStream(
                client.getInputStream()));
    byte [] readBytes = input.readUpTo();
    return readBytes;
}
```

Using client as a field

- Makes the code harder to understand
- Does not allow multiple connections

Keep Separate Concerns Separate
Methods should do one (conceptual) thing

```
public void run( int port ) {
    Handler textLog = new FileHandler("logfile.txt", true);
    textLog.setFormatter( new SimpleFormatter() );
    textLog.setLevel(Level.All);
    log.addHandler( textLog);

    server = new ServerSocket( port);
    log.info( blah);
    while (true) {
        Socket client = server.accept();
        blah;
```

```
public class Server {

    private static Logger log;

    static {
        Handler textLog = new FileHandler("logfile.txt", true);
        textLog.setFormatter( new SimpleFormatter() );
        textLog.setLevel(Level.All);
        log = Logger.getLogger( "Server");
        log.addHandler( textLog);
    }

public Server( int port ) {
    this.port = port;
}

public void run() {
    server = new ServerSocket( port);
    log.info( blah);
    while (true) {
        Socket client = server.accept();
        blah;
```

Keep Separate Concerns Separate

Servers do many different type of things

Log
Accept client connections
Handle multiple clients
Read messages
Parse messages
Send messages
Handle threads
Save & retrieve data

Keep separate things separate

```
public class TorrentData {
   public MetaData getFile(String id) { blah; }

   public byte[] getPiece(String fileId, int pieceIndex) { }

   public void setPiece(String fileId, int pieceIndex,
      byte[] peice) { }

   public ArrayList search( String name) { }

   etc.
```

Can implement & test independent of network code

Can change later to database

Replace case statements with Polymorphism

```
public void writeMessage( Object message) {
    if (message instanceof HandShake) {
        HandShake handShake = (HandShake) message;
        10 lines of code to extract data out of handShake
        blah
        blah
        blah
        blah
        blah
        blah
        blah
        blah
        writeMessage(extractedData);
    }
    else if(message instanceof EndConnection) {
        EndConnection end = (EndConnection) message;
        8 lines of code to extract data out of end
        blah
        blah
        blah
        blah
        blah
        blah
        writeMessage(extractedData);
    }

    continue for 3.5 pages
    }
}
```

Using Polymorphism

```
public void writeMessage( Object message) {
        writeMessage(message.toBytes());
}
```

The removed lines go to each individual Message class

- Makes it easier to test code
- Keeps operations with data
- Reduced dependencies between classes

Replace case statements with Polymorphism

```
public void run() {
    do {

        BittorrentMessage request = in.readMessage();

        switch (request.id() ) {
            case SEARCH_REQUEST:
                processSearchRequest( request);
                break;
        case REQUEST:
                sendMessagePiece( request );
                break;
        etc.

        default:
            sendErrorMessage();
            break;
        }

    } while (!request.isEndConnection() );
}
```

```
public void run() {
    do {
        BittorrentMessage request = in.readMessage();
        request.processRequest( needed data);
        }
    } while (!request.isEndConnection() );
}
```