
Notes for COMP 354 Software Engineering

Peter Grogono

`grogono@cs.concordia.ca`

December 1995

Department of Computer Science, Concordia University
1455 de Maisonneuve Blvd. West, Montréal, Québec, H3G 1M8

Contents

1	What is Software Engineering?	6
2	The Course and the Project	7
2.1	The Course	7
2.2	The Project	8
3	Software Development	10
3.1	The Waterfall Model	10
3.2	Requirements Analysis (SRS)	10
3.3	Design	11
3.4	Implementation	11
3.5	Delivery and Maintenance	12
3.6	Software Tools	12
3.7	Software Engineering Concerns	13
4	The Software Requirements Document	13
4.1	Writing Requirements	15
4.2	Summary	15
5	Definitions, Qualities, and Principles	16
5.1	Definitions for Software Engineering	16
5.2	Software Qualities	16
5.3	Software Engineering Principles	18
6	Process Models	21
6.1	Waterfall Model	21
6.2	Evolutionary Model	22
6.3	Prototypes	22
6.4	Spiral Model	24
6.5	Assessment of Models	24

7 Design	25
7.1 Overview of Design	25
7.1.1 Design Documentation	25
7.1.2 Architectural Design	25
7.1.3 Module Interface Specifications	25
7.1.4 Internal Module Description	26
7.2 Remarks on Design	26
7.3 Varieties of Architecture	27
7.3.1 Hierarchical Architecture	28
7.3.2 Layered Architecture	28
7.3.3 General Architecture	28
7.3.4 Event-Driven Architecture	28
7.3.5 Subsumption Architecture	29
7.4 Designing for Change	30
7.5 Module Design	31
7.5.1 Language Support	31
7.5.2 Examples of Modules	31
7.5.3 A Recipe for Module Design	32
7.6 Design Notations	33
7.7 Design Strategies	33
7.7.1 Functional Design	33
7.7.2 Structured Analysis/Structured Design (SA/SD)	34
7.7.3 Jackson Structured Design (JSD)	35
7.7.4 Design by Data Abstraction	35
7.8 Object Oriented Design	36
7.8.1 Object Oriented Programming	36
7.8.2 Responsibility-Driven Design	42
7.9 Functional or Object Oriented?	44
7.9.1 Functional Design (FD)	44
7.9.2 Object Oriented Design (OOD)	44
7.10 Writing MIS and IMD	44
7.10.1 Module Interface Specifications	45
7.10.2 Internal Module Designs	45

<i>CONTENTS</i>	4
8 Formal Specification Techniques	48
8.1 Introduction to Z Notation	48
8.2 Advantages and Disadvantages of Formal Specification	54
9 Validation and Verification	54
9.1 Varieties of Testing	55
9.2 Designing Tests	56
9.2.1 Guidelines for Black Box Testing	57
9.2.2 Guidelines for White Box Testing	57
9.3 Stages of Testing	59
9.4 Testing Strategies	59
9.4.1 Top-down Testing	59
9.4.2 Bottom-up Testing	60
9.5 Testing Procedures	60
9.5.1 When do we stop?	61
9.6 Preparing Test Cases	62
10 Reviews, Walkthroughs, and Inspections	62
10.1 Experience Reports	64
11 Software Metrics	64
11.1 Measurement Theory	65
11.2 Lines of Code (LOCs)	66
11.3 Software Science	67
11.4 Cyclomatic Complexity	68
11.5 Function Points	69
11.6 Dos and Don'ts for Metrication	70
11.7 Summary	70
12 Cleanroom Software Engineering	71
12.1 The Cleanroom Process	71
12.2 Box Structure	71
12.3 Functional Verification	73
12.4 Statistical Testing	73
12.5 Results	74

13 Object Oriented Development	75
13.1 Analysis	76
13.1.1 System Object Model	79
13.1.2 Interface Model	79
13.2 Design	82
13.2.1 Object Interaction Graphs	82
13.2.2 Visibility Graphs	83
13.2.3 Class Descriptions	84
13.2.4 Inheritance Graphs	84
13.2.5 Principles of Good Design	84
13.3 Implementation	85
13.3.1 Coding	85
13.3.2 Performance	85
13.3.3 Review	86
13.3.4 Error Handling	86
13.3.5 General	87
13.4 Reuse	87
13.5 Other Process Models	88
13.6 Advantages and Disadvantages of Object Oriented Development	89
14 Miscellaneous Topics	90
14.1 Software Tools	90
14.2 Computer-Aided Software Engineering (CASE)	90
14.3 Single-Point Control	90
14.4 Standards	90
15 Case Studies	90
15.1 RADARSAT Payload Computer Software System	90
15.2 A Software Disaster	93
16 Bibliography	93

1 What is Software Engineering?

What is the difference between “software engineering” and “programming”? Why is COMP 354 not called “Advanced Programming”? What is software engineering? To answer the question, consider a sequence of projects of increasing size.

1. You write a program for yourself. You:
 - know what you want;
 - make it up as you go along;
 - throw it away.
2. You write a program for someone else.
 - Ex: assignment for professor.
 - Client/supplier, customer/programmer, one on one.
 - You know what the client wants.
 - You must satisfy the client.
 - Throw it away afterwards.
3. Consider a project that is too large to manage yourself.
 - Someone must **organize** the work.
 - You need a **team**.
 - Team members must **co-operate**.
 - It helps to design interfaces and to write code that meets interface specifications.
 - May not throw it away.
4. A small industrial contract.
 - Need several programmers.
 - There will be several users.
 - Enhancements and corrections will be required.
 - People on the project will have to **communicate**.
5. A large software project (data from a telecom project)
 - 10^7 LOC (lines of code);
 - 10^4 work-years development time;
 - 20% changes/year: corresponds to 2×10^6 LOC changes installed while running;
 - 2K sites with a different version at each site;
 - 10^4 make files;
 - 3–4 days for complete compile (“build”).

The course project is roughly at level 3.

Trends with increasing software size:

life time	1 day	→ 25 years or more
maintenance	0%	→ 80% of overall cost
people	individuals	→ hordes

Stuart Feldman's technology/size classification:

LOC	Discipline
10^3	Mathematics
10^4	Science
10^5	Engineering
10^6	Sociology — “crowd control”
10^7	Politics — “projects need employees”

Brooks (1978) says:

“The distinctive concerns of software engineering are (in 1975 and 1995):

- how to design and build a set of programs into a **system**;
- how to design and build a program or a system into a robust, tested, documented, and supported **product**;
- how to maintain intellectual control over **complexity** in large doses.”

2 The Course and the Project

2.1 The Course

COMP 354 consists of lectures (3 hours/week), tutorials (1 hour/week), and labs (2 hours/week). Some lab sessions include demonstrations.

Evaluation You must obtain a passing mark (approximately 50%) in both quizzes and project work. There will be three quizzes, worth 15% each, in weeks 4, 8, and 12. The project is worth 55%.

Quiz 1	15%
Quiz 2	15%
Quiz 3	15%
Project ϕ 1	15%
Project ϕ 2	15%
Project ϕ 3	15%
Project — individual work	10%
<hr/> Total	<hr/> 100%

Principal Text *Fundamentals of Software Engineering*. Ghezzi, Jazayeri, and Mandrioli. Prentice Hall 1991. (This book is referred to simply as “Ghezzi” for the remainder of these notes.)

Reference Texts are reserved in the library: see course handout.

Course Notes Greg Butler's lectures are available at the Copy Centre in the Hall Building. Ask the Copy Centre for **everything** concerning COMP 354 — they should also have notes on LATEX, UNIX, X windows, etc.

Computing Resources We use “greeknet” (alpha, beta, etc); these are SUN SPARC-stations running UNIX. The workstations are in H-962; you can access them from other sites, from home, etc. Only a small proportion of the project work can be done on non-UNIX platforms.

We use **electronic mail** to communicate with other individuals and teams. There is a BBS called `comp354`, to which you should subscribe, and a shared account `~comp354`.

To subscribe to the BBS, make sure you do not have a directory called `Mail/commp354` and enter

```
bbsub comp354
```

To incorporate new postings (signalled by the “BB” icon when you login), enter

```
bbinc comp354
```

To view the first new message, enter

```
show +comp354
```

You can use other MH commands for BBS as usual: `next`, `prev`, `scan`, `forw`, `repl`. You can use `rmm` but it is not really necessary because BBS messages are not stored in your account.

The programming language for the course is C with X libraries. The documentation tool is LATEX. These are requirements, not options.

2.2 The Project

The project is the most important part of the course. It is designed to provide experience both in managing the development medium-scale program and in working with other people.

Project Organization The class is split into **groups**. Each group completes the project independently from the other groups.

A group should have between 9 and 12 members; the optimal size is 10. A group consists of a **coordinator** and three **teams**. A team consists of a **team leader** and 2 to 4 **team members**.

Preferably, the coordinator should have some experience of organizing and administering or, at last, should be a person that the members of the group respect. Similarly, team leaders should be able to manage their teams.

The teams in a group are called System Requirements Specification (SRS), Design, and Implementation, Validation, and Verification (IVV). Each team has a period of heavy work: for SRS, weeks 2–5; for Design, weeks 5–9; and for IVV, weeks 9–13. Teams may “lend” members to other teams to balance the workload.

The instructor is both the “client” and a “manager”. It is a group’s responsibility to determine what the client wants. When problems arise, coordinators may consult the manager.

Instructions go “down”: the instructor instructs the coordinator, who instructs the team leaders, who instruct the team members. Reports go “up”.

Teams will meet outside class and lab times; the members of each team must agree on a convenient time and place for meetings. Occasionally, it may be useful to have a meeting of the entire group.

You will be asked to evaluate both your own work and the work of other members of your group. This is difficult, especially for coordinators.

You may spend more time arguing than doing “technical” work — that is a normal part of working with other people.

The key is to encourage **co-operation** and avoid **competition** (unlike other courses). The work must be shared as equally as possible by everyone.

Project Requirements This is a general outline of the project. Details will be provided in the first tutorial (Tuesday, 12 September).

General Requirements

- All programs must be written in C. (Or possibly C++, with my permission — but C++ and X do not mix well.)
- The product must run under UNIX using X windows.

Specific Requirements The project involves extending an existing software product (produced by one of last year’s best groups).

You will be given documentation and code for a Graph Editor (GE). The GE provides facilities for drawing, moving, and labelling nodes and arcs in a window.

The project is to extend GE into a FSM Analyzer (FSM = Finite State Machine). Nodes correspond to states and arcs correspond to transitions. The user can create a FSM by manipulating graphical objects and can then animate the FSM by inputting state transitions.

3 Software Development

3.1 The Waterfall Model

The Waterfall Model (WM) is an early **lifecycle model** (Royce 1970). (*William Royce*, software engineer, died August 1995.) WM is based on engineering practice; it works well if the requirements are well-understood and do not change — this rarely happens in practice. The Waterfall Model is important in the same sense as Newton's Theory of Gravity: it's wrong, but you can't understand relativistic gravitation if you do not understand Newtonian gravitation.

A software project is divided into phases. There is feedback from each phase to the previous phase, but no further.

1. Requirements Analysis (SRS team, weeks 1–5).
2. Design and Specification (Design team, weeks 5–9).
3. Coding and Module Testing (IVV team, weeks 9–11).
4. Integration and System Testing (IVV team, week 12).
5. Delivery and maintenance (Everybody, week 13).

WM is **document driven**. Requirements analysis yields a document that is given to the designers; design yields a document that is given to the implementors; implementation yields documented code.

3.2 Requirements Analysis (SRS)

Write a System Requirements Document (SRD) that describes in precise detail what the customer wants.

Find out what the client wants. This should include what the software should do and also:

- likely and possible enhancements;
- platforms (machines, OS, programming language, etc);
- cost;
- delivery schedule;
- terms of warranty and maintenance;
- user training.

DoD Report on software engineering practices (Glass 1991, pages 17–22). Requirements are hard. The big problems are managerial, not technical. Specific recommendations:

- Use evolutionary [requirements] acquisition to reduce risk.

- Remove any dependence on the assumptions of the “waterfall” model.
- Provide the ability to do rapid prototyping in conjunction with users.

Note: The SRD does **not** say how the software works.

Major deliverable: SRD.

3.3 Design

Design a software system that satisfies the requirements. Design documentation has three parts:

Architecture Document (AD) An overall plan for the components of the system.

The AD is sometimes called High-level Design Document (HDD).

Module Interface Specifications (MIS) Description of the services provided by each software module.

Internal Module Design (IMD) Description of how the module implements the services that it provides.

In the AD, each module is a “black box”. The MIS describes each module as a black box. The IMD describes each module as a “clear box”.

Each requirement in SRD should be traceable to a feature in the design documents.

Depending on the project and the requirements, it may be necessary to create a formal (mathematical) **specification**. Alternatively, selected critical parts of the system may be formally specified.

Major deliverable: AD, MIS, and IMD.

3.4 Implementation

Implement and test the software, using the design documents. Testing requires the development of **test plans**, based on SRD, which must be followed precisely. Roughly: for each requirement, there should be a test.

Warning: the IVV team could work independently of the Design team, starting before week 9 and ignoring thw work of the Design team. This might yield a good product but would not lead to a good evaluation.

Major deliverable: source code and test results.

3.5 Delivery and Maintenance

The product consists of all the documentation generated and well-commented source code.

Maintenance (not part of the term project!) includes:

Correcting: removing errors;

Adapting: for a new processor or OS or to new client requirements;

Perfecting: improving performance in speed or space.

Maintenance is 60% to 80% of total budget for typical industrial software. This implies the need for **high quality work in the early stages**. Good documentation and good coding practice make maintenance easier, cheaper, and faster.

Reverse engineering is a rapidly growing field. Many companies have MLOCs of **legacy code** developed 20 or 30 years ago in old languages (e.g. COBOL, FORTRAN, PL/I) with little supporting documentation. Tools are used to determine how it works.

Delivery also includes customer assistance in the form of manuals, tutorials, training sessions, response to complaints.

The requirements of high quality are the same as the requirements for maintainability. Maintenance is the solution, not the problem (Glass 1991, pages 49–51 and 53–56).

3.6 Software Tools

Software tools are an important part of software development. The larger the project, the more important it is to use tools in its development.

- Editor.
- Compiler and Linker.
- Version control system (RCS).
- Software measurement (DATRIX).
- Specification checkers (OBJ3, Larch Prover).
- Test generators.
- Graph editors for DFDs and other diagrams.
- CASE tools for integrated development.¹
- Browsers, library managers, etc.

¹Reference to CASE'95 Proceedings.

3.7 Software Engineering Concerns

We have a perfect record on software schedules — we have never made one yet and we are always making excuses. (General Bernard Randolph, commander of USAF Systems Command)

An example of a USAF contract: the Douglas C-17 cost \$500M more than planned. There were 19 on-board computers, 80 microprocessors, and 6 different programming languages. GAO Report:

The C-17 is a good example of how *not* to approach software development when procuring a major weapons system.

The report also criticized the Waterfall Model. (Cite CACM.)

Software engineering is concerned with the following, amongst other things.

Products: software created; quality.

Paper: internal documentation; user manuals.

Processes: How is software created? How is quality evaluated and ensured?

Power Tools: editors, etc. See above.

People: technical, management, and social skills.

Are you a Software Engineer? You need:

- a thorough knowledge of programming, including several programming languages, paradigms, data structures, algorithms, . . .
- communication skills, including reading, writing, conversing, presenting, working with team members, managers, clients, . . .
- understanding of the application domain and jargon.

4 The Software Requirements Document

The SRD is not covered well in Ghezzi although there are many references to “requirements”.

The SRD 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.

A well-written SRD will reduce development effort by avoiding (expensive) changes later, in design and implementation phases.

Notation: \diamond = good, \clubsuit = bad.

Characteristics of a good SRD:

- The SRD should define all of the software requirements **but no more**. In particular, the SRD should not describe any design, verification, or project management details.
 - \clubsuit “The table is ordered for binary search.”
 - \clubsuit “The table is organized for efficient search.”
 - \diamond “The search must be completed in time $O(\log N)$.”
- The SRD 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 SRD style.
 - Careful usage.
 - \clubsuit “The checksum is read from the *last* record.”
Does “last” mean (a) at end of file, (b) most recently read, or (c) previous?
 - \diamond “. . . from the final record of the input file.”
 - \diamond “. . . from the record most recently processed.”
- The SRD 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 640Kb”), and external interfaces. The SRD must define the response of the program to all inputs.
- The SRD must be **verifiable**. A requirement is *verifiable* if there is an effective procedure that allows the product to be checked against the SRD.
 - \clubsuit “The program must not loop”
 - \clubsuit “The program must have a nice user interface.”
 - \diamond “The response time must be less than 5 seconds for at least 90% of queries.”
- The SRD must be **consistent**. A requirement must not conflict with another requirement.
 - \clubsuit “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 SRD 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 SRD must be **traceable**.
 - The origin of each requirement must be clear. (Implicitly, a requirement comes from the client; other sources should be noted.)
 - The SRD may refer to previous documents, perhaps generated during negotiations between client and supplier .
 - The SRD must have detailed numbering scheme.
- The SRD must be usable during the later phases of the project. It is **not** written to be thrown away! A good SRD should be of use to maintenance programmers.

The SRD 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.

4.1 Writing Requirements

- Include input/output specifications.
- Give representative, but possibly incomplete, examples.
- Use models: mathematical (e.g. regular expressions); functional (e.g. finite state machines); timing (e.g. augmented FSM).
- Distinguish mandatory, desirable, and optional requirements.
 - ◇ “The user interface must use X Windows exclusively.”
 - ◇ “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.”
 - ◇ “Sound effects are desirable but not required.”
- Anticipate change. Distinguish what should not change, what may change, and what will probably change.
 - ◇ “The FSM diagram will contain at least nodes and arcs.”
 - ◇ “The software may eventually be required to run on machines with the EBCDIC character set.”

4.2 Summary

- What, not how.
- No data structures or modules.
- The product, not the process.

5 Definitions, Qualities, and Principles

5.1 Definitions for Software Engineering

Product — what we are trying to build.

Process — the methods we use to build the product.

Method — a guideline that describes an activity. Methods are general, abstract, widely applicable. Example: top-down design.

Technique — a precise rule that defines an activity. Techniques are precise, particular, and limited. Example: loop termination proof.

Tool — a mechanical/automated aid to assist in the application of a methodology. Examples: editor, compiler, . . .

Methodology — a collection of techniques and tools.

Rigor — careful and precise reasoning. Example: an SRD should be rigorous.

Formal — reasoning based on a mechanical set of rules (“formal system”). Example: programming language, predicate calculus.

Use **rigor** as much as possible. Use **formality** when suitable tools are available (compilers, parser generators, proof checkers, . . .

A comparison between formality and rigor:

Formal:

$$\begin{array}{lll} \textit{Command} & \longrightarrow & \textit{Action} [\textit{Parameters}] ";" \\ \textit{Action} & \longrightarrow & \text{"go"} \mid \text{"stop"} \\ \textit{Parameters} & \longrightarrow & \textit{Number} \{ ", " \textit{Number} \} \end{array}$$

Rigorous:

A \langle command \rangle is either "go" or "stop" optionally followed by one or more parameters. Each parameter is a number. Parameters are separated by commas (","). The \langle command \rangle is terminated with a semicolon (";").

5.2 Software Qualities

Ghezzi 2.2–2.4. “Sciences” consist of precise definitions of basic concepts and deductions from the definitions. “Software Engineering” uses a large number of rather vague words, indicating that it is not — yet — a science.

Good software is:

Correct. The software performs according to the SRD. The SRD may be too vague (although it should not be) — in this case, conformance to a specification is needed.

Reliable . This is a weaker requirement than “correct”. E-mail is reliable — messages usually arrive — but probably incorrect.

Robust. The software behaves well when exercised outside the requirements. For example, software designed for 10 users should not fall apart with 11 users.

Performance. The software should have good space/time utilization, fast response times, and the worst response time should not be too different from the average response time.

Friendly. The software should be easy to use, should not irritate the user, and should be consistent.

◇ “The screen always mirrors the state.”

◇ “One key — one effect. E.g. F1 for help.”

Verifiable. A common term that is not easily defined; it is easier to verify a compiler than a word-processor.

Maintainable.

- Easy to correct or upgrade.
- Code traceable to design; design traceable to requirements.
- Clear simple code; no hacker’s tricks.
- Good documentation.
- Simple interfaces between modules.
- More later.

Reusable. (Current buzzword!) Programmers tend to re-invent wheels. We need abstract modules that can be used in many situations. Sometimes, we can produce a sequence of products, each using code from the previous one.

Example: accounting systems.

OO techniques aid reuse.

Portable. The software should be easy to move to different platforms. This implies few OS and hardware dependencies. Recent developments in platform standards (PCs, UNIX, X, . . .) have aided portability.

Portability and efficiency are incompatible. Highly portable systems consist of many layers, each layer hiding local details. Recent achievements in portability depend on fast processors and large memories.

Interoperable. (Another current buzzword!) The software should be able to cooperate with other software (word-processors, spread-sheets, graphics packages, . . .).

Productivity. Ghezzi, pages 32–33.

Timeliness. Ghezzi, pages 33–34.

Visibility. All steps must be documented.

Maintainer’s questions (“Why does the screen go blank when I do this?”) must be answerable from the SRD.

See also Ghezzi 2.3 *Qualities for Particular Applications* and 2.4 *Measurement of Quality* (we will do more later).

5.3 Software Engineering Principles

There are a number of general principles that apply to many areas, including aspects of software engineering.

Separation of Concern This is a very important principle that has many applications in Software Engineering. Frequently, we have a large, complex problem with many inter-related aspects. To deal with such problems, **separate concerns** and look at each concern separately.

Examples:

- Specification (what) *vs* implementation (what). This applies to program modules, procedures, functions, statements, ADTs, . . .
- Correctness *vs* efficiency. Get it working first, then attend to performance. (But always choose the best algorithm that you know!)
- Functional *vs* non-functional requirements.

Examples of non-functional requirements:

 - Where are the terminals?
 - How many terminals?
 - What furniture do we need?
 - Where is the power outlet?
- Editor design: text manipulation *vs* display.
- Application code *vs* user interface code.
- In-memory processing *vs* disk access. A typical sequence is logical data → blocked data → disk buffers → disk controller.

In DOS: file name (user level) → file descriptor (DOS level) → disk address (BIOS level) → driver routines.
- Form *vs* content in word processing. Example: Latex style files.

Modularity Every large system must be divided into **modules** so we can understand it.

Each module performs a **set of tasks**.

Modules may be nested. Nesting suggests a tree-structure, but this is misleading. Usually, modules are constructed on **layers**, with each layer using the modules below it, but not above it. The implied topology is a **directed, acyclic graph** or **DAG**.

The important attributes of modules are **cohesion** and **coupling**.

Cohesive is a property of modules. A cohesive module provides a small number of closely related services.

◇ “Create, add entry, find entry, delete entry.”

♣ “store a variable in the array, update the screen, and sound an alarm after 5 p.m.”

Coupling is a property of systems. Modules are **loosely coupled** if the communication between them is simple.

◇ “Modules form clusters with few interconnections.”

♣ “One modules needs all of the others.”

♣ “Every module needs every other module. Cf. spaghetti code.”

The **goal** is: **high** cohesion and **low** coupling.

Language Support for Modularity

Standard Pascal. Almost none. Nested procedures provide hierarchical structure.

Turbo Pascal. Units provide quite good modularity.

C. Separate compilation helps, but all responsibility is left to programmer.

C++. Classes provide modularity, but C++ is still file based.

Modular Languages. These emerged during the 70s: Ada, Modula-*n* and provide “true” modularity.

Object oriented languages. The “pure” OOLs provide classes, which are a good basis for modularity.

Abstraction It is sometimes best to concentrate on **general aspects** of the problem while carefully removing **detailed aspects**. Cf. what *vs* how in “Separation of Concern”. Examples:

- History of circuit diagrams.
- Manual transmission (gear level, clutch) abstracts to automatic transmission (gear lever, rarely used).

- “How” abstracts to “what”.
- Memory addresses abstract to variable names.
- Code performing a task abstracts to a procedure name.
- Bit string (C) abstracts to set (Pascal).
- In **concurrent programming**, we abstract away from linear time: given events E and E' , we need to know only which must occur first. If delay is important, we have abstracted too much!
- Mathematics is the language of abstraction! Sets, functions, relations, graphs, trees, logic . . . are used because they provide models of things that we need.

An example of mathematical abstraction: consider the undergraduate curriculum:

- Course: a set, C .
- Pre/co-requisites $\subseteq C \times C$.
- Credits: $C \rightarrow \mathcal{R}$.
- Degree: $\mathcal{P}(C) \rightarrow \mathcal{B}$.

Anticipating Change This has already been covered in our discussion of the SRD (these notes 4).

General rule: write all documents and code under the assumption that they will subsequently be corrected, adapted, or changed by somebody else.

- Detailed documentation, including comments in code.
- Many cross-references.
- No dependencies, especially hidden dependencies.
- Distinguish “fundamental assumptions” (may be hard-wired into the code) from “likely changes” (should not be hard-wired).

Fundamental assumptions: ASCII character set; fixed-width font; 2D display (unlikely to become 3D display).

Likely changes: improved search/replace commands; edit more than one file; wider selection of graphical objects.

Generality A general solution is often:

- not much harder to write than a special-purpose solution;
- more likely to be re-used; and
- perhaps a little less efficient.

Examples:

- In language processing, we can write a **special** parser for each grammar that we encounter or write a **general** program that constructs a parser from a grammar.

- Spreadsheets generalize specific accounting problems. They can be applied to an even wider class of problems than foreseen by their inventors: e.g. cellular automata.
- Language independent debuggers generalize from earlier debuggers.

Almost all programming proceeds in the direction of generalization. Example: the phases of compiler construction are automated to different degrees, but constructing a compiler is now mostly a matter of using the appropriate code generators.

Generality needs support from the programming language. Current languages do not support generality well; new OO languages may be better (abstract classes, frameworks, . . .).

Generalization is related to abstraction.

Incrementality It is easier to make small changes to a working system than to rebuild the system. Why? Because if the modified system does not work, the errors must have been introduced by the small changes — **provided** that there are no hidden dependencies! (This leads into *Process Models*, which we do next.)

Read Ghezzi Chapter 3 *Software Engineering Principles*.

Quotations for Discussion There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies. (C. A. R. Hoare)

Every module should keep a secret. (Parnas 1972)

Design for change. (Parnas 1979).

Rules are my very humble, obedient servants. (Josef Haydn)

6 Process Models

Ghezzi 7.1. A **process model** is a description of a way of developing software. A process model may also be a methodology for software development.

6.1 Waterfall Model

See these notes 3.1 and Ghezzi pages 361–373. The waterfall model is: old (Royce 1970); document-driven; based on (alleged!) engineering practice; still used (e.g. in some IBM departments).

Good features:

- simple to understand;
- **phases** are important even if their sequence is not;

- works for well-understood problems;
- keeps managers happy.

Bad features:

- does not allow for change;
- does not work for novel or poorly understood problems;
- produces inaccurate estimates;
- does not allow for changing requirements;
- plethora of documents lead to “bureaucratic” project management with more concern for the existence/size of documents than their meaning.

6.2 Evolutionary Model

See Ghezzi pages 374–376. The evolutionary model is **increment driven** and **cyclical**:

1. deliver something (this is the “increment”);
2. measure “added value” to customer (may be positive and negative);
3. adjust design and objectives as required.

Evolution often requires **prototypes**.

6.3 Prototypes

A **prototype** is a preliminary version that serves as a model of the final product. Examples:

- A wood/clay model of a car (now replaced by CAD).
- (BBC) A full-size model of a grand piano built to determine whether a piano could be moved onto a concert stage (the model could not get through the door of the carpenter’s shop).
- Software Prototypes:
 - Emulate the user interface (UI) and see if people like it. (May lead to vapour-ware.)
 - Develop application code without UI to assess feasibility.
 - Use a HLL to build a prototype that will be written in a LLL. “Fast prototyping”: e.g. build APL prototype before FORTRAN product [Gomaa and Scott 1981]. Trade fast programming and fast execution.

There are several kinds of prototype.

Throwaway Prototype A throwaway prototype is not part of the final product. Throwaway prototypes should:

- be fast to build;
- help to clarify requirements and prevent misunderstanding;
- warn implementers of possible difficulties.

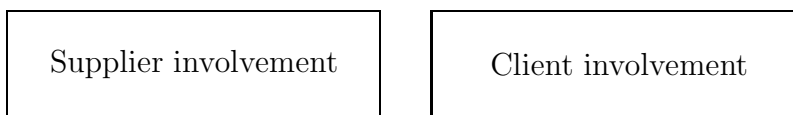
Some languages are suited to prototyping: APL, LISP, SML, Smalltalk. Others are not: FORTRAN, COBOL, C.

A prototype meets a **clearly identified subset** of requirements. Examples: it may provide a realistic UI but not provide full functionality; or it may provide a functional subset without meeting performance criteria.

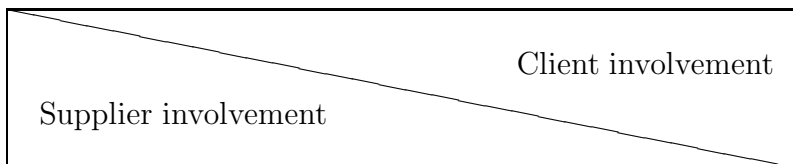
Evolutionary Prototype Evolutionary prototypes become part of the final product. They are usually written in the final language of the application. They fit well into the evolutionary model:

1. Develop a system that meets a well-understood (and possibly small) subset of the requirements.
2. Deliver the system and obtain feedback from the client.
3. Choose next-best understood requirement and work on that.

Incremental Prototype Even if all requirements are understood, the product may be developed as a sequence of **working components**. The idea is to avoid a sudden shock at the end of development when the client sees the product for the first time. Instead of:



We prefer more concurrency and fewer sudden surprises:



6.4 Spiral Model

The **spiral model** is Barry Boehm's (1986) formalization of the evolutionary model. See picture: Ghezzi, page 381.

The spiral model is based on **risks**. In **risk analysis**, we identify risks and respond to them before they endanger the whole project.

The spiral model envisaged by Boehm has four phases:

1. Identify objectives, alternatives, and constraints.
2. Evaluate alternatives and assess risks.
3. Develop according to established objectives and verify that these objectives are met.
4. Review results obtained during the current cycle. Plan another iteration if required.

WM is roughly “once around the spiral”. A typical industrial-scale project requires from three to six iterations.

6.5 Assessment of Models

It is hard to do large-scale comparative studies in software engineering, but there have been a few attempts (Boehm, Gray, and Seewaldt 1984).

- Waterfall development provides: good management of the process; poor response to clients; a large final product; a short test phase.
- Spiral development provides: short development time; good response to changes in requirements; a small final product.
- Consensus: the spiral is better than the waterfall, especially for products that are not well understood.
- “In the old days, we **wrote** software; then, for a while, we **built** software; nowadays, we **grow** software” (Brooks 1978, pages 200–1). The growth concept is due to Harlan Mills, *Top-down programming in large systems*, 1971.

Reading for Quiz 1

- | | |
|------------------|---|
| Ghezzi 1991 | Chapters 1, 2, 3;
Sections 7.1, 7.2. |
| Sommerville 1989 | Chapters 1, 2. |
| Course notes. | |

7 Design

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 design documents for the 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.

7.1.2 Architectural Design

The AD provides a **module diagram** and a brief **description** of the role of each module.

7.1.3 Module Interface Specifications

Each module provides a set of **services**. A module interface describes each service provided by the module.

“Services” are usually functions (used generically: includes “procedure”). 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.

To specify a function, give:

- 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 =  $X \in Table$ 
```

The ideas here are formalized in Larch: more later.

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, ...).

7.2 Remarks on Design

What designers actually do (Glass 1991, page 27):

- 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.

Reading Ghezzi 1991:

- 4.2 — Modularization
 - 4.2.3.1 — Text design notation (fine)
 - 4.2.3.2 — Graphical design notation (use with care)
- 4.5 — Case study (symbol table in detail)

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”.

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.
- Hierarchical architectures are easy to do.
- May be suitable for simple applications.
- Do not scale well to large applications.
- Leaves of the tree 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 (Dijkstra 1968) and Multics (MIT, Bell Labs, General Electric) (Corbato et al. 1965). (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.

See also these notes 7.3.4.

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 Generation
Layer 2	Database Manager	User Interface	

Event-driven version:

Layer 0	User Interface		
Layer 1	Regular Processing	Correction Processing	Report Generation
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 (Brooks 1986), 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 `typedefs` 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.
- 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 Dee.

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:

Line makeline (*Point* p_1 , p_2)
Point makepoint (*Line* l_1 , l_2)
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)?

Reading Chapter 4 of Ghezzi is about design. It covers the same topics as the lectures with somewhat different emphasis. Omit 4.6, on concurrent software design.

7.6 Design Notations

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

Diagrams (“graphical design notation”, GDN in Ghezzi) 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.

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”.

Good feature: 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 is popular in the industry (Yourdon and Constantine 1979).

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, Fig. 1 shows a simple dataflow diagram and Fig. 2 a corresponding structure chart. Unfortunately, the transformation from DFD to SC is informal, although guidelines exist (Ghezzi, pages 395–9).

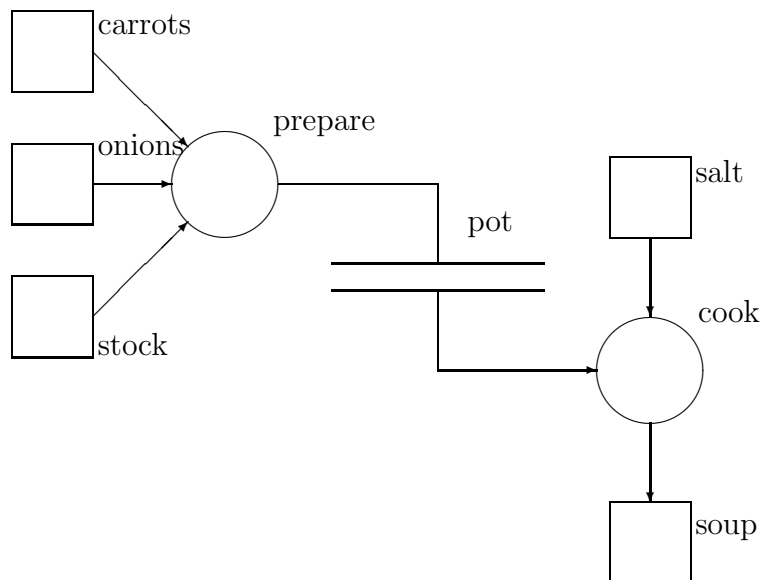


Figure 1: A Dataflow Diagram

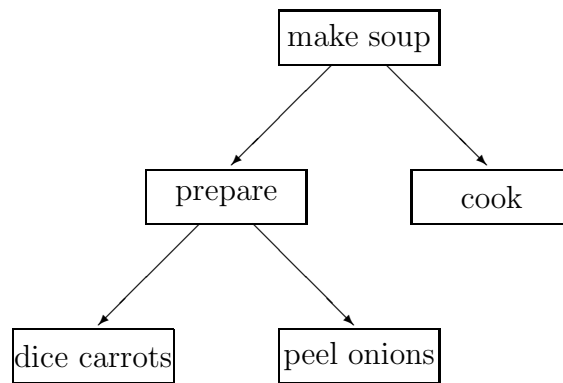


Figure 2: A Structure Chart

7.7.3 Jackson Structured Design (JSD)

Jackson Structured Design (JSD) has been pioneered by Michael Jackson²(1975, 1983) JSD is data-driven — the software design is based on relationships between data entities — but is not (as some 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.

²“He’s the sort of person who thinks Michael Jackson is a singer and James Martin is a computer scientist.” — Anon.

- 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)
  .....

```

7.8 Object Oriented Design

Note: these notes 13 describes the complete object oriented methodology.

It makes little sense to discuss object oriented design without first describing the basic ideas of object oriented programming. Hence the next section.

7.8.1 Object Oriented Programming

Theory follows practice (Glass 1991, pages 3–7). 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**.³ 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.⁴ 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.
- **Inspectors** returns values.
- **Mutators** change objects.

A stack object might have:

³The term “class” dates from Simula67 (Birtwistle, Dahl, Myhrhaug, and Nygaard 1973).

⁴Objects are intensional, not extensional.

- 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.⁵ In a program, we could write $A := D$ where A : *Animal* and D : *Dog*.

- 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.

⁵Consider “Animals have legs” and “Dogs bark”.

Meyer (1988, page 241) 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 (1988) 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  $Payment \geq \$10$ 
    ensures animal delivered

class DogShop
  inherits PetShop
  method Order
    requires  $Payment \geq \$5$ 
    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 Fig. 3. The structure of the system is determined by the main program; the library is a collection of individual components.

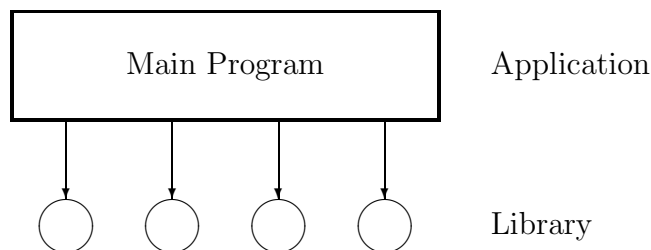


Figure 3: A Program Using Components from a Library

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 Fig. 4. The structure of the system is determined by the framework; the application is a collection of individual components.

The best-known framework is the MVC triad of Smalltalk (Goldberg and Robson 1983). It is useful for simulations and other applications. The components of the triad are a model, a view and a controller — see Fig. 5.

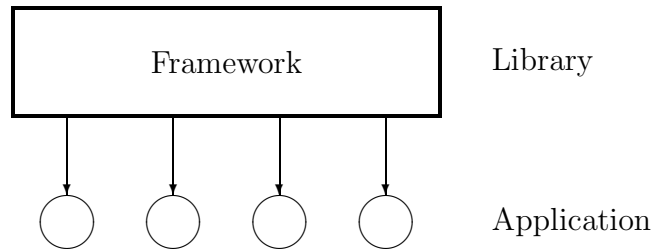


Figure 4: A Program Using a Framework from a Library

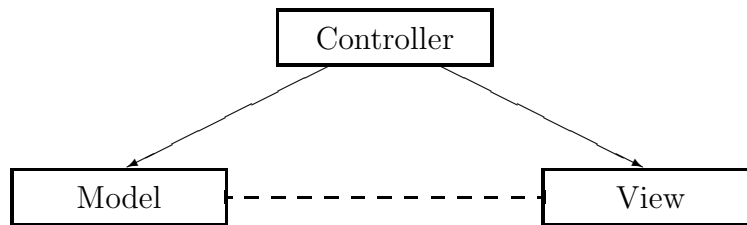


Figure 5: The Model-View-Controller Framework

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* (1989) 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, Wilkerson, and Wiener 1990, Appendix A). 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.

See Ghezzi, pages 115 and 121–2.

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 → first)
  p = p → next

```

write

```

for each object in objectlist do
  draw(object)

```

or even

```

∀ o ∈ 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  $0 \leq N < Max$  and P(N)
  then F(N)

```

or even

```

if ∃ N .  $0 \leq N < Max \wedge P(N)$ 
  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) ⇒ .....
  (0, H, L - 1, 2H - 1) ⇒ .....
  (0, 2H, L - 1, 3H - 1) ⇒ .....
  .....

```

- 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
   $x < 0 \Rightarrow$  NegAction
   $x = 0 \Rightarrow$  ZeroAction
   $x > 0 \Rightarrow$  PosAction

```

We can use these guidelines to improve parts of the GraphX design. Here is part of the specification of *GetnextObject* from page 22 of the GraphX specification.

```

if NID  $\neq$  NIL then
  if NID.next  $\neq$  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 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.

Part of *SearchByMarkers* on pages 23–24 of GraphX:

```

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

```

Reading From Ghezzi.

Read Chapter 4. Section 4.2.1.2 (*is-component-of relation*) is not very useful (a module uses its components, so why is this relation different from *uses*?). Section 4.2.6.1 (stepwise-refinement) is useful. Section 4.4 (anomalies) is useful but will not be in Quiz #2. Section 4.6 (concurrent software) can be ignored.

Read sections 5.1—5.5 (specification), ignoring 5.5.3 (Petri nets).

8 Formal Specification Techniques

A **formal specification** is a specification based on a mathematical theory (for example, first-order logic with extensions for sets and other simple structures). We can use mathematical techniques to reason about formal specifications: for example, we can prove properties of specifications.

8.1 Introduction to Z Notation

The Z notation is one of the best and best known (Spivey 1988; Spivey 1989). It was designed at Oxford and is used more widely in Europe than North America.

We can introduce sets simply by naming them.

$[NAME, DATE]$

A specification has a Name, a set of declarations, and a set of predicates.

<i>BirthdayBook</i>
<i>known</i> : $\mathbb{P} NAME$
<i>birthday</i> : $NAME \mapsto DATE$
<i>known</i> = dom <i>birthday</i>

This specification defines a state. It is **satisfied** by sets that have the properties required by the predicates.

known = { John, Mike, Susan }
birthday = { John \mapsto 25-Mar,
Mike \mapsto 20-Dec,
Susan \mapsto 20-Dec }.

We can give specifications of function and procedures. If a procedure alters the state, we indicate this by writing ΔN , where N is the Name of a previously defined specification. The “?” after *Name* and *Date* indicates that these are input variables. A prime (') after a variable indicates its value in the post-state.

<i>AddBirthday</i>
Δ <i>BirthdayBook</i>
<i>name?</i> : $NAME$
<i>date?</i> : $DATE$
<i>name?</i> \notin <i>known</i>
<i>birthday'</i> = <i>birthday</i> \cup { <i>name?</i> \mapsto <i>date?</i> }

The idea is that the new Name will be added to the set of names Known to the system. We can prove that the specification has the desired effect on the state. The proof uses conventional logic and set theory (cf. COMP 231).

known'
= dom *birthday'* invariant after
= dom(*birthday* \cup { *name?* \mapsto *date?* }) spec. of *AddBirthday*
= dom *birthday* \cup dom { *name?* \mapsto *date?* } fact about dom
= dom *birthday* \cup { *name?* } fact about dom
= *known* \cup { *name?* }. invariant before

When we define a function, the state should not change. We indicate this by writing ΞN , where N is the Name of a specification. The “!” after *Date* indicates that it is an output variable.

<i>FindBirthday</i>
$\exists \textit{BirthdayBook}$
$\textit{name?} : \textit{NAME}$
$\textit{date!} : \textit{DATE}$
$\textit{name?} \in \textit{known}$
$\textit{date!} = \textit{birthday}(\textit{name?})$

The output may be a set. The next specification returns the set of cards that we must send on a particular day.

<i>Remind</i>
$\exists \textit{BirthdayBook}$
$\textit{today?} : \textit{DATE}$
$\textit{cards!} : \mathbb{P} \textit{NAME}$
$\textit{cards!} = \{ n : \textit{known} \mid \textit{birthday}(n) = \textit{today?} \}$

Lecture 16 Tuesday, 31 Oct

Initializing is straightforward: the set *Known* should be empty.

<i>InitBirthdayBook</i>
$\textit{BirthdayBook}$
$\textit{known} = \emptyset$

We can declare enumerations by listing the alternatives.

$\textit{REPORT} ::= \textit{ok} \mid \textit{already_known} \mid \textit{not_known}$

The following simple specification introduces *Success*, satisfied whenever *result* has a particular value.

<i>Success</i>
$\textit{result!} : \textit{REPORT}$
$\textit{result!} = \textit{ok}$

An important feature of Z is that we can combine entire specifications by using their names and logical connectives.

$\textit{AddBirthday} \wedge \textit{Success}$.

The next specification sets *Result* if a *Name* is already in the birthday book.

AlreadyKnown $\exists \text{BirthdayBook}$ $\text{name?} : \text{NAME}$ $\text{result!} : \text{REPORT}$
$\text{name?} \in \text{known}$ $\text{result!} = \text{already_known}$

A **robust specification** is a specification that handles all cases. In this example, $RAddBirthday$ is a robust version of $AddBirthday$.

$$RAddBirthday == (AddBirthday \wedge Success) \vee AlreadyKnown.$$

Note that we could write the schema $RAddBirthday$ in full.

$RAddBirthday$ $\Delta \text{BirthdayBook}$ $\text{name?} : \text{NAME}$ $\text{date?} : \text{DATE}$ $\text{result!} : \text{REPORT}$
$(\text{name?} \notin \text{known} \wedge$ $\text{birthday}' = \text{birthday} \cup \{\text{name?} \mapsto \text{date?}\} \wedge$ $\text{result!} = \text{ok}) \vee$ $(\text{name?} \in \text{known} \wedge$ $\text{birthday}' = \text{birthday} \wedge$ $\text{result!} = \text{already_known})$

We can build robust versions of $FindBirthday$ and $Remind$ in a similar way. First, we introduce the concept of a name that isn't known.

$NotKnown$ $\exists \text{BirthdayBook}$ $\text{name?} : \text{NAME}$ $\text{result!} : \text{REPORT}$
$\text{name?} \notin \text{known}$ $\text{result!} = \text{not_known}$

The robust version of $FindBirthday$ checks that the name is known.

$$RFindBirthday == (FindBirthday \wedge Success) \vee NotKnown.$$

The $Remind$ operation can be called at any time: it never results in an error, so the robust version need only add the reporting of success:

$$RRemind == Remind \wedge Success.$$

From Specification to Design

We use a pair of arrays to represent the birthday book. The formalization of an array is a mapping from natural numbers to values of a set.

$$\begin{aligned} names &: \mathbb{N}_1 \rightarrow NAME \\ dates &: \mathbb{N}_1 \rightarrow DATE. \end{aligned}$$

The element $names[i]$ of the array is simply the value $names(i)$ of the function, and the assignment $names[i] := v$ is exactly described by the specification

$$names' = names \oplus \{i \mapsto v\}.$$

We must ensure that there are no duplicate names. The active array entries are $1..hwm$ (hwm stands for “high water mark”).

$\begin{aligned} & \textit{BirthdayBook1} \\ & names : \mathbb{N}_1 \rightarrow NAME \\ & dates : \mathbb{N}_1 \rightarrow DATE \\ & hwm : \mathbb{N} \end{aligned}$
$\begin{aligned} & \forall i, j : 1 .. hwm \bullet \\ & \quad i \neq j \Rightarrow names(i) \neq names(j) \end{aligned}$

We can define the set of known names and the correspondence between names and birthdates.

$\begin{aligned} & \textit{Abs} \\ & \textit{BirthdayBook} \\ & \textit{BirthdayBook1} \end{aligned}$
$\begin{aligned} & known = \{ i : 1 .. hwm \bullet names(i) \} \\ & \forall i : 1 .. hwm \bullet \\ & \quad birthday(names(i)) = dates(i) \end{aligned}$

To add a new entry, we increment the array index. If f is a function, then $f \oplus \{x \mapsto y\}$ is the same function except that $f(x) = y$.

$\begin{aligned} & \textit{AddBirthday1} \\ & \Delta \textit{BirthdayBook1} \\ & name? : NAME \\ & date? : DATE \end{aligned}$
$\begin{aligned} & \forall i : 1 .. hwm \bullet name? \neq names(i) \\ & hwm' = hwm + 1 \\ & names' = names \oplus \{hwm' \mapsto name?\} \\ & dates' = dates \oplus \{hwm' \mapsto date?\} \end{aligned}$

We can show:

1. Whenever *AddBirthday* is legal in some abstract state, the implementation *AddBirthday1* is legal in any corresponding concrete state.
2. The final state which results from *AddBirthday1* represents an abstract state which *AddBirthday* could produce.

It is now a simple matter to write code to implement the specification.

```

procedure AddBirthday (Name: NAME; Date: DATE);
begin
  hwm := hwm + 1;
  names[hwm] := Name;
  dates[hwm] := Date
end;

```

A search is abstracted by existential quantification.

$\frac{\textit{FindBirthday1}}{\exists \textit{BirthdayBook1} \quad \textit{name?} : \textit{NAME} \quad \textit{date!} : \textit{DATE}}$
$\exists i : 1..hwm \bullet \textit{name?} = \textit{names}(i) \wedge \textit{date!} = \textit{dates}(i)$

```

procedure FindBirthday (Name: NAME; var Date: DATE);
var i: integer;
begin
  i := 1;
  while names[i] <> Name do
    i := i + 1;
  Date := dates[i]
end;

```

Initialization is straightforward: $hwm = 0$.

$\frac{\textit{InitBirthdayBook1}}{\textit{BirthdayBook1} \quad \textit{hwm} = 0}$
--

We can show that initialization leaves the set of known names empty:

$$\begin{aligned}
 \textit{known} &= \{ i : 1..hwm \bullet \textit{names}(i) \} && \text{from } \textit{Abs} \\
 &= \{ i : 1..0 \bullet \textit{names}(i) \} && \text{from } \textit{InitBirthdayBook1} \\
 &= \emptyset. && \text{since } 1..0 = \emptyset
 \end{aligned}$$

The code is obvious:

```
procedure Initialize;
begin
  hwm := 0
end;
```

Further examples are given in (Hayes 1987). They include a symbol table, a telephone network, the UNIX file system, and parts of the specification for IBM's CICS (Customer Information Control System).

8.2 Advantages and Disadvantages of Formal Specification

- + Formal specification requires a detailed and thorough study of the design. This study may be the greatest benefit of formal specification.
- + Languages such as Z and VDM provide full first-order logic with set theory: consequently, they are powerful and expressive.
- We can use the formal specification to prove properties of the design.
In general, we cannot prove that a formal specification is correct, because the requirements are stated informally.
- Many programmers do not like formal notation (or may be afraid of it).
- Some things are hard to specify.

There is a trade-off between model-based specification languages, such as Z and VDM, and property-based specification languages such as OBJ, Clear, and ActOne. Paradoxically, the property-based languages are in some senses more abstract, but they are also executable.

9 Validation and Verification

This is the “VV” in “IVV Team”. Usage of these words varies. Ghezzi (Chapter 6) uses “verification” for all purposes and hardly mention “validation”. Some writers use “V&V” as a single noun. We make the following distinction, following (Boehm 1979).

- | | | |
|--------------|---|--|
| Validation | = | have we built the right product? |
| | = | check that product matches SRD |
| Verification | = | have we built the product right? |
| | = | are all internal properties satisfied? |

We can also say, approximately:

- | | | | | |
|--------------|---|-----------|---|-----------|
| validation | ~ | black box | ~ | semantics |
| verification | ~ | clear box | ~ | syntax |

For example, if we write a program and compile it, we are *verifying* it. If we then check that it does what we expect, we are *validating* it.

Or, if we have a function F with

```
function  $F$ 
  requires  $P$ 
  ensures  $Q$ 
```

then verification checks that P and the execution of the body imply Q and validation checks that Q corresponds to a requirement.

Ideally, all properties of the program should be validated: correctness, performance, reliability, robustness, portability, maintainability, user friendliness, In practice, it is usually not feasible to validate everything.

Some results may be precise: tests passed or failed, or a percentage of tests passed. Other results may be subjective: user friendliness.

9.1 Varieties of Testing

We can perform verification and validation by **testing**. A single test can provide both validation and verification. A *failure* (e.g. system crashes) reveals an internal fault in the system (verification). An incorrect result indicates that the software does not meet requirements (validation).

The main problem with testing is that **testing can never be complete**.

The following classification is from (Glass 1991, pages 45–47). It is shown in tabular form in Fig. 6. The bullets in Fig. 6 indicate the kind of testing that is most likely in the specified category.

1. Goal-driven testing.
 - (a) Requirements-driven testing. Develop a *test-case matrix* (requirements vs tests) to insure that each requirement undergoes at least one test. Tools are available to help build the matrix.
 - (b) Structure-driven testing. Construct tests to cover as much of the logical structure of the program as possible. A *test coverage analyzer* is a tool that helps to ensure full coverage.
 - (c) Statistics-driven testing. These tests are run to convince the client that the software is working by running typical applications. Results are often statistical.
 - (d) Risk-driven testing. These tests check “worst case” scenarios and boundary conditions. They ensure robustness.
2. Phase-driven testing.

- (a) Unit testing. Test individual components before integration.
- (b) Integration testing. Assemble the units and ensure that they work together.
- (c) System testing. Test the entire product in a realistic environment.

	Unit	Integration	Acceptance
Requirements			•
Structure	•	•	
Statistical			•
Risk	•	•	•

Figure 6: Classification of Testing

9.2 Designing Tests

A **test** has two parts:

- A procedure for executing the test. This may include instructions for getting the system into a particular state, input data, etc.
- An expected result, or permitted range of results.

Ideal: one test for each requirement in the SRD. It is very unlikely that this ideal will be met. The goal, however, is to “cover” the SRD as completely as possible. Cf. the “test matrix” above.

Suppose that the SRD contains the following requirement:

When the user enters X and Y , the program displays $X + Y$.

Suppose X and Y are 32-bit numbers. This requirement calls for 2^{64} tests!

In practice, of course, we assume some form of **continuity**. If the program adds a few numbers correctly, and it works at the boundaries, we **assume** that it adds all numbers correctly.

There are two possibilities:

Black box testing: We choose tests without knowledge of how the program works, i.e. based on requirements only.

White box testing: We choose test based on our knowledge of how the program works.

Example: if the program contains

```

if  $N \leq 1000$ 
  then ....
  else ....

```

then testing arbitrary values of N would be black box testing but tests that distinguish $N \leq 1000$ and $N > 1000$ would be white box testing.

9.2.1 Guidelines for Black Box Testing

- Test for success and failure.
Requirement: “The maximum length of a line is 255 characters.”
Tests: lines with length l such that $l \leq 255$ and $l > 255$.
- Test boundary conditions.
Test: $l = 255$.
- Test as many *combinations* as feasible.
Requirement: an editor requires special treatment of tab characters and has special actions at the right side of the window.
Test: tabs, right side of window, **and** tab character at extreme right.

9.2.2 Guidelines for White Box Testing

The general idea is to ensure that every component of the program is exercised. Possible criteria include:

All statements Every statement in the program must be exercised during testing.

All Edges All edges of the control graph must be exercised. (This is very similar to “all statements”.)

All Branches Each possibility at a branch point (**if** or **case** statement) should be exercised.

All Paths Exercise all paths: usually intractable.

Fig. 7 shows a simple example (Ghezzi, page 276). Consider the following tests:

Test	X	Z
1	0	1
2	1	3
3	0	3
4	1	1

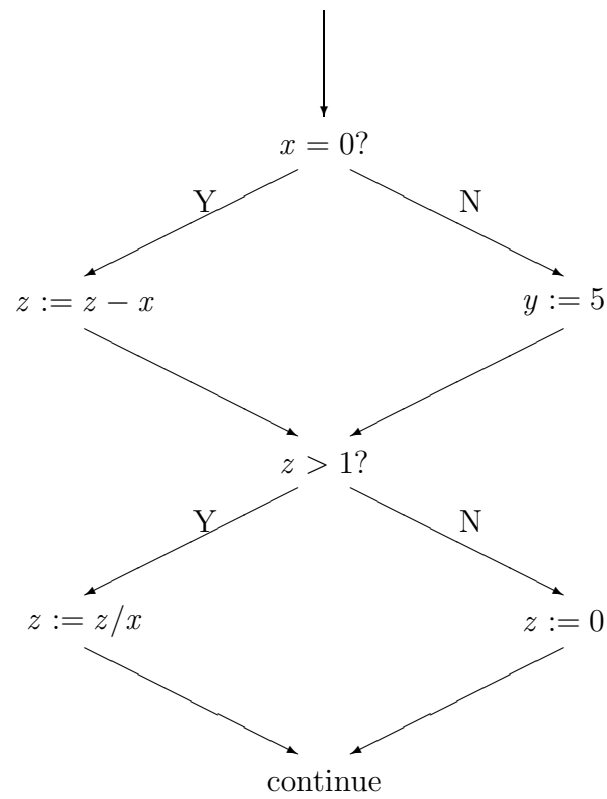


Figure 7: A Simple Flowgraph

We could use either $\{1,2\}$ or $\{3,4\}$ to cover all edges. But $\{3,4\}$ reveals the “divide by zero” error, whereas $\{1,2\}$ does not.

The “all paths” test would include all of $\{1,2,3,4\}$ and would reveal the error. The number of paths, however, will clearly be very large in a practical case.

See (Parrish and Zweben 1995) for further discussion of the relationships between the different testing criteria.

9.3 Stages of Testing

Unit Testing: test an individual unit or basic component of the system. Example: test a function such as *sqrt*.

Module Testing: test a module that consists of several units, to validate the interaction of the units and the module interface.

Subsystem Testing: test a subsystem that consists of several modules, to validate module interaction and module interfaces.

Integration Testing: test the entire system.

Acceptance Testing; test the entire system thoroughly with real data to satisfy the customer that the system meets the requirements. (This is the only kind of test that I will see!)

9.4 Testing Strategies

The strategies described here apply to *development* testing, not acceptance testing.

Should we test top-down or bottom-up?

9.4.1 Top-down Testing

- goes from system to subsystem to module to unit;
- requires that we write stubs for parts of the system that are not yet completed.

A **stub**:

- replaces a module or unit for the purposes of testing;
- must return with a valid response;
- may do nothing useful;
- may always do the same thing;
- may return random values;
- may handle specific cases.

Examples of stubs.

- A stub for a file subsystem
 - always returns handle of a test file
 - randomly says that the file is not accessible
- A stub for an event handler
 - always returns coordinates of the centre of the window
 - returns random coordinates inside or outside the window

Advantages and disadvantages of top-down testing:

- + Catches design errors (but these should have been caught in design reviews!).
- + Enables testing to start early in the implementation phase.
- It is hard to write effective stubs.
- Stubs take time to design and write.

9.4.2 Bottom-up Testing

Test units, then modules, then subsystems, then system. We require **drivers** to exercise each unit or module because its environment does not exist yet. Drivers must

- provide environment and simulated input
- check outputs

Advantages and disadvantages of bottom-up testing:

- + Each component is tested before it is integrated into a larger component.
- + Debugging is simplified because we are always working with reliable components.
- Drivers must be written; drivers are usually more complex than stubs and they may contain errors of their own.
- Important errors (e.g. design errors) may be caught late in testing — perhaps not until integration.

Clearly, both top-down and bottom-up testing has a cost: we need to write additional code (“scaffolding”) to run the tests. usually, top-down testing is cheaper, because stubs are simple to write. Building the support code for bottom-up testing is more difficult and the support code itself may contain errors if interfaces are not properly understood.

Mixed strategies are also possible. We can aim at gradual refinement of the entire system, doing mostly top-down testing, but with some bottom-up testing.

The order of testing and the order of implementation must be chosen together. Clearly, top-down testing requires top-down coding.

9.5 Testing Procedures

We assume a **Test Set** with *Max* tests. Here is a simple procedure for testing.

```

for  $N := 1$  to  $Max$  do
   $Test(N)$ 
  if  $Failed(N)$ 
    then  $FixError(N)$ 

```

The problem with this procedure is that fixing an error may invalidate an earlier test. The following procedure corrects this problem.

```

repeat
  for  $N := 1$  to  $Max$  do
     $Test(N)$ 
  for  $N := 1$  to  $Max$  do
    if  $Failed(N)$ 
      then  $FixError(N)$ 
until no errors

```

This is safe, but very slow. The final procedure is a compromise between these extremes; it is called **regression testing**.

```

 $N := 1$ 
while  $N \leq Max$  do
   $Test(N)$ 
  if  $Failed(N)$ 
    then
       $FixError(N)$ 
       $N := 1$ 
    else
       $N := N + 1$ 

```

Regression testing is tedious to do by hand, especially if there are a hundred or more tests. Common practice is to use **test tools** that run through tests, stopping when a test fails.

9.5.1 When do we stop?

Ideal: stop when all tests succeed.

Practice: stop when the cost of testing exceeds the cost of shipping with errors.

Microsoft: let the customer find the errors :).

As the number of errors gets smaller, the cost of finding errors increases. Shipping with errors has a cost, because customer support is needed, but this cost falls with the number of remaining errors. Fig. 8 is based on this reasoning: it shows that there is a time T_0 at which the cost of shipping with errors falls below the cost of finding further errors. The problem, of course, is that it is hard to know when T_0 has been reached!

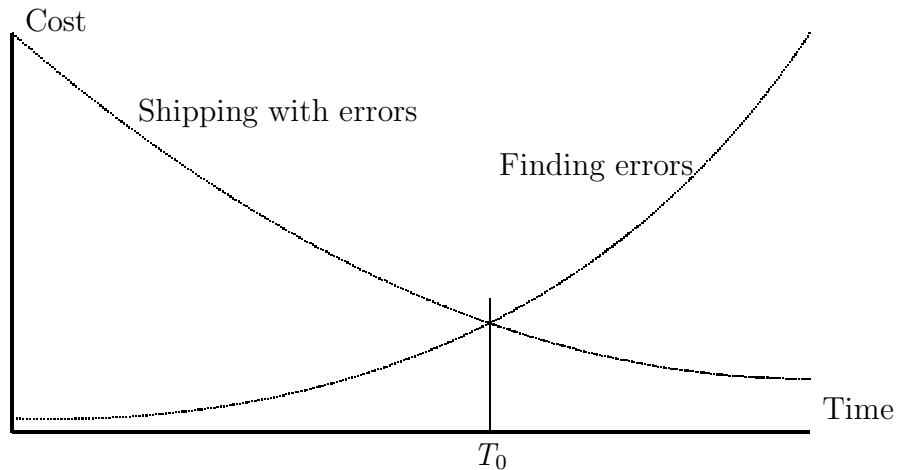


Figure 8: Deciding when to ship

9.6 Preparing Test Cases

A **Test Set** consists of a list of tests. Each test should include the following three components.

Purpose For an acceptance test, the purpose is an SRD item. For a unit or subsystem test, the purpose is an internal requirement based on the design.

Data The environment (i.e. state of the system when the test is conducted), inputs to functions, etc.

Expected Result The effect of conducting the test, the value returned by a function, the effect of a procedure, etc.

A **Test Plan** has the following components.

- A description of the phases of testing. For example: unit, system, module.
- The objectives of the testing phase (verify module, validate subsystem, etc).
- A schedule that specifies who does what to which and when.
- The relationship between implementation and testing schedules (don't schedule a test before the component is written!).
- Tracing from tests to requirements.
- How test data are generated.
- How test results are recorded.

10 Reviews, Walkthroughs, and Inspections

The basic idea of reviews, walkthroughs, and inspections is the same: a team examines a software document during a meeting. Studies have shown that errors are found more

effectively when a group of people work together than when people work individually. Common features include:

- a small group of people;
- the person responsible for the document (analyst, designer, programmer, etc) should attend;
- one person is responsible for recording the discussion;
- managers must **not** be present, because they inhibit discussion;
- errors are recorded, but are **not** corrected.

During a **review**:

- the author of the document presents the main themes;
- others criticize, discuss, look for omissions, inconsistencies, redundancies, etc.
- faults and potential faults are recorded.

During a **walkthrough**:

- Each statement or sentence is read by the author;
- others ask for explanation or justification if necessary.

Example:

“Since $n > 0$, we can divide”
“How do you know that $n > 0$?”

During an **inspection**:

- code is carefully examined, with everyone looking for common errors.

As usual, there is variation in usage: an IBM “inspection” is close to what we have called a “walkthrough”.

Some general rules:

- Teams prepare in advance, e.g. by reading the documentation.
- Meetings are not long — at most 3 hours — so concentration can be maintained.
- A moderator is advisable to prevent discussions from rambling.
- The author may be required to keep silent except to respond to questions. If the author explains what s/he thought s/he was doing, others may be distracted from what is actually written.
- All members must avoid possessiveness and egotism, cooperating on finding errors, not defending their own contributions.

10.1 Experience Reports

Glen Russell (1991) report on inspection of “ultralarge-scale developments” at Bell-Northern Research.

- 2.5 MLOCS was inspected over 7 years.
- Inspection rate was 150 LOCS/hour.
- Meetings were 2 hours.
- A 4 person team found, on average, 4 defects/hour.
- Effectiveness depended on speed. At 150 LOCS/hour, teams found 37 defects/KLOC; at 750 LOCS/hour, they found only 8 defects/KLOC.

A study of debugging at BNR showed that an individual debugging at a workstation found between 0.2 and 0.4 defects/hour. Inspection is therefore between 2 and 5 times as effective as debugging as a way of finding software defects.

Other results:

- Design reviews are far more cost effective than code reviews, but code reviews find more errors (Glass 1991, page 41). (Note that design errors are much more expensive to correct after they have been frozen in code.)
- In one experiment, code reading detected more *interface* faults than other methods, but functional testing detected more *control* faults than other methods (Glass 1991, page 43).
- “One reliability assurance technique — testing — was not cost effective.”
- A more recent study at University of Maryland and Bell Labs shows smaller benefits of inspection (Porter, Siy, Toman, and Votta 1995). Only 15% of the problems found during *preparation* turn out to be actual defects; other items found are “false positives” (50%) or pertain to nonfunctional or maintenance issues (35%). Two reviewers are better than one, but further increases in team size had little effect.

11 Software Metrics

We have discussed a number of qualities of software (these notes 5.2) but we have no way of quantifying them. When is a program “too complex” or “sufficiently portable”? In general, it is difficult or impossible to assign numerical quantities to attributes such as user-friendliness or interoperability, but we can attempt to quantify the “harder” qualities, such as efficiency, complexity, and maintainability.

Why? To monitor progress, estimate time to completion,

When? At all stages of the project.

What? Any quantity that can be measured and that will give reliable information about the product.

11.1 Measurement Theory

Much of the work on software metrics lacks a scientific basis. This section, based on (Fenton 1994), outlines such a basis.

Consider peoples' height. Informally, we say that “ Y is short”, “ X is taller than Y ”, and so on. This illustrates the idea of *measurement*. We can formalize the measurement of height by measuring peoples' height with a ruler, calibrated in (say) inches. If X 's height is 70 inches and Y 's height is 64 inches, the mathematical relationship $70 > 64$ justifies the informal statement “ X is taller than Y ”.

In the theory of measurement, an **empirical relation system** is a pair (C, R) in which C is a set of entities and R is a set of relations. (For simplicity, we will assume that R is a single relation.) In the example above:

$$\begin{aligned} C &= \text{the set of people} \\ R &= \text{the relation “is taller than”} \end{aligned}$$

Since empirical relations are informally defined, we cannot reason with them. Reasoning requires numbers. A **numerical relation system** is a pair (N, P) in which N is a set of numbers (usually the integers or the reals) and P is a relation on N . If we define

$$\begin{aligned} N &= \text{the set of real numbers in } [0, 120] \\ P &= \text{the relation } > \end{aligned}$$

then we can map (C, R) to (N, P) as follows: for $C \rightarrow N$, convert a person into that person's height in inches; for $R \rightarrow P$, convert “is taller than” into “ $>$ ”.

A numerical relation system is a **representation** of an empirical representation system if the **representation condition** is satisfied. For our example, the representation condition is

$$x \text{ is taller than } y \Leftrightarrow h(x) > h(y)$$

where $h(x)$ is the height of person x in inches.

The example of height is particularly straightforward. Other measurements may not be as clear. Students, for example, are often compared informally by saying “ X is smarter than Y ”. A possible representation is “ X has a higher GPA than Y ”. But the representation condition is no longer so clear. Consider these records:

Student	<i>Grades</i>			
X	B	B	B	B
Y	A	C	A	C
Z	A	A	C	C

All three students have the same GPA, but do they have the same “smartness”? We might say that X is consistently average, Y is uneven, and Z seems to have had a bad session.

Once we have a numerical scale, there are various kinds of transformation that we can apply to it. If we multiply everyone's height by 2.54, we obtain heights in centimetres, but we do not change any relationships. Also, it makes sense to say “ X is twice as tall as Y ”. Thus height is a **ratio scale**. Multiplying by a constant is an **admissible transformation** for a ratio scale.

GPA, on the other hand, is not a ratio scale. GPA is calculated by converting marks to letter grades and then assigning numbers to the letter grades. Multiplying the marks by a constant does not necessarily multiply the GPA by the same constant. It does not make sense to say “ X is twice as smart as Y ”.

There are even more serious problems with “intelligence quotients” (IQ) because it is not clear what is being measured. The fact that IQ is normally distributed suggests that many factors contribute to it and a multidimensional scale would be better.

A relation R is a **strict weak order** if:

- it is *asymmetric* (xRy implies $\neg yRx$), and
- it is *negatively transitive* (there are no values x , y , and z such that xRy , yRz , and zRx).

Cantor's Theorem The empirical relation system (C, R) has a representation in $(\mathcal{R}, <)$ if and only if R is a strict weak order. (\mathcal{R} is the set of real numbers.)

The difficulty with software metrics is that we can most easily measure *internal characteristics* of the program but we want information about the *external characteristics* of the program (Henderson-Sellers 1995). Fig. 9 compares internal characteristics, which are easily measured and not very interesting, and external characteristics, which are hard to measure but useful to have. Fig. 10 shows the relationship between these two kinds of measurement. The arrow labelled “Link?” indicates the relationship that we would like to have: from objective measurements, we can infer external characteristics.

Internal Characteristics	External Characteristics
size	complexity
control flow complexity	understandability
intermodule coupling	modifiability
modular cohesion	testability
	maintainability
	quality

Figure 9: Internal and External Characteristics

11.2 Lines of Code (LOCs)

Even such a simple and intuitive measure as LOCs is difficult to define precisely.

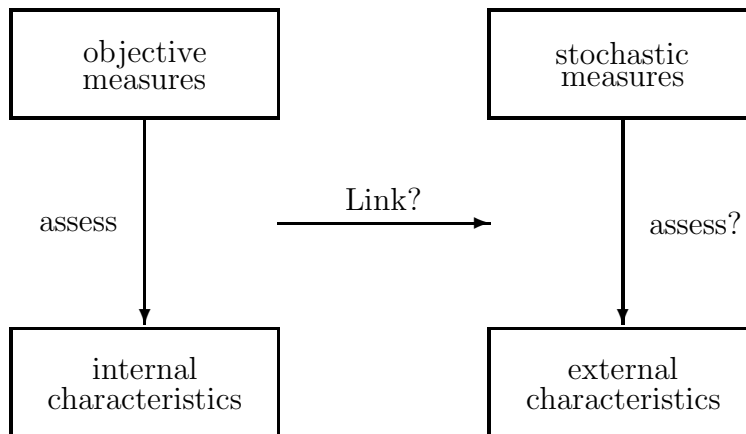


Figure 10: Measurements and Characteristics

- Do we count blank lines, lines with comments, lines that contain just “**begin**” or just “{”? Jones identified eleven different ways of counting lines with 500% variation.
- What is being measured? Is a long program better or worse than a short program? Is the amount of effort required to write a program proportional to the LOCs in the program?
- LOCs clearly depend on the programming language we use, but how?

In summary, LOCs provide a mapping from programs to reals, but there is no obvious empirical relationship.

British Telecom measures programs in kilometres: at 6 lines/inch, we can write 200 KLOCs/kilometre. A 50 MLOC program corresponds to 250 kilometres.

11.3 Software Science

Software Science is a theory developed by Halstead (Ghezzi, pages 340–343). We describe it here with slightly simplified notation. Let

- a = the number of distinct operators in the program
- b = the number of distinct operands in the program
- X = the total number of operators in the program
- Y = the total number of operands in the program

Operators include **begin/end** pairs, labels, and so on, as well as conventional operators such as + and −. Define

- $V = a + b$ = the vocabulary of the program
- $S = X + Y$ = the size of the program

Since it is unlikely that the program has many identical parts, we assume that the program consists of S/V distinct substrings each of length V .

For a given alphabet of size k , there are k different strings of length r . In particular, there are V^V different strings of length V . This gives us an upper bound for S : since $S/V \leq V^V$, we have $S \leq V^{V+1}$.

In practice, operators and operands tend to alternate, so we can refine this relation to $S \leq V \cdot a^a \cdot b^b$.

Halstead argues that the program must contain all of the 2^S ordered subsets of S elements. Thus

$$2^S = V \cdot a^a \cdot b^b.$$

Thus we can obtain the following estimate \hat{S} of S :

$$\hat{S} = \log_2 V + a \log_2 a + b \log_2 b.$$

The reasoning seems dubious. Nevertheless, if we take a sample of programs and compare S , the actual size of the program, with \hat{S} , the estimated size according to Halstead's theory, the results agree within 10%!

11.4 Cyclomatic Complexity

McCabe's theory measures the complexity of an algorithm rather than the size of a program. We draw the control flow graph of the program and define

- e = the number of edges
- n = the number of nodes
- p = the number of connected components (usually 1)

Then the **cyclomatic complexity** of the graph, C , is given by

$$C = e - n + 2p.$$

McCabe says that well-structured modules should have a cyclomatic complexity between 3 and 7, with 10 as the upper limit.

Unfortunately, cyclomatic complexity fails to be a measurement by the criterion above. Consider the three flowgraphs shown in Fig. 11; their cyclomatic complexities are 3, 3, 5, and 5. But their subjective complexities are quite different. In particular, flowgraphs with loops are perceived as being more complex than flowgraphs without loops, although the number of loops⁶ does not affect the cyclomatic complexity.

⁶That is, the direction of the arrows.

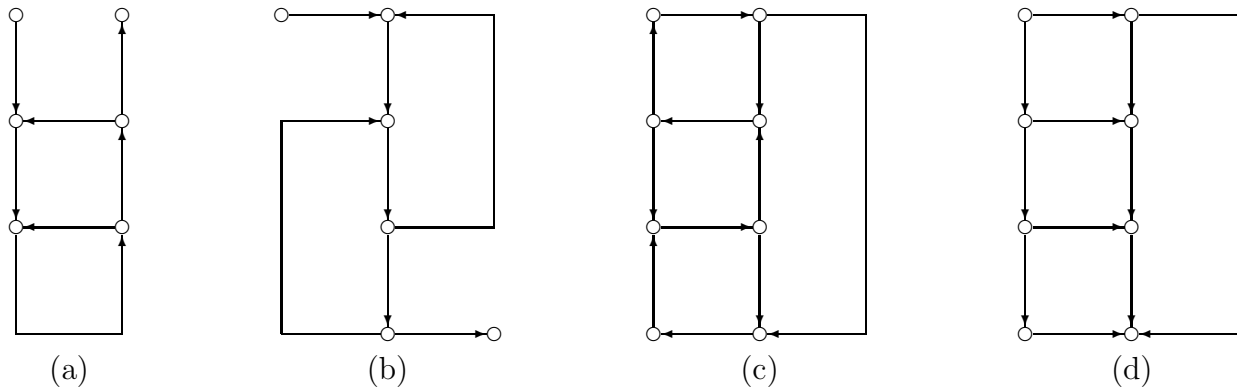


Figure 11: Four Flowgraphs

11.5 Function Points

Function points are an attempt to derive a single number that correlates well with programmer productivity. That is, if we know the number of function points in a program, we can estimate how many work-hours will be required to implement it (Ghezzi, pages 421–423). Function points were proposed by Albrecht and Gaffney (1983) and modified by Symons (1988).

1. Identify system components as seen by end user. Note that this can be done from requirements documents, or perhaps design documents.
2. Classify them using a weighting scheme such as the one in the following table.

Component	Simple	Average	Complex
Inputs	3	4	6
Outputs	4	5	7
Inquiries	7	10	15
External files	5	7	10
Internal files	3	4	6

The components are:

- Input Something the user provides, e.g. keyboard input.
- Output Something the system provides, e.g. screen display.
- Inquiry A user input that requires system response.
- File Function points are business-oriented.

3. Compute UPF (Unadjusted Function Points) by summing products of components and weights.
4. Compute TCF (Technical Complexity Factor) from 14 “characteristics” and “degrees of influence”.

5. FP (function points) = $UPF \times TCF$.
FP provides a measure of “system size”.
6. Compute person-hours = $A \times FP + B$ for suitable constants A and B . These constants depend on the programming language, but are consistent for a given language.

11.6 Dos and Don'ts for Metrication

Measurement is likely to succeed if measurement results:

- are used to make decisions;
- are communicated and accepted outside the measuring department;
- are accumulated for at least two years.

Measurement is likely to fail if:

- the purpose is not clearly defined;
- measurement is perceived to be irrelevant;
- programmers perceive measurement as critical of their performance;
- overworked staff are given additional work;
- management ignores the results of measurement;

11.7 Summary

Can we measure program complexity?

Halstead's theory: based on statistical expectations; some quite good predictions; pseudoscience.

McCabe's theory: based on flowgraph properties; a few predictions; pseudoscience.

Function points: based on detailed study of IS programs; quite useful; semiscience.

How important is it to measure software? According to Glass, the major problem — and the cause of many software “failures” — is the failure to estimate accurately (Glass 1991, pages 151 and 215). In other words, late delivery and budget excesses are not really a failure of software engineering but rather a failure to **estimate** the amount of work required to complete a project.

See (Kitchenham, Pfleeger, and Fenton 1995) for a discussion on the validation of software measurement tools.

See (Khoshgoftaar and Oman 1994) for various discussions of software metrics.

12 Cleanroom Software Engineering

The **cleanroom software engineering** methodology was introduced by Harlan and Linger (1987) and has been evaluated by Selby *et al.* (Selby, Basili, and Baker 1987). The major points are:

- precise functional specifications
- design verification
- statistical testing

12.1 The Cleanroom Process

Fig. 12 shows the cleanroom process graphically.

The **Specification Team**:

- develops a **box structured specification**
- defines a pipeline of **software increments** accumulating to a final product
- includes:
 1. estimated use statistics
 2. function and performance requirements

The **Development Team**:

- develops a **box structured design**
- codes from design
- verifies that the code meets the specification for each increment
- does **not** use a computer for compiling or debugging
- corrects failures detected during certification

12.2 Box Structure

A **box** maintains data and is accessible only via operations.⁷

A **black box**:

- provides an external view of the box
- is defined as a function $B : X^n \longrightarrow Y$, in which X^n is a *sequence of inputs* (the *canonical traces*) and Y is an *output*.

A **State Box**:

- provides an intermediate view of the box
- is defined as a *state*

⁷A box is rather like an object.

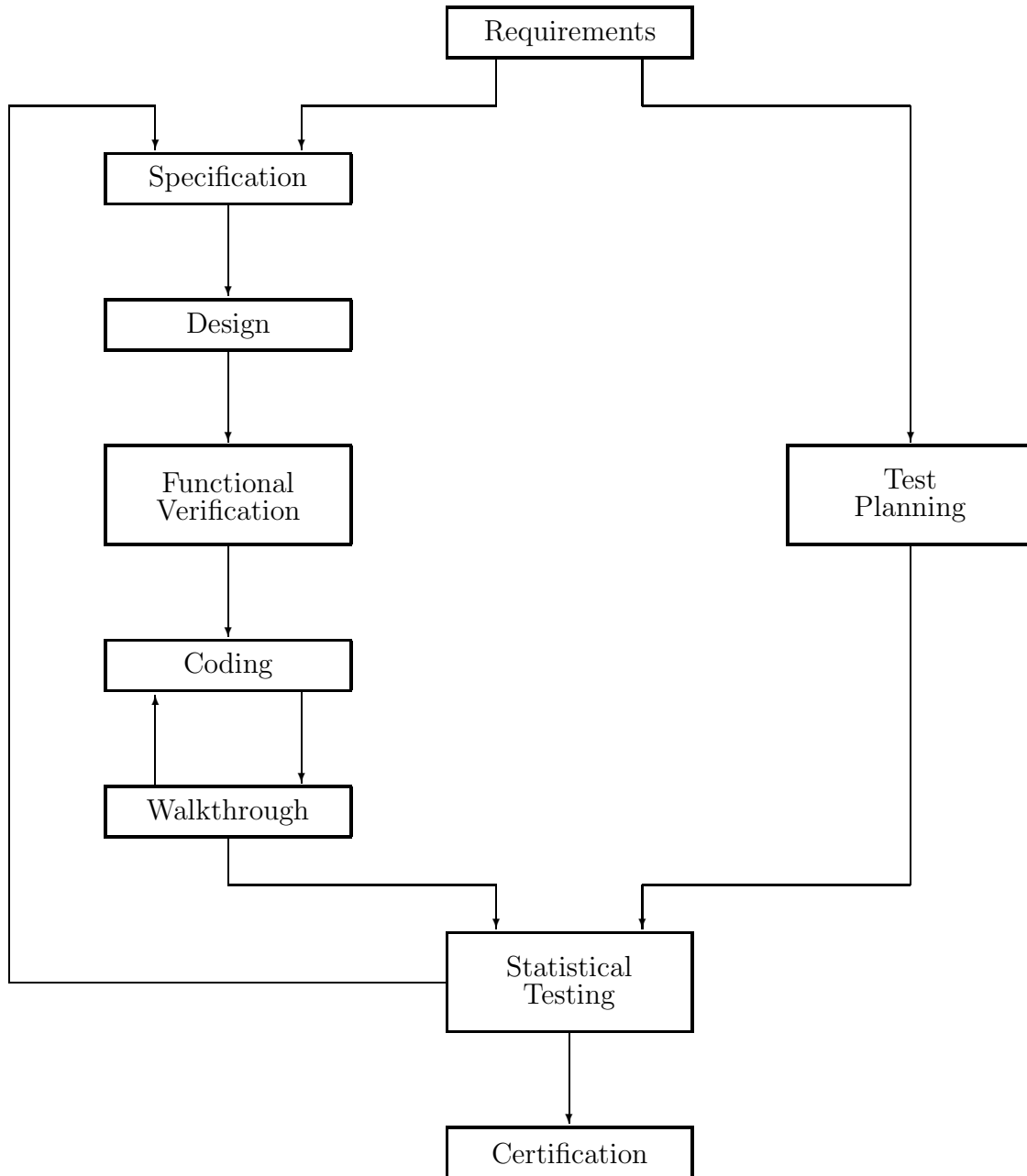


Figure 12: Cleanroom Software Engineering

- each input determines a state transition
- each output is a projection of the state

A **Clear Box**:

- provides an internal view of the box
- implements state transitions using structured code
- developed by top-down refinement down to code level

12.3 Functional Verification

The correctness of design with respect to the specification and the correctness of code with respect to the design is verified by proving that functionality is maintained. Proofs are checked by **formal software inspection**.

Inspections last 2 to 3 hours. Each team prepares for the inspection. During one hour

- 10 pages of high level design; or
- 5 pages of detailed design; or
- 100–200 code statements

are inspected. Inspections are identified as I_n , where:

- I_0 checks that the box design corresponds to the requirements;
- I_1 checks that the data structures and control correspond to the design; and
- I_2 checks that the code corresponds to the control.

Inspections are performed as often as needed, not just once.

12.4 Statistical Testing

Markov Chains Let S_1, \dots, S_n be a set of states. Define the matrix M of *transition probabilities*: M_{ij} is the probability of a transition from state S_i to S_j . Clearly:

$$\forall i . \sum_j M_{ij} = 1.$$

For example, a three-state transition matrix might look like this:

$$M = \begin{bmatrix} 0.1 & 0.3 & 0.6 \\ 0.3 & 0.1 & 0.6 \\ 0.4 & 0.5 & 0.1 \end{bmatrix}$$

A **chain** is a sequence of states. From M_{ij} , we can compute the probability of:

- a transition
- a chain of transitions

- the probability that the system is in state S_i after n steps
- the expected time that the system spends in state S_i
- the number of $S_i \rightarrow S_j$ transitions in a given chain
-

The **usage chain** is a model of how the system is used or how it is expected to be used. A usage chain may be **informed** (based on previous use or simulation of the system) or **uninformed** (based on guesses or estimates of usage patterns).

The usage chain drives a **test case generator**. **Testing chains** model the expected results of tests.

The **certification team** uses statistical analysis and testing to certify the code in accordance with usage statistics. Testing focuses on the parts of the software likely to be used most intensively. Usage chains are used to

- compute reliability (mean time between failure, mean time to failure); and
- determine when to stop testing.

Advantage of chains: it is possible to predict which software components will be used most heavily and design test accordingly.

Disadvantage of chains: calculations are highly dependent on *a priori* assumptions unless “informed” data is available.

12.5 Results

Errors before testing are typically 5/KLOC (compared to the “normal” industrial rate of 50/KLOC). Most errors are trivial and easy to fix because they are not a consequence of design errors.

Examples:

- A 25 KLOC Pascal program ran on a (US) national network of 20 processors for 10 months without failures (US census).
- An 80 KLOC COBOL Structuring Facility was successfully written in PL/I.
- Aerospace (shuttle) applications from 25 to 500 KLOC were completed with “zero defects”.
- A 65 KLOC Wheelwriter program has been used by millions of customers with no reported failures.

Fig. 13 shows experience obtained from the development of a small application (10 months, 4 people working half time, 2 KLOC FoxBase).

Activity	%
Planning	5
Requirements	39
Specification and design	22
Verification	5
Code and walkthrough	6
Test planning	22
Testing	1

Figure 13: Experience Report for a Cleanroom Project

13 Object Oriented Development

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) (Rumbaugh, Blaha, Premerlani, Eddy, and Lorensen 1991).
- Booch's method (Booch 1991).
- Objectory (Jacobson, Christerson, Jonsson, and Övergaard 1992).
- Class Responsibility Collaborator (CRC) (Wirfs-Brock, Wilkerson, and Wiener 1990).
- Fusion (Coleman et al. 1994).

The Fusion method, 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 requirements 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

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. Fig. 14 shows part of a data dictionary.

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 14: Extract from a Data Dictionary

13.1 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.

Example:

<i>Instructor</i>
Name
Subject

The extension of a class is the set of objects that inhabit it.

Example:

<i>Instructor</i>	
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.

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 constraints** 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 (Jill, 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**).

Example: 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: \longrightarrow reads "specializes to".

Instructor \longrightarrow *FullTimeInstructor*
Instructor \longrightarrow *PartTimeInstructor*
PartTimeInstructor \longrightarrow *GradInstr*
Student \longrightarrow *GraduateStudent*
GraduateStudent \longrightarrow *GradInstr*

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.

13.1.1 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.

13.1.2 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*: Fig. 15 shows an example of a schema.

Operation:	<i>replace-gun</i> .
Description:	Disables the gun and the pump so no more gas can be dispensed; creates a new transaction.

Reads:	Display.
Changes:	supplied <i>pump</i> , supplied <i>gun-status</i> , supplied <i>holster-switch</i> , supplied <i>timer</i> , supplied <i>terminal</i> , new <i>transaction</i> .
Sends:	<i>timer</i> : {start message}, <i>terminal</i> : {transaction}.
Assumes:	<i>holster-switch</i> is released.
Result:	<i>holster-switch</i> is depressed. <i>gun</i> is disabled. If the <i>pump</i> was initially enabled, then: The <i>pump-status</i> is disabled; a <i>transaction</i> has been enabled with the value of the <i>pump</i> display and sent to the terminal; and a start message has been sent to the <i>timer</i> . else: there is no effect.

Figure 15: 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. Fig. 16 shows the syntax provided by Fusion. Using this syntax, we can describe the lifecycle of the vending-machine as

$$((iq \cdot rc)^* \cdot iq \cdot (iq \cdot rc)^* \cdot iq \cdot pb \cdot ec)^*$$

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

$$\mathbf{lifecycle} \text{ GasStation} : ((delivery \cdot payment)^* \mid out-of-service)^* \cdot archive$$

<i>Expr</i>	→	<i>input-event</i>	
		<i>#output-event</i>	
		<i>x . y</i>	<i>x</i> followed by <i>y</i>
		<i>x*</i>	0 or more <i>xs</i>
		<i>x+</i>	1 or more <i>xs</i>
		[<i>x</i>]	0 or 1 <i>x</i>
		<i>x y</i>	<i>x</i> and <i>y</i> concurrently
		(<i>x</i>)	precedence

Figure 16: Syntax for Lifecycle Models

```

delivery = enable-pump .
            [start-pump-motor] .
            [ (preset-volume | preset-amount) ]
            remove-gun-from-holster .
            press-trigger .
            (pulse . #display-amount)* .
            (release-trigger | cut-off-supply) .
            replace-gun .
            #start-timer .
            [turn-off-motor]

```

Figure 17: Expansion of *delivery* lifecycle

The item *delivery* has complex behaviour that is expanded in Fig. 17.

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.

13.2 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.

13.2.1 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 identifies 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.

13.2.2 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).

13.2.3 Class Descriptions

Fusion class descriptions are conventional. Inheritance is indicated by an **isa** clause; attributes are listed; and methods are listed. Fig. 18 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 18: A Fusion Class Description

13.2.4 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.

13.2.5 Principles of Good Design

- 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 graphs⁸
- The root of an inheritance tree should be an abstract class⁹
- Grade levels of inheritance — ensure that the siblings at each level of the inheritance hierarchy should differ in the same way
- Use inheritance for subtyping — subclasses should always respect the properties of their superclass(es)

13.3 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.

13.3.1 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.

13.3.2 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)

⁸Some authors advocate deep inheritance graphs! The best rule is probably to match the opportunities for inheritance to the application.

⁹Some authors say that a concrete class should not have descendants.

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.

13.3.3 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.

13.3.4 Error Handling

In Fusion, an error is defined as *the violation of a precondition*¹⁰. Fusion distinguishes the *detection* of errors and *recovery* from errors.

¹⁰This is not a good definition: it seems preferable to use pre/post-conditions to define the logical properties of the sstem, not its practical properties.

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.¹¹

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**.)

13.3.5 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 $u + v$ can be deallocated neither by the caller nor the callee!

13.4 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;

¹¹This has been compared to wearing your parachute during ground practice but removing it when you fly, but the analogy is weak.

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.¹²

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.

13.5 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.

¹²My view is that this problem can be largely overcome by the use of suitable programming languages and development tools.

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.

13.6 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.

14 Miscellaneous Topics

14.1 Software Tools

Tools should be simple and accessible. Basic toolkit: requirements processor, data dictionary, design processor, conditional compilation, source-level and interactive debuggers, cross-reference generator, call structure analyzer, profiler, configuration manager, word processor, database manager, text editor, file comparer (Glass 1991, pages 85–90).

14.2 Computer-Aided Software Engineering (CASE)

CASE will help, but will not give an order of magnitude improvement (Glass 1991, pages 91–93).

14.3 Single-Point Control

Glass recommends single-point control (Glass 1991, pages 57–60). Examples:

- constant definitions;
- useful functions;
- table-driven code;
- all text in a file.

14.4 Standards

Standards should be simple and reasonable. Standards must be enforced, automatically if possible (Glass 1991, pages 81–83).

15 Case Studies

15.1 RADARSAT Payload Computer Software System

This section is based on an account of the development of software for a radar satellite (Nguyen 1994).

Processor: MIL-STD-1750A. PLC = PayLoad Computer.

Language: Ada with Tartan MIL-STD-1750A Compilation System.

The PLC software has three components:

- IPL (Initial Program Load) ROM to upload operational software.
- Control Program for monitoring hardware and executing tables.

- Application Logic, in the form of uploadable tables.

For an application of the satellite:

- Customer requests image
- Ground station converts request to tables
- Tables are verified by ground spacecraft simulator
- Tables are uploaded to satellite
- PLC executes the tables

The software was developed by a process called CMRS (Composite methodology for Real-Time System) that combines waterfall and spiral process models. In the diagram:

— represents —	
solid line	“information invocation direction”
circle-with-arrow	actual data flow (read, write, update)
dashed line	control flow
rectangle	activities
clouds	(vague) requirements
hexagon	information storage

The phases of CMRS are Requirements Analysis, Design, Implementation, and Operation. Each phase is iterated until the expected model is obtained.

All phases applied object oriented (OO) concepts: OO Analysis, OO Design, OO Programming. Since the target language was Ada, which does not support inheritance fully, inheritance is not used.

Requirements Analysis Phase This phase used Ward-Mellor Real-Time Structured Analysis (RT/SA) to obtain an Environment Model consisting of a context diagram and an event list.

For each input event, the Functional Flow and Data Transform (FFDT) methodology was used to analyse the effect of the event on the system.

OO Analysis was performed using the Shlaer-Mellor methodology and the Cadre Teamwork CASE tool. This yields an Entity Relationship Diagram (ERD), an Object Communication Diagram (OCD), a State Transition Diagram (STD), a Data Flow Diagram (DFD), and a Data Dictionary (DD). Then five models are constructed: the Information Model (IM), the Communication Model (CM), and the State Model (SM), the Process Model (PM), and the Requirements Model (RM).

A throwaway prototype, written in Borland-C++ was used to prototype the RM.

Design Design used Booch's OOD methodology, in which a high-level object is assembled from component objects (bottom-up). The design was prototyped in Ada as follows:

- Map each high-level object to a task.
- Group tasks that are not required to run concurrently.
- Represent each object as a package specification.
- Build a tasking model to identify deadlock conditions.
- Code and test the package bodies.

Seven designs were developed; the last one was used.

Implementation Phase Formal implementation and testing was based on the design prototype. The Software Test Description (STD) document was derived directly from the Software Requirements Specification (SRS) to ensure that all requirements were validated without knowledge of the implementation.

Reviews Formal reviews were required by the customer; informal reviews were also used. Customer representatives attended most of the walkthroughs and code inspections.

- Structured Walkthroughs (Yourdon)
- Structured Inspections (Fagan).

The first phase of integration yielded 16 defects. One defect was a requirements error (it was not possible to change the Temperature Limits table in Standby Mode) and the others were simple coding errors. Fig. 19 shows some other data about the finished system.

Code size	10,150	semicolons
Memory	105,000	bytes (192K available)
Tasks	11	
Subprograms	278	
Test cases	151	
Analysis	17	months
Design	10	months
Implementation	4	months
Testing	6	months

Figure 19: Metrics for RADARSAT Project

15.2 A Software Disaster

The following “disaster” happened to Volvo; it is described by Tom Gilb (1988).

- Large company
- Used consultants
- Study by think-tank (15 person-years over 2 years)
- Large, modern computers available
- Latest methodology was used
- Database software
- Corporation continued to pay (several million dollars) after the budget was exhausted

The result:

- All functions worked correctly
- Some transactions took 20 minutes
- 20,000 transactions/day required — much too slow!
- Adding a new site required 2 years development time
- A new site installation was scheduled every 6 months

After 5 years and 160 person-years of effort the project was cancelled. What went wrong?

- Lack of courage — no whistle blowers. “I know the project will fail, but my part works.”
- Failure to identify critical attributes. (There would be 20,000 transactions/day. The average transaction time must therefore be significantly less than 4 seconds. But this was not mentioned in the requirements!)

16 Bibliography

Albrecht, A. J. and J. E. Gaffney Jr. (1983, November). Software function, source lines of code, and development effort prediction: a software science validation. *IEEE Trans. Software Engineering SE-9*(6), 639–648.

Birtwistle, G., O.-J. Dahl, B. Myrhaug, and K. Nygaard (1973). *SIMULA Begin*. Auerbach (New York).

Boehm, B. W. (1979). Software engineering: R & D trends and defense needs. In P. Wegner (Ed.), *Research Directions in Software Technology*. Cambridge, Mass.: MIT Press.

Boehm, B. W. (1986, March). A spiral model of software development and enhancement. In *Proc. International Workshop on the Software Process and Software Environments*, pp. 21–42. Reprinted in *ACM Software Engineering Notes*, 11:4 (August 1986) and in *IEEE Computer*, 21(5):61–72 (May 1988).

- Boehm, B. W., T. Gray, and T. Seewaldt (1984, May). Prototyping *vs.* specifying: a multi-project experiment. *IEEE Trans. Software Engineering* 10(3), 133–145.
- Booch, G. (1991). *Object Oriented Design with Applications*. Benjamin/Cummings.
- Brooks, F. P. (1978). *The Mythical Man-Month*. Addison-Wesley. Anniversary Edition, 1995.
- Brooks, R. A. (1986, March). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 14–23.
- Coleman, D. et al. (1994). *Object-Oriented Development: the Fusion Method*. Prentice Hall.
- Corbato, F. J. et al. (1965). A new remote-accessed man-machine system. In *Proc. AFIPS 1965 FJCC*, pp. 185–247.
- Dijkstra, E. W. (1968, May). The structure of the THE multiprogramming system. *Comm. ACM* 11(5), 341–346.
- Fenton, N. (1994, March). Software measurement: a necessary scientific basis. *IEEE Trans. Software Engineering* 20(3), 188–198.
- Gilb, T. (1988). *Principles of Software Engineering Management*. Addison-Wesley.
- Glass, R. L. (1991). *Software Conflict: Essays on the Art and Science of Software Engineering*. Yourdon/Prentice Hall. QA 76.758 G53 91.
- Goldberg, A. and D. Robson (1983). *Smalltalk-80: The Language and its Implementation*. Addison-Wesley.
- Hayes, I. (Ed.) (1987). *Specification Case Studies*. Prentice Hall International.
- Henderson-Sellers, B. (1995, October). OO metrics programme. *Object Magazine* 5(6), 73–79,95.
- Jackson, M. (1975). *Principles of Programming Design*. Academic Press.
- Jackson, M. (1983). *System Development*. Prentice Hall International.
- Jacobson, I., M. Christerson, P. Jonsson, and G. Övergaard (1992). *Object-Oriented Software Engineering*. Addison-Wesley.
- Khoshgoftaar, T. M. and P. Oman (1994, September). Guest Editors' introduction: Software metrics. *IEEE Computer* 27(9), 13–15. See also other articles in this issue.
- Kitchenham, B., S. L. Pfleeger, and N. Fenton (1995, December). Towards a framework for software measurement validation. *IEEE Trans. Software Engineering* 21(12), 929–944.
- Meyer, B. (1988). *Object-oriented Software Construction*. Prentice Hall International. Second Edition, 1997.

- Mills, H. D., M. Dyer, and R. Linger (1987, September). Cleanroom software engineering. *IEEE Software* 4(5), 19–25.
- Nguyen, M. T. (1994, July). Development of RADARSAT payload computer software system. *SPAR Journal of Engineering and Technology* 3, 45–58.
- Parnas, D. L. (1972, December). On the criteria to be used in decomposing systems into modules. *Comm. ACM* 15(12), 1053–1058.
- Parnas, D. L. (1979, March). Designing software for ease of extension and contraction. *IEEE Trans. Software Engineering* 5(2), 128–138.
- Parrish, A. S. and S. H. Zweben (1995, December). On the relationships among the all-uses, all-DU-paths, and all-edges testing criteria. *IEEE Trans. Software Engineering* 21, 1006–1009.
- Porter, A., H. Siy, C. Toman, and L. Votta (1995). An experiment to assess the cost-benefits of code inspections in large scale software development. In G. E. Kaiser (Ed.), *Proc. Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 92–103.
- Royce, W. W. (1970). Managing the development of large software systems: Concept and techniques. In *1970 WESCON Technical Papers, Western Electric Show and Convention*, pp. A/1–1–A/1–9. Reprinted in *Proceedings of the 11th International Conference on Software Engineering*, Pittsburgh, May 1989, pp. 328–338.
- Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice Hall.
- Russell, G. (1991, January). Inspection in ultralarge-scale development. *IEEE Software* 8(1), 25–31.
- Selby, R. W., V. R. Basili, and F. T. Baker (1987, September). Cleanroom software development: an empirical evaluation. *IEEE Trans. Software Engineering* 13(9), 1027–1037.
- Spivey, J. (1988). *Understanding Z*. Cambridge University Press.
- Spivey, J. (1989). *The Z Notation: A Reference Manual*. Prentice Hall International.
- Symons, C. R. (1988, January). Function point analysis: difficulties and improvements. *IEEE Trans. Software Engineering* 14(1), 2–11.
- Wirfs-Brock, R. and B. Wilkerson (1989). Object oriented design: a responsibility driven approach. In *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, pp. 71–75.
- Wirfs-Brock, R., B. Wilkerson, and L. Wiener (1990). *Designing Object-Oriented Software*. Prentice Hall.
- Yourdon, E. and L. Constantine (1979). *Structured Design*. Prentice Hall.