SOEN 341   Software Process

Lecture Notes

Peter Grogono

January 2003

Department of Computer Science
Concordia University
Montreal, Quebec

Contents

1 About the Course

1.1 General Information

1.1.1 Instructor

Name:	Peter Grogono
Office:	LB 927–17
Telephone:	848 3012
Email:	grogono@cs.concordia.ca
Lectures:	Wednesday and Friday, 1015–1130, H 535–2
Tutorial:	Friday, 1315–1405, H 403
Office hours:	Wednesday, 1145–1245, LB 927–17
Web page (course):	`http://www.cs.concordia.ca/~teaching/soen341`
Web page (Sec S):	`http://www.cs.concordia/~faculty/grogono/SOEN341`

Note that there are ***two*** web pages for the course: the first (in the table above) is for all three sections of the course and the second is for Section S only.

1.1.2 Background

The purpose of SOEN 341 is to introduce the basic concepts of industrial software development and to prepare you for subsequent software engineering courses such as: SOEN 342 *Software Requirements and Specifications*; SOEN 343 *Software Design*; SOEN 345 *Software Quality Control*; and SOEN 383 *Software Project Management*.

You should have had some previous experience in programming, preferably object oriented programming using C++ or Java (COMP 249 or equivalent), some knowledge of data structures and algorithms (COMP 352 or equivalent), and some knowledge of the principles of technical documentation (SOEN 282 or equivalent).

1.1.3 Text Book

The recommended text for the course is:

- *Software Engineering Process with the UPEDU*. Pierre Robillard and Philippe Kruchten (with Patrick d'Astous). Addison Wesley (Pearson Education), 2003. ISBN 0–201–75454–1. `http://www.yoopeedoo.org/index.asp`.

1.1.4 Course Outline

The discipline of engineering. Introduction to Software Engineering. Software engineering *versus* "programming". Duties and responsibilities of a software engineer. Working as a member of a team; leading a team. Software life cycles and process models. RUP and UPEDU. From process to project; project management. Phases of the software process:

requirements, specification, design, implementation, testing, delivery, and maintenance. Case studies of software engineering practice.

1.1.5 Computing Facilities

You should use DCS Computing Laboratory H–929 which is equipped with PCs running Windows.

1.1.6 Project

The major practical component of the course will be an object oriented program designed and implemented by a team. Each team will consist of five or six students. We will discuss the roles of team members during classes. Each team will implement an application chosen from a list that will be provided by the instructor.

1.1.7 Evaluation

- You will be evaluated on the basis of quizzes (30%), assignments (20%), and a programming project (50%).
- You will do the quizzes and assignments individually and the project as a member of a team.
- There will be four or five assignments. The assignments will be primarily theoretical, since the project provides the practical component. Late assignments will be discounted at 20% per day.
- There will be quizzes during the fourth, eighth, and twelfth weeks of the course. (Probable dates: Friday, 30 January; Friday, 6 March; and Friday, 4 April.)

1.2 The Course Project

The programming project contributes 50% of your marks for the course (see above).

Each project will be completed by a team of about 5 students. Instructors will divide students into teams; the teams may make minor adjustments to their membership.

1.2.1 Schedule

Friday	10	January	Instructor collects names
Wednesday	15	January	Instructor proposes teams
Wednesday	22	January	Teams submit names of team leader and team members

1.3 Project Deliverables

Each team must submit various deliverables as the project proceeds.

Proposal. The proposal is a brief (one or two page) description of the project that the team has decided to implement. Instructors will read the proposals carefully in order to decide whether they are appropriate for the course. They may suggest changes (either simplifications or extensions) of the proposed project or they may request a new proposal for a more appropriate project. Since it is best to avoid writing two proposals, you should consult the instructor before the due date if you have any doubts about the acceptability of your project.

Design. The design document consists of three components:

1. a general description of the software;
2. a high-level design showing the methods provided by each class and the interactions between classes; and
3. a detailed design showing the attributes of each class and describing how each of its methods will be implemented.

Instructors will review the designs and give their opinions to the proposing team. The purpose of the design review is mainly to avoid a situation in which a team gets stuck trying to implement an infeasible design.

Demonstration. Teams will demonstrate the programs they have written to the instructor. The purpose of the demonstration is to give the instructor an opportunity to see the program running, to try it out, and to discuss the project informally with the team. All team members must attend the demonstration.

Final Submission. To complete the project, each team must submit the following documents:

1. A ***User Manual*** explaining how to operate the program from a user's perspective.
2. ***Design Documentation***, as described above, but possibly modified in the light of experience.
3. ***Implementation Notes*** describing particular features of the implementation, problems encountered and how they were solved, etc.
4. Complete source code.

In addition, you must submit an ***individual*** report of your own experience in working on the project: what you contributed, what was easy, what was hard, whether you had disagreements, etc.

For items 1 through 4 above, all members of a team will receive the same mark — up to 75% of the total marks for the project. Instructors will use the individual report to determine each team members's contribution to the project — up to 25% of the total marks for the project.

1.3.1 Choosing a Project

You have to design and implement a software application. The preferred programming languages are C++ and Java.

The program can be quite ambitious (it is worth 5 students × 3 credits × 50% = 7.5 credits). The objective of the project, however, is a ***high quality software product***: software that is user-friendly, robust, and well-documented. ***Credit will be given for quality rather than quantity.***

1.4 Working in Teams

Why teams?

- Teams mean fewer projects and less work for me.
- Industrial programming is always done in teams.
- Employers want collaborators not stars.
- You can complete a larger project by working with others.

1.4.1 Roles of Team Members

One member of the team will be chosen by the team as its leader. The team leader acts as the communication channel between the instructor and the team. Apart from this, the responsibilities of the team leader are the same as for other members of the team.

Each team decides how to allocate its resources. One possibility would be for all of the members to collaborate in all of the tasks. At the other extreme, the team could allocate one person for design, one person for implementation, etc. In practice, something in between these extremes is probably best. For example, two team members could take responsibility for the design, but they would also consult other team members during the design process. Keep in mind the project deliverables when assigning roles to team members.

It is the team's responsibility to decide who should do what. There are two requirements:

- There must be a ***team leader***. The team leader's job is to interface between the team and the instructor. Apart from this special role, the team leader performs the same work as other members of the team.
- The final submission must include an individual statement explaining what your personal role in the team was.

Individual team members should have particular responsibilities. For example, one member could be responsible for the design. Two or three members, or even the whole team, might work together on he design, but this person has the responsibility of ensuring that the design is completed by the deadline. The same goes for other aspects of the project:

- The ***proposal*** is a brief description of the project that the team intends to work on.
- The ***design*** is a detailed description of how the software is organized and how it will work.

- ***Implementing*** is the process of writing the code from the design.
- ***Testing*** is the process of checking that the code meets its specifications. (There are no detailed specifications for this project, but it is still possible to test the program against the proposal and the design.)
- The project must be well ***documented***.

Disputes. Arguments within teams are common. If possible, the team should resolve disagreements by itself. If this doesn't work, the next step is to consult the tutor. If the tutor cannot solve the problem, the team leader should inform the instructor, who will probably arrange to meet with the whole team.

Credit. Another potential problem with team work is that the instructor must give equal credit to all team members. Team members may perceive this as unfair if some did more work than others. Problems of this kind should be solved by the team itself as far as possible. Individuals who feel that were treated unfairly should say so in their individual reports.

1.5 Introduction to Software Development

A ***process*** is a collection of organized activities that lead to a ***product***.

In particular, a ***software process*** is a collection of organized activities that lead to a ***software product***.

Software development is traditionally divided into various components, which we describe briefly here. For small projects, the components are developed in the order shown; for larger projects, the components are interleaved or applied repetitively with gradually increasing refinement. Large products are usually delivered as a sequence of increments rather than in one big lump.

1.5.1 Traditional Components

Requirements: The purpose of requirements analysis is to find out what the program is expected to do. This is done in the ***application domain***. The result of requirements analysis might be that we don't need a program at all.

Specification: A specification is a precise description of what the program is supposed to do (but ***not*** how it works). The specification bridges the application domain and the ***implementation domain*** because it describes the external behaviour of the computer system. The specification can also be viewed as a ***contract*** between the client who wants the program and the software company that is supplying it.

Design: The design is a precise description of the internal structure of the program. The design of an object oriented program is usually a description of the classes and their interactions.

Design is the most important, but also most difficult, phase of software development. If the design is good, implementation should be straightforward. If the design is bad, the project may fail altogether.

Implementation: Implementation consists of writing the code: it is the activity that is often called “programming”.

Testing: The code is tested thoroughly to ensure that the finished program meets its specification.

Maintenance: The program is shipped to the client(s) but the supplier continues to correct and upgrade it.

It is easy to underestimate the cost and importance of maintenance. On average, for large software projects, maintenance accounts for 80% of the total cost of the project.

1.6 Clients and Suppliers

Whenever software is produced, there is a ***client*** and a ***supplier***. If the program is very simple, the client and the supplier may be the same person. At the next level, an individual might write a program for several people. At the largest scale, a company constructs software for another company.

Whatever the size of the project, we will always use “client” to mean an entity that wants something and “supplier” to mean an entity that is prepared to create it.

2 The Waterfall Model

The first software process to be formally proposed was the ***Waterfall Model***. The waterfall model as usually presented includes the following phases, to be executed and completed in the following order:

Requirements Analysis
System Design
Program Design
Coding
Unit and Integration Testing
System Testing
Acceptance Testing
Operation and Maintenance

The name "waterfall model" suggests that there is no going back: once a phase is completed, the documentation for that phase must not be altered.

An early modification to the original waterfall model was to allow feedback between consecutive steps. For example, during coding, you could modify the program design but not the system design.

The waterfall ***as described here*** is not a good model: we will discuss its weaknesses later.

The following quotation is from a web discussion of the waterfall and other software process models (ff"http://c2.com/cgi/wiki?WaterFall):

> The original waterfall model, as presented by Winston Royce in *Managing Development of Large Scale Software Systems* (Proceeding of IEEE WESCON, August 1970) actually made a lot of sense, and does not resemble at all what is portrayed as a Waterfall in the preceeding entries. What Royce said was that there are two essential steps to develop a program: analysis and coding; but precisely because you need to manage all the intellectual freedom associated with software development, you must introduce several other "overhead" steps, which for a large project are:
>
> - System requirements
> - Software requirements
> - Analysis
>   - Program design
>   - Coding
>   - Testing
>   - Operations
>
> Many of Royce's ideas would be considered obsolete today, but I don't think the problem lies so much with his model as with the cheap (not in terms of money, of course) and rigoristic ways of misunderstanding what became an engineering and

project management "standard". This "standard" is what became "An Acceptable Way Of Failing".

Royce's valuable ideas must be understood before we simplistically discard the Waterfall. For example, when he advised to do some things twice he was postulating what we can call an embryonic form of iterative development. Also, some features of the Waterfall are inevitable: as Alistair Cockburn has said elsewhere (`http://members.aol.com/acockburn/papers/vwstage.htm`) "how can you get to do testing if you have not coded?".

The main problem has not been the Waterfall, but the schematicism involved in a "standard box for all projects". The variety of possible routes to follow is enormous: under many circumstances, training and empowering users to do their own computing will be a radical alternative not only to analysis and design, but to coding as well (after all, as Emiliano Zapata would have said, applications should belong to those who make a living out of them). Under other circumstances, the best way to enhance team collaboration is to design the database through a diagram that makes the objects visible to all participants. — Jaime Gonzalez

2.1 The Waterfall Model as a Process

The waterfall model describes a ***process*** for software development.

- The Waterfall Model is ***document driven***. Each step of the process yields ***documents***.

  For example, when Requirements Analysis has been completed, there is a Requirements Document. Before coding starts, there must be a set of Design Documents.

- Documents produced during one step are needed for the next step and possibly for later steps.

  For example, the Requirements Document is needed for design, the next step. Later, the Requirements Document is needed to ensure that the developed product meets the requirements during Acceptance Testing.

- Managers like the waterfall model because progress is observable and measurable. The transitions between steps become project "milestones" that indicate progress made. Documents are tangible evidence of progress.

  (The belief that we can monitor progress in this simple-minded way is an illusion. This is one of the weaknesses of the waterfall model.)

- We can assign people with different skills to the various phases. The waterfall model suggests roles such as:

  – analyst (also called "systems analyst");
  – designer;
  – programmer (now called "development engineer");
  – tester (now called "test engineer";
  – trainer;
  – maintainer (now called "maintenance engineer").

- We can estimate cost by adding the estimated costs of each phase and then adding a safety factor.

  The problem is that we do not have enough information during the early phases to make accurate predictions about the effort needed, and hence the cost, of later phases.

  Many software projects fail because of cost overruns. The suppliers estimate $3 million and, when they have spent $6 million, the clients cancel the contract and expensive law suits follow.

2.2 Prescription and Description

There are two kinds of models.

- A ***prescriptive model*** describes what ought to be done (just as you get a "prescription" for medicine that you ought to take).
- A ***descriptive model*** describes what actually happens (as when the medicine makes you feel even worse).

The waterfall model was introduced as a descriptive model: it was intended to describe what happens in engineering projects and to use these observations for future projects.

The waterfall model became a prescriptive model: it proposes a method for software development that is hard to follow exactly and has several problems when it is followed exactly.

2.3 Why is the waterfall a poor model?

The waterfall model is based on engineering practice but it is not clear that experience gained from construction of physical objects is going to work for software development.

The waterfall model is driven by documents. Documents may reflect what readers want to hear rather than what is actually happening. Slick documentation may cover up a project that is in turmoil.

Figure 1 compares and engineering activities, such as building a bridge, to constructing a complex software product. The waterfall model works for bridges because bridge-building is well-understood (although there are failures, usually when new designs are tried). The reason that it does not work for programming should be evident from the lists above: the software development process is not well-understood; scientific foundations are lacking; and software requirements change.

Although the waterfall model in its original form does not work, the ***phases*** given above are nevertheless ***essential for any large software project***. However, they should not be applied one at a time in a particular order.

In the next section, we outline other possible approaches briefly. Then we illustrate the phases in the context of a small application.

2.4 Other Approaches

Problems with the waterfall model have been recognized for a long time.

Building a bridge	Writing a program
the problem is well understood	some problems are understood, others are not
sound scientific principles underly bridge design	the scientific principles underlying software construction are still being developed
the strength and stability of a bridge can be calculated with reasonable precision	it is not feasible to calculate the correctness of a complex program with methods that are currently available
the requirements for a bridge typically do not change much during building	requirements typically change during all phases of development
there are many existing bridges	although there are many existing programs, there are application areas in which only a few programs have been written
there are not all that many kinds of bridge	there are very many different kinds of application
when a bridge collapses, there is a detailed investigation and report	when a program fails, the reasons are often unavailable or even deliberately concealed
engineers have been building bridges for thousands of years	programmers have been writing programs for 50 years or so
materials (wood, stone, iron, steel) and techniques (making joints in wood, carving stone, casting iron, extruding steel) change slowly	hardware and software changes rapidly

Figure 1: A comparison of bridge building and programming

The emphasis in the waterfall model is on documents and ***writing***. Fred Brooks (***The Mythical Man Month***, Anniversary Edition, 1995, page 200):

> "I still remember the jolt I felt in 1958 when I first heard a friend talk about ***building*** a program, as opposed to ***writing*** one."

The "building" metaphor led to many new ideas: planning; specification as blueprint; components; assembly; scaffolding; etc. But the idea that planning ***preceded*** construction remained.

In 1971, Harlan Mills (IBM) proposed that we should ***grow*** software rather than build it. We begin by producing a very simple system that runs but has minimal functionality and then add to it and let it grow. Ideally, the software grows like a flower or a tree; occasionally, however, it may spread like weeds.

The fancy name for growing software is ***incremental development***. There are many variations on this theme. The principal advantages of incremental development include:

- there is a working system at all times;
- clients can see the system and provide feedback;
- progress is visible, rather than being buried in documents;

- some kinds of errors can be avoided.

Incremental development also has some disadvantages:

- design errors become part of the system and are hard to remove;
- clients see possibilities and want to change requirements.

2.5 Errors

In general:

- a process model must recognize the possibility of errors and provide ways of correcting them;
- errors made early in development tend to be more serious (that is, more expensive to fix) than errors made later;
- the kind of errors made depend to some extent on the process model.

As an example, consider an error in the requirements. With the waterfall model, the error may not be noticed until acceptance testing, when it is probably too late to correct it. (Note that the client probably does not see the software running until the acceptance tests.)

You will sometimes see claims that an error in requirements costs 100 times (or 1,000 times, or "exponentially more") than an error in coding. These claims ***have not been substantiated***. It ***is*** true that, in a large project, the earlier that errors are caught, the better. Modern techniques, such as incremental development and prototyping, however, avoid these extreme situations.

In the incremental model, there is a good chance that a requirements error will be recognized as soon as the corresponding software is incorporated into the system. It is then not a big deal to correct it.

The waterfall model relies on careful review of documents to avoid errors. Once a phase has been completed, there is no stepping back. It is difficult to verify documents precisely and this is, again, a weakness of the waterfall model.

3 Extreme Programming

3.1 Twelve Features

Extreme Programming (XP) is a recent approach to software development introduced by Kent Beck (*Extreme Programming.* Kent Beck. Addison Wesley 2000.) The twelve key features of XP, outlined below in Beck's words, are:

The Planning Game Quickly determine the scope of the next release by combining business priorities and technical estimates. As reality overtakes, update the plan.

Small Releases Put a simple system into production quickly, then release new versions on a very short cycle.

Metaphor Guide all development with a simple shared story of how the whole system works.

Simple Design The system should be designed as simply as possible at any given moment. Extra complexity is removed as soon as it is discovered.

Testing Programmers continually write unit tests, which must run flawlessly for development to continue. Customers write tests indicating theat features are finished.

Refactoring Programmers restructure the system without changing its behaviour to remove duplication, improve communication, simplify, or add flexibility.

Pair Programming All production code is written with two programmers at one workstation.

Collective Ownership Anyone can change code anywhere in the system at any time.

Continuous Integration Integrate and build the system many times a day, every time a task is completed.

40-hour Week Work no more than 40 hours a week as a rule. Never allow overtime for the second week in a row.

On-site Customer include a real, live customer on the team, available full-time to answer questions.

Coding Standards Programmers write all code in accordance with rules emphasizing communication throughout the code.

These ideas are not new. They have been tried before and there have been many reports of failure. The point of XP is that, ***taken together***, these techniques do constitute a workable methodology.

3.2 Objections and Justifications

We will consider each point in turn, explaining it briefly, pointing out its disadvantages, and then showing how it works in the context of XP.

3.2.1 The Planning Game

Description: The plan tells you what to do during the next few days. It is not detailed, and it can be changed as required.

Drawbacks: A rough plan is not a sufficient basis for development. Constantly updating the plan would take too long and would upset customers.

Why it works in XP: The clients can update the plan themselves, using estimates provided by the programmers. The plan is sufficient to give the clients a vision of what you can achieve. Short releases will reveal defects in the plan. A client who is working with the team can spot potential for changes and improvements.

Kent Beck compares planning to driving. If you want to drive along a straight road, you don't carefully point the car in the right direction and then drive with your eyes closed. Instead, you point the car roughly in the right direction and then make small corrections as you go.

3.2.2 Small Releases

Description: A ***release*** is a working version of the software. Between releases, the software may be in an inconsistent state. "Small" releases mean obtaining a working version every week, or every month, rather than every six months, or every year. (Cf. Microsoft's ***daily builds***.)

Drawbacks: Small releases mean that too much time is spent on getting the release to work.

Why it works in XP: Planning focuses attention on the most important parts of the system, so even small releases are useful to clients. With continuous integration, assembling a release does not take much effort. Frequent testing reduces the defect rate and release testing time. The design is simple but may be elaborated later.

Any XP project starts with two programmers working on the first release. This pair generates enough work for a second pair to start, and so on.

3.2.3 Metaphor

Description: The metaphor is a "story" about the system. It provides a framework for discussing the system and deciding whether features are appropriate. A well-known example of a metaphor is the Xerox "desk-top" metaphor for user-interface design. Another is the "spread sheet" metaphor for accounting. Game are their own metaphor: knowledge of the game helps to define the program.

Drawbacks: A metaphor doesn't have enough detail. It might be misleading or even wrong.

Why it works in XP: There is quick feedback from real code to support the metaphor. Clients know the metaphor and can use it as a basis for discussion. Frequent refactoring means that you understand the practical implications of the metaphor.

3.2.4 Simple Design

Description: A simple design is an outline for a small software component. It has the smallest number of features that meet the requirements of the current phase. *As simple as possible, but not simpler* — Einstein.

Drawbacks: A simple design may have faults and omissions. Implementing the design might bring unanticipated problems to light. Components with simple designs might not integrate correctly into the system.

Why it works in XP: Refactoring allows you to correct design errors and omissions. The metaphor helps to keep the design process on track. Pair programming helps to avoid silly mistakes and to anticipate design problems.

3.2.5 Testing

Description: Write large numbers of simple tests. Provide a fully automatic testing process, so that all tests can be run with one click. Write tests ***before*** starting to code.

Drawbacks: There isn't time to write all those tests. Programmers don't write tests — testing teams do.

Why it works in XP: If the design is simple, the tests should be simple too. With pair programming, one partner can think of tests while the other is coding. Seeing tests work is good for morale. Clients like seeing tests working.

There are many tests and most of them are run automatically. Code that is "obviously correct" is not tested.

3.2.6 Refactoring

Description: Refactoring is making changes to the software so that it conforms to the design, to coding standards, and is simple and elegant. As code is developed, it has a tendency to get messy; refactoring is removing the messiness.

Drawbacks: Refactoring takes too long and is hard to control (when do you stop?). Refactoring may break a working system.

Why it works in XP: Collective ownership of code makes refactoring easier and more acceptable to programmers. Coding standards reduce the task of refactoring. Pair programming makes refactoring less risky and adventurous. The design is simple. You have a set of tests that you can run at any time during the refactoring process. Continuous integration gives rapid feedback about refactoring problems. You are not over-worked and over-tired and are unlikely to make silly mistakes.

3.2.7 Pair Programming

Description: Two programmers work together at a workstation. One has the keyboard and enters code and tests. The other watches, thinks, comments, criticizes, and answers questions. Periodically, the pair exchange roles. Pair programming is one of the more controversial features of XP, yet most programmers who have tried it say they like it.

Drawbacks: Pair programming is slow. What if people argue all the time?

Why it works in XP: Coding standards avoid trivial arguments. No one is over-tired or over-worked. Simple design and writing tests together helps to avoid misunderstanding. Both members of the pair are familiar with the metaphor and can discuss their work in its terms.

If you don't like your partner, find someone else. If one partner knows a lot more than the other, the second person learns quickly.

3.2.8 Collective Ownership

Description: The code is owned by the whole team. Anyone can make changes to any part of the system. This contrasts with traditional processes, in which each piece of code is ***owned*** by an individual or a small team who has complete control over it and access to it.

Drawbacks: It is very dangerous to have everything potentially changing everywhere! The system would be breaking down all the time. Integration costs would soar.

Why it works in XP: Frequent integration avoids breakdowns. Continuously writing and running tests also avoids breakdowns. Pair programmers are less likely to break code than individual programmers. Coding standards avoid trivial arguments (the "curly bracket wars").

Knowing that other people are reading your code makes you work better. Complex components are simplified as people understand them better.

3.2.9 Continuous Integration

Description: The system is rebuilt very frequently, perhaps several times a day. (Don't confuse ***continuous integration*** with ***short releases***, in which a new version with new features is built.) The new system is tested.

Drawbacks: Integration takes too long to be repeated frequently. Frequent integration increases the chances of accidentally breaking something.

Why it works in XP: Tests are run automatically and quickly, so that errors introduced by integration are detected quickly. Pair programming reduces the number of changed modules that have to be integrated. Refactoring maintains good structure and reduces the chance of conflicts in integration.

Simple designs can be built quickly.

3.2.10 40-hour Week

Description: Many software companies require large amounts of overtime: programmers work late in the evening and at weekends. They get over-tired, make silly mistakes. They get irritable, and waste time in petty arguments. The XP policy ensures that no one works too hard. Developers work no more than 40 hours in any one week. If they work overtime this week, they are not allowed to work overtime next week.

Drawbacks: 40 hours a week is not enough to obtain the productivity required for competitive software (at least, in the US!).

Why it works in XP: Good planning increases the value per hour of the work performed; there is less wasted time. Planning and testing reduces the frequency of unexpected surprises. XP as a whole helps you to work rapidly and efficiently.

3.2.11 On-site Customer

Description: A representative of the client's company works at the developer's site all the time. (This may not be feasible if the client is very small, e.g., a one-person company.) The client is available all the time to consult with developers and monitor the development of the software.

Drawbacks: The representative would be more valuable working at the client's company.

Why it works in XP: Clients can contribute, e.g., by writing and commenting on tests. Rapid feedback for programmer questions is valuable.

3.2.12 Coding Standards

Description: All code written must follow defined conventions for layout, variable names, file structure, documentation, etc. Note that this is a ***local*** standard, not a standard defined by XP.

Drawbacks: Programmers are individualists and refuse to be told how to write their code.

Why it works in XP: The XP process makes them proud to be on a winning team.

3.3 Assessment

Kent Beck estimates there are between 500 and 1500 XP projects in progress now. Opponents believe there are fewer: Craig Larman says that the ***only*** XP processes are those that Beck supervises.

The problem is that some groups pick a subset of the twelve features of XP and claim that they are "doing XP", whereas Beck's viewpoint is that XP consists of applying ***all*** twelve features on a project.

XP is an appropriate paradigm for small and medium projects. It has not been tried with teams of more than about 60 programmers and it is not clear that it would succeed at this scale.

Nevertheless, XP contains a lot of good ideas and a lot of good psychology. XP assumes that people are most productive if they are doing high-quality work and are not over-worked. XP recognizes that documentation can be a liability rather than an asset.

4 Introduction to UPEDU

During the 1980s, many approaches to software development were introduced. Some people proposed entire methodologies while others proposed components of a methodology. The players included:

Grady Booch introduced object oriented design and modelling techniques;

Erich Gamma introduced design patterns;

David Harel introduced state charts for dynamic modelling;

Ivar Jacobson introduced use cases;

Bertrand Meyer defined programming by contract;

James Odell introduced classification concepts;

James Rumbaugh defined Object Modelling Technique (OMT);

Steven Shlaer and Stephen Mellor defined object lifecycles.

In the early stages, most of these people hyped their own ideas and put down other people's ideas. It gradually became clearer that they were all talking about pretty much the same thing, and they began to combine their ideas rather than competing. Grady Booch founded Rational Software Corporation and several of the others joined it. Today, Rational claims revenues of about $500 million per year; 98% of the top (Fortune 100) companies use Rational tools for software development.

Rational's first significant product was the ***Unified Modelling Language*** (UML). UML defines a large number of symbols and diagrams that are used to model software and system products. The basic idea is that a system cannot be represented by one diagram: many diagrams are needed, and each diagram provides a diffferent view of the system. Since it would be infeasible to maintain consistency of a large collection of diagrams, Rational provide ***Computer Aided Software Engineering*** (CASE) tools to draw and modify diagrams. In some cases, it is possible to generate parts of the code automatically from diagrams.

Rational's next product was the ***Rational Unified Process*** (RUP). RUP is a very general model that can be specialized in many different ways. For example, both the Waterfall Model (WM) and Extreme Programming (XP) can be described using RUP, although it is unlikely that a RUP developer would actually use either of them.

RUP is an ***iterative*** process that is intended for the ***incremental development*** of software. The phases of the WM (requirements, design, implementation, test, etc.) can be recognized in RUP, but they may occur many times during development. For example, requirements may be derived for each of many components instead of once only for the entire product.

UPEDU is a simplified version of RUP. "UPEDU" stands for Unified Process for Education.

UPEDU is based on ***development cycles***. A project generally consists of several development cycles that may be performed sequentially or in an overlapping way (a new cycle begins before previous cycles are complete).

UPEDU uses a number of words in a particular way. The following sections explain how these words are used; Section 4.4 provides brief definitions of each word.

4.1 Lifecycles and Processes

A ***lifecycle*** is the entire history of a product. It begins with an idea for a product and ends when the product ceases to exist. In between, there will many activities: determining the requirements, designing the product, constructing it, testing it, maintaining it, and so on.

A ***process*** is a discipline for moving a product through its lifecycle. There may be different processes corresponding to a single lifecycle. For example, many buildings have a similar lifecycle: planning, foundation, walls, roof, finishing, demolition. However, the process of contructing the building depends on its size and function.

UPEDU consists of a collection of ***development cycles***. The development cycles may be sequential (that is, each cycle follows the next) but usually they are not: on a large project, several development cycles may be in progress concurrently.

A development cycle has four phases. The end of each phase is marked by a ***milestone***. The four phases, and their milestones (in *italics*) are:

Inception: Create a justification for this phase; may involve requirements analysis, risk analysis, prototyping. *(Objective.)*

Elaboration: Develop and analyze models for this phase; may involve development plans, user documentation, release description. *(Architecture.)*

Construction: Code the program or component and perform quality assurance. *(Operational Capability.)*

Transition: Transfer the program or component to the user; ensure that the user has adequate documentation and other resources to make use of it. *(Release.)*

4.2 Disciplines: what people actually do

A software process should define: ***who*** does the work; ***what*** they do; ***when*** they do it; and ***how*** they do it. A set of such definitions is a ***discipline***. There are two main families of disciplines: ***engineering disciplines*** and ***management disciplines***. The result of completing a discipline is a model of the system. A ***model*** is one of several kinds of abstract and simplified representation of the system or one of its parts. In increasing level of detail, we may use a diagram, a description, and code to describe a software component.

Engineering Disciplines

Requirements Discipline: find out what the system is supposed to do and build a ***use-case model***.

Analysis and Design Discipline: translate the requirements into a specification that describes how to implement the system and build a ***design model***.

Implementation Discipline: write and test the code for each component, giving an ***implementation model***.

Test Discipline: assess the level of quality achieved and build a ***test model***.

Person	Design	Implement	Test
Ann	A		C
Bill		B	
Chris	A	B	
Diane		B	C
Eddie	A		C

Figure 2: Roles in a small project

Management Disciplines

Configuration and Change Management Discipline: control modifications and releases; report project status; build and maintain a ***project repository***.

Project Management Discipline: select objectives, manage risks, overcome constraints, construct a ***schedule***.

Roles, Activities, and Artifacts The "central triangle" of UPEDU consists of the following relationship: ***roles*** perform ***activities*** that modify ***artifacts***. We examine each of the three components of the triangle in turn.

Role: Why do we say that a "role" performs an activity rather than saying that a "person" performs the activity? The reason is that people perform different roles. One of the goals of UPEDU is to get away from the stereotyped view that there are designers, coders, testers, and so on. Instead, there is a design role, a coding role, and a testing role; one person may perform different roles at different times. A role may also be performed by a team rather than a single person.

Figure 2 illustrates roles in a small project. The letters A, B, and C stand for different development cycles. During cycle A, Ann, Chris, and Eddie share the role of designer; during cycle B, Bill, Chris, and Diane share the role of implementor; and during cycle C, Ann, Diane, and Eddie share the role of tester. As a consequence of their work, Ann is familiar with both design and testing, Chris is familar with both design and implementation, and so on.

Activity: An activity is a small, usually indivisible, piece of work with a definite outcome. We can always say unambiguously that an activity has either been performed or it has not. Of course, an activity may be performed in steps — thinking, planning, designing, writing, correcting, reporting, and so on — but none of these intermediate parts appear in the project.

Activities are typically short, varying from a few hours to a few days or two. An activity is performed by a single role (which, as above, may be more than one person).

An activity modifies an artifact. In some cases, it may create the entire artifact; in other cases, it may modify the artifact. For example, one activity might create Version 1.0

of the design, and another activity might incorporate changes in the design leading to Version 1.1.

Artifact: An artifact is a piece of information produced by completing an activity. Artifacts include reports, documents, files, models, diagrams, source code, executable code, and so on.

Artifacts are used as inputs as well as outputs. For example, the job of a coder is to use the design (an artifact) to produce source code (another artifact).

4.3 Iterations

As we have already seen, software construction proceeds by iterating (or repeating) development cycles. Although each development cycle consists of the same four phases (inception, elaboration, construction, and transition), the relative weight of each discipline varies from one cycle to another.

For example, the first development cycle will usually focus on gathering requirements. Late in the project, a development cycle would be more likely to focus on coding and testing. However, since almost all projects involve changing requirements, the requirements may have to be considered in late cycles as well as early ones.

Consequently, software development should be seen as the application of all disciplines simultaneously, but with different focuses depending on the state of the project.

The choice of development cycles, and the choice of activities within a development cycle, can be systematic or opportunistic. Realistically, this distinction should be viewed not as either/or but rather a continuous spectrum of which we are defining the end points.

In ***systematic development***, the developers have access to all of the information they need and iterations proceed in an orderly, planned way. The project schedule might predict the expected work for several months ahead.

In ***opportunistic development***, the developers do not have enough information to plan far ahead. They might be expecting revised requirements, or waiting for the results of a timing test. Under these circumstances, the choice of the next iteration may not be made until the transition phase of the current iteration.

Entire projects may be systematically or opportunistically developed. In practice, however, it is more likely that the balance will change during the project. A new project might start opportunistically but, after a while, stabilize and become systematic. More frequently, a project starts systematically and, when problems arise, drifts towards opportunism.

Opportunism does not matter provided that the project stays under control. The development cycle model is intended to help retain control of the project.

In the waterfall model, it may become clear during the implementation phase that the requirements cannot be implemented as planned. This can lead to a panic mode of development in which programmers spend many hours of overtime trying to do impossible things. Edward Yourdon has a name for such projects ("death march") and has written a book about them: *Death March: The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects* (Yourdon Computing Series).

4.4 UPEDU Glossary

UPEDU uses many words in a particular way. This section provides a brief definition of each special word for convenience; the words are described in more detail in the rest of the notes.

Activity: a small, measurable piece of work performed by a ***role***.

Analysis and Design Discipline: a ***discipline*** to obtain a detailed description of how the product will work.

Artifact: a tangible object that is produced by a ***role*** performaing an ***activity*** during product development. Artifacts include documents, reports, test results, analyses, pieces of code, etc.

Concept: see ***inception***.

Configuration and Change Management Discipline: a ***discipline*** to obtain a detailed description of how changes, modifications, enhancements, and corrections to the product will be controlled.

Construction: the third phase of an iteration, during which the product of the iteration is built, coded, or assembled.

Design: The second phase of an iteration, during which decisions about how the product of the iteration will work are considered and plans are made. Also called ***elaboration phase***.

Discipline: a set of ***activities*** that are performed together in a well-defined way.

Elaboration: see ***design***.

Implementation Discipline: a ***discipline*** to obtain a detailed description of how the product will be coded and tested.

Inception: The first phase of an iteration, during which all aspects and implication of the iteration are considered but not much is actually done. Also called ***concept phase***.

Life Cycle: a dscription of the sequence of events needed to create a product.

Milestone: a well-defined point in time at which a phase of the project ends and another one begins. A milstone is often associated with an ***artifact*** such as a report or finished piece of code.

Model: a simplified or abstracted view of a system or a system component. A model may be a diagram, a description, or even a piece of code.

Process: a set of ***disciplines*** that are followed in order to build a product. A process defines Who is doing What, When, and How.

Requirements Discipline: a ***discipline*** to obtain a detailed description of the desired behaviour of the product.

Role: a set of responsibilities that enable a person to complete an ***activity***. A role may be performed by one or several people; a person may play different roles at different times.

Transition: the fourth and final phase of an iteration, during which the product of the iteration is assessed and tested (also called ***final phase***). "Transition" emphasizes that this iteration will be linked to the next iteration.

5 Modelling

Modelling is an important activity in Software Engineering. In this section, we look at the role played by modelling in various disciplines, and how modelling contributes to software development.

Who models? Amost any discipline in which complex artifacts are constructed makes use of modelling. Architects and designers of buildings, bridges, ships, cars, shopping centres, highways, and software all make use of models in one form or another.

Why model? Modelling helps us to understand a complex artifact without having to build the artifact itself. More importantly, modelling may help us to understand a particular aspect of the artifact. This is illustrated in the table below, in which the left column defines an aspect of interest and the right column gives examples of models that can contribute to understanding that aspect.

Aspect	Examples
Visual appearance	buildings, bridges, cars, aircraft, ships, cameras, shopping centres
Aerodynamic behaviour	aircraft, cars, bridges
Hydrodynamic behaviour	ships, harbours, bridge foundations
Acoustic behaviour	concert halls, cinemas, convention centres
Sight lines (visibility)	theatres, concert halls

A model may also be an effective way of explaining the artifact to a client. Architects, for example, construct models of buildings and building complexes so that their clients can appreciate the visual effect of the propsed building in its environment.

Increasingly, physical models are being replaced by computer simulations. These simulatoins are, of course, just another form of model.

What is a model? A model accurately represents one or more chosen aspects of a complex artifact that (usually) has not yet been built. The model is smaller, simpler, and cheaper than the actual artifact.

The aspect that the model represents is often appearance. For example, in the car industry, the potential popularity of a new kind of car is assessed by showing models of it to people who might buy it. The model does not have a motor and cannot be used to judge performance, comfort, or drivability.

Other aspects of the model are also important. Models of cars and aircraft are tested in wind tunnels to determine how the artifact will behave on the road or in the air. Models of ships are tested in tanks that can simulate motion through the water and the effect of waves. Increasdingly, even this kind of modelling is being replaced by computer simulation, as computers become more powerful and simulation techniques become more sophisticated.

A flight simulator is a particularly sophisticated form of model: it models the cockpit of an aircraft in gret detail, but nothing else. The behaviour of the model is close enough to the real thing that pilots receive extensive training in simulators.

A diagram is a simple form of model. Electronic engineers draw diagrams of circuits ("schematics") and discuss and analyze them before constructing the actual circuit. Circuits are understood so well that the behaviour of the circuit is unlikely to differ much from the behavioour predicted from the schematic.

Software is usually modelled with diagrams. There is a long history of diagramming in software development; many kinds of diagrams were used at one time but are now considered obsolete. The Universal Modeling Language (UML) is based on diagrams.

6 Requirements

The ***requirements*** for a software project consist of a detailed description, written in non-technical language, of the capability that the completed system must have in order to satisfy the clients.

By "non-technical language" here, we mean language that does not use software- or computer-related terminology. The language may be "technical" in the client's domain. For example, the requirements for a patient-monitoring system might contain medical jargon.

The requirements are written for the benefit of stakeholders of the system. A ***stakeholder*** is any person or group who will be affected by the system. Stakeholders include clients, suppliers, end users, project managers, software engineers, and perhaps others. For a Flight Control System, even the public might be considered a stakeholder, because everyone wants assurance that flying is safe.

Specifically, requirements are written for:

Clients: to ensure that they get what they want or need.

Designers: to ensure that they design the system that is wanted.

Implementors: to ensure that coding details are correct.

Testers: to ensure that the system does what it is supposed to do.

Technical writers: to prepare user manuals and other documentation.

Trainers: to prepare procedures for training end users.

A set of requirements is often referred to as an ***SRS***, short for Software Requirements Specification.

Requirements are ***not fixed*** but change over the duration of the project. Change rates vary from 1% to 10% per month (even higher rates are possible but make the project difficult to manage). Causes of change include:

- Internal changes: the developers realize that a requirement is not feasible or, alternatively, that greater functionality can be provided without extra effort.
- Technology changes: bigger memories; bigger, faster disks; better monitors; faster networks; etc.
- Economic changes: a downturn or upturn in the economy may make a product more or less useful.
- Consumer preferences: a market survey may show that people will no longer pay more than $20 for a product with an estimated street price of $25.
- Competition: another company releases a similar product with features that our product does not have.
- Regulations: the government introduces new requirements for products in our domain.
- Clients: change their needs on the basis of prototypes or other studies.

Building a large product on the basis of a fixed set of requirements is a recipe for disaster. The principal weakness of the Waterfall Model was its failure to accept that requirements change during the development of a product.

6.1 Activities

In UPEDU, requirements consists of five ***activities***. As usual, the terms "analysts" and "reviewers" refer to actors, not to people.

Understand the problem (analysts)

(1) elicit client needs

(2) identify actors and use cases

Define the system (analysts)

(3) structure use-case models

(4) detail use cases

Review (reviewers)

(5) review requirements

6.2 Artifacts

The results of requirements gathering are used to construct five ***artifacts***:

1. **Vision:** a non-technical, high-level view of the system; may be written by clients or suppliers.
2. **Glossary:** a dictionary of domain-specific terms and special usages.
3. **Software Requirements Specification:** often divided into two parts, called either "SRS" and "Supplementary Requirements" or "Functional requirements" and "Non-functional requirements".
4. **Use Cases:** description of the interactions between users and the system.
5. **Use Case Model:** a complete description of all the actors who will use the system and how they interact with it.

6.2.1 Vision

The ***Vision*** typically answers questions such as (not all of the questions will be applicable to any one product):

- What is this system for?
- Why does it work this way?
- What do users need?

- What are the goals and objectives of the system?
- What are the intended markets?
- What environment should users work in?
- What platforms are supported?
- What are the main features of the product?

6.2.2 Glossary

The ***Glossary*** ensures that everyone involved in the project agrees on the meaning of particular words and phrases. It is important because many words are used in quite different ways in different disciplines. For example, “stroke” means one thing to a car mechanic and another to a cardiologist.

6.2.3 SRS

The ***SRS*** has a number of important functions. It provides the basis for:

- agreement between customer and supplier. There may be other components of the agreement, such as legal documents.
- costing and scheduling.
- validation and verification. You cannot test software unless you know what it is supposed to do.
- all forms of maintenance.

The distinction between ***functional*** and ***non-functional*** requirements is not sharp. Functional requirements describe the “function” of the system in a precise way. For example:

Event: user clicks on `File` button in main toolbar.

Response: system displays the `File Menu` with options to `Open`, `Close`, `Save`, `Save As`, and `Quit`.

Non-functional requirements include characteristics that are hard to define precisely, such as usability, reliability, supportability, and performance.

A requirement that is “non-functional” in one context may be “functional” in another. For example, the requirement “the system shall respond in 1 second” might be non-functional for an accounting system but functional for a flight-control system.

6.2.4 Use Cases

Use cases describe interactions between the end users and the system.

- A ***use case*** describes a single interaction.
- A ***use case class*** describes a group of related use cases, often involving several actors.
- A ***scenario*** is an instance of a use case class.
- A ***use case model*** includes all of the use cases for the system.

Figure 3 shows a simple use case (adapted from *Object Oriented Software Engineering: a Use Case Driven Approach* by Ivar Jacobson *et al.*, pages 348–9).

6.3 Eliciting Requirements

Clients and suppliers work together during requirements gathering. The clients explain their business and the intended role of the software. The analysts observe, ask questions, and conduct formal interviews. There are three main aspects to focus on:

- the needs of the clients
- understanding the problem domain
- understanding the form of solution that is required

6.3.1 An Example

As an example, suppose that you are eliciting requirements for a grocery store that wishes to automate its operations. Your activities might include:

- watching how the store operates
- interviewing people who work in different parts of the store (including areas inaccessible to the public, such as store-rooms and accounting offices)
- estimating volumes required (how many customers are served, how many items do they buy, how many suppliers deliver goods to the store, how frequent are the deliveries, etc.)
- determining the shelf-life of various kinds of goods (to determine re-order rates)
- determining the amount of stock held

6.3.2 Hints

Here are some hints for writing requirements.

- Include input/output specifications.
- Give representative, but possibly incomplete, examples.

General Course

1. The `office personnel` insert a request for a customer withdrawal at a certain date and a certain warehouse. The window below [not shown here] is displayed.
2. The information filled in by the `office personnel` when giving the command issuing a customer withdrawal.
3. The information filled in by the `office personnel` is `customer`, `delivery date`, and `delivery place`. The name of the `office personnel` staff member is filled when the window is opened but can be changed. The staff member selects items from the browser and adds the quantities to be withdrawn from the warehouse.
4. The following items are checked immediately:
   (a) the customer is registered;
   (b) the items ordered are in the warehouse
   (c) the customer has the right to withdraw these items.
5. The system initiates a plan to have the items at the appropriate warehouse at the given date. If necessary, the system initiates transport requests.
6. At the date of delivery, the system notifies a `warehouse worker` of the delivery.
7. The `warehouse worker` issues a request to the `forklift operator` to move the items requested to the loading platform. The `forklift operator` executes the transportation.
8. When the customer has fetched the items, the `warehouse worker` marks the withdrawal as `processed`.

Alternative Courses

- *There are not enough items in the warehouse:*
- *Customer is not registered:*
- *Customer does not have the right to withdraw these items:*

  The system notifies the `office personnel` and the withdrawal is not executed.

Figure 3: A simple use case. `Typewriter font` indicates words with special significance.

- Use models: mathematical (e.g. regular expressions); functional (e.g. finite state machines); timing (e.g. augmented FSM).
- Distinguish mandatory, desirable, and optional requirements.

  $\surd$ The user interface must use X Windows exclusively.

  $\surd$ The software must provide the specified performance when executed with 16Mb of RAM. Preferably, it should be possible to execute the software with 8Mb of RAM.

  $\surd$ Sound effects are desirable but not required.
- Anticipate change. Distinguish what should not change, what may change, and what will probably change.

  $\surd$ The FSM diagram will contain at least nodes and arcs.

  $\surd$ The software may eventually be required to run on machines with the EBCDIC character set.

A well-written SRS will reduce development effort by avoiding (expensive) changes later, in design and implementation phases. The following examples include both good usage (indicated by $\surd$) and bad usage (indicated by $\times$) (see also Robillard, page 85).

- The SRS should define all of the software requirements **but no more**. In particular, the SRS should not describe any design, verification, or project management details.

  $\times$ The table is ordered for binary search.

  $\times$ The table is organized for efficient search.

  $\surd$ The search must be completed in time $O(\log N)$.
- The SRS must be **unambiguous**.
  - There should be exactly one interpretation of each sentence.
  - Special words should be defined. Some SRDs use a special notation for words used in a specific way: `!cursor!`.
  - Avoid "variety" — good English style, but not good SRS style.
  - Careful usage.

    $\times$ The checksum is read from the *last* record.

    Does "last" mean (a) at end of file, (b) most recently read, or (c) previous?

    $\surd$ . . . from the final record of the input file.

    $\surd$ . . . from the record most recently processed.
- The SRS must be **complete**. It must contain all of the significant requirements related to functionality (what the software does), performance (space/time requirements), design constraints ("must run in 64Mb"), and external interfaces. The SRS must define the response of the program to all inputs.
- The SRS must be **verifiable**. A requirement is *verifiable* if there is an effective procedure that allows the product to be checked against the SRS.

  $\times$ The program must not loop.

  $\times$ The program must have a nice user interface.

  $\surd$ The response time must be less than 5 seconds for at least 90% of queries.

- The SRS must be **consistent**. A requirement must not conflict with another requirement.

  × When the cursor is in the text area, it appears as an I-beam. . . . During a search, the cursor appears as an hour-glass.

- The SRS must be **modifiable**. It should be easy to revise requirements safely — without the danger of introducing inconsistency. This requires:
  - good organization;
  - table of contents, index, extensive cross-referencing;
  - minimal redundancy.

- The SRS must be **traceable**.
  - The origin of each requirement must be clear. (Implicitly, a requirement comes from the client; other sources should be noted.)
  - The SRS may refer to previous documents, perhaps generated during negotiations between client and supplier .
  - The SRS must have detailed numbering scheme.

- The SRS must be usable during the later phases of the project. It is **not** written to be thrown away! A good SRS should be of use to maintenance programmers.

The SRS is prepared by both the supplier with help and feedback from the client.

- The client (probably) does not understand software development.
- The supplier (probably) does not understand the application.

6.4 Requirements Workshop

A ***requirements workshop*** is an effective way of getting requirements quickly. It should be an intensive meeting in which many or all of the interested parties (stakeholders) get together. Features of a successful workshop include:

- Clients, end users, designers, programmers, testers, and others are all given a chance to hear about the system and to comment on it. It is unlikely that there will be opportunities for this kind of interaction later in the development process.
- Results from the workshop should be quickly (or even immediately) available.
- Brainstorming may be productive: participators are encouraged to submit any ideas, however wild; some of the ideas may turn out to be useful, or to lead the discussion into fruitful areas.
- "Storyboarding" means outlining possible scenarios involving the system. The stories may draw attention to aspects that might otherwise have been neglected. Some stories may be developed into use cases.
- People who will be involved in the final system can act out their roles. Others can simulate their roles.
- Existing requirements can be critically reviewed.

6.5 Prototyping

Prototyping is an important technqiue that can be practiced at any part of the development cycle but is often viewed as a part of requirements elicitation.

A ***prototype*** is a piece of software (or, occasionally, hardware) that provides some of the functionality of the system.

- A ***UI prototype*** demonstrates how the user interface of the system will work but does not provide any functionality. Users can see what screens and windows will look like, can experiment with menu hierarchies, can look at presentations of data, but cannot actually do anything useful. A UI prototype helps clients and end users decide whether the UI will be sufficient for their purposes, easy to use, easy to learn, etc.
- A ***Wizard of Oz*** prototype simulates the system by having a person "behind the scenes" simulating responses. For example, users might enter queries and the "wizard" invents replies. This is more realistic than giving dummy or random replies.
- Key algorithms may be prototyped. If the main risk is that the algorithm might give incorrect answer, a high-level language can be used for quick results. If the main risk is poor performance, the inner loops of the algorithm can be coded and used in timing tests.

There are three main kinds of prototypes:

A throwaway prototype is built quickly, used for its intended purpose, and then abandoned.

An evolutionary protoype is built according to standard coding practices but provides just enough functionality to serve its purpose as a prototype. It is then allowed to evolve into a part of the system by completing the unimplemented parts.

An operational prototype becomes a part of the final system. It is built according to standard coding practices and is a complete component, but it provides the functionality of only one (possibly small) part of the system.

The problem with a throwaway prototype is that people (especially managers) are reluctant to throw something away because it seems like a waste of effort. This view is incorrect, because the throwaway prototype has fulfilled its intended function, which is to clarify requrements. The prototype is likely to be a "quick and dirty" implementation but, since it appears to do something, managers may demand that it be "cleaned up" and used in the system. This is a mistake: code that was coded in a "quick and dirty" manner will always be quick and dirty, never polished.

One way to ensure that a prototype intended to be thrown away is ***not*** kept is to write it in a language that is different from the system programming language. For example, if the target language is C++ or Java, prototypes might be written in Perl or Javascript.

The problem with evolutionary and operational prototypes is to ensure that they are written in accordance with system standards. Since they are "prototypes", programmers may feel that they do not warrant as much effort as production code. Since they are built early in the project, system standards may not even have been established.

In a truly incremental project, the distinction between prototypes and production code is not sharp. Prototypes are simply the first increments of the product to be delivered.

7 Design

This section is adapted from notes made from an earlier version of a software engineering course. It mentions process models such as Fusion that are predecessors of RUP.

Design is conveniently split into three parts: the architecture of the system, the module interfaces, and the module implementations. We give an overview of these and then discuss each part in more detail.

7.1 Overview of Design

7.1.1 Design Documentation

The recommended design documents for this course consist of:

AD	— Architectural Design
MIS	— Module Interface Specifications
IMD	— Internal Module Design

The document names also provide a useful framework for describing the design process.

The documents may be based on UPEDU templates and they may be supplemented with selected UPEDU diagrams.

7.1.2 Architectural Design

The AD provides a **module diagram** and a brief **description** of the role of each module. In UPEDU, the AD also includes diagrams and descriptions of use cases.

7.1.3 Module Interface Specifications

Each module provides a set of **services**. A module interface describes each service provided by the module. Most modern designs are object oriented: in this case, a module corresponds roughly to a class.

"Services" are usually functions. A module may also provide constants, types, and variables. Constants may be provided by functions which always return the same result: there is a slight loss of efficiency, but a change does not require recompiling the entire system. It is best not to export variables; if variables are exported, they should be read-only (inlined access functions in C++).

To specify a function, give its:

- name;
- argument types;
- a **requires clause** — a condition that must be true on entry to the function;

- an **ensures clause** — a condition that will be true on exit from the function;
- further comments as necessary.

The requires clause is a constraint on the caller. If the caller passes arguments that do not satisfy the requires clause, the effect of the function is unpredictable.

The ensures clause is a constraint on the implementer. The caller can safely assume that, when the function returns, the ensures clause is true.

The requires and ensures clause constitute a contract between the user and implementor of the function. The caller guarantees to satisfy the requires clause; in return, the implementor guarantees to satisfy the ensures clause.

Example of a function specification:

```
double sqrt (double $x$)
requires:  $x \geq 0$
ensures:   $|result^2/x - 1| < 10^{-8}$
```

Example of a complete but simple module:

```
module NameTable
imports NameType
Boolean ValidTable

  void create ()
    requires:
    comment:  could also write requires nothing
    ensures:  ValidTable, $Entries = 0$

  void insert (NameType X)
    requires:  ValidTable
    ensures:  $X \in Table$
    comment:  no error if $X \in Table$ before call

  void delete (NameType X)
    requires:  ValidTable, $X \in Table$
    ensures:  $X \notin Table$

  Bool lookup (NameType X)
    requires:  ValidTable
    ensures:  $result = X \in Table$
```

7.1.4 Internal Module Description

The IMD has the same structure as the MIS, but adds:

- data descriptions (e.g. binary search tree for NameTable);
- data declarations (types and names);
- a description of how each function will work (pseudocode, algorithm, narrative, . . .).

A UPEDU class diagram provides both MIS and IMD, but the IMD requires additional documentation to describe the precise effect of each function.

7.2 Remarks on Design

What designers actually do?

- Construct a mental model of a proposed solution.
- Mentally execute the model to see if it actually solves the problem.
- Examine failures of the model and enhance the parts that fail.
- Repeat these steps until the model solves the problem.

Design involves:

- understanding the problem;
- decomposing the problem into goals and objects;
- selecting and composing plans to solve the problem;
- implementing the plans;
- reflecting on the product and the process.

But when teams work on design:

- the teams create a shared mental model;
- team members, individually or in groups, run simulations of the shared model;
- teams evaluate the simulations and prepare the next version of the model.
- Conflict is an inevitable component of team design: it must be managed, not avoided.
- Communication is vital.
- Issues may "fall through the cracks" because no one person takes responsibility for them.

7.3 Varieties of Architecture

The AD is a "ground plan" of the implementation, showing the major modules and their interconnections.

An arrow from module A to module B means "A needs B" or, more precisely, "a function of A calls one or more of the functions of B".

The AD diagram is sometimes called a "Structure Chart". In UPEDU, it is a class diagram.

The AD is constructed in parallel with the MIS. A good approach is to draft an AD, work on the MIS, and then revise the AD to improve the interfaces and interconnections.

7.3.1 Hierarchical Architecture

The Structure Diagram is a tree. Top-down design:

- tends to produce hierarchical architectures;
- is easy to do;
- may be suitable for simple applications;
- does not scale well to large applications;
- yields leaves of the tree that tend to be over-specialized and not reusable.

7.3.2 Layered Architecture

The structure diagram has layers. A module may use only modules in its own layer and the layer immediately below ("closed" architecture) or its own layer and all lower layers ("open" architecture).

- Layers introduced by THE system and Multics (MIT, Bell Labs, General Electric). (UNIX was designed in opposition to Multics).
- Programs with "utility functions" are (informal) layered systems.
- Requires a combination of top-down and bottom-up design. Top-down ensures that overall goals are met. Bottom-up ensures that lower layers perform useful and general functions.
- High layers perform high-level, general tasks. Low layers perform specialized (but not too specialized!) tasks.
- Modules in low layers should be reusable.

7.3.3 General Architecture

Arbitrary connections are allowed between modules.

- Not recommended: cf. "spaghetti code".
- May be an indication of poor design.
- Avoid cycles. Parnas: "nothing works until everything works".

7.3.4 Event-Driven Architecture

In older systems, the program controlled the user by offering a limited choice of options at any time (e.g. by menus).

In a modern, event-driven system, the user controls the program. User actions are abstracted as **events**, where an event may be a keystroke, a mouse movement, or a mouse button change.

The architecture consists of a module that responds to events and knows which application module to invoke for each event. For example, there may be modules related to different windows.

This is sometimes called the Hollywood approach: "Don't call us, we'll call you". Calling sequences are determined by external events rather than internal control flow.

Modules in an event-driven system must be somewhat independent of one another, because the sequence of calls is unknown. The architecture may be almost inverted with respect to a hierarchical or layered architecture.

Example: layered system.

Layer 0		Controller	
Layer 1	Regular Processing	Correction Processing	Report Processing
Layer 2	Database Manager	User Interface	

Event-driven version:

Layer 0		User Interface	
Layer 1	Regular Processing	Correction Processing	Report Processing
Layer 2		Database Manager	

Here is a possible layered architecture for a graph editor.

Layer 0		X Windows (simplib)	
Layer 1	User Interface	Command Processor	
Layer 2	Editor	Filer	Menu
Layer 3	Graph	I/O	
Layer 4		Objects	
Layer 5	Node	Arrow	Text
Layer 6	grdraw	grdialoglib	

7.3.5 Subsumption Architecture

A **subsumption architecture**, sometimes used in robotics, is an extension of a layered architecture. The lower layers are autonomous, and can perform simple tasks by themselves. Higher layers provide more advanced functionality that "subsumes" the lower layers. Biological systems may work something like this. Subsumption architectures tend to be robust in that failure in higher layers does not cause failure of the entire system.

7.4 Designing for Change

What might change?

1. Users typically want more:
   - commands;
   - reports;
   - options;
   - fields in a record.

   Solutions include:

   Abstraction Example: abstract all common features of commands so that it is easy to add a new command. The ideal would be to add the name of a new command to a table somewhere and add a function to implement the command to the appropriate module.

   Constant Definitions There should be no "magic numbers" in the code.

   Parameterization If the programming language allows parameterization of modules, use it. C++ provides templates. Ada packages can be parameterized.

2. Unanticipated errors may occur and must be processed.

   Incorporate general-purpose error detection and reporting mechanisms. It may be a good idea to put all error message text in one place, because this makes it easier to change the language of the application.

3. Algorithm changes might be required.

   Usually a faster algorithm is needed. As far as possible, an algorithm should be confined to a single module, so that installing a better algorithm requires changes to only one module.

4. Data may change.

   Usually a faster or smaller representation is needed. It is easy to change data representations if they are "secrets" known to only a small number of modules.

5. Change of platform (processor, operating system, peripherals, etc)

   Keep system dependencies localized as much as possible.

6. Large systems exist in multiple versions:
   - different releases
   - different platforms
   - different devices

   We need **version control** to handle different versions. RCS is a version control program.

   Versions often form a tree, but there are variations. The more complicated the "version graph", the harder it is to ensure that all versions are consistent and correct.

Note that a single technique can accommodate a large proportion of changes: good module structure, and secrets kept within modules. Avoid distributing information throughout the system.

7.5 Module Design

Ideas about module design are important for both AD and MIS.

7.5.1 Language Support

The programming language has a strong influence on the way that modules can be designed.

- Turbo-Pascal provides **units** which can be used as modules. A unit has an "interface part" and an "implementation part" that provide separation of concern.
- C does not provide much for modularization. Conventional practice is to write a `.h` file for the interface of a module and a `.c` file for its implementation. Since these files are used by both programmers and compilers, the interfaces contain information about the implementation. For example, we can (and should) use `typedef`s to define types, but the `typedef` declarations must be visible to clients.
- Modula-2 (based on MESA, developed at Xerox PARC) provides **definition modules** (i.e. interfaces) and **implementation modules**. The important idea that Wirth got from PARC is that the interfaces can be compiled.
- Ada provides **packages** that are specifically intended for writing modular programs. Unfortunately, package interfaces and package bodies are separate, as in Modula-2.
- C++ does not provide any language features for structuring programs. The conventional way of using C++ is to write a header file and an implementation file for each class. A group of related classes may be contained in a single header/implementation pair of files.
- The modern approach is to have one physical file for the module and let the compiler extract the interfaces. This is the approach used in Eiffel and Java.

7.5.2 Examples of Modules

Example: a utility module for geometry.

Secret: Representations of geometric objects and algorithms for manipulating them.

Interface: Abstract data types such as *Point*, *Line*, *Circle*, etc. Functions such as these (which would be constructors in C++):

```
Line makeline (Point p1, p2)
Point makepoint (Line l1, l2)
Circle makecircle (Point c, float r)
```

Implementation: Representations for lines, circles, etc. (In C, these may be exposed in `.h` files. This is unfortunate but unavoidable.) Implementation of each function.

Example: a stack module.

Secret: How stack components are stored (array, list, etc).

Interface: Functions *Create*, *Push*, *Pop*, *Empty*, *Full*.

Implementation: For the array representation, *Full* returns *true* if there are no more array elements available. The list representation returns *false* always (assuming memory is not exhausted — but that is probably a more serious problem).

Example: a screen manager.

Secret: The relationship between stored and displayed data.

Invariant: The objects visible on the screen correspond to the stored data.

Interface:

Display:	add object to store and display it.
Delete:	erase object and remove it from store.
Hide:	erase object (but keep in store).
Reveal:	display a hidden object.

Implementation: An indexed container for objects (or pointers to objects) and functions to draw and erase objects.

7.5.3 A Recipe for Module Design

1. Decide on a secret.
2. Review implementation strategies to ensure feasibility.
3. Design the interface.
4. Review the interface. Is it too simple or too complex? Is it coherent?
5. Plan the implementation. E.g. choose representations.
6. Review the module.
   - Is it self-contained?
   - Does it use many other modules?
   - Can it accommodate likely changes?
   - Is it too large (consider splitting) or too small (consider merging)?

7.6 Design Notations

A design can be described by diagrams, by text, or (preferably) by both.

Diagrams are useful to provide an overview of the design, showing how the parts relate to one another. Text ("textual design notation", TDN) can be more precise, and as detailed as necessary, but it may not be easy to see the overall plan from a textual design.

Text and graphics are complementary, with graphics working at a higher level of abstraction. Some people favour elaborate graphical notations with thick, thin, and dashed lines; single, double, and crossed arrow heads; ovals, rectangles, and triangles; and so on. My opinion is that a graphical notation should be simple, reflecting the essential relationships between components, and the detail should be in the text.

There are many advantages in having a design notation that is sufficiently formal to be manipulated by software tools. Then we can use computers to help us design software.

The UPEDU uses notations adapted from the RUP. The diagrams include class diagrams, interaction diagrams, collaboration diagrams, and state diagrams.

7.7 Design Strategies

If you are a good designer, you can use any design strategy. If you are a bad designer, no strategy will help you.

We need a **strategy**, or **plan**, to develop the design. Strategies and architectures are related, in that a particular strategy will tend to lead to a particular architecture, but there is not a tight correspondence. For example, functional design tends to give a hierarchical architecture, but does not have to.

7.7.1 Functional Design

- Base the design on the **functions** of the system.
- Similar to writing a program by considering the procedures needed.
- A functional design is usually a hierarchy (perhaps with some layering) with "main" at the root.
- Compatible with "top-down design and stepwise refinement".

Strengths:

- Functional design works well for small problems with clearly-defined functionality.

Weaknesses:

- Leads to over-specialized leaf modules.
- Does not lead to reusable modules.
- Emphasis on **functionality** leads to poor handling of *data*. For example, data structures may be accessible throughout the system.

- Poor "information hiding" (cf. above).
- Control flow decisions are introduced early in design and are hard to change later.

7.7.2 Structured Analysis/Structured Design (SA/SD)

Structured Design/Structured Analysis (SA/SD) is a methodology for creating functional designs that was popular in the industry but is now somewhat outdated.

1. Formalize the design as a **Data Flow Diagram** (DFD). A DFD has terminators for input and output, data stores for local data, and transformers that operate on data. usually, terminators are squares, data stores are parallel lines, and transformers are round. These components are linked by labelled arrows showing the **data flows**.
2. Transform the DFD into a **Structure Chart** (SC). The SC is a hierarchical diagram that shows the modular structure of the program.

For example, Figure 4 shows a simple dataflow diagram and Figure 5 a corresponding structure chart. Unfortunately, the transformation from DFD to SC is informal, although guidelines exist.

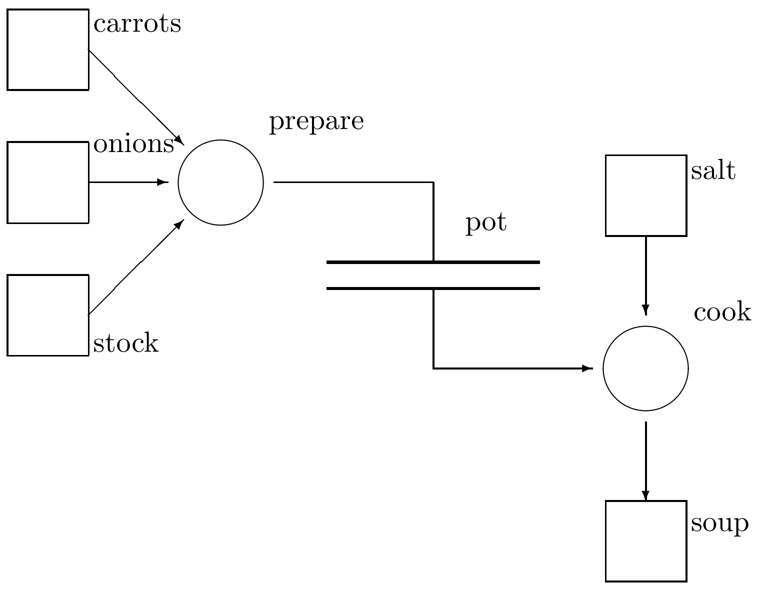

Figure 4: A Dataflow Diagram

7.7.3 Jackson Structured Design (JSD)

Jackson Structured Design (JSD) has been pioneered by Michael Jackson[1] JSD is data-driven — the software design is based on relationships between data entities — but is not (as some

[1] "He's the sort of person who thinks Michael Jackson is a singer and James Martin is a computer scientist." — Anon.

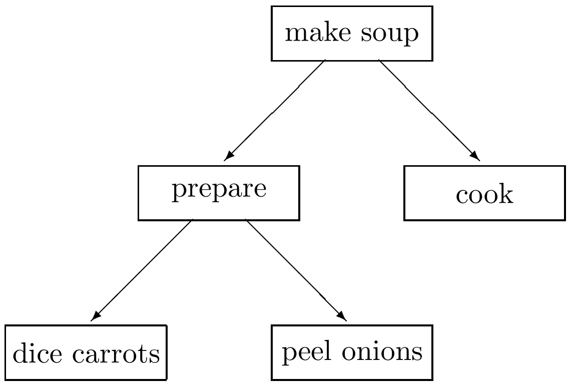

Figure 5: A Structure Chart

believe) object oriented.

7.7.4 Design by Data Abstraction

Data abstraction (i.e. abstract data types) historically preceded object oriented design (discussed next).

1. Choose data structures needed for the application.
2. For each data structure, design a module that hides the representation and provides appropriate functions.
3. Build layers using these modules.

Note that this is at least partly a **bottom-up** approach.
Strengths:

- Data representations are hidden inside modules.
- Control flow decisions are deferred until late in design.
- Data is less likely to change than functionality.
- Code is likely to be reusable.

Weaknesses:

- Must have a clear vision of the finished product, otherwise unnecessary or inappropriate data structures may be introduced.
- A module can either implement one instance of an ADT (restricted) or export a type (leads to awkward notation).

```
    module Stack
```

```
  exports StackType
  .....
end
.....
var S: StackType
begin
  Stack.Push(S, x)
  .....
```

7.7.5 Design by Use Cases

Some people advocate basing the design on use cases. The problem with this approach is that it may get the "boundaries" of the system (interactions between the system and its users) right but fail to do a good job of the "heart" of the system where the real work is done.

However, use cases can provide an effective means of testing a design: if the design cannot handle a use case, it has to be improved.

7.8 Object Oriented Design

It makes little sense to discuss object oriented design without first describing the basic ideas of object oriented programming. Hence the next section (which may be omitted by those who are already familiar with object oriented methodology).

7.8.1 Object Oriented Programming

Theory follows practice. Practitioners should listen to theorists, but only when the theory has matured. Examples: there is a mature theories for data structures and algorithms, but there is not yet a theory for object oriented programming.

The object oriented approach extends ADTs. A module in an OOL is called a **class**.[2] A class declaration contains declarations of **instance variables**, so-called because each instance of the class gets a copy of them. An instance of a class is called an **object**.

An **object** has:

- **state** (i.e. data);
- **identity** (e.g. address);
- **methods** (aka procedures and functions).

Equal states do not imply equal objects.[3] In real life, a person may be arrested because they have the same description and SSN as another person: this is an example of equal "states" but unequal "objects".

Methods are usually characterized as:

- **Constructors** create new objects.

[2] The term "class" dates from Simula67.
[3] Objects are intensional, not extensional.

- **Inspectors** returns values.
- **Mutators** change objects.

A stack object might have:

- Constructor: *Create*.
- Inspectors: *Empty*, *Full*, *Top*.
- Mutators: *Push*, *Pop*.

Classes in object oriented programming play the role of modules in procedural programming. In fact, a class is somewhat more restricted than a module: it is essentially a module that exports one type and some functions and procedures.

A **class**:

- is a collection of objects that satisfy the same protocol (or provide the same services);
- may have many instances (which are the objects).

All of this can be done with conventional techniques. OOP adds some new features.

Inheritance Suppose we have a class *Window* and we need a class *ScrollingWindow*. We could:

- rewrite the class *Window* from scratch;
- copy some code from *Window* and reuse it;
- **inherit** *Window* and implement only the new functionality.

```
class ScrollingWindow
  inherits Window
  — new code to handle scroll bars etc.
  — redefine methods of Window that no longer work
```

Inheritance is important because it is an **abstraction mechanism** that enables us to develop a **specialized class** from a **general class**.

Inheritance introduces a two new relationships between classes.

- The first relationship is called **is-a**. For example: "a scrolling window **is a** window."

  If X is-a Y it should be possible to replace Y by X in any sentence without losing the meaning of the sentence. Similarly, in a program, we should be able to replace an instance of Y by an instance of X, or perform an assignment $Y := X$.

  For example, a dog is an animal. We can replace "animal" by "dog" in any sentence, but not the other way round.[4] In a program, we could write $A := D$ where $A : \mathit{Animal}$ and $D : \mathit{Dog}$.

[4]Consider "Animals have legs" and "Dogs bark".

- The second relationship is called **inherits-from**. This simply means that we borrow some code from a class without specializing it. For example, *Stack* inherits-from *Array*: it is not the case that a stack is an array, but it is possible to use array operations to implement stacks.

Meyer combines both kinds of inheritance in a single example called "the marriage of convenience". A *STACK* Is an abstract stack type that provides the functions of a stack without implementing them. A *FIXED_STACK* is a stack implemented with an array. It inherits stack properties from *STACK* and the array implementation from *ARRAY*. Using the terms defined above, *FIXED_STACK* is-a *STACK* and *FIXED_STACK* inherits-from *ARRAY*.

The is-a relation should not be confused with the has-a relation. Given a class *Vehicle*, we could inherit *Car*, but from *Car* we should not inherit *Wheel*. In fact, a car **is a** vehicle and **has a** wheel.

There are various terminologies.

- *Window* is a **parent** and *ScrollingWindow* is a **child**. (An advantage of these terms is that we can use ancestor and descendant as terms for their transitive closures.)
- *Window* is a **superclass** and *ScrollingWindow* is a **subclass**.
- *Window* is a **base class** and *ScrollingWindow* is a **derived class**. (This is C++ terminology and is probably the most easily remembered.)

Organization of Object Oriented Programs Consider a compiler. The main data structure in a compiler is the **abstract syntax tree**; it contains all of the relevant information about the source program in a DAG. The DAG has various kinds of node and, for each kind of node, there are several operations to be performed.

	ConstDecl	VarDecl	ProcDecl	Assign	ForLoop
Construct					
Check					
Print					
Optimize					
Generate					

Corresponding to each box in the diagram, we must write some code. In a functional design, the code would be organized by **rows**. For example, there would be a procedure *Print*, consisting of a `switch` (or `case`) statement, with a case for each column.

In an object oriented program, the code would be arranged by **columns**. We would declare an (abstract) class *ASTNode* and each of the columns would correspond to a class inheriting from *ASTNode*. The class would contain code for each of the operations.

The language mechanism that enables us to do this is *dynamic binding*. We can do the following:

```
AstNode Node;
.....
Node := ForNode;
Node.Print
```

- The assignment statement is legal because we are allowed to assign a value of a derived class to a variable of the base class. In general, we can write $x := y$ if y is-a x, but not the other way round.
- The statement *Node.Print* is legal because *AstNode* and all its derived classes provide the method *Print*. Note that the compiler cannot determine the class (type) of *Node*.
- At run-time, the program must choose the appropriate code for *Print* based on the class of *Node*. A simple implementation uses a pointer from the object to a class descriptor consisting of an array of pointers to functions.

With a functional design, it would be easy to add a row to the table: this corresponds to writing one function with a case for each node type. But it is unlikely that we would want to do this because the functional requirements of a compiler are relatively static. It is harder to add a column to the table, because this means **altering** the code in many different places.

With the object oriented design, it is easy to add a column to the table, because we just have to declare a new derived class of *Node*. On the other hand, it is hard to add a row, because this would mean adding a new method to each derived class of *Node*.

On balance, the pros and cons favour the object oriented approach. All changes are **additions**, rather than changes, and the expected changes (adding to the varieties of a data structure) are easier to make.

Dynamic Binding Consider these classes:

```
class Animal
  method Sound .....
class Dog
  inherits Animal
  method Sound
    return "bark"
class Cat
  inherits Animal
  method Sound
    return "miaow"
```

and assume

```
A : Animal
C : Cat
D : Dog
```

The assignments $A := C$ and $D := C$ are allowed, because a cat is-a animal and a dog is-a animal. We can also write

```
if P then A := C else D := C
A.Sound
```

The compiler cannot determine which version of *Sound* to call, because the correct method depends on the value of the predicate P. Consequently, it must generate code to call either *Sound* in class *Cat* (if P is true) or *Sound* in class *Dog* (if P is false). The "binding" between the name "*Sound*" and the code is established at run-time. We say that object oriented languages use **dynamic binding** for method calls. Conventional languages use **static binding** — the function name and code are matched at compile-time.

Contracts Meyer introduced the expression "programming by contract". A method has **requires** and **ensures** clauses and is not required to do anything at all if its **requires** clause is not satisfied on entry. The contractual approach reduces redundant code because the implementor is not required to check preconditions. For example, a *squareroot* function that requires a positive argument does not have to check the value of its argument.

Meyer also explained how contracts could be used to ensure correct behaviour in subclasses. Consider the following classes.

```
class PetShop
  method Order
    requires:  Payment ≥ $10
    ensures:  animal delivered

class DogShop
  inherits PetShop
  method Order
    requires:  Payment ≥ $5
    ensures:  dog delivered
```

The **ensures** clause is *strengthened*: delivering a dog is a stronger requirement than delivering an animal. (To see this, note that if you asked for an animal and received a dog, you would have no grounds for complaint. But if you asked for a dog and received some other kind of animal, you would complain.)

Surprisingly, the **requires** clause is *weakened*: the dog shop requires only $5. To see why, consider the following code.

```
P : PetShop
D : DogShop
A : Animal
.....
P := D
A := P.Order(12)
```

The assignment $P := D$ is valid because *DogShop* inherits from *PetSHop*. The result of P . *Order*(10) will be a dog, and we can assign this dog to A.

The payment for the dog, $12, must satisfy the **requires** clause of *Order* in *PetShop*, because P is a *PetShop*, and it does so ($12 \geq 10$). It must also satisfy the **requires** clause of *Order* in *DogShop*, because at run-time P is actually a *DogShop*.

Note that if we had strengthened the **requires** clause of *Order* in *DogShop* as in

```
    method Order
      requires:  Payment ≥ $15
      ensures:  dog delivered
```

then the statement $A := P\,.\,Order(12)$ would have compiled but have failed at run-time. In summary, the rule that we *weaken* the **requires** clause and *strengthen* the **ensures** clause in an inherited class has the desired effect: the static meaning of the program is consistent with its run-time effect.

Frameworks An **abstract class** defines a pattern of behaviour but not the precise detailed of that behaviour: for example, see *AstNode* above. A collection of abstract classes can provide an architecture for a family of programs. Such a collection is called a **framework**.

We can use predefined functions, together with functions that we write ourselves, to construct a library. The main program obtains basic services by calling functions from the library, as shown in Figure 6. The structure of the system is determined by the main program; the library is a collection of individual components.

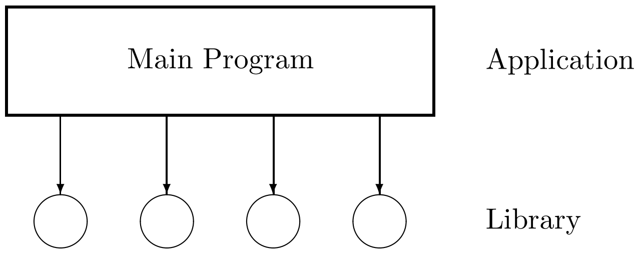

Figure 6: A Program Using Components from a Library

Using object oriented techniques, we can reverse the relationship between the library and the main program. Instead of the application calling the library, the library calls the application. A library of this kind is referred to as a **framework**. A framework is a coherent collection of classes that contain deferred methods. The deferred methods are "holes" that we fill in with methods that are specific to the application. The organization of a program constructed from a framework is shown in Figure 7. The structure of the system is determined by the framework; the application is a collection of individual components.

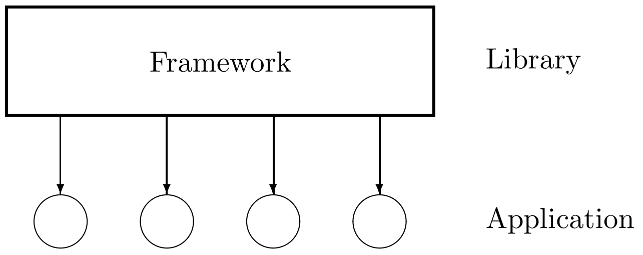

Figure 7: A Program Using a Framework from a Library

The best-known framework is the MVC triad of Smalltalk. It is useful for simulations and other applications. The components of the triad are a model, a view and a controller — see

Figure 8.

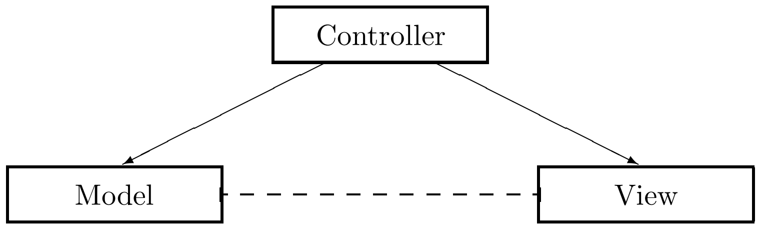

Figure 8: The Model-View-Controller Framework

The Model is the entity being simulated. It must respond to messages such as *step* (perform one step of the simulation) and *show* (reveal various aspects of its internal data).

The View provides one or more ways of presenting the data to the user. View classes might include *DigitalView*, *GraphicalView*, *BarChartView*, etc.

The Controller coordinates the actions of the model and the view by sending appropriate messages to them. Roughly, it will update the model and then display the new view.

Smalltalk provides many classes in each category and gives default implementations of their methods. To write a simulation, all you have to do is fill in the gaps.

A framework is a kind of upside-down library. The framework calls the application classes, rather than the application calling library functions. Frameworks are important because they provide a mechanism for **design re-use**.

7.8.2 Responsibility-Driven Design

Wirfs-Brock *et al* suggest that the best way to find good objects is to consider the **responsibility** of each object. The following summary of their method is adapted from Wirfs-Brock. The ideas here are useful for other design methodologies, as well as OOD.

Classes

1. Make a list of noun phrases in the requirements document. Some nouns may be hidden (for example, by passive usage).
2. Identify candidate classes from the noun phrases using the following guidelines.
    - Model physical objects.
    - Model conceptual entities.
    - Use a single term for each concept.
    - Avoid adjectives. (A class *HairyDog* is probably a mistake: it's *Dog* with a particular value, or perhaps derived from *Dog*.)
    - Model: categories of objects, external interfaces, attributes of an object.
3. Group classes with common attributes to produce candidates for abstract base classes.

4. Use categories to look for classes that may be missing.
5. Write a short statement describing the purpose of each class.

Responsibilities

1. Find responsibilities:
    - Recall the purpose of each class.
    - Extract responsibilities from the requirements.
    - Identify responsibilities from relations between classes.
2. Assign responsibilities to classes:
    - Evenly distribute system intelligence.
    - State responsibilities as generally as possible.
    - Keep behaviour with related information.
    - Keep information about one thing in one place.
    - Share responsibilities among related classes.
3. Find additional responsibilities by looking for relationships between classes.
    - Use "is-kind-of" to find inheritance.
    - Use "is-analogous-to" to find base classes.
    - Use "is-part-of" (or "has-a") to find suppliers and clients.

Collaborations

1. Find collaborations by examining the responsibilities associated with classes. Examples: Which classes does this class need to fulfill its responsibilities? Who needs to use the responsibilities assigned to this class?
2. Identify additional collaborations by examining relations between classes. Consider "is-part-of", "has-knowledge-of", and "depends-on".
3. Discard classes that have no collaborations.

The process continues by building class hierarchies from the relationships already established. Wirfs-Brock *et al* also provide conventions for drawing diagrams.

7.9 Functional or Object Oriented?

7.9.1 Functional Design (FD)

- FD is essentially top-down. It emphasizes control flow and tends to neglect data.
- Information hiding is essentially bottom-up. It emphasizes encapsulation ("secrets") in low-level modules.

- In practice, FD must use both top-down and bottom-up techniques.
- FD uses function libraries.
- FD is suitable for small and one-off projects.

7.9.2 Object Oriented Design (OOD)

- OOD is also top-down and bottom-up, but puts greater emphasis on the bottom-up direction, because we must define useful, self-contained classes.
- OOD emphasizes data and says little about control flow. In fact, the control flow emerges implicitly at quite a late stage in the design.
- Inheritance, used carefully, provides separation of concern, modularity, abstraction, anticipation of change, generality, and incrementality.
- OOD is suitable for large projects and multi-version programs.

7.10 Writing MIS and IMD

Design is not implementation. Module interface specifications (MIS) should be at a higher level of abstraction than internal module designs (IMD), and IMD should be at a higher level than code. One way of achieving a higher level is to write **declarative** ("what") statements rather than **procedural** ("how") statements.

7.10.1 Module Interface Specifications

Use requires/ensures where possible, otherwise clear but informal natural language (e.g. English).

You can use pre/post instead of requires/ensures. The advantage of requires/ensures is that these verbs suggest the use of the predicates more clearly than the labels pre and post.

An MIS often uses some of the syntax of the target language. This helps the implementors. Pascal style:

```
function Int_String (N : integer) : String
  ensures:  result is the shortest string representing N
```

C style:

```
int String_Int (char * s)
  requires:  s consists of optional sign followed by decimal digits
  ensures:  result is the integer represented by s
```

When there are several cases, it may help to use several pairs of requires/ensures classes. Rather than

```
void Line (Point p, Point q)
  ensures:  if p and q are in the workspace
      then a line joining them is displayed
      else nothing happens.
```

write this:

```
void Line (Point p, Point q)
  requires: p and q are in the workspace
  ensures: a line joining p and q is displayed

  requires: either p or q is outside the workspace
  ensures: nothing
```

In the general case, a specification of the form

```
requires: nothing
ensures: if $P$ then $X$ else $Y$
```

can be more clearly expressed in the form

```
requires: $P$
ensures: $X$
requires: $\neg P$
ensures: $Y$
```

7.10.2 Internal Module Designs

Try to describe the implementation of the function without resorting to implementation details. Here are some guidelines.

- Loops that process all components can be expressed using universal quantifiers.

  Rather than

  ```
  $p =$ objectlist
  while ($p$) do
    draw($p \rightarrow$ first)
    $p = p \rightarrow$ next
  ```

  write

  ```
  for each object in objectlist do
    draw(object)
  ```

  or even

  ```
  $\forall o \in$ objectlist
    draw($o$)
  ```

- Loop statements that search for something can be expressed using existential quantifiers.

  Rather than

  ```
  $N := 0$
  Found := false
  while $N <$ Max and not Found do
    if $P(N)$
      then Found := true
      else $N := N + 1$
  if Found
    then $F(N)$
  ```

  write

```
if there is an $N$ such that $0 \le N < Max$ and $P(N)$
  then $F(N)$
```

or even

```
if $\exists N \,.\, 0 \le N < Max \wedge P(N)$
  then $F(N)$
```

- Conditional statements can be replaced by functions.

  Rather than

  ```
  if $x < xMin$ then $x := xMin$
  ```

  write

  ```
  $x := max(x, xMin)$
  ```

- For mouse selection, define the concept of a **bounding box**. A bounding box is denoted by a 4-tuple (x_l, y_t, x_r, y_b) where (x_l, y_t) is the top-left corner and (x_r, y_b) is the bottom-right corner. Then we can write

  ```
  case MousePosition in
    $(0, 0, L-1, H-1) \Longrightarrow$ .....
    $(0, H, L-1, 2H-1) \Longrightarrow$ .....
    $(0, 2H, L-1, 3H-1) \Longrightarrow$ .....
    .....
  ```

- Conditional statements can be replaced by (generalized) case statements.

  Rather than

  ```
  if $x < 0$
    then NegAction
  else if $x = 0$
    then ZeroAction
  else PosAction
  ```

  write

  ```
  case
    $x < 0 \Longrightarrow$ NegAction
    $x = 0 \Longrightarrow$ ZeroAction
    $x > 0 \Longrightarrow$ PosAction
  ```

We can use these guidelines to improve existing designs. Here is part of the specification of *GetnextObject* from a specification.

```
if NID ≠ NIL then
  if NID.next ≠ NIL then
    NID := NID.next
    if NID.dirty then
      GetNextObject := NID.next
      else GetNextObject := NID
  else if Head.dirty then
    GetnextObject := NIL
    else GetNextObject := Head.next
```

Problems:

- The pseudocode is is very procedural and constrains the implementor unnecessarily.
- The function has a bad specification. We can guess the reason for the special case `NID = NIL`, but it produces a murky function.
- The use of "dirty bit" is incorrect. Conventionally, a dirty bit is set when a structure is changed, not when it is deleted. A better term would be "deleted bit".
- The code appears to be incorrect. What happens if there are two consecutive deleted components?

An NL description of *GetnextObject*:

> Return the first non-deleted object after *NID* in the object list, or `NIL` if there is no such object. If *NID* = `NIL` on entry, return the first such object.

Here is another part of the specification, for *SearchByMarkers*:

```
found := false
if head.dirty or head.next = NIL then
  SearchByMarkers := NIL
else
  p := head.next
  while p ≠ NIL and not found do
    if not p.dirty then
      found := InRange(p.obj, mx, my)
    if not found then
      p := p.next
  if found then SearchByMarkers := p
  else SearchByMarkers := NIL
```

Pseudocode for *SearchByMarkers*:

```
if there is a $p$ such that $\neg p.dirty$ and
    $InRange(p \rightarrow obj, Mx, My)$
  then return $p$
  else return NIL
```

A more concise description:

```
if $\exists p \,.\, \neg p.dirty \wedge$
    $InRange(p \rightarrow obj, Mx, My)$
  then return $p$
  else return NIL
```

8 Object Oriented Development

This section is adapted from notes made from an earlier version of a software engineering course. It mentions process models such as Fusion that are predecessors of RUP.

Object oriented methods started off with programming languages such as Simula (1967) and Smalltalk (1976) and gradually crept backwards through the life cycle: OO programming require OO design, which in turn required OO analysis. A variety of methods have been proposed for object oriented software development, including:

- Object Modeling Technique (OMT).
- Booch's method
- Objectory
- Class Responsibility Collaborator (CRC).
- Fusion.
- RUP/UPEDU.

The Fusion method,[5] developed at Hewlett-Packard (UK) is the most recent method and it is, to some extent, based on earlier methods. We describe it here. The Fusion model is presented in "waterfall" form but it can be adapted for a spiral (risk-driven) approach or for incremental development.

The major phases of the Fusion method (as for other methods) are as follows. Note that all phases produce **models** of the system.

Analysis produces a specification of what the system does, based on a requrements document. The result of analysis is a **specification document**.

Analysis models describe:

- Classes that exist in the system
- Relationships between the classes
- Operations that can be performed on the system
- Allowable sequences of operations

Design starts from the specification document and determines how to obtain the system behaviour from the given requirements. The result is an **architecture document**.

Design models describe:

- how system operations are implemented by interacting objects
- how classes refer to one another and how they are related by inheritance
- attributes and operations for each class

[5]Fusion seems to have evolved into ColdFusion, a process for developing fancy websites.

Implementation starts from the architecture document and encodes the design in a programming language. The result is **source code**.

Design features are mapped to code as follows:

- inheritance, references, and attributes are implemented using corresponding features of the target language (specifically, class definitions)
- object interactions are implemented as methods in the appropriate classes
- state machines (usually not explicitly coded) define permissible sequences of operations

A **data dictionary** is maintained for all stages of development. Figure 9 shows part of a data dictionary. The data dictoinary is called a ***glossary*** in UPEDU.

customer	agent	delivers gas using the gun and makes payment
timer	agent	turns off the pump motor a certain time after the pump has been disabled
enable-pump	sys op	enables the pump to start pumping gas unless the pump is out of service
remove-gun	sys op	enables the gun to start pumping gas when the trigger is depressed
start-timer	event	starts the timer for turning off the pump motor
display-amount	event	details of the current delivery are shown on the display

Figure 9: Extract from a Data Dictionary

8.1 Analysis

The main purpose of analysis is to identify the objects of the system. Analysis yields an **object model** and an **interface model**.

Important features of objects include:

- An object can have one or more **attributes** associated with it. Each attribute is a value chosen from a basic type such as *Boolean*, *integer*, *float*, or *string*.

  (Note: the Fusion method does not allow attributes to be objects *during the analysis phase.*)

  Example: a **Person** object might have attributes name, address, SIN, ...

- The **number** of attributes, and the **name** of each attribute, is fixed (for a given object).

- The **value** of an attribute can change during the lifetime of the object.

- An object can be **identified** (or: "has identity"). An object may be a specific person, organization, machine, event, document, ...

  If X and Y are objects, we distinguish "X and Y are the same object" from "X and Y have equal attributes". This is like real life: two people can have the same name and yet be distinct persons.

- In the analysis phase, we do not consider the operations (or methods) associated with an object. Operations are considered during design.

Objects are grouped into sets, called **Classes**. The instances of a class have the same attributes, but the value of their attributes are not necessarily the same. We represent classes by giving a **class name** and listing the attributes, as in UPEDU.

The ***extension*** of a class is the set of objects that inhabit it. The left box below shows a class declaration; the right box shows a class extension: the instances of the class are Anne, Bill, and Chris.

Instructor
Name
Subject

Instructor	
Name	Subject
Anne	Chemistry
Bill	English
Chris	Math

An object may be **associated** with one or more other objects. For example, instructors may be associated with courses. Corresponding to an association between objects, we define a **relationship** between classes. (These distinctions seems to be blurred in UPEDU.)

Example: the relationship *teaches* exists between the class *Instructor* and the class *Course*. The extension of the relationship is a set of pairs, such as

$$\{(Anne, CHEM201), (Bill, ENGL342)\}.$$

Relationships may have **cardinality contraints** associated with them. In general, a relationship may be one-to-many ($1 : N$), many-to-one ($N : 1$), or many-to-many ($M : N$).

Example: an instructor may teach several courses, but each course has only one instructor. Thus the relation *teaches* is one-to-many. The relation *takes* (a student *takes* a course) is many-to-many.

The classes participating in a relationship may have **roles**. (If we draw a diagram for a relationship, with nodes indicating the participating classes and the relationship, the roles are edge labels.)

Example: *child of* is a relationship between the class *Person* and *Person*. We can add useful information to this relationship by introducing the roles *parent* and *child* (a parent may have many children; each child has exactly two parents).

Relationships may have **attributes**.

Example: consider the relationship *takes* between class *Student* and class *Test*. We would like to associate a mark with each (*Student*, *Test*) pair. The mark cannot be an attribute of *Student*, since a student may take many exams, nor can it be an attribute of *Test*, since many students take a given exam. Consequently, we must associate the mark with the relationship *takes* itself.

Relationships may involve more than two classes.

Example: we could extend the relation *takes* to include class *Room*. An element of this relationship would be a triple $(\text{Jill}, \text{midterm}, H441)$.

Another way of combining classes is to put one or more classes into an **aggregate** class.

Example: we can build an aggregate class *Option* as follows. *Option* has:

- an attribute *name* (e.g. `"Software Systems"`).
- component classes *Student* and *Course* linked by *takes* (the only students in an option are those who take the courses appropriate for the option).

Generalization allows us to abstract the common properties of several **subclasses** (or **derived classes**) into a **superclass** (or **base class**).

Eaxmple: suppose that there are undergraduate courses and graduate courses. Then the class *Course* is a generalization of the two kinds of course. That is, we could make *Course* a superclass with the subclasses *GradCourse* and *UGradCourse*.

Once we have generalized, we can move the common attributes of the subclasses into the superclass.

Example: since all courses have attributes *time* and *room*, these could be attributes of class *Course*. But only undergraduate courses have tutorials, so we could not make *tutorial* an attribute of *Course*.

Multiple generalization is also allowed.

Example: → reads "specializes to".

$$\begin{array}{ll}
\mathit{Instructor} & \rightarrow \mathit{FullTimeInstructor} \\
\mathit{Instructor} & \rightarrow \mathit{PartTimeInstructor} \\
\mathit{PartTimeInstructor} & \rightarrow \mathit{GradInstr} \\
\mathit{Student} & \rightarrow \mathit{GraduateStudent} \\
\mathit{GraduateStudent} & \rightarrow \mathit{GradInstr}
\end{array}$$

Note that generalization and its inverse, specialization, correspond to "inheritance" in OO programming languages.

The analysis phase yields two models: the System Object Model and and Interface Model.

8.1.1 System Object Model

The collection of classes will include classes that belong to the environment and classes that belong to the system. The **System Object Model** (SOM) excludes the environment classes and focuses on the system classes. The SOM describes the **structure** of the system but not its behaviour.

8.1.2 Interface Model

The system can also be modelled as a collection of interacting **agents**. One agent is the system itself; the other agents are "things in the world", such as users, hardware devices, and so on. Agents communicate by means of **events**. We are interested primarily in communication with the system, although it may be necessary during analysis to consider communication between agents other than the system.

An **input event** is an event sent by an external agent to the system. An **output event** is an event sent by the system to an external agent.

Events are considered to be instantaneous, atomic, and asynchronous (the sender does not wait for an acknowledgement that the event has been received).

Input events lead to **state changes** in the system, possibly leading to output events. An input event and its effect on the system is called a **system operation**. Only one system operation can take place at a given time.

The **Interface Model** (IM) of the system is the set of system operations and the set of output events.

It is important to note that the IM, as described here, is a *black box* model: the only events that we can observe at this level are those that link the system to its environment. Activity inside the system is hidden. However, we can place restrictions on the system's behaviour. For example, we can say that an event of type B must always be preceded by an event of type A.

The IM can be further subdivided into the operation model and the life-cycle model.

The **Operation Model** (OM) describes the effect of each input event on the state of the system. The OM consists of a set of *schemas* ("schemata" for classical scholars). A schema corresponds to a use case in UPEDU. Figure 10 shows an example of a schema.

Operation:	*replace-gun*.
Description:	Disables the gun and the pump so no more gas can be dispensed; creates a new transaction.
Reads:	Display.
Changes:	`supplied` *pump*, `supplied` *gun-status*, `supplied` *holster-switch*, `supplied` *timer*, `supplied` *terminal*, `new` *transaction*.
Sends:	*timer*: {start message}, *terminal*: {transaction}.
Assumes:	*holster-swtich* is released.
Result:	*holster-switch* is depressed. *gun* is disabled. If the *pump* was initially enabled, then: The *pump-status* is disabled; a *transaction* has been enabled with the value of the *pump* display and sent to the terminal; and a start message has been sent to the *timer*. else: there is no effect.

Figure 10: A Fusion Schema

The **Life-cycle Model** (LM) describes the allowable sequences of system operations.

Example: consider a vending-machine that accepts two quarters and emits a chocolate bar. The input events are *input quarter* (iq) and *press button* (pb). The output events are *reject* *coin* (rc) and *emit chocolate* (ec). The following sequences are allowed:

- (Regular sequence) iq, iq, pb, ec.
- (Reject coin) iq, rc, iq, iq, pb, ec.

Note that we need a more complex formalism (e.g. a finite state machine) to describe all sequences with a finite formula. Figure 11 shows the syntax provided by Fusion. Using this syntax, we can describe the lifecycle of the vending-machine as

$$((\mathsf{iq} \bullet \mathsf{rc})^* \bullet \mathsf{iq} \bullet (\mathsf{iq} \bullet \mathsf{rc})^* \bullet \mathsf{iq} \bullet \mathsf{pb} \bullet \mathsf{ec})^*$$

$Expr$	$\rightarrow$	*input-event*	
	$\mid$	*#output-event*	
	$\mid$	$x \bullet y$	x followed by y
	$\mid$	x^*	0 or more xs
	$\mid$	x^+	1 or more xs
	$\mid$	$[x]$	0 or 1 x
	$\mid$	$x \| y$	x and y concurrently
	$\mid$	(x)	precedence

Figure 11: Syntax for Lifecycle Models

Here is another lifecycle, from a gas-station example:

`lifecycle` *GasStation* : ((*delivery* • *payment*)* | *out-of-service*)* • *archive*

The item *delivery* has complex behaviour that is expanded in Figure 12.

$$\begin{array}{lll}
\textit{delivery} & = & \textit{enable-pump}\bullet \\
& & [\textit{start-pump-motor}]\bullet \\
& & [\ (\textit{preset-volume} \mid \textit{preset-amount})\ ] \\
& & \textit{remove-gun-from-holster}\bullet \\
& & \textit{press-trigger}\bullet \\
& & (\textit{pulse} \bullet \#\textit{display-amount})^*\bullet \\
& & (\textit{release-trigger} \mid \textit{cut-off-supply})\bullet \\
& & \textit{replace-gun}\bullet \\
& & \#\textit{start-timer}\bullet \\
& & [\textit{turn-off-motor}]
\end{array}$$

Figure 12: Expansion of *delivery* lifecycle

Object oriented analysis has been criticized because it is too implementation oriented: the critics say that objects should not be introduced so early in development. The Fusion method avoids this problem:

- Fusion uses a highly abstract form of "object" in the analysis phase.
- All objects are (or can be) considered, not only those that will become part of the final system.
- Objects that contain other objects are not allowed.
- Methods are not assigned to objects.
- Generalization and specialization are used, but not inheritance.

Thus it should be possible to complete analysis without too much implementation bias.

8.2 Design

The analysis models define a system as a collection of objects that has a specified response to particular operations. The purpose of the design phase is to find a strategy for implementing the specification. The output of the design phase consists of four parts.

- **Object Interaction Graphs** describe how objects interact at run-time.
- **Visibility Graphs** describe object communication paths.
- **Class Descriptions** describe class interfaces.
- **Inheritance Graphs** describe inheritance relationships between classes.

8.2.1 Object Interaction Graphs

Each operation of the system is described by an object interaction graph (OIG). The graph is a collection of boxes linked by arrows. The boxes represent *design objects* and the arrows represent *message passing.*

A **design object** is an extended version of an analysis object. In addition to attributes, a design object has a *method interface.* The arrows that enter a design object in an OIG are labelled with method names and arguments.

When a message is sent to a design object, the corresponding method is assumed to be invoked. The method completes its task before returning control to the caller. Task completion may require sending messages to other objects.

Invoking the method *Create* of an object creates a new instance of the object.

A design object with a dashed outline represents a collection of objects of the same class. The Fusion method provides conventions for sending a message to a collection: for example, it is possible to send a message to every member of a collection and to iterate through the members of a collection.

OIGs are developed as follows.

1. Identify the objects involved in a system operation. The `reads` and `changes` clause of the schemas identify the objects that are touched or modified by an operation.
2. Establish the role of each object.
   - Identify the object that initiates the operation (this object is called the **controller**.
   - Identify the other objects (these are called the **collaborators**). The `sends` clause of a schema identifes messages sent by the object.
3. Choose appropriate messages between objects.
4. Record how the objects interact on an OIG.

After the OIGs have been constructed, they can be checked.

- Each object in the SOM should appear in at least one OIG. (Otherwise the object is apparently not required: there is something wrong with the SOM or with the design.)
- The effect of each OIG should agree with the operations described in the OM.

8.2.2 Visibility Graphs

If an OIG requires an object X to communicate with an object Y, then X must contain a reference to Y. Visibility graphs (VGs) describe the references that objects need to participate in operations.

A VG consists of **visibility arrows**; **client boxes** (which have at least one visibility arrow emerging from them); and **server boxes** (which have at least one arrow entering them).

A server is **bound** to a client if the client has exclusive access to the server at all times. In this case, the server is drawn as a box inside the client box. A server that is not bound is said to be **unbound**. Obviously, an unbound server cannot be shared.

Several factors must be taken into account while constructing the visibility graph:

- If X needs to access Y only when its method m is invoked, a reference to Y can be passed as an argument of m. This is called a *dynamic reference.*
- If X needs to access Y many times, the reference should be created when X is created (it could, for example, be an argument of the constructor). This is called a *permanent reference.*
- A server may have a single client or many clients. Shared servers have a single border in the OIG; exclusive servers have a double border.
- References may be *constant* (they never change) or *variable* (they may change at run-time).

8.2.3 Class Descriptions

Fusion class descriptions are conventional. Inheritance is indicated by an **isa** clause; attributes are listed; and methods are listed. Figure 13 shows an example of a class description.

```
class Pump
  attribute pump-id:  integer
  attribute status; pump-status
  attribute constant terminal:  Terminal
  attribute constant clutch:  Clutch
  attribute constant motor:  exclusive bound Motor
  attribute constant timer:  Timer
  method enable
  method disable
  method is-enabled:  Boolean
  method delivery-complete
endclass
```

Figure 13: A Fusion Class Description

8.2.4 Inheritance Graphs

The analysis phase may have yielded examples of specialization and generalization. During design, these can be examined to see if they are candidates for implementation by inheritance.

There is not necessarily a precise match between specialization (a problem domain concept) and inheritance (a programming language concept). For example, it may not be helpful to implement an **isa** relationship using inheritance. Conversely, it may be possible to use inheritance (as a code reuse mechanism) even when specialization does not occur in the design models.

8.2.5 Principles of Good Design

The following ideas may be helpful for designing classes. They all sound good, but some of them can be harder to apply than appears at first.

- Minimize object interactions
- Cleanly separate functionality
  1. a simple function should not be spread across several objects
  2. a single object should not provide many functions
- Look for subsystems — groups of objects that form coherent units
- Restrict reuse based on inheritance — often composition is better
- Develop shallow inheritance graphs[6]
- The root of an inheritance tree should be an abstract class[7]
- Grade levels of inheritance — ensure that the siblings at each level of the inheritance hierarchy should differ in the same way

[6]Some authors advocate deep inheritance graphs! The best rule is probably to match the opportunities for inheritance to the application.

[7]Some authors say that a concrete class should not have descendants.

- Use inheritance for subtyping — subclasses should always respect the properties of their superclass(es)

8.3 Implementation

The main goals of implementation are **correctness** and **economy** (the software should not make excessive use of time or space). Fusion is not associated with any particular programming language, and implementation details will vary between languages.

Implementation is divided into three parts: **coding**, **performance**, and **testing**. These parts could be carried out sequentially but, in practice, it is best to interleave them.

8.3.1 Coding

Lifecycle models are translated into finite state machines which are implemented in the programming language. This is straightforward, apart from interleaving ($x \| y$).

Class descriptions from the design phase can easily be mapped into class declarations of most OO languages. **Bound** objects can be embedded; **unbound** objects can be represented as pointers. If the language does not provide embedded objects (e.g. Smalltalk) then all objects are represented as pointers. Constant attributes are initialized when the object is created but cannot be assigned.

8.3.2 Performance

Good performance is not achieved by a final "tuning" step but must be taken into account throughout implementation. A **profiling tool** should be used if possible, and effort should be devoted to improving "hot spots" (portions of code that are executed very often).

- performance cannot be obtained as an afterthought
- optimizing rarely executed code is not effective
- profile the system in as many ways as you can

Consider language features such as

- inlined methods
  1. don't inline large functions (C++ does not allow this)
  2. excessive inlining will lead to "bloat" which may actually worsen performance on a paged system
  3. when a class with inlines is changed, *all clients* must be recompiled
- using references rather than values (in C++, passing an object by value requires the creation and destruction of an object)
- avoiding virtual methods (C++ only: actually the overhead of virtual methods is not very great)

Note that, in C++, dynamic binding requires pointers to objects but operator overloading requires objects, not pointers. It is therefore difficult to build a "clean" system which uses objects or pointers consistently.

8.3.3 Review

All code should be **reviewed**. In Fusion terminology, *inspection* require that code be read and understood by people other than the authors. *Testing* checks the actual behaviour of the system.

It is hard to understand programs and particularly hard to understand OO programs. Since methods are often small (2–5 lines average) tracing even a simple system operation can be lengthy and tedious. With dynamic binding, it can be effectively impossible.

Reviewers should pay particular attention to tricky language features, such as casts.

Code can be tested at different levels in an OO system:

Function: test individual methods.

Class: test invariants and check outputs for allowed sequences of method invocations.

Cluster: test groups of collaborating objects.

System: test entire system with external inputs and outputs.

8.3.4 Error Handling

In Fusion, an error is defined as *the violation of a precondition*[8]. Fusion distinguishes the *detection* of errors and *recovery* from errors.

Error Detection We could detect errors by writing code to test the precondition in every method, but this is not a good idea:

- It may be difficult, inefficient, or impossible to express the precondition in the programming language. (Consider a matrix inversion function with precondition "$|M| \neq 0$" or a graph algorithm with the precondition "G is acyclic".) Preconditions may refer to global properties of the system that cannot be checked within the method.
- Some preconditions are actually *class invariants* that should be true before and after every method of the class is executed.

Sometimes preconditions are included during testing but omitted (e.g. by conditional compilation) in production versions.[9]

Error Recovery The golden rule is: *don't check for an error whose presence you cannot handle properly.*

The best way to recover from errors is to use an *exception handling mechanism.* Exceptions usually require language support. (Recent versions of C++ have exception handling using `try` and `catch`.)

[8]This is not a good definition: it seems preferable to use pre/post-conditions to define the logical properties of the system, not its practical properties.

[9]This has been compared to wearing your parachute during ground practice but removing it when you fly, but the analogy is weak.

8.3.5 General

The complete Fusion methodology includes detailed descriptions of how to use constants, pointers, references, virtual methods, inlines, and so on.

It is very hard to perform memory management in a large OO system. Some languages provide garbage collectors; otherwise you're on your own. To see the difficulty, consider a class *Matrix* and the expression $f(x \times y)$ where x and y are instances of *Matrix*. The evaluation of $x \times y$ will yield an object. If the object is on the stack, it will be deallocated at block exit: no problem. But we might prefer to work with pointers to save time and space. Then $x \times y$ can be deallocated neither by the caller nor the callee!

8.4 Reuse

Although "reuse" is a current hot topic, we have been reusing software in many way all along:

- repeated execution of a program with different inputs;
- repeated invocations of a function with different arguments;
- reuse of people's knowledge;
- code and design scavenging;
- importing of library routines;

The object oriented approach adds to these by providing reusable *classes* (really just a "large" library routines) and *frameworks* (reusable designs). The following specific features of the OO approach enhance reuse.

- Classes that correspond to real-world entities may be used in several applications, provided that the appropriate abstractions are made.

  This kind of reuse does not always work. It is unlikely that a single class *Car* would be suitable for both a traffic simulation and a car factory.
- Encapsulation helps reuse by minimizing clients' exposure to implementation changes and by focusing attention on (hopefully stable) interfaces.
- Inheritance supports reuse because it allows classes to be "changed but not changed". When we inherit from a class, *we do not change the original class*, which can continue to be used in one or more applications.
- Polymorphism allows new components to be introduced into a system without changing the calling environment.
- Parameterization allows one class to play many roles. It is an older, and often more effective technique, than inheritance.

Unfortunately, inheritance can also be an *impediment* to reuse, especially if it is "fine-grain" inheritance (in which only one or two features are added at each level of the inheritance hierarchy). Heavy use of inheritance, and deep hierarchies, can make software hard to understand.[10]

[10] My view is that this problem can be largely overcome by the use of suitable programming languages and development tools.

Another problem is that there are two quite different kinds of inheritance: subtype inheritance and reuse inheritance. These do not always cooperate and sometimes work in opposition to one another.

Developers should be aware of the potential for reuse during all phases of software development: analysis, design, and implementation. Each of the artefacts created is a candidate for reuse. Artefacts from early stages may be more useful than artefacts from late stages: a reusable design may be more effective than reusable code.

8.5 Other Process Models

The Fusion method can be adapted to spiral, or risk-driven, design by iterating the following sequence:

1. Define the product.
2. Define the system architecture.
3. Plan the project. Perform risk analysis, and consider going back to step 2.
4. Analyse, design, and implement all or part of the product (this is the Fusion process). Perform risk analysis, and consider going back to step 1 or 2.
5. Deliver product.

The Fusion model can be used incrementally by having several concurrent Fusion processes out of step with one another. For example, at a given time, process A may deliver an increment, while process B is undergoing implementation, and process C is undergoing design.

8.6 Advantages and Disadvantages of Object Oriented Development

The following advantages have been claimed for object oriented development.

Data Abstraction allows the implementation of a class to change without changing its interface.

Compatibility Interface conventions make it easier to combine software components.

Flexibility Classes provide natural units for task allocation in software development.

Reuse The class concept provides greater support for reusing code.

Extensibility The loose coupling of classes, and careful use of inheritance, make it easier to extend OO programs than functional programs.

Maintenance The modular structure of OO programs makes them easier to maintain than functional programs.

But the following problems have also been reported.

Focus on code: There is too much emphasis on programming languages and techniques, to the detriment of the development process. Specifically, analysis and design models are not abstract enough.

Team work: It is not clear how to allocate OO development work amongst teams. There is a tendency towards one person/class.

Difficult to find objects: Experience helps, but it is hard to find the appropriate objects for an application first time round.

Failure of function-oriented methods: Traditional techniques of function-oriented analysis and design do not work well if the target language is OO.

Management problems: New management styles are required for OO software development; general and effective techniques have not been reported.

9 Implementation

Implementation is the process of building a software product from the design documents. Much of implementation is programming, or coding, but there are other things to do. Implementation is not a particularly creative process; if the design work has been done well, coding is almost mechanical, although it is best done by skilled programmers.

9.1 The Implementation Discipline (UPEDU)

UPEDU identifies five principal activities that are performed during implementation:

Plan Integration. Make a plan for developing the various components of the system and integrating them into progressively larger subsets of the product. For a very small system, this step can be omitted.

The goal of planning is to avoid major jumps where things can go wrong. Instead of developing 50 modules and then putting them altogether at once, figure out how subunits of 5 modules can be assembled, then 10 or 15 modules, and, when that is achieved, go for 50 modules.

Implement Components. This is what most people mean by "implementation". Programmers, working from the detailed design document of each module, write code for the modules.

Coding is an important activity that we will discuss at some length later (Section 9.2).

Fix Defects. The code will contain errors, some of which are immediately obvious. For example, a module might compile with errors. These defects are fixed quickly. Unit testing (the next item) will also reveal defects that must be corrected.

Test Units. The completed modules are tested individually (that is, without linking them into a larger part of the system). It will usually be necessary to write special code to call the functions of the module. There are various names for this code: ***driver***, ***scaffolding***, ***test harness***, etc.

Review Code. The code is reviewed carefully. Reviewing is an important part of the process because it may reveal deep, or latent, errors that do not appear in tests. Some studies suggest that reviewing code is a more effective way of removing defects than testing. (The comparison is made with respect to a metric such as defects detected per person-hour.)

Techniques for reviewing will be discussed in detail later in the course.

9.1.1 Traceability

Developers must be able to follow a feature of the system through the requirements, design, and implementation stages. For example, given a function in a class, we should be able to quickly find that function in the design and to identify the requirement that it is supposed to meet. When this is possible, the system is ***traceable***. Traceability can be provided by cross-referencing, documentation tools (see Section 9.2.8), and other means.

Some development systems provide ***round-trip engineering***. Ideally, it should be possible to work in two directions:

- make a change to the design and use the development tool to generate new code;
- make a change to the code and use the development tool to modify the design documents.

Current tools (e.g., those from Rational) permit limited forms of round-trip engineering. For example, class declarations can be changed in both directions. Round-trip engineering of algorithmic code is beyond the state-of-the-art and will probably remain so for some time.

9.1.2 Information Needed for Implementation

Implementors need three kinds of information:

- The ***design documents*** are obviously necessary and are the primary source of information.
- ***Algorithmic*** information from the application domain may be required. Programmers are expected to know how to implement hash tables, balanced trees, and so on, but they may not know how to compute insurance premiums or highway curves.
- ***Rationale*** information explains why a particular strategy was adopted in favour of other strategies that might look equally good. Developers should both make use of rationale information and include it in comments, so that maintenance programmers don't make mistakes that the designers avoided.

  For example, the design might specify synchronous communication between processors using RPC (remote procedure call). The rationale for this decision is that messages are blocked when the network is busy and the network is therefore less likely to become overloaded. If the rationale is omitted, a smart programmer might decide to use MOM (message-oriented middleware) for communication because it is easier to implement and seems to be more efficient because it is asynchronous and non-blocking. The problem with MOM is that, when activity is high, the network may become saturated with waiting messages.[11]

9.2 Coding Conventions

When the design has been completed, coding begins. Code must be written in such a way that people can read it. Compare

```
void calc (double m[],char *g){
double tm=0.0; for (int t=0;t<MAX_TASKS;t++) tm+=m[t];
int i=int(floor(12.0*(tm-MIN)/(MAX-MIN))); strcpy(g,let[i]);}
```

and

```
void calculateGrade (double marks[], char *grade)
{
    double totalMark = 0.0;
    for (int task = 0; task < MAX_TASKS; task++)
        totalMark += marks[task];
```

[11] See `http://www.sei.cmu.edu/str/descriptions/rpc_body.html` for details.

```
        int gradeIndex =
            int(floor(12.0 * (totalMark - minMarks) / (maxMarks - minMarks)));
        strcpy(grade, letterGrades[gradeIndex]);
    }
```

These two function definitions are identical as far as the compiler is concerned. For a human reader, however, the difference is like night and day. Although a programmer can understand what `calc()` is doing in a technical sense, the function has very little "meaning" to a person.

The second version of the function differs in only two ways from the first: it makes use of "white space" (tabs, line breaks, and blanks) to format the code, and it uses descriptive identifiers. With these changes, however, we can quite easily guess what the program is doing, and might even find mistakes in it without any further documentation.

In practice, we would include comments even in a function as simple as this. But note how the choice of identifiers reduces the need for comments.

The three basic rules of coding are:

- Use a clear and consistent layout.
- Choose descriptive and mnemonic names for constants, types, variables, and functions.
- Use comments when the meaning of the code by itself is not completely obvious and unambiguous.

These are generalities; in the following sections we look at specific issues.

9.2.1 Layout

The most important rule for layout is that *code must be indented according to its nesting level.* The body of a function must be indented with respect to its header; the body of a `for`, `while`, or `switch` statement must be indented with respect to its first line; and similarly for `if` statements and other nested structures.

You can choose the amount of indentation but you should be consistent. A default `tab` character (eight spaces) is too much: three or four spaces is sufficient. Most editors and programming environments allow you to set the width of a tab character appropriately.

Bad indentation makes a program harder to read and can also be a source of obscure bugs. A programmer reading

```
    while (*p)
        p->processChar();
        p++;
```

assumes that `p++` is part of the loop. In fact, it isn't, and this loop would cause the program to "hang" unaccountably.

Although indentation is essential, there is some freedom in placement of opening and closing braces. Most experienced C++ programmers position the braces for maximum visibility, as in this example:

```
Entry *addEntry (Entry * & root, char *name)
    // Add a name to the binary search tree of file descriptors.
{
    if (root == NULL)
    {
        root = new Entry(name);
        if (root == NULL)
            giveUp("No space for new entry", "");
        return root;
    }
    else
    {
        int cmp = strcmp(name, root->getName());
        if (cmp < 0)
            return addEntry(root->left, name);
        else if (cmp > 0)
            return addEntry(root->right, name);
        else
            // No action needed for duplicate entries.
            return root;
    }
}
```

Other programmers prefer to reduce the number of lines by moving the opening braces ("{") to the previous line, like this:

```
Entry *addEntry (Entry * & root, char *name) {
    // Add a name to the binary search tree of file descriptors.
    if (root == NULL) {
        root = new Entry(name);
        if (root == NULL)
            giveUp("No space for new entry", "");
        return root;
    } else {
        int cmp = strcmp(name, root->getName());
        if (cmp < 0)
            return addEntry(root->left, name);
        else if (cmp > 0)
            return addEntry(root->right, name);
        else
            // No action needed for duplicate entries.
            return root;
    }
}
```

The amount of paper that you save by writing in this way is minimal and, on the downside, it is much harder to find corresponding pairs of braces.

In addition to indentation, you should use blank lines to separate parts of the code. Here are some places where a blank line is often a good idea but is not essential:

- between method declarations in a class declaration;
- between variable declarations in a class declaration;
- between major sections of a complicated function.

Here are some places where some white space is essential. Put one or more blank lines between:

- `public`, `protected`, and `private` sections of a class declaration;
- class declarations (if you have more than one class declaration in a file, which is not usually a good idea);
- function and method declarations.

9.2.2 Naming Conventions

Various kinds of name occur within a program:

- constants;
- types;
- local variables;
- attributes (data members);
- functions;
- methods (member functions).

It is easier to understand a program if you can guess the kind of a name without having to look for its declaration which may be far away or even in a different file. There are various conventions for names. You can use:

- one of the conventions described here;
- your own convention;
- your employer's convention.

You may not have an option: some employers require their programmers to follow the company style even if it is not a good style.

As a general rule, *the length of a name should depend on its scope.* Names that are used pervasively in a program, such as global constants, must have long descriptive names. A name that has a small scope, such as the index variable of a one-line `for` statement, can be short: one letter is often sufficient.

Another rule that is used by almost all C++ programmers is that *constant names are written with upper case letters and may include underscores.*

9.2.3 Brown University Conventions

Brown University has a set of coding standards used for introductory software engineering courses. Most of the conventions are reasonable, but you can invent your own variations. Here are some of the key points of the "Brown Standard", with my (P.G.) comments in *[square brackets]*:

- File names use lower case characters only.

  UNIX systems distinguish cases in file names: `mailbox.h` and `MailBox.h` are different files. One way to avoid mistakes is to lower case letters only in file names.

  Windows does *not* distinguish letter case in file names. This can cause problems when you move source code from one system to another. If you use lower case letters consistently, you should not have too many problems moving code between systems. Note, however, that some Windows programs generate default extensions that have upper case letters!

- Types and classes start with the project name.

  An abbreviated project name is allowed. For example, if you are working on a project called `MagicMysteryTour`, you could abbreviate it to `MMT` and use this string as a prefix to all type and class names: `MMTInteger`, `MMTUserInterface`, and so on.

  [I don't think this is necessary for projects. The components of a project are usually contained within a single directory, or tree of directories, and this is sufficient indication of ownership. The situation is different for a library, because it must be possible to import library components without name collisions.]

- Methods start with a lower case letter and use upper case letters to separate words. Examples: `getScore`, `isLunchTime`.

  [I use this notation for both methods `and` *attributes (see below). In the code, you can usually distinguish methods and attributes because method names are followed by parentheses.]*

- Attributes start with a lower case letter and use underscores to separate words. Examples: `start_time`, `current_task`.

- Class constants use upper case letters with underscores between words. Examples: `MAXIMUM_TEMPERATURE`, `MAIN_WINDOW_WIDTH`.

- Global names are prefixed with the project name. Example: `MMTstandardDeviation`.

  [I use the same convention for methods and global functions. When a method is used in its own class, this can lead to ambiguity. In many cases, however, you can recognize methods because they appear as `o.method()` *or* `p->method()`*. My rule for global variables is* ***avoid them.****]*

- Local variables are written entirely in lower case without underscore. Examples: `index`, `nextitem`.

Whatever convention you use, it is helpful to distinguish attributes from other variables. In *Large-Scale C++ Software Design*, John Lakos suggests prefixing all attributes with `d_`. This has several advantages; one of them is that it becomes easy to write constructors without having to invent silly variations. For example:

```
Clock::Clock (int hours, int minutes, int seconds)
{
    d_hours = hours;
    d_minutes = minutes;
    d_seconds = seconds;
}
```

9.2.4 Hungarian Notation

Hungarian notation was introduced at Microsoft during the development of OS/2. It is called "Hungarian" because its inventor, Charles Simonyi, is Hungarian. Also, identifiers that use this convention are hard to pronounce, like Hungarian words (if you are not Hungarian, that is).

If you do any programming in the Windows environment, you will find it almost essential to learn Hungarian notation. Even if you have no intention of ever doing any Windows programming, you should consider adopting this convention, because it is quite effective.

Hungarian variable names start with a small number of lower case letters that identify the type of the variable. These letters are followed by a descriptive name that uses an upper case letter at the beginning of each word.

For example, a Windows programmer knows that the variable `lpszMessage` contains a long pointer to a string terminated with a zero byte. The name suggests that the string contains a message of some kind.

The following table shows some commonly used Hungarian prefixes.

`c`	character
`by`	unsigned `char` or byte
`n`	short integer (usually 16 bits)
`i`	integer (usually 32 bits)
`x, y`	integer coordinate
`cx, cy`	integer used as length ("count") in X or Y direction
`b`	boolean
`f`	flag (equivalent to boolean)
`w`	word (unsigned short integer)
`l`	long integer
`dw`	double word (unsigned long integer)
`fn`	function
`s`	string
`sz`	zero-terminated string
`h`	handle (for Windows programming)
`p`	pointer

9.2.5 Commenting Conventions

Comments are an essential part of a program but you should not over use them. The following rule will help you to avoid comments:

Comments should not provide information that can be easily inferred from the code.

There are two ways of applying this rule.

- Use it to eliminate pointless comments:

  ```
  counter++;   // Increment counter.

  // Loop through all values of index.
  for (index = 0; index < MAXINDEX; index++)
  { .... }
  ```

- Use it to improve existing code. When there is a comment in your code that is not pointless, ask yourself "How can I change the code so that this comment becomes pointless?"

  You can often improve variable declarations by applying this rule. Change

  ```
  int np;       // Number of pixels counted.
  int flag;     // 1 if there is more input, otherwise 0.
  int state;    // 0 = closed, 1 = ajar, 2 = open.
  double xcm;   // X-coordinate of centre of mass.
  double ycm;   // Y-coordinate of centre of mass.
  ```

  to

  ```
  int pixelCount;
  bool moreInput;
  enum { CLOSED, AJAR, OPEN } doorStatus;
  Point massCentre;
  ```

There should usually be a comment of some kind at the following places:

- At the beginning of each file (header or implementation), there should be a comment giving the project that the file belongs to and the purpose of this file.

- Each class declaration should have a comment explaining what the class is for. It is often better to describe an *instance* rather than the class itself:

  ```
  class ChessPiece
  {
      // An instance of this class is a chess piece that has a position,
      // has a colour, and can move in a particular way.
  ....
  ```

- Each method or function should have comments explaining what it does and how it works.

  In many cases, it is best to put a comment explaining *what the function does* in the header file or class declaration and a different comment explaining *how the function works* with the implementation. This is because the header file will often be read by a client but the implementation file should be read only by the owner. For example:

  ```
  class MathLibrary
  {
  public:
      // Return square root of x.
      double sqrt (double x);
      ....
  };

  MathLibrary::sqrt (double x)
  {
      // Use Newton-Raphson method to compute square root of x.
      ....
  }
  ```

- Each constant and variable should have a comment explaining its role *unless its name makes its purpose obvious.*

9.2.6 Block Comments

You can use "block comments" at significant places in the code: for example, at the beginning of a file. Here is an example of a block comment:

```
/*************************************/
/*                                   */
/*       The Orbiting Menagerie      */
/*       Author:  Peter Grogono      */
/*       Date:      24 May 1984      */
/*         (c) Peter Grogono         */
/*                                   */
/*************************************/
```

Problems with block comments include:

- they take time to write;
- they take time to maintain;
- they become ugly if they are not maintained properly (alignment is lost, etc.);
- they take up a lot of space.

Nevertheless, if you like block comments and have the patience to write them, they effectively highlight important divisions in the code.

9.2.7 Inline Comments

"Inline comments" are comments that are interleaved with code in the body of a method or function. Use inline comments whenever the logic of the function is not obvious. Inline comments *must respect the alignment of the code*. You can align them to the current left margin or to one indent with respect to the current left margin. Comments must always confirm or strengthen the logical structure of the code; they must never weaken it.

Here is an example of a function with inline comments.

```
void collide (Ball *a, Ball *b, double time)
{
    // Process a collision between two balls.

    // Local time increment suitable for ball impacts.
    double DT = PI / (STEPS * OM_BALL);

    // Move balls to their positions at time of impact.
    a->pos += a->vel * a->impactTime;
    b->pos += b->vel * a->impactTime;

    // Loop while balls are in contact.
    int steps = 0;
    while (true)
    {

        // Compute separation between balls and force separating them.
        Vector sep = a->pos - b->pos;
        double force = (DIA - sep.norm()) * BALL_FORCE;
        Vector separationForce;
        if (force > 0.0)
        {
            Vector normalForce = sep.normalize() * force;

            // Find relative velocity at impact point and deduce tangential force.
            Vector aVel = a->vel - a->spinVel * (sep * 0.5);
            Vector bVel = b->vel + b->spinVel * (sep * 0.5);
            Vector rVel = aVel - bVel;
            Vector tangentForce = rVel.normalize() * (BALL_BALL_FRICTION * force);
            separationForce = normalForce + tangentForce;
        }

        // Find forces due to table.
        Vector aTableForce = a->ballTableForce();
        Vector bTableForce = b->ballTableForce();
        if (    separationForce.iszero() &&
                aTableForce.iszero() &&
                bTableForce.iszero() && steps > 0)
            // No forces: collision has ended.
```

```
                break;

            // Effect of forces on ball a.
            a->acc = (separationForce + aTableForce) / BALL_MASS;
            a->vel += a->acc * DT;
            a->pos += a->vel * DT;
            a->spin_acc = ((sep * 0.5) * separationForce + bottom * aTableForce) / MOM_INT;
            a->spinVel += a->spin_acc * DT;
            a->updateSpin(DT);

            // Effect of forces on ball b.
            b->acc = (- separationForce + bTableForce) / BALL_MASS;
            b->vel += b->acc * DT;
            b->pos += b->vel * DT;
            b->spin_acc = ((sep * 0.5) * separationForce + bottom * bTableForce) / MOM_INT;
            b->spinVel += b->spin_acc * DT;
            b->updateSpin(DT);

            steps++;
        }

        // Update motion parameters for both balls.
        a->checkMotion(time);
        b->checkMotion(time);
    }
```

9.2.8 Documentation Tools

There are many tools available to assist with documentation. Using a tool has many advantages: it forces all developers to follow standards; it simplifies distribution of documentation (most tools can build web pages); and it reduces the probability that the documentation will become out of date.

Popular tools include:

- Javadoc for Java programs

  `http://java.sun.com/j2se/javadoc/`

- Doxygen for "C++, C, Java, IDL and to some extent PHP and C#" programs

  `http://www.stack.nl/~dimitri/doxygen/`

  For an example of Doxygen documentation, see

  `http://www.cs.concordia.ca/~faculty/grogono/CUGL/cugl.html`

  There are links to both the HTML and the PDF versions of the documentation.

- Cocoon for C++

  `http://www.stratasys.com/software/cocoon/`

9.2.9 Standard Types

Type representations vary between C++ implementations. For example, the type `int` may be represented as 16, 32, or 64 bits. Consequently, it is common practice for experienced C++ programmers to define a set of types for a program and use them throughout the program. Here is one such set:

```
typedef  int           Integer;

typedef  char          Boolean;

typedef  char          Character;

typedef  double        Float;

typedef  char *        Text;

typedef  const char *  ConstText;
```

To see why this is helpful, consider the following scenario. Programmer A develops a program using these types for a system with 64-bit integers. The program requires integer values greater than 2^{32}. Programmer B is asked to modify the source code so that the program will run on processors for which `int` is represented by 32-bit integers. Programmer B changes one line

```
typedef long Integer;
```

and recompiles the program. Everything will work, provided that `long` is implemented using 64-bits. (If it isn't, B can type `long long` instead.)

Other applications include: a programmer using an up to date compiler can redefine `Boolean` as `bool` (which is included in the C++ Standard); and can redefine `Text` to allow unicode characters.

The need for `typedef` statements is perhaps less than it used to be, because there is less variation between architectures than there used to be. With the final demise of DOS and its segmented address space, most processors provide 32-bit integers and pointers. The distinction between 16 and 32 bits was important because $2^{16} = 65,536$ is not a large number. The distinction between 32 and 64 bits is less important since $2^{32} = 4,294,967,296$ is large enough for many purposes. Since the introduction of `bool` (and the constants `true` and `false`), it is not really necessary to define `Boolean`. Nevertheless, you may wish to use `typedef`s to improve the portability of your code.

10 Testing

We want to avoid *software failure.* There are many different ways in which software may fail.

- A requirement was omitted.

  Example: the client loses a master file and complains that the program should have maintained backups of important files automatically. This is a serious "failure" from the client's point of view but we cannot blame the software product for the failure because there was no requirement for automatic backups.

- The requirements were unimplementable.

  Example: the client complains that the system is so busy with housekeeping that it never gets any work done. Investigation shows that both hardware and software are performing correctly, but the hardware configuration is inadequate for the client's needs.

- The specification was incorrect.

  Example: a student logs in to the University's Student Information System and requests her grade for a non-existent course; the system crashes because the specification stated that a course number consisted of four letters and three digits and defined no further validation.

- There was an error in the high-level design.

  Example: The component for remote access must be linked to the client database in order to validate customers who access the system by telephone. Since the link was omitted in the high-level design, the people working on the detailed design did not notice the need for validation.

- There was an error in the detailed design.

  Example: the specification required unique keys for a database but the design did not enforce uniqueness. The program was implemented according to the design but does not meet its specification.

- There was an error in the code.

  Example: the design states that no action is to be performed if a certain pointer is `NULL`. The programmer omitted the test for `NULL` and the program crashed when it attempted to dereference the pointer.

Here is the official terminology:

- In each of the examples above, a *fault* causes a *failure.*
- The process of discovering what caused a failure is called *fault identification.*
- The process of ensuring that the failure does not happen again is called *fault correction* or *fault removal.*

"Bug" is a popular term for "fault". There is a story to the effect that the first hardware failure was caused by a bug that was trapped between relay contacts in an early computer. This story is almost certainly mythical, because engineers were using the word "bug" before any computers had been built. But the usage is important: the word "bug" suggests that faults wander into the code from some external source and that they are therefore random occurrences beyond the programmer's control. This is *not possible* for software: if your code has faults, it is because you put them there.

We have seen that faults can exist in requirements, specifications, and designs. We have also seen ways of avoiding such faults (e.g., code review). This section is concerned mainly with faults in the code.

There are essentially two ways of ensuring that the number of faults in code is small or — ideally — zero. One is to prove that the code has no faults and the other is to perform tests designed to find faults. Much effort has been put into proof techniques for programs, but the resulting systems are not widely used in industry. (The main exception is in the UK, where there are legal requirements for certification for certain kinds of critical application.)

Consequently, the main technique for fault detection is testing. Dijkstra has said: ***testing can reveal only the presence of faults, never their absence***. In other words, you can never be certain that your program contains no faults but, with sufficient testing, you can be confident that the number of remaining faults is acceptably small.

10.1 Coding Errors

There are many kinds of fault that can exist in code; some of them are listed below.

Syntax Errors. Most syntax errors are detected by the compiler, but a few are not. (Some people would argue that all syntax errors are caught by the compiler by definition, and that a compiled program therefore cannot contain syntax errors.) However, a declaration such as

```
char*   p, q;
```

is best classified as a syntax error (if `q` is meant to be a pointer) but the compiler does not report it.

Algorithmic Errors. A method does not execute according to its specification because there is an error in the coding logic. Some common causes are:

- incorrect initialization;
- "off by one" error (example: starting at 1 rather than 0, stopping at N rather than $N - 1$);
- incorrect condition (example: forgetting de Morgan's laws);
- misspelling a variable name; type errors that remain undetected in the presence of implicit conversion.

Precision Errors. These arise when a programmer uses an `int` where a `long` was needed, or a `float` where a `double` was needed. Precision errors can also occur as the result of implicit conversions. A common example is expecting 2/3 to evaluate to 0.666667 rather than 0 (note that the operands might be variables rather than integer literals).

Documentation Errors. The comments and accompanying code are inconsistent with one another. If the code is correct, there is not actually a fault. However, there is a strong likelihood of a fault being introduced later by a programmer who believes the documentation rather than the code.

Stress Errors. These are faults that occur when the system is stretched beyond its designed limits. For example, a simple operating system that had an array of size N for active users might fail when user $N + 1$ tries to log in. Two kinds of failure are possible: if the new user is refused but the system continues to run, it is a *soft failure*; if the system crashes, it is a *hard failure*.

Capacity Errors. A word processor is required to handle documents up to 2 Gb in size but its performance is unacceptably slow for documents over 1 Mb.

Timing Errors. A system with multiple processes or threads fails occasionally because two events occur in the wrong order or simultaneously. Errors of this kind are very hard to diagnose.

Performance Errors. The requirements state that the system must respond in less than 1 second but in fact the response to some kinds of query is more than 10 seconds.

Recovery Errors. The system is "fault tolerant": that is, it is supposed to detect certain kinds of failure and recover gracefully. A failure to recover from an anticipated failure is a "recovery error".

Hardware Errors. Hardware errors are usually, but not necessarily, beyond the programmer's control. However, safety-critical systems may anticipate hardware failures and, when such a failure occurs, handle it by switching to a stand-by system, for example.

Error of Omission. Something which should be present is not. For example, a variable has not been initialized.

Error of Commission. Something that should not be present is. For example, an exception handler for a low-level failure is included when handling should have been delegated to a higher level component of the system.

10.2 Fault Classification

Various groups, mostly industrial, have attempted to classify faults. Appropriate fault classification, combined with systematic detection and recording of faults, can provide insight into the strengths and weaknesses of the software development process.

The *Orthogonal Defect Classification* system (Table 7.1, page 282), developed at IBM, has the following categories of defect (defect $\equiv$ fault):

- function;
- interface;
- checking/validation;
- assignment/initialization;

- timing/serialization;
- build/package/merge (repositories, version control);
- documentation;
- algorithm.

Hewlett-Packard have another fault classification system (Pfleeger, Figure 7.1, page 283) that they use for "profiling" problems within divisions of the company. Here are the results for one division (see also Figure 7.2, page 284):

Fault Category	Occurrence (%)
Logic	32
Documentation	19
Computation	18
Other code	11
Data handling	6
Requirements	5
Process/interprocess	5
Hardware	4

10.3 Varieties of Testing

There are various ways to detect faults: reviews are important but, at some stage, it is usually necessary to resort to *testing*.

A simple definition of testing: execute the system (or one of its components) and compare the results with anticipated results.

The "anticipated results" are the results predicted by the requirements if we are testing the whole system. It may be harder to pin down the anticipated results when we are testing components but, in general, they should be predicted by the design documents.

There are various kinds of testing; not all of the kinds listed below would be needed in small or medium scale projects.

Unit Testing. (Also called **module testing** or **component testing**.) We test a component of the system in isolation. The component will expect inputs from other system components and will generate outputs that are sent to other components. For unit testing, we must construct an environment for the component that provides suitable inputs and records the generated outputs.

Integration Testing. We test all of the components of the system together to ensure that the system as a whole "works" (that is, compiles and doesn't crash too often).

Function Testing. Assuming that the system "works" , we test its functions in detail, using the requirements as a basis for test design.

Performance Testing. We measure the system's speed, memory and disk usage, etc., to confirm that it meets performance requirements.

Acceptance Testing. The previous tests are usually performed by the suppliers. When the suppliers believe the software is ready for delivery, they invite the clients in to confirm that they accept the software.

Installation Testing. When the clients have accepted the software, it is installed at their site (or sites) and tested again.

10.4 Testing Teams

When you test your own code, it is tempting to be "friendly" to it. After all, you don't want your nice, new program to crash and, partly subconsciously, you will avoid giving it test data that might make it crash.

This is the wrong attitude! A tester should be as nasty to the code as possible. The goal of testing is to make the software fail.

It follows that programmers are not the best people to test their own code. It is better to assemble a separate *testing team* of people who have no interest in the code and who get rewarded for smashing it.

Programmers don't like people who smash their code. It is important to develop psychological habits that avoid "blaming" developers for errors in their code. The viewpoint that everyone should adopt is: "we all want the system to work as well as possible and we will all do what we can to remove errors from it". This attitude is sometimes called (following Gerald Weinberg) *egoless programming.*

Random tests are not very useful. The first task of the testing team is to design a battery of tests.

There is an important distinction between *black box testing* and *white box testing.* The terms come from electrical engineering; instead of "white" box, we can also so "clear" or "open" box.

Black box testing tests the component without making use of any knowledge of how it works.

Clear box testing tests the component using information about how it works.

Suppose that you are testing a function

```
    double sin (double x);
```

that is supposed to compute the trigonometric function $\sin x$ for given real values of x.

You execute the function with various arguments and check the results against a table of trigonometric functions. This is *black box* testing.

Now suppose that you look at the body of the function and find something like Figure 14 (or, hopefully, something more efficient). The code suggests various tests: $x < 0$, $x > 0$, $x < 2\pi$, $x > 2\pi$, $x < \frac{\pi}{2}$, $x > \frac{\pi}{2}$, $x < 10^{-6}$, and $x > 10^{-6}$, as well as tests at the boundaries: $x = 0$, $x = 2\pi$, $x = \frac{\pi}{2}$, $x = 10^{-6}$, and $x > 10^{-6}$. This is *clear box* testing.

```
double sin (double x)
{
    if (x < 0.0)
        return - sin(-x);
    while (x > TWO_PI)
        x -= TWO_PI;
    if (x > HALF_PI)
        return sin(PI - x);
    if (x < 1e-6)
        return x;
    double result = x;
    for (i = 3; i <= 11; i += 2;)
        result -= (result * x * x)/(i * (i - 1));
    return result;
}
```

Figure 14: Implementation of sin function

Here is a more formal view of black/clear boxes that was developed as part of IBM's "clean-room" project.

- Define a component by a function $I \to O$ (input to output). This is a "black box" with no hidden states.
- Define a component by a function $I \times H \to O$ (input and history to output, where "history" is the sequence of previous inputs). This is a "black box" with hidden states (we can tell this because the output depends on the "history" as well as the current input.
- Define a component by a function $I \times \Sigma \to O \times \Sigma$ (input and state to output and new state). This is a "grey box": we know that it has a state and we know how the state changes, but we do not know how it works.
- Define a component by a function $I \times \Sigma \to O \times \Sigma$ and provide a procedure explaining how the state changes. This is a "clear box".

10.5 Testing Components

A *test* is a procedure:

- Choose data that will be provided to the component as input.
- Decide, before executing the component, what the expected results should be.
- Execute the component with the chosen data and compare the actual and expected results.
- If the expected and actual results are the same, the test has *passed*. Otherwise the test has *failed*.

A *test set* is a collection of tests.

Regression testing on a test set is carried out as follows:

- Perform tests from the test set one after another.
- When a test fails, stop testing and wait until the fault has been corrected.
- Start the tests again from the beginning.

The definitions given above must be adapted to the circumstances.

- Inputs may consist of keystrokes, mouse movements and clicks, and other events, instead of just "data". Tests should specify exactly what events will occur and in what order. Tests may even specify timing ("the left mouse button is clicked twice with less than 0.5 seconds between clicks").
- Outputs may appear in many forms: a window is created, destroyed, or turned grey; a popup menu appears; a list of files is displayed; etc.

Regression testing can become very tedious if a test set contains many tests that require user actions:

Test 23.4: Help functions.

- Move the cursor to the `Help` icon.
- Click the left mouse button once. A popup menu opens.
- Move the cursor to the `Contents` box.
- Click the left mouse button once.
- . . . .

A test set should be a comprehensive collection of specific tests for the software component. Testing is generally more effective if information about the implementation is used — that is, it is at least partially "white box".

- Include *valid data*: data that the component is supposed to process and for which the expected results are known.
- Include *invalid data*: data that the component is not supposed to process.
- Include *boundary cases.* For example, if a table is supposed to store up to 20 items, test it with (at least) 0, 19, 20, and 21 items.

The *coverage* of a test set is the code that will be exercised by performing all the tests in the test set.

- *Statement testing* ensures that all statements are executed.
- *Branch testing* ensures that, wherever there is a "branch point" (`if` or `switch` statement), each branch is executed in at least one test.
- *Path testing* executes each path in the component (considered as a directed graph).

There are other, more elaborate possibilities. For example, we can ensure that each definition/use pair is tested.

10.6 Integration Testing

When components have been tested, they can be incorporated into the growing system. ***Integration testing*** means testing a group of components together. There are various ways in which integration testing can be carried out:

Bottom-up Integration. First test the components C_n that don't use any other components; then test the components C_{n-1} that use the components C_n but no others; and so on.

In bottom-up integration, the components that a component uses have already been tested and can be used in the test on the assumption that they work correctly. However, we will need to build ***drivers*** to call the components, because the code that will eventually call them has not yet been tested. Drivers are written as simply as possible but should realistically exercise the components under test.

Top-down Testing. First test the components C_0 that are not called by any other components; then test the components C_1 that are called by components C_0; and so on.

In top-down integration, the components that call the component under test can be used on the assumption that they work correctly. However, we will need to build ***stubs*** that simulate the components called by the component under test. Stubs are written as simply as possible: they respond to all reasonable requests but return very simple results.

Big-Bang Integration. First test the components individually, then link all the components together, click `Run`, and see what happens (usually the system crashes immediately). For systems of any complexity, big-bang integration is unlikely to be efficient. The problem is that, when failures occur, it is hard to figure out what caused them.

Sandwich Integration. This is a combination of top-down and bottom-up, the objective being to converge on a layer in the "middle" of the system for which higher and lower components have already been tested. The following advantages are claimed for sandwich testing:

- Integration testing can start early in the testing process, because groups of components can be tested.
- Top-level components (control) and bottom-level components (utilities) are both tested early.

It is interesting to note that we can perform neither pure bottom-up nor pure top-down integration testing if there are cycles in the dependency graph. If components $C_1, C_2, \ldots, C_n$ for a cycle, we are forced to test all of the simultaneously. This is a very good reason to avoid cycles in the component dependency graph!

The key points for integration testing are:

- Each component should be tested exactly once.
- A single test should involve as few components as possible, preferably one.
- A component should never be modified for the purpose of testing.

- It will usually be necessary to write additional code (stubs, drivers, etc.) specifically for testing. It is a good idea to keep this code so that it can be used for testing improved versions of the system after release.

10.7 Testing Object Oriented Code

A high proportion of new code is object oriented. As such, it presents a few special problems for testing.

- Testing classes individually is difficult in the presence of inheritance.

  Suppose that a parent class P has children $C_1, C_2, \ldots$. Then methods in C_i may invoke methods in P. If an error in C_i is caused by an error in P, correcting the error in P may break code in C_j $(i \neq j)$.

- Dynamic binding can also make testing harder. A statement of the form `p->f()` may invoke different methods each time it is called, depending on the classs of `*p`.

Although OO code has special problems, it should also have certain advantages with respect to testing. In particular, if classes respect encapsulation, it should be possible to test them independently of classes that use them (although not classes that they use).

10.8 Testing Strategies

Testing should always proceed according to a predefined plan rather than in an *ad hoc* fashion. (*Ad hoc* testing produces odd hacks.)

A test plan consists of a collection of tests and a strategy for applying them.

- The plan should describe what happens if a test fails. Usually, the failure is noted and tests proceed. However, if the test is a particularly significant one, testing may have to be delayed until the fault is corrected. For example, if the system fails on startup, the testing team have to wait until the problem is fixed.

- The plan should describe what happens when the faults revealed by tests have been corrected. A common strategy is called ***regression testing***. Assume that the tests are numbered $1, 2, 3, \ldots$ and that test n fails. After the fault has been corrected, regression testing requires that tests $1, 2, \ldots, n$ must all be repeated.

  Regression testing takes a long time. If possible it should be automated (see below). Otherwise, a careful analysis should be performed to see if all of the previous tests must be repeated or whether some of them are, in some sense, independent of test n.

- Test results must be carefully recorded.

Automation. If possible, testing should be ***automated***. This requires a program that can:

- invoke the component under test with suitable test data;

- observe the effect of the test;

- decide whether to stop or continue, depending on the results for the test.

Automatic testing is fairly straightforward if we are dealing with functions that accept paremeters and return results. It is more complicated if we are testing, for example, a windows-based application. However, there are testing tools that will record a sequence of events (keystrokes, mouse movements, etc.) and "play them back" during testing.

Although it is not strictly "testing", software tools can be used to check code both statically and dynamically.

- ***Static analyzers*** check that variables are initialized before they are used, that all statements can be reached, that components are linked in appropriate ways, etc. Compilesr perform some static analysis, of course, but they tend to be better at within-component analysis than at between-component analysis.
- ***Test case generators*** analyze the source code a generate tests based on it. These tools can ensure, for example, that there are tests for each branch of every condition statements. Of course, if the code contains errors, a test case generator may generate inappropriate tests.
- ***Dynamic analyzers*** make measurements while the code is running. They can collect statistics about the frequency of execution of various blocks and look for potential synchronization problems in concurrent processes.
- ***Capture and replay*** tools record a sequence of user actions and can play them back later. User actions include keystrokes, mouse movements, and mouse clicks. These tools avoid the time-consuming and error-prone task of repeating a complicated series of actions whenever a test has to be run.

When do you stop testing? The testing strategy must include a criterion for terminating testing. Ideally, we would terminate testing when there are no faults left. In practice, this would mean testing for ever for most systems. We can make the following observations:

- Assuming that testing is continuous, the rate of fault detection is likely to fall.
- The accumulated cost of testing increases with time.
- The value of the product probably decreases with time (for example, competitors get to the market).

At some time, there will be a break-even point where it is better to release the product with residual faults than it is to continue testing. Of course, it is not always easy to decide exactly when the break-even point has been reached!

10.9 Avoid test failures: read your code

It is often more productive to read the code than to run tests on it. After you have observed a fault, you have to read the code anyway, to see what went wrong. Often, you can avoid the need for debugging by spotting the error before running the tests. There are several ways of reading code.

Desk Checking. Get into the habit of reading code that you have just written ***before*** running it. This often saves time because simple errors sometimes lead to very strange symptoms.

Code Walkthrough. During a "walkthrough", a programmer presents code to a small group line by line, explaining the purpose of each line. Members of the group read the lines as they are presented to ensure that the code actually does what the presenter claims it does. It is important to place emphasis on the code itself, not on the person who wrote it. When people question features of the code, the programmer's job is not to defend it but to admit that it is wrong or explain why it is correct although it appears to be wrong.

It is best not to attempt to correct errors during the walkthrough; the important thing is to identify errors for subsequent correction.

Code Inspection. Code inspection was introduced in 1976 at IBM by Mike Fagan. An inspection is like a walkthrough in that code is studied carefully but an inspection involves another set of documents, which can be requirements, specifications, or designs. The purpose of the inspection is to check that the code does in fact meet the requirements, satisfy the specifications, or implement the designs. The people present include the originators of the documents as well as the implementors of the code.

Inspections are sometimes conducted as three-phase processes. There is a preliminary meeting at which the goals of the inspection are identified. Then team members study the code individually, noting any errors that they find. Finally, there is another meeting of the inspection team during which the errors are presented and discussed.

There has been a substantial amount of research into the effectiveness of code reviews. Most of the results confirm that *reading and inspecting code is more effective than debugging.* That is, the number of faults identified for each person-hour spent is greater when the people are inspecting code than when they are debugging it.

10.10 Project Requirements

Your project should include a ***test plan*** and a ***test report***.

- A minimal test plan should describe your testing strategy and outline the kind of test that you will perform. A detailed test plan would include a list of all the tests that you intend to perform and the expected outcome of each test.
- The test report should describe the process of testing, the number of tests that passed and failed, and the action that you took in response to failed tests.

11 System Testing

Until now, we have discussed program testing. Once the code is working, we can test the system as a whole.

In some cases, the code forms the major part of the system and, when program testing is complete, there is not much more to do. In the majority of cases, however, the code component of a software engineering project is only one part of the total system.

System testing has much in common with program testing and there is no need to dwell on the similarities. For example: ***faults*** in the system lead to ***failures***; the failures must be ***identified*** and the corresponding faults ***corrected***; after correcting the faults, it is advisable to run ***regression*** tests to ensure that new faults have not been introduced.

Suppose that the product is flight-control software for a new kind of aircraft. The ***system*** is the aircraft as a whole. Here is a simplified version of a possible test plan:

Function Testing. The software is thoroughly tested in an environment that simulates the aircraft. That is, the software is provided with inputs close to those that it would receive in actual flight, and it must respond with appropriate control signals.

When the function test results have been accepted, we have a ***functioning system***.

Performance Testing. The so-called "non-functional requirements" of the software are thoroughly tested. Does it respond within acceptable time limits? Are there any circumstances in which available memory might be exhausted? Does it degrade gracefully when things go wrong? If there are duplicated subsystems, is control transferred between them when expected?

Performance tests are listed in the text book and will not be described in detail here. They include: stress, volume, configuration, compatibility, regression, security, timing, environment, quality, recovery, maintenance, documentation, and human factors (or usability) tests.

After performance testing, we have a ***verified*** system.

Acceptance Testing. The tests are run for the clients, still on simulated equipment. Experienced pilots will "fly" the simulated aircraft and will practice unusual as well as standard manouvres.

After acceptance testing, the clients agree that this is the system that they want. The system is said to be ***validated***.

Installation Testing. The software is installed on an actual aircraft and tested again. These tests are conducted slowly and carefully because of the consequences of an error. Several days may be spent during which the plane travels at less than 100 km/h on the runway. Then there will be short "flights" during which the plane gets a few metres off the ground. Only when everyone is fully confident that flight can be controlled will the test pilot perform a true take-off and fly a short distance.

If installation testing goes well, the system is put into use. The maintenance phase begins.

11.1 Verification and Validation

The terms "verification" and "validation" are often bundled together as "V&V". There is, however, a useful distinction that we can make between them:

Verifying the system is the process of ensuring that the software is internally consistent and that all functions perform according to specifications. Informally, verification answers the question: Have we built the ***system right***?

Validating the system is the process of ensuring that the software meets the clients needs. Informally, verification answers the question: Have we built the ***right system***?

11.2 Unique Aspects of System Testing

There are some aspects of system testing that are significantly different from program testing. These are discussed briefly here.

Configuration. The software may have to run in many different configurations. In the worst case, each platform may be unique. Stu Feldman says that,in telecom software, it is not unusual to have several thousand makefiles for a system. All of these must be tested — on a common platform, if possible, but probably on the platform for which they are intended.

Versions. The word "version" is ambiguous because its informal meaning is that of later versions replacing earlier versions. But it is also used in the sense of "version for platform A", "version for platform B", etc. Here, we will assume that each "version" is intended for a particular platform and application.

Releases. The word "release" is the formal term that corresponds to the informal use of "version". The software is issued in a series of numbered ***releases***, each of which is intended to be an improvement on its predecessors. There is usually a distinction between ***major releases*** which incorporate significant enhancements and ***minor releases*** which correct minor problems. Releases are numbered so that "Release 4.3" would be the third revision of the fourth major release.

Production System. After the system has been released to clients, the developers must keep an identical copy so that they can check client's complaints. This copy is called the "production system".

Development System. In addition to the production system, the developers should maintain a separate system into which they can incorporate corrections and improvements. At well-defined points, the "development system" is delivered to the clients, as a new release, and becomes the "production system". At that time, a new development system is started.

Deltas. Software systems can become very large; maintaining complete copies of all of the files of many different versions can require enormous amounts of storage and — more importantly, since storage is cheap — involve significant copying time. A solution is to maintain one complete set of files for the entire system and, for each version, a set of files that define the differences particular to the version. The difference files are called "deltas".

Use of deltas must be supported by an effective tool to manage them, otherwise disaster ensues. There should be a simple and reliable way of combining the basic system files with a set of delta files to obtain a release.

A problem with deltas is that, if the base system is lost, all releases are lost as well. The base system must therefore be regarded as a document of great value.

The UNIX utility SCCS (Source Code Control System) uses deltas to maintain multiple versions of files.

11.3 Reliability, Availability, Maintainability

A system is ***reliable*** to the extent that it can operate without failure for a given time interval. The time is measured while the system is running, not while it is idle. A system that is required to run for 5 seconds a year, and always performs flawlessly for those 5 seconds, is entirely reliable. Reliability is usually measured on a scale of 0 to 1 in which 0 means the system never operates and 1 means that it always operates.

A system is ***available*** to the extent that it is operating correctly when it is needed. If the system is currently running, it is available. If it is "down" for repairs, it is not available. Availability time is real time, not execution time.

A system is ***maintainable*** to the extent that, after a failure, it can be made to operate again within a given period of time with a given probability. Whereas hardware systems are usually unavailable during maintenance, there are an increasing number of software systems that are required to operate while they are being maintained.

We can quantify these ideas as follows. Assume for simplicity that the system is supposed to operate continuously (for example, it might be software for a telephone switch). We measure the time intervals ***between*** failures; the average of these times is the ***mean time to failure*** or *MTTF*.

When the system fails, we record the time taken to repair it and restore operations. The average of these times is the ***mean time to repair*** or *MTTR*.

We define the ***mean time between failures*** as

$$MTBF = MTTF + MTTR$$

According to the text book:

$$\begin{aligned} Reliability &= \frac{MTTF}{1 + MTTF} \\ Availability &= \frac{MTBF}{1 + MTBF} \\ Maintainability &= \frac{1}{1 + MTTR} \end{aligned}$$

Note that:

- each of these parameters has a value between 0 and 1;
- the parameters are meaningless.

Example: A system has $MTTF = 4$ days and $MTTR = 20$ minutes $= 0.0138$ days. If we use days as the unit of time:

$$\begin{aligned} \mathit{Reliability} &= 0.8 \\ \mathit{Availability} &= 0.9863 \end{aligned}$$

But, if we use minutes as the unit of time:

$$\begin{aligned} \mathit{Reliability} &= 0.9998 \\ \mathit{Availability} &= 0.9998 \end{aligned}$$

The following is a more sophisticated analysis, adapted from pp. 622–3 of *Software Engineering: an Engineering Approach* by Peters and Pedrycz (Wiley, 2000).[12]

Assume that the system has two states, *up* (i.e., operating correctly) and *down* (i.e., not operating). Suppose that, during a time interval Δt, the transition probabilities are:

$$\begin{aligned} \mathit{up} \rightarrow \mathit{down} &= \lambda\,\Delta t \\ \mathit{down} \rightarrow \mathit{up} &= \mu\,\Delta t \end{aligned}$$

This system has the following *transition probability matrix*:

$$\begin{bmatrix} 1 - \lambda\,\Delta t & \lambda\,\Delta t \\ \mu\,\Delta t & 1 - \mu\,\Delta t \end{bmatrix}$$

Suppose that $p_u(t)$ is the probability that the system is *down* at time t and $p_d(t)$ is the probability that the system is *up* at time t. Then we can infer from the transition probability matrix that:

$$\begin{aligned} p_u(t)(1 - \lambda\,\Delta t) + p_d(t)\mu\,\Delta t &= p_u(t + \Delta t) \\ p_u(t)\lambda\,\Delta t + p_d(t)(1 - \mu\,\Delta t) &= p_d(t + \Delta t) \end{aligned}$$

We could solve this as a differential equation to obtain the dynamic behaviour of the system, but it is easier to obtain the steady state solution by putting $\Delta t = 0$. This gives the same equation twice:

$$\begin{aligned} -\lambda p_u + \mu p_d &= 0 \\ \lambda p_u - \mu p_d &= 0 \end{aligned}$$

However, we know from probability theory that $p_u + p_d = 1$. Thus we can deduce:

$$\begin{aligned} p_u &= \frac{\mu}{\lambda + \mu} \\ p_d &= \frac{\lambda}{\lambda + \mu} \end{aligned}$$

It seems reasonable to define *availability* as p_u, since this is the probability that the system is "up" at a given time.

[12] A good text book for this course — but for the fact that it costs about \$140.

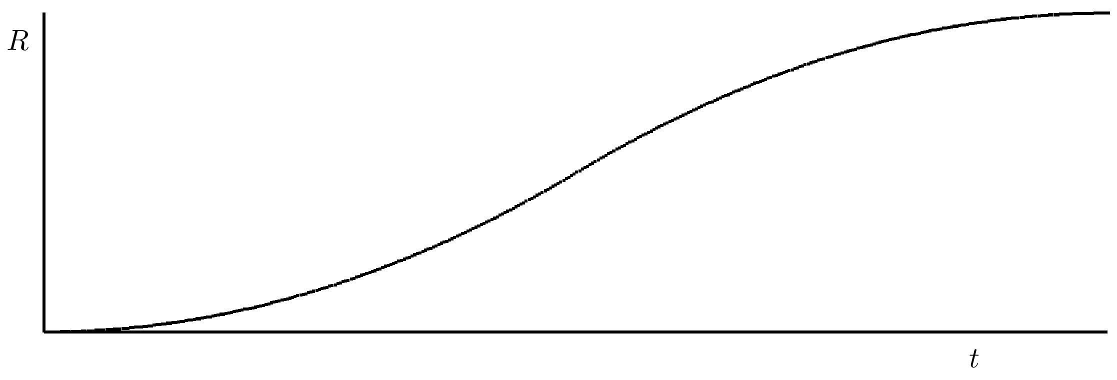

Figure 15: Plot of reliability, R, as a function of time, t.

Example: Assume that the probability that a system will fail during a one hour run is 1% and that the probability of repairing it within an hour after failure is 90%.

Then $\lambda = 0.01$, $\mu = 0.9$, and $\lambda + \mu = 0.91$. We have $p_u = 0.9/0.91 = 0.989$, so we can say that the availability of the system is 98.9%.

11.4 Reliability during development

During the development, we (hopefully) remove defects from the software. Thus we can assume that the reliability improves. In general, we expect (Figure 15):

- The reliability is initially very low (close to zero), because most components will have faults.
- The reliability improves monotonically during development, as faults are removed.
- The reliability tends asymptotically to 1.

As a consequence, we can say that the cost of testing (measured as the time required to detect and correct one fault) must increase (the fewer the faults, the harder they are to find).

As the reliability increases, the cost of releasing the software decreases. To see this, suppose we release the software at an early stage, when there are many faults. There will be numerous complaints from clients, and the developers will spend a lot of time attending to them. Even worse, potential sales will fall as news of the unreliability spreads. Conversely, if we wait until the software is practically perfect before releasing it, there will be few complaints, and the cost of release will be low.

Now consider the total cost, the sum of the cost of testing and the cost of releasing, as a function of time, t. For small t, it is high because of client complaints. For large t, it is large because tests are expensive. Therefore (as economists say), there must be a value of t for which the total cost is minimal; this is the best time to release the software: see Figure 16.

11.5 Testing with UPEDU

In this section, we review the UPEDU approach to testing. Not surprisingly, there is some overlap between this section Section 10.

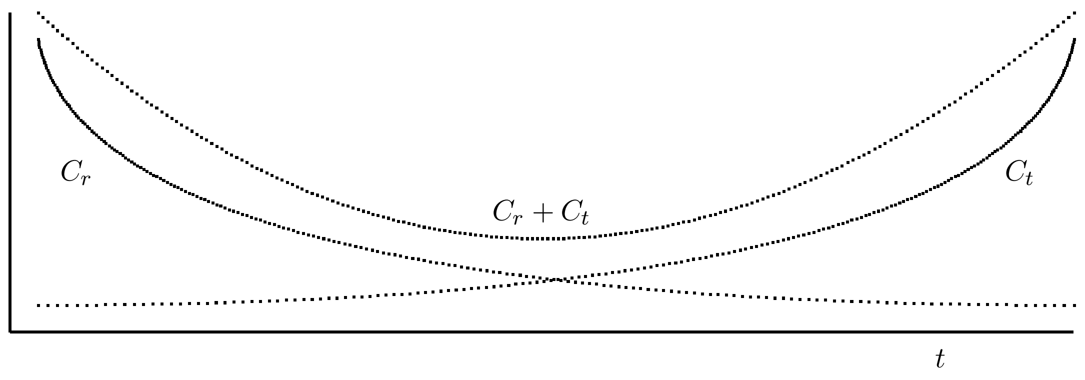

Figure 16: Plot of the cost of releasing, C_r, the cost of testing, C_t, and their sum, against time, t.

UPEDU recognizes two common errors of testing and takes steps to avoid them. The errors are:

- Making testing the last step of software development.
- Assigning the task of testing to a separate "testing team".

Testing is important not only because faults are detected but also because it provides information about the quality of the software. When testing is left until the end, managers usually underestimate the time required for it because they have no idea just how bad the code really is. Tests performed concurrently with development enable progress to be accurately estimated and quality problems to be addressed before they become serious.

Separating development and testing teams encourages casualness amongst programmers. The attitude is: "I'll get it roughly right and the testers can find the bugs for me." A development team that is performing concurrent tests develops a sense of pride: they want the tests to pass first time.

Against this argument, as mentioned above, is the problem that progammers favour their own code. This is a risk, but it can be reduced by emphasizing "egoless programming": a failed test simply means that there is a problem to be solved, not that the programmers's status is diminished.

Iterative development encourages continuous testing. Each increment yields a product that can, and should, be thoroughly tested. Early tests are straightforward because the first increments are quite simple. Later on, the software becomse mre complex, but the tests eveolve in parallel with it, so there are no large jumps. For safety, when a new increment is tested, all previous tests should be applied as well, to ensure that no faults have been injected.

11.5.1 Test Components

Stubs and drivers, introduced in Section 10.6, are examples of test components. There are other kinds of test component.

Stub: A small piece of code that simulates the interface of a component that is not under test but is needed by the component under test.

Driver: A small piece of code that simulates the users of the component under test by exercising its interface.

Zombie: A general term for a minimal driver or stub.

Throwaway: A small piece of code that is written to test a component once and then discarded.

Script: A script, or progam, usually written in a high-level language such as perl or Python, that runs a set of tests and reports the results.

Scaffolding: Any code that is written for the purpose of testing but does not become part of the final product.

Test components must be written very carefully. It can happen that a test succeeds but it is later found that both the test code and the system code were wrong. Test code should be as simple as possible to maximize the probability that it does not contain any errors. When very complex and safety-critical systems are being tested, it may even be necessary to write tests for the test code!

11.5.2 Testing Activities

Testing Environment The first step in testing is to create a ***testing environmment***. In simpole cases, the testing environment might just be a PC on the tester's desk. Even here, it is important to ensure that the PC is set up in the same way as the client's PCs will be set up. Otherwise the tester may report satisfaction only to find that the client cannot run the software because of missing DLLs, for example.

For more complex systems, the testing environment may be very complex. Testing flight control software, for example, might require an expensive simulator. In the final stages, the actual aircraft would be needed.

Test Cases UPEDU associates ***test cases*** with ***use cases***. Since a use case describes a single transaction between a user and the system, it is natural to design tests to check that system behaviour corresponds to use cases. In general, each use case will generate a number of individual tests, depending on the possible paths and outcomes of the use case.

Use cases are only one part of the requirements. Other parts of the requirements may require special testing.

12 Delivery and Maintenance

12.1 Delivery

Although delivery is important in practice, we will not spend a lot of time on it in this course. The main point to note is that the delivery process depends strongly on the kind of system.

- "Shrink-wrap" software is delivered to customers through retail stores. The product must be essentially self-contained because the suppliers cannot afford to spend time with individual clients; the best they can do is provide some kind of assistance via internet or telephone. This implies that: the software should be easy to install, learn, and use; and that all documentation, printed or on-line, comes with the package. The cost of shrink-wrap software ranges from about \$5 to \$5 000.
- Above \$5 000 or so, customers expect additional services. The suppliers may offer: help with installation; on-site acceptance tests; and training courses, at either their own or the customer's site.
- When the software is developed for an individual client, there will probably be a close relationship between the supplier and the client. As we have seen, the client will be involved with the software from an early stage, such as requirements analysis. Delivery will be just one step — albeit an important step — in the software development process. Delivery is not the last step; rather, it signals a transition from development to maintenance.

Software will not be used unless its users understand it and are comfortable with it. "Shelfware" is the name give to software purchased, sometimes at considerable cost, and then never used. To avoid your product becoming shelfware:

- make the software as easy to use as possible;
- write clear and unambiguous user manuals;
- provide on-line help, where applicable;
- write all documentation from the point of view of the user rather than the programmer.

12.2 Maintenance

After software has been delivered, in whatever form, it must be ***maintained***. Studies support the "80–20 rule": over the entire lifespan of a software product, 20% of the effort is in development and 80% is in maintenance.

12.2.1 Varieties of Maintenance

The word "maintenance" suggests lubricating your bicycle or taking an applicance to Mr FixIt to have a part replaced. This kind of maintenance clearly does not apply to software, because code does not get stiff or wear out. Software maintenance is generally divided into four varieties.

Perfective Maintenance improves the system even though it is not failing. The improvements may be internal and invisible to users: for example, a search tree might be replaced by a look-up table to simplify the code without changing performance. Other improvements may be noticeable to users: for example, performance might be improved or short-cut keys provided for frequent operations.

Adaptive Maintenance is concerned with adaptation to changing conditions. If the operating system on which the system runs is upgraded, the system may not work at all or may need to be modified to take advantages of the new OS. In either case, maintenance is required even though there is nothing wrong with the system itself.

Corrective Maintenance responds to faults. When the system fails, maintenance programmers determine the cause of the fault, decide on a strategy for correcting it, and make the correction.

Preventive Maintenance is concerned with preventing failures before they occur. The maintenance programmers may notice weak spots in the code that might cause faults. Preventive maintenance is often carried out in conjunction with corrective maintenance: while correcting a fault, the maintenance programmers examine the code for similar faults and change it to avoid them.

The varieties are listed approximately in order of importance. According to one large study, the relative times spent on these activities are:

Activity	% Effort
Perfective maintenance	50
Adaptive maintenance	25
Corrective maintenance	21
Preventive maintenance	4

12.2.2 The Maintenance Team

After the suppliers have delivered a product, it is essential that they devote some resources to maintaining it. The maintenance team can be of any size from one person to thousands of people.[13]

Responsibilities of the maintenance team include:

- understanding the software;
- becoming familiar with the system documentation;
- answering questions about the way in which the system works (or is supposed to work);
- analysing faults and problems;
- locating the cause of faults in the code and correcting the code;

[13]Exact data is rare, but here is one specific example: in 1990, WordPerfect employed 600 people whose sole responsibility was to answer telephone complaints.

- keeping system documentation up to date (changes to code imply changes to documents);
- restructuring design documents and the corresponding code;
- deleting parts of documents and code that are no longer used;
- adapting existing functions to changing requirements;
- adding new functions;
- managing change.

There are many problems associated with maintenance. Here are brief discussions of a few of them.

Understanding the Software. One study showed that 47% of maintenance programmers' time is spent in understanding the software. This is a natural consequence of several factors:

- Most developers keep their "best" programmers for development, considering them too important for maintenance. Consequently, maintenance programmers are usually not top calibre.
- Newly-hired programmers are often given maintenance tasks before they are allowed to work on development.
- Developers are closely associated with the software for a long time and come to know it well. Maintainers may be given a component with a problem and told to "fix it". It may take them a long time to become familar with the overall structure of the system

Management. There is usually a lot of pressure from managers to get faults repaired rapidly to please the client. Conscientious programmers who spend a lot of time analysing and trying to understand the fault receive less praise than sloppy programmers who find a "quick fix". As the quick fixes accumulate, the software becomes harder to understand and repair. The short-term policy of rapid repair has the long-term consequence of software that is "brittle" — it is poorly structured and breaks under the slightest strain.

Morale. Maintenance programmers often have a lower status than developers. This is unjustified because maintenance programming is in some ways harder than development programming. Nevertheless, the perception that the maintainers are "second tier" programmers is common and has the effect that every maintainer wants to be "promoted" to development.

The morale problem can be handled by a good manager who treats all programmers as equals and who requires everyone to take turns at development and maintenance. This approach has the added advantage that doing maintenance gives the development programmers a better understanding of the importance of clarity, simple solutions, and good documentation.

Longevity. The long life of software creates maintenance problems. A large system may be used for 20 years or more. Moore's Law says that hardware performance doubles about every 18 months. During the 20–year lifespan of the software, the hardware that it runs on can be expected to improve by a factor of $2^{20/1.5} \approx 10\,000$. Design decisions that seemed reasonable during the initial development phase may look stupid in the light of such improvements.

The classic example of a longevity problem was the much-exaggerated "Y2K" problem, introduced by the decision to assign two digits rather than four digits to the "year" field in many programs. Performance, however, was *not* the major factor in the Y2K problem. The decision to use two digits for the year was not made because memory was expensive but because it simply did not occur to the designers that their code would still be running in 1999.

Managing Change. The maintenance team must know, at all times, the exact status of each component of the system. Each time there is a change request for any component, the some or all of the following information should be recorded.

***Synchronization*:** when is the change to be made?

***Identification*:** who will make the change?

***Naming*:** what components of the system are affected?

***Authorization*:** who gave permission for the change to be made?

***Cancellation*:** who can cancel the change request?

***Routing*:** who will be informed of the change?

***Delegation*:** who is responsible for the change after it has been made?

***Valuation*:** what is the priority of the change?

***Authentication*:** (after the change has been made) was the change made correctly?

12.2.3 Software Rejuvenation

The traditional approach to maintenance is that you maintain the system until it becomes unaminatainable — that is, the cost of keeping it running exceeds the benefit of running it. At this time, the system is scrapped and either rewritten from scratch or replaced by a new system designed to different requirements.

A more modern approach is to attempt to ***rejuvenate*** the software. If rejuvenation succeeds, the system becomes easier to maintain and its useful life can be extended. There are various approaches to rejuvenation, some simple and others more drastic.

***Redocumentation*:** the system is analysed thoroughly and a new set of documents describing it is prepared. This does not improve the software in itself, but may make other forms of rejuventation feasible.

***Reverse Engineering*:** the current design of the system is inferred from its code. This may be necessary either because the original design documents have been lost or because the design has changed as a result of extensive maintenance. Software tools for reverse engineering exist.

***Reengineering*:** involves both reverse and "forward" engineering, first to discover how the system works and then to improve its structure without changing its functionality.

***Restructuring*:** the code is modified in various ways to simplify maintenance. For example, very old FORTRAN programs are structured primarily with `IF/GOTO` statements because FORTRAN did not have `WHILE/DO` and `IF/THEN/ELSE` statements. The code can be rewritten using the modern statements, either manually or (better) automatically.

***Refactoring*:** the code is modified as in restructuring, but the changes tend to be larger. For example, related components may be merged, or functionality may be moved from one component to another, more appropriate, component.

***Paradigm Change*:** structured code may be rewritten in an object oriented way, or procedural code may be rewritten in a purely functional way.

13 Case Studies

13.1 The Therac-25 Saga

There was a consortium consisting of Atomic Energy Canada Limited (AECL) and CGR, a French company. The consortium's products included: Neptune; Sagittaire; Therac-6 (a 6 MeV X-ray accelerator consisting of Neptune and a PDP/11[14] controller); Therac-20 (a 20 MeV dual-mode (X rays or electrons) accelerator consisting of Sagittaire and a PDP/11 controller). All of these products were designed to administer controlled doses of radiation to people with cancer.

The Therac-6 and Therac-20 could be operated without the computer. "Industry standard hardware safety features and interlocks" were retained in both machines. Software control was added for convenience; even if the software failed, operations would be restricted by hardware.

Around 1975, AECL developed the Therac-25 using "double pass" technology, in which the energy source is a magnetron rather than a klystron and the acceleration path is folded so as to require less space. The Therac-25 generates electron or X ray beams at up to 25 MeV. High doses permit treatment at greater depth in the body.

Key points about the Therac-25 design:

- Therac-25 was controlled by a PDP/11.
- The software was an essential component of the system, not an optional add-on.
- Therac-25 relied on software correctness for safety: it did not have electrical and mechanical interlocks like the Therac-6 and Therac-20.
- Some of the software from the Therac-6 and Therac-20 was reused.

The first prototype of Therac-25 was produced in 1976. A commercial version became available in 1982. In March 1983 a safety analysis performed by AECL made these assumptions:

- Software errors had been extensively tested with a hardware simulator. Any residual errors were not included in the analysis.
- Software does not degrade due to wear, fatigue, or reproduction.
- Computer execution errors are caused by faulty hardware and random errors induced by alpha particles and electromagnetic noise.

The fault tree obtained from the analysis includes "computer failure". The probability that the computer would select the wrong energy level was estimated to be 10^{-11}. The probability that the computer would select the wrong mode was estimated to be 10^{-9}. The report provides no justification for these numbers and seems to imply that they arise from hardware failure, not software failure.

[14]The PDP/11 was an early minicomputer developed by Digital Equipment Corporation.

13.1.1 Problems.

Five Therac-25s were installed in the USA and six in Canada. Between 1985 and 1987, there were six accidents involving radiation overdoses. The Therac-25s were recalled in 1987. Typical accidents:

- Patient: “You burned me”. Diagnosis revealed one or two doses of radiation in the range 15K to 20K rads. Typical doses are around 200 rads; 1K rads can be fatal. This patient did not die but required extensive surgery.
- The operator activated the Therac-25 but it shut down after 5 seconds with an “H-tilt” error message and displayed “no dose”. The operator pressed “P” (proceed — standard procedure) and the Therac-25 again shut down. This sequence was repeated four times. The patient complained of an “electric tingling shock”. She received between 13K and 17K rads and died four months later.

AECL could not reproduce the failures but suspected a “transient failure in a microswitch” used to determine turntable position.

- The computer reads a 3-bit signal giving the status of three microswitches in the turntable that determines the orientation of the radiation gun. A 1-bit error could produce an ambiguity in the position.
- An additional problem was that a plunger, used to lock the turntable at one of three positions, could be extended when the turntable was far out of position; this would also lead to a false position reading.

AECL modified the software to: read the turntable position correctly in the presence of a 1-bit error; and to check for “in transit” status of switches to provide additional assurance that the switches were working and the turntable was moving correctly.

AECL claimed that the accident rate of the new system was improved by “at least five orders of magnitude” (that is, accident rate should be less than previous accident rate by a factor of 10^5 or more). However, AECL said that it could not “be firm on the exact cause of the accident”.

There were two further studies of the Therac-25:

- Gordon Symonds of the Canadian Radiation Protection Bureau (CRPB) wrote a report on the Therac-25 that mentioned the microswitch problem and design errors in both the hardware and the software. In November 1985, CRPB requested AECL to make four modifications to the Therac-25 based on the CRPB report.
- The Ontario Cancer Foundation (OCF) said that the Therac-25 needed an independent system to verify turntable position and recommended a potentiometer.

AECL made the microswitch changes as described above, but did not comply with the other requirements of the CRPB. The CRPB asked that treatment be terminated after a dose-rate malfunction, but AECL simply reduced the number of retries allowed from five to three. AECL did not install the potentiometer recommended by OCF.

The next report incident occurred in March 1986 at a clinic in Texas. The operator:

- entered the required radiation prescription;
- realized that she had pressed “x” instead of “e” (most treatments are for X rays);
- used the ↑ key to reposition the cursor on the “x” and changed it to “e”;
- typed RETURN several times to confirm the other entries;
- hit “b” (“Beam on”) to begin the treatment. The Therac-25 shut down with the message “Malfunction 54”, described as a “dose input 2” error. The Therac-25 indicated 6 units delivered from a dose of 202 units.
- The operator pressed “P” and the Therac-25 again shut down.
- By this time, the patient had had enough: he felt as though “he had hot coffee poured all over his head” and had had “an electric shock”. The subsequent investigation showed that he had received between 16K and 25K rads. He died five months later.

The Therac-25 was shut down for a couple of days while AECL tested it and found that it could not give patients electric shocks. The Therac-25 resumed operation. In April 1986, another patient at the same clinic reported a feeling like being hit on the side of the face, a flash of light, and a sizzling sound. He died three weeks later.

A physicist at the Texas clinic performed his own tests on the Therac-25. He discovered that rapid data entry to lead to massive overdoses. After practicing data entry, he could turn a 100 rad dose into a 4K rad overdose. AECL claimed at first they could not reproduce the error but, after further trials, discovered that accidental doses of up to 25K rads could be delivered.

Extended analyses followed. In June 1986, AECL produced a corrective action plan (CAP) proposing software modifications to prevent the specific sequence of events noted in Texas. Some electrical safety circuits were also added. AECL claimed that this CAP improved safety by “many orders of magnitude”.

In January 1987, there were problems with a Therac-25 at Yakima Valley Memorial Hospital. A patient given treatment then died in April. AECL decided that this problem was not caused by “hardware alone” but could have been caused by software failure — a different failure from that in Texas.

13.1.2 Diagnosis.

The following causes for accidents were found:

- One problem was caused by an 8-bit counter used to record the execution of

  ```
  SetUpTest.
  SetUpTest reschedules itself automatically and will be called hundreds
  of times during the set-up procedure.  If the operator presses ‘‘set’’
  after exactly 256 executions of
  SetUpTest, the counter is zero and the ‘‘upper collimator position check’’
  is not performed.
  ```

- `The Therac-25 takes time to respond to commands: for example, the bending magnets take up to 8 seconds to move ino position. The Therac-25 software was written as a collection of concurrent processes communicating by shared variables. The conclusion is that some of the problems were due to incorrect updating of shared variables.`

13.2 The Ariane 5 Rocket Failure

On 4 June 1996, an Ariane 5 rocket was launched for the first time. About 30 seconds after lift-off, at an altitude of 3700 metres, the rocket went off course, broke up, and exploded. The cost of the accident was approximately $500 million.

The problem was a software error in the Inertial Reference System (SRI). Computations are performed to align the SRI and should be completed 9 seconds before lift-off. However, the computation is allowed to continue for the first 50 seconds of the flight so that, in the event of a short delay, the launch does not have to be cancelled because the SRI is not ready (it takes several hours to reset).

The SRI computation includes seven conversions that store a 64-bit quantity into a 16-bit word. If the actual value requires more than 16 bits, the hardware raises an exception that must be handled. Careful analysis revealed that, in three of the seven conversions, overflow could not occur, and no exception handler was provided. For the other four cases, exception handlers were provided.

The Ariane 5 failed because one of the three unchecked conversions failed, the hardware raised an exception, there was no handler and — in accordance with requirements — the control computer shut down.

The SRI code was written for the Ariane 4 and was correct for Ariane 4 trajectories. It was not correct for Ariane 5 trajectories and that is why the Ariane 5 crashed.

Peter Ladkin (see web page) diagnoses this as a requirements error:

- The conversion was deliberately left unchecked.
- This was because engineering analysis for the Ariane 4 showed that overflow could not occur.
- The requirements for Ariane 4 were transferred to the Ariane 5 without further analysis.
- The system was tested against Ariane 4 IRS requirements and Ariane 5 Flight Control System (FCS) requirements.
- There was no integrated testing (IRS-FCS).

This seems plausible: a JPL study of safety-critical systems concluded that 90% of failures are due to requirements errors.

Conclusion: avoid such errors by fixing the requirements process and including integrated testing.

Other lessons:

- Good practice includes explicit documentation of data range assumptions. A few programming languages provide this; most don't.

- Where an exception can occur, and a handler is not provided, the documentation should say explicitly "no handler" and, preferably, explain why. A code inspection would then reveal potential problems of the Ariane kind.

13.3 Case Study: RADARSAT Payload Computer Software System

Contractor: Spar Space Systems

Designer: Manh The Nguyen (BCS, Concordia, 1982)

RADARSAT:

- first Canadian remote sensing spacecraft;
- launched in 1995;
- planned operational life: 5 years;
- imaging with Synthetic Aperture Radar (SAR) with
- horizontally polarized C-band microwave radiation
- wavelength 5.6 cm.
- First space craft to implement ScanSAR to obtain wider image
- Orbit: 800 km near-polar
- Resolution: 10 to 100 m
- Image width: (on ground) 45 to 500 km
- Incidence angle: $20°$ to $50°$
- Resolution/image/incidence controlled from ground
- Permits imaging through heavy cloud night and day
- Can measure: soil texture, vegetation conditions, sea-ice concentrations and dynamics, and geological surface structures.

RADARSAT consists of two modules: Bus Module and SAR Payload Module.
Bus Module:

1. Command, Telemetry, and Ranging (CT&R) allows Mission Control System (MCS) to communicate with Payload Module
2. Attitude Control Subsystem (ACS) has yaw, roll, and pitch sensors that enable the Attitude Control Processor (ACP) to maintain altitude accurately
3. Electrical Power Subsystem (EPS) provides regulated power (solar cells and storage batteries)
4. Propulsion subsystem for orbital adjustments

5. Thermal subsystem uses thermostatically controlled heaters to maintain all components within preset temperature limits

SAR Payload Module:

1. SAR Antenna (ANT) transmits selectable beam shapes
2. Receiver (RCVR) detects radar return signal and digitizes it
3. High Rate Tape Recorder (HRTR) stores SAR data
4. X-band Downlink Transmitter (DLTX) modulates digitized SAR data for X-band transmission
5. Low Power Transmitter (LPT) generates coded waveforms for radar pulses
6. High Power Microwave Circuits (HPMC) converts LPT signal to high peak-power pulse for radar
7. Calibration (CALN) subsystem routes signals between SAR components
8. Timing and Data Handling (T&DH) generates all reference frequencies and timing signals
9. Payload Computer System (PLC)
   - controls everything
   - health-monitoring and fault analysis for all subsystems
   - primary Payload status monitoring function
   - interfaces Bus module to MCS

Subsystems in the Bus are controlled directly by Mission Control via the CT&R.

Subsystems in the Payload are controlled by the PLC.

The PLC hardware is based on Spacecraft On-board Modular Microcomputer (SOMM) which consists of modules connected by a proprietary Spar Bus protocol called Unit Bus (UB).

PLC Software:

- Initial Program Load ROM (IPL) uploads operational software
- Control Program monitors spacecraft "health", analyses faults, controls equipment, and executes Tables
- Application Logic, expressed as Tables

These are later described as Computer Software Configuration Items (CSCIs):

- Payload Computer Initialize Software (PCIS) (= IPL) resides in ROM
- Payload Computer Application Software (PCAS) or Control Program

Normal scenario:

- Ground Station generates operational tables based on client's request
- verifies tables with Ground Station simulator
- uploads Tables to satellite
- starts operation

There are two kinds of table:

- One-Shot Table which triggers one or more control operations that are executed continuously
- Sequence Command Table which defines a sequence of One-Shot operations with delays between them

Rationale for this approach:

- Evolution of Control Program continues after launch but at a low rate
- Operational complexity occurs on the ground
- Programmed operations can be verified on the ground by simulation
- Simple integration scheme for the spacecraft

13.3.1 Software Development Methodology

Goal: high quality software at a reasonable cost.
Difficulties:

1. Software solutions are often applied to problems that are new and poorly defined.
2. Discontinuity in input/output flows make mathematical modelling hard.
3. Complexity level of software increases exponentially with the size of the program.

In spite of these problems, software is more economical than hardware.
PCAS was developed by Composite Methodology for Real-Time System Software Development Process (CMRS SDP) developed at Spar by Nguyen (and Bishun).

- cloud represents (uncertain) initial user requirements
- rectangles represent activities
- hexagon represents information storage
- "Circle with arrows" indicates the actual data flow (read/write/update)
- Solid lines indicate the information invocation direction
- dashed lines indicate control flow that triggers the start of each activity

CMRS tries to balance:

- Waterfall Model — start next phase only when current phase is completely understood.
- Spiral Model — initially problem is not understood; repeat analysis – design – coding – testing until problem understood.

CMRS Process:

- Requirements Analysis
  - rapid throwaway prototype (help clients to identify needs and examine feasibility issues)
  - Semantic information model
- Design
- Implementation
- Operation
- Each phase is iterated until the expected model is obtained.
- All phases applied OO Analysis, OO Design, OO Programming based on Ada.

The project preferred OO "composition" to "generalization" (or inheritance):

- timing constraints are more controllable with composition
- behaviour inheritance (required for real-time) is not yet mature (1995)
- hard to systematize class hierarchy identification process
- Ada does not support inheritance

Requirements Analysis Phase

1. Client's requirements gathered in textual form
2. Experts in each domain (Radio frequency, microwave, operations . . . .) interviewed
3. Ward/Mellor Real-Time Structured Analysis (RT/SA) applied to obtain Environment Model consisting of a context diagram and an event list
4. For each input event, Functional Flow and Data Transform (FFDT) methodology applied (an FFDT chart seems roughly equivalent to a "use case")
5. Shlaer-Mellor OOA methodology applied using Cadre Teamwork CASE tool, yielding Entity Relationship Diagram (ERD); Object Communication Diagram (OCD); State Transition Diagram (STD); Data Flow Diagram (DFD); and Data Dictionary (DD). This involves building:
   (a) an Information Model

   (b) a Communication Model
   (c) a State Model
   (d) a Process Model

   The final Requirements Model was then translated into Hatley-Pirbhai notation according to MIL-STD-2167A.

6. Protoype the Requirements Model using Borland C++.
7. Use the prototype to show clients how the system operates.

Design Phase Process:

- Generate and evaluate a set of designs
- Select one design
- Prototype selected design in Ada
- Test prototype in Target Environment
- End when there is a design that satisfies the requirements

Criteria:

- memory utilization
- processing speed
- maintainability
- expandability
- technology and risks of new technology
- schedule
- correctness (only factor not compromised!)

Methodology:

- Based on Booch's OOD methodology
- high level objects assembled from component objects

Prototyping:

- each high level object mapped to an Ada task
- number of tasks reduced by grouping objects not required to run concurrently
- represent each object as an Ada Package Specification
- Identify possible deadlocks, etc., and design countermeasures
- code and test the programs

Implementation Phase

- Formal implementation, testing, and integration of selected design.
- Perform McCabe (complexity) analysis on final source code
- Construct a Software Test Description (STD) from the Software Requirements Specification (SRS) so that all requirements could be veified without knowledge of implementation details
- Conduct formal/informal reviews with/without clients

Metrics

PCAS code size	10,150 semicolons
RAM	105Kw used out of 192Kw available
Ada tasks	11
Subroutines	278
FQT test cases	151
Analysis phase	17 months
Implementation phase	10 months
Testing	6 person-months
Designs	7
Integration defects	16
Requirements defects	1

Lessons Learned

- Some drawbacks of Shlaer-Mellor methodology identified
- OOA methodology was simple and straightforward
- Object identification was systematic (because SA and FFDT are well-understood, mature methodologies)
- Cadre Teamwork modelling notation for OOA facilitated client/developer communication
- Design process produced a high quality design
- Ada provided a close mapping between design constructs and final code (task synchronization features?)
- Phase transitions were smooth
- Clients attended most code walkthroughs and structured inspections — few surprises!

14 Industrial Practices

14.1 Measurement: COCOMO

COCOMO (COnstructive COst MOdel) is a system for estimating software costs. Measurement techniques and cost estimation are discussed fully in other courses; the brief introduction provided here is intended to help you to understand Question 1 of Assignment 4.

COCOMO was originally described by Barry Boehm in 1981 (*Software Engineering Economics*, Prentice Hall). Since 1981, it has been extensively refined; the current version is called COCOMO II.

The simplest form, Basic COCOMO, is based on the formula

$$E \quad = \quad bS^c$$

in which E denotes *effort*, S denotes the *estimated size* of the project (this can be measured in KLOCs (thousands of lines of code), function points, or some other way), and b and c are constants that depend on the kind of project. The following table shows suitable values for b and c for the three main kinds of project.

- An *organic* project is developed by a small, experienced team working in a familiar environment.
- An *embedded* project is part of a system with a highly constrained environment, such as air traffic control, that is developed by an experienced team.
- A *semidetached* project is intermediate between organic and embedded.

Project type	b	c
organic	2.4	1.05
semidetached	3.0	1.12
embedded	3.6	1.20

The next table shows the effort for projects of various sizes and kinds.

KLOC	organic	semidetached	embedded
1	2.4	3.0	3.6
10	26.9	39.6	57.1
100	302.1	521.3	904.2
1000	3390.0	6872.0	14333.0

Intermediate COCOMO takes into account a number of factors other than the kind of project and the number of lines of code. They are based on fifteen or so attributes called *cost drivers* because they are assumed to affect productivity and hence costs. The COCOMO model defines a set of factors for each cost driver. Here is one example:

Cost driver	very low	low	nominal	high	very high	extra high
Product complexity	0.70	0.85	1.00	1.15	1.30	1.65

We start by estimating the complexity of the product. If we think it is "very low", then we multiply the effort estimate by 0.70; if we think is is high, we multiply the cost estimate by 1.15; and so on. Obviously, we are guessing to some extent but, as we gain experience, our guesses should improve. Since most companies produce a limited number of kinds of software, their guesses can eventually become quite accurate.

Here are the other cost drivers:

Product attributes: reliability required; database size; product complexity.

Computer attributes: execution time constraints; main storage constraints; virtual machine volatility; computer turnaround time.

Personnel attributes: analyst capability; applications experience; programmer capability; virtual machine experience; programming language experience.

Project attributes: use of modern programming techniques; use of software tools; required development schedule.

The basic formula becomes

$$E = bS^c\, m(\mathbf{X})$$

where $\mathbf{X}$ is a vector derived from the cost drivers and m is a weighting function.

To use COCOMO properly, you have to know how to choose values for all the cost drivers. There is not space or time to do that here, so we just note the following for personnel experience:

Experience	months	weight
Extra low	less than 3	1.29
Very low	3 to 5	1.29
Low	5 to 9	1.13
Nominal	9 to 12	1.00
High	12 to 24	0.91
Very high	24 to 48	0.82
Extra high	more than 48	0.82

Detailed COCOMO uses the same cost drivers but splits them up according to development phases or *stages*. During the first stage, *application composition*, complexity is measured in *object points*. During the second stage, *early design*, complexity is measured in *function points*. During the last stage, *postarchitecture*, complexity is measured in LOCs.

Object points have little to do with object oriented programming: an object point is a screen (or window in modern systems), a report, or a software component (usually a function or procedure). *Function points* are like object points but they take a number of additional factors into account: the number of inputs and outputs; the number of links between components; and so on.

14.2 Programming Languages and Notations

The following are very brief sketches of some of the programming languages used in the software industry.

C++:

- inherits faults of C
- complex
- subtle bugs
- preprocessor
- multiple inheritance
- concurrency in libraries only
- syntax is not context-free
- hard to develop software tools
- memory management is hard
- open

Smalltalk:

- pure object oriented
- automatic memory management
- slower than C++
- closed
- quasi-concurrency
- single inheritance
- used on Wall Street

Eiffel:

- industrial strength
- automatic memory management
- faster than C++
- open
- multiple inheritance
- secure concurrency

- complete development methodology
- few users (?)
- few vendors

Java:

- C syntax
- interpreted — slow
- platform independent
- single inheritance
- automatic memory management
- insecure concurrency
- badly design (but rapidly changing) libraries
- acceptance growing for small applications

Ada:

- not fully object oriented
- sound concurrency
- no automatic memory management
- used mainly for aerospace and "defence"
- good development tools
- good for Software Engineering

Common LISP Object System (CLOS):

- fully object-oriented
- multiple inheritance (with ordered superclasses)
- multi-dispatching
- before/after functions
- extensible
- automatic memory management
- slower than C++
- good prototyping language

UML

- Grady Booch wrote *Object Oriented Analysis and Design with Applications* (1991, revised edition, 1994);
- James Rumbaugh designed Object Modelling Technique (OMT), first presented at OOPSLA in 1987; and
- Ivar Jacobson published *Object-Oriented Software Engineering* in 1992 and introduced "use cases" to object oriented software development.
- David Harel introduced *statecharts.*

- A diagramming notation should be simple and precise. UML is quite complex: the "Notation Summary" occupies 60 pages and describes numerous symbols, keywords, and fonts. This is partly due to the origins of UML as a "union" of three separate notations.
- Because of its complexity, it is hard to use UML in a casual way. Like a programming language, you have to use it intensively for a significant period of time to become familiar with all of its details.
- Also because of its complexity, it is impractical to use UML as a hand-written notation. To use it effectively you need software that allows you to draw and modify diagrams and to check properties of diagrams.

- Much of UML is defined rather vaguely. There have been many attempts to provide a formal semantics, but none has been very successful so far.
- There is a tremendous variety of software. It is unlikely that any *single* notation will be effective for all applications. Yet UML claims to be a universal notation, suited to all conceivable problems.
- Most of the enthusiasm about UML comes from higher management. People who have to use it are less enthusiastic about its capabilities.

14.3 Theory and Practice

The Role of Theory

- Mathematics: existence and continuity of functions; existence of integral transforms; completeness of functional bases.
- Electrical engineer: obtain steady state response from Fourier transform and transient response from Laplace transform.

- Mathematics: completeness and decidability of logics; proof theory; model theory.
- Software designer: practical ways of applying logic???

Software Engineering:

- Much discussion about "process" because there is no general technique for developing software.
- Specific techniques will emerge (some already have) for particular applications.
- Software metrics = which beans do you count?
- The programming languages, development tools, and maintenance tools that we currently use are inadequate for SE.

Doing it alone:

Some highly talented programmers have always insisted that the best software is written by a highly talented programmer rather than by a team (quotes below). This is probably true, but Software Engineering is about software projects that are too large for one programmer, however highly talented, to complete in a reasonable time.

"The first step in programming is imagining. Just making it crystal clear in my mind what is going to happen. Once I have the structure fairly firm and clear in my mind then I write the code." Charles Simonyi (Bravo, Word, Excel).

"You have to simulate in your mind how the program is going to work." Bill Gates (BASIC).

"I believe very strongly that one person . . . . should do the design and high-level structure. You get a consistency and elegance when it springs from one mind." John Page (PFS:FILE).

"I can do anything that I can make an image of in my head." Jonathan Sachs (Lotus 1–2–3).

"To do the kind of programming I do takes incredible concentration and focus . . . . keeping all the different connections in you brain at once." Andy Hertzfeld (MacOS)

Above a certain level of complexity, however, "having it all in your head" doesn't work. That level depends on the designer, and the designers quoted are clearly above average, but some systems are too complex for any one person to "have in the head". This implies that we must have a *notation* for design.