Designing classes (Spacebook)
Lecture 10

Waterford Institute of Technology

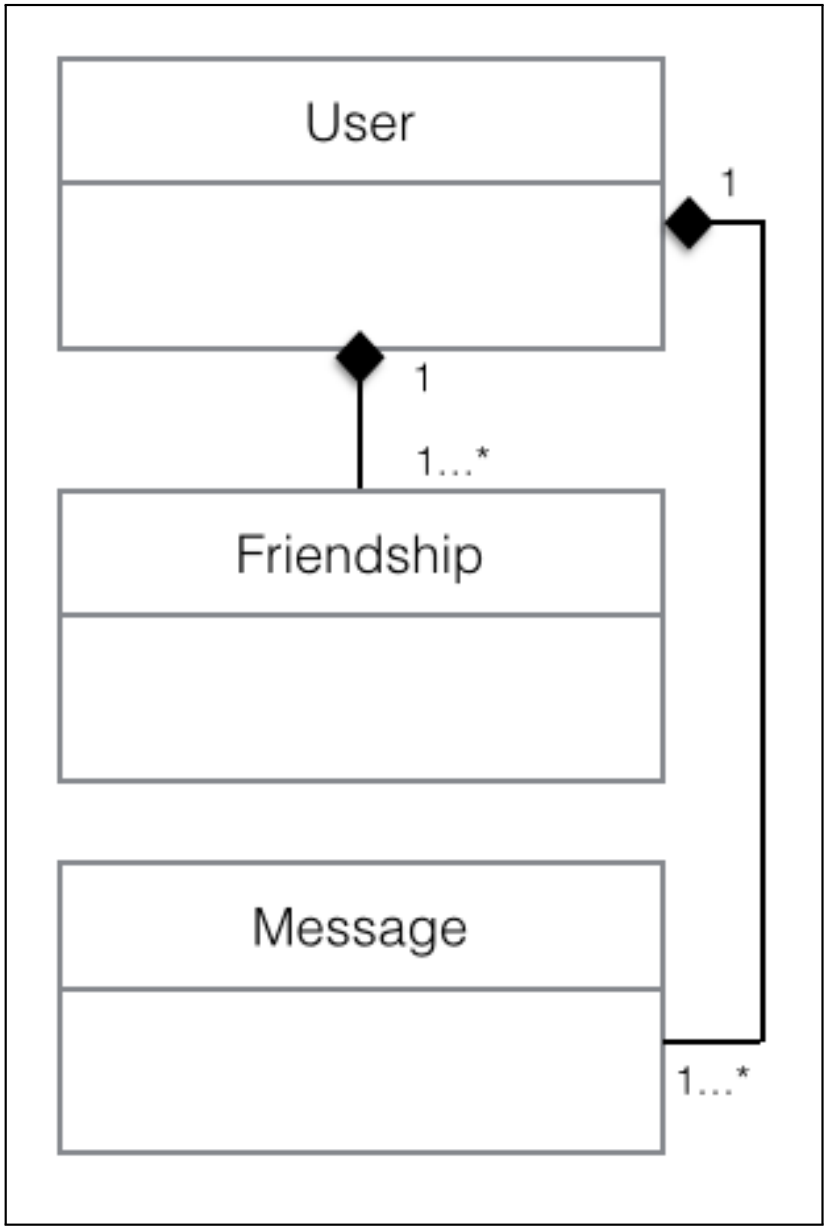
March 20, 2014

John Fitzgerald

Class design

Good class design facilitates

- Production robust code
- Reduction risk of error
- Easier maintenance
- Expansion
- Maximizing software's value



Class design rules

Outline Structure

Begin with list variables ordered thus:

- *public static* constants
- *private static* variables
- *private* instance variables

Next are constructors

- Begin with default
- Remaining ordered by number arguments

Then methods in this order

- *public* methods
- Support *private* methods

Rule 1: Classes should be small

```
public class User {
    public static int MAX_NMR_USER = 99;
    private static int MIN_CONTRIB = 10;
    private String name;
    //Constructors
    public User(){}
    public User(String name) {
        this.name = name;
    }
    //Methods
    public String getName() {
        return qualifiedName();
    }
    private String qualifiedName() {
        return "Spacebook member "+name;
    }
}
```

Cohesion

Cohesion: A measure how closely related module members are

- Classes
- Their methods
- Functionality each method

A class should be

- Reponsible for single task
- Task should be logical entity

A method should be

- Responsible for one task

```
public class LowCohesion {
    private User user;
    private Friendship friendship;
    private Donation donation;
    private WITPress blog;
    private BIABank account;
    private String userCollege;
    private String homeAddress;
    public User(User user) {
        this.user = user;
    }
    public void setBlog(WITPress blog) {
        this.blog = blog;
    }
    //methods that use fields
    ...
}
```

Example low cohesion class

Cohesion

Cohesive classes - high cohesion desirable

- Have small number instance variables
- Each method should manipulate one or more of these

```
public class Stack {
    private int topOfStack = 0;
    ArrayList<String> elements = new ArrayList<>();
    public int size() { return topOfStack;}
    public void push(String element) {
        topOfStack++;
        elements.add(element);
   }
   public String pop() {
        int element = elements.get(--topOfStack);
        elements.remove(topOfStack);
        return element; }
}
```

Example high cohesion class

Coupling

Tight coupling

Strong interconnectedness between classes

Public instance variables

Tight Coupling

```
public class Traveller
{
    Car car = new Car();
    public void beginJourney() {
        car.move();
    }
}

class Car {
    public void move() {...}
}
```

Tight Coupling

```
public class Traveller
{
    Plane plane = new Plane();
    public void beginJourney()
    {
        plane.move();
    }
}

class Plane {
    public void move() {...}
}
```

Coupling

Loose coupling

Weak interconnectedness between classes

Loose Coupling

```
public class Traveller
{
    Vehicle v;
    public setVehicle(Vehicle v) {
        this.v = v;
    }
    public void beginJourney() {
        v.move();
    }
}
```

Loose Coupling

```
public interface Vehicle
{
  public void move();
}

class Car implements Vehicle
{
  public void move() {...}
}
```

Spacebook v0

Initial naive design

- Low cohesion
  - Class responsible for several tasks
- Tight coupling
  - Publicly accessible instance variables

```
public User(String firstName, String lastName, String email, String password)
{
public String firstName;
public String lastName;
public String email;
public String password;
public String[] friends;
public String[] messagesTo;
public String[] messagesFrom;
}
```

Spacebook v0

Using fixed arrays

- Setting maximum array size in advance
  - A serious limitation
- Easy to exceed array bounds
  - No checks in place
- Difficult to manipulate
  - Add or remove elements

```
public static final int MAX_NUMBER_FRIENDS = 100;
public static final int MAX_NUMBER_MESSAGES = 100;
```

Spacebook v0

Out of bounds

- New String instance created
- No pre-assignment check *friends[]*
- If *numberFriends* not less than `MAX_NUMBER_FRIENDS`
  - Program crashes
- An example array bounds violation
  - No checks in place

```
public void befriend(String name)
{
  friends[numberFriends] = new String(name);
  numberFriends += 1;
}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
    at testJava.OutOfBounds.main(OutOfBounds.java:7)
```

Spacebook v1

Improved class design

Three classes identified in previous version class User

- User
- Friendship
- Message

```
//User
public String firstName;
public String lastName;
public String email;
public String password;
//Friendship
public String[] friends;
//Message
public String[] messagesTo;
public String[] messagesFrom;
```

Spacebook v1

User

User contains personal identification details only

- First and last name
- Email
- Plain text password

```
public class Use
{
  public String firstName;
  public String lastName;
  public String email;
  public String password;
}
```

Spacebook v1

Friendship

Friendship contains references to

- Initiator or source of friendship
- Target of friendship
- Initiator and target are *User* types

```
private User sourceUser;
private User targetUser;
```

```
//constructor
public Friendship(User sourceUser, User targetUser)
{
    this.sourceUser = sourceUser;
    this.targetUser = targetUser;
}
```

Spacebook v1

User has Friendship

User class has Friendship field

- Limited to a single Friendship instance
  - Later Spacebook versions to address this shortcoming
- Initiator and target are *User* types

```
public class User
{
    private String firstName;
    private String lastName;
    private String email;
    private String password;
    private Friendship friendships;
    ...
}
```

Spacebook v1

User | Friendship

User methods to manipulate Friendship

- Create a new friendship (befriend)
  - Only one friendship possible
  - Existing friendship dropped
- End a friendship (unfriend)

```
public void befriend(User friend)
    {
        friendships = new Friendship(this, friend);
    }

    public void unfriend()
    {
        friendships = null;
    }
```

Spacebook v2

Multiple Friendships

Replace private `Friendship friendships`

With private `ArrayList<Friendship> friendships`

- Allows many friendships
- Friendships easily added and removed

```
public class User
{
    private String firstName;
    private String lastName;
    private String email;
    private String password;
    private ArrayList<Friendship> friendships = new ArrayList<>();
    ...
}
```

Spacebook v2

Add friendship

Add a new friendship (befriend)

- *this* is reference to originator of friendship
- Target of friendship is actual parameter (argument)
- Using ArrayList method *add*

```
public void befriend(User friend)
{
    Friendship friendship = new Friendship(this, friend);
    friendships.add(friendship);
}
```

Spacebook v2

Remove friendships

End existing friendship (unfriend)

- Remove *friend* from list friendships
- Use for-each loop to traverse list
- Return true indicates removal succeeded

```
public boolean unfriend(User friend)
{
    for(Friendship friendship : friendships)
    {
        if(friendship.getTargetUser() == friend)
        {
            friendships.remove(friendship);
            return true;
        }
    }
    return false;
}
```

Spacebook v3

Messaging

Recall initial approach to messaging

```
public class User {
    public String[] messagesTo;
    public String[] messagesFrom;
}
```

New approach

- Design new class Message
- Replace String[] with list Message objects
- Create inbox and outbox to organize messages

```
public class User
    private ArrayList<Message> inbox = new ArrayList<>();
    private ArrayList<Message> outbox = new ArrayList<>();
    ...
 }
```

Spacebook v3

A Message class

Message class

- Stores the message text in String object
- Has references to User instances that identify
  - Originator of message (from)
  - Target of message (to)

```
public class Message
{
    private String messageText;
    private User from;
    private User to;
}
```

Spacebook v3

Manipulate Message class

- Current User writes message
- Invokes *sendMessage*
  - Passes message and target user as parameters
- Messages copied to both inbox and outbox

```
User homer = new User("Homer","Simpson","hs@simpson.com", "secret");
User barney = new User("Barney","Gumble","bg@gumble.com", "secret");
homer.sendMessage("barney", "oh what a good friend am I");
```

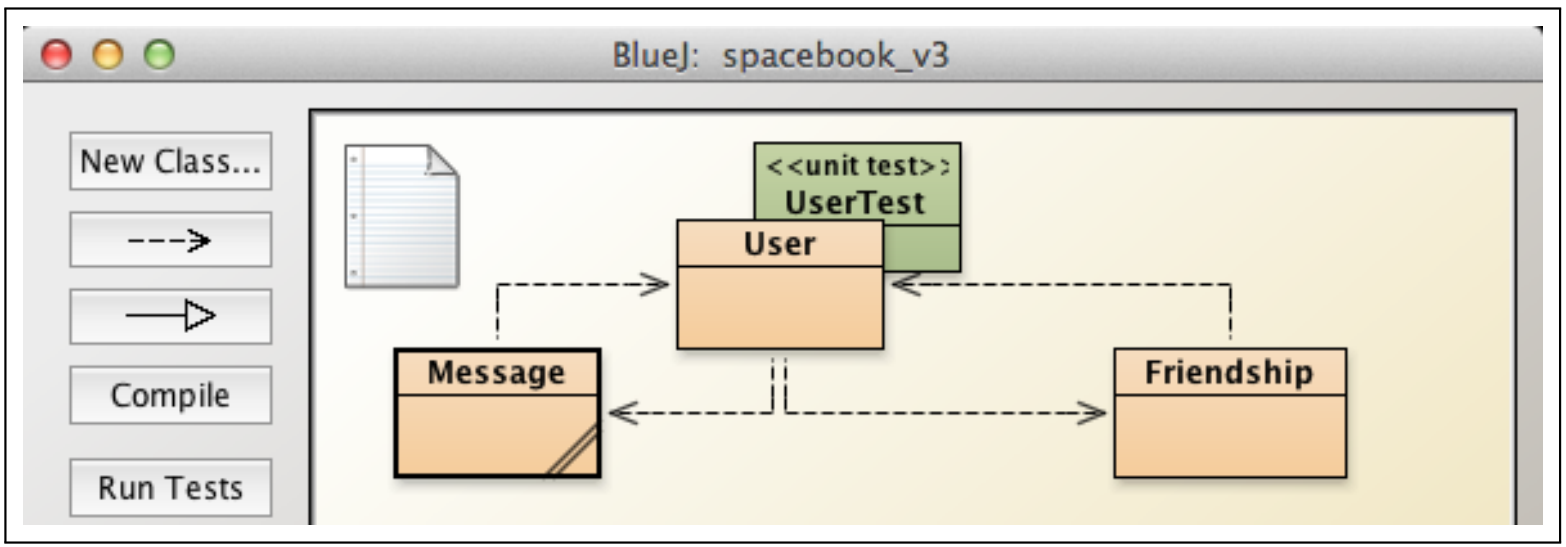
```
public void sendMessage(User to, String messageText)
{
    Message message = new Message(this, to, messageText);
    outbox.add(message);
    to.inbox.add(message);
}
```

Spacebook v3

Unit Test

BlueJ Unit testing

- Discussed in earlier lecture
- Create UserTest in BlueJ
- Could avail of BlueJ GUI to write test code
- Or write code manually



Spacebook v3

Unit Test User class

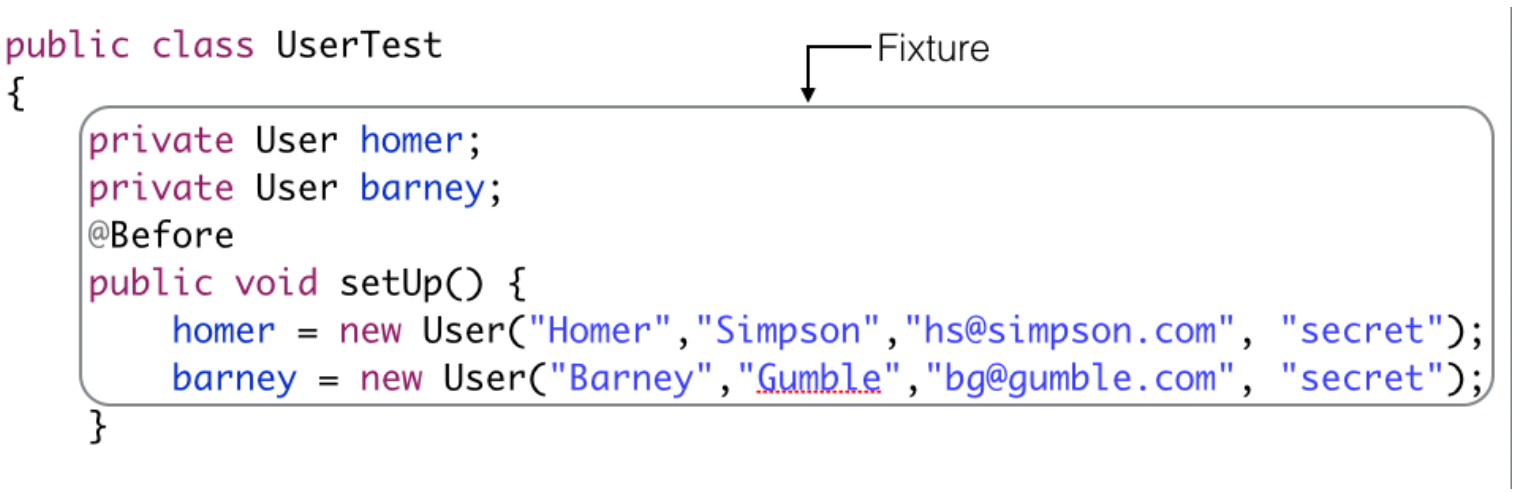
Example basic test

```
public class UserTest
{
    private User homer;
    private User barney;
    @Before
    public void setUp() {
        homer = new User("Homer","Simpson","hs@simpson.com", "secret");
        barney = new User("Barney","Gumble","bg@gumble.com", "secret");
    }
    @Test
    public void testMessages() {
        homer.sendMessage("barney", "oh what a good friend am I");
        barney.sendMessage(homer, "you gotta be kidding");
        int numberMsgsInBox = homer.getNumberInboxMsgs();
        assertsEqual(true, numberMsgsInBox, 1);
    }
}
```

Spacebook v3

Unit Test User class

Example unit test fixture




```
public class UserTest
{
    private User homer;
    private User barney;
    @Before
    public void setUp() {
        homer = new User("Homer","Simpson","hs@simpson.com", "secret");
        barney = new User("Barney","Gumble","bg@gumble.com", "secret");
    }
```

Spacebook v3

Unit Test User class

Example unit test method

Test method

```
@Test
public void testMessages() {
    homer.sendMessage("barney", "oh what a good friend ar
    barney.sendMessage(homer, "you gotta be kidding");
    int numberMsgsInBox = homer.getNumberInboxMsgs();
    assertsEqual(true, numberMsgsInBox, 1);
}
```