7042 Praktikum in Software Engineering

Prof. O. Nierstrasz

Sommersemester 1998

1. Praktikum — Software Engineering

Lecturer: Prof. Oscar Nierstrasz
Office: Schützenmattstr. 14/103
Tel. 631.4618
Email: oscar@iam.unibe.ch

Secretary: Frau I. Huber, Tel. 631.4692

Assistants: Markus Lumpe, Marc Heissenbüttel, Thomas Hofmann

WWW: http://www.iam.unibe.ch/~scg

Overview

- Goals of this workshop
- Project overview
- Schedule: *milestones, deliverables*
- Analysis and Design documents: *guidelines*
- Prototyping: *requirements validation and iterative development*
- Testing: *coverage and regression tests*
- Teamwork: *roles and responsibilities*
- Tools: *UML, rcs, make, SNiFF+, ...*

Goals of this Workshop ...

Methodological skills

- Practising Requirements Collection and Specification
- Practising Responsibility-Driven Design
- Evaluating Implementation Strategies
- Prototyping

Practical skills

- Working with incomplete requirements
- Developing a complete product
- Teamwork

Technical skills

- Using UML
- Programming with Java
- Testing

Project Overview

Programme Committee Support System

- Submission of papers by authors
- Bidding for papers to review
- Distribution and submission of review forms
- Conflict detection and report generation
- Keeping track of accepted/rejected papers
- Access rights

Wish list

- Both WWW and email support
- Java as implementation language of choice (alternatives?)
- Adaptable to different review forms and review procedures

More details to follow ...

Project Characteristics

Several characteristics of “real” projects:

- Open, changing requirements
- Existing procedures in place
- Distributed, multi-platform
- New technology, methods imposed
- Ideal choice of implementation platform unclear

Non-issues:

- No legacy data or software
- No integration with existing applications

Schedule

	Meeting	*Homework/Consultation*
1) 25.03.98	Introduction; form teams	Prepare interview questions
2) 01.04.98	Requirements interview; introduction to Java	Specify requirements; prepare presentation; start prototyping ...
3) 08.04.98	Deliver and present initial requirements spec; feedback	Revise requirements spec; specify coarse design (architecture)
4) 15.04.98	Deliver design documents and revised requirements; exchange designs with other teams	Prepare design review (of competing design); start prototyping UI
5) 22.04.98	Design review; feedback	Revise coarse design; prepare detailed design
6) 29.04.98	Deliver detailed design spec; ...	Prototype UI; design test cases; schedule lab sessions
7) 06.05.98	Validate UI and test cases; trouble-shooting (lab)	Implementation

	Meeting	*Homework/Consultation*
8) 13.05.98	Validate UI and test cases; trouble-shooting (lab)	Implementation ...
9) 20.05.98	*Open* ...	
10) 27.05.98	*Open* ...	
11) 03.06.98	*Open* ...	
12) 10.06.98	Deliver final application (source) and documentation (including final A&D docs)	
13) 17.06.98	Feedback and Testat	
14) 24.06.98		

Analysis and Design

Requirements Specification

- Describe *what* is required, not *how* it will be implemented — don't confuse requirements specification and design!
- Formalize scenarios, domain objects, functional and non-functional requirements
- Use UML to formalize your models; use text to explain them
- Specify enough detail so that *someone else* could design a solution
- Keep specification up-to-date as requirements are refined

Design

- Apply responsibility-driven design
- Evaluate technical alternatives and document design decisions
- Keep it simple; add complexity only when necessary
- Anticipate changing requirements
- Identify and factor out reusable components

Prototyping

Prototyping is an essential activity carried out during *all phases* of the software process.

Requirements validation

- Prototype a user interface as early as possible to validate your requirements specification.

Evaluating design decisions

- Prototype parts of your design to evaluate feasibility and usability of technical alternatives.

Iterative development

- Integrate parts as early as possible to always have a running prototype of the target application that can be tested and demoed.

Testing

Coverage

- Design tests that will exercise all required/implemented functionality
- Check that all possible execution paths are tested
  - Apply both black-box and white-box testing

Regression

- Automate testing so that all tests can be carried out after any system change
- Set up tests so they can run in either
  - “verbose” mode (i.e., logging every interesting event), or in
  - “silent” mode (i.e., only reporting when and where tests have failed)

Responsibilities

Guidelines

- Each team member should assume a set of well-defined *responsibilities*
- but the work should be *distributed* amongst team members
- Assign responsibilities according to the skills of your team members

Sample responsibilities

- Project Administrator
- Chief programmer/architect
- Backup programmer
- Tester/test case developer
- Toolsmith
- Component librarian
- Documentation editor

Supporting roles

Client

- answer questions about requirements ☞ *email log*
- accept/reject requirements specs
- evaluate prototypes, final system

System support

- system administration
- maintain installation of required software
- (limited) help for technical problems ☞ *email log*

Consultants

- meet regularly (minimum weekly) with their teams
- oversee quality of work
- give advice concerning software process, technical solutions etc.
- crisis detection; trouble-shooting

Forming Teams

1. Identify your skills: *strong and weak points*
   - ☞ What skills would complement your own?

2. Round table: *20 seconds to present yourself*
   - ☞ What do you have to offer; who are you looking for?

3. Form teams of five: *look for suitable partners*
   - ☞ Seek complementary skills that cover responsibilities

4. Prepare your strategy and tactics:
   - ☞ What questions do you need to ask of the client?
   - ☞ What interactions do you anticipate with other teams?

Tools

Use (at least) the following tools!

UML	Use UML to document all your models (esp. requirements specification and design).
SNiFF+	SNiFF+ integrates software development tools (compilers, version management etc.) and provides support for development of software in teams.
rcs	Use version control for all text documents (i.e., both source code and documentation).
make	Use make to automate compilation, installation, testing and cleanup.
javadoc	Automate generation of HTML documentation from source code.

Many other tools are available — use them!

If you aren't sure what tool you should use to get a job done, just ask!

Component Development

Consider developing components wherever you identify a generally useful abstraction.

- Well-design components will be easier to adapt and reuse when the application evolves.
- A component may be either a piece of software (class, package) of a final application or a specialized tool to help in the development or testing of the application.

Consider marketing components to other teams:

- A team can deliver a (substantial) component set instead of a complete project if it finds *at least two* other client teams (subject to approval of the course instructors!)
- A component provider must supply (according to contract):
  - complete software with documentation
  - test drivers
  - maintenance support

2. Problem Description

ESEC Program Committee Support

- Overview of PC activities
  - Paper submission
  - Distribution of papers
  - Reviewing
  - Ranking/conflict detection
  - PC Meeting/paper selection
- Special situations
- Data Files
- Authorization
- Tailoring

Overview of ESEC PC activities

ESEC is the European Software Engineering Conference. ESEC 99 will be held in Toulouse in Sept. 1999. The Programme Committee (PC) is responsible for selecting original, scientific papers to be presented at the conference.

- Authors submit papers to the PC Chair (deadline early 99)
- The PCC distributes papers for evaluation to the PC (3 reviews per paper)
- PC members (PCMs) review papers and send review forms back to the PCC
- The PCC collects, analyzes and ranks the reviews
- The PCC distributes conflicting reviews to PCMs
- At the PC meeting papers are discussed and accepted/rejected (Spring 99)
- The PCC informs authors whether their papers are accepted or not and sends back extracts of the reviews
- Authors of accepted papers send camera-ready copy to the PCC (possibly incorporating changes requested in the reviews)
- The PCC prepares the camera-ready proceedings for publication
- Authors present their papers at the conference

ESEC PC Software Support

The PCC would like a simple system to manage papers and reviews for ESEC 99, that can be adapted easily to other conferences with slightly different review procedures.

- Paper submission (WWW registration)
- Review management
  - bidding and distribution
  - review submission (email or WWW)
  - status reports
  - ranking and conflict detection
- Correspondence with authors
  - generation of acceptance/rejection letters (from templates)
  - extraction of comments to authors from reviews

Paper submission

Most papers have multiple authors, but one (not necessarily the first) is designated as the *contact author* for correspondance with the PCC.

- Papers should be submitted electronically (strict deadline)
  - URL of PDF or postscript document
  - some authors may not have access to the internet ...
    (PCC must be able to handle exceptions)
- Authors may be required to register intent to submit a paper (by WWW form)
  - title, authors, keywords, abstract, contact information
  - deadline ~ one week before paper deadline
- Receipt of intent and paper submissions must be acknowledged by email

Distribution of papers

Each paper should be reviewed by (at least) 3 PCMs. Each PCM should get roughly the same number of papers to review:

(~ 100 papers * 3 reviews / paper) / 20 PCMs = ~ 15 reviews / PCM

- Papers should be matched to PCMs' research interests
  - use keywords, abstract, reference list, etc.
- Papers should *not* be reviewed by a PCM with a conflict of interest
  - i.e., previous research collaboration ...
- If "intent to submit" is received in advance, PCMs may *bid* for papers
  - distribute abstracts electronically (email or WWW access)
  - PCMs rank papers: e.g. *interested, indifferent, not interested, conflict*

Paper distribution cannot be fully automated, but it would be useful for an initial distribution to be automatically generated.

Reviewing

Although PCMs are responsible for their own reviews, they may use additional reviewers to evaluate papers.

- Review forms consist of several parts, commonly:
  - paper identification (#, title etc.)
  - reviewer identification (PCM/reviewer)
  - comments for author
  - additional comments for PC
  - decision (ABCD): strong accept / weak accept / weak reject / strong reject
  - expertise (XYZ): expert / knowledgeable / non-expert
- There may also be various other fields:
  - summary, theme
  - rating of presentation, originality, relevance etc.

Sample Review Form

Review forms should be simple. The precise format is arbitrary, but it should be easy to add different kinds of fields and to check for missing information.

```
# Please fill in the parts labelled “XXX”.  Do not otherwise modify this form,
# as it will be electronically processed.
TITLE:XXX
AUTHOR:XXX
REVIEWER:XXX

# Assign a numeric score from 1 to 4 using the following criteria:
# 4- Accept(I will argue for acceptance)
# 3- Weak Accept(I vote to accept, but don’t mind if it is rejected)
# 2- Weak Reject(I vote to reject, but don’t mind if it is accepted)
# 1- Reject(I will argue for rejection)
OVERALL RATING:XXX
TECHNICAL QUALITY:XXX
ORIGINALITY: XXX
RELEVANCE TO ESEC 99:XXX
PRESENTATION:XXX

# Use the rating [X=Expert/Y=Knowledgeable/Z=Not my field]
REFEREE’S EXPERTISE:XXX

# Short summary of the rationale for your recommendation (3 lines max):
XXX

# Detailed comments to authors:
XXX
```

Review submission

Reviews should be submitted to the PCC at least one week before the PC meeting so that the results can be analysed, reports generated, conflicts identified and special problems detected.

Review *must* be submitted electronically, either as plain ASCII text (email) or via WWW forms:

- Some PCMs like to fill review forms on-line (WWW)
  - ☞ It must be possible to revise reviews after they have been entered
- Other PCMs prefer off-line reviewing (so they can work at home, on the train, etc., or to exchange reviews with extra reviewers)
  - ☞ must be able to handle *both* email and WWW forms
  - ☞ email forms must use a simple format that can be easily parsed and checked for consistency (missing or duplicate fields etc.)

Ranking/conflict detection

Papers that will be accepted for presentation at the conference *cannot* be automatically selected on the basis of the review forms because of the different subjective impressions of the reviewers.

It is useful, however, to rank papers into groups with high/low chances of acceptance

Papers can be ranked according to various schemes:

- Weighted average of scores
  - generally not very useful since AAD gets same rank as BBC
- Group by best/worst score (AA, AB, AC, etc.)
  - AAD (AD) is different from BBC (BC)
- Identify papers where there is strong disagreement (“conflict”)
  - i.e., high/low = AD, BD, AC
- It is useful to have lists of papers sorted also by paper number, reviewer etc.

PCMs should be notified of disagreements in advance!

PC Meeting/paper selection

Although some smaller conferences and workshops select accepted papers by email discussion, or at the discretion of the PCC, ESEC, ECOOP and other larger conferences require the PC to meet during 1-2 days to arrive at a consensus which papers to accept.

Discussions roughly follow these rules:

- Papers are discussed if there is someone in favour of acceptance (a “champion”)
- Papers with uniformly high scores (AA, AB) are usually accepted
- If there is no champion for a paper it will be rejected
- Papers with low rankings (CC, CD, DD) are rejected with little or no discussion unless someone wants to change their score
- Borderline papers (BB, BC) and conflicts (AD, AC) may need to be discussed in depth
- The PCC tries to get the PC to arrive at a consensus on all papers
  - ☞ (in case of unresolvable conflicts, the PC may have to vote on a paper)

Acceptance/Rejection

Authors are informed by email (or post) of acceptance/rejection after the meeting

- Authors receive an extract of the reviews of their papers
  - ☞ i.e., comments for the authors
- In some cases, the PCC may have additional instructions
  - ☞ i.e., obligatory changes for conditional acceptance

Special situations

Various problematic situations always arise that need to be dealt with in an exceptional way. The most common problems are:

- Conflicts of interest:
  - The concerned PCMs should leave the room when the paper is discussed
  - PC authored papers should be discussed as group
- Underrepresented papers (may need extra reviews)
  - missing reviews
  - incomplete reviews
  - no expert reviewers
  - absent PCMs (may need email/phone consultation)

Special situations should be identified and handled as early as possible (i.e., before the PC Meeting).

Data Files

The PCC needs to manage (at least) the following information:

- Authors (contact information)
- Papers (title, authors, contact author, abstract, keywords, URL)
- PCMs (contact information, interests)
- Reviewers (contact information)
- Reviews (filled out forms)
- Various reports (ranked reviews; completed reviews by PCM; ...)

Authorization

The entire review process is sensitive. Although all information will be accessible through the WWW, only authorized PCMs will be able to access papers, reviews and other information concerning the review process.

- The PCC has access to *all* information
- PCMs will be assigned passwords for WWW access
- All PCMs may see all abstracts and papers
- Non-PCMs may access nothing
- Reviews are accessible to PCMs when all reviews for a paper are in
- A PCM may not see a review if he or she has a conflict of interest

Tailoring

The PCC should be able to tailor the system once it is in place:

- add or change fields of the review form
- add or change PCMs
- add new kinds of report generation
- easily generate ad hoc statistics (e.g., submission of papers by country)

3. An Introduction to Java

Overview

- Java vs. C++
- Java language features: packages, classes, exceptions ...
- The Java API
- Applets

Texts:

- David Flanagan, *Java in a Nutshell*, O'Reilly, 1996
- Mary Campione and Kathy Walrath, *The Java Tutorial* , The Java Series, Addison-Wesley, 1996

On-line resources:

- Locally installed Java resources (on-line tutorial, language spec, etc):

  `http://www.iam.unibe.ch/~scg/Resources/Java/`

Java

Language design influenced by existing OO languages (C++, Smalltalk ...):

- Strongly-typed, concurrent, pure object-oriented language
- Syntax, type model influenced by C++
- Single-inheritance but multiple subtyping
- Garbage collection

Innovation in support for network applications:

- Standard API for language features, basic GUI, IO, concurrency, network
- Compiled to bytecode; interpreted by portable abstract machine
- Support for native methods
- Classes can be dynamically loaded over network
- Security model protects clients from malicious objects

Java applications do not have to be installed and maintained by users

Java and C++ — Similarities and Extensions

Java resembles C++ only superficially:

Similarities:

- primitive data types (in Java, platform independent)
- syntax: control structures, exceptions ...
- classes, visibility declarations (`public`, `private`)
- multiple constructors, `this`, `new`
- types, type casting

Extensions:

- garbage collection
- standard classes (Strings, collections ...)
- packages
- standard abstract machine
- final classes

Java and C++ — Simplifications

Whereas C++ is a hybrid language, Java is a pure object-oriented language that eliminates many of the complex features of C++:

Simplifications:

- no pointers — just references
- no functions — can declare `static` methods
- no global variables — can declare `public static` variables
- no destructors — garbage collection and `finalize` methods
- no linking — dynamic class loading
- no header files — can define `interface`
- no operator overloading — only method overloading
- no member initialization lists — `super` constructor can be called
- no preprocessor — `static final` constants and automatic inlining
- no multiple inheritance — can implement multiple interfaces
- no structs, unions, enums — typically not needed
- no templates — but generics will likely be added ...

The "Hello World" Program

`helloWorld` objects can be instantiated by any client

only classes can be declared (pure OO)

class methods behave like global functions

Every program must have a `main` method declared in some class

`String` is a standard class

```
// My first Java program!
public class helloWorld {
   public static void main (String argv[]) {
      System.out.println("Hello World");
   }
}
```

a class in the package java.lang

a public class variable

a public method

Packages

A Java program is a collection of classes organized into *packages*

- At least one class must have a `public static void main()` method
- The first statement of a source file may declare the package name:

```
package games.tetris;
```

- Source files (e.g., helloWorld.java) are compiled to bytecode files (e.g., helloWorld.class), one for each target class
- Class files must be stored in subdirectories corresponding to the package hierarchy
- When using classes, either the full package name must be given:

```
java.lang.System.out.println(“Hello World”);
```

  or classes from the package may be *imported:*

```
import java.lang.*; // this package is always imported by default
```

- Class names are usually capitalized for readability:

```
a.b.c.d.e.f(); // which is the name of the class?!
```

Java Basics

Java's primitive data types and control statements resemble those of C/C++:

Primitive Data Types:

```
boolean byte char double float int long short void
```

Literals:

```
false null true
```

Control flow:

```
if ( boolean ) { Statements } else { Statements }
for ( boolean ) { Statements }
while ( boolean ) { Statements }
do { Statements } while ( boolean )
switch ( variable ) {
   case label : Statements;
      break; ...
   default : ... break;
}
```

Classes and Objects

The encapsulation boundary is a class (not an object):

```
public class Point {
   private double x, y;   // not accessible to other classes (even subclasses)

   // constructors:
   public Point (double xCoord, double yCoord) { x = xCoord; y = yCoord; }
   public Point (Point p) { x = p.x; y = p.y; } // can access private data here

   // public methods:
   public double getX ( )             { return x; }
   public void   setX (double xCoord){ x = xCoord; }
   public double getY ( )             { return y; }
   public void   setY (double yCoord){ y = yCoord; }
   public double distance ( )         { return Math.sqrt(x*x + y*y); }
}
```

In pure OOLs, (non-primitive) objects are passed by reference, not by value:

```
int a = 3, b = 4;          // a and b are primitive objects
Point p1 = new Point(a,b);// p1 is a reference to an object (NB: a & b coerced!)

int c = a;                 // c gets value of a
c = 8;                     // c gets new value; a is unchanged

Point p2 = p1;             // p2 refers to p1
Point p3 = new Point(p1); // p3 is a copy of p1
p2.setX(c);                // The object p1 and p2 refer to is modified
```

Garbage Collection

In Java (as in Smalltalk and Eiffel), objects no longer referred to are automatically garbage-collected:

- no need to explicitly `delete` objects
- no destructors need to be defined
- no need to write reference-counting code
- no danger of accidentally deleting objects that are still in use

You can still exercise extra control:

- Cleanup activities can be specified in a `finalize` method
  - useful for freeing external resources (files, sockets etc.)
- Objects you no longer need can be explicitly “forgotten”
  - you can explicitly forget objects by assigning the value `null` to a variable (this is the initial value of declared, but unassigned variables)

Inheritance

A subclass *extends* a superclass, inheriting all its features, and possibly overriding some or adding its own:

```
public class Circle extends Point {
   private double r;

   public Circle (double xCoord, double yCoord, double radius) {
      super(xCoord, yCoord);    // call Point constructor
      r = radius;
   }

   public Circle (Circle c) {
      super(c);                 // call Point constructor with c as Point
      r = c.r;
   }

   public double getR ( )             { return r; }
   public void   setR (double radius){ r = radius; }
   public double distance ( )         { return super.distance() - r; }
}
```

Public superclass features can always be accessed, even if overridden.

Dynamic Binding

One of the key features of object-oriented programming is *dynamic binding* — the actual method that will be executed in response to a request depends on the dynamic type of target, not the static type of the reference:

```
Point p = new Circle(5, 12, 4);
System.out.println("p.distance() = " + p.distance());
```

yields:

```
p.distance() = 9
```

In pure OOLs, all methods are dynamically bound by default.

Static binding is the exception:

- `static` methods belong to classes, so are statically bound
- `private` methods have purely local scope
- `final` methods cannot be overridden, so are statically bound

Downcasting

Dynamic binding can cause type information to be lost:

```
Point p = new Circle(5, 12, 4);    // p refers to a Circle – upcast ok
Circle c1 = p;                     // compile-time error! – can't downcast
```

Type information can be recovered at run-time by explicit tests and casts:

```
if (p instanceof Circle) {         // run-time test
   c1 = (Circle) p;                // explicit run-time downcast ok
}
```

An attempt to cast to an invalid type will raise an exception at run-time:

```
p = new Point(3,4);
c1 = (Circle) p;                   // invalid downcast raises run-time exception
```

Feature Visibility

Features can be declared with different degrees of visibility:

- `private` — accessible only within the class body
- `public` — accessible everywhere
- `protected` — accessible to subclasses *and* to members of the same package
  - ☞ allows access to cooperating classes
- default (no modifier) — accessible throughout the package only
  - ☞ allows package access but prevents all external access

Modifiers

In addition to feature visibility, modifiers can specify several other important attributes of classes, methods and variables:

- `abstract` — unimplemented method; class must also be declared abstract
  - ☞ method signature is followed by semi-colon instead of body
- `final` — class/method/variable cannot be overridden by subclass
- `static` — method/variable belongs to class, not instances; implicitly final
- `native` — method implemented in some other language, usually C

Exceptions

A class must declare which exceptions it `throws`, or it must `catch` them:

```
public class TryException {
   public static void main(String args[]) {
      try {
         alwaysThrow(0);          // NB: we never get past this point
         alwaysThrow("hello");
      } catch (NumException e) {
         System.out.println("Got NumException: " + e.getMessage());
      } catch (StringException e) {
         System.out.println("Got StringException: " + e.getMessage());
      } finally {
         System.out.println("Cleaning up");
      }
   }
   public static void alwaysThrow(int arg) throws NumException {
      throw new NumException("don't call me with an int arg!");
   }
   public static void alwaysThrow(String arg) throws StringException {
      throw new StringException("don't call me with a String arg!");
   }
}
```

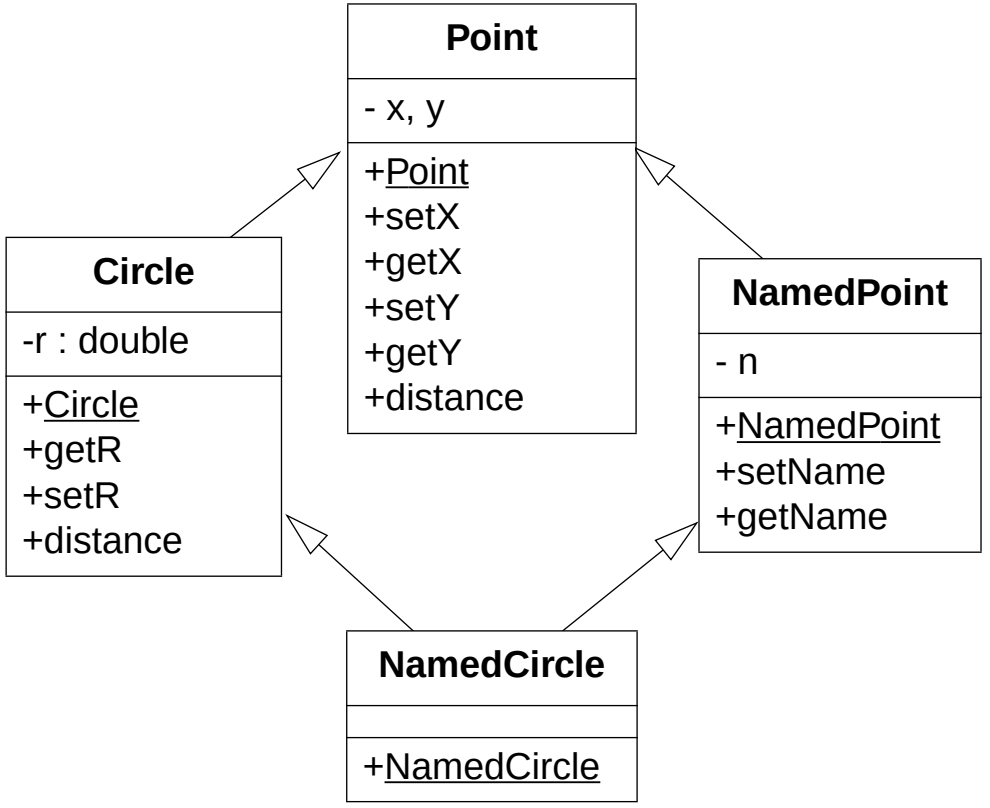
Defining Exceptions

You can define your own exception classes that inherit from Exception
Typically, you will only define constructors:

```
// Most exception classes look like this:
public class NumException extends Exception {
   public NumException() { super(); }
   public NumException(String s) { super(s); }
}
public class StringException extends Exception {
   public StringException() { super(); }
   public StringException(String s) { super(s); }
}
```

Multiple Inheritance

Although conceptually elegant, multiple inheritance poses significant pragmatic problems for language designers:



Which version of distance() should be inherited by NamedCircle?

Interfaces

An interface declares methods but provides no implementation:

```
interface Named {
   public void setName (String name);
   public String getName ( );
}
```

A Java class can extend at most one superclass, but may implement multiple interfaces:

```
public class NamedCircle extends Circle implements Named {
   private NamedObject n; // object composition vs. inheritance
   public NamedCircle (double xCoord, double yCoord, double radius, String name) {
      super(xCoord, yCoord, radius); // call Circle constructor
      n = new NamedObject(name);     // compose a NamedObject instance
   }
   public void setName (String name) { n.setName(name); } // forwarding
   public String getName ( ) { return n.getName(); }
}
```

Reusable behaviour can be encapsulated as a separate class:

```
public class NamedObject implements Named {
   private String n;
   public NamedObject (String name) { n = name; }
   public void setName (String name) { n = name; }
   public String getName ( ) { return n; }
}
```

Overriding and Overloading

Overridden methods have the same name and argument types

Overloaded methods have the same name but different argument types

```
public class A {
   public void f (float x)   { System.out.println("A.f(float)"); }
   public void g (float x)   { System.out.println("A.g(float)"); }
}

public class B extends A {
   public void f (float x)   { System.out.println("B.f(float)"); }
   public void g (int x)     { System.out.println("B.g(int)"); }
}
```

Overloaded methods are disambiguated by their arguments:

```
B b = new B();    // both dynamic and static type B
A a = b;          // static type is A but dynamic type is B

b.f(3.14f);       // B.f(float) -- overridden
b.f(3);           // B.f(float) -- 3 is converted to 3.0
b.g(3.14f);       // A.g(float) -- not overridden
b.g(3);           // B.g(int) -- overloaded

a.f(3.14f);       // B.f(float) -- overridden
a.f(3);           // B.f(float) -- 3 is converted to 3.0
a.g(3.14f);       // A.g(float) -- not overridden
a.g(3);           // A.g(float) -- g(int) does not exist in SuperClass!
```

Arrays

Arrays are polymorphic objects:

- Can declare arrays of any type

```
int[] array1;
MyObject s[];
```

- Can build array of arrays

```
int a[][] = new int[10][3];
a.length --> 10
a[0].length --> 3
```

Creating arrays

- An empty array:

```
int list[] = new int [50];
```

- Pre-initialized:

```
String names[] = { “Marc”, “Tom”, “Pete” };
```

- Cannot create static compile time arrays

```
int nogood[20];  // compile time error
```

Arrays and Generics

Arrays are the only polymorphic containers in Java:

```
Point [] pa = new Point[3];
pa[0] = new Point(3,4);
pa[1] = new Point(5,12);
Point p = pa[0];              // ok -- pa is an array of Points
```

It is not possible to program other kinds of polymorphic containers:

```
Stack s = new Stack();        // defined in package java.util
s.push(pa[0]);
s.push(pa[1]);
// p = s.pop();               // compile-time error -- s.pop() returns an Object
p = (Point) s.pop();          // ok -- run-time cast
```

The Java API

java.lang. contains essential Java classes, including numerics, strings, objects, compiler, runtime, security, and threads. This is the only package that is automatically imported into every Java program.

java.awt. Abstract Windowing Toolkit

java.applet. enables the creation of applets through the Applet class.

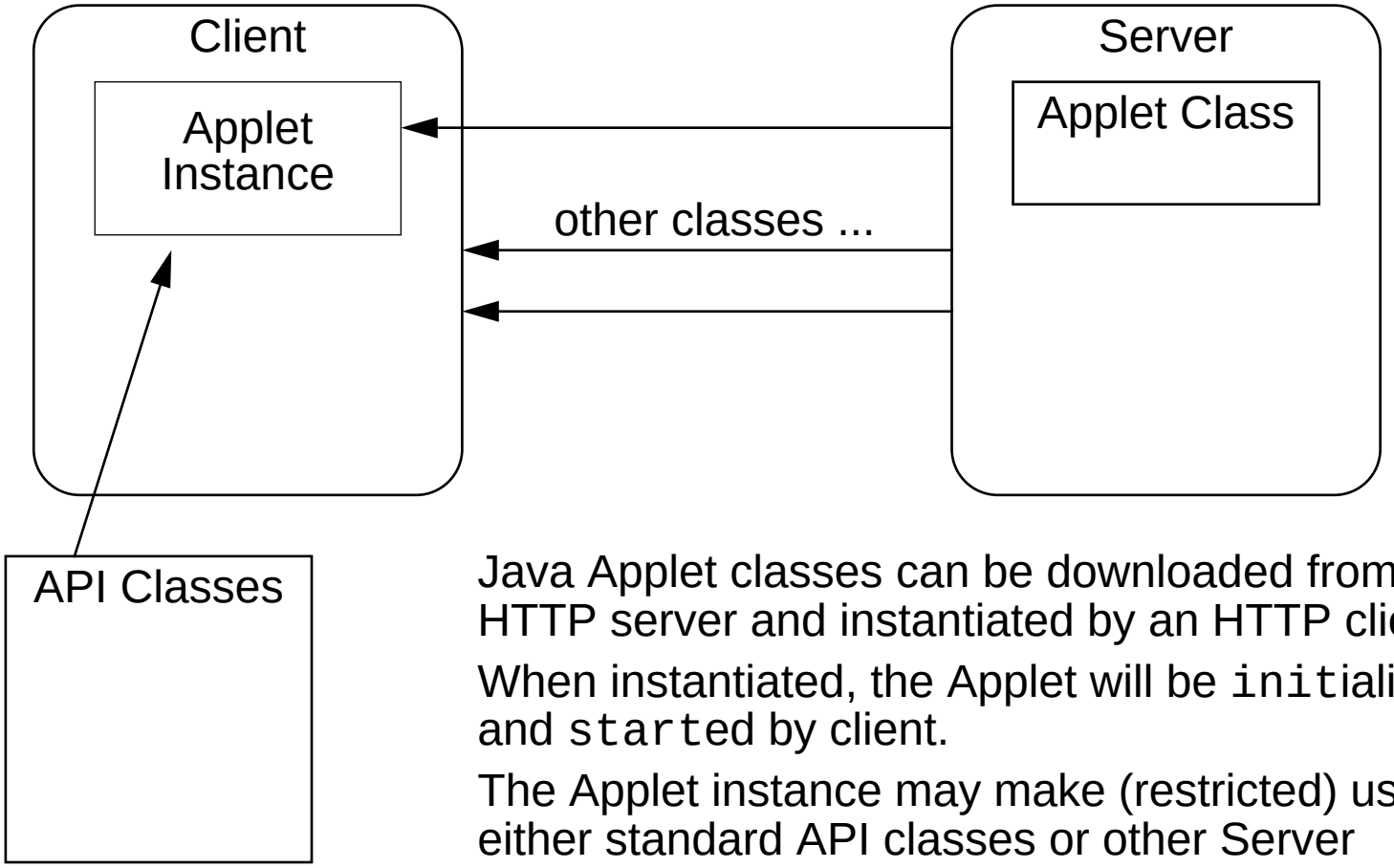
java.io. provides classes to manage input and output streams to read data from and write data to files, strings, and other sources.

java.util. contains miscellaneous utility classes, including generic data structures, bit sets, time, date, string manipulation, etc.

java.net. provides network support, including URLs, TCP sockets, UDP sockets, IP addresses, and a binary-to-text converter.

And many others ...

Applets



Java Applet classes can be downloaded from an HTTP server and instantiated by an HTTP client.

When instantiated, the Applet will be `initialized` and `started` by client.

The Applet instance may make (restricted) use of either standard API classes or other Server classes to be downloaded dynamically.

NB: objects are *not* downloaded, only classes!

The Hello World Applet

The simplest Applet:

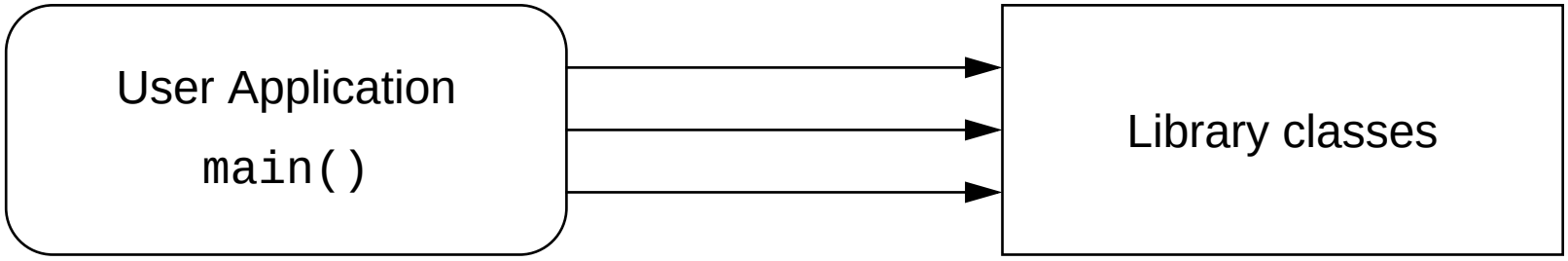
```
// From Java in a Nutshell, by David Flanagan.

import java.applet.*;  // To extended Applet
import java.awt.*;     // Abstract windowing toolkit

public class HelloApplet extends Applet {
   // This method displays the applet.
   // The Graphics class is how you do all drawing in Java.
   public void paint(Graphics g) {
      g.drawString("Hello World", 25, 50);
   }
}                      // NB: there is no main() method!
```

HTML applet inclusion:

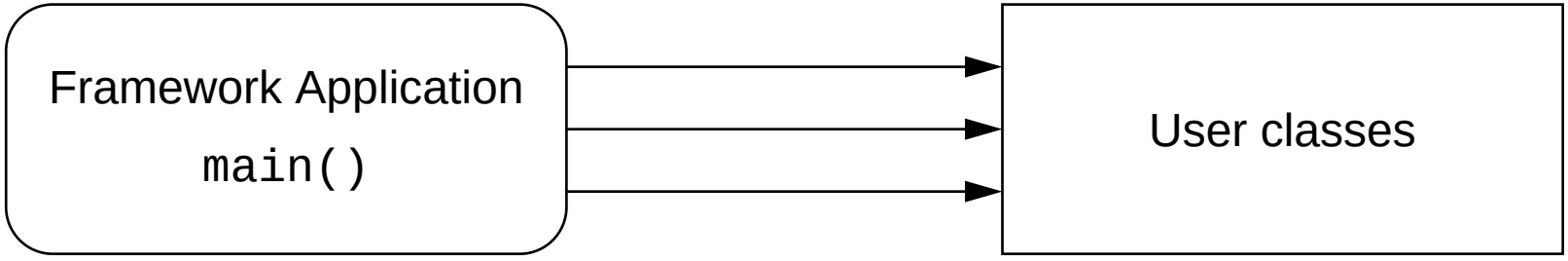
```
<title>Hello Applet</title>
<hr>
<applet codebase="HelloApplet.out" code="HelloApplet.class" width=200 height=200>
</applet>
<hr>
<a href="HelloApplet.java">The source.</a>
```

Frameworks vs. Libraries

In traditional application architectures, user applications make use of library functionality in the form of procedures or classes:



A framework reverses the usual relationship between generic and application code. Frameworks provide *both* generic functionality *and* application architecture:



Essentially, a framework says: “Don’t call me — I’ll call you.”

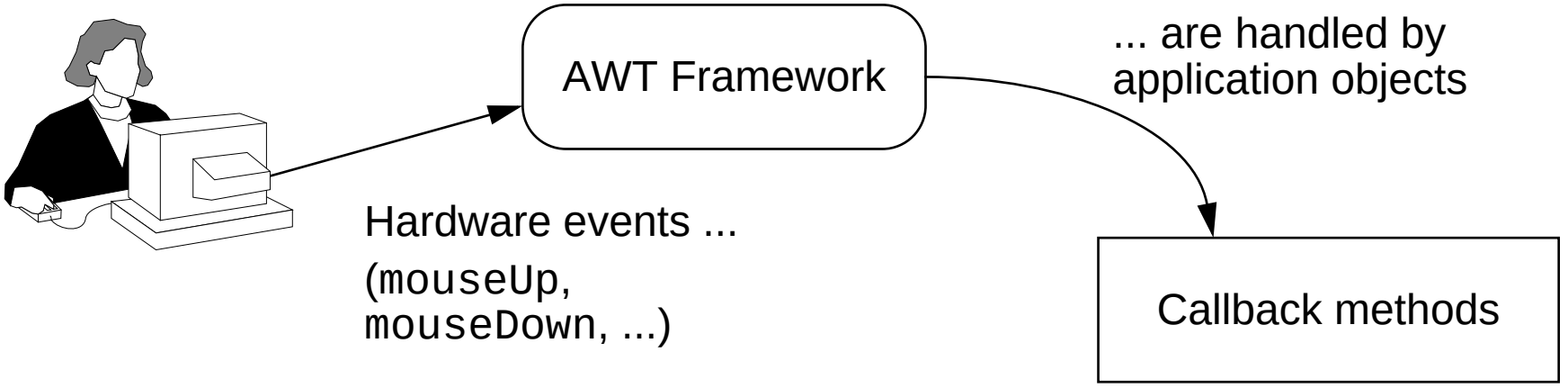
Standalone Applets

An Applet is just a user object instantiated by the Applet framework:

```
// Adapted from Java in a Nutshell, by David Flanagan.
// A simple example of directly instantiating an Applet.

import java.applet.*;
import java.awt.*;

public class HelloStandalone {
   public static void main(String args[]) {
      Applet applet = new HelloApplet();
      Frame frame = new AppletFrame("Hello Applet", applet, 300, 300);
   }
}

class AppletFrame extends Frame {
   public AppletFrame(String title, Applet applet, int width, int height) {
      super(title);                 // Create the Frame with the specified title.

      this.add("Center", applet);   // Add the applet to the window.
      this.resize(width, height);   // Set the window size.
      this.show();                  // Pop it up.

      applet.init();                // Initialize and start the applet.
      applet.start();
   }
}
```

Events

Instead of actively checking for GUI events, you can define callback methods that will be invoked when your GUI objects receive events:



`Component` is the superclass of all GUI components (including `Frame` and `Applet`) and defines all the callback methods that components must implement.

The Scribble Applet

`Scribble` is a simple Applet that supports drawing by dragging the mouse:
NB: This example uses the (deprecated) Java 1.0 event model!

```
// Adapted from Java in a Nutshell, by David Flanagan.

import java.applet.*;
import java.awt.*;

public class Scribble extends Applet {
   private int last_x = 0;
   private int last_y = 0;
   private Button clear_button;

   // Called to initialize the applet.
   public void init() {
      this.setBackground(Color.white);          // Set the background colour
      clear_button = new Button("Clear");       // Create a Button
      clear_button.setForeground(Color.black);
      clear_button.setBackground(Color.lightGray);
      this.add(clear_button);                   // Add it to the Applet
   }
```

Responding to Events

```
   // Called when the user clicks the mouse to start a scribble
   public boolean mouseDown(Event e, int x, int y) {
      last_x = x; last_y = y; return true; // Always return true if event handled
   }
   // Called when the user scribbles with the mouse button down
   public boolean mouseDrag(Event e, int x, int y) {
      Graphics g = this.getGraphics();
      g.setColor(Color.black); g.drawLine(last_x, last_y, x, y);
      last_x = x; last_y = y; return true;
   }
   // Called when the user clicks the button
   public boolean action(Event event, Object arg) {
      // If the Clear button was clicked on, handle it.
      if (event.target == clear_button) {
         Graphics g = this.getGraphics();
         Rectangle r = this.bounds();
         g.setColor(this.getBackground());
         g.fillRect(r.x, r.y, r.width, r.height);
         return true;
      } // Otherwise, let the superclass handle it.
      else return super.action(event, arg);
   }
}
```

Running the Scribble Applet