Software Design

Models, Tools & Processes

Alan Blackwell

Cambridge University
Computer Science Tripos Part 1a

But first:

Introduction to Java (with BlueJ)

Three lectures adapted from

Objects First with Java:
A Practical Introduction using BlueJ

First lecture

- using BlueJ
- objects
- classes
- methods
- parameters
- fields

Second lecture

- constructors
- assignment statements
- conditional statements
- collections
- loops
- iterators

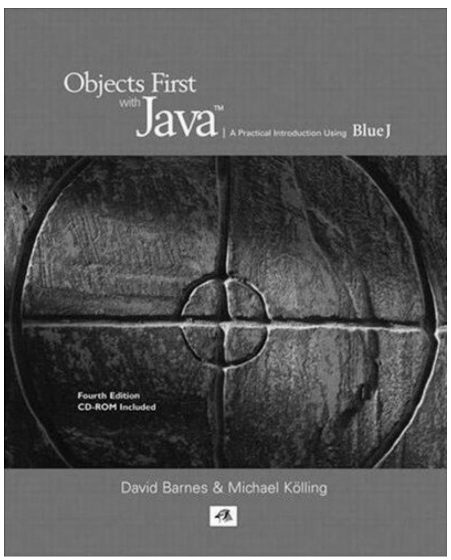
Third lecture

- arrays
- inheritance
- polymorphism
- data types
- equality and identity

Lectures 4-12

- Software engineering
- Unified modeling language
- Object-oriented design
- Programming style
- Development processes
- User-centred design
- Testing and debugging
- Configuration and releases
- Prototyping
- Agile development

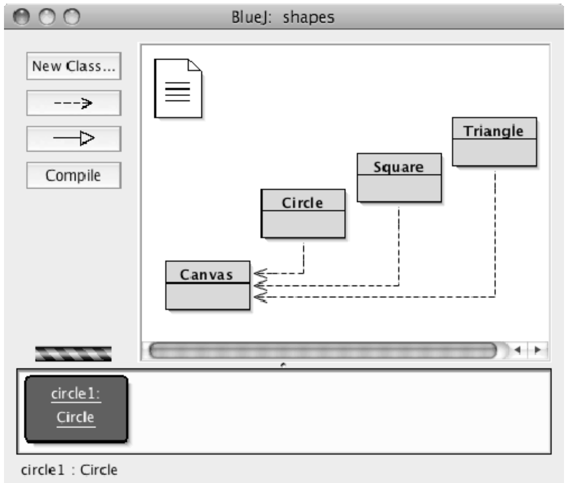
BlueJ book



BlueJ book

- (Many first editions still around Cambridge from 2004)

BlueJ Demo



Fundamental concepts

- object
- class
- method
- parameter
- data type

Objects and classes

- objects
  - represent ‘things’ from the real world, or from some problem domain (example: “the red car down there in the car park”)
- classes
  - represent all objects of a kind (example: “car”)

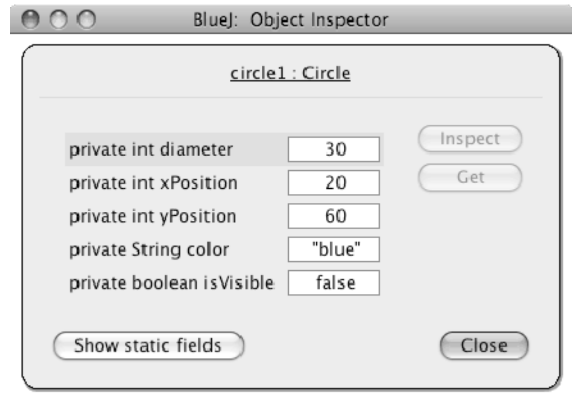
Methods and parameters

- objects have operations which can be invoked (Java calls them *methods*)
- methods may have *parameters* to pass additional information needed to execute
  - as with ML functions

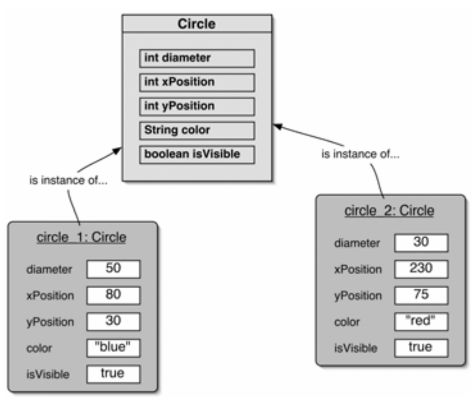
Other observations

- many *instances* can be created from a single class
- an object has *attributes*: values stored in *fields*.
- the class defines what fields an object has, but each object stores its own set of values (the *state* of the object)

State



Two circle objects



Source code

- Each class has source code (Java code) associated with it that defines its details (fields and methods).

Return values

- Methods may return a result via a “return value”.
  - another respect in which methods are like ML functions
  - in fact, many people may accidentally call Java methods ‘functions’
- Some ancestors of Java:
  - C and Pascal have ‘functions’
  - Smalltalk has ‘methods’
  - C++ has ‘member functions’

Understanding class definitions

Looking inside classes

Main concepts to be covered

- fields
- constructors
- methods
- parameters
- assignment statements
- conditional statements

BlueJ example: Ticket machine

Use BlueJ to explore the behavior of simple application:

- See the naive-ticket-machine project.
- Machines supply tickets of a fixed price.
  - How is that price determined?
- How is ‘money’ entered into a machine?
- How does a machine keep track of the money that is entered?

Ticket machine – internal view

- Interacting with an object gives us clues about its behavior.
- Looking inside allows us to determine how that behavior is provided or implemented.
- All Java classes have a similar-looking internal view.

Basic class structure

```
public class TicketMachine
{
    Inner part of the class omitted.
}
```

outer wrapper of TicketMachine

```
public class ClassName
{
    Fields
    Constructors
    Methods
}
```

contents of a generic class

Fields

- Fields store values for an object.
  - also called “instance variables”.
- Use *Inspect* in BlueJ to view an object’s fields.
- Fields define the *state* of an object.

```
public class TicketMachine
{
    private int price;
    private int balance;
    private int total;

    Constructor and
      methods omitted.
}
```

visibility modifier

type

variable name

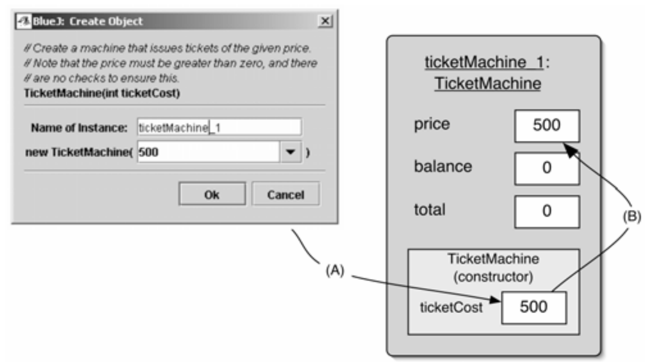
```
private int price;
```

Constructors

- Constructors initialize object state.
- They have the same name as their class.
- They store initial values into the fields.
- They often receive external parameter values for this.

```
public TicketMachine(int ticketCost)
{
    price = ticketCost;
    balance = 0;
    total = 0;
}
```

Passing data via parameters



Assignment

- Values are stored into fields (and other variables) via assignment statements:
  - *variable = expression;*
  - **`price = ticketCost;`**
- A variable stores a single value, so any previous value is lost.

Accessor methods

- Methods implement the behavior of objects.
- Accessors provide information about an object, getting access to its state
- Methods have a structure consisting of a header and a body.
- The header defines the method's *signature*.
  `public int getPrice()`
- The body encloses the method's statements.

Accessor methods

visibility modifier

return type

method name

```
public int getPrice()
{
    return price;
}
```

parameter list (empty)

return statement gets value based on object state

start and end of method body (*block*)

Mutator methods

- Have a similar method structure: header and body.
- But these are used to *mutate* (i.e., change) an object's state.
- Achieved through changing the value of one or more fields.
  - Typically contain assignment statements.
  - Typically receive parameters.

Mutator methods

return type (`void`)

visibility modifier

method name

parameter

```
public void insertMoney(int amount)
{
    balance += amount;
}
```

assignment statement

field being changed

Printing from methods

```
public void printTicket()
{
    // Simulate the printing of a ticket.
    System.out.println("##################");
    System.out.println("# The BlueJ Line");
    System.out.println("# Ticket");
    System.out.println("# " + price + " cents.");
    System.out.println("##################");
    System.out.println();

    // Update the total collected with the balance.
    total += balance;
    // Clear the balance.
    balance = 0;
}
```

Improving basic ticket machines

- Their behavior is inadequate in several ways:
  - No checks on the amounts entered.
  - No refunds.
  - No checks for a sensible initialization.
- How can we do better?
  - We need more sophisticated behavior.

Making choices

```
public void insertMoney(int amount)
{
    if(amount > 0) {
        balance += amount;
    }
    else {
        System.out.println(
               "Use a positive amount: ”
               + amount
          );
    }
}
```

Making choices

boolean condition to be tested - gives a *true* or *false* result

‘if’ keyword

actions if condition is true

```
if (perform some test) {
    Do the statements here if the
     test gave a true result
}
else {
    Do the statements here if the
     test gave a false result
}
```

‘else’ keyword

actions if condition is false

Local variables

- Fields are one sort of variable.
  - They store values through the life of an object.
  - They define the state of the object.
  - They are accessible from all methods of the class.
- Methods can include shorter-lived variables.
  - They exist only as long as the method is being executed.
  - They are only accessible from within the method.
  - They are not considered part of the object state.

Local variables

A local variable

No visibility modifier

```
public int refundBalance()
{
    int amountToRefund;
    amountToRefund = balance;
    balance = 0;
    return amountToRefund;
}
```

Object interaction

Creating cooperating objects

A digital clock

11:03

Abstraction and modularization

- **Abstraction** is the ability to ignore details of parts to focus attention on a higher level of a problem.
- **Modularization** is the process of dividing a whole into well-defined parts, which can be built and examined separately, and which interact in well-defined ways.

- Much more on this later in the course …

Modularizing the clock display

11:03 One four-digit display?

Or two two-digit displays? 11 03

Implementation - NumberDisplay

```
public class NumberDisplay
{
    private int limit;
    private int value;

    Constructor and
    methods omitted.

}
```

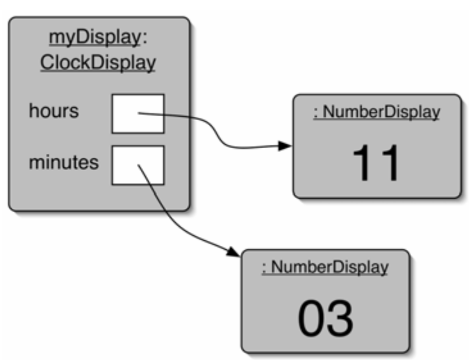
Implementation - ClockDisplay

```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;

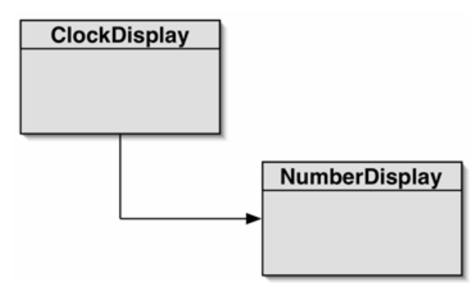
    Constructor and
    methods omitted.

}
```

Object diagram



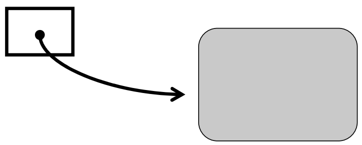
Class diagram



Primitive types vs. object types

`SomeObject obj;`

“object” type



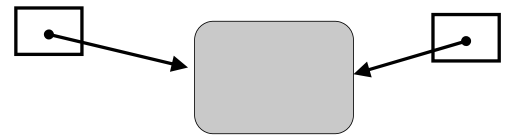
`int i;`

32

“primitive” type

Primitive types vs. object types

`SomeObject a;` **`SomeObject b;`**



`b = a;`

`int a;` **`int b;`**

32 32

Source code: NumberDisplay

```
public NumberDisplay(int rollOverLimit)
{
    limit = rollOverLimit;
    value = 0;
}

public void increment()
{
    value = (value + 1) % limit;
}
```

Source code: NumberDisplay

```
public String getDisplayValue()
{
    if(value < 10)
        return "0" + value;
    else
        return "" + value;
}
```

Objects creating objects

```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;
    private String displayString;

    public ClockDisplay()
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        updateDisplay();
    }
}
```

Method calling

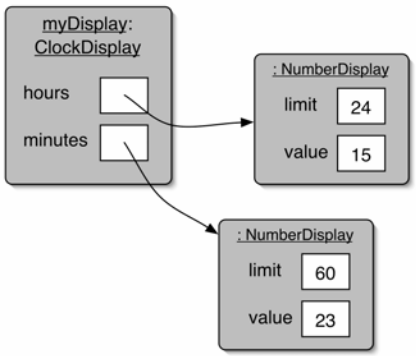
```
public void timeTick()
{
    minutes.increment();
    if(minutes.getValue() == 0) {
        // it just rolled over!
        hours.increment();
    }
    updateDisplay();
}
```

Internal method

```
/**
 * Update the internal string that
 * represents the display.
 */

private void updateDisplay()
{
    displayString =
        hours.getDisplayValue() + ":" +
        minutes.getDisplayValue();
}
```

ClockDisplay object diagram



Passing state between objects

in class NumberDisplay:

```
public NumberDisplay(int rollOverLimit);
```

formal parameter

in class ClockDisplay:

```
hours = new NumberDisplay(24);
```

actual parameter

Method calls

- internal method calls
  - Call using:

    `updateDisplay();`
  - Define using:

    `private void updateDisplay()`
- external method calls – call using:

  `minutes.increment();`

  object . methodName (parameter-list)

Review

- Class bodies contain fields, constructors and methods.
- Fields store values that determine an object's state.
- Constructors initialize objects.
- Methods implement the behavior of objects
- Fields, parameters and local variables are all variables.
- Fields persist for the lifetime of an object.
- Parameters are used to receive values into a constructor or method.

Review

- Local variables are used for short-lived temporary storage.
- Objects can make decisions via conditional (if) statements.
- A true or false test allows one of two alternative courses of actions to be taken.
- Objects can create other objects
- They can interact via method calls
- An object can call its own internal methods

Concepts

- abstraction
- modularization
- classes define types
- class diagram
- object diagram
- object references
- primitive types
- object types
- object creation
- internal/external method call

Grouping objects

Collections and iterators

1.0

Main concepts to be covered

- Collections
- Loops
- Iterators
- Arrays

The requirement to group objects

- Many applications involve collections of objects:
  - Personal organizers.
  - Library catalogs.
  - Student-record system.
- The number of items to be stored varies.
  - Items get added.
  - Items get deleted.

Example: a personal notebook

- Notes may be stored.
- Individual notes can be viewed.
- There is no limit to the number of notes.
- It will tell how many notes are stored.
- BlueJ example *notebook1* project.

Class libraries

- Collections of useful classes.
- We don’t have to write everything from scratch.
- Java calls its libraries *‘packages’*.
- Grouping objects is a recurring requirement.
  - The `java.util` package contains classes for doing this.

```
import java.util.ArrayList;

/**
 * ...
 */

public class Notebook
{
    // Storage for any number of notes.
    private ArrayList notes;

    /**
     * Initialise the notebook.
     */
    public Notebook()
    {
        notes = new ArrayList();
    }

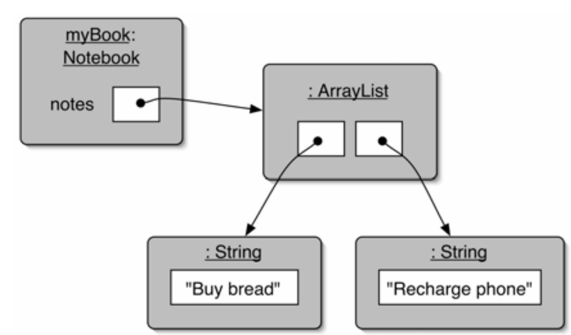
    ...
}
```

Define which package the class will come from

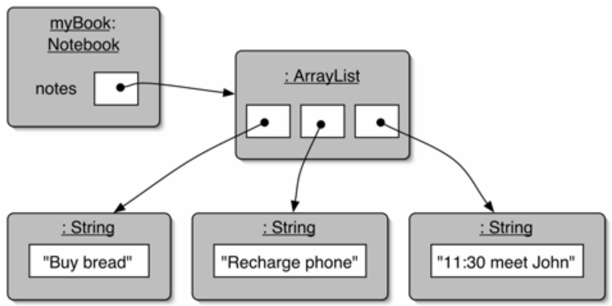
Declare variable of type ArrayList

Create an instance of ArrayList class

Object structures with collections



Adding a third note



Features of the collection

- It increases its capacity as necessary.
- It keeps a private count (**`size()`** accessor).
- It keeps the objects in order.
- Details of how all this is done are hidden.
  - Does that matter? Does not knowing how prevent us from using it?
  - The size operation can be 'delegated' without knowing how it is achieved, so long as the signature is known

Using the collection

```
public class Notebook
{
    private ArrayList notes;
    ...

    public void storeNote(String note)
    {
        notes.add(note);
    }

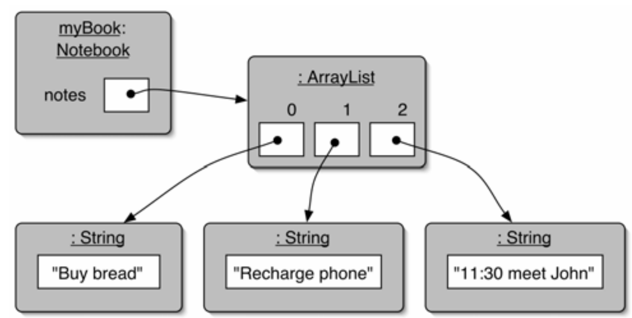
    public int numberOfNotes()
    {
        return notes.size();
    }

    ...
}
```

Adding a new note

Returning the number of notes (*delegation*).

Index numbering



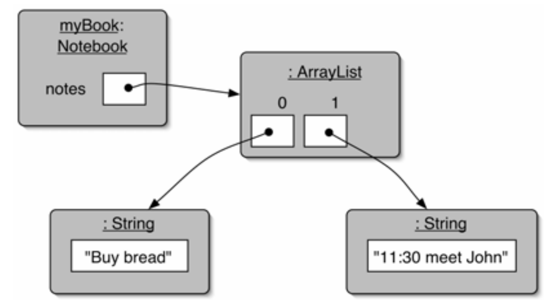
Retrieving an object

Index validity checks

```
public void showNote(int n[redacted]umber)
{
    if(noteNumber < 0) {
        // This is not a va[redacted]d note number.
    }
    else if(noteNumber < numberOfNotes()) {
        System.out.println(notes.get(noteNumber));
    }
    else {
        // This is not a valid n[redacted]te number.
    }
}
```

Retrieve and print the note

Removal may affect numbering



Review

- Collections allow an arbitrary number of objects to be stored.
- Class libraries usually contain tried-and-tested collection classes.
- Java's class libraries are called *packages*.
- We have used the `ArrayList` class from the `java.util` package.

Review

- Items may be added and removed.
- Each item has an index.
- Index values may change if items are removed (or further items added).
- The main `ArrayList` methods are `add`, `get`, `remove` and `size`.

Iteration

- We often want to perform some actions an arbitrary number of times.
  - E.g., print all the notes in the notebook. How many are there?
- Most programming languages include *loop statements* to make this possible.
- Java has three sorts of loop statement.
  - We will focus on its *while loop*.

'While' loop – pseudo code

General form of a while-loop

while keyword

Boolean test

```
while ( loop condition ) {
    loop body – code to be
    repeated
}
```

Statements to be repeated

Example to print every note

```
while(there is at least one more note to be printed) {
    show the next note
}
```

A Java example

```
/**
 * List all notes in the notebook.
 */

public void listNotes()
{
    int index = 0;
    while(index < notes.size()) {
        System.out.println(notes.get(index));
        index++;
    }
}
```

Increment by one

Iterating over a collection

java.util.Iterator

Returns an Iterator object

```
Iterator it = myCollection.iterator();
while(it.hasNext()) {
    call it.next() to get the next object
    do something with that object
}
```

```
public void listNotes()
{
    Iterator it = notes.iterator();
    while(it.hasNext()) {
        System.out.println(it.next());
    }
}
```

Review

- Loop statements allow a block of statements to be repeated.
- A Java while loop allows the repetition to be controlled by a boolean expression.
- Collection classes have special `Iterator` objects that simplify iteration over the whole collection.

Fixed-size collections

- Sometimes the maximum collection size can be pre-determined.
- Programming languages usually offer a special fixed-size collection type: an *array*.
- Java arrays can store objects or primitive-type values.
- Arrays use a special syntax.

Creating an array object

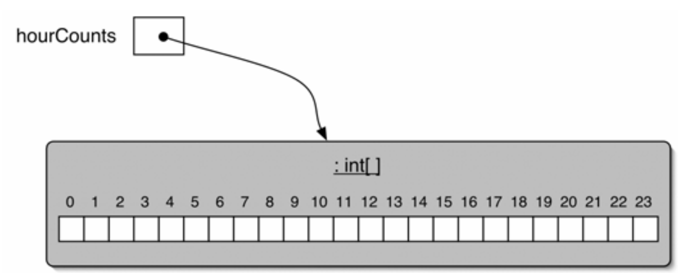
Array variable declaration

Array object creation

```
public class LogAnalyzer
{
    private int[] hourCounts;
    private LogfileReader reader;

    public LogAnalyzer()
    {
        hourCounts = new int[24];
        reader = new LogfileReader();
    }
    ...
}
```

An array in memory



Using an array

- Unlike collections
  - Removing items doesn't change numbering
  - But size doesn't change dynamically either
- Square-bracket notation is used to access an array element:
  - **`hourCounts[...]`**

    Can be number, variable or expression
- Elements are used like ordinary variables.
  - On the left of an assignment:
    - **`hourCounts[hour] = ...;`**
  - In an expression:
    - **`adjusted = hourCounts[hour] – 3;`**
    - **`hourCounts[hour]++;`**

The 'for' loop

- Similar to a while-loop.
- Often used to iterate a fixed number of times.
- Often used to iterate over an array.

'For' loop - pseudo-code

General form of a for-loop

```
for(initialization; condition; post-body action) {
    statements to be repeated
}
```

Equivalent in while-loop form

```
initialization;
while(condition) {
    statements to be repeated
    post-body action
}
```

A Java example

for-loop version

```
for(int hour = 0; hour < hourCounts.length; hour++) {
    System.out.println(hour + ": " + hourCounts[hour]);
}
```

while-loop version

```
int hour = 0;
while(hour < hourCounts.length) {
    System.out.println(hour + ": " + hourCounts[hour]);
    hour++;
}
```

Review

- Arrays are appropriate where a fixed-size collection is required.
- Arrays use special syntax.
- For-loops offer an alternative to while-loops when the number of repetitions is known.
- For-loops are often used to iterate over arrays.

The Java class library

- Thousands of classes
- Tens of thousands of methods
- Many useful classes that make life much easier
- A competent Java programmer must be able to work with the libraries.

Reading class documentation

- Documentation of the Java libraries in HTML format;
- Readable in a web browser
- Class API: *Application Programmers' Interface*
- Interface description for all library classes

Interface vs implementation

The documentation includes:

- the name of the class;
- a general description of the class;
- a list of constructors and methods
- return values and parameters for constructors and methods
- a description of the purpose of each constructor and method

→ the interface of the class

Interface vs implementation

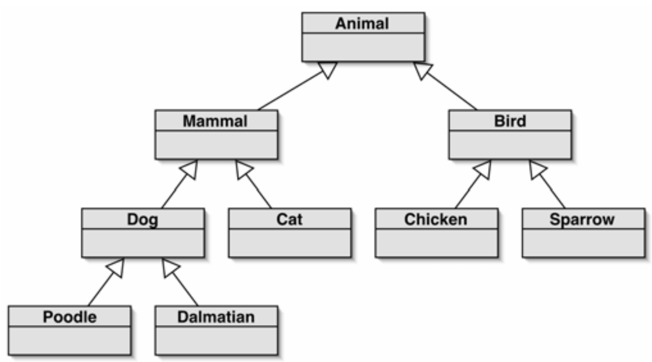
*The documentation does **not** include*

- private fields (most fields are private)
- private methods
- the bodies (source code) for each method

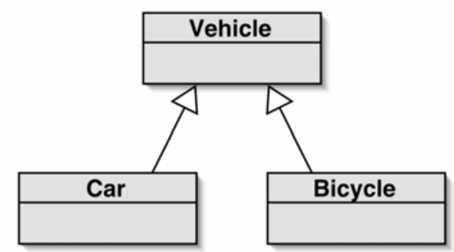
→ the implementation of the class

Inheritance

Inheritance hierarchies



Simple example



Using inheritance

- define one *superclass* : Vehicle
- define *subclasses* for Car and Bicycle
- the superclass defines common *attributes*
- the subclasses *inherit* the superclass attributes that they will have in common
- the subclasses add their own attributes

- In Java, attributes are defined using fields, and subclasses as an ‘extends’ relationship between classes.

Inheritance in Java

no change

```
public class Vehicle
{
    ...
}
```

different here

```
public class Bicycle extends Vehicle
{
    ...
}
```

```
public class Car extends Vehicle
{
    ...
}
```

Superclass

```
public class Vehicle
{
    private String colour;
    private int kerbWeight;

    // constructor and methods
    // omitted
}
```

Subclasses

```
public class Car extends Vehicle
{
    private int engineSize;
}
```

```
public class Bicycle extends Vehicle
{
    private int numberGears;
    private boolean stabilisers;
}
```

Inheritance and constructors

```
public class Vehicle
{
    private String colour;
    private int kerbWeight;

    /**
     * Initialise the fields of Vehicle.
     */
    public Vehicle(String paint, int weight)
    {
        kerbWeight = weight;
        colour = paint;
    }
}
```

Inheritance and constructors

```
public class Car extends Vehicle
{
    private int engineSize;

    /**
     * Constructor for objects of class Car
     */
    public Car(String paint,
               int weight, int engine)
    {
        super(paint, weight);
        engineSize = engine;
    }
}
```

calls superclass constructor

Superclass constructor call

- Subclass constructors must always contain a 'super' call.
- The ‘super’ call must be the first statement in the subclass constructor.
- If none is written, the compiler inserts one (without parameters)
  - Be careful: this will work only if the superclass has defined a suitable constructor without parameters

Subclasses and subtyping

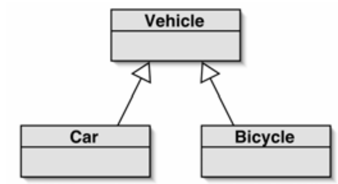
- Classes define types.
- Subclasses define subtypes.
- Objects of subclasses can be used where objects of supertypes are required.
  (This is called *substitution*)
- Compare:
  - A bicycle is a type of vehicle
  - I’m going to work by vehicle
  - I’m going to work by bicycle
  - I’m going to work by car

subtype

statement about supertype

substitutions

Subtyping and assignment

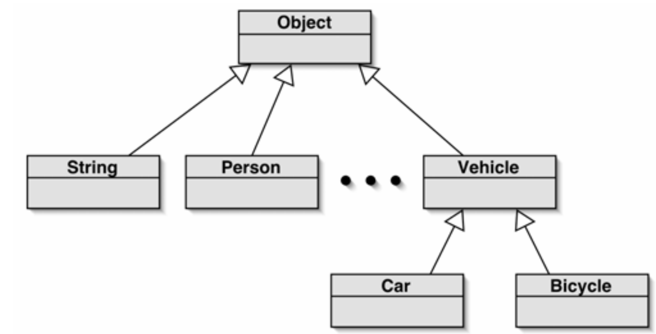


subclass objects may be assigned to superclass variables

```
Vehicle v1 = new Vehicle();
Vehicle v2 = new Car();
Vehicle v3 = new Bicycle();
```

The Object class

all classes inherit from `Object`



Polymorphism

- Java collection classes are *polymorphic*
  - *They can operate on many different types*
- The elements stored in a collection are actually defined to be of type Object.
  - Parameters and return values of the collection class's mutator and accessor methods are also defined as type Object:

```
public void add(Object element)

public Object get(int index)
```

Casting

- Allowed to assign subtype to supertype.
- Not allowed to assign supertype to subtype!

`String s1 = myList.get(1);` ***error!***

- Why?
  - ***My vehicle is rolling → my bicycle is rolling*** OK
  - ***I pedal my bicycle → I pedal my vehicle*** Not OK
- Casting makes it seem OK:

`String s1 = (String) myList.get(1);`

(but must be sure this really will be a String!)

A few more Java features

The do … while loop

while loop

`while` keyword

Boolean test

```
while(loop condition) {
    loop body
}
```

Statements to be repeated

do … while loop

`do` keyword

Body executes at least once

```
do {
    loop body
} while (loop condition)
```

`while` keyword

Boolean test *after* loop

The switch statement

```
switch (expression)
{
    case value:
        statements;
        break;
    case value:
        statements;
        break;
    case value:
        statements;
        break;
    default:
        statements;
        break;
}
```

Value to switch on

Any number of case labels

Stops execution "falling through" to next case

Optional: guarantees *something* happens

Example of switch

```
char response = reader.getChar();

switch (response)
{
    case 'y':
        answer = "Loves me";
        break;
    case 'n':
        answer = "Loves me not";
        break;
    default:
        answer = "Confusion";
        break;
}
```

switch versus if … else

```
char response = reader.getChar();

if (response == 'y') {
    answer = "Loves me";
}
else if (response == 'n') {
    answer = "Loves me not";
}
else {
    answer = "Confusion";
}
```

Primitive data types (1)

- **`boolean`**
  - Only two values: **`true`** or **`false`**
- **`int`**
  - 32 bit integer (literals: **`1294123`**, **`-88123`**)
- **`byte`**
  - 8-bit integer (literals: **`24`**, **`-2`**)
- **`short`**
  - 16 bit integer (literals: **`5409`**, **`-2004`**)
- **`long`**
  - 64 bit integer (literals: **`4232663531`**, **`55L`**)

Primitive data types (2)

- **`char`**
  - Unicode character (16 bit) (literals: **`'m'`**, **`'\u00F6'`**)
- **`float`**
  - Single precision floating point (literals: **`43.889F`**)
- **`double`**
  - Double precision floating point (literals: **`45.63`**, **`2.4e5`**)

Operators

- Assignment (all primitive and object types)
  - `=`
- Arithmetic (numeric ➔ numeric)
  - `+  -  /  *  %`
- Increment (numeric ➔ numeric)
  - `+=  -=  ++  --`
- Relational (most types ➔ boolean)
  - `==  !=  >  <  <=  >=`
- Boolean (boolean ➔ boolean)
  - `&&  ||  ^  !`

Strings: a common trap

```
if(input == "bye") {
   ...
}
```

tests ***identity***: is this the same instance of string?

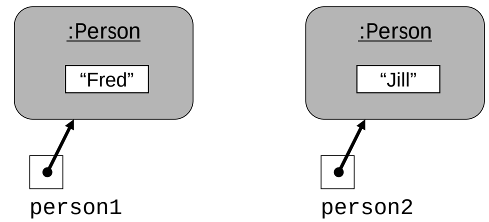
```
if(input.equals("bye")) {
   ...
}
```

tests ***equality***: does this string have the same letters?

- Strings should (almost) always be compared using **`.equals`**

Identity vs equality 1

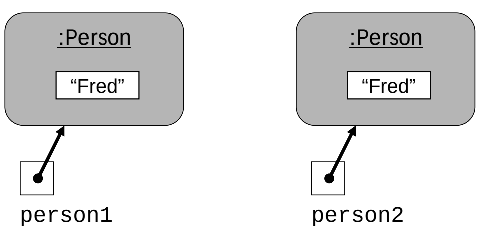
Other (non-String) objects:



`person1 == person2 ?`

Identity vs equality 2

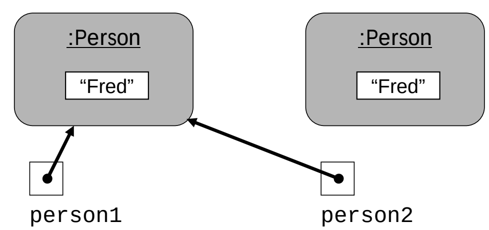
Other (non-String) objects:



`person1 == person2 ?`

Identity vs equality 3

Other (non-String) objects:

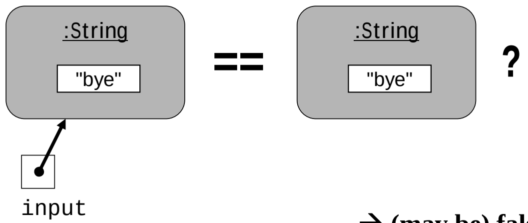


`person1 == person2 ?`

Identity vs equality (Strings)

```
String input = reader.getInput();
if(input == "bye") {
   ...
}
```

"==" tests identity

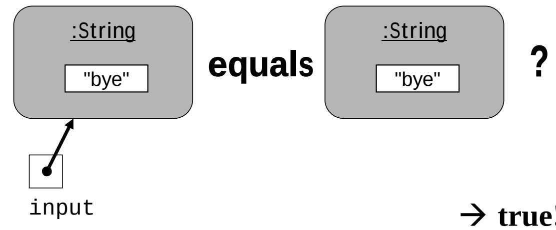


→ **(may be) false!**

Identity vs equality (Strings)

```
String input = reader.getInput();
if(input.equals("bye")) {
   ...
}
```

"`equals`" tests equality



→ **true!**

Software Design

Models, Tools & Processes

Alan Blackwell

Cambridge University
Computer Science Tripos Part 1a

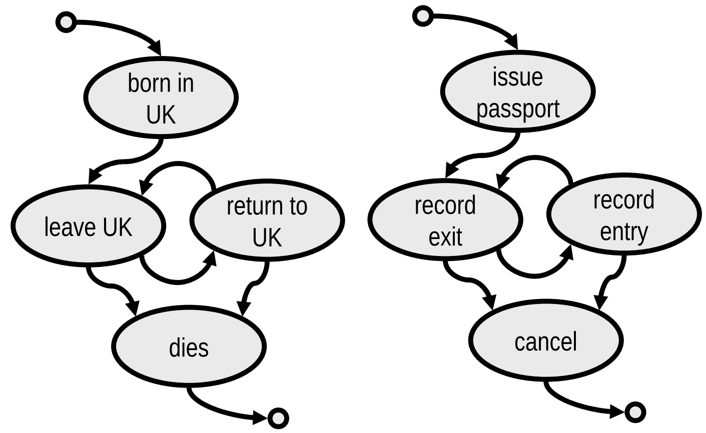
How hard can it be?

- State what the system should do
  - $\{D_1, D_2, D_3 \ldots\}$
- State what it shouldn't do
  - $\{U_1, U_2, U_3 \ldots\}$
- Systematically add features
  - that can be proven to implement D_n
  - while not implementing U_n

How hard can it be …

- The United Kingdom Passport Agency
  - http://www.parliament.the-stationery-office.co.uk/pa/cm199900/cmselect/cmpubacc/65/6509.htm
- 1997 contract for new computer system
  - aimed to improve issuing efficiency, on tight project timetable
  - project delays meant throughput not thoroughly tested
  - first live office failed the throughput criterion to continue roll-out
  - second office went live, roll out halted, but no contingency plan
  - rising backlog in early 1999, alongside increasing demand
  - passport processing times reached 50 days in July 1999
  - widespread publicity, anxiety and panic for travelling public
  - telephone service overloaded, public had to queue at UKPA offices
  - only emergency measures eventually reduced backlog
- So how hard can it be to issue a passport?
  - … let's try some simple definition

… to define this system?



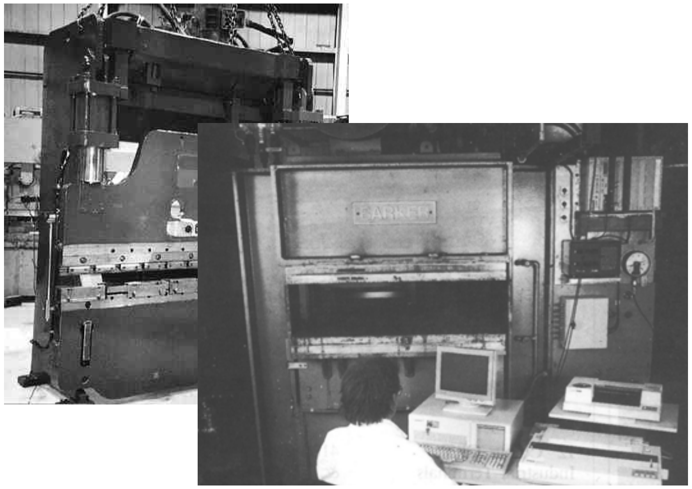
How hard can it be …

… to define a simple bureaucracy?

Why is the world complicated?

- Bureaucratic systems are complex because managers (and people) always mess up
  - Passports
  - Ambulance systems (more in part 1B)
  - University financial systems (later in this course)
- What about physical systems, which don't rely on people to work?
  - Start with known characteristics of physical device.
  - Assemble behaviours to achieve function
  - This is how engineering products (bridges and aircraft) are designed.

How hard can it be …



… to define a physical system?

Design and uncertainty

- A good programmer should be able to:
  - Create a system that behaves as expected.
  - Behaves that way reliably.
- But a good designer must also:
  - Take account of the ***unexpected***.
- A well-designed software system is not the same as a well-designed algorithm.
  - If the requirements change or vary, you might replace the algorithm,
  - But it's seldom possible to replace a whole system.

What is the problem?

- The problem is ***not*** that we don't understand the computer.
- The problem ***is*** that we don't understand the problem!
- Does computer science offer any answers?
- The good news:
  - We've been working on it since 1968
- The bad news:
  - There is still no "silver bullet"!
    (from great IBM pioneer Fred Brooks)

Introduction

A design process based on knowledge

Pioneers – Bavarian Alps, 1968

- 1954: complexity of SAGE air-defence project was under-estimated by 6000 person-years …
  - … at a time when there were only about 1000 programmers in the whole world!
  - … “Software Crisis!”

- 1968: First meeting on “Software Engineering” convened in Garmisch-Partenkirchen.

Design and ignorance

- Some say software engineering is the part that is too hard for computer scientists.
- But the real change was understanding the importance of what you ***don’t*** know
  - dealing with uncertainty, lack of knowledge …
  - … but trying to be *systematically* ignorant!
- Design is a process, not a set of known facts
  - process of learning about a problem
  - process of describing a solution
  - at first with many gaps …
  - eventually in sufficient detail to build the solution

Learning by building models

- The software design process involves gaining knowledge about a problem, and about its technical solution.
- We describe both the problem and the solution in a series of *design models*.
- Testing, manipulating and transforming those models helps us gather more knowledge.
- One of the most detailed models is written in a programming language.
  - Getting a working program is almost a side-effect of describing it!

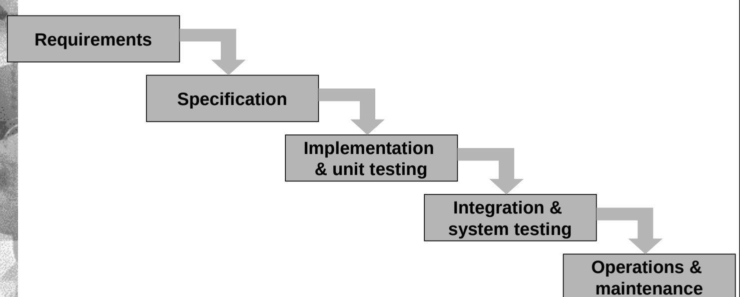
Unified Modeling Language

- **Use Case** diagrams - interactions with / interfaces to the system.
- **Class** diagrams - type structure of the system.
- **Collaboration** diagrams - interaction between instances
- **Sequence** diagrams - temporal structure of interaction
- **Activity** diagrams - ordering of operations
- **Statechart** diagrams - behaviour of individual objects
- **Component** and **Deployment** diagrams - system organisation

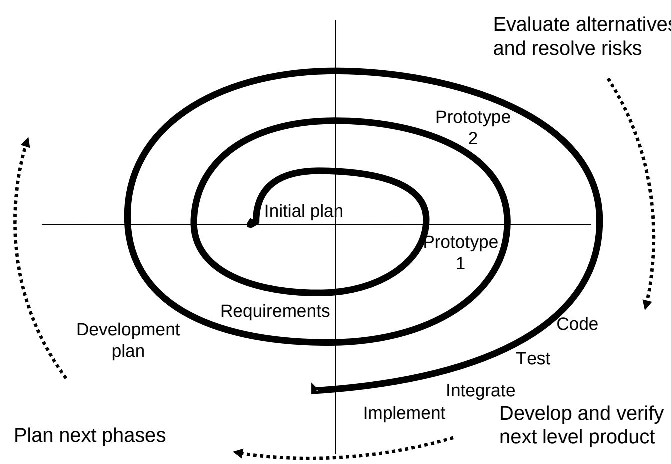
Outline for the rest of the course

- Roughly follows stages of the (UML-related) *Rational Unified Process*
  - Inception
    - structured description of what system must do
  - Elaboration
    - defining classes, data and system structure
  - Construction
    - object interaction, behaviour and state
  - Transition
    - testing and optimisation
- Plus allowance for *iteration*
  - at every stage, and through all stages

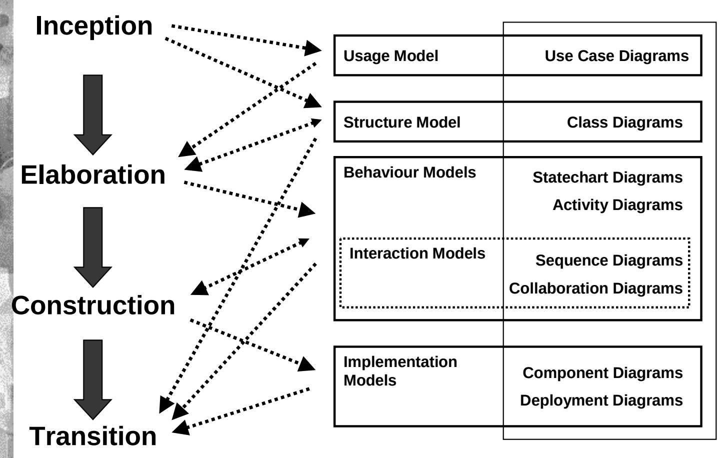
Older terminology: the “waterfall”



Modern alternative: the “spiral”



The Design Process vs. The Design



Books

- ***Code Complete:*** A practical handbook of software construction
  - Steve McConnell, Microsoft Press 2004 (2nd edition)
- ***UML Distilled:*** A brief guide to the standard object modeling language
  - Martin Fowler, Addison-Wesley 2003 (3rd edition)
- Further:
  - *Software Pioneers,* Broy & Denert
  - *Software Engineering,* Roger Pressman
  - *The Mythical Man-Month,* Fred Brooks
  - *The Design of Everyday Things,* Donald Norman
  - *Contextual Design,* Hugh Beyer & Karen Holtzblatt
  - *The Sciences of the Artificial*, Herbert Simon
  - *Educating the Reflective Practitioner*, Donald Schon
  - *Designing Engineers*, Louis Bucciarelli

Exam questions

- This syllabus appeared *under this name* for the first time in 2006 (without Java element):
  - Software Design 2006, Paper 2, Q7
- But syllabus was previously introduced as:
  - Software Engineering II 2005, Paper 2, Q8
- Some components had previously been taught elsewhere in the Tripos:
  - Programming in Java 2004, Paper 1, Q10
  - Software Engineering and Design 2003 Paper 10, Q12 and 2004 Paper 11, Q11
  - Additional Topics 2000, Paper 7, Q13

Inception phase

structured description of system usage and function

Pioneers – Tom DeMarco

- Structured Analysis
  - 1978, Yourdon Inc
- Defined the critical technical role of the system analyst
  - Analyst acts as a middleman between users and (technical) developers
- Analyst's job is to construct a functional specification
  - data dictionary, data flow, system partitioning

How can you capture requirements?

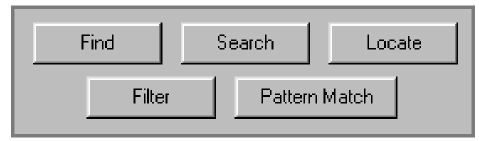
Analysing requirements

- Analysis usually involves (re)negotiation of requirements between client and designer.
  - Once considered "*requirements capture*".
  - Now more often "*user-centred design*".
- An "interaction designer" often replaces (or works alongside) traditional systems analysts.
  - Professional interaction design typically combines research methods from social sciences with visual or typographic design skills (and perhaps CS).

Interaction design bugs

Interaction design bugs

Interaction design bugs



From Interface Hall of Shame

The psychological approach

- Anticipate what will happen when someone tries to use the system.
  - Design a “conceptual model” that will help them (and you) develop shared understanding.
- The gulf of execution:
  - System users know ***what*** they want to achieve, but can’t work out ***how*** to do it.
- The gulf of evaluation:
  - Systems fail to give suitable feedback on what just happened, so users never learn what to do.
- See Norman: *Design of Everyday Things*.
  - Far more detail to come in Part II HCI course

The anthropological approach

- Carry out fieldwork:
  - Interview the users.
  - Understand the context they work in.
  - Observe the nature of their tasks.
  - Discover things by observation that they might not have told you in a design brief.
- Collaborate with users to agree:
  - What problem ought to be solved.
  - How to solve it (perhaps by reviewing sketches of proposed screens etc.).

Ethnographic field studies

- Understand real detail of user activity, not just official story, theories or rationalisations.
- Researchers work in the field:
  - Observing context of people's lives
  - Ideally participating in their activities
- Academic ethnography tends to:
  - Observe subjects in a range of *contexts*.
  - Observe over a substantial *period of time*.
  - Make full record of both *activities* and *artefacts*.
  - Use transcripts of video/audio recordings.

Design 'ethnography'

- Study division of labour and its coordination
- Plans and procedures
  - When do they succeed and fail?
- Where paperwork meets computer work
- Local knowledge and everyday skills
- Spatial and temporal organisation
- Organisational memory
  - How do people learn to do their work?
  - Do formal/official methods match reality?
- See Beyer & Holtzblatt, *Contextual Design*

Interviews

- Field work usually includes interviews
  - ***Additional*** to requirements meetings with client
- Often conducted in the place of work during 'contextual enquiry' (as in Beyer & Holtzblatt)
  - emphasis on user tasks, not technical issues
- Plan questions in advance
  - ensure all important aspects covered
- May be based on theoretical framework, e.g.
  - *activities*, *methods* and *connections*
  - *measures*, *exceptions* and domain *knowledge*

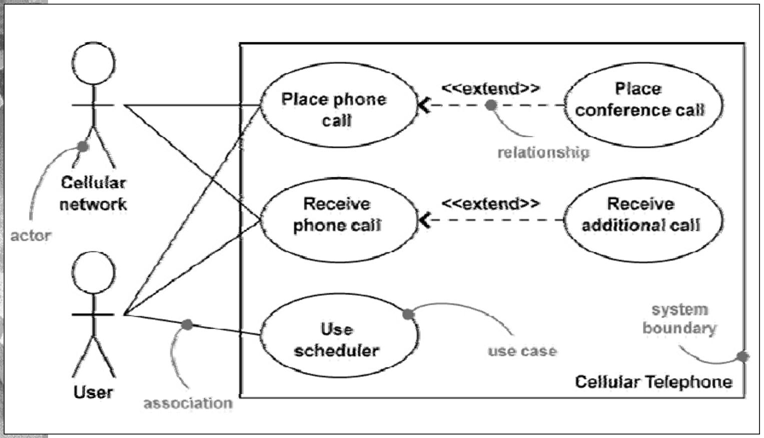
User Personas

- This is a way to 'distil' information about users
  - from field work, interviews, user studies etc
  - into a form that is more useful to design teams.
- Write fictional portraits of individuals representing various kinds of user
  - give them names, jobs, and personal history
  - often include photographs (from libraries ,actors)
- Help software engineers to remember that customers are not like them …
  - … or their friends …
  - … or anyone they've ever met!

Designing system-use scenarios

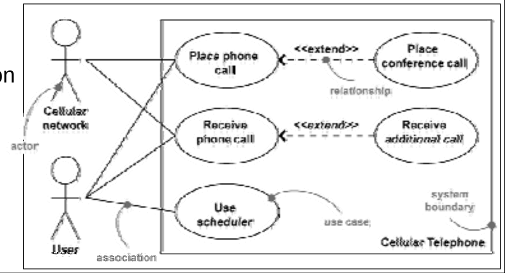
- Aim is to describe the human activity that the system has to carry out or support.
  - Known as *use cases* in UML
- Use cases help the designer to discover and record interactions between software objects.
- Can be refined as a group activity, based on personas, or in discussion with clients.
- May include mock-ups of screen designs, or physical prototypes.
- Organised and grouped in use case diagrams

UML Use Case diagram



UML Use Case diagram

- Actors
  - play system *role*
  - may not be people
- Use case
  - like a scenario
- Relationships
  - include
  - extend
  - generalisation



Objects in a scenario

- The ***nouns*** in a description refer to 'things'.
  - A source of classes and objects.
- The ***verbs*** refer to actions.
  - A source of interactions between objects.
  - Actions describe object behavior, and hence required methods.

Example of problem description

The cinema booking system should store seat bookings for multiple theatres.

Each theatre has seats arranged in rows.

Customers can reserve seats and are given a row number and seat number.

They may request bookings of several adjoining seats.

Each booking is for a particular show (i.e., the screening of a given movie at a certain time).

Shows are at an assigned date and time, and scheduled in a theatre where they are screened.

The system stores the customers' telephone number.

Nouns

The cinema booking system should store seat bookings for multiple theatres.

Each theatre has seats arranged in rows.

Customers can reserve seats and are given a row number and seat number.

They may request bookings of several adjoining seats.

Each booking is for a particular show (i.e., the screening of a given movie at a certain time).

Shows are at an assigned date and time, and scheduled in a theatre where they are screened.

The system stores the customers' telephone number.

Verbs

The cinema booking system should store seat bookings for multiple theatres.

Each theatre has seats arranged in rows.

Customers can reserve seats and are given a row number and seat number.

They may request bookings of several adjoining seats.

Each booking is for a particular show (i.e., the screening of a given movie at a certain time).

Shows are at an assigned date and time, and scheduled in a theatre where they are screened.

The system stores the customers' telephone number.

Extracted nouns & verbs

Cinema booking system
Stores (seat bookings)
Stores (telephone number)

Theatre
Has (seats)

Movie

Customer
Reserves (seats)
Is given (row number, seat number)
Requests (seat booking)

Time

Date

Seat booking

Show
Is scheduled (in theatre)

Seat

Seat number

Telephone number

Row

Row number

Scenario structure: CRC cards

- First described by Kent Beck and Ward Cunningham.
  - Later innovators of “agile” programming (more on this later in course)
- Use simple index cards, with each cards recording:
  - A *class* name.
  - The class’s *responsibilities*.
  - The class’s *collaborators*.

Typical CRC card

Class name	**Collaborators**
Responsibilities	

Partial example

`CinemaBookingSystem`	*Collaborators*
`Can find movies by title and day.`	**`Movie`**
`Stores collection of movies.`	**`Collection`**
`Retrieves and displays movie details.`	
`...`	

Refinement of usage model

- Scenarios allow you to check that the problem description is clear and complete.
- Analysis leads gradually into design.
  - Talking through scenarios & class responsibilities leads to elaborated models.
- Spotting errors or omissions here will save considerable wasted effort later!
  - Sufficient time should be taken over the analysis.
  - CRC was designed to allow (in principle) review and discussion with analysts and/or clients.

Elaboration

defining classes, data and system structure

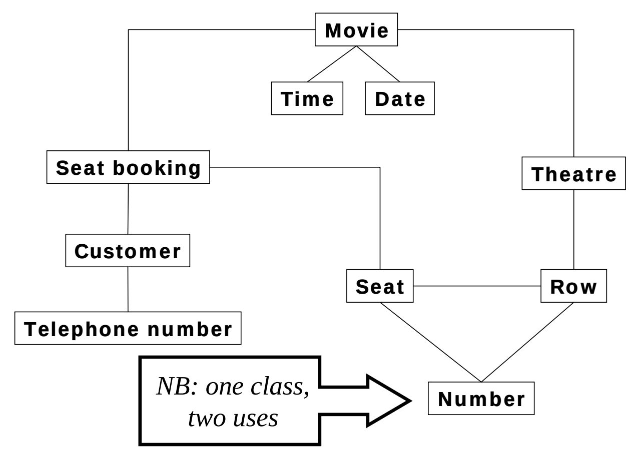
Pioneers – Peter Chen

- Entity-Relationship Modeling
  - 1976, Massachusetts Institute of Technology
- User-oriented response to Codd's relational database model
  - Define attributes and values
  - Relations as associations between things
  - Things play a *role* in the relation.
- E-R Diagrams showed entity (box), relation (diamond), role (links).
- Object-oriented Class Diagrams show class (box) and association (links)

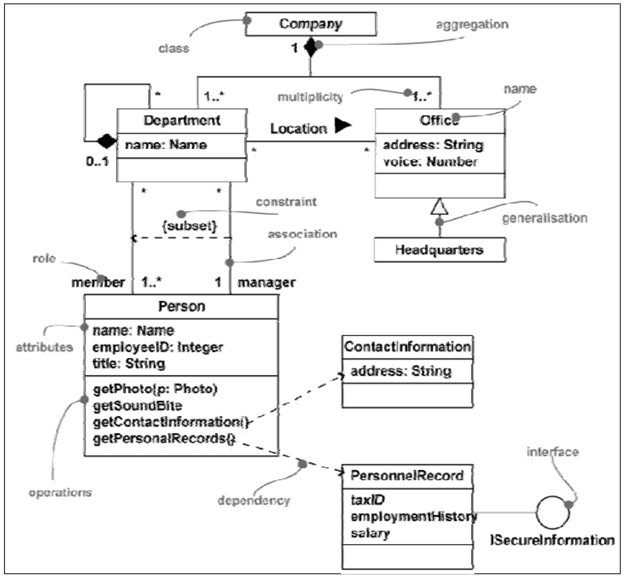
Review of objects and classes

- objects
  - represent 'things' in some problem domain (example: "the red car down in the car park")
- classes
  - represent all objects of a kind (example: "car")
- operations
  - actions invoked on objects (Java "*methods")*
- instance
  - can create many instances from a single class
- state
  - all the attributes (field values) of an instance

Typical classes and associations

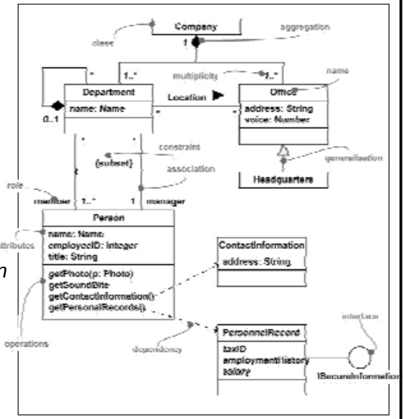


UML Class diagram



UML Class diagram

- Attributes
  - *type* and *visibility*
- Operations
  - signature and visibility
- Relationships
  - *association*
    - with *multiplicity*
    - potentially *aggregation*
  - *generalisation*



Association and aggregation

The cinema booking system should store seat bookings for multiple theatres.

Each theatre has seats arranged in rows.

Customers can reserve seats and are given a row number and seat number.

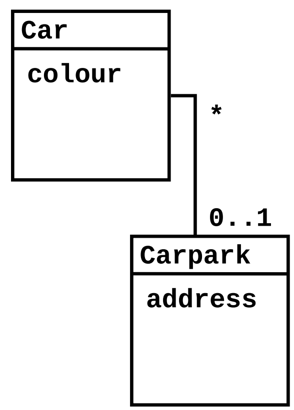
They may request bookings of several adjoining seats.

Each booking is for a particular show (i.e., the screening of a given movie at a certain time).

Shows are at an assigned date and time, and scheduled in a theatre where they are screened.

The system stores the customers' telephone number.

Implementing association in Java

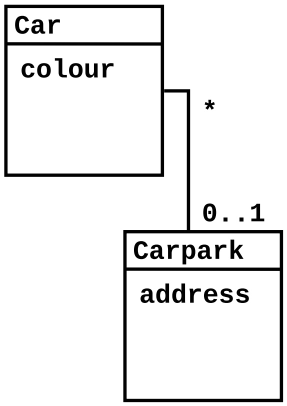



```
public class Car {
  private String colour;
  private Carpark park;

...

park_me (Carpark where)
{
   park = where;
}
```

Multiple association in Java




```
public class Carpark {
  private String address;
  private ArrayList my_cars;

...

add_car (Car new_car)
{
   my_cars.add(new_car);
}
```

Implementing multiple associations

Summary from 3 Java lectures:

- Most applications involve collections of objects
  - java.util package contains classes for this
- The number of items to be stored varies
  - Items can be added and deleted
  - Collection increases capacity as necessary
  - Count of items obtained with size()
  - Items kept in order, accessed with *iterator*
- Details of how all this is done are hidden.

Class design from CRC cards

- Scenario analysis helps to clarify application structure.
  - Each card maps to a class.
  - Collaborations reveal class cooperation/object interaction.
- Responsibilities reveal public methods.
  - And sometimes fields; e.g. “Stores collection ...”

Refining class interfaces

- Replay the scenarios in terms of method calls, parameters and return values.
- Note down the resulting method signatures.
- Create outline classes with public-method stubs.
- Careful design is a key to successful implementation.

Dividing up a design model

- Abstraction
  - Ignore details in order to focus on higher level problems (e.g. aggregation, inheritance).
  - If classes correspond well to types in domain they will be easy to understand, maintain and reuse.
- Modularization
  - Divide model into parts that can be built and tested separately, interacting in well-defined ways.
  - Allows different teams to work on each part
  - Clearly defined interfaces mean teams can work independently & concurrently, with increased chance of successful integration.

Pioneers – David Parnas

- Information Hiding
  - 1972, Carnegie Mellon University
- How do you decide the points at which a program should be split into pieces?
  - Are small modules better?
  - Are big modules better?
  - What is the optimum boundary size?
- Parnas proposed the best criterion for modularization:
  - Aim to hide design decisions within the module.

Information hiding in OO models

- Data belonging to one object is hidden from other objects.
  - Know *what* an object can do, not *how* it does it.
  - Increases independence, essential for large systems and later maintenance
- Use Java visibility to hide implementation
  - Only methods intended for interface to other classes should be public.
  - Fields should be private – accessible only within the same class.
  - *Accessor* methods provide information about object state, but don't change it.
  - *Mutator* methods change an object's state.

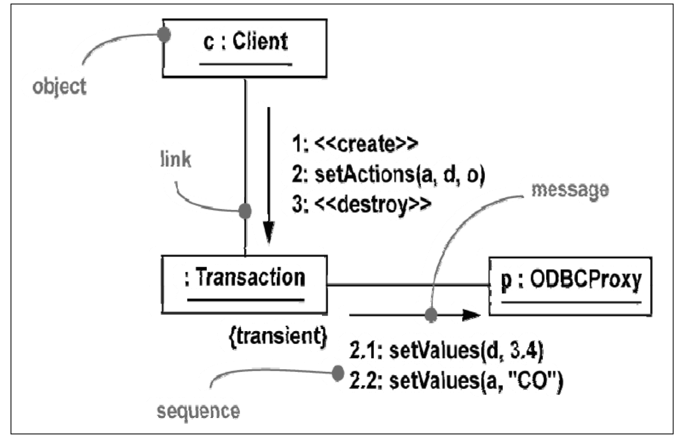
Cohesion in OO models

- Aim for high cohesion:
  - Each component achieves only "one thing"
- Method (functional) cohesion
  - Method only performs out one operation
  - Groups things that must be done together
- Class (type) cohesion
  - Easy to understand & reuse as a domain concept
- Causes of low, poor, cohesion
  - Sequence of operations with no necessary relation
  - Unrelated operations selected by control flags
  - No relation at all – just a bag of code

Construction

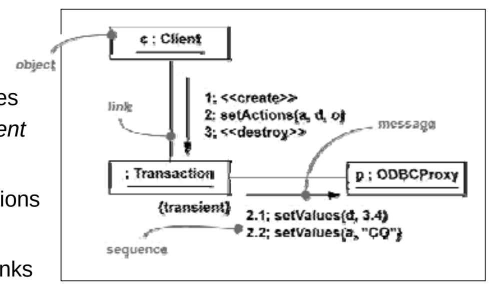
object interaction, behaviour and state

UML Collaboration diagram

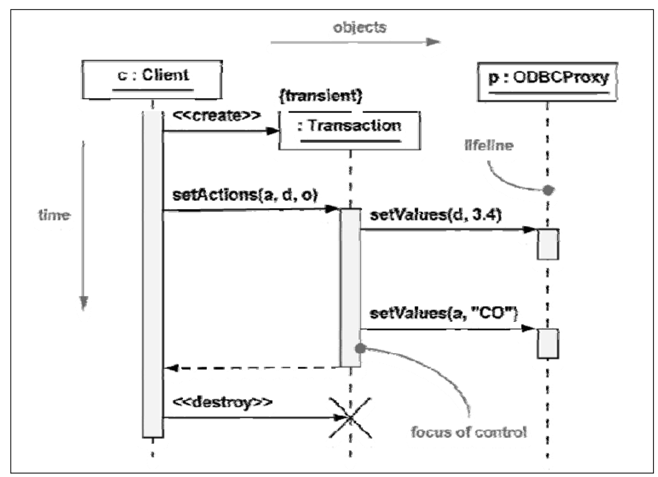


UML Collaboration diagram

- Objects
  - class instances
  - can be *transient*
- Links
  - from associations
- Messages
  - travel along links
  - numbered to show *sequence*

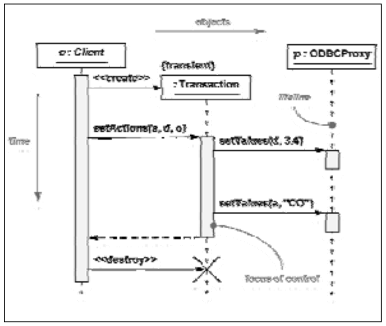


UML Sequence diagram



UML Sequence diagram

- Interaction again
  - same content as collaboration
  - emphasises time dimension
- Object *lifeline*
  - objects across page
  - time down page
- Shows *focus of control*



Loose coupling

- Coupling: links between parts of a program.
- If two classes depend closely on details of each other, they are *tightly coupled*.
- We aim for *loose coupling*.
  - keep parts of design clear & independent
  - may take several design iterations
- Loose coupling makes it possible to:
  - achieve reusability, modifiability
  - understand one class without reading others;
  - change one class without affecting others.
- Thus improves maintainability.

Responsibility-driven design

- Which class should I add a new method to?
  - Each class should be responsible for manipulating its own data.
  - The class that owns the data should be responsible for processing it.
- Leads to low coupling & "client-server contracts"
  - Consider every object as a server
  - Improves reliability, partitioning, graceful degradation

Interfaces as specifications

- Define method *signatures* for classes to interact
  - Include parameter and return types.
  - Strong separation of required functionality from the code that implements it (information hiding).
- Clients interact independently of the implementation.
  - But clients can choose from alternative implementations.

Interfaces in Java

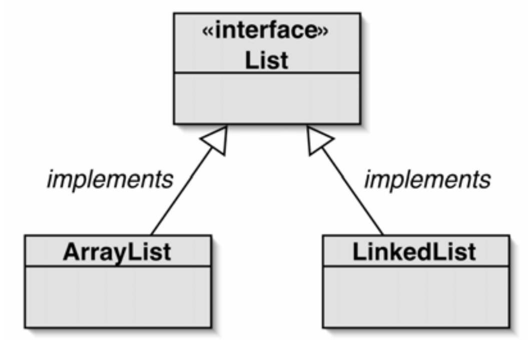
- Provide specification without implementation.
  - Fully *abstract* – define interface only
  - Implementing classes don't inherit code
- Support not only polymorphism, but *multiple inheritance*
  - implementing classes are still subtypes of the interface type, but allowed more than one "parent".

```
public class ArrayList implements List

public class LinkedList implements List
```

Note difference from 'extends' keyword used for sub-classing

Alternative implementations



Causes of error situations

- Incorrect implementation.
  - Does not meet the specification.
- Inappropriate object request.
  - E.g., invalid index.
- Inconsistent or inappropriate object state.
  - E.g. arising through class extension.
- Not always programmer error
  - Errors often arise from the environment (incorrect URL entered, network interruption).
  - File processing often error-prone (missing files, lack of appropriate permissions).

Defensive programming

- Client-server interaction.
  - Should a server assume that clients are well-behaved?
  - Or should it assume that clients are potentially hostile?
- Significant differences in implementation required.
- Issues to be addressed
  - How much checking by a server on method calls?
  - How to report errors?
  - How can a client anticipate failure?
  - How should a client deal with failure?

Argument values

- Arguments represent a major ‘vulnerability’ for a server object.
  - Constructor arguments initialize state.
  - Method arguments often control behavior.
- Argument checking is one defensive measure.
- How to report illegal arguments?
  - To the user? ***Is*** there a human user? Can the user do anything to solve the problem? If not solvable, what should you suggest they do?
  - To the client object: return a diagnostic value, or throw an exception.

Example of diagnostic return

```
public boolean removeDetails(String key)
{
    if(keyInUse(key)) {
        ContactDetails details =
                       (ContactDetails) book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;
        return true;
    }
    else {
        return false;
    }
}
```

Diagnostic OK

Diagnostic not OK

Client response to diagnostic

- Test the return value.
  - Attempt recovery on error.
  - Avoid program failure.
- Ignore the return value.
  - Cannot be prevented.
  - Likely to lead to program failure.
- Exceptions are preferable.

Exception-throwing

- Special feature of some languages
  - Java does provide exceptions
- Advantages
  - No ‘special’ return value needed.
  - Errors cannot be ignored in the client.
- Disadvantages (or are they?)
  - The normal flow-of-control is interrupted.
  - Specific recovery actions are encouraged.

Example of argument exception

```
public ContactDetails getDetails(String key)
{
    if(key == null) {
        throw new NullPointerException(
                        "null key in getDetails");
    }
    if(key.trim().length() == 0) {
        throw new IllegalArgumentException(
                        "Empty key passed to getDetails");
    }
    return (ContactDetails) book.get(key);
}
```

Error response and recovery

- Clients should take note of error notifications.
  - Check return values.
  - Don't 'ignore' exceptions.
- Include code to attempt recovery.
  - Will often require a loop.

Example of recovery attempt

```
// Try to save the address book.
boolean successful = false;
int attempts = 0;
do {
    try {
        addressbook.saveToFile(filename);
        successful = true;
    }
    catch(IOException e) {
        System.out.println("Unable to save to " + filename);
        attempts++;
        if(attempts < MAX_ATTEMPTS) {
            filename = an alternative file name;
        }
    }
} while(!successful && attempts < MAX_ATTEMPTS);
if(!successful) {
    Report the problem and give up;
}
```

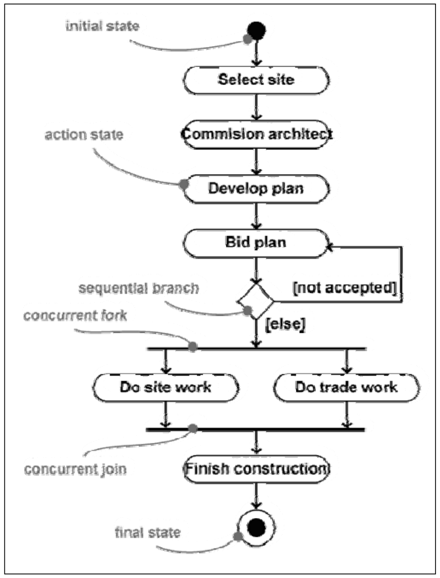
Error avoidance

- Clients can often use server query methods to avoid errors.
  - More robust clients mean servers can be more trusting.
  - Unchecked exceptions can be used.
  - Simplifies client logic.
- May increase client-server coupling.

Construction inside objects

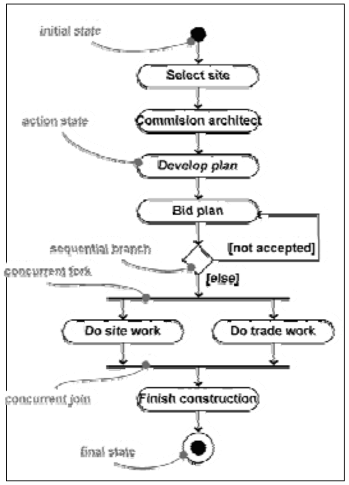
object internals

UML Activity diagram



UML Activity diagram

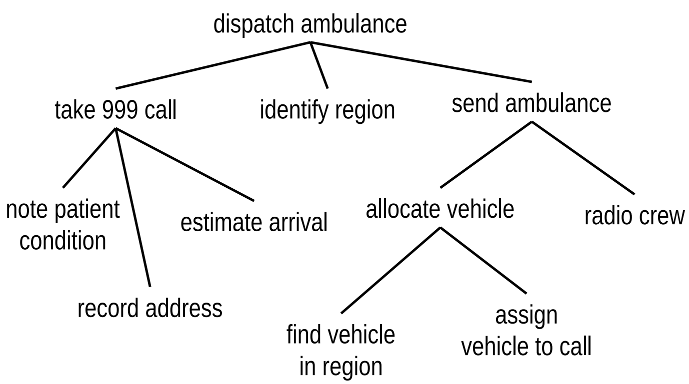
- Like flow charts
  - Activity as action states
- Flow of control
  - transitions
  - branch points
  - concurrency (fork & join)
- Illustrate flow of control
  - high level - e.g. workflow
  - low level - e.g. lines of code



Pioneers – Edsger Dijkstra

- Structured Programming
  - 1968, Eindhoven
- Why are programmers so bad at understanding dynamic processes and concurrency?
  - (ALGOL then – but still hard in Java today!)
- Observed that “go to” made things worse
  - Hard to describe what state a process has reached, when you don’t know which process is being executed.
- Define process as nested set of execution blocks, with fixed entry and exit points

Top-down design & stepwise refinement



Bottom-up construction

- Why?
  - Start with what you understand
  - Build complex structures from well-understood parts
  - Deal with concrete cases in order to understand abstractions
- Study of expert programmers shows that real software design work combines top-down and bottom up.

Modularity at code level

- Is this piece of code (class, method, function, procedure … “routine” in McConnell) needed?
- Define what it will do
  - What information will it hide?
  - Inputs
  - Outputs (including side effects)
  - How will it handle errors?
- Give it a good name
- How will you test it?
- Think about efficiency and algorithms
- Write as comments, then fill in actual code

Modularity in non-OO languages

- Separate source files in C
  - Inputs, outputs, types and interface functions defined by declarations in “header files”.
  - Private variables and implementation details defined in the “source file”
- Modules in ML, Perl, Fortran, …
  - Export publicly visible interface details.
  - Keep implementation local whenever possible, in interest of information hiding, encapsulation, low coupling.

Source code as a design model

- Objectives:
  - ***Accurately*** express logical structure of the code
  - ***Consistently*** express the logical structure
  - Improve ***readability***
- Good visual layout shows program ***structure***
  - Mostly based on white space and alignment
  - The compiler ignores white space
  - Alignment is the single most obvious feature to human readers.
- Like good typography in interaction design: but the “users” are other programmers!

Code as a structured model

```
public int Function_name (int parameter1, int parameter2)

// Function which doesn’t do anything, beyond showing the fact
// that different parts of the function can be distinguished.

    int local_data_A;
    int local_data_B;

    // Initialisation section
    local_data_A = parameter1 + parameter2;
    local_data_B = parameter1 - parameter2;
    local_data_B++;

    // Processing
    while (local_data_A < 40) {
        if ( (local_data_B * 2) > local_data_A ) then {
            local_data_B = local_data_B – 1;
        } else {
            local_data_B = local_data_B + 1;
        }
        local_data_C = local_data_C + 1;
    }
    return local_data_C;
}
```

Expressing local control structure

```
while (local_data_C < 40) {
    form_initial_estimate(local_data_C);
    record_marker(local_data_B – 1);
    refine_estimate(local_data_A);
    local_data_C = local_data_C + 1;
} // end while
```

```
if ( (local_data_B * 2) > local_data_A ) then {
    // drop estimate
    local_data_B = local_data_B – 1;
} else {
    // raise estimate
    local_data_B = local_data_B + 1;
} // end if
```

Expressing structure within a line

- Whitespacealwayshelpshumanreaders
  - `newtotal=oldtotal+increment/missamount-1;`
  - `newtotal = oldtotal + increment / missamount - 1;`
- The compiler doesn’t care – take care!
  - `x = 1   *   y+2   *   z;`
- Be conservative when nesting parentheses
  - `while ( (! error) && readInput() )`
- Continuation lines – exploit alignment
  - ```
    if ( ( aLongVariableName && anotherLongOne ) |
         ( someOtherCondition() ) )
    {
     …
    }
    ```
    ```

Naming variables: Form

- Priority: full and accurate (*not* just short)
  - Abbreviate for pronunciation (remove vowels)
    - e.g. CmptrScnce (leave first and last letters)
- Parts of names reflect conventional functions
  - Role in program (e.g. “count”)
  - Type of operations (e.g. “window” or “pointer”)
  - Hungarian naming (not really recommended):
    - e.g. pscrMenu, ichMin
- Even individual variable names can exploit typographic structure for clarity
  - **`xPageStartPosition`**
  - **`x_page_start_position`**

Naming variables: Content

- Data names describe domain, not computer
  - Describe what, not just how
  - **`CustomerName`** better than **`PrimaryIndex`**
- Booleans should have obvious truth values
  - **`ErrorFound`** better than **`Status`**
- Indicate which variables are related
  - **`CustName`**, **`CustAddress`**, **`CustPhone`**
- Identify globals, types & constants
  - C conventions: g_wholeApplet, T_mousePos
- Even temporary variables have meaning
  - **`Index`**, not **`Foo`**

Pioneers – Michael Jackson

- Jackson Structured Programming
  - 1975, independent consultant, London
- Describe program structure according to the structure of input and output streams
  - Mostly used for COBOL file processing
  - Still relevant to stream processing in Perl
- Data records (items in collection, elements in array) require a code loop
- Variant cases (subtypes, categories, enumerations) require conditional execution
- Switching between code and data perspectives helps to learn about design complexity and to check correctness.

Structural ***roles*** of variables

- Classification of what variables do in a routine
  - Don’t confuse with data types (e.g. int, char, float)
- Almost all variables in simple programs do one of:
  - fixed value
  - stepper
  - most-recent holder
  - most-wanted holder
  - gatherer
  - transformation
  - one-way flag
  - follower
  - temporary
  - organizer
- Most common (70 % of variables) are fixed value, stepper or most-recent holder.

Fixed value

- Value is never changed after initialization
- Example: input radius of a circle, then print area
  - variable r is a *fixed value,* gets its value once, never changes after that.
- Useful to declare “final” in Java (see variable PI).

```
public class AreaOfCircle {

  public static void main(String[] args) {
      final float PI = 3.14F;
      float r;
      System.out.print("Enter circle radius: ");
      r = UserInputReader.readFloat();
      System.out.println(“Circle area is " + PI * r * r);
  }
}
```

Stepper

- Goes through a succession of values in some systematic way
  - E.g. counting items, moving through array index
- Example: loop where multiplier is used as a stepper.
  - outputs multiplication table, stepper goes through values from one to ten.

```
public class MultiplicationTable {

  public static void main(String[] args) {
      int multiplier;
      for (multiplier = 1; multiplier <= 10; multiplier++)
            System.out.println(multiplier + " * 3 = "
                  + multiplier * 3);
  }
}
```

Most-recent holder

- Most recent member of a group, or simply latest input value
- Example: ask the user for input until valid.
  - Variable s is a most-recent holder since it holds the latest input value.

```
public class AreaOfSquare {

  public static void main(String[] args) {
      float s = 0f;
      while (s <= 0) {
            System.out.print("Enter side of square: ");
            s = UserInputReader.readFloat();
      }
      System.out.println(“Area of square is " + s * s);
  }
}
```

Most-wanted holder

- The "best" (biggest, smallest, closest) of values seen.
- Example: find smallest of ten integers.
  - Variable smallest is a *most-wanted holder* since it is given the most recent value if it is smaller than the smallest one so far.
  - (i is a *stepper* and number is a *most-recent holder*.)

```
public class SearchSmallest {
  public static void main(String[] args) {
      int i, smallest, number;
      System.out.print("Enter the 1. number: ");
      smallest = UserInputReader.readInt();
      for (i = 2; i <= 10; i++) {
            System.out.print("Enter the " + i + ". number: ");
            number = UserInputReader.readInt();
            if (number < smallest) smallest = number;
      }
      System.out.println("The smallest was " + smallest);
  }
}
```

Gatherer

- Accumulates values seen so far.
- Example: accepts integers, then calculates mean.
  - Variable sum is a *gatherer* the total of the inputs is gathered in it.
  - (count is a *stepper* and number is a *most-recent holder*.)

```
public class MeanValue {

   public static void main(String[] argv) {
        int count=0;
        float sum=0, number=0;
        while (number != -999) {
                System.out.print("Enter a number, -999 to quit: ");
                number = UserInputReader.readFloat();
                if (number != -999) { sum += number; count++; }
        }
        if (count>0) System.out.println("The mean is " +
                                                sum / count);
   }
}
```

Transformation

- Gets every value by calculation from the value of other variable(s).
- Example: ask the user for capital amount, calculate interest and total capital for ten years.
  - Variable interest is a *transformation* and is always calculated from the capital.
  - (capital is a *gatherer* and i is a *counter*.)

```
public class Growth {
   public static void main(String[] args) {
        float capital, interest;   int i;
        System.out.print("Enter capital (positive or negative): ");
        capital = UserInputReader.readFloat();
        for (i = 1; i <=10; i++) {
                interest = 0.05F * capital;
                capital += interest;
                System.out.println("After "+i+" years interest is "
                        + interest + " and capital is " + capital);
        }
   }
}
```

One-way flag

- Boolean variable which, once changed, never returns to its original value.
- Example: sum input numbers and report if any negatives.
  - The *one-way flag* neg monitors whether there are negative numbers among the inputs. If a negative value is found, it will never return to false.
  - (number is a *most-recent holder* and sum is a *gatherer*.)

```
public class SumTotal {
   public static void main(String[] argv) {
        int number=1, sum=0;
        boolean neg = false;
        while (number != 0) {
                System.out.print("Enter a number, 0 to quit: ");
                number = UserInputReader.readInt();  sum += number;
                if (number < 0) neg = true;
        }
        System.out.println("The sum is " + sum);
        if (neg) System.out.println("There were negative numbers.");
   }
}
```

Follower

- Gets old value of another variable as its new value.
- Example: input twelve integers and find biggest difference between successive inputs.
  - Variable previous is a *follower*, following current.

```
public class BiggestDifference {
   public static void main(String[] args) {
      int month, current, previous, biggestDiff;
      System.out.print("1st: "); previous = UserInputReader.readInt();
      System.out.print("2nd: "); current = UserInputReader.readInt();
      biggestDiff = current - previous;
      for (month = 3; month <= 12; month++) {
        previous = current;
        System.out.print(month + "th: ");
        current = UserInputReader.readInt();
        if (current - previous > biggestDiff)
          biggestDiff = current - previous;
      }
      System.out.println("Biggest difference was " + biggestDiff);
   }
}
```

Temporary

- Needed only for very short period (e.g. between two lines).
- Example: output two numbers in size order, swapping if necessary.
  - Values are swapped using a *temporary* variable tmp whose value is later meaningless (no matter how long the program would run).

```
public class Swap {
    public static void main(String[] args) {
        int number1, number2, tmp;
        System.out.print("Enter num: ");
        number1 = UserInputReader.readInt();
        System.out.print("Enter num: ");
        number2 = UserInputReader.readInt();
        if (number1 > number2) {
                tmp = number1;
                number1 = number2;
                number2 = tmp;
        }
        System.out.println(“Order is " + number1 + “," + number2 + ".");
    }
}
```

Organizer

- An array for rearranging elements
- Example: input ten characters and output in reverse order.
  - The reversal is performed in *organizer* variable word.
  - tmp is a *temporary* and i is a *stepper*.)

```
public class Reverse {
    public static void main(String[] args) {
        char[] word = new char[10];
        char tmp; int i;
        System.out.print("Enter ten letters: ");
        for (i = 0; i < 10; i++) word[i] = UserInputReader.readChar();
        for (i = 0; i < 5; i++) {
                tmp = word[i];
                word[i] = word[9-i];
                word[9-i] = tmp;
        }
        for (i = 0; i < 10; i++) System.out.print(word[i]);
        System.out.println();
    }
}
```

Verifying variables by role

- Many student program errors result from using the same variable in more than one role.
  - Identify role of each variable during design
- There are opportunities to check correct operation according to constraints on role
  - Check stepper within range
  - Check most-wanted meets selection criterion
  - De-allocate temporary value
  - Confirm size of organizer array is invariant
  - Use compiler to guarantee final fixed value
- Either do runtime safety checks (noting efficiency tradeoff), or use language features.

Type-checking as modeling tool

- Refine types to reflect meaning, not just to satisfy the compiler (C++ example below)
- Valid (to compiler), but incorrect, code:
  - **`float totalHeight, myHeight, yourHeight;`**
  - **`float totalWeight, myWeight, yourWeight;`**
  - **`totalHeight = myHeight + yourHeight + myWeight;`**
- Type-safe version:
  - **`type t_height, t_weight: float;`**
  - **`t_height totalHeight, myHeight, yourHeight;`**
  - **`t_weight totalWeight, myWeight, yourWeight;`**
  - **`totalHeight = myHeight + yourHeight + myWeight;`**

Compile error!

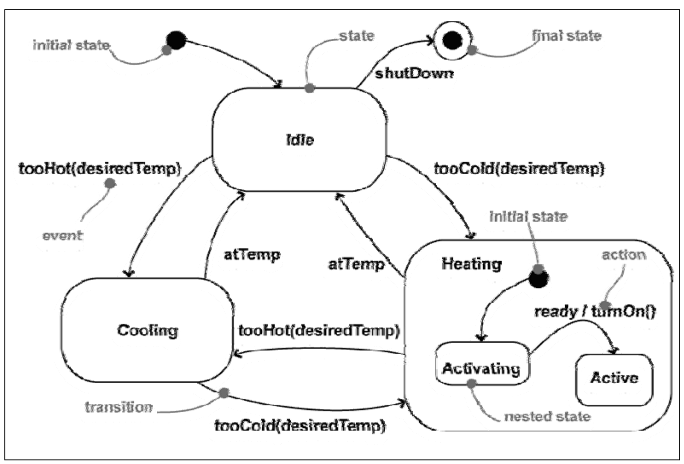
Language support for user types

- Smalltalk
  - All types are classes – consistent, but inefficient
- C++
  - Class overhead very low
  - User-defined types have no runtime cost
- Java
  - Unfortunately a little inefficient
  - But runtime inefficiency in infrequent calculations far better than lost development time.

Construction of data lifecycles

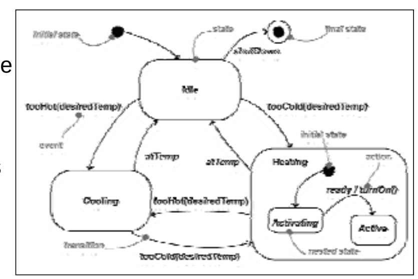
object state

UML Statechart diagram



UML Statechart diagram

- Object lifecycle
  - data as state machine
- Harel statecharts
  - nested states
  - concurrent substates
- Explicit initial/final
  - valuable in C++
- Note inversion of activity diagram



Maintaining valid system state

- Pioneers (e.g. Turing) talked of proving program correctness using mathematics
- In practice, the best we can do is confirm that the state of the system is consistent
  - State of an object valid before and after operation
  - Parameters and local variables valid at start and end of routine
  - Guard values define state on entering & leaving control blocks (loops and conditionals)
  - Invariants define conditions to be maintained throughout operations, routines, loops.

Pioneers – Tony Hoare

- Assertions and proof
  - 1969, Queen's University Belfast
- Program element behaviour can be defined
  - by a *post-condition* that will result …
  - … given a known *pre-condition*.
- If prior and next states accurately defined:
  - Individual elements can be composed
  - Program correctness is potentially provable

Formal models: Z notation

$$\begin{array}{|l}
\textit{BirthdayBook} \\
\hline
\textit{known} : \mathbb{P}\, \textit{NAME} \\
\textit{birthday} : \textit{NAME} \nrightarrow \textit{DATE} \\
\hline
\textit{known} = \operatorname{dom} \textit{birthday} \\
\hline
\end{array}$$

- Definitions of the *BirthdayBook* state space:
  - *known* is a set of NAMEs
  - *birthday* is a partial map from NAMEs to DATEs
- Invariants:
  - *known* must be the domain of *birthday*

Formal models: Z notation

$$\begin{array}{|l}
\textit{AddBirthday} \\
\hline
\Delta \textit{BirthdayBook} \\
\textit{name?} : \textit{NAME} \\
\textit{date?} : \textit{DATE} \\
\hline
\textit{name?} \notin \textit{known} \\
\textit{birthday}' = \textit{birthday} \cup \{\textit{name?} \mapsto \textit{date?}\} \\
\hline
\end{array}$$

- An operation to change state
  - *AddBirthday* modifies the state of *BirthdayBook*
  - Inputs are a new *name* and *date*
  - Precondition is that *name* must not be previously known
  - Result of the operation, *birthday'* is defined to be a new and enlarged domain of the *birthday* map function

Formal models: Z notation

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Remind}} \\ \hline
\Xi \textit{BirthdayBook} \\
\textit{today?} : \textit{DATE} \\
\textit{cards!} : \mathbb{P}\, \textit{NAME} \\ \hline
\textit{cards!} = \{\, n : \textit{known} \mid \textit{birthday}(n) = \textit{today?} \,\} \\ \hline
\end{array}
$$

- An operation to inspect state of *BirthdayBook*
  - This schema does not change the state of *BirthdayBook*
  - It has an output value (a set of people to send *cards* to)
  - The output set is defined to be those people whose birthday is equal to the input value *today*.

Advantages of formal models

- Requirements can be analysed at a fine level of detail.
- They are declarative (specify what the code should do, not how), so can be used to check specifications from an alternative perspective.
- As a mathematical notation, offer the promise of tools to do automated checking, or even proofs of correctness (“verification”).
- They have been applied in some real development projects.

Disadvantages of formal models

- Notations that have lots of Greek letters and other weird symbols look scary to non-specialists.
  - Not a good choice for communicating with clients, users, rank-and-file programmers and testers.
- Level of detail (and thinking effort) is similar to that of code, so managers get impatient.
  - If we are working so hard,
    why aren’t we just writing the code?
- Tools are available, but not hugely popular.
  - Applications so far in research / defence / safety critical
- Pragmatic compromise from UML developers
  - “Object Constraint Language” (OCL).
  - Formal specification of some aspects of the design, so that preconditions, invariants etc. can be added to models.

Language support for assertions

- Eiffel (pioneering OO language)
  - supported pre- and post-conditions on every method.
- C++ and Java support “assert” keyword
  - Programmer defines a statement that must evaluate to boolean true value at runtime.
  - If assertion evaluates false, exception is raised
- Some languages have debug-only versions, turned off when system considered correct.
  - Dubious trade-off of efficiency for safety.
- Variable roles could provide rigorous basis for fine-granularity assertions in future.

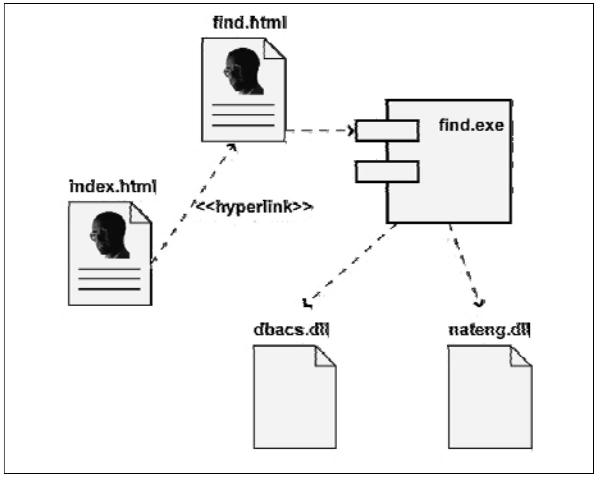
Defensive programming

- Assertions and correctness proofs are useful tools, but not always available.
- Defensive programming includes additional code to help ensure local correctness
  - Treat function interfaces as a contract
- Each function / routine
  - Checks that input parameters meet assumptions
  - Checks output values are valid
- System-wide considerations
  - How to report / record detected bugs
  - *Perhaps* include off-switch for efficiency

Construction using objects

components

UML Component diagram



Component documentation

- Your own classes should be documented the same way library classes are.
- Other people should be able to use your class without reading the implementation.
- Make your class a 'library class'!

Elements of documentation

Documentation for a class should include:

- the class name
- a comment describing the overall purpose and characteristics of the class
- a version number
- the authors’ names
- documentation for each constructor and each method

Elements of documentation

The documentation for each constructor and method should include:

- the name of the method
- the return type
- the parameter names and types
- a description of the purpose and function of the method
- a description of each parameter
- a description of the value returned

javadoc

- Part of the Java standard
- Each class and method can include special keywords in a comment explaining the interface to that class
- During javadoc compilation, the keyword information gets converted to a consistent reference format using HTML
- The documentation for standard Java libraries is all generated using javadoc

javadoc example

Class comment:

```
/**
 * The Responder class represents a response
 * generator object. It is used to generate an
 * automatic response.
 *
 * @author     Michael Kölling and David J. Barnes
 * @version    1.0  (1.Feb.2002)
 */
```

javadoc example

Method comment:

```
/**
 * Read a line of text from standard input (the text
 * terminal), and return it as a set of words.
 *
 * @param  prompt  A prompt to print to screen.
 * @return  A set of Strings, where each String is
 *          one of the words typed by the user
 */
public HashSet getInput(String prompt)
{
   ...
}
```

Transition

testing and optimisation

What is the goal of testing?

- A) To define the end point of the software development process as a managed objective?
- B) To prove that the programmers have implemented the specification correctly?
- C) To demonstrate that the resulting software product meets defined quality standards?
- D) To ensure that the software product won't fail, with results that might be damaging?

Testing and quality

- Wikipedia
  - "Software testing is the process used to assess the ***quality*** of computer software. It is an empirical technical investigation conducted to provide stakeholders with information about the ***quality*** of the product or service under test, with respect to the context in which it is intended to operate."
- Edsger Dijkstra
  - "Program testing can be used to show the ***presence*** of bugs, but ***never*** to show their absence"

Remember design as learning?

- Design is the process of learning about a problem and describing a solution
  - at first with many gaps …
  - eventually in sufficient detail to build it.
- We describe both the problem and the solution in a series of *design models*.
- Testing those models in various ways helps us gather more knowledge.
- Source code is simply the most detailed model used in software development.

Learning through testing

A bug is a system’s way of telling you that you don’t know something (P. Armour)

- Testing searches for the ***presence*** of bugs.
- Later: ‘debugging’ searches for the ***cause*** of bugs, once testing has found that a bug exists.
  - The manifestation of an bug as observable behaviour of the system may well occur some ‘distance’ from its cause.

Testing principles

- Look for violations of the interface contract.
  - Aim is to find bugs, ***not*** to prove that unit works as expected from its interface contract
  - Use positive tests (expected to pass) in the hope that they ***won’t*** pass
  - Use negative tests (expected to fail) in the hope that they ***don’t*** fail
- Try to test *boundaries* of the contract
  - e.g. zero, one, overflow, search empty collection, add to a full collection.

Unit testing priorities

- Concentrate on modules most likely to contain errors:
  - Particularly complex
  - Novel things you’ve not done before
  - Areas known to be error-prone
- Some habits in unit test ordering
  - Start with small modules
  - Try to get input/output modules working early
    - Allows you to work with real test data
  - Add new ones gradually
  - You probably want to test critical modules early
    - For peace of mind, not because you expect errors

How to do it: testing strategies

- Manual techniques
  - Software inspections and code walkthrough
- Black box testing
  - Based on specified unit interfaces, not internal structure, for test case design
- White box testing
  - Based on knowing the internal structure
- Stress testing
  - At what point will it fail?
- 'Random' (unexpected) testing
  - Remember the goal: most errors in least time

Pioneers – Michael Fagan

- Software Inspections
  - 1976, IBM
- Approach to design checking, including planning, control and checkpoints.
- Try to find errors in design and code by systematic *walkthrough*
- Work in teams including designer, coder, tester and moderator.

Software inspections

- A low-tech approach, relatively underused, but more powerful than appreciated.
- Read the source code in execution order, acting out the role of the computer
  - High-level (step) or low-level (step-into) views.
- An expert tries to find common errors
  - Array bound errors
  - Off-by-one errors
  - File I/O (and threaded network I/O)
  - Default values
  - Comparisons
  - Reference versus copy

Inspection by yourself

- Get away from the computer and 'run' a program by hand
- Note the current object state on paper
- Try to find opportunities for incorrect behaviour by creating incorrect state.
- Tabulate values of fields, including invalid combinations.
- Identify the state changes that result from each method call.

Black box testing

- Based on interface specifications for whole system or individual modules
- Analyse input ranges to determine test cases
- Boundary values
  - Upper and lower bounds for each value
  - Invalid inputs outside each bound
- Equivalence classes
  - Identify data ranges and combinations that are 'known' to be equivalent
  - Ensure each equivalence class is sampled, but not over-represented in test case data

White box testing

- Design test cases by looking at internal structure, including all possible bug sources
  - Test each independent path at least once
  - Prepare test case data to force paths
  - Focus on error-prone situations (e.g. empty list)
  - The goal is to find as many errors as you can
- Control structure tests:
  - conditions – take each possible branch
  - data flow – confirm path through parameters
  - loops – executed zero, one, many times
  - exceptions – ensure that they occur

Stress testing

- The aim of stress testing is to find out *at what point* the system will fail
  - You really *do* want to know what that point is.
  - You *have to keep going* until the system fails.
  - If it hasn't failed, you haven't done stress testing.
- Consider both volume and speed
- Note difference from *performance testing*, which aims to confirm that the system will perform as specified.
  - Used as a contractual demonstration
  - It's not an efficient way of finding errors

Random testing

- There are far more combinations of state and data than can be tested exhaustively
- Systematic test case design helps explore the range of possible system behaviour
  - But remember the goal is to make the system fail, not to identify the many ways it works correctly.
- Experienced testers have an instinct for the kinds of things that make a system fail
  - Usually by thinking about the system in ways the programmer did not expect.
  - Sometimes, just doing things at random can be an effective strategy for this.

Regression testing

- 'Regression' is when you go backwards, or things get worse
  - Regression in software usually results from re-introducing faults that were previously fixed.
  - Each bug fix has around 20% probability of reintroducing some other old problem.
  - Refactoring can reintroduce design faults
- So regression testing is designed to ensure that a new version gives the same answers as the old version did

Regression testing

- Use a large database of test cases
- Include all bugs reported by customers:
  - customers are much more upset by failure of an already familiar feature than of a new one
  - reliability of software is relative to a set of inputs, so better test inputs that users actually generate!
- Regression testing is boring and unpopular
  - test automation tools reduce mundane repetition
  - perhaps biggest single advance in tools for software engineering of packaged software

Test automation

- Thorough testing (especially regression testing) is time consuming and repetitive.
- Write special classes to test interfaces of other classes automatically
  - "test rig" or "test harness"
  - "test stubs" substitute for unwritten code, or simulate real-time / complex data
- Use standard tools to exercise external API, commands, or UI (e.g. mouse replay)
  - In commercial contexts, often driven from build and configuration tools.

Unit testing

- Each unit of an application may be tested.
  - Method, class, interface, package
- Can (should) be done *during* development.
  - Finding and fixing early lowers development costs (e.g. programmer time).
  - Build up a test suite of necessary harnesses, stubs and data files
- JUnit is often used to manage and run tests
  - you will use this to check your practical exercises
  - www.junit.org

Fixing bugs – ‘debugging’

- Treat debugging as a series of experiments
  - As with testing, debugging is about learning things
- Don’t just make a change in the hope that it might fix a bug
  - Form a hypothesis of what is causing the unexpected behaviour
  - Make a change that is designed to test the hypothesis
    - If it works – good, the bug is fixed
    - If not – good, you’ve learned something
  - Either way, remember to check what else you broke

Debugging strategy

- Your goal is to understand the nature of the error, not disguise the resulting symptom
- Step 1: THINK
  - Which is the relevant data?
  - Why is it behaving that way?
  - Which part is correct, and which incorrect?
- Step 2: search and experiment
  - Backtrack from the place that is incorrect
  - Make tests on local state in each place
  - Try to localise changes

Print statements

- The most popular debugging technique.
- No special tools required.
- All programming languages support them.
- But often badly used …
  - Printing things at random in hope of seeing something wrong
- Instead:
  - Make a hypothesis about the cause of a bug
  - Use a print statement to test it
- Output may be voluminous in loops
  - Turning off and on requires forethought.

Debugging walkthroughs

- Read through the code, explaining what state changes will result from each line.
- Explain to someone else what the code is doing.
  - They might spot the error.
  - The process of explaining might help you to spot it for yourself (the cardboard software engineer)
- Can be done on-screen from source code, on paper (as in a software inspection), or using a debugger

Debuggers

- Usual features include:
  - Breakpoints
    - As with print statements, can be used to test state at a particular program point, but can then also …
  - Step-over or step-into methods/routines
    - Identify specific routine or statement responsible for unexpected effect.
  - Call sequence (stack) inspectors
    - Explore parameters preceding unexpected effect
  - Object and variable state inspectors
    - Also continuous “watch” windows.
- However, debuggers are both language-specific and environment-specific.

If all else fails …

- Sleep on it.

Classic testing advice

- The Art of Software Testing
  - Glenford J. Myers
  - John Wiley, 1979
- Seven Principles of Software Testing
  - Bertrand Meyer, ETH Zürich and Eiffel Software
  - IEEE Computer, August 2008, 99-101
- Slightly interesting notes
  - Myers and Meyer are different people
  - Meyer was the inventor of the Eiffel language

Myers’ classic book

Table 2.1
Vital Program Testing Guidelines

Principle Number	*Principle*
1	A necessary part of a test case is a definition of the expected output or result.
2	A programmer should avoid attempting to test his or her own program.
3	A programming organization should not test its own programs.
4	Thoroughly inspect the results of each test.
5	Test cases must be written for input conditions that are invalid and unexpected, as well as for those that are valid and expected.
6	Examining a program to see if it does not do what it is supposed to do is only half the battle; the other half is seeing whether the program does what it is not supposed to do.
7	Avoid throwaway test cases unless the program is truly a throwaway program.
8	Do not plan a testing effort under the tacit assumption that no errors will be found.
9	The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.
10	Testing is an extremely creative and intellectually challenging task.

Myers' 10 principles

- A necessary part of a test case is a definition of the expected output or result.
- A programmer should avoid attempting to test his or her own program.
- A programming organisation should not test its own programs.
- Thoroughly inspect the results of each test.

Myers' 10 principles (cont.)

- Test cases must be written for input conditions that are invalid and unexpected, as well as for those that are valid and expected.
- Examining a program to see if it does not do what it is supposed to do is only half the battle; the other half is seeing whether the program does what it is not supposed to do.
- Do not plan a testing effort under the tacit assumption that no errors will be found.

Myers' 10 principles (cont.)

- Avoid throwaway test cases unless the program is truly a throwaway program.
- The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.
- Testing is an extremely creative and intellectually challenging task.

Meyer's new classic article


SOFTWARE TECHNOLOGIES

Seven Principles of Software Testing

Bertrand Meyer, ETH Zürich and Eiffel Software

Testing is about producing failures.

While everyone knows the theoretical limitations of software testing, in practice we devote considerable effort to this task and would consider it foolish or down-

for testing, echoed in the Wikipedia definition (http://en.wikipedia.org/wiki/Software_testing): "Software testing is the process used to assess the quality of computer software. Software testing is an empirical technical investigation conducted to

have evidenced 550, 540, and 530 faults, the trend is encouraging, but the next run is unlikely to find no faults, or 100. (Mathematical reliability models allow more precise estimates, credible in the presence of a sound long-term data collection process.)

The only incontrovertible connection is negative, a falsification in the Popperian sense: A failed test gives us evidence of nonquality. In addition, if the test previously passed, it indicates regression and points to possible quality problems in the program and the development process. The most famous quote about testing expressed this memorably: "Program testing," wrote Edsger Dijkstra, "can be used to show the presence of bugs, but never to show their absence!"

Less widely understood (and probably not intended by Dijkstra) is what this means for testers: the best possible self-advertisement. Surely, any technique that uncovers faults holds great interest for all "stakeholders," from managers to developers and customers.

Rather than an indictment, we

Meyer’s 7 principles

- Principle 1: Definition
  - To test a program is to try to make it fail.
- Principle 2: Tests versus specs
  - Tests are no substitute for specifications.
- Principle 3: Regression testing
  - Any failed execution must yield a test case, to remain a permanent part of the project’s test suite.

Meyer’s 7 principles (cont.)

- Principle 4: Applying ‘oracles’
  - Determining success or failure of tests must be an automatic process.
- Principle 4 (variant): Contracts as oracles
  - Oracles should be part of the program text, as contracts. Determining test success or failure should be an automatic process consisting of monitoring contract satisfaction during execution.
- Principle 5: Manual and automatic test cases
  - An effective testing process must include both manually and automatically produced test cases.

Meyer’s 7 principles (cont.)

- Principle 6: Empirical assessment of testing strategies
  - Evaluate any testing strategy, however attractive in principle, through objective assessment using explicit criteria in a reproducible testing process.
- Principle 7: Assessment criteria
  - A testing strategy’s most important property is the number of faults it uncovers as a function of time.

Cost of testing

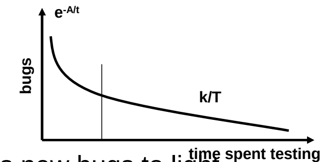
- Testing can cost as much as coding
- Cost of rectifying bugs rises dramatically in later phases of a project:
  - When validating the initial design – moments
  - When testing a module after coding – minutes
  - When testing system after integration – hours
  - When doing field trials – days
  - In subsequent litigation – years!
  - ...
- Testing too late is a common failing
- Save time and cost by design for early testing

When to stop testing

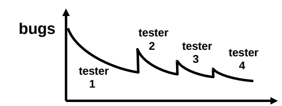
- Imagine you are working on a project in which the timetable has allocated three months to testing.
- When testing, you successfully find:
  - 400 bugs in the first month
  - 200 bugs in the second month
  - 100 bugs in the third month
- What are the chances that you have found all the bugs?
  - Managing a large-scale testing process requires some kind of statistical model.
- But ***not*** a good idea to use this as an incentive for release targets, productivity bonuses etc
  - Programmers are smart enough to figure out basic statistics if there is money involved.

When to stop testing

- Reliability growth model helps assess
  - mean time to failure
  - number of bugs remaining
  - economics of further testing, ...
- Software failure rate
  - drops exponentially at first
  - then decreases as K/T



- But changing testers brings new bugs to light



Other system tests

- Security testing
  - automated probes, or
  - a favour from your Russian friends
- Efficiency testing
  - test expected increase with data size
  - use code profilers to find hot spots
- Usability testing
  - essential to product success
  - will be covered in further detail in Part II

Testing efficiency: optimisation

- Worst error is using wrong algorithm
  - e.g. lab graduate reduced 48 hours to 2 minutes
  - Try different size data sets – does execution time vary as N, 2N, N^2, N^3, N^4, k^N ...?
- If this is the best algorithm, and you know it scales in a way appropriate to your data, but still goes too slow for some reason, ask:
  - How often will this program / feature be run?
  - Hardware gets faster quickly
  - Optimisation may be a waste of ***your*** time

Testing efficiency: optimisation

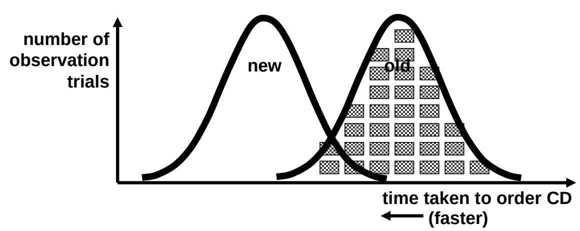
- When optimisation is required
  - First: check out compiler optimisation flags
  - For *some parts* of extreme applications
    - Use code profiler to find hotspots/bottlenecks
    - Most likely cause: overuse of some library/OS function
  - When pushing hardware envelope
    - Cache or pre-calculate critical data
    - Recode a function in C or assembler
    - Use special fast math tricks & bit-twiddling
    - Unroll loops (but compilers should do this)
- But if this is an interactive system …
  - … how fast will the user be?

User interface efficiency

- Usability testing can measure speed of use
  - How *long* did Fred take to order a book from Amazon?
  - How many *errors* did he make?
- But every observation is different.
  - Fred might be faster (or slower) next time
  - Jane might be consistently faster
- So we compare averages:
  - over a number of trials
  - over a range of people (experimental subjects)
- Results usually have a normal distribution

Experimental usability testing

- Experimental *treatment* is some change that we expect to have an effect on usability:
  - Hypothesis: we expect new interface to be faster (& produce less errors) than old one



- Expected answer: *usually* faster, but not *always*

Debugging user errors

- Assess a user's conceptual model of system
  - Important to find *typical* sample users
- Users talk continuously while performing a defined experimental task: "think-aloud"
  - record to audio/video + screen capture
  - transcribed for detailed study of what user thinks is happening
  - code and classify events
  - look for *breakdowns* in usage/understanding.
- Can be used to assess usability of prototypes, even "paper prototypes"

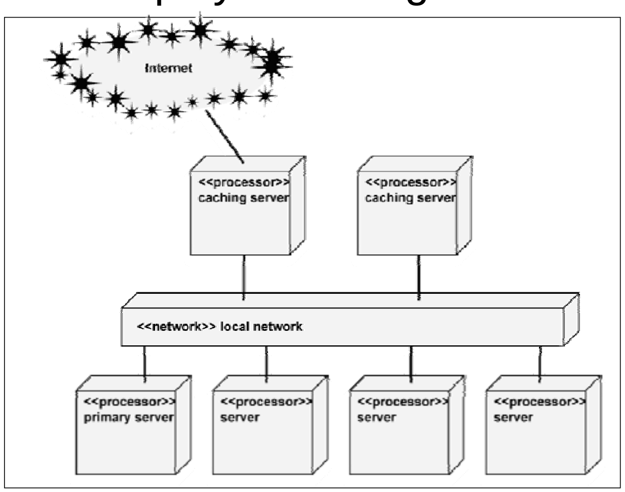
Usability testing in the field

- Brings advantages of ethnography / contextual task analysis to testing phase of product development.
- Case study: Intuit Inc.'s *Quicken* product
  - originally based on interviews and observation
  - follow-me-home programme after product release:
    - random selection of shrink-wrap buyers;
    - observation while reading manuals, installing, using.
  - Quicken success was attributed to the programme:
    - survived predatory competition, later valued at $15 billion.

Iterative Development

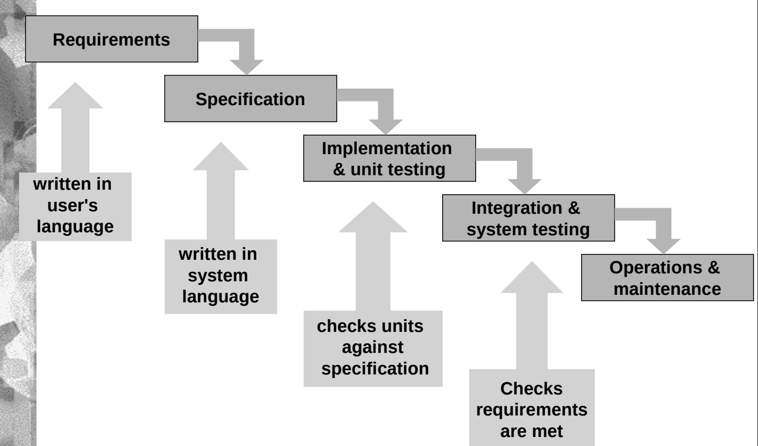
within any design phase or any combination of phases

UML Deployment diagram

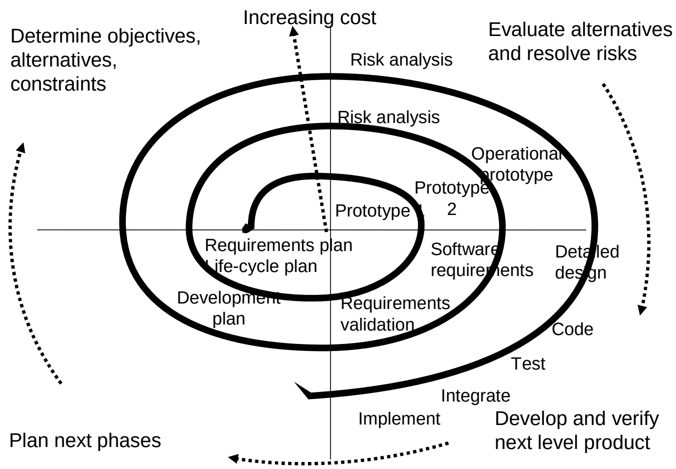


The Waterfall Model

- (Royce, 1970; now US DoD standard)



Spiral model (Boehm, 88)



Prototyping

- Supports early investigation of a system.
  - Early problem identification.
- Incomplete components can be simulated.
  - e.g. always returning a fixed result.
  - May want to avoid random or time-dependent behavior which is difficult to reproduce.
- Allows early interaction with clients
  - Perhaps at inception phase of project
  - Especially (if feasible) with actual users!
- In product design, creative solutions are discovered by building many prototypes

Prototyping product concepts

- Emphasise appearance of the interface, create some behaviour with scripting functions:
  - Visio – diagrams plus behaviour
  - Animation tools – movie sequence
  - JavaScript – simulate application as web page
  - PowerPoint – 'click-through' prototype
- Cheap prototypes are good prototypes
  - More creative solutions are often discovered by building more prototypes.
  - Glossy prototypes can be mistaken for the real thing – either criticised more, or deployed!

Prototypes without programming

- Low-fidelity prototypes (or mockups)
  - Paper-and-glue simulation of interface
  - User indicates action by pointing at buttons on the paper "screen"
  - Experimenter changes display accordingly
- "*Wizard of Oz*" simulation method
  - Computer user interface is apparently operational
  - Actual system responses are produced by an experimenter in another room.
  - Can cheaply assess effects of "intelligent" interfaces

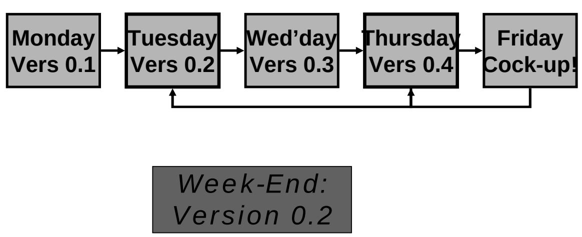
Software continues changing

- Even after project completion!
- There are only two options for software:
  - Either it is continuously maintained …
  - … or it dies.
- Software that cannot be maintained will be thrown away.
  - Not like a novel (written then finished).
  - Software is extended, corrected, maintained, ported, adapted…
- The work will be done by different people over time (often decades).

Configuration management

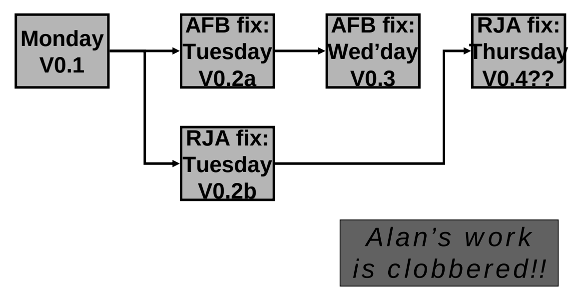
- Version control
- Change control
- Variants
- Releases

Version control



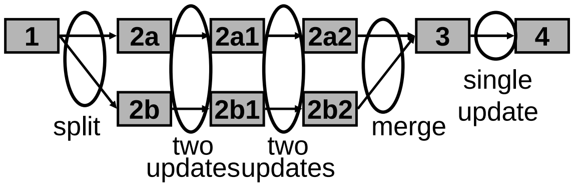
- Record regular "snapshot" backups
  - often appropriate to do so daily
- Provides ability to "roll back" from errors
- Useful even for programmers working alone

Change control



- Essential in programming teams
- Avoid the "clobbering" problem
  - Older tools (RCS, SCCS) rely on locking
  - More recent (CVS) automate merging

Variants from branch fixes



- Branching (from local fixes) results in a tree of different versions or “variants”
- Maintaining multiple branches is costly
  - Merge branches as often as possible
  - Minimise number of components that vary in each branch (ideally only one configuration file)
  - If necessary, conditional compile/link/execution can merge several variants into one

Builds and Releases

- Record actual configuration of components that were in a product release, or an overnight build integrating work of a large team.
  - Allows problems to be investigated with the same source code that was delivered or tested
  - Often includes regression testing as part of build process
- Also allow start of development on next release while testing and supporting current release
  - Universal requirement of commercial software development (at least after release 1.0!)
  - Bug fixes made to 1.0.1 are also expected to be there in 2.0, which requires regular merging
- Think about this: ‘About Internet Explorer’ reported: 6.0.2900.2180.xpsp2.070227-2254

Localizing change

- One aim of reducing coupling and responsibility-driven design is to localize change.
- When a change is needed, as few classes as possible should be affected.
- Thinking ahead
  - When designing a class, think what changes are likely to be made in the future.
  - Aim to make those changes easy.
- When you fail (and you will), *refactoring* is needed.

Refactoring

- When classes are maintained, code is often added.
  - Classes and methods tend to become longer.
- Every now and then, classes and methods should be refactored to maintain cohesion and low coupling.
  - e.g. move duplicated methods into a superclass
- Often removes code duplication, which:
  - is an indicator of bad design,
  - makes maintenance harder,
  - can lead to introduction of errors during maintenance.

Refactoring and testing

- When refactoring code, it is very important to separate the refactoring from making other changes.
  - First do the refactoring only, without changing the functionality.
  - Then make functional changes after refactored version shown to work OK.
- ***Essential*** to run regression tests before and after refactoring, to ensure that nothing has been broken.

Beyond waterfalls and spirals

- User-centred design
- Participatory design
- Agile development: 'XP'

User-centred Design

- Focus on 'end-users', not just specifications from contract and/or client
- Use ethnographic methods at inception stage
- Design based on user conceptual models
- Early prototyping to assess conceptual model
- Contextual evaluation to assess task relevance
- Frequent iteration

Participatory Design

- Users become partners in the design team
  - Originated in Scandinavian printing industry
  - Now used in developing world, with children, …
- PICTIVE method
  - Users generate scenarios of use in advance
  - Low fidelity prototyping tools (simple office supplies) are provided for collaborative session
  - The session is videotaped for data analysis
- CARD method
  - Cards with screen-dumps on them are arranged on a table to explore workflow options

Xtreme Programming' (XP)

- Described in various books by Kent Beck
- An example of an *agile* design methodology
  - Increasingly popular alternative to more "corporate" waterfall/spiral models.
- Reduce uncertainty by getting user feedback as soon as possible, but using actual code
  - Typical team size = two (*pair programming*).
  - Constant series of updates, maybe even daily.
  - Respond to changing requirements and understanding of design by refactoring.
- When used on large projects, some evidence of XD (Xtreme Danger)!

Would XP have helped CAPSA?

- Now Cambridge University Financial System
- Previous systems:
  - In-house COBOL system 1966-1993
    - Didn't support commitment accounting
  - Reimplemented using Oracle package 1993
    - No change to procedures, data, operations
- First (XP-like?) attempt to change:
  - Client-server "local" MS Access system
  - To be "synchronised" with central accounts
  - Loss of confidence after critical review
- May 1998: consultant recommends restart with "industry standard" accounting system

CAPSA project

- Detailed requirements gathering exercise
  - Input to supplier choice between Oracle vs. SAP
- Bids & decision both based on optimism
  - 'vapourware' features in future versions
  - unrecognised inadequacy of research module
  - no user trials conducted, despite promise
- Danger signals
  - High 'rate of burn' of consultancy fees
  - Faulty accounting procedures discovered
  - New management, features & schedule slashed
  - Bugs ignored, testing deferred, system went live
- "Big Bang" summer 2000: CU seizes up

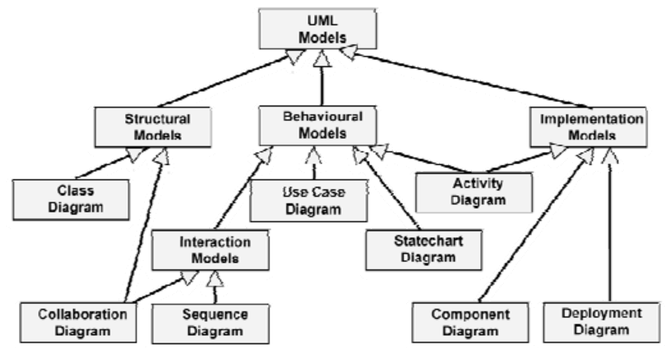
CAPSA mistakes

- No phased or incremental delivery
- No managed resource control
- No analysis of risks
- No library of documentation
- No direct contact with end-users
- No requirements traceability
- No policing of supplier quality
- No testing programme
- No configuration control

CAPSA lessons

- Classical system failure (Finkelstein)
  - More costly than anticipated
    - £10M or more, with hidden costs
  - Substantial disruption to organisation
  - Placed staff under undue pressure
  - Placed organisation under risk of failing to meet financial and legal obligations
- Danger signs in process profile
  - Long hours, high staff turnover etc
- Systems fail systemically
  - not just software, but interaction with organisational processes

UML review: Modelling for uncertainty



The ‘quick and dirty’ version

- Plan using general UML phase principles
- Make sure you visit / talk to end-users
  - show them pictures of proposed screens
- Write use case “stories”
  - note the parts that seem to be common
- Keep a piece of paper for each class
  - write down attributes, operations, relationships
  - lay them out on table, and “talk through” scenarios
- Think about object multiplicity and lifecycle
  - collections, state change, persistence
- Test as early as possible

Software Design: beyond “correct”

The requirements for design conflict and cannot be reconciled. All designs for devices are in some degree failures, either because they flout one or another of the requirements or because they are compromises, and compromise implies a degree of failure ... quite specific conflicts are inevitable once requirements for economy are admitted; and conflicts even among the requirements of use are not unknown. It follows that all designs for use are arbitrary. The designer or his client has to choose in what degree and where there shall be failure. … It is quite impossible for any design to be the “logical outcome of the requirements” simply because, the requirements being in conflict, their logical outcome is an impossibility.

David Pye, *The Nature and Aesthetics of Design* (1978).