

# SOEN 345    Software Quality Control

## Lecture Notes

Peter Grogono

January 2002

Department of Computer Science  
Concordia University  
Montreal, Quebec

## Contents

<b>1</b>	<b>About the Course</b>	<b>1</b>
1.1	General Information . . . . .	1
1.2	Motivation . . . . .	2
1.3	Failures, Defects, and Errors . . . . .	7
<b>2</b>	<b>Testing</b>	<b>10</b>
2.1	Black Boxes and Glass Boxes . . . . .	10
2.2	Equivalence Classes . . . . .	11
2.3	Clear Box Testing . . . . .	15
2.4	Automated Tests . . . . .	16
2.5	Regression Testing . . . . .	17
2.6	Cleanroom testing . . . . .	17
2.7	Testing in XP . . . . .	20
2.8	Kinds of Testing . . . . .	21
2.9	Modelling . . . . .	22
2.10	Conclusion . . . . .	30
<b>3</b>	<b>Coding</b>	<b>31</b>
3.1	Common Problems . . . . .	31
3.2	Basic Concepts . . . . .	33
3.3	General Coding Rules . . . . .	38
3.4	Physical Design . . . . .	46
3.5	Friends and Enemies . . . . .	50
3.6	Levels . . . . .	53
3.7	Levelization . . . . .	55
3.8	Levelization: Summary . . . . .	67
3.9	Insulation . . . . .	68
3.10	Designing a Function . . . . .	70
3.11	Summary . . . . .	77
<b>4</b>	<b>Interlude: Java</b>	<b>79</b>
4.1	Objects in Java . . . . .	80
4.2	Method Parameters . . . . .	82
4.3	Strings in Java . . . . .	83
4.4	Class Parameters . . . . .	85
4.5	Types . . . . .	85
4.6	Confusion . . . . .	86

<b>5</b>	<b>Implementing Design Patterns</b>	<b>87</b>
5.1	Singleton . . . . .	87
5.2	State . . . . .	88
5.3	Visitor . . . . .	96
5.4	Strategy . . . . .	105
5.5	Composite . . . . .	107
5.6	Designing with Patterns . . . . .	109
5.7	Assessing Design Quality . . . . .	114
<b>6</b>	<b>Varia</b>	<b>115</b>
6.1	Automated Testing . . . . .	115
6.2	Assertions increase test effectiveness . . . . .	115
6.3	Pisces Safety Net . . . . .	116
6.4	Do Standards Improve Quality? . . . . .	118
6.5	<i>N</i> -Version Programming . . . . .	119
6.6	Testing for Reliability . . . . .	121
6.7	Aspect-Oriented Programming . . . . .	123

## List of Figures

1	A Defect Type Standard . . . . .	9
2	State transition diagram and the corresponding matrix . . . . .	18
3	Input for transition-based test generator . . . . .	20
4	Kinds of testing . . . . .	22
5	Effect of $\delta$ on predicted testing time . . . . .	26
6	Declarations and Definitions . . . . .	34
7	Internal and external linkage . . . . .	34
8	Good and bad things in a header file . . . . .	35
9	Inheritance Hierarchies . . . . .	39
10	Layering Hierarchies . . . . .	39
11	Outline of a component . . . . .	46
12	Hidden dependencies . . . . .	50
13	A system with binary tree structure . . . . .	54
14	Improved code for graphical editor (guards omitted) . . . . .	59
15	Effect of class <code>ShapeConstructor</code> on NCCD . . . . .	60
16	<b>State</b> Pattern . . . . .	89
17	Vending Machine Transitions . . . . .	90
18	Comparison of traversal techniques . . . . .	104

19	Generalized double-entry . . . . .	110
20	Functions of a Form . . . . .	111
21	Various kinds of Form . . . . .	111
22	Functions of a Field . . . . .	112
23	Various kinds of Field . . . . .	112

# SOEN 345 Software Quality Control

## 1 About the Course

### 1.1 General Information

**Background** SOEN 345 *Software Quality Control* is a study of what “quality” means in the context of software development and how we can achieve a level of quality that is both acceptable and predictable.

You should understand basic concepts of measurement and metrics as they relate to software development, from SOEN 337, and you should also be familiar with the basic principles of software development, from SOEN 341.

**Text Book** There is no text book for this course. Course notes and references for reading will be provided during the term.

I have made use of a number of articles and books in preparing the course. Some of them are referenced in these notes, and there is a complete bibliography on the web page. The references are provided for your convenience, in case you want to pursue topics in greater depth. You are not expected to read the references unless specifically told to do so.

**Course Outline** What is “quality”? How do we assess the quality of software? Qualitative aspects of: testing; coding; design, specification; and requirements. Inspections and reviews. Quantitative approaches to quality control. Processes: prototyping, incremental development, cleanroom, extreme programming. PSP and TSP. Industrial case studies.

**Resources** You can use your Internet browser to obtain assignments, example programs, lecture notes, and other information about the course at

<http://www.cs.concordia.ca/~teaching/soen345/>

**Computing Facilities** You will be required to complete a few simple programming tasks for this course but there will not be a major programming project. You will need access to a computer that has a C++ or Java development environment.

### Evaluation

- ◇ You will be evaluated on the basis of assignments (20%), midterm examinations (20%), and a final examination (60%).
- ◇ There will be four or five assignments. The assignments will be primarily theoretical.

**Overlap** Some of the topics covered in this course are also covered in other SOEN courses. This is not surprising, because the need to achieve quality is ubiquitous in software engineering. Furthermore, some topics that are touched upon elsewhere, such as testing, will be treated with more depth in this course.

## 1.2 Motivation

People have reported problems with software. The ACM Risks Forum<sup>1</sup> has a large repository of problems that have been caused by software defects. Highlights include:

- ◊ Thousands of gallons of radioactive water released from the Bruce nuclear power station.
- ◊ A \$32 billion overdraft that cost the Bank of New York \$5 million in interest — for one day!
- ◊ Patients killed and injured by the Therac-25 radiation therapy machine (see SOEN 341 notes).
- ◊ 250 engineers spent a year finding and correcting 30,000 defects in Windows NT.

We can do some simple calculations based on the example. Suppose that it costs \$100K/yr to employ an engineer (this is an underestimate: we should include not only the engineer's salary but also the cost of office space, equipment, etc.). Testing NT cost about \$25M and each defect cost about \$800 to detect and correct. Here is more data in this vein:

- ◊ The most expensive defects to repair are those that customers report. In one year, IBM processed 13,000 customer-reported defects at a cost of \$250M. The cost, about \$20K/defect, includes both correcting the defect and ensuring that customers have them. (Humphrey 1997, page 159)

Specific quotations:

- ◊ Word 3.0 for the Macintosh, delivered in February 1987 after having been promised for July 1986, had approximately seven hundred bugs — several which destroyed data or “crashed” the program. Microsoft had to ship a free upgrade to customers within two months of the original release, costing more than \$1 million. (Cusamano and Selby 1995)
- ◊ Studies have shown that for every new large-scale software system that is put into action, two others are cancelled. The *average* software development overshoots its schedule by half; *large* projects generally do worse. And some three-quarters of all large systems are “operational failures” that either do not function as intended or are not used at all. (Gibbs 1994)
- ◊ Microsoft's newest version of Windows, billed as the most secure ever, contains several serious flaws that allow hackers to steal or destroy a victim's data files across the Internet or implant rogue computer software. . . . A Microsoft official acknowledged that the risk to consumers was unprecedented because the glitches allow hackers to seize control of all Windows XP operating system software without requiring a computer user to do anything except connect to the Internet. Microsoft made available on its Web site a free fix for both home and professional editions of Windows XP and forcefully urged consumers to install it immediately. . . . Ted Bridis, Associated Press, 20 Dec 2001.

<http://digitalmass.boston.com/news/2001/12/20/microsoft.html>

[Comment by Peter G. Neumann: The vulnerabilities involve the universal plug-and-play features, and were discovered by a team at eEye Digital Security Inc. of Aliso Viejo, Calif., led by Marc Maiffret. There were also subsequent reports that the free fix was not adequate. By the way, the free fix can arrive automatically with “drizzle”, which allows MS to upgrade for you. *PGN says beware of mechanisms that offer automatic upgrades*, no matter how convenient they may seem. The article also quotes Microsoft's

---

<sup>1</sup><http://catless.ncl.ac.uk/Risks/>

departing corporate security officer, Howard Schmidt, who is about to join Richard Clarke in the White House, expressing frustration about continuing threats from overflows: “I’m still amazed that we allow these things to occur”.; PGN] (Risks Forum, December 2001)

- ◇ Most US enterprises are seriously deficient in defect removal technology; they tend to depend primarily on testing . . . . The actual technologies of defect removal are advancing rapidly, and leading-edge enterprises can now exceed 99% in cumulative defect removal efficiently. Unfortunately, the overall US norms for defect removal appear to hover around 75%. (Jones 1991)

On the other hand, there are success stories:

- ◇ The Ericsson Telecom OS32 operating system . . . . was a 70–person project lasting 18 months, producing 350 KLOC of code in assembler language and C. The defect density in testing was only 1/KLOC. The project was a great success . . . . (Stavely 1999, page 6)

How can we account for these differing experiences? Is there something that Ericsson does right and Microsoft does wrong? At least for the applications mentioned, Ericsson seems to have achieved a level of quality that Microsoft has not. This suggests that the lack of quality in software is not due to ignorance, but rather to inadequate practices.

In order to discuss these issue, we will need answers to several questions:

- ◇ Is there a “quality” problem in software development?
- ◇ What, exactly, do we mean by “quality”?
- ◇ Can we measure quality?
- ◇ If we can measure quality, what steps can we take to achieve it?

**The -ility Problem** Many discussions of software quality are given in terms of “-ility words”. We are told that desirable features of software include maintainability, reliability, and usability. But, since these terms are not defined precisely, it is difficult to know how to achieve them and even how to know whether we have achieved them.

In this course, we will try to avoid -ilities. Instead, we will look for ways of measuring software quality and ways of ensuring that our software performs well when analyzed by these metrics.

**Organization of the Course** The sequence of topics will loosely follow the waterfall model, but backwards. The waterfall model proceeds from abstract concepts (requirements and specifications) to concrete methodologies (coding and testing). Reversing the sequence yields a bottom-up approach in which we create a foundation of sound coding and testing practices, and then develop equally sound techniques for design and specification that ultimately come first.

**Initial Assumptions** There are a few aspects of quality that more or less everyone seems to agree upon. They are listed here and will not be mentioned much afterwards.

- ◇ Quality is in the eye of the user. A program might be “perfect” — in the sense that it meets all its requirements and is free of defects — but useless if the users don’t understand how to use it, don’t want to use it, or are frightened of using it.
- ◇ Quality is not something that is added at the end of development — for example, by a Quality Control Team — in the way that chocolate powder is sprinkled on cappuccino.

It is more like the bay leaf that is added to the soup or stew at an early stage and kept in it until the very end.

- ◊ Quality is not something that applies to a particular person or product. For people, quality is an approach to life, an attitude that does not accept inferior work and persists throughout their careers. For products, quality applies to institutions and companies rather than the particular products, or product families, that they produce.

The goals of the course include:

- ◊ recognizing good quality work;
- ◊ improving the quality of your own work;
- ◊ contributing to high-quality team work;
- ◊ obtaining high-quality work from a team;
- ◊ developing a life-style based on high quality.

Most people have one or more activities that they enjoy doing and strive to do better. Examples include: sports activities such as running, cycling, swimming, skiing, mud-wrestling, and billiards; recreational activities such as walking, bird-watching, sunbathing, and paragliding; games such as chess, bridge, poker, and tic-tac-toe. The common feature of these activities is that we not only try to do better but we also use *metrics* to monitor our progress.

Programming is a skill that we can develop in the same way as one of these activities. You should aim not only to become a better programmer, but also to *know* that you are a better programmer. This means monitoring your performance and noting how it improves — or being concerned about lack of improvement.

2

## The Hall Of Shame

Good-Nature and Good-Sense must ever join;  
To err is Humane; to Forgive, Divine. (Pope 1711)

We all make mistakes; errors are not something we should be ashamed of. The trick is to become familiar with our own patterns of making mistakes so that we can anticipate and recognize them and perhaps make fewer in future.

Here are some classic errors that I have made in recent work (noted because of the remarkable response of the C++ compiler).

This class declaration:

```
#ifndef CONTROL_H
#define CONTROL_H

class SliderBase
{
public:
    SliderBase(int pwin, int x, int y, int w, int h);
    display();
    reshape(int w, int h);
    mouse(int x, int y);
};
```

```

    int getWin() { return win; }
private:
    int pwin;        // Parent window
    int win;         // This window
    int x;
    int y;
    int w;
    int h;
};

```

produced:

```

e:\graphics\controls\control.h(7) : error C2629: unexpected 'class SliderBase ('
e:\graphics\controls\control.h(7) : error C2238: unexpected token(s) preceding ';'
e:\graphics\controls\control.cpp(32) : error C2511: 'SliderBase::SliderBase' :
overloaded member function 'void (int,int,int,int,int)' not found in 'SliderBase'
    e:\graphics\controls\control.h(5) : see declaration of 'SliderBase'
e:\graphics\controls\control.cpp(55) : fatal error C1004: unexpected end of file found

```

This innocuous-looking code:

```

for (int p = 0; p < NUMPOINTS; p++)
{
    cout << p << ' ';
    for (int q = 0; q < 3; q++)
        cout << points[p][q] << ' ';
    cout << endl;
}

```

produced 18 errors and 4 warnings:

```

E:\Graphics\Bezier\bezcurve.c(32) : error C2143: syntax error : missing ';' before 'type'
E:\Graphics\Bezier\bezcurve.c(32) : error C2143: syntax error : missing ';' before 'type'
E:\Graphics\Bezier\bezcurve.c(32) : error C2143: syntax error : missing ')' before 'type'
E:\Graphics\Bezier\bezcurve.c(32) : error C2143: syntax error : missing ';' before 'type'
E:\Graphics\Bezier\bezcurve.c(32) : error C2065: 'p' : undeclared identifier
E:\Graphics\Bezier\bezcurve.c(32) : warning C4552: '<' : operator has no effect; expected operator with side-effect
E:\Graphics\Bezier\bezcurve.c(32) : error C2059: syntax error : ')'
E:\Graphics\Bezier\bezcurve.c(33) : error C2143: syntax error : missing ';' before '{'
E:\Graphics\Bezier\bezcurve.c(34) : error C2065: 'cout' : undeclared identifier
E:\Graphics\Bezier\bezcurve.c(34) : warning C4552: '<<' : operator has no effect; expected operator with side-effect
E:\Graphics\Bezier\bezcurve.c(35) : error C2143: syntax error : missing ';' before 'type'
E:\Graphics\Bezier\bezcurve.c(35) : error C2143: syntax error : missing ';' before 'type'
E:\Graphics\Bezier\bezcurve.c(35) : error C2143: syntax error : missing ')' before 'type'
E:\Graphics\Bezier\bezcurve.c(35) : error C2143: syntax error : missing ';' before 'type'
E:\Graphics\Bezier\bezcurve.c(35) : error C2065: 'q' : undeclared identifier
E:\Graphics\Bezier\bezcurve.c(35) : warning C4552: '<' : operator has no effect; expected operator with side-effect
E:\Graphics\Bezier\bezcurve.c(35) : error C2059: syntax error : ')'
E:\Graphics\Bezier\bezcurve.c(36) : error C2146: syntax error : missing ';' before identifier 'cout'
E:\Graphics\Bezier\bezcurve.c(36) : error C2296: '<<' : illegal, left operand has type 'float'
E:\Graphics\Bezier\bezcurve.c(36) : error C2297: '<<' : illegal, right operand has type 'float'
E:\Graphics\Bezier\bezcurve.c(37) : error C2065: 'endl' : undeclared identifier
E:\Graphics\Bezier\bezcurve.c(37) : warning C4552: '<<' : operator has no effect; expected operator with side-effect

```

Finally, this little data declaration:

```

GLfloat PT1[12][2] =
{

```

```

    { 0,      -0.1f  },
    { 0.04f,  0      },
    { 0.04f,  0.2f  },
    { 0.04f,  0.4f  },
    { 0.044f, 0.6f  },
    { 0.056f, 0.75f },
    { 0.1f,   0.75f },
    { 0.064f, 0.8f  },
    { 0.036f, 0.85f },
    { 0.016f, 0.9f  },
    { 0.004f, 0.95f },
    { 0f,     1     }
};

```

produced no less than 66 error messages:

```

E:\Graphics\CUGL\cugl.cpp(982) : error C2059: syntax error : 'bad suffix on number'
E:\Graphics\CUGL\cugl.cpp(982) : error C2061: syntax error : identifier 'f'
E:\Graphics\CUGL\cugl.cpp(983) : error C2143: syntax error : missing ';' before '}'
E:\Graphics\CUGL\cugl.cpp(983) : error C2143: syntax error : missing ';' before '}'
E:\Graphics\CUGL\cugl.cpp(983) : error C2143: syntax error : missing ';' before '}'
E:\Graphics\CUGL\cugl.cpp(986) : error C2143: syntax error : missing ';' before 'for'
E:\Graphics\CUGL\cugl.cpp(986) : error C2143: syntax error : missing ')' before ';'
E:\Graphics\CUGL\cugl.cpp(986) : error C2143: syntax error : missing ';' before '<'
E:\Graphics\CUGL\cugl.cpp(986) : error C2501: 'x' : missing storage-class or type specifiers
E:\Graphics\CUGL\cugl.cpp(986) : error C2143: syntax error : missing ';' before '<'
E:\Graphics\CUGL\cugl.cpp(986) : error C2143: syntax error : missing ';' before '++'
E:\Graphics\CUGL\cugl.cpp(986) : error C2501: 'x' : missing storage-class or type specifiers
E:\Graphics\CUGL\cugl.cpp(986) : error C2086: 'x' : redefinition
E:\Graphics\CUGL\cugl.cpp(986) : error C2143: syntax error : missing ';' before '++'
E:\Graphics\CUGL\cugl.cpp(986) : error C2059: syntax error : ')'
E:\Graphics\CUGL\cugl.cpp(987) : error C2143: syntax error : missing ')' before ';'
E:\Graphics\CUGL\cugl.cpp(987) : error C2143: syntax error : missing ';' before '<'
E:\Graphics\CUGL\cugl.cpp(987) : error C2501: 'y' : missing storage-class or type specifiers
E:\Graphics\CUGL\cugl.cpp(987) : error C2143: syntax error : missing ';' before '<'
E:\Graphics\CUGL\cugl.cpp(987) : error C2143: syntax error : missing ';' before '++'
E:\Graphics\CUGL\cugl.cpp(987) : error C2501: 'y' : missing storage-class or type specifiers
E:\Graphics\CUGL\cugl.cpp(987) : error C2086: 'y' : redefinition
E:\Graphics\CUGL\cugl.cpp(987) : error C2143: syntax error : missing ';' before '++'
E:\Graphics\CUGL\cugl.cpp(987) : error C2059: syntax error : ')'
E:\Graphics\CUGL\cugl.cpp(990) : error C2501: 'glNewList' : missing storage-class or type specifiers
E:\Graphics\CUGL\cugl.cpp(990) : error C2373: 'glNewList' : redefinition; different type modifiers
    c:\program files\microsoft visual studio\vc98\include\gl\gl.h(1329) : see declaration of 'glNewList'
E:\Graphics\CUGL\cugl.cpp(990) : error C2078: too many initializers
E:\Graphics\CUGL\cugl.cpp(991) : error C2501: 'revolve' : missing storage-class or type specifiers
E:\Graphics\CUGL\cugl.cpp(991) : error C2373: 'revolve' : redefinition; different type modifiers
    e:\graphics\cugl\cugl.h(348) : see declaration of 'revolve'
E:\Graphics\CUGL\cugl.cpp(991) : error C2078: too many initializers
E:\Graphics\CUGL\cugl.cpp(992) : error C2556: 'int __cdecl glEndList(void)' :
    overloaded function differs only by return type from 'void __stdcall glEndList(void)'
    c:\program files\microsoft visual studio\vc98\include\gl\gl.h(1227) : see declaration of 'glEndList'
E:\Graphics\CUGL\cugl.cpp(992) : error C2373: 'glEndList' : redefinition; different type modifiers
    c:\program files\microsoft visual studio\vc98\include\gl\gl.h(1227) : see declaration of 'glEndList'
E:\Graphics\CUGL\cugl.cpp(995) : error C2501: 'setMaterial' : missing storage-class or type specifiers
E:\Graphics\CUGL\cugl.cpp(995) : error C2373: 'setMaterial' : redefinition; different type modifiers
    e:\graphics\cugl\cugl.h(451) : see declaration of 'setMaterial'
E:\Graphics\CUGL\cugl.cpp(996) : error C2556: 'int __cdecl glPushMatrix(void)' :
    overloaded function differs only by return type from 'void __stdcall glPushMatrix(void)'
    c:\program files\microsoft visual studio\vc98\include\gl\gl.h(1362) : see declaration of 'glPushMatrix'
E:\Graphics\CUGL\cugl.cpp(996) : error C2373: 'glPushMatrix' : redefinition; different type modifiers
    c:\program files\microsoft visual studio\vc98\include\gl\gl.h(1362) : see declaration of 'glPushMatrix'
E:\Graphics\CUGL\cugl.cpp(997) : error C2501: 'glRotatef' : missing storage-class or type specifiers
E:\Graphics\CUGL\cugl.cpp(997) : error C2373: 'glRotatef' : redefinition; different type modifiers

```

```

c:\program files\microsoft visual studio\vc98\include\gl\gl.h(1400) : see declaration of 'glRotatof'
E:\Graphics\CUGL\cugl.cpp(997) : error C2078: too many initializers
E:\Graphics\CUGL\cugl.cpp(998) : error C2501: 'glCallList' : missing storage-class or type specifiers
E:\Graphics\CUGL\cugl.cpp(998) : error C2373: 'glCallList' : redefinition; different type modifiers
c:\program files\microsoft visual studio\vc98\include\gl\gl.h(1160) : see declaration of 'glCallList'
E:\Graphics\CUGL\cugl.cpp(999) : error C2556: 'int __cdecl glPopMatrix(void)' :
overloaded function differs only by return type from 'void __stdcall glPopMatrix(void)'
c:\program files\microsoft visual studio\vc98\include\gl\gl.h(1357) : see declaration of 'glPopMatrix'
E:\Graphics\CUGL\cugl.cpp(999) : error C2373: 'glPopMatrix' : redefinition; different type modifiers
c:\program files\microsoft visual studio\vc98\include\gl\gl.h(1357) : see declaration of 'glPopMatrix'
E:\Graphics\CUGL\cugl.cpp(1002) : error C2501: 'setMaterial' : missing storage-class or type specifiers
E:\Graphics\CUGL\cugl.cpp(1002) : error C2373: 'setMaterial' : redefinition; different type modifiers
e:\graphics\cugl\cugl.h(451) : see declaration of 'setMaterial'
E:\Graphics\CUGL\cugl.cpp(1003) : error C2556: 'int __cdecl glPushMatrix(void)' :
overloaded function differs only by return type from 'void __stdcall glPushMatrix(void)'
c:\program files\microsoft visual studio\vc98\include\gl\gl.h(1362) : see declaration of 'glPushMatrix'
E:\Graphics\CUGL\cugl.cpp(1004) : error C2501: 'glRotatof' : missing storage-class or type specifiers
E:\Graphics\CUGL\cugl.cpp(1004) : error C2373: 'glRotatof' : redefinition; different type modifiers
c:\program files\microsoft visual studio\vc98\include\gl\gl.h(1400) : see declaration of 'glRotatof'
E:\Graphics\CUGL\cugl.cpp(1004) : error C2078: too many initializers
E:\Graphics\CUGL\cugl.cpp(1005) : error C2501: 'glCallList' : missing storage-class or type specifiers
E:\Graphics\CUGL\cugl.cpp(1005) : error C2373: 'glCallList' : redefinition; different type modifiers
c:\program files\microsoft visual studio\vc98\include\gl\gl.h(1160) : see declaration of 'glCallList'
E:\Graphics\CUGL\cugl.cpp(1006) : error C2556: 'int __cdecl glPopMatrix(void)' :
overloaded function differs only by return type from 'void __stdcall glPopMatrix(void)'
c:\program files\microsoft visual studio\vc98\include\gl\gl.h(1357) : see declaration of 'glPopMatrix'
E:\Graphics\CUGL\cugl.cpp(1009) : error C2501: 'setMaterial' : missing storage-class or type specifiers
E:\Graphics\CUGL\cugl.cpp(1009) : error C2373: 'setMaterial' : redefinition; different type modifiers
e:\graphics\cugl\cugl.h(451) : see declaration of 'setMaterial'
E:\Graphics\CUGL\cugl.cpp(1010) : error C2556: 'int __cdecl glPushMatrix(void)' :
overloaded function differs only by return type from 'void __stdcall glPushMatrix(void)'
c:\program files\microsoft visual studio\vc98\include\gl\gl.h(1362) : see declaration of 'glPushMatrix'
E:\Graphics\CUGL\cugl.cpp(1011) : error C2501: 'glCallList' : missing storage-class or type specifiers
E:\Graphics\CUGL\cugl.cpp(1011) : error C2373: 'glCallList' : redefinition; different type modifiers
c:\program files\microsoft visual studio\vc98\include\gl\gl.h(1160) : see declaration of 'glCallList'
E:\Graphics\CUGL\cugl.cpp(1012) : error C2556: 'int __cdecl glPopMatrix(void)' :
overloaded function differs only by return type from 'void __stdcall glPopMatrix(void)'
c:\program files\microsoft visual studio\vc98\include\gl\gl.h(1357) : see declaration of 'glPopMatrix'
E:\Graphics\CUGL\cugl.cpp(1014) : error C2501: 'glDeleteLists' : missing storage-class or type specifiers
E:\Graphics\CUGL\cugl.cpp(1014) : error C2373: 'glDeleteLists' : redefinition; different type modifiers
c:\program files\microsoft visual studio\vc98\include\gl\gl.h(1210) : see declaration of 'glDeleteLists'
E:\Graphics\CUGL\cugl.cpp(1014) : error C2078: too many initializers
E:\Graphics\CUGL\cugl.cpp(1015) : error C2143: syntax error : missing ';' before '}'
E:\Graphics\CUGL\cugl.cpp(1015) : error C2143: syntax error : missing ';' before '}'
E:\Graphics\CUGL\cugl.cpp(1015) : error C2143: syntax error : missing ';' before '}'
E:\Graphics\CUGL\cugl.cpp(1019) : error C2143: syntax error : missing ';' before '{'
E:\Graphics\CUGL\cugl.cpp(1019) : error C2447: missing function header (old-style formal list?)

```

### 1.3 Failures, Defects, and Errors

**Terminology** Terminology is important. Consider the following conversation:

Manager: “So how’s it going?”  
Programmer: “Oh, it’s fine — just a few bugs left.” Manager: [walks away, relieved]

Each of those “bugs”, however, could cause the program to freeze, hang, crash, or blow-up (choose your jargon), perhaps while a client was running it. Let’s revise the dialogue:

Manager: “So how’s it going?”  
Programmer: “Oh, it’s fine — just a few time-bombs left.”  
Manager: “That’s not very encouraging.”

Here is the official terminology for programming errors (Lethbridge and Laganière 2001, Chapter 10):

- ◇ A *failure* is an unacceptable behaviour exhibited by a system.
- ◇ A *defect* is a flaw in any aspect of the system, including the requirements, the design, and the code, that contributes, or may contribute, to the occurrence of one or more failures.
- ◇ An *error* is a mistake or inappropriate decision by a software developer that leads to the introduction of a defect into the system.
- ◇ Errors are *injected* into the system by software engineers.

The most obvious kind of failure is a program crash, but there are many other kinds. The other kinds are actually more important, because complete crashes are relatively rare. An error in calculation, omitting to update a database, a keystroke that puts the user into an unexpected situation, are all failures.

The relationship between defects and failures may be many-to-many. One defect may cause several different kinds of failure, and some failures depend on the existence of several defects. This is one reason why we must be very careful in testing: it's too easy to say "Here's the problem!" without realizing that "here" is only part of the problem. When we find a defect, we must ask "Is this defect a *necessary and sufficient condition* for the failure?" More on this later.

We talk about "injecting errors" to avoid the viewpoint that errors "just happen". It wasn't that "the dog ate your homework"; what happened was that you left your homework on the floor while knowingly in an environment containing a hungry dog.

Since a defect is "a flaw in *any* aspect of the system", we can use the definition of defect as a basis for a definition of quality:

**Quality** is a property of software that has zero – or perhaps a very small number — of defects.

Points to note:

- ◇ We usually do not know that software has defects until it has failed. Moreover, if a program has not failed, we cannot infer that it has no defects. These points are perhaps obvious but they are worth bearing in mind to avoid a false sense of security.
- ◇ A weakness of the definition of quality is that it does not capture the subjective component of quality: different people may have different opinions about the same software. Many people regard the fact that MS Word is WYSIWYG (or, as Brian Kernighan puts it: "what you see is all you've got"), some (including me) regard it as a defect — there is not enough information on the screen to tell me why the text appears as it does, forcing me to make guesses about how Word works.
- ◇ Since "defect" is defined in terms of "failure", we have to decide what we mean by "failure". Does it mean "fail to conform to requirements" or "fail to make the users happy"?

**Classification of Defects** It is helpful to divide defects into categories. In a software development organization, the same categories should be used by everyone involved in development. Using a standard categorization helps managers to track defects and identify problem

Number	Type	Description
10	Documentation	comments, messages
20	Syntax	spelling, punctuation, typos, instruction formats
30	Build, Package	change management, library, version control
40	Assignment	declaration, duplicate names, scope, limits
50	Interface	procedure calls and references, I/O, user formats
60	Checking	error messages, inadequate checking
70	Data	structure, content
80	Function	logic, pointers, loops, recursion, computation, function defects
90	System	configuration, timing, memory
100	Environment	design, compile, test, other support system problems

Figure 1: A Defect Type Standard

areas. Table 1 shows a Defect Type Standard developed at one of IBM's research laboratories (Chillarege, Bhandari, Chaar, Halliday, Moebus, Ray, and Wong 1992) (reproduced as Table 12.1 of (Humphrey 1997)). The categories in this standard were derived by studying the defects in a wide range of IBM products. A different organization, producing different kinds of software, might come up with a quite different set of categories.

There are ten categories in Table 1, numbered 10, 20, . . . , 100. Intervening numbers are added only when it is clear what they should be. For example, if defect reports show that the most common syntax errors are omitting ";" and writing "=" instead of "==", then these defects might be assigned subcategories 21 and 22.

You will find it helpful to make up your own list of defect types.

- ◇ Start by simply recording the errors that you inject into your code.
- ◇ When you see a pattern developing, introduce suitable major categories.
- ◇ When you have enough information (at least a few hundred errors), you can introducing subcategories.

Becoming familiar with the types of error that you inject will help you in two ways.

- ◇ You will become more careful when writing the kind of code where you are prone to making errors.
- ◇ You will know what to look for when inspecting your code and debugging.

I have a tendency to inject errors into `for` loops: updating the wrong index; confusing loop index with the limit; getting the end points wrong. Since I am aware of this, I have learned to write `for` loops very carefully and to check them first when my programs hang.

## 2 Testing

Informally, we may act as if “testing” means “try it and see what happens”. For software engineering, we need something a bit more precise. (Lethbridge and Laganière 2001, page 349)

- ◇ *Testing* is the process of deliberately trying to cause failures in a system in order to detect defects in the system.
- ◇ A test is *effective* if it reveals defects.
- ◇ A test is *efficient* if it can be performed quickly and has a high probability of revealing a defect.
- ◇ A *high-yield test strategy* is a collection (or recipe for creating a collection) of effective and efficient tests.

Of course, we do not bring things into existence just by defining them. We still have to show that effective and efficient tests exist and that there are ways of finding or generating them.

### 2.1 Black Boxes and Glass Boxes

The following terminology is common in engineering.

- ◇ A *black-box test* proceeds by providing certain inputs to the system and observing the outputs. Black-box testing can be carried out (in principle) without any knowledge of the internal structure of the system.
- ◇ A *glass-box test* is designed by using knowledge of the internal structure of the system.

Glass-box testing is sometimes called “white-box” testing (since “white” is an antonym for “black”). Many people consider this usage misleading because a box can be white and yet opaque; “glass box” suggests that we can see inside, which is the intent of the expression “glass-box testing”. An acceptable alternative is “clear-box testing”.

**Example 1.** It is easy to illustrate the difference between black-box and glass-box testing. Suppose that we are testing a program with the following specification:

Given real numbers  $a$ ,  $b$ , and  $c$ , find real values of  $x$ , if any exist, such that  $ax^2 + bx + c = 0$ .

In black-box testing, we would provide various values of  $a$ ,  $b$ , and  $c$ , and check that the program’s outputs satisfy the equation to within some tolerance.

In glass-box testing, we would look at the code and discover a statement of the form

```
if (sqrb - sqrt(sqrb - 4 * a * c) >= 0)
    .... // (1)
else
    .... // (2)
```

Having found this statement, we would design tests that execute both of the statements (1) and (2). □

## 2.2 Equivalence Classes

There is one kind of testing that we define only in order to immediately dismiss it from consideration:

- ◇ A set of tests is *exhaustive* if it checks the system under all possible conditions.

There are extremely few situations in which exhaustive testing is feasible. When we have changed a light-bulb, we generally test the repaired system by ensuring that the bulb lights up when the switch is on and does not light up when the switch is off. (Actually, this is not really exhaustive: what does the bulb do during a power surge or when hit with a hammer?)

Unfortunately, few programs are as simple as this. A program which reads one number — to be specific, let’s say a C++ value of type `int` — should be able to accept  $2^{31} - 1 \approx 2$  billion valid inputs. A modern PC could run through the values of `int` quite quickly — but who is going to look at the results?

Since we cannot test exhaustively, we look for groups of input values for which we expect that the algorithm will behave in a similar way. In Example 1, we expect a quadratic equation solver to use one tactic if  $b^2 - 4ac \geq 0$  and another tactic if  $b^2 - 4ac < 0$  (and it might even distinguish the case  $b^2 - 4ac = 0$ ). This suggests an equivalence relation, defined like this:

Inputs  $I$  and  $I'$  are *equivalent* (notation:  $I \sim I'$ ) if we believe that the system will use the same strategy for each input.

Although the definition is informal, it is easy to see that the relation, if it has any meaning at all, is an equivalence relation (but see below as well!):

- ◇ Reflexive:  $I \sim I$ , because the system behaves in only one way for any particular input.
- ◇ Symmetric: if we believe  $I \sim I'$ , it would be unreasonable not to believe  $I' \sim I$ .
- ◇ Transitive: similarly, if we believe  $I_1 \sim I_2$  and  $I_2 \sim I_3$ , it would be unreasonable not to believe  $I_1 \sim I_3$ .

We must have a clear idea of what “input” means in this context. Providing input should put the system into a well-defined state. This is easy enough to do when we are testing a quadratic equation solver but may be harder to do when we are testing an operating system. Let’s put this concern aside for now, while we consider simple examples.

**Example 2.** The input is supposed to represent a calendar month. Here are some of the possibilities:

- ◇ The program expects the user to enter an integer in the range  $1, 1, \dots, 12$ . The equivalence classes are:

$$\begin{aligned} C_0 &= \text{inputs corresponding to integers in } 1, 2, \dots, 12 \\ C_1 &= \text{inputs corresponding to integers less than } 1 \\ C_2 &= \text{inputs corresponding to integers greater than } 12 \\ C_3 &= \text{inputs that are not valid integers} \end{aligned}$$

The valid inputs are those in class  $C_0$ ; it is highly likely that the program treats those differently from other inputs. The other classes are less certain. However, it is likely that

$C_3$  is distinguished. It doesn't really matter whether (or how) the program distinguishes  $C_1$  and  $C_2$  because we should test for both anyway.

Depending on how its input routines are written, the program might or might not accept the inputs 00005 or +7 as valid. By specifying the input as an *integer*, rather than as a string of characters (which is what it actually is), we are evading this issue.

- ◇ Another possibility is that the program reads the month as a string of digits. Suitable equivalence classes might be:

$$\begin{aligned} C_0 &= \text{strings matching } [1-9] \mid 1[0-2] \\ C_1 &= \text{strings matching } 0[1-9] \mid 1[0-2] \\ C_2 &= \text{the empty string} \\ C_3 &= \text{strings consisting entirely of digits} \\ C_4 &= \text{strings containing non-digit characters} \end{aligned}$$

This case makes it clear that the equivalence classes are not mutually exclusive. Although *each* class partitions the set of input data, *different* classes may contain the same input data. For example, the input string "10" belongs to  $C_0$ ,  $C_1$ , and  $C_3$ .

- ◇ Yet another possibility is that the user is required to enter the month as a string of three letters. Then the equivalence classes would be:

$$\begin{aligned} C_0 &= \text{strings in the set } \{ \text{"jan"}, \text{"feb"}, \dots, \text{"dec"} \} \\ C_1 &= \text{strings in the same set as } C_0 \text{ ignoring case} \\ C_2 &= \text{strings in the set } \{ \text{"january"}, \text{"february"}, \dots, \} \\ C_3 &= \text{strings in the same set as } C_2 \text{ ignoring case} \\ C_4 &= \text{the empty string} \\ C_5 &= \text{other strings} \end{aligned}$$

□

Our "beliefs" about the system probably won't match the facts exactly, but this shouldn't matter all that much.

- ◇ We might choose more equivalence classes than actually exist. In the last example above, the program might test for strings in  $C_0$  and consider all others to be incorrect. The consequence is that we may do a few more tests than necessary; the cost will be greater, but better safe than sorry.
- ◇ Alternatively, we might underestimate the number of equivalence classes needed. A typical example would be testing the accuracy of a standard function. Testing a function such as  $\sin x$ , for example, we would think of divisions such as  $x = 0$ ,  $0 < x < \pi$ ,  $x = \pi$ , and so on. Efficient algorithms, however, might distinguish very small values of  $x$  (for which the approximation  $\sin x \approx x$  is good enough), a range in which a simple polynomial (such as  $x - x^3/3!$ ) is good enough, and a general case where iteration is required. If we are doing black-box testing, we cannot hope to guess the correct classes but it is more likely for this kind of application we would be able to choose classes on the basis of the actual code.

Introducing equivalence classes reduces the number of tests required from the cardinality of the set of possible inputs to a much smaller, and usually manageable, number.

If our belief about the structure of the program is correct, we need to do only one test for each equivalence class. In practice, it is advisable to perform several tests for each class. Experience shows that defects are more likely to occur at the boundaries of the classes than in the middle.

**Example 3.** If the program reads months as integers, the acceptable values are 1, 2, . . . , 12. Likely errors include:

- ◇ accepting 0;
- ◇ not accepting 1;
- ◇ not accepting 12; and
- ◇ accepting 13.

□

Here is a simple procedure for designing tests, based on the foregoing ideas:

- ◇ Divide the inputs into equivalence classes.
- ◇ Determine the boundaries of the equivalence classes.
- ◇ Choose some *typical values* (away from the boundaries) for testing.
- ◇ Choose some *boundary values* for testing.
- ◇ For good measure, choose a few *random values* for testing.

Only the simplest programs have a single input: usually, there will be several, or even many, different inputs. Inputs may be related in different ways:

- ◇ Inputs may be *uncoupled* or *independent*.

When testing an email program, the GET and SEND functions can probably be handled separately.

- ◇ Inputs may be *loosely coupled*.

Someone using a word processor might select a font, a style (bold, italic, etc.), and a font size. In principle, these choices should be independent. In practice, they might be coupled in the sense that some fonts may not have a complete set of styles and size. For fonts such as Wingdings, the concept of style may not be meaningful.

- ◇ Inputs may be *tightly coupled*.

For example, if the first input asks the user to choose miles or kilometres and the second input asks for the distance, validation of the distance depends on the user's choice of units. Similarly, if the user is asked to enter a date in the form day/month/year, the permissible values of 'day' depend on the value entered for 'month'.

The following formalization may make the concept of input coupling clearer. Suppose that a program has two inputs that we represent as a pair  $(i, j)$ . We define two sets:

$$\begin{aligned} \mathcal{S} &= \{ (i, j) \mid \text{the program succeeds with inputs } i \text{ and } j \} \\ \mathcal{F} &= \{ (i, j) \mid \text{the program fails with inputs } i \text{ and } j \} \end{aligned}$$

The inputs are uncoupled iff

$$(i, j) \in \mathcal{F} \Rightarrow \forall k. (i, k) \in \mathcal{F} \wedge \forall k. (k, j) \in \mathcal{F}$$

This definition suggests that we can test uncoupled inputs independently because, if the program fails for some value of the first input, it will fail for that value whatever the value of the second input. And similarly for the second input.

Conversely, the inputs are coupled iff

$$(i, j) \in \mathcal{F} \Rightarrow \exists k. (i, k) \in \mathcal{S} \vee \exists k. (k, j) \in \mathcal{S}$$

This definition suggests that we must be cautious with testing because, if the program fails for a pair of inputs  $(i, j)$ , it may nevertheless succeed for some other value of  $i$  *or* some other value of  $j$ .

There will be equivalence classes associated with each input. For thorough testing in the worst case, we must design tests that cover all of the possibilities. If there are  $n$  uncoupled inputs, and input  $i$  requires  $t_i$  tests, we will need  $t_1 \times t_2 \times \dots \times t_n$  tests. Coupling may reduce the number of tests required.

**Example 4.** Consider the quadratic equation  $ax^2 + bx + c - 0$  again. We can immediately identify these three classes:

- ◇  $b^2 < 4ac$ : no real roots.
- ◇  $b^2 = 4ac$ : one real root.
- ◇  $b^2 > 4ac$ : two real roots.

These classes are based on the familiar formula for solving quadratics:

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Since the formula clearly doesn't work when  $a = 0$ , we should include tests for this case. The equation becomes  $bx + c = 0$  with solution  $x = -c/b$  and there are further problems if  $b = 0$ . We can add these classes:

- ◇  $a = 0, b \neq 0$ : one root.
- ◇  $a = 0, b = 0, c \neq 0$ : no roots.
- ◇  $a = 0, b = 0, c = 0$ : all values of  $x$  satisfy the equation!

□

**Example 5.** For another kind of application, consider testing a program with a **File** menu. Assume the submenu choices are **Open**, **Save**, **Save as**, and **Exit**.

- ◇ For **Open**, there are three classes: the path name is good and the file exists; the path name is good, but the file does not exist; and the path name is invalid.
- ◇ For **Save**, there is only one class: see if it works.
- ◇ For **Save as**, there are two classes: the new path name is good; and the new path name is invalid.
- ◇ For **Exit**, there is again only one class: does it work?

This gives seven equivalence classes. However, there is another pair of equivalence classes that are orthogonal to these:

- ◇ The program already has one or more files open.
- ◇ The program does not have any files open.

The behaviour of `Exit`, for example, clearly depends on whether there are open files. Thus the original number of equivalence classes must be double, giving at least 14 classes. □

**Example 6.** The specification says:

The landing gear must be deployed whenever the plane is within 2 minutes from landing or taking-off, or within 2000 feet of the ground. If visibility is less than 1000 feet, then the landing gear must be deployed whenever the plane is within 3 minutes from landing or lower than 2500 feet.

This gives the following equivalence classes:

Specification	Possibilities	Classes
Time since take-off	within 2 minutes after take-off; 2–3 minutes after take-off; more than 3 minutes after take-off	3
Time to landing	within 2 minutes of landing; 2–3 minutes before landing; more than 3 minutes before landing	3
Relative altitude	less than 2000 feet; 2000–2500 feet; more than 2500 feet	3
Visibility	less than 1000 feet; more than 1000 feet	2
Landing gear deployed	true; false	2

The number of equivalence classes is  $3 \times 3 \times 3 \times 2 \times 2 = 108$ . This is quite a small number, and the code is safety-critical, so several tests within each class are advisable.

Testing the code while the aircraft is flying is risky because the cost of failure might be high. The tests should be conducted in a simulated environment. □

### 2.3 Clear Box Testing

We can test more efficiently with knowledge of the source code. There is a trade-off: if we use more information, we can test more thoroughly, but the number of test cases may become infeasibly large. Strategies include:

- ◇ **All paths:** every distinct path through the component is executed at least once (usually infeasible).
- ◇ **All definition-use paths:** every path from each definition to the use of the defined name is executed at least once (may be feasible for some components).
- ◇ **All uses:** at least one path from each definition to each use of the defined name is executed (usually feasible).
- ◇ **All predicates:** every path from each definition to each predicate (condition) that uses the variable is executed.

- ◇ **All computations:** every path from each definition to each computational use of the variable is executed.
- ◇ **All branches:** each conditional statement (e.g., `if` and `switch`) is tested for each possible target (usually feasible and definitely advisable).

Clear-box testing is usually performed on system components (e.g., functions and classes) rather than on the entire system. In fact, none of the strategies described above would be feasible for a system of significant size. Consequently, the basis for test design is the detailed design of the system rather than the requirements.

4

## 2.4 Automated Tests

There are many reasons for not having people perform tests: there are too many tests, people apply the tests incorrectly through carelessness, people do not record test results meticulously, and so on. Consequently, most tests should be applied automatically. There are three issues to consider.

- ◇ How are the tests generated?
- ◇ How are the tests applied?
- ◇ How are the results measured?

Tests can be generated manually, semi-automatically, or automatically.

- ◇ A test engineer may write a sequence of tests in a formal notation that can be interpreted by a test engine.
- ◇ A test engineer may write a sequence of instructions that guide a test generating program. For example, the kinds of test and number of each kind might be specified.
- ◇ A test generator may generate tests from specifications or from source code. For example, if the coding standards require pre- and post-conditions for every function, a test generator can generate input data that satisfies pre-conditions and the test engine can test that the results satisfy post-conditions.

The tests are applied by a *test engine* or *test driver*. For function testing, the test engine is linked to the object code and generates calls to functions.

Test drivers can also test GUIs by simulating keyboard and mouse events and analyzing the changes on the screen. Two techniques are available:

- ◇ In the “show me” method, a person performs all the tests once, using the keyboard and mouse. The test system monitors the user’s actions and repeats them during the tests.
- ◇ With the automatic approach, the test engineer specifies the tests (“move mouse to (10,60), click left,...”) and the test driver performs the operations.

Tools can also perform tests of a different kind on the source code. In fact, this is usually much simpler to do. Tools exist for:

- ◇ Code analysis. Code can be checked for correct syntax, type matching, initialization, call consistency, and so on. The program `lint` is a well-known analyzer for C code. Recently, compilers have taken over many of the roles of code analyzers.

- ◇ Structure checking. Tools can be used to draw structure graphs of various kinds from source code. The graphs can be checked against the design either manually or automatically. CASE tools such as Rational Rose typically perform structural checking during development.
- ◇ Data analyzer. Tools can be used to check that the way in which data is represented in the program is consistent with requirements.

## 2.5 Regression Testing

We have not yet addressed the issue of what to do when a defect has been corrected. Do we assume that the previous test results are still valid? The safe answer is “no”: once we have changed the software we should run all the tests again.

**Regression tests** are tests that are repeated because a change renders their results invalid. The change is not necessarily a corrected defect: whenever software is modified for any reason, it is best to run regression tests.

Repeating a complete test set after one correction may be excessive. A typical test strategy would divide tests into batches. At the end of a batch, the defects are reported and the corrections are carried out. Then all tests up to and including that batch are repeated.

## 2.6 Cleanroom testing

“Clear box” is generally understood to mean “making use of the code” but there are other kinds of testing that make use of knowledge about the system and are therefore not strictly black-box.

**Cleanroom Software Engineering** is a software development technique that was developed at IBM (Mills, Dyer, and Linger 1987). It is easier to fabricate chips in “clean rooms” that are dust-free than it is to remove the dust from chips after fabrication. Harlan Mills reasoned analogously that it would be easier to develop defect-free software than to remove defects after coding. His goal was to introduce mathematical techniques into software engineering:

- ◇ Use mathematical techniques to verify the correctness of programs early in the development cycle
- ◇ Use statistical techniques for testing late in the development cycle

Strict cleanroom discipline bypasses unit-testing and goes directly to integration testing. (Some people do not feel that this is a good idea (Beizer 1997).) The tests are designed to find the places where the software is most likely to fail, and this is assumed to be on the paths that are executed most frequently (Poore and Trammell 1998).

If you want to obtain information about the population of a country, you don’t ask every member of the population. Instead, you sample the population, do measurements on the sample, and use statistics to infer properties of the population. Analogously, statistical testing envisages all possible uses of a software system, selectively samples them, and defines test cases based on the sample.

A **usage model** describes all possible scenarios of software use at a particular level of abstraction. Usage models can be constructed before code is written and refined after coding is finished and the system is put into use. If previous versions of the system exist, usage models can be based on them.

Usage models are built by identifying states of the system and transitions between them; thus a usage model is a finite state machine. We can think of the usage model as a directed graph  $G = (V, E)$  with labels: vertex labels identify states and edge labels give transition probabilities (and possibly the reason for the transition). As far as possible, the graph is built from subgraphs with a single entry and a single exit; this convention enables the system to be modelled at different levels of abstraction, because a subgraph can be expanded for a detailed view or treated as a single vertex for a high-level view.

The size of models is classified as follows:

- ◇ even “small models” (20 vertexes and 100 edges) can be useful;
- ◇ “typical” models have around 500 vertexes and 2,000 edges;
- ◇ “large” models with more than 2,000 vertexes and 20,000 edges are manageable.

Transition probabilities are estimated from historical or projected use of the application. If the projections are wrong, they can be corrected when the software is up and running. It may be possible to express usage in terms of constraints and to derive the transition probabilities from the constraints. Constraints include:

- ◇ Structural constraints are determined by the model itself
- ◇ Usage constraints express information about the expected ways in which the system will be used
- ◇ Management constraints express information about how the system ought to be used

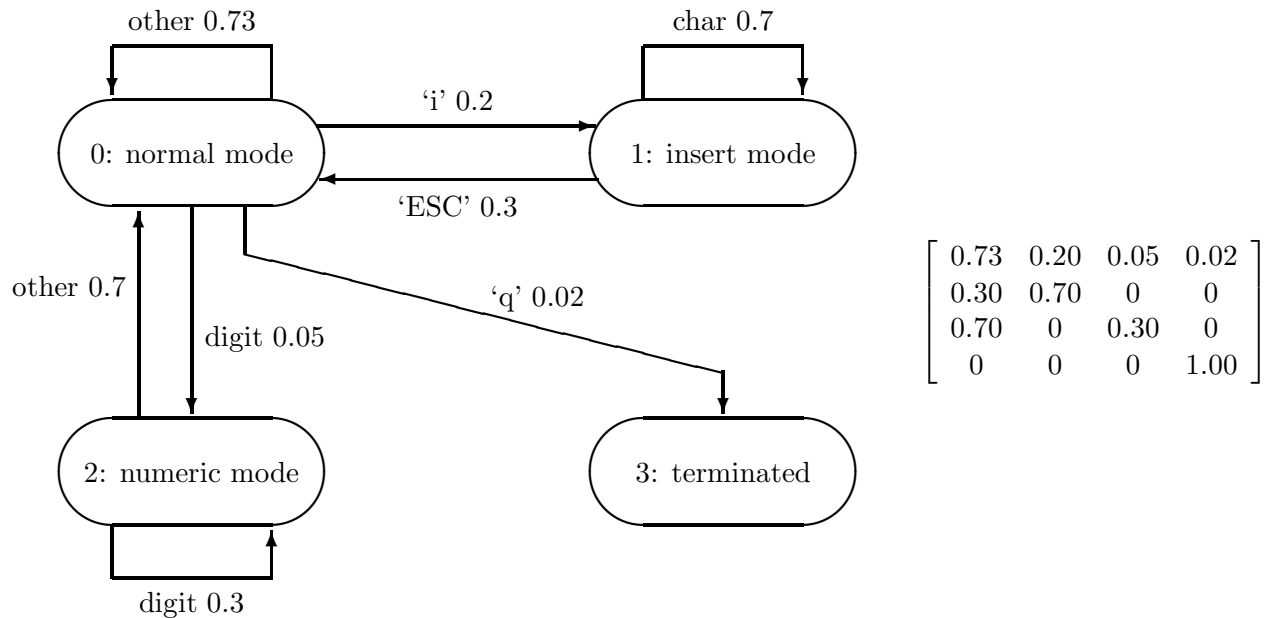


Figure 2: State transition diagram and the corresponding matrix

The sum of the probabilities of the transitions leaving a particular state is 1. The transition probabilities form a matrix  $M$  in which  $M_{ij}$  is the probability of a transition from state  $i$  to state  $j$ . Figure 2 shows a transition diagram for part of an editing program, and the corresponding matrix of transition probabilities. The matrix is used in the following way: if  $\mathbf{p}$  is a vector giving state probabilities (that is,  $p_i$  is the probability that the system is in state

$i$  and  $\sum p_i = 1$ ) then

$$\mathbf{q} = \mathbf{p}M^n$$

is the vector of probabilities after  $n$  state transitions. The statistical model is called a **Markov chain** and it is based on the assumption that the system is “memoryless” — that is, the transition probabilities depend only on the current state and not on previous states.

As  $n$  gets larger, the rows of the matrix get closer together. In the limit,  $M^n$  as  $n \rightarrow \infty$ , they are identical, and  $M_{ij}^\infty$  is the probability that, at any given time, the system is in state  $i$ . (The example above does not illustrate this well because state 3 “terminated” is a sink. After many transitions, the system will reach state 3 and stay there, giving the limiting probability vector  $(0, 0, 0, 1)$ .)

Markov Process theory shows that the transition matrix can be used to derive a number of properties of the system, including:

- ◇ The fraction of execution time that the system will spend in a given state
- ◇ The probability of occurrence of each state in a run of given duration
- ◇ The expected number of occurrences of a state during a given time period
- ◇ The expected number of runs before a given state will occur for the first time
- ◇ The expected number of state transitions in a given time period

Analyzing model characteristics sometimes leads to surprises in a complex system: events that were expected to be rare turn out to be quite frequent; events that were expected turn out to be quite likely. Of course, any such predictions should be validated once the software is running.

The Cleanroom process uses the probabilities as a basis for testing: the number of tests for a particular state is proportional to the probability that the system is in that state. This ensures that the most active parts of the code get the most testing. Experience shows that this method is effective at finding the defects that actually lead to failures.

Both random and non-random tests are used. Non-random tests include:

- ◇ Model coverage tests. Graph theory is used to determine a set of tests that covers all edges; if this test set is feasible, it is used.
- ◇ Mandatory tests. Any tests that are specifically required for safety, security, etc., are applied.
- ◇ Critical states. States that are critical for safety or security reasons, etc., are tested even if they are unlikely to occur in actual use.
- ◇ Regression tests. Tests are repeated after a failure has occurred and the corresponding defect repaired.

After the non-random tests are performed, random testing begins.

- ◇ Estimate the number of random test cases that can be run within the budget.
- ◇ Generate these cases using statistical methods and the transition matrix.
- ◇ Define the best-case scenario: what can be inferred about the quality of the software if no failures occur during random testing?
- ◇ Define the worst-case scenario(s): what will be the consequence of various failure scenarios?

- ◇ Analyze the coverage of the generated tests.
- ◇ Check that testing will be worth while: the best case scenario should predict that the software will be of acceptable quality; the worst case scenarios should ensure that the project is still viable.

Test cases can be generated automatically from the probability data. Figure 3 shows the input to a test generator. It is a grammar in which production rules correspond to state transitions; the grammar rules are annotated with the transition probabilities. One alternative for each rule does not have a probability; the generator can find this value because it knows that the probabilities must sum to 1.

```

NormalMode  : "i"   InsertMode [0.2]
              | "q"   Terminated [0.2]
              | digit NumericMode [0.05]
              | other NormalMode
InsertMode   : "ESC" NormalMode
              | other InsertMode [0.7]
NumericMode  : digit NumericMode [0.3]
              | other NormalMode
Terminated   :

```

Figure 3: Input for transition-based test generator

Some people do not agree that statistical testing is effective (Beizer 1997). “You cannot find a bug unless you execute the code that has the bug” — but the cleanroom approach is that, if the code is never executed, the bug doesn’t matter.

There are also situations in which the rarely-executed code is the most important. In avionics, for example, most of the flight time is spent cruising at more or less constant speed and altitude. Only a small proportion of the time is spent in landing and taking-off. Yet failures during landing and take-off are most likely to be critical failures. Presumably, cleanroom practitioners recognize that there are situations in which “unlikely” does not mean “doesn’t matter”.

## 2.7 Testing in XP

*Extreme Programming* (XP) is a relatively new software process introduced by Kent Beck (2000). Key features of XP include:

- ◇ Early, concrete, and continuing feedback from short cycles.
- ◇ Incremental planning.
- ◇ Flexible schedule, adapting to business needs.
- ◇ Reliance on automated tests written by programmers that catch defects early.
- ◇ Communication is by voice, tests, and source code.
- ◇ Evolutionary design.

An important feature of XP is *pair programming*.

All production code is written with two people looking at one machine, with one keyboard and one mouse (Beck 2000, page 58).

Each member of the pair has a role (they may also exchange roles if they want to). One member uses the keyboard and the mouse and is thinking about how to implement the next part of the code under development. The other member sits back, watches the screen, and thinks strategically:

- ◇ Is this approach going to work?
- ◇ What are some other test cases that might not pass yet?
- ◇ Is there a simpler way to do this?

We will look at XP in depth later in the course; for now, we focus on the testing strategy. There are two main points:

- ◇ Tests are written before the code that they test.
- ◇ Tests are run automatically.

Suppose that the programming pair is ready to write a member function for a class. They proceed as follows:

- ◇ Think of ways in which the function can be tested. Trivial tests, which are certain to pass, are ignored. The good tests are the ones that have a good chance of failing if the code is not very robust.
- ◇ Code the tests. The tests must run automatically: that is, no data entry from the user is allowed, and the only user action required is one keystroke or mouse-click.
- ◇ Run the tests. They will fail, because the function hasn't been written yet, but we now know that the test-set compiles.
- ◇ Code the function.
- ◇ Run tests and revise the code until all tests pass (“the green light comes on”).

The XP strategy is unusual, but it pays off. Beck claims:

- ◇ Programmers become more productive.
- ◇ Writing the tests first helps them to focus on requirements rather than implementation.
- ◇ Because testing is easy, programmers test frequently. As a result, most defects are found in a few minutes rather than in long “debugging” sessions.
- ◇ Code quality improves, because all code has been thoroughly tested.
- ◇ Defects are found at the earliest possible time.

Clearly, the XP approach requires discipline. Many people say that programmers should not test code at all — testing should be left to testing teams who are unfamiliar with implementation details. The theory of XP is that if the tests are written before the code, these objections will lose some of their force.

## 2.8 Kinds of Testing

The precise way in which we conduct tests depends on the purpose of the tests. There are five main phases of testing. For each, we can give:

- ◇ a *name*
- ◇ *criteria* telling us whether the tests pass or fail
- ◇ a *goal* specifying the quality factor achieved by the tests

Name	Criteria	Goal
Function Testing	Components satisfy design specifications	All components achieve individual functionality requirements
Integration Testing	The integrated components satisfy functional requirements	The components perform correctly when they are integrated into the system
Performance Testing	The integrated components satisfy non-functional system requirements	The system performs as well as the designers expected
Acceptance Testing	The system meets the client's expectations	The client confirms the quality of the system
Installation Testing	The system functions correctly in its target environment	The system meets quality requirements on intended platforms

Figure 4: Kinds of testing

**Example 7.** Assume that we have developed software for controlling a radiation therapy machine in hospitals.

- ◇ **Function** testing ensures that all components work correctly according to their design specifications. Testing requires scaffolding because the components are running in isolation rather than as part of the system.
- ◇ **Integration** testing that all components work correctly when they have been linked together into a single system. Scaffolding is required only to simulate system inputs and outputs.
- ◇ **Performance** testing ensures that the system meets requirements of safety, security, accuracy, speed, storage, robustness, reliability, and any other criteria that were established during the requirements phase.
- ◇ **Acceptance** testing is carried out with clients. The system may be connected to actual therapy machines or simulators. The tests are performed, or at least monitored, by people who will actually use the system.
- ◇ **Installation** testing ensures that the software works in a hospital when connected to an actual therapy machine. Radiation doses are measured to ensure that they are within required tolerances. Several tests should be carried out before people are subjected to therapy.

□

5

## 2.9 Modelling

**Mathematical Note.** We will use some simple mathematics in this course. In most situations, simple math is appropriate for software engineering. (Formal specification is an exception, because it is based on high order logic.) Most mathematical models are *parametric*, which means simply that they consist of formulas with various constants and that the values

of the constants must be estimated, for example by statistical reasoning. Complex models with many parameters run the “GIGO” risk — garbage in, garbage out. A simple model with one or two parameters may give useful insights with only a small amount of calculation.

The following tips are useful for doing mathematics:

- ◇ Check that formulas are dimensionally consistent: don’t add hours to months or multiply people by computers.
- ◇ Check that formulas make sense: do they have an appropriate value when  $t = 0$  or when  $t \rightarrow \infty$ ? Is the amount of effort needed positive rather than negative?
- ◇ Choose parameters carefully. In particular, favour pure (dimensionless) values to values with dimensions.

□

Thorough testing is one method that we can use to achieve high quality. But we have not yet addressed important questions:

- ◇ How do we know when the quality is good enough?
- ◇ How do we know if the tests are accomplishing their goals?
- ◇ How do we know when to stop testing?

All of these questions can be answered, at least to some extent, by monitoring the progress of testing. It helps to have some idea of what to expect, so that we can compare our measurements against our expectations. Thus we need a *model* of the testing process.

There are a number of models and they vary in complexity. There is no point in using a complex model if the predictions of a simple model are sufficient for our needs. Complex models tend to require large amounts of data (because they have more parameters to estimate) and, if the data is not available, the greater precision of the model isn’t available either.

The models assume that the software has a number of defects and that this number changes with time. As the number of defects gets smaller, the quality gets higher. At a certain point, the quality is judged to be high enough to release the software to the clients.

The stopping point will depend on the application. A computer game for the retail market might be released with a number of defects, provided that the defects were not of a kind that would lead typical consumers to return the product. Avionics software would not be released for use in flight until the estimated number of defects was very small.

**Notation.** We will use the following notation for all models unless otherwise stated.

- $t$  = the current time: testing starts when  $t = 0$
- $N$  = the number of defects at time  $t$
- $N_0$  = the number of defects at time 0
- $\lambda$  = a rate constant with dimension  $T^{-1}$

Note that we do not necessarily know the value of  $N$  (in fact, estimating  $N$  is the whole point of modelling!). Nevertheless, we assume that  $N$  is a meaningful quantity that we could measure precisely if we had sufficient resources.

**The Linear Model.** The linear model assumes that defects are discovered and corrected at a constant rate. We have

$$\frac{dN}{dt} = -\lambda$$

and, integrating,

$$N = N_0 - \lambda t$$

Testing is complete when  $N_0 - \lambda t = 0$ , or  $t = N_0/\lambda$ . The linear model is too simple to be of much use.

**The Nonhomogeneous Poisson Model.** The NHPP is based on four assumptions:

- ◇ A latent defect will trigger a failure during testing.
- ◇ The number of failures is proportional to the number of defects remaining in the system.
- ◇ All defects have an equal and random probability of being observed.
- ◇ The defects responsible for failures are removed as soon as the failures have been observed.

These assumptions are quite reasonable: the third is probably the least realistic. The second suggests that

$$\frac{dN}{dt} = -\lambda N$$

Integrating gives

$$\ln N = -\lambda t + C$$

We can check the usefulness of this model by plotting  $\ln N$  against  $t$ : the points should fall roughly on a straight line. We can rewrite the solution in the form

$$N = N_0 e^{-\lambda t}$$

The number of defects decays in the same way as radioactive material. Although  $N_0 e^{-\lambda t}$  never becomes zero for real numbers, the number of defects is an integer and we are really talking about  $\lfloor N_0 e^{-\lambda t} \rfloor$ , which may eventually drop to zero.

**Example 8.** A book on software quality quotes a study based on this model (Yeh 1993, page 148). The model predicts that plotting  $\ln(dN/dt)$  against  $t$  should give a straight line. From the plot (and perhaps a linear regression calculation) we should be able to find constants  $a$  and  $b$  that approximately satisfy

$$\ln\left(-\frac{dN}{dt}\right) = a + bt$$

(Note that, since the number of defects is supposed to be decreasing,  $dN/dt$  is negative. Consequently, we have to negate it before taking its logarithm.) The study quoted gave  $a = 1.71$  and  $b = -0.0116$ . In the notation of our model,

$$\begin{aligned} dN/dt &= -\lambda N_0 e^{-\lambda t} \\ \text{so } b &= -\lambda \\ \text{and } a &= \ln(\lambda N_0) \\ \text{giving } \lambda &= 0.0116 \\ \text{and } N_0 &\approx 477 \end{aligned}$$

It is most unfortunate that no units are provided for this example. It is quite likely that  $N_0 \approx 477$  indicates that there were 477 defects before testing but what are we to make of  $\lambda = 0.0116$ ? This is a rate, but is it measured in defects per hour, per day, or per century?  $\square$

**Example 9.** Motorola use this model for *zero-failure testing* (Brettschneider 1989). Let

$$\begin{aligned} N_t &= \text{target projected number of failures} \\ N_s &= \text{number of failures observed so far} \\ t_s &= \text{number of hours of testing up to last failure} \end{aligned}$$

We compute  $t$  using the formula:

$$t = t_s \frac{\ln(N_t/(\frac{1}{2} + N_t))}{\ln((\frac{1}{2} + N_t)/(N_s + N_t))}$$

Then we expect that, after  $t$  hours of testing without any failures, the number of remaining defects is no more than the target number,  $N_t$ .

For example, assume a 33 KLOC program for which 15 failures have been detected in 500 hours of testing. During the last 50 hours of testing, no failures have been reported. The goal is 0.03 defects/KLOC.

The projected number of failures is  $(0.03/1000) \times 33000 = 1$ . Thus

$$\begin{aligned} t &= \frac{\ln(1/1.5) \times (500 - 450)}{\ln(1.5/(15 + 1))} \\ &\approx 77 \end{aligned}$$

and we infer that 77 failure-free hours of testing are needed. Since we have already achieved 50 hours without failure, we should test for a further  $77 - 50 = 27$  hours. Of course, if a failure occurs during this period, we must correct it and repeat the calculation with new data.  $\square$

Unfortunately, Brettschneider (1989) does not show how this formula is derived. The reasoning was presumably along the following lines: assume that the failure detection rate falls as  $e^{-\lambda t}$ . Then, during any time interval  $[t_1, t_2]$ , the number of defects falls from  $N_1$  to  $N_2$  where

$$\begin{aligned} N_1 &= N_0 e^{-\lambda t_1} \\ N_2 &= N_0 e^{-\lambda t_2} \end{aligned}$$

and therefore

$$\begin{aligned} \frac{N_1}{N_2} &= \frac{N_0 e^{-\lambda t_1}}{N_0 e^{-\lambda t_2}} \\ &= e^{\lambda(t_2 - t_1)} \end{aligned}$$

We need to know the time interval  $t$  to go from  $N_t + \frac{1}{2}$  defects to  $N_t$  defects, where  $N_t$  is the target number of defects. This time is given by

$$-\lambda t = \ln\left(\frac{N_t}{N_t + \frac{1}{2}}\right) \quad (1)$$

We already know the time spent on testing so far,  $t_s$ , and the number of defects found,  $N_s$ . Thus

$$-\lambda t_s = \ln \left( \frac{N_t + \frac{1}{2}}{N_t + N_s} \right) \quad (2)$$

Dividing (1) by (2) gives

$$\frac{t}{t_s} = \frac{\ln \left( \frac{N_t}{N_t + \frac{1}{2}} \right)}{\ln \left( \frac{N_t + \frac{1}{2}}{N_t + N_s} \right)}$$

A problem with this formula is that the value  $\frac{1}{2}$  seems arbitrary and yet the result depends strongly on it. The tables in Figure 5 show values of  $t$  when  $t_s = 500, N_s = 20$ , and  $\delta$  is the value used instead of  $\frac{1}{2}$ . In the left table,  $N_t = 1$  and in the right table  $N_t = 5$ , showing that the dependence on  $\delta$  is not limited to very small values of the target number of defects.

$\delta$	$t_s$	$\delta$	$t_s$
0.1	16.1586	0.1	6.22867
0.2	31.8499	0.2	12.4889
0.3	47.1512	0.3	18.7823
0.4	62.1244	0.4	25.1100
0.5	76.8201	0.5	31.4736
0.6	91.2799	0.6	37.8745
0.7	105.539	0.7	44.3139
0.8	119.628	0.8	50.7934
0.9	133.571	0.9	57.3141

Figure 5: Effect of  $\delta$  on predicted testing time

**Proportional Defect Injection** Assume that the defect detection rate is  $\lambda N$ , as above, but that correction injects defects at a rate  $\mu N$ . This is consistent with the idea that the maintenance programmer injects, on average,  $\mu/\lambda$  defects for each defect corrected. Then

$$\begin{aligned} \frac{dN}{dt} &= (\mu - \lambda)N \\ N &= N_0 e^{(\mu - \lambda)t} \end{aligned}$$

There are three cases:

- $\mu < \lambda$  : testing removes defects, but at a slower rate
- $\mu = \lambda$  : the number of defects remains constant
- $\mu > \lambda$  : the number of defects increases exponentially

None of these situations is desirable. Large, old systems may exhibit  $\mu > \lambda$  characteristics during maintenance; this is a signal to get rid of the software.

**Constant Defect Injection** Another possibility is that correction injects defects at a constant rate  $\kappa N_0$  that is independent of the number of defects already there.<sup>2</sup> This might be a model of corrective and adaptive maintenance, in which work is being done on the program all the time, with enhancements as well as corrections.

$$\frac{dN}{dt} = \kappa N_0 - \lambda N$$

Before even trying to solve this equation, we can see that, if  $\kappa = \lambda$  then, at time  $t = 0$  when  $N = N_0$ ,

$$\begin{aligned} \frac{dN}{dt} &= \lambda N_0 - \lambda N_0 \\ &= 0 \end{aligned}$$

and so  $N = N_0$  always.

Re-arranging the equation for easier integration yields

$$\frac{dN}{N - N_0(\kappa/\lambda)} = -\lambda dt$$

and actually performing the integration gives

$$\ln(N - N_0(\kappa/\lambda)) = -\lambda t + C$$

Substituting  $t = 0$  and  $N = N_0$  gives

$$C = \ln(N_0 - N_0(\kappa/\lambda))$$

and therefore

$$-\lambda t = \ln\left(\frac{N - N_0(\kappa/\lambda)}{N_0 - N_0(\kappa/\lambda)}\right)$$

or

$$e^{-\lambda t} = \frac{N - N_0(\kappa/\lambda)}{N_0 - N_0(\kappa/\lambda)}$$

which we can re-arrange to obtain

$$N = N_0 \left[ e^{-\lambda t} + \frac{\kappa}{\lambda} (1 - e^{-\lambda t}) \right]$$

As we have already noted, if  $\kappa = \lambda$  then  $N = N_0$  always. In all cases, as  $t \rightarrow \infty$ ,  $N \rightarrow N_0 \left( \frac{\kappa}{\lambda} \right)$ . The number of defects tends to a constant which may be either more or less than  $N_0$ , depending on  $\kappa/\lambda$ .

---

<sup>2</sup>It might seem easier to use a single variable for the rate. The advantage of using  $\kappa N_0$  is that  $\kappa$  has the same dimensions as  $\lambda$  (i.e.,  $T^{-1}$ ) and it is easier to see that the equations are dimensionally correct.

**When do we stop testing?** In an ideal world, we would stop testing when there were no defects left. There are two problems with this naive approach:

- ◇ We can never be certain that there are no defects left.
- ◇ By the time that we are reasonably confident that there are no defects left, there may no longer be a market for our product.

Consequently, we anticipate shipping the product with defects. The question becomes: how many?

The following very simple analysis shows that it is possible to make quantitative estimates of optimal shipping time. In practice, there would be a larger number of factors to consider and the models (and calculations) might be considerably more sophisticated.

We will assume:

- ◇ The NHPP model, so that the number of defects expected at time  $t$  is  $N_0e^{-\lambda t}$ .
- ◇ The cost of fixing a defect **before** delivering the software is  $B$  dollars/defect.
- ◇ The cost of fixing a defect **after** delivering the software is  $A$  dollars/defect.
- ◇ The cost of delaying delivery is  $C$  dollars/day. (This number corresponds to the revenue of the software, which might be sales/day or rental/day, depending on the kind of application.)
- ◇ We stop testing at time  $T$ .

When we deliver the software:

$$\begin{aligned} \text{number of defects fixed} &= N_0(1 - e^{-\lambda T}) \\ \text{number of defects present} &= N_0e^{-\lambda T} \end{aligned}$$

The total cost is the sum of the cost of fixing defects before and after delivery, and the lost revenue due to late delivery:

$$\begin{aligned} C_{\text{tot}} &= AN_0e^{-\lambda T} + BN_0(1 - e^{-\lambda T}) + CT \\ &= BN_0 + (A - B)N_0e^{-\lambda T} + CT \end{aligned}$$

We can note the following properties of this formula:

- ◇ For large negative values of  $T$ ,  $e^{-\lambda T}$  dominates and  $C_{\text{tot}} \approx (A - B)N_0e^{-\lambda T}$ .
- ◇ When  $T = 0$ ,  $e^{-\lambda T} = 1$ , and  $C_{\text{tot}} = AN_0 =$  the cost of fixing all defects after delivery, as we would expect.
- ◇ For large positive values of  $T$ ,  $e^{-\lambda T} \approx 0$ , and  $C_{\text{tot}} \approx BN_0 + CT$ , as we would expect.

We can deduce from this that  $C_{\text{tot}}$  has a minimum value, and hence there is an optimal time for delivery. However, this minimum is only of use to us if it corresponds to a positive value of  $T$ .

Simplifying and differentiating with respect to  $T$  gives

$$\frac{dC_{\text{tot}}}{dt} = -\lambda N_0(A - B)e^{-\lambda T} + C$$

which is zero when

$$e^{-\lambda T} = \frac{C}{\lambda N_0(A - B)} \tag{3}$$

Solving for  $T$  gives

$$T = -\frac{1}{\lambda} \ln \left( \frac{C}{\lambda N_0 (A - B)} \right)$$

$T$  is positive if the logarithm is negative, which gives us the condition for the existence of an optimal delivery time:

$$\frac{C}{\lambda N_0 (A - B)} < 1 \quad (4)$$

Since the number of defects at time  $T$  is  $N_0 e^{-\lambda T}$ , (3) says that, if we deliver at time  $T$ , the delivered software will have

$$N_0 e^{-\lambda T} = \frac{C}{\lambda (A - B)}$$

defects left. We can infer from this:

- ◇ If the expected rate of return,  $C$ , is large, it pays to ship with more defects.
- ◇ If the cost of post-delivery defects,  $A$ , is large, it pays to ship with fewer defects.

**Example 10.** Here is a numeric example. Assume:

$$\begin{aligned} N_0 &= 1000 \text{ initial defects} \\ B &= 100 \text{ \$/day to fix defects before delivery} \\ A &= 1000 \text{ \$/day to fix defects after delivery} \\ C &= 1000 \text{ \$/day lost revenue for late delivery} \\ \lambda &= 0.01 \text{ initial correction rate is } 1000 \times 0.01 = 10 \text{ defects/day} \end{aligned}$$

Then (4) tells us that the optimal testing time is

$$-\frac{1}{\lambda} \ln \left( \frac{C}{\lambda N_0 (A - B)} \right) \approx 219 \text{ days}$$

and (5) tells us that the software will be shipped with

$$\frac{C}{\lambda (A - B)} \approx 111 \text{ defects}$$

If we increase  $C$  to 10000 dollars/day, then

$$\frac{c}{\lambda N_0 (A - B)} \approx 1.11 > 1$$

and there is no optimal delivery time.  $\square$

**Conclusion: Models for Testing**

- ◇ Models are not precise but without modelling you are working in the dark.
- ◇ Simple models have been found to work well enough.
- ◇ Whatever you do, *at least* keep records of defects found and corrected.
- ◇ We need ways of estimating Mean Time To Failure (MTTF) and Mean Time To Repair (MTTR).
- ◇ From the client's point of view, both MTTF and MTTR are important, because both affect down-time.
- ◇ Very high reliability presents problems. Telecom standards are typically 5 minutes/year of down-time. The proportion of down-time is

$$\frac{5}{60 \times 24 \times 365} \approx 10^{-5}$$

For very high reliability applications, such as avionics, the proportion is required to be as low as  $10^{-9}$ . Since this cannot be achieved by simple testing methods, other techniques, such as formal verification, are required.

**2.10 Conclusion**

- ◇ “Testing” means testing *and* correcting.
- ◇ Testing can ensure a desired level of quality — up to a point!
- ◇ Testing cannot make bad software good. If your code is lousy, your product will be lousy, too.
- ◇ Testing cannot correct early errors, made in the requirements, specification, or design stages. The best we can do is install work-arounds.
- ◇ Acceptance testing (by clients) may reveal requirements errors. Then what? Solution: keep clients involved in the process so that this doesn't happen.

## 3 Coding

Much of the material in this section is based on *Large-Scale C++ Software Design* by John Lakos (1996). This is a book intended for practitioners and it is much too long and complex for an undergraduate course. However, if you are interested in becoming a professional software developer, it is probably a useful investment. In SOEN 345, we will only skim the surface of the topics that Lakos discusses in depth.

Why C++? C++ is currently the most popular language for the development of large-scale software systems for which quality (in the sense of functionality, safety, performance, maintainability, etc.) is important. C++ is also a complex language: if you learn to manage C++ projects, Java should not be a problem.

In the introduction, Lakos says (page 1):

This book is about how to design very large, *high-quality* software systems. It is intended for experienced C++ software developers who strive to create highly maintainable, highly testable software architectures. This book is not a theoretical approach to programming; it is a thorough, practical guide to success, drawing from years of experience of expert C++ programmers developing huge, multi-site systems. We will demonstrate how to design systems that involve hundreds of programmers, thousands of classes, and millions of lines of C++ source code.

### 3.1 Common Problems

Code quality suffers if:

- ◇ bad layout makes the code hard to read
- ◇ bad comments — or lack of comments — makes the code hard to understand
- ◇ bad identifiers make the code hard to understand
- ◇ lack of supporting documentation makes it hard to find out what the code is supposed to be doing

We will address these issues later, but they are relatively minor. Here are some of the problems that can make or break large-scale software development.

#### 3.1.1 Cyclic Dependencies

Cyclic dependencies might seem improbable, but they can arise quite easily. Here's a simple example:

```

class Person
{
    ....
    private:
        Car *fleet[12];
    ....
};

class Car
{
    ....
    private:
        Person *owner;
    ....
};

```

Cyclic dependencies make unit testing very difficult or even impossible. The problem is that, in order to test one unit of the cycle, we need the other, and *vice versa*. In practice, cycle typically do not involve just two classes, but many classes.

### 3.1.2 Link-Time Dependencies

All of the functions of a class are linked into the executable whether you use them or not. The only way to avoid this is to use a library, because library functions are linked only if they are actually called. Some programmers write “Winnebago classes” that include numerous functions that might be useful rather than only those functions that are actually needed for the application. (Sometimes, this is done in the name of “reusable code”). If many programmers in a project do this, link times become longer and executables become larger.

Early implementations of C++ produced executables of well over a megabyte for the familiar “Hello, world” program. Modern implementations are somewhat better because code has been moved from resident classes to libraries.

### 3.1.3 Compile-Time Dependencies

Changing a header file that is read by many system components will require re-compilation of all of those components. Often, however, the change that you have made does not affect most of the components, and the extra compile time is wasted time.

A simple example is a header file that declares error codes used by the system. Such a header file is likely to be `#included` by most system components. It has certain advantages: for example, it might simplify the handling of error reports in different languages. As the project grows, however, adding an error code has an ever-increasing penalty in the form of increased compilation time.

In many cases, header files are `#included` unnecessarily. Programmers aren’t sure whether the `#includes` are needed or not, and put them in “just in case”.

Lakos cites an early C++ project at his company, Mentor Graphics, where compile-time dependencies were not seen as a concern early in the development of a CAD system. At a later stage of development, a network of workstations required more than a week to compile the system!

### 3.1.4 The Global Name Space

The global name space of a C++ program tends to grow rapidly with increasing size. In a well-disciplined (and recent) project, C++ namespaces can be used to reduce the problem, but it can still occur.

Here is the problem: developers, perhaps numbering in the hundreds, introduce names into their code. Due to poor discipline, these names get into the global namespace. When integration testing starts, numerous name collisions occur and large amounts of code have to be changed to avoid them.

Here is another example cited by Lakos. He needed to find the definition of the class (?) `TargetId` used in the statement

```
TargetId id;
```

He ran “`grep TargetId`” on all of the header files of the project. But, since there were several thousand, `grep` complained of “too many files”. Lakos then wrote a shell script that split the header files into 26 groups based on the first letter of their name and called `grep` for each group. This technique eventually revealed the definition of `TargetId`.

### 3.1.5 Logical and Physical Design

Most discussions of design refer to logical design. *Logical design* principles tell us how to divide the system up into classes, how to choose between aggregation or inheritance, whether a class should have a copy constructor, when to use a free function rather than a member function, and so on.

Logical design issues are very important; but, for a large system, they are not the only design issues. *Physical design* principles tell us how the system should be organized into directories, libraries, and files and how to reduce compile-time and link-time dependencies.

Consider the choice of whether to declare a function `inline`. Logically, inlining a function makes no difference to the system, because it has no effect on functional behaviour. Inlining does affect non-functional characteristics: it tends to make the program run faster at the expense of using more memory. It is easy to overlook another important consequence of inlining: if the implementation of the function is ever changed, all units that use it will have to be recompiled. Since it is often the small, low-level utility functions that are inlined, this means that a small change could result in recompiling the entire system.

### 3.1.6 Reuse

Writing software that is reusable is a commendable goal that has attracted considerable attention during the last few years. It is important to realize that coding for reuse does have a cost.

A unit intended for a particular application can be designed to be optimal (in time, or space, or both) for that application. It provides only the functionality needed for the application and does so in a simple and efficient way. The same unit, intended for reuse in several systems or even many systems, might have to be more general and therefore larger and less efficient.

## 3.2 Basic Concepts

### 3.2.1 Declarations and Definitions

C++ makes a well-defined (though not widely understood) distinction between declarations and definitions. Figure 6 shows some declarations and definitions.

- ◇ A *declaration* introduces a name into a scope. It is not an error if the same declaration occurs more than once in a scope.
- ◇ A *definition* provides a unique description of a type, variable, function, or class within a program. Definitions must not be repeated within a scope.

Declarations: <pre> int fun(double); class Widget; typedef unsigned int Flags; friend Fido; extern bool Disaster; </pre>	Definitions: <pre> int n; char *p; extern int glob = 5; static bool Running; static int f() { ... } inline int g() { ... } enum DIR { UP, DOWN }; const double PI = 3.14159; struct Point { ... }; class Vector { ... }; </pre>
--	---

Figure 6: Declarations and Definitions

### 3.2.2 Linkage

C++ also makes a distinction between internal and external linkage. A *translation unit* is the code that the C++ compiler sees when it compiles a `.cpp` file: it includes everything read from the header files as well as the code in the file itself. Some names in the translation unit get in to the object file created by the compiler and others do not. Only the names that get into the object file can affect linking.

- ◇ A name has *internal linkage* if it is local to the translation unit.
- ◇ A name has *external linkage* if it appears in the object file created from a translation unit.

The important point is that names with external linkage can cause name collisions when the program is linked, but names with internal linkage cannot. Figure 7 shows examples of internal and external linkage.

Internal linkage: <pre> static int k; extern int m; double f(int); enum Switch { ON, OFF }; class Point {     public:         Point(); }; class Vector; inline bool operator==(...) {...} </pre>	External linkage: <pre> int k; class Shape {     static int numShapes; }; Point &amp; Point::operator+=(...) {...} Point operator+(...) {...} </pre>
--	--

Figure 7: Internal and external linkage

### 3.2.3 Header Files

Header files are also called “.h files”. The definitions above have the following consequences for header files:

- ◇ It is usually an *error* to put a definition with external linkage into a header file. (The compiler will complain that the symbol is defined more than once.)
- ◇ It is usually a *bad idea* to put a definition with internal linkage into a header file. These definitions pollute the global name space and occupy space in every unit that `#includes` the header file.

In summary: *don't put definitions in header files* even if the compiler doesn't tell you not to. Header files are for *declarations*. Figure 8 summarizes what should and should not be in a header file.

Code in header file	Remarks
<code>int k;</code>	Illegal
<code>extern int BUFSIZE = 120;</code>	Illegal
<code>const int NUMPOINTS = 20;</code>	Bad practice
<code>static double x;</code>	Bad practice
<code>static void print() {...}</code>	Bad practice
<code>class Radio {</code>	OK
<code>static int numRadios;</code>	OK
<code>static const double TWOPI;</code>	OK
<code>char *message;</code>	OK
<code>void tune(double);</code>	OK
<code>};</code>	
<code>inline void tune(double freq) {...}</code>	OK
<code>int Radio::numRadios;</code>	Illegal
<code>double Radio::PI = 3.14159;</code>	Illegal
<code>void Radio::setVol() {...}</code>	Illegal

Figure 8: Good and bad things in a header file

### 3.2.4 Implementation Files

Implementation file are also called “.cpp files” or simply “.c files”. Some entities in an implementation file must have external linkage (because otherwise the file would be useless) but external linkage should be used only when necessary.

Suppose that `counter` and `max` are used by functions with the implementation file but are not needed outside it. Don't write them like this:

```
int counter;
double max(double x, double y) { return x > y ? x : y; }
double min(double x, double y) { return x > y ? y : x; }
```

Instead, write this:

```
static int counter;
static double max(double x, double y) { return x > y ? x : y; }
inline min(double x, double y) { return x > y ? y : x; }
```

### 3.2.5 typedef

A `typedef` declaration creates an alias for an existing type. It does not create a new type and may give the illusion of type safety.

```
typedef double Inches;
typedef double Pounds;

class Person
{
public:
    Inches getHeight();
    void setWeight(Pounds w);
    ....
};

Person p;
p.setWeight(p.getHeight());
```

Some typedefs from `windef.h`:

```
typedef unsigned long    DWORD;
typedef int              BOOL;
typedef unsigned char    BYTE;
typedef unsigned short   WORD;
typedef float            FLOAT;
typedef FLOAT            *PFLOAT;
typedef BOOL near        *PBOOL;
typedef BOOL far         *LPBOOL;
typedef BYTE near        *PBYTE;
typedef BYTE far         *LPBYTE;
typedef int near         *PINT;
typedef int far          *LPINT;
typedef WORD near        *PWORD;
typedef WORD far         *LPWORD;
typedef long far         *LPLONG;
typedef DWORD near       *PDWORD;
typedef DWORD far        *LPDWORD;
typedef void far         *LPVOID;
typedef CONST void far   *LPCVOID;
typedef int              INT;
typedef unsigned int     UINT;
typedef unsigned int     *PUINT;
```

Some legitimate uses of `typedef`:

- ◇ Preserving size across different platforms.

```
#ifdef __ALPHA__
typedef int Integer;
```

```

#endif
#ifdef __WINDOWS__
typedef long int Integer;
#endif
....

```

- ◇ Defining complex types that would clutter the code if used in full every time:

```
typedef int & (Person::*PCPMFDI)(double, double) const;
```

declares PCPMFDI to be a pointer to a `const` member function of class `person` taking two double arguments and returning a reference to an `int`.

### 3.2.6 assert

Use `assert` statements! They provide an easy way of building tests into the code. The overhead during development is small and can be removed altogether for production code (by `#defining NODEBUG` during compilation).

Use assertions and exceptions appropriately.

- ◇ An assertion failure should mean that the code is logically flawed. Clients should never see an assertion failure — if they do, they should submit an error report. Every assertion failure indicates a defect that must be corrected. The purpose of `assert` statements is to help find the defects.
- ◇ A problem that could occur during normal execution may be handled by an exception. The exception mechanism allows a function to “give up” and pass control back to a higher level that can handle the exception in an appropriate way.
- ◇ If the problem can be handled locally, then neither an assertion or an exception is needed.

**Warning:** the code in an `assert` statement must *not* be required for execution. Don’t do this:

```

String::String() : size(DEFAULT_SIZE), length(0)
{
    assert(data = new char[size]);
}

```

Instead:

```

String::String() : size(DEFAULT_SIZE), length(0)
{
    data = new char[size];
    assert(data);
}

```

### 3.2.7 Logical Relations

There are three logical relations between program components that we will refer to frequently: *Isa*, *Uses In The Interface*, and *Uses In The Implementation*. These are all illustrated by the following code:

```

class Vehicle { .... };

class Truck : public Vehicle
{
    public:
        Person getOwner();
    private:
        Wheel wheels[4];
};

```

- ◇ Truck isa Vehicle
- ◇ Truck uses-in-the-interface Person
- ◇ Truck uses-in-the-implementation Wheel

It is better to use “isa” rather than “inherits” or “derives” because there is no doubt about the direction of the dependency (a `Truck` depends on a `Vehicle`).

It is sometimes even better to say “is a kind of” to avoid confusion between “isa” and “is an instance of”.

Uses-in-the-interface is clearly a closer relationship than uses-in-the-implementation because a client has to know about the former but not the latter.

Uses-in-the-implementation is sometimes divided into subcategories:

C Uses T	the class C has a member function that names T
C HasA T	the class C contains an instance of T
C References T	the class C contains a pointer to T
C WasA T	the class C privately inherits from T

The relation *uses* includes both uses-in-the-interface and uses-in-the-implementation.

### 3.2.8 Inheritance versus Layering

Large object oriented programs typically contain two hierarchies of classes.

- ◇ The *inheritance hierarchy* is defined by the *inheritance relation* (base classes and derived classes in C++ terminology).
- ◇ The *layering hierarchy* is defined by the *uses relation*.

Overuse of inheritance is a common mistake in object oriented programming. Use inheritance only for strictly “is a kind of” relations: see Figure 9. Use layering for all other relations: see Figure 10.

## 3.3 General Coding Rules

**Rule 1** *A header file must have include guards.*

In other words, the file `header.h` should have the general form:

```

class FormField
  class Name
  class Address
  class PhoneNumber
  class Eaddress
  class MemberNumber

class Container
  class RandomAccess
  class Array
  class BinaryTree
  class LimitedAccess
  class Stack
  class Queue
  class Deque

```

Figure 9: Inheritance Hierarchies

```

class Network
  class WorkStation
    class Processor
    class Memory
    class Keyboard
    class Monitor
  class Switch

class Report
  class Cover
  class Preface
  class TableofContents
  class Chapter
    class Section
    class Figure
    class Table
  class Appendix
  class Index

```

Figure 10: Layering Hierarchies

```

#ifndef HEADER_H
#define HEADER_H

// Declarations

#endif

```

The purpose is to prevent a header file being read twice in one compilation unit. The form `HEADER_H` is just a convention; you should choose a convention and stick to it. If the project standards define a convention, use it.

**Rule 2** *All data members must be `private`.*

Sometimes, a class has data members that must be accessed by other objects. Making them `public` avoids the need for `get/set` member functions. There are several reasons for *not* making them `public`:

- ◊ If a data member can be freely changed by external objects, the class loses control of its data. The need for `public` data members may be a sign that there is something wrong with the design.
- ◊ If `get` and `set` member functions are absolutely necessary, they can be defined as inline functions. The cost of calling them should be no more than the cost of accessing the data directly.
- ◊ It is essential to hide information within the class to allow for implementation changes during development and maintenance.

Suppose that class `Rectangle` was declared like this (some parts omitted):

```

class Rectangle
{
public:
    void move(Vector disp)
    {
        bottomLeft += disp;
        topRight += disp;
    }
    Point getTopRight()
    {
        return topRight;
    }
private:
    Point bottomLeft;
    Point topRight;
};

```

We can change the representation without changing the interface. Note the trade-off (trivial in this case): `move` becomes faster and `getTopRight` becomes faster.

```

class Rectangle
{
public:
    void move(Vector disp)
    {
        bottomLeft += disp;
    }
    Point getTopRight()
    {
        return bottomLeft + diagonal;
    }
private:
    Point bottomLeft;
    Vector diagonal;
};

```

If the data members had been `public`, this change would have required all clients to modify their code.

(As we have shown the example, all clients would have been required to recompile. This could have been avoided by putting the member function definitions in a separate file. However, they would then not be inline functions.)

- ◇ A well-designed class should have *invariants* — predicates that are made true by constructors and maintained by mutators. Depending on the application, we could treat an unsatisfied invariant as an exception or as an assertion failure. In the following code, the constructor throws an exception for an illegal rectangle and attempting an illegal move has no effect.

```

class Rectangle
    // Invariant: left.x >= 0 && right.x <= 1600
{

```

```

public:
    Rectangle(Point p, Point q) : bottomLeft(p), topRight(q)
    {
        if (bottomLeft.x < 0 || topRight.x > 1600)
            throw BadRectangle;
    }
    void move(Vector disp)
    {
        if ((topRight + disp).x <= 1600)
        {
            bottomLeft += disp;
            topRight += disp;
        }
    }
    Point getTopRight()
    {
        return topRight;
    }
private:
    Point bottomLeft;
    Point topRight;
};

```

Alternatively, we could use assertions: the program fails when the invariant condition is violated.

```

class Rectangle
    // Invariant: left.x >= 0 && right.x <= 1600
{
public:
    Rectangle(Point p, Point q) : bottomLeft(p), topRight(q)
    {
        assert(bottomLeft.x >= 0);
        assert(topRight.x <= 1600);
    }
    void move(Vector disp)
    {
        assert((topRight + disp).x <= 1600);
        bottomLeft += disp;
        topRight += disp;
    }
    ....
};

```

**Rule 3** *Avoid names with external linkage at file scope.*

This means: don't define a variable or a function in a header file or implementation file unless it is qualified by `static` or enclosed in a scope (e.g., a `struct` or `class` declaration).

The reason is that a name of this kind gets into the global name space. This creates two problems:

- ◇ If different programmers introduce the same name, linking will fail.
- ◇ Anyone who discovers the name can declare it and use it. For variables, this means that anyone can alter your variable!

To restrict a name to one file, you can use `static` definitions, which have internal linkage.

Sometimes, however, you really need a name to be used in several different program components. The solution that Lakos gives is to use `structs`. With modern C++, there is a better solution: namespaces. The basic concepts are described by the following code, which only skims the surface of what namespaces can do.

First, we declare a namespace in an implementation file. The namespace declaration contains declarations of the names we want to use and is followed by definitions of those names. Note that we must qualify each of the names being defined.

```
// a.cpp
#include <cmath>

namespace Globals
{
    extern const double PI;
    double sqr(double x);
    double hyp(double x, double y);
}

const double Globals::PI = 4 * atan(1);

double Globals::sqr(double x)
{
    return x * x;
}

double Globals::hyp (double x, double y)
{
    return sqrt(sqr(x) + sqr(y));
}
```

Next, we write a header file for people who want to use our names. This file contains another declaration of the namespace, but it may only contain a subset of the names that we declared originally. (In many cases, you want to make all of the names accessible. Then you need only one namespace declaration and you put it into the header file.)

```
// namespace.h
#ifndef NAMESPACE_H
#define NAMESPACE_H

namespace Globals
{
    extern const double PI;
    double hyp(double x, double y);
}
```

```
#endif
```

There are several ways in which clients can use your namespace. One way is to qualify each of the names that they use from it.

```
// b.cpp
#include "namespace.h"

bool isRightTriangle(double a, double b, double c)
{
    return c == Globals::hyp(a,b);
}
```

Another is to write `using` declarations for each of the names that they want to use.

```
// c.cpp
#include "namespace.h"

using Globals::hyp;

bool TestRight(double a, double b, double c)
{
    return c == hyp(a,b);
}
```

A third way is to open the entire namespace with a `using namespace` declaration. Note that we do not have to do this at file scope: in the example, we make the names available within the function body only.

```
// d.cpp
#include "namespace.h"

bool AnotherTest(double a, double b, double c)
{
    using namespace Globals;
    return c == hyp(a,b);
}
```

9

**Rule 4** *Avoid enumerations, typedefs, and constant definitions at file scope in header files.*

Header files in most C++ programs consist mainly of class declarations. Any enumerations, typedefs, or constants that the class needs can be declared within the class declaration or in the implementation file.

In the following header file, the enumeration `DIRECTION` is defined within the scope of the class rather than at file scope. The only name at file scope in this header file is `X`.

```
// x.h
#ifndef X_H
#define X_H

class X
{
public:
    X();
    enum DIRECTION { LEFT, RIGHT, UP, DOWN };
    DIRECTION whichWay();
private:
    DIRECTION dir;
};
#endif
```

In the implementation file, we have to qualify the name of the enumeration and its values. Note that the constant `initialDir` is defined in the implementation file, not the header file. (This assumes that clients don't need to use it.)

```
// x.cpp
#include "x.h"

const X::DIRECTION initialDir = X::LEFT;

X::X() : dir(initialDir)
{
}

X::DIRECTION X::whichWay()
{
    return dir;
}
```

**Rule 5** *Avoid using preprocessor macros in header files except for `#include` guards.*

If you `#define` something in a header file, every file that `includes` that file obtains the definition. This can lead to very strange results.

```
// y.h
#ifndef Y_H
#define Y_H

#define BAD 0
#define GOOD 1

#endif

// z.cpp
#include "y.h"
```

```
enum { GOOD, BAD };
```

The harmless looking code in `z.c` produces mysterious compiler diagnostics:

```
error C2059: syntax error : 'constant'
error C2143: syntax error : missing ';' before '}'
error C2143: syntax error : missing ';' before '}'
```

Preprocessor macros make almost anything possible in C++. For example, no matter how carefully you declare class `HighSecurity`, you cannot prevent a malicious client from writing

```
// cheat.cpp
#define private public
#include "HighSecurity.h"
```

**Rule 6** *Document the interface.*

While all documentation is important, it is more important to document the interface (the header file) than the implementation. The reason is that anyone who uses your component will have to read its header file, but only you (or perhaps members of your team) will have to read the implementation file.

The documentation should be *appropriate*. Documentation in a header file should be aimed at people who will use the component; they should not need to know how it works, how the data is represented, or what clever optimizations you have made. Write down exactly what they need to know in order to use your component, no more and no less.

**Rule 7** *If there are circumstances in which the behaviour of your component is undefined, say so explicitly.*

This is bad:

```
double log(double x);
    // Return the natural logarithm of x if x is positive.
```

This is better:

```
double log(double x);
    // x >= 0: returns natural logarithm of x.
    // x < 0: behaviour is undefined.
```

The “undefined behaviour” might in fact be an assertion failure. The `assert` statement not only performs a safety check but also contributes to the documentation of the function definition in the implementation file:

```
double log(double x)
{
    assert(x > 0);
    ....
}
```

Here is an alternative form of the documentation:

```
double log(double x);
    // x >= 0: returns natural logarithm of x.
    // x < 0: throws exception "Log: negative argument".
```

In general, as we have mentioned previously, coding standards are helpful but not essential — you don't want to irritate your star programmers by imposing stringent but meaningless standards on them. In header files, however, *coding standards should be applied*, because each person working on the project must read many header files; a consistent format will avoid wasted time.

### 3.4 Physical Design

Most discussions of design focus on the *logical* aspects of design; topics include:

- ◇ What is the relationship between two classes?
- ◇ How can coupling between classes be reduced?
- ◇ Are classes sufficiently cohesive?

In small and medium systems, the logical design decisions are the most important. In large systems, physical design decisions are also important.

*Physical* aspects of design include: the organization of the program into files, directories, and processors; the dependencies between files and how they affect compilation, linking, and execution; measures that must be taken to ensure that the system and its components are testable, maintainable, and reusable. A properly designed component can be lifted as a single unit from one system and used in another system. As well as being the basic units for physical design, components are the basic units for *reuse*.

A *component* is the basic, indivisible unit of physical design. Physically, a component consists of a *header file* and an *implementation file*. Logically, it typically consists of one or more classes and associated constants, enumerations, and free functions. Figure 11 outlines a typical component.

```
// component.h                                // component.cpp
    #ifndef COMPONENT_H                        #include "component.h"
    #define COMPONENT_H
                                                // other #includes
    #includes // avoid if possible!           // declarations and definitions
                                                // declarations and definitions
    #endif
```

Figure 11: Outline of a component

In all cases, there is a *physical dependency*: the implementation file depends on the header file because it contains the `#include` directive. In most cases, there will be other physical dependencies as well, because either file may contain additional `#includes`.

**Rule 8** *Entities declared within a component should not be defined outside the component.*

Suppose that you declare a class `Widget` in a header file `Gadget.h`. Then all of the definitions associated with class `Widget` must be in the file `Gadget.cpp`.

### 3.4.1 Header Files

The following rules are for writing the header files of a component.

**Rule 9** *Declare classes, structures, unions, and free operator functions at file scope in header files.*

**Rule 10** *Define classes, structures, unions, and inline functions at file scope in header files.*

**Rule 11** *Prefer declarations to definitions.*

This header file won't compile:

```
// x.h
#ifndef X_H
#define X_H

class X
{
public:
    X();
    enum DIRECTION { LEFT, RIGHT, UP, DOWN };
    DIRECTION whichWay();
    friend ostream & operator<<(ostream & os, X x);
private:
    DIRECTION dir;
};
#endif
```

The reason is that the compiler does not recognize `ostream`. Many people fix this problem by writing

```
#include <iostream.h>
```

in the header file, thereby forcing the compiler to read several thousand lines of code. In fact, the only thing that you have to add to the header file is

```
class iostream;
```

In general, a class declaration is sufficient if the header file mentions a pointer or a reference to a class. Only if the header file mentions the class itself do you need to `#include` the class declaration.

In the following header file, the compiler needs to see the complete definition of class `Wheel` but needs only a declaration for class `Person`.

```
// car.h
#ifndef CAR_H
#define CAR_H

#include "wheel.h"
class Person;

class Car
{
private:
    Wheel wheels[4];
    Person *owner;
};
#endif
```

In this example, the classes `Car` and `Wheel` are very closely related: a `Car` contains an array of `Wheels`. There are other ways of defining these classes that might be better:

- ◇ The class definitions of both `Wheel` and `Car` could be put into a single header file. This avoids the need for `#include` in a header file.
- ◇ If class `Car` is the only class that uses `Wheel`, then the definition of `Wheel` could be nested inside the definition of `Car`.

### 3.4.2 Implementation Files

**Rule 12** *The first `#include` directive in an implementation file must be the header file that belongs to the implementation file.*

For example:

```
\\ whatsit.cpp
#include "whatsit.h"
....
```

The purpose of this rule is to ensure that header files do not require implicit information. In other words, it must be possible to compile them in isolation. Suppose we declared `Car` as above, but without some of the `#include` directives:

```
// car.h
#ifndef CAR_H
#define CAR_H

class Car
{
private:
    Wheel wheels[4];
    Person *owner;
};
#endif
```

We could write `car.cpp` like this:

```
#include "person.h"
#include "wheel.h"
#include "car.h"

....
```

This works fine, but it will create a problem when someone else `#includes` `car.h` but does not `#include` `person.h` and `wheel.h`.

Most of the time, things like this don't matter. The first person who finds that `car.h` doesn't work by itself reports the problem to the team managing `Car` and they fix it. But, every now and then, someone rushing to fix a problem makes a small change at the last minute and suddenly the system fails to compile. Then there's a panic while everyone rushes about trying to find the cause of the problem.

**Rule 13** *Clients of a component must only require to use the `#include` directive for the component in their implementation file.*

Suppose that component `Menu` depends on component `Window`. Then the implementation file for `Menu` should start like this:

```
// menu.cpp
#include "menu.h"
#include "window.h"

....
```

No other `#include` directives should be necessary to express this dependency.

Note that this implies that it will sometimes be necessary to have `#include` directives in header files.

This rule applies to inheritance dependencies. Suppose there is a base class `Parent` and a derived class `Child`. If you want to use the derived class only, you should only need one directive: `#include "child.h"`. This implies that either `Parent` and `Child` are defined in the same file or that the header file looks like this:

```
// child.h
#ifndef CHILD_H
#define CHILD_H

#include "parent.h"

class Child
{
    ....
}
```

**Rule 14** *All definitions with external linkage in the implementation file must have corresponding declarations in the header file.*

Suppose that a system contains the implementation files shown in Figure 12. There are two hidden dependencies in this system (they are “hidden” in the sense that there is no evidence of them in the header files):

- ◊ Sneaky depends on the definition of `size` in Doubtful.
- ◊ Doubtful depends on the definition of `funny()` in Sneaky.

```
// doubtful.cpp                                // sneaky.cpp
    int size;                                    extern int size;
    void funny()                                  void funny();
    {
        ....
    }
```

Figure 12: Hidden dependencies

If this rule is followed consistently, the system will have the following property:

All physical dependencies in the system can be inferred from the `#include` files.

This is a very useful property. In fact, many commercial tools read header files *only* when inferring dependencies. Violations of Rule 14 will prevent these tools from working correctly!

### 3.4.3 Digression: redundant include guards

Include guards avoid the problem of the compiler *processing* the header file more than once but the compiler still has to *read* the header file until it finds `#endif`. In very large systems, this can be avoided by inserting *redundant include guards*, as shown below. Note that this requires a consistent convention for include guard identifiers.

```
#include "car.h"

#ifndef PERSON_H
#include "person.h"
#endif

#ifndef WHEEL_H
#include "wheel.h"
#endif

....
```

10

## 3.5 Friends and Enemies

In physical design, we distinguish close friends and distant friends.

- ◊ A *close friend* is a friend defined in the same component.
- ◊ a *distant friend* is a friend defined in a different component.

**Rule 15** *Avoid distant friends.*

The following example illustrates the dangers of distant friends. First, we define a class `Jail`. A jail has a distant friend class `Key` and a `private` function `release`. There are other member functions but they are not relevant to the example.

```
// jail.h
    #ifndef JAIL_H
    #define JAIL_H

    class Key;

    class Jail
    {
        friend Key;
        void release();
        void beat();
        void starve();
        void torture();
        ....
    };

    #endif
```

The implementation file for class `Jail` defines `release` and other functions, not shown here.

```
// jail.h
    #include "jail.h"
    #include <iostream.h>

    void Jail::release()
    {
        cout << "Escaped!" << endl;
    }

    ....
```

Prisoners in the jail are allowed to have visitors. An instance of `Visitor` clearly has to know how to find the jail.

```
// visitor.h
    #ifndef VISITOR_H
    #define VISITOR_H

    class Jail;

    class Visitor
    {
    public:
```

```

        Visitor(Jail & jail);
};

#endif

```

One day, we let a visitor in, and the prisoner escapes!

```

// escape.cpp
#include "jail.h"
#include "visitor.h"

int main()
{
    Jail jail;
    Visitor bugsy(jail);
    return 0;
}

>a.out
Escaped!
>

```

How did this happen? The implementation file for `Visitor` defines a *local* class called `Key`. This class has no external linkage, but qualifies as the “friend” named in class `Jail`. The constructor for `Key` calls the private function `release`. The compiler allows this because `Key` is a friend.

```

// visitor.cpp
#include "visitor.h"
#include "jail.h"

class Key
{
public:
    Key(Jail & jail)
    {
        jail.release();
    }
};

Visitor::Visitor(Jail & jail)
{
    Key key(jail);
}

```

Presented in this way, the insecurity is obvious. In a large system consisting of hundreds of classes, however, there are at least two ways in which such a situation could plausibly occur.

- ◊ Harry Hacker has promised his manager that he will have his component working by Monday morning. At 5 p.m. on Friday, Harry realizes that his code will work only if

he can call `Jail::release()`. Carol Careless, who wrote the `Jail` class with its distant friend, has just left town for a weekend Snooker Tournament. In desperation, Harry concocts the `Visitor` so that his code will work. The tests are run successfully, Harry moves on to something else, and the kludge is forgotten — for a while.

- ◇ The system is large, coding is behind schedule and getting out of control. Responsibilities have been assigned, but there are some fuzzy areas. Looking through the code for missing components, Jack and Jill both realize that no one has implemented the class `Key` mentioned in `Jail.h`. Working in separate rooms, they quickly code the component that defines and uses `Key`. The overnight build is about to start and, to avoid possible linkage problems, both programmers define `Key` as a local class in an implementation file. The system compiles and everything seems fine . . . until Janet, who owns `Jail`, discovers that its `private` function `release` executes although the `Key` object — the only one that she knows about — is not active.

### 3.6 Levels

A component  $X$  *depends on* a component  $Y$  if  $X$  cannot be compiled or linked without  $Y$  being present.

If the rules above have been followed, we can assert:

Component  $X$  depends on component  $Y$  if and only if there is an `#include "y.h"` directive in either `x.h` or `x.cpp`.

If this is true, we can construct a dependency graph for the system on the basis of `#include` directives (in header files and implementation files) only. The graph includes components that our system uses but which are not part of it, such as vendor-supplied libraries.

The *dependency graph*  $G = (V, E)$  for a system has one vertex  $v \in V$  corresponding to each component of the system and a directed edge  $(u, v) \in E$  if component  $u$  depends on component  $v$ .

*Levelling* is a procedure that assigns a *level number* to each vertex of the dependency graph as follows:

- ◇ A component that is external to our system has level 0.
- ◇ A component that may be dependent on external components but does not depend on any component in our system has level 1.
- ◇ A component that depends on a component at level  $n - 1$  but on no components at higher levels has level  $n$ .

The levelling procedure will terminate if and only if the dependency graph has no cycles. As we have already seen, there are good reasons for avoiding cycles.

There are CASE tools that perform levelling by scanning header files. Following the rules that we have given is important because levelling may be carried out automatically.

Levelling makes *component testing* feasible. We can test bottom-up (levels 1, 2, 3, . . .) or top-down (levels  $n, n - 1, n - 2, \dots$ ). Both kinds of testing require scaffolding: bottom-up testing requires drivers and top-down testing requires stubs.

Levelling also has the important effect of making the system easier to understand. As with testing, a new maintenance programmer may choose to learn the system either bottom-up or top-down, according to taste. Either way, the level numbers provide guidance to the dependencies in the system.

### 3.6.1 Dependency Metrics

The *cumulative component dependency* (CCD) of a (sub)system is the sum over all components  $C_i$  of the (sub)system of the number of components needed to test each  $C_i$  incrementally.

The rationale for CCD is that testing time is dominated linking: the CCD provides a rough estimate of the time required to link for each test.

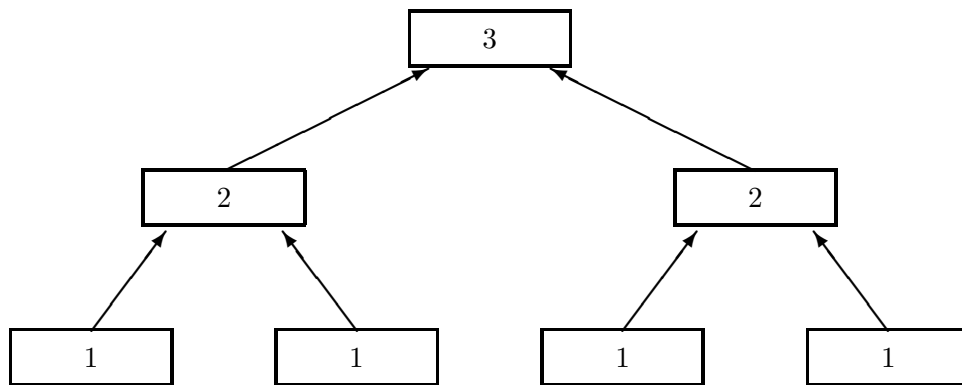


Figure 13: A system with binary tree structure

Figure 13 shows a system with binary tree structure. The CCD is computed as follows:

4 components at level 1 with no dependents	$4 \times 1 = 4$
2 components at level 2, each with 2 dependents	$2 \times 3 = 6$
1 component at level 3 with 6 dependents	$1 \times 7 = 6$
CCD	17

In general, the CCD for a balanced binary tree with  $n$  components is  $(n + 1) \log_2(n + 1) - n$ . In the example above,  $n = 7$  and  $(7 + 1) \log_2(7 + 1) - 7 = 17$ .

In the worst case, the dependency graph for a system with  $n$  components is the (cyclic) complete graph with  $n$  vertices. In this case, each of the  $n$  components depends on all of the other  $n - 1$  components and the CCD is  $n^2$ .

In practice, the binary-tree structure is the best possible and the complete graph is the worst possible. Thus the CCD will lie between these extremes. For 100 components, the CCD lies between 571 and 10,000.

A system does not have to be cyclic to have a high CCD. A *vertical system* has components  $C_1, C_2, \dots, C_n$  such that  $C_{n+1}$  depends on  $C_n$ . In order to test  $C_i$ , we have to link  $C_1, C_2, \dots, C_i$ . Thus the CCD is  $\frac{1}{2}n(n + 1)$ , which also increases quadratically with  $n$ .

The following condition is a consequence of the CCD for a vertical system:

A system of  $n$  components with  $\text{CCD} > \frac{1}{2}n(n + 1)$  has at least one cyclic dependency.

The *average component dependency* (ACD) of a system with  $n$  components is  $\frac{\text{CCD}}{n}$ .

From the results above, we have:

ACD for a binary-tree system with $n$ components	$O(\log n)$
ACD for a vertical system	$O(n)$
ACD for a cyclic system	$O(n)$

The ACD is useful because it enables us to estimate the effect of changing an interface. On average, changing one interface will affect ACD components of the system. For the system in Figure 13,  $\text{ACD} = 17/7 \approx 2.4$ : changing one interface will, on average, require checking 2.4 components. For a cyclic system of 7 components, changing one interface requires checking 7 components, as we would expect.

The *normalized cumulative component dependency* (NCCD) is the CCD of the system divided by the CCD of a tree-structured system of the same size.

A tree-like system has  $\text{NCCD} = 1$ . If  $\text{NCCD} < 1$ , the system must be very loosely coupled (the dependency graph is not connected!) and is probably not useful. Normally, NCCD will be greater than 1, but not much. An NCCD significantly larger than 1 is an indication that the system is too tightly coupled and may contain cycles.

Now we know that low values of NCCD are good, we can consider the problem of how to achieve them

### 3.7 Levelization

*Levelization* is the process of organizing (or re-organizing) software to reduce its ACD. Since this requires that we assign levels to each component and then reduce the number of compile-time and link-time dependencies, it is sometimes called “levelization”. “Coupling reduction” would be another name.

In this section, we describe several miniature examples of excessive coupling and its avoidance. As presented, the examples are trivial. But similar situations do arise in real-life software development and it is important to know how to recognize the problems and what to do about them.

The initial design of a system is often “clean”. Levelization problems are considered and solved. Later in the development cycle, quick fixes to problems may introduce new dependencies. Maintenance programmers who do not understand the system well may also introduce new dependencies, often for apparently good reasons such as performance or convenience. For this reason, we present some of the example below as maintenance scenarios.

**Example 11.** A system has two classes, `Window` and `Rectangle`:

```

// window.h                                // rectangle.h

class Window                                class Rectangle
  // A window is defined by its           // A rectangle is defined by its
  // centre, width, and height.           // bottom left and top right corners.
{
public:
  Window(int xc, int yc,                   Rectangle(int xBL, int yBL,
          int width, int height);          int xTR, int yTR);
  ....
};                                          };

```

These classes were originally introduced for different purposes, but it turns out that programmers frequently convert `Windows` to `Rectangles` and *vice versa*.

A maintenance programmer, trying to be helpful, adds two conversion functions. They are `inlined` for efficiency.

```

// window.h                                // rectangle.h

#include "rectangle.h"                       #include "window.h"

class Window                                class Rectangle
  // A window is defined by its           // A rectangle is defined by its
  // centre, width, and height.           // bottom left and top right corners.
{
public:
  Window(int xc, int yc,                   Rectangle(int xBL, int yBL,
          int width, int height);          int xTR, int yTR);
  Window(const Rectangle & r)             Rectangle(const Window & w);
    // new constructor
  {
    ....
  }
  ....
};                                          };

```

This change introduces a *cyclic dependency* into the system: we can no longer compile, link, or test one class without the other. □

**Improvement 1.** The first improvement that we can make is to move the `#includes` to the implementation files. Although this removes the cyclic dependency between header files, it doesn't really fix the problem because there is still a cyclic dependency in the component dependency graph. Also, we have slowed down the system because the conversion functions can no longer be `inlined` (this might or might not be critical, depending on the importance of performance for this system).

**Improvement 2.** The maintenance programmer apparently wanted to keep the two classes at the same level. But this is not necessary, because we could solve the problem by introducing an asymmetric dependency — undesirable, but better than a cyclic dependency.

For example, we could leave `Rectangle` alone but make `Window` dependent on it. We could avoid the `#include` directive in `window.h`, but only by making the conversion function not `inline`.

```
// window.h

#include "rectangle.h"

class Window
    // A window is defined by its
    // centre, width, and height.
{
public:
    Window(int xc, int yc,
           int width, int height);
    Window(const Rectangle &);
    operator Rectangle() const
    {
        ....
    }
    ....
};

// rectangle.h

class Rectangle
    // A rectangle is defined by its
    // bottom left and top right corners.
{
public:
    Rectangle(int xBL, int yBL,
              int xTR, int yTR);
    ....
};
```

**Improvement 3.** A better solution than either of the above is to introduce a third class that uses both `Window` and `Rectangle`. The original classes are left unchanged, and therefore independent of one another. The new class looks like this:

```
// windirect.h

#ifndef WINDRECT_H
#define WINDRECT_H

class Window;
class Rectangle;

class WindRect
{
public:
    static Window RectToWin(const Rectangle & r);
    static Rectangle WinToRect(const Window & w);
};

#endif
```

### 3.7.1 Levelization with Inheritance

Inheritance introduces one obvious dependency into a system: a derived class depends on its base class. This dependency is unavoidable. However, inheritance may introduce other

dependencies into the system that are avoidable, provide that we do the physical design carefully.

A *class hierarchy* consists of a *base class* and one or more *derived classes*. An inheritance hierarchy is not necessarily a tree but it is necessarily acyclic.

**Example 12.** A graphical object editor has an inheritance hierarchy consisting of a base class `Shape` and derived classes `Triangle`, `Square`, `Circle`, etc. To reduce dependencies, class `Shape` is the only class in the hierarchy that is accessible to the rest of the system (this is an example of the *Façade pattern*). Since other system components must be able to construct instances of graphical objects, class `Shape` contains the following code:

```
enum SHAPETYPE { TRIANGLE, SQUARE, CIRCLE };

Shape *Shape::create(SHAPETYPE st)
{
    switch (st)
    {
        case TRIANGLE:
            return new Triangle();
        case SQUARE:
            return new Square();
        case CIRCLE:
            return new Circle();
    }
}
```

This function introduces cyclic dependencies between `Shape` and each of its subclasses. The code above shows that `Shape` depends on each subclass. The code for each subclass shows the reverse dependency:

```
class Triangle : public Shape { .... };
class Square : public Shape { .... };
class Circle : public Shape { .... };
```

□

**Improvement:** Move the `create` function to a new class `ShapeConstructor` that is know to each subclass of `Shape`. This increases the number of levels in the system but removes the cyclic dependencies. Figure 14 provides an outline of the resulting code. To obtain a concrete shape, clients write:

```
#include "shapeconstructor.h"

....

Shape *box = ShapeConstructor::create(ShapeConstructor::SQUARE);
```

```
// shape.h
    class Shape { .... };

// square.h
    class Square : public Shape { .... };

// triangle.h
    class Circle : public Shape { .... };

// circle.h
    class Triangle : public Shape { .... };

// shapeconstructor.h

class Shape;

class ShapeConstructor
{
public:
    enum SHAPETYPE { TRIANGLE, SQUARE, CIRCLE };
    static Shape *create(SHAPETYPE st)
    {
        switch (st)
        {
            case TRIANGLE:
                return new Triangle();
            case SQUARE:
                return new Square();
            case CIRCLE:
                return new Circle();
        }
    }
};
```

Figure 14: Improved code for graphical editor (guards omitted)

Suppose that the editor itself has several components, all of which use `Shape`, but only one of which uses `ShapeConstructor`. Figure 15 shows the NCCDs for various numbers of `Shape` subclasses and editor components. Note that the coupling reduction improves significantly as the number of classes and components grows.

System		NCCD	
Shape classes	Editor components	Without <code>ShapeConstructor</code>	With <code>ShapeConstructor</code>
4	4	1.90	1.28
4	31	1.33	0.90
31	4	7.23	1.06
31	32	6.30	0.77

Figure 15: Effect of class `ShapeConstructor` on NCCD

**Abstract Base Classes.** The simplest kind of inheritance hierarchy consists of a *base class* and a number of *derived classes*.

The base class defines a *protocol* that all of the derived classes must support. The protocol is defined as a set of virtual functions in the base class. C++ provides two ways of defining these functions:

- ◇ A *pure virtual* function has no implementation in the base class. The notation is

```
void f() = 0;
```

A class that has one or more pure virtual functions cannot be instantiated (that is, no objects belonging to the class can be constructed).

- ◇ A virtual function can be implemented in the base class. The implementation provides default behaviour that may be overridden in derived classes.

A class in which no functions have implementations is called an *abstract base class*. (Java interfaces are essentially abstract classes.)

**Rule 16** *All base classes should be abstract.*

The problem with this rule is that it sometimes seems to lead to redundancy and inefficiency. If there is common code in the derived classes, it seems sensible to move it into the base class.

**Example 13.** The following code is a small extract from an accounting package. Class `Form` is the base class of a hierarchy. The hierarchy includes classes such as `Cheque`, `Deposit`, and `Adjustment`, corresponding to the different kinds of transactions and the way in which they are presented to the user — as forms on the screen.

Here is part of class `Form`:

```
class Form
{
```

```

public:
    virtual void display() = 0;
    // Other pure virtual functions
protected:
    void setField(int field);
    int checkTax();
    void showTrans();
    int numFields;
    char *title;
};

```

Note that the `protected` functions are not virtual. They are not overridden in derived classes because the behaviour they describe is common to all of the derived classes. Consequently, class `Form` violates Rule 16. Note also that `Form` has `protected` data members that are used by derived classes. □

To avoid violating Rule 16, we could:

- ◇ Copy the definitions of the `protected` functions into each of the derived classes. There are several objections to doing this:
  - The executable gets larger because of duplicated code.
  - Changing one function probably requires changing all of them.
  - Even worse, the functions may become customized within their classes, giving us a system with numerous small variations on a common theme.
  - It is never a good idea to duplicate code.
- ◇ The alternative, and preferable, approach, is to introduce a new base class that contains information common to the hierarchy but is not part of the interface.

If we follow the alternative approach, class `Form` becomes an abstract class:

```

class Form
{
public:
    virtual void display() = 0;
    // Other pure virtual functions
};

```

The new class, `FormImplementor` has a `protected` interface, because its only clients are the classes derived from it.

```

class FormImplementor
{
protected:
    FormImplementor();
    void setField(int field);
    int checkTax();
    void showTrans();
    int numFields;
};

```

```

        char *title;
        ....
};

```

A client of `FormImplementor` must `#include` both header files and use multiple inheritance to get the information it needs from both `Form` and `FormImplementor`

```

// cheque.h
#ifndef CHEQUE_H
#define CHEQUE_H

#include "form.h"
#include "formimpl.h"

class Cheque : public Form, public FormImplementor
{
public:
    Cheque(int size);
    void display();
};

#endif

// cheque.cpp
#include "cheque.h"
#include <iostream.h>

Cheque::Cheque(int size)
: FormImplementor(size)
{
}

void Cheque::display()
{
    ....
}

```

### 3.7.2 Opaque Pointers

A pointer is *opaque* if the definition of its class is not included in the current translation unit.

Normally, a pointer implies a dependency. For example, if an implementation file contains the following statement, it would also have to contain `#include "thing.h"`:

```

Thing *p = ....;
....
p->doSomething();

```

However, if `p` was an opaque pointer, then only the declaration `class Thing;` would be necessary.

**Example 14.** Suppose we have a class `Screen` that has pointers to a number of instances of `Widget`. (Presumably, the widgets will be displayed on the screen.) A `Widget` client might want to know how many widgets are currently on the screen. Here is a first attempt at an implementation. Note that:

- ◇ A client who owns a pointer `pw` to a `Widget` can execute
 

```
numParents = pw->numParentWidgets();
```
- ◇ The file `screen.cpp` must have `#include "widget.cpp"` because `Screen::refresh()` calls `Widget::refresh()`.
- ◇ The file `Widget.cpp` must have `#include "screen.h"` because `Widget::numParentWidgets()` calls `Screen::getNumWidgets()`.

Consequently, we have a cyclic physical dependency. Each class has a pointer to the other, and neither pointer is opaque. □

Here is the `Screen` component:

```
// screen.h
#ifndef SCREEN_H
#define SCREEN_H

class Widget;

class Screen
{
public:
    Screen();
    void addWidget(Widget *w);
    int getNumWidgets();
    void refresh();
private:
    int numWidgets;
    Widget *widgets[20];
};

#endif

// screen.cpp
#include "screen.h"
#include "widget.h"

Screen::Screen()
    : numWidgets(0)
{
}

void Screen::addWidget(Widget *w)
{
    widgets[numWidgets++] = w;
}
```

```

    }

    int Screen::getNumWidgets()
    {
        return numWidgets;
    }

    void Screen::refresh()
    {
        for (int w = 0; w < numWidgets; w++)
            widgets[w]->refresh();
    }

```

And here is the Widget component:

```

// widget.h
#ifndef WIDGET_H
#define WIDGET_H

class Screen;

class Widget
{
public:
    Widget (Screen *screen);
    int numParentWidgets();
    void refresh();
private:
    Screen *myScreen;
};

#endif

// widget.cpp
#include "widget.h"
#include "screen.cpp"

Widget::Widget (Screen *screen)
    : myScreen(screen)
{
}

int Widget::numParentWidgets()
{
    return myScreen->getNumWidgets();
}

void refresh()
{

```

```

    ....
}

```

We can remove the cyclic dependency by making one of the pointers opaque. It is most appropriate in this example to make the pointer `Screen*` in `Widget` opaque.

To do this, we have to rewrite class `Widget` in such a way that neither of its files need the directive `#include "screen.h"`. To do this, we must ensure that `Widget` does not call any of `Screen`'s functions. We can, however, include a function that returns a pointer to the `Widget`'s `Screen`.

Here is the new `Screen` component:

```

// screen.h
#ifndef SCREEN_H
#define SCREEN_H

class Widget;

class Screen
{
public:
    Screen();
    void addWidget(Widget *w);
    int getNumWidgets();
    void refresh();
private:
    int numWidgets;
    Widget *widgets[20];
};

#endif

// screen.cpp
#include "screen.h"
#include "widget.h"

Screen::Screen()
    : numWidgets(0)
{
}

void Screen::addWidget(Widget *w)
{
    widgets[numWidgets++] = w;
}

int Screen::getNumWidgets()
{
    return numWidgets;
}

```

```

    }

    void Screen::refresh()
    {
        for (int w = 0; w < numWidgets; w++)
            widgets[w]->refresh();
    }

```

And here is the new `Widget` component:

```

// widget.h
#ifndef WIDGET_H
#define WIDGET_H

class Screen;

class Widget
{
public:
    Widget (Screen *screen);
    Screen *parent() const;
private:
    Screen *myScreen;
};

#endif

// widget.cpp
#include "widget.h"

Widget::Widget (Screen *screen)
    : myScreen(screen)
{
}

Screen *Widget::parent() const
{
    return myScreen;
}

```

We have removed the cyclic dependency: it is possible to compile and link `Widget` without the `Screen` class. We can test `Widget` in isolation: all we need is a dummy screen class that provides a stub for `getNumWidgets()`.

A minor disadvantage of the new code is that a client of `Widget` needs to know about `Screens` in order to find how many widgets are displayed. This is not a serious problem, because the client would probably have to know about screens anyway:

```

#include "screen.h"
#include "widget.h"

```

```

{
    ....
    numParents = w.parent()->getNumWidgets();
    ....
}

```

12

### 3.8 Levelization: Summary

There are a number of other techniques for levelization. Instead of discussing them in detail, we will briefly summarize some of them.

#### 3.8.1 Dumb Data

Suppose we are designing a horseracing system. There will be a number of horses, and several classes that need to know about horses — `Track`, `Race`, `Bet`, and so on. A natural solution would be to allow these classes to have pointers to instances of `Horse`. Given that there will be other dependencies between the classes, however, it is likely that this will create tight, or even cyclic, coupling.

We can avoid this by including an array of horses in one of the classes — probably `Track` — and letting other classes use indexes into this array.

- ◇ Since array indexes are `ints` (or perhaps `shorts`), they do not lead to any compile-time or link-time dependencies.
- ◇ An index reveals less information than a pointer.
- ◇ There is a trade-off: it may be necessary to provide additional functions such as

```
char * Track::getHorseName(int hIndex);
```

to access horses and their attributes.

- ◇ If the number of references is very large (unlikely for horses but quite possible for another application, such as transistors on a chip) the saving of space could be significant: 16 bits is probably enough for an array index, but 32 bits are needed for a pointer.

#### 3.8.2 Redundancy

As a general rule, it is best to avoid duplicating code. If a small function or class is used in several places in a system, however, providing local copies where it is needed will lead to less coupling than providing one copy with many links to it. If changes become necessary, clients can change their own copies without affecting other components.

In many cases, a small amount of redundant data can change a physical dependency to a “name only” dependency with no compile-time or run-time consequences.

### 3.8.3 Manager Classes

Don't be afraid to introduce "manager" classes even when they don't seem strictly necessary. For example, suppose that you have decided to store a collection of `Nodes` in a linked list. Adding the data member

```
Node *next;
```

to the class declaration confuses things, because it's not clear whether `Node *p` means that `p` is a pointer to a `node` or a pointer to a list of `Nodes`.

It is better to introduce two new classes:

- ◇ `List` manages a list of `Links`.
- ◇ `Link` is a link object in a list. Each `Link` contains a pointer to the next `Link` and a pointer to a `Node`.

Obviously, we should generalize this by using templates whenever possible. Even better, use the facilities provided by STL.

### 3.8.4 Encapsulation

It is not necessary for every class to be fully encapsulated. If we protect a subsystem using the `Facade` pattern, the facade class must hide all details of the subsystem. But the classes that belong to the subsystem need not be fully encapsulated, because they should never be accessed directly by clients.

### 3.8.5 Summary

Consider using any of the following techniques to levelize and reduce coupling within a system:

- ◇ Move mutually dependent functionality higher in the physical hierarchy
- ◇ Move common functionality lower in the hierarchy
- ◇ Use opaque pointers where feasible
- ◇ Use dumb data where possible
- ◇ Introduce manager classes
- ◇ Introduce facades for encapsulation

13
----

## 3.9 Insulation

A class is *encapsulated* if a client cannot write calls that depend on the implementation. Insulation is a stronger property: a class is *insulated* if clients do not have a compile-time dependency on the class declaration.

Consider the following (incomplete) declarations of a class `Stack`.

```

class Stack
{
public:
    Stack();
    void push(int value);
    int pop();
private:
    int size;
    int *elements;
};

class StackLink;

class Stack
{
public:
    Stack();
    void push(int value);
    int pop();
private:
    StackLink *stackData;
};

```

Since both declarations provide the same public member functions, we can substitute one for the other without affecting client code. Nevertheless, all clients would have to be recompiled, because of differences in the *private* parts of the classes. Consequently, each version of class `Stack` *is* encapsulated but *is not* insulated.

Insulation does not matter much for small systems which can be recompiled in a few seconds or a few minutes. It matters quite a lot for large systems which may take hours or days to recompile. Consequently, we look briefly at some methods for insulating classes.

Insulation is not an all-or-nothing property. There are situations in which leaks can be reduced easily but eliminated only with difficulty. Each reduction in leakage (or improvement in insulation) reduces the probability that a change will trigger recompilation. Consequently, even small improvements may be worth while.

### 3.9.1 Inheritance

Inheritance, even private inheritance, creates a compile-time dependency — a change to the interface of the base class will force recompilation of all clients of the derived class.

If this is a problem, replace inheritance from the base class by an opaque pointer to the base class. It may be necessary to define a few new functions that “forward” messages from clients (who call the old derived class), but changing the old base class no longer requires recompilation.

### 3.9.2 Private Member Functions

Private member functions cannot be called by clients but are nevertheless part of the physical interface: changing them may trigger recompilation. Private member functions can sometimes be replaced by `static` free functions declared in the implementation file.

Since the new free functions are not member functions, they do not have access to the class data. This problem can be fixed by passing them the information they need as arguments.

### 3.9.3 Protected Member Functions

The purpose of `protected` member functions is to provide one interface for normal clients (`public` functions) and another for clients who want to derive a class from this one. There are two disadvantages with `protected` functions:

- ◇ They can be called by anyone who chooses to inherit from your class

- ◇ The constitute a compile-time dependency for all clients

The best way to eliminate `protected` functions is to introduce another class that provides those functions. This class is completely invisible to regular clients. Clients who wish to derive from the original class must also derive from the new class.

### 3.9.4 Protocol Classes

Protocol classes (which correspond to Java interfaces) are nearly perfect insulators. In C++, a protocol class:

- ◇ there are no user-defined constructors
- ◇ there are no private or protected member functions
- ◇ there are no data
- ◇ there is a non-pure virtual destructor
- ◇ all member functions apart from the destructor are pure virtual

A protocol class serves purely as an interface to another class that is implemented in the usual way. Clients know only the protocol class. They construct instances of this class, if they need to, using a factory method (or class).

## 3.10 Designing a Function

C++ provides many options for defining functions. The following sections discuss the pros and cons of various choices.

### 3.10.1 Should it be an operator?

- ◇ Use operators when the notation will be convenient, but not confusing, for clients.
- ◇ The interpretations of the operators should be natural, not forced. Tricks, such as using `operator~` to reverse a string, are not helpful.
- ◇ As far as possible, operators should obey conventional algebraic laws. For example, `+` should be commutative and identities such as  $a + (-b) = a - b$  should hold.
- ◇ Operators should also respect C++ conventions. For example, all assignment operators, and the pre-decrement and pre-increment operators, return modifiable lvalues.
- ◇ Be careful to avoid memory leaks. It is very easy to implement operators by value, but more or less impossible to implement with pointers (because temporaries are allocated and must be deleted).
- ◇ If you choose to implement operators, implement a complete set. Clients will be annoyed if they discover that they have to write `a = a + b` instead of the C++ idiom `a += b`.

### 3.10.2 Should it be a member or free operator?

There are two criteria: preserve commutativity and avoid unexpected conversions.

Member functions are typically asymmetric. For example, if we declare

```
class Vector
{
    public:
        Vector operator*(double s);
        ....
};
```

then we can write  $v * s$  but not  $s * v$  (in which  $v$  is a vector and  $s$  is a scalar, i.e., `double`). But if we declare

```
class Vector
{
    friend Vector operator*(Vector v, double s);
    friend Vector operator*(double s, Vector v);
    public:
        ....
};
```

then both  $v * s$  and  $s * v$  will work (provided that we avoid ambiguity).

Both operands of a free function are subject to implicit conversion, but only the right operand of a member function can be converted.

```
class String
{
    public:
        String(const char *s);
        ....
};

String & operator+=(String & left, const String & right)
{
    ....
}

int main()
{
    const char *y = "silly ";
    String z("world!");
    y += z;
}
```

This code compiles but has no effect on `y`. The reason is that the `String` constructor allows `y` to be converted from `char *` to `String`, but stores this `String` in a temporary. What is even more confusing is that if we follow this code with

```
String x("Hello, ");
x += y += z;
```

then `x` does get the value "Hello, silly world".

Note that there are a few anomalies in C++. For example, we might expect `==` and `+` to behave more or less in the same way (as symmetric binary operators) but they don't. The reason is that `x == y` compiles if both `x` and `y` are both convertible to pointers, because pointers can be compared, but `x + y` compiles only if there is an appropriate overloaded function definition in scope.

The following operators must be member functions (they cannot be declared as free functions):

```
=    []    ->    ()    (T)    new    delete
```

### 3.10.3 Should it be virtual?

The basic rule is that a function should be declared `virtual` in a base class if its behaviour is (or might be) overridden in a derived class.

It is possible to define symmetric operators as free functions and still maintain polymorphic behaviour.

```
class Shape
{
public:
    virtual int compare(const Shape &) const = 0;
    ....
};

inline bool operator==(Shape &lhs, Shape &rhs)
{
    return lhs.compare(rhs) == 0;
}

inline bool operator>=(Shape &lhs, Shape &rhs)
{
    return lhs.compare(rhs) >= 0;
}

....
```

Virtual functions implement variation in *behaviour*; data members implement variation in *value*. Don't use a virtual function where a data member would do. For example, `getColour()` probably doesn't have to be a virtual function: the colour attribute can be represented by a data member in the base class.

### 3.10.4 Should it be pure virtual?

If a function is declared pure virtual in a base class, then any concrete derived class must provide a definition.

### 3.10.5 Should it be static?

Use `static` for a member function that belongs in a class but which does not refer to the data members of an individual object.

Note that `static` member functions cannot access data members unless the data members are `static`.

```
class Widget
{
    public:
        Widget() { instances++; }
        ~Widget() { instances--; }
        static int howMany() { return instances; }
    private:
        static int instances;
};

Widget::instances = 0;
```

### 3.10.6 Should it be const?

Functions that do not change the object *physically* should be declared `const`.

Unfortunately, C++'s notion of `const` is physical, not logical. For example, consider a class `HashTable` with a function `find`. If `find` can reorganize the hash table while searching, it cannot be declared `const`, even though reorganization does not change the hash table from the user's point of view (except perhaps for access time).

The compiler's view of `const` is not always intuitive. In class `String`, below:

- ◇ `MakeEmpty` can be declared `const` because the value of the pointer `data` is not changed, even though what it points to is changed.
- ◇ `getRepRef` cannot be declared `const` because it returns a reference to the pointer; a client could write

```
String s("hello");
getRepRef(s) = "goodbye";
```

```
class String
{
    public:
        void makeNull() { data = NULL; }
        void makeEmpty() const { data[0] = '\0'; }
        char *getRep() const { return data; }
        char *&getRepRef() { return data; }
    private:
        char *data;
};
```

### 3.10.7 Should it be public, protected, or private?

Any member function that clients are expected to use should obviously be declared `public`. `protected` and `private` functions should be avoided if possible because they introduce compile-time dependencies (see Section 3.9.1).

### 3.10.8 How should it return a value?

There are five ways to return a `Thing`:

```
Thing a(...);           // return by value
const Thing & b(...);  // return by reference
const Thing * c(...);  // return by pointer
void d(Thing * ret);    // return by argument
void e(Thing & ret);    // return by reference argument
```

Return by value is always safe but may be inefficient, because the copy constructor and destructor are called.

Return by reference is faster than return by value but can only be used if there is a reference to return. It is a *serious mistake* to return a reference to a local object. In the code for `Complex` functions below, `operator=` is correct, because there is an object to return a reference to, and `operator+` is wrong, because the result is local.

```
class Complex
{
public:
    Complex & operator=(Complex & rhs)
    {
        re = rhs.re;
        im = rhs.im;
        return *this;
    }
    Complex & operator+(Complex & rhs)    // Wrong!!
    {
        Complex result(re + rhs.re, im + rhs.im);
        return result;
    }
private:
    double re;
    double im;
};
```

Returning a pointer requires caution. It is legal to use `new` to allocate space for an object and then return a pointer to it, but there must be a logical place to put the matching `delete`. It is best to return pointers only to objects that existed before the function was called. For example, a search function `find` can return a pointer to the object it finds, because that object existed before `find` was called.

Note that it is possible to return a NULL pointer (e.g., if no appropriate object was found) but there is no corresponding “null reference”.

The same considerations apply to return by argument and return by reference argument. In most cases, return by reference argument is a *bad idea*, both because the reference is probably local and it is not possible to indicate failure by returning NULL.

### 3.10.9 Should it return const?

There is rarely any point in `const` return by value because it restricts the client. A `const` result cannot be used by another function that is not itself `const`.

It does make sense — sometimes — to return a `const` reference or a `const` pointer.

### 3.10.10 Should arguments be required or optional?

Use default arguments whenever possible, but be aware of silent conversions. The program

```
class Point
{
public:
    Point(int x = 0, int y = 0) : x(x), y(y) {}
    friend Point operator+(const Point & lhs, const Point & rhs)
    {
        return Point(lhs.x + rhs.x, lhs.y + rhs.y);
    }
    friend ostream & operator<<(ostream & os, Point p)
    {
        return os << '(' << p.x << ', ' << p.y << ')';
    }
private:
    int x;
    int y;
};

void main()
{
    Point a = 3;
    Point b = a + 5;
    cout << a << " " << b << endl;
}
```

prints

```
(3,0) (8,0)
```

Default arguments are best for built-in types (`int`, `double`, etc.). Using them for user-defined types is likely to be inefficient because the default values must be constructed even if they are not used.

**3.10.11 Should arguments be passed by value, reference, or pointer?**

Built-in types can be passed in any convenient way. User-defined types should usually be passed by `const` reference. The following example shows an exception to this rule.

```
class BigInteger
{
    public:
        BigInteger(int i);
        ....
};

class BigIntegerCollection
{
    public:
        int add(const BigInteger & bi)
        {
            // Store address of bi in collection
            ....
        }
        bool isMember(const BigInteger & bi) { .... }
        ....
};

void main()
{
    BigIntegerCollection bic;
    bic.add(1);
    bic.add(2);
    bic.add(3);
    bool flag = bic.isMember(2);
}
```

The problem here is that the integers 1, 2, and 3 are converted to `BigIntegers` by the constructor, creating a temporary object. The address of this object is passed to the function `add` and the object is then deleted. The call to `isMember` is likely to crash the program!

**3.10.12 Should arguments be `const`?**

If arguments are passed by value, there is little point in declaring them to be `const`. In

```
void f(int i) { .... }
```

the value of `i` in the function is a copy of the argument, not the argument itself. Consequently, it is of no interest to the client whether it is altered in the function. The only effect of writing

```
void f(const int i) { .... }
```

is to prevent the implementor of `f` from altering `i`.

On the other hand, when arguments are passed by reference, they should be declared `const`, unless the function needs to alter their values.

### 3.10.13 Should it be a friend?

Friends should be avoided whenever possible because friends tend to make maintenance harder. We can often avoid friend functions by careful planning.

```
class Complex
{
    public:
        Complex & operator+=(const Complex & rhs)
        {
            re += rhs.re;
            im += rhs.im;
            return *this;
        }
        ....
};

Complex operator+(const Complex & lhs, const Complex & rhs)
{
    return Complex(lhs) += rhs;
}
```

### 3.10.14 Should it be inline?

Inlining is a trade-off: if the function body is large, inlining it may cause code bloat, because code for the entire function is generated at every call site. For small functions, inlining can improve efficiency but reduces insulation.

Simple access functions (`getX`, `putX`) and simple operators are unlikely to change often. Inlining is efficient and the loss of insulation probably doesn't matter much.

## 3.11 Summary

There is a tendency for the number of dependencies in a system to increase in proportion to the square of the number of components. (Remember that a connected graph  $G = (V, E)$  has between  $n - 1$  and  $n(n - 1)/2$  edges, where  $n = |V|$ .) For small systems, this doesn't matter much but, as the system size grows, coupling becomes increasingly important.

High coupling is bad for several reasons:

- ◇ Highly-coupled code is hard to understand and maintain
- ◇ High coupling increases testing time because many components are needed for each test
- ◇ High coupling increases the time required to compile and link

A common observation is that, for large systems, *complexity resides in the connections*. The system may consist of thousands of components, each of which is easy to understand. But the behaviour of the system depends not so much in the individual components as the way in which they work together, which may be very hard to understand.

Well-designed systems tend to be hierarchical. Not in the simple sense, of having a tree structure, but in a deeper sense: the system consists of a few subsystems; each subsystem has a few subsystems; and so on. We can view such a system at different levels of abstraction and, at each level, it is easy to understand.

Many people think that low coupling is achieved by careful design. This is partly true, but good design can only reduce *logical* coupling. To obtain the benefits of low coupling, we must also reduce *physical* coupling.

## 4 Interlude: Java

“Quality” applies to language design as well as to system development. This section reviews Java as an exercise in language design. It is based on *A Critique of Java* by Harold Thimbleby. You can obtain the full report from Thimbleby’s website: <http://www.cs.mdx.ac.uk/harold>.

A *trap* is an aspect of language design that may lead to unexpected problems that are hard to track down. Here are some Java traps.

- ◇ Appending “l” (lower case ell) to a number tells the compiler to use 64 bits to store it. In

```
71
71
```

the first number is 71 (stored in 32 bits) and the second number is 7 (stored in 64 bits). The designers admit this weakness (Gosling 1998, page 108):

L is preferred over 1 because 1 (lowercase L) can easily be confused with 1 (the digit 1).

So why not insist on L?

- ◇ In

```
A
A
```

the first letter is upper case Roman ‘A’ and the second letter is upper case Greek ‘Alpha’. These symbols have different unicodes but the same appearance.

- ◇ Java copies the old style C notation for casts. Thus we have to write the top expression instead of the simpler expression below (which uses postfix casts):

```
((AClass) a.elementAt(n)).action()
```

```
a.elementAt(n)(AClass).action()
```

- ◇ `throw`, `catch`, and `finally` take compound statements only. Thus we must write the top statement rather than the simpler bottom statement.

```
try { a = b / c; }
```

```
try a = b / c;
```

- ◇ There are several different kinds of initialization, depending on the kind of variable (local variable, class variable, parameter) and the type (primitive, class, array). Sometimes initialization is automatic, sometimes it isn’t.

A *barrier* is an unnecessary limitation on expressiveness that makes programs harder to write than they should be.

The first mention of `import` (Gosling 1998, page 25):

And here is a version that uses `import` to declare the type `Date`:

```
import java.util.Date;
....
```

The next reference (Gosling 1998, page 210) mentions restrictions:

A programmer who wants to use the `attr` package could put the following line near the top of a source file (after any package declaration but before anything else).

This restriction leads to unnecessarily complicated examples (Gosling 1998, page 327):

```
// We have imported java.io.* and java.util.*
public String[] ls(String dir, String opts)
....
```

## 4.1 Objects in Java

We define a duck to have two feet. Ducks calculate how many legs they have by counting their feet.

```
class Duck
{
    int feet = 2;
    public int legs() { return feet; }
}
```

A lame duck is a duck with only one foot. Since lame ducks are ducks, we define `LameDuck` as a subclass of `Duck`.

```
class LameDuck
{
    int feet = 1;
    public int legs() { return feet; }
}
```

Since lame ducks are ducks, we can assign a lame duck object to a duck variable:

```
Duck d = new LameDuck();
```

The object `d` is a duck with one leg and two feet!

Since all ducks are the same, and all lame ducks are the same, it makes sense to declare `feet` and `legs` as `static`. If we do this, the object `d` has two legs and two feet.

Java designers explain (Gosling 1998, page 69):

You've already seen that method overriding enables you to extend existing code by reusing it with objects of expanded, specialized functionality not foreseen by the inventor of the original code. But, where fields are concerned, it is hard to think of cases in which hiding them is a useful feature.

However (Gosling 1998, page 70):

Hiding fields is allowed in Java because implementors of existing superclasses must be free to add new `public` or `protected` fields without breaking subclasses.

This is a weak and spurious reason because there are a number of ways of breaking subclasses in Java. For example, making any method `final` breaks all subclasses that override that method.

However (Gosling 1998, page 70):

Purists might well argue that classes should have only `private` data, but Java lets you decide on your style.

If Java had required all fields to be `private`, introducing a new field would not break a subclass. Thus Java prefers efficiency to clarity and chastises programmers who prefer clarity by calling them “purists”.

Let us allow ducks to lay eggs. Both `Duck` and `LameDuck` have inner classes that allow them to lay eggs.

Inner classes are contained within their outer classes and are always `private`. Fields and methods of inner classes are hidden and do not override anything hidden by another class.

```
// Hatch a duck's egg
new Duck(). new Egg(). hatch()

// Hatch a lame duck's egg
new LameDuck. new Egg(). hatch()

// Hatch a duck's egg
((Duck) new LameDuck()). new Egg(). hatch()
```

In the last case, the ducklings will be confused because they have different numbers of legs and feet: we have a duck's egg, but the object referring to it is a lame duck, so the duck's method `legs` is overridden. The following example may clarify this:

```
class Duck
{
    int legs() { return 2; }
    int check() { return legs(); }
}

class LameDuck extends Duck
{
    int legs() { return 1; }
}

((Duck) new LameDuck()).check() ==> 1
```

Java provides two mechanisms to prevent overriding:

- ◇ declaring a method `private` makes it invisible to the subclass
- ◇ declaring a method `final` allows it to be used but no overridden in the subclass

```

class Duck
{
    int legs() { return 2; }
    class Egg
    {
        int checkEggLegs() { return legs(); }
    }
}

class LameDuck extends Duck
{
    int legs() { return 1; }
}

((Duck) new LameDuck()). new Egg(). checkEggLegs() ==> 1

```

The methods `legs` called in `checkEggLegs` is from `LameDuck` — although no instance of `Egg` has a `legs` method.

Making `legs` `private` in `Duck` leads to `IllegalAccessException` from `Egg` although this function is not actually called.

Here is a clarification of this issue (Gosling 1998, page 52–3):

A nested class can use other members of its enclosing class — including private fields — without qualification because it is part of the enclosing class’s implementation. An inner class can simply name the members of its enclosing object to use them.

But naming a member without qualification can yield an *overridden* member! This does not seem consistent with (Gosling 1998, page 114):

Nesting is needed to make local code robust. If hiding outer variables were not allowed, adding a new field to a class or interface could break existing code that used variables of the same name. Scoping is meant as a protection for the system as a whole rather than support for reusing identifiers.

This raises the question: can ducks hide nested eggs?

The examples are admittedly artificial, but the point is important: if short programs containing only a few lines are confusing, how will we understand large systems?

## 4.2 Method Parameters

Lame ducks are “politically correct” — they consider themselves to be the equal of *any* duck, lame or otherwise. We provide lame ducks with the method

```
public boolean equals (Duck d) { return true; }
```

We need this method because (Gosling 1998, page 73):

. . . . `equals` methods should be overridden if you want to provide a notion of equality different from the default implementation provided by the `Object` class. The default is that any two different objects are not equal . . . .

This works for lame ducks, but not for ducks:

```
LameDuck ld = new LameDuck();
ld.equals(new Duck())    ==>    true

Duck d = new LameDuck();
d.equals(new Duck())    ==>    false
```

In the second example, it is clear that `equals` in `LameDuck` has not overridden the default behaviour of `Object`. Why not?

The `equals` method of `LameDuck` does not override the `equals` method for `Duck` because `Duck` inherits from `Object`, and `equals` in `Object` takes an `Object` as a parameter.

We can correct the problem with `equals` but only at the expense of run-time checking:

```
public boolean equals (Object d)
{
    if (d instanceof Duck)
        return true;
    else
        return super.equals(d);
}
```

We cannot define an `equals` method in such a way that incorrect usage (wrong type parameter) would be an error. This is because `equals` in `Object` expects an `Object` parameter, and any object can be treated as an `Object`. For example, we can compare ducks and matrices. However, we *can* use appropriate code to detect the error at run-time — inefficiently.

### 4.3 Strings in Java

Classes `String` and `StringBuffer` are incomplete in various respects but we cannot complete them by inheritance, because they are `final`.

To delete a character from a `StringBuffer`, we have the choice of constructing a new `StringBuffer` from two substrings taken from the original `StringBuffer`, or creating a new class with a method `deleteChar`.

`x.equals(y)` returns `false` if `x` and `y` are distinct `StringBuffers` that contain the same characters. You can't override `StringBuffer.equals` but this does what you would expect:

```
x.toString().equals(y.toString())
```

Note that neither of the following alternatives works, because `Object.equals` expects an object:

```
x.toString().equals(y)
x.equals(y.toString())
```

The same problem occurs with hash codes. It is natural to create a hash table of `StringBuffers`, but they have to be converted to `Strings` before the hash code is computed — a waste of time.

Class `String` and `StringBuffer` are closely related, and in fact can be implicitly converted to each other, but they do not support a common interface.

There is one situation in which `toString` is called implicitly (in all other situations, the programmer must write it). The expression `s+o`, in which `s` is a `String` and `o` is any object, is automatically converted to `s+o.toString()`. This is inconsistent with other functions which expect `String` parameters and also with other conversion functions such as `toInt`.

An interesting consequence of the implicit conversion is that Java's `+` operator is not associative. If `s` is the `String` "x" then:

```
s + 1 + 2    ==>    "x12"
(s + 1) + 2  ==>    "x12"
s + (1 + 2)  ==>    "x3"
```

Consider:

```
String a = new String("x");
String b = new String("x");
```

and

```
String c = new String("x").intern();
String d = new String("x").intern();
```

After these assignments, the `Strings` `c` and `d` are equal, but no other pair is equal.

The effect of `intern()` (a name derived from LISP) is to return a pointer to a `String` belonging to a pool of `Strings`, putting a new `String` into the pool only if necessary. If you use `intern` *all the time*, then all your strings will be unique. But if you omit `intern` just once, you will have a `String` that is incomparable with other `Strings`. As noted by the designers:

Using equality operators on `String` objects does not work as expected. (Gosling 1998, page 133)

For example, `==` probably works correctly in the following code:

```
String str = "CUSEC";
// ....
if (str == "CUSEC")
    ....
```

But be careful — this trick works only if you are sure that all string references are references to string literals. (Gosling 1998, page 166)

#### 4.4 Class Parameters

Java does not provide parameterized classes (what C++ calls “template” classes). If you want lists of things, you have two choices:

- ◊ write generic code for lists of `Object`s and then put whatever you like into the lists (every class is a subclass of `Object`);
- ◊ write specific code for each type of list you need (list of `Integer`, list of `String`, etc.).

The first approach sacrifices type checking and the second requires duplicated code.

Java’s approach might be called “classical object oriented”. However, most designers of object oriented languages quickly realized that there was a problem here and provided some form of parameterized class (e.g., C++, *Dee*, *Eiffel*, etc.).

Java does have one kind of parameterized class: the array. There is also a class `Vector` which has no special properties and could have been defined by any programmer.

Suppose we have a program written using arrays and we decide to convert to `Vectors`. We have to make the following conversions:

```
b = a[i]    ==>    b = a.elementAt(i).intValue();
a[i] = 4    ==>    a.setElementAt(new Integer(4), i);
```

Compare C++, in which we can overload operator `[]`.

Array initialization is inconsistent. These two sequences are equivalent:

```
int k = 4;

int k;
k = 4;
```

but these two are not:

```
int a[] = { 1, 2, 3 };

int a[];
a = { 1, 2, 3 };
```

because the second one is illegal. Note, however, that we *can* write:

```
int a[];
{
    int aa[] = { 1, 2, 3 };
    a = aa;
}
```

#### 4.5 Types

We note that there is no problem in Java with “questionable” assignments (Gosling 1998, page 121):

Java prevents incompatible assignments by forbidding anything questionable.

This raises the question as to whether the following assignment is “questionable”:

```
Character c = (Character) new Object();
```

Apparently, it isn't, because it compiles just fine but, when we run it, it throws `java.lang.ClassCastException`.

Java has no enumerations. Values that should be enumerated are defined as constant integers (as in C). The following call has no type errors, because the parameters (after the first two) are all integers, but doesn't work because the ordering is wrong:

```
new Event(target, when, 0, 0, Event.F1, Event.SHIFT_MASK, Event.GOT_FOCUS);
```

## 4.6 Confusion

The keyword `final` is confusing (rather like C's `static`):

- ◇ A `final` subclass cannot be subclassed (implying greater efficiency)
- ◇ A `final` method cannot be overridden
- ◇ A `final` field has a constant value but can be hidden
- ◇ A `final` parameter has a constant value (although its fields can be changed if it's an object)

Even the designers are confused (Gosling 1998, page 53):

Therefore, [local inner classes] cannot be private, protected, static, or final, because these modifiers apply only to class members.

But a local inner class *can* be subclassed and any class *can* be `final`. Final fields are constants, and *can* be hidden in a subclass but a `final` class does not have subclasses so its fields *cannot* be hidden.

## 5 Implementing Design Patterns

In this section, we will discuss design patterns. The focus will be less on the design patterns themselves (since this should have been covered in other courses) but on the actual implementation of patterns.

### 5.1 Singleton

The `Singleton` pattern (Gamma, Helm, Johnson, and Vlissides 1995, page 127) is not very interesting in itself, but we describe it briefly here because it is needed for the `State` pattern, described next.

There are situations in which we must ensure that there is *exactly one* instance of an object at run-time. The purpose of the `Singleton` pattern is to ensure this.

Here is the declaration of a simple version of `Singleton` in C++:

```
class Singleton
{
public:
    static Singleton *getInstance();
private:
    Singleton() {}
    static Singleton *instance;
};
```

- ◇ The `public static` function `getInstance` allows clients to obtain a pointer to the unique instance by writing

```
Singleton::getInstance()
```

- ◇ Since the constructor is `private`, it cannot be called by clients.
- ◇ The `private` part of the class contains a `static` pointer to the unique instance of `Singleton`.
- ◇ The function `getInstance` constructs a new instance if necessary otherwise just returns a pointer to the existing instance.

```
Singleton *Singleton::getInstance()
{
    if (instance == NULL)
        instance = new Singleton;
    return instance;
}
```

- ◇ Somewhere in the program, we must initialize the `static` pointer to `NULL`. According to *The Annotated C++ Reference Manual* (Ellis and Stroustrup 1990), “static members of a global class are initialized exactly like global objects and only in file scope”. This means that the following statement cannot be nested inside any structure:

```
Singleton *Singleton::instance = NULL;
```

- ◇ The `public static` function `getInstance` allows clients to obtain a pointer to the unique instance by writing

```
Singleton::getInstance()
```

- ◇ Since the constructor is `protected`, it cannot be called by clients but it can be called by subclasses of `Singleton`.
- ◇ The `private` part of the class contains a `static` pointer to the unique instance of `Singleton`.

The following test prints the address of three `Singleton` objects:

```
void main()
{
    cout << (void *)Singleton::getInstance() << endl;
    cout << (void *)Singleton::getInstance() << endl;
    cout << (void *)Singleton::getInstance() << endl;
}
```

The following output shows that there is indeed only one actual object:

```
0x00300110
0x00300110
0x00300110
```

## 5.2 State

The `State` pattern (Gamma, Helm, Johnson, and Vlissides 1995, page 305) allows an object to behave differently according to its internal state.

It is easy to implement objects with this property:

```
class Changeable
{
public:
    Changeable() { state = STOPPED; }
    void act()
    {
        switch(state)
        {
        case STOPPED:
            // ....
            break;
        case ASCENDING:
            // ....
            break;
        case DESCENDING:
            // ....
            break;
        }
    }
}
```

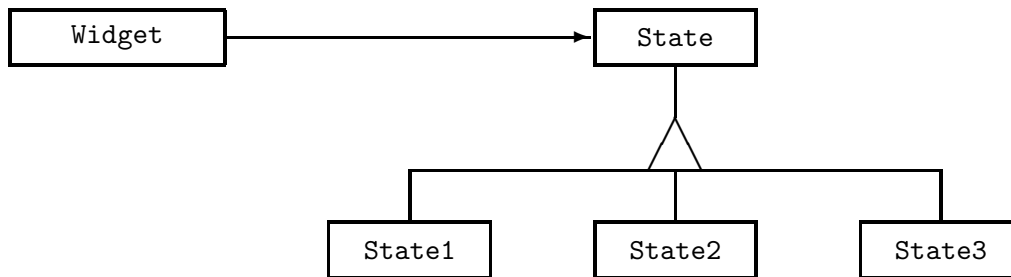


Figure 16: State Pattern

```

private:
    enum State { STOPPED, ASCENDING, DESCENDING } state;
};
  
```

The disadvantage of this implementation is that many of the member functions will need `switch` statements, making maintenance difficult and error-prone.

An alternative, and sometimes better, solution, is to use an inheritance hierarchy in which each derived class corresponds to a particular state. Figure 16 shows the basic idea. The object is represented as an instance of class `Widget` with a pointer to an instance of `State1`, `State2`, or `State3`, depending on the current state of the `Widget`. However, there are some implementation details that have to be worked out.

- ◇ How many objects actually exist at any one time?

To avoid frequent creation and deletion, the `Statei` objects should be permanent, so that state changes are implemented by pointer assignment.

- ◇ Where is the state of the object actually stored?

This depends somewhat on the application. In most cases, there will be state information associated with the object that will have to be stored in `Widget`.

It is also possible that the state information required depends on the state, in which case it might be possible to store state information in the `Statei` objects. However, this seems intuitively less likely.

- ◇ How are state changes effected?

The `Widget` could control its own state. But it is often more natural for that state changes to be initiated by the behaviour functions in the `Statei` objects. This implies that these objects must be able to tell the `Widget` to change its state. Consequently, there will be circular dependencies: `Widget` points to `Statei` and `Statei` points to `Widget`. This does not have to be a bad thing if we can put all of the objects into a single component.

**Example: a Vending Machine** We will use a simple vending machine to illustrate the implementation of the `State` pattern. Figure 17 shows the state transitions of the vending machine. The user actions are:

- ◇ insert a *coin*;
- ◇ *request* an item;
- ◇ ask for *return* of the coins.

	Ready	Receiving	Stuck
Ready	request	coin	
Receiving	return/request	coin	coin
Stuck	return		

Figure 17: Vending Machine Transitions

Initially, the vending machine is in state **Ready**. A *request* has no effect but inserting a coin puts it into state **Receiving**. Further coins maintain this state, but too many coins may lead to state **Stuck**, from which the only recourse is to ask for the coins to be returned. Figure 17 does not show everything: an item request has no effect if insufficient coins have been inserted, and the vending machine does not return change (although this would be trivial to add).

We begin with the class `VendingMachine`.

- ◇ `VendingMachine` stores its own state variables: in this case, the only variable is `total`, which is the amount of money inserted so far.
- ◇ Clients of a vending machine can construct an instance, insert a coin, request an item, or ask for coins to be returned.
- ◇ `VendingMachine` provides functions for accessing and changing the state (`getTotal`, `setTotal`, `addTotal`). These functions will be used by the behaviour functions. They are `public` but could be made `private` if we made the behaviour classes `friends`.
- ◇ `VendingMachine` has a pointer to an instance of `State`. Subclasses of this class will define the behaviour of the vending machine in different states.
- ◇ The behaviour objects can change the state of the vending machine by passing a pointer to the new state to the function `change`.

```
class VendingMachine
{
public:
    // Client interface
    VendingMachine() { total = 0; }
    void insertCoin(int amount) { state->insertCoin(amount); }
    void requestItem(int item) { state->requestItem(item); }
    void returnCoins() { state->returnCoins(); }

    // private interface
    int getTotal() { return total; }
    void setTotal(int tot) { total = tot; }
    void addTotal(int amount) { total += amount; }
    void change(State *newState) { state = newState; }

    // debugging aid
    void displayState()
    {
        cout <<
            "State = " << state->displayState() <<
```

```

        " Total = " << total << endl;
    }
private:
    State* state;
    int total;
};

```

Class `State` is an abstract base class.

- ◊ Since all behaviours must know about the vending machine, it contains a **protected** pointer to the vending machine.
- ◊ The interface provides the client functions `insertCoin`, etc.
- ◊ There is a function to change states.
- ◊ There is a function to associate a vending machine to a behaviour function. This is needed because it is difficult to pass an argument to the constructor of a behaviour object — see below.

```

class State
{
public:
    virtual void insertCoin(int amount) = 0;
    virtual void requestItem(int item) = 0;
    virtual void returnCoins() = 0;
    virtual char *displayState() = 0;
    void change(State *s);
    void setvm(VendingMachine *vm);
protected:
    VendingMachine *owner;
};

```

The implementation of class `State` is fairly obvious.

```

void State::setvm(VendingMachine *vm)
{
    owner = vm;
}

void State::change(State *s)
{
    owner->change(s);
}

```

We are now ready to define the behaviour classes. In this example, these classes have no state of their own (because the vending machine itself stores all of the state information). It is wasteful to create behaviour objects for each vending machine because all vending machines can share a single set of behaviour classes. Consequently, we define behaviour classes as **Singleton** classes.

Each behaviour class:

- ◇ must implement the client functions
- ◇ has a `static` function that returns the unique instance
- ◇ has a function that displays the name of the state (for debugging)
- ◇ has a function `getInstance` returns a pointer to the unique instance

```
class Ready : public State
{
public:
    void insertCoin(int amount);
    void requestItem(int item) { }
    void returnCoins();
    char *displayState() { return "Ready"; }
    static Ready *getInstance();
protected:
    Ready() {}
private:
    static Ready* instance;
};
```

Implementation of class `Ready` is straightforward.

- ◇ Inserting a coin initializes the amount and changes the state to `Receiving`.
- ◇ Requesting an item has no effect (alternative: report an error)
- ◇ Returning coins sets the total amount to zero

```
void Ready::insertCoin(int amount)
{
    owner->setTotal(amount);
    change(Receiving::getInstance());
}

void Ready::returnCoins()
{
    cout << owner->getTotal() << "returned" << endl;
    owner->setTotal(0);
}

Ready *Ready::getInstance()
{
    if (instance == NULL)
        instance = new Ready;
    return instance;
}

Ready *Ready::instance = NULL;
```

The other behaviour classes are very similar. Class `Receiving`:

- ◇ Accepts coins, but goes into state `Stuck` if the total amount exceeds 200
- ◇ Accepts requests for items, but only if the total amount is not less than the cost of the item
- ◇ allows coins to be returned

```

class Receiving : public State
{
public:
    void insertCoin(int amount);
    void requestItem(int item);
    void returnCoins();
    char *displayState() { return "Receiving"; }
    static Receiving *getInstance();
protected:
    Receiving() {}
private:
    static Receiving* instance;
};

void Receiving::insertCoin(int amount)
{
    owner->addTotal(amount);
    if (owner->getTotal() > 200)
        change(Stuck::getInstance());
}

void Receiving::requestItem(int item)
{
    int costs[] = { 75, 125, 75, 150 };
    if (
        item >= 0 &&
        item < 4 &&
        owner->getTotal() >= costs[item] )
    {
        cout << item << " delivered" << endl;
        owner->setTotal(0);
        change(Ready::getInstance());
    }
    else
        cout << "Request refused" << endl;
}

void Receiving::returnCoins()
{
    cout << owner->getTotal() << "returned" << endl;
    owner->setTotal(0);
    change(Ready::getInstance());
}

```

```

Receiving *Receiving::getInstance()
{
    if (instance == NULL)
        instance = new Receiving;
    return instance;
}

Receiving *Receiving::instance = NULL;

```

Class `Stuck` does not respond to inserted coins or requests, but if you ask for your coins back, it gives them to you and goes into state `Ready`.

```

class Stuck : public State
{
public:
    void insertCoin(int amount) {}
    void requestItem(int item) {}
    void returnCoins();
    char *displayState() { return "Stuck"; }
    static Stuck *getInstance();
protected:
    Stuck() {}
private:
    static Stuck* instance;
};

void Stuck::returnCoins()
{
    cout << owner->getTotal() << "returned" << endl;
    owner->setTotal(0);
    change(Ready::getInstance());
}

Stuck *Stuck::getInstance()
{
    if (instance == NULL)
        instance = new Stuck;
    return instance;
}

Stuck *Stuck::instance = NULL;

```

The following test of the vending machine allows the user to make requests via the keyboard. It displays the state of the vending machine at the beginning of each cycle.

```

void main()
{

```

```

VendingMachine *pvm = new VendingMachine;
Ready::getInstance()->setvm(pvm);
Receiving::getInstance()->setvm(pvm);
Stuck::getInstance()->setvm(pvm);
pvm->change(Ready::getInstance());

bool vending = true;
while(vending)
{
    pvm->displayState();
    char c;
    int i;
    cout << "Coin, Item, Return, Quit? ";
    cin >> c;
    switch(c)
    {
        case 'c':
            cout << "Amount? ";
            cin >> i;
            pvm->insertCoin(i);
            break;
        case 'i':
            cout << "Item (0-3)? ";
            cin >> i;
            pvm->requestItem(i);
            break;
        case 'r':
            pvm->returnCoins();
            break;
        case 'q':
            vending = false;
            break;
    }
}
}

```

Here is a dialog with the vending machine:

```

State = Ready  Total = 0
Coin, Item, Return, Quit? c
Amount? 50
State = Receiving  Total = 50
Coin, Item, Return, Quit? c
Amount? 25
State = Receiving  Total = 75
Coin, Item, Return, Quit? i
Item (0-3)? 1
Request refused
State = Receiving  Total = 75

```

```

Coin, Item, Return, Quit? c
Amount? 50
State = Receiving Total = 125
Coin, Item, Return, Quit? i
Item (0-3)? 1
1 delivered
State = Ready Total = 0
Coin, Item, Return, Quit? q

```

17

### 5.3 Visitor

A “visitor” is an object that is passed to each node of a data structure in turn. The visitor performs a particular computation at the node. We define a simple problem and then show three ways of solving it; the third solution is the **Visitor** pattern.

Suppose we are designing a symbolic algebra system. A data structure represents an algebraic expression with constants and variables at the leaves and operators at the internal nodes. The expression that we will use as a running example is  $x + 2 \times y$ . Its representation consists of five nodes: a **Sum** node, a **Prod** node, two **Var** nodes, and a **Const** node.

There are two operations that we would like to apply to an expression: evaluate it and print it. In the example, we will simplify the evaluation by assuming that all variables are zero.

#### 5.3.1 Recursive Functions

Consider first how we would have solved the problem before the days of object oriented programming. We would first define a **struct** capable of storing a node of any type. The **struct** consists of a *tag*, called **kind** here, and a **union** for the various different kinds of data. Note that **Sum** and **Prod** use the same **kind**: a **struct** with pointers to the left and right subtrees.

```

struct Node
{
    enum { CONST, VAR, SUM, PROD } kind;
    union
    {
        int intValue;
        char *varName;
        struct
        {
            Node *leftNode;
            Node *rightNode;
        };
    };
};

```

The next step is to write constructor functions for each kind of node.

```

Node *makeConst(int i)
{
    Node *n = new Node;
    n->kind = Node::CONST;
    n->intValue = i;
    return n;
}

Node *makeVar(char *name)
{
    Node *n = new Node;
    n->kind = Node::VAR;
    n->varName = name;
    return n;
}

Node *makeSum(Node *l, Node *r)
{
    Node *n = new Node;
    n->kind = Node::SUM;
    n->leftNode = l;
    n->rightNode = r;
    return n;
}

Node *makeProd(Node *l, Node *r)
{
    Node *n = new Node;
    n->kind = Node::PROD;
    n->leftNode = l;
    n->rightNode = r;
    return n;
}

```

The evaluator is a function that uses `switch` to distinguish node types.

```

int eval(Node *p)
{
    switch(p->kind)
    {
        case Node::CONST:
            return p->intValue;
            break;
        case Node::VAR:
            return 0;
            break;
        case Node::SUM:
            return eval(p->leftNode) + eval(p->rightNode);
            break;
    }
}

```

```

        case Node::PROD:
            return eval(p->leftNode) * eval(p->rightNode);
            break;
    }
}

```

The printer is a very similar function that also uses a `switch` statement to distinguish node types.

```

void print(Node *p)
{
    switch(p->kind)
    {
        case Node::CONST:
            cout << p->intValue;
            break;
        case Node::VAR:
            cout << p->varName;
            break;
        case Node::SUM:
            print(p->leftNode);
            cout << " + ";
            print(p->rightNode);
            break;
        case Node::PROD:
            print(p->leftNode);
            cout << " * ";
            print(p->rightNode);
            break;
    }
}

```

Here is a simple program that demonstrates the functions in use. It constructs the expression  $x + 2 \times y$ , prints it, and evaluates it.

```

void main()
{
    Node *pp =
        makeSum(
            makeVar("x"),
            makeProd(
                makeConst(2),
                makeVar("y") ) );
    print(pp);
    cout << endl << eval(pp) << endl;
}

```

The disadvantage of this solution is that all of the functions contain `switch` statements. If the number of node types gets large, these functions will be difficult to understand and maintain.

With this organization, it is hard to analyze the behaviour of a particular node type. The code for products, for example, is half in `eval` and half in `print`. As the number of operations grows, the code for each node type becomes increasingly scattered and fragmented.

### 5.3.2 An Inheritance Hierarchy

Inheritance enables us to “invert” the structure of the solution. We declare an abstract base class `Node` and a derived class for each kind of node. The base class has `virtual void` functions for each operation (`eval` and `Print`). The derived classes provide implementations for each of these functions.

```
class Node
{
public:
    virtual int eval() = 0;
    virtual void print() = 0;
};

class Const : public Node
{
public:
    Const(int i) : iv(i) {}
    int eval() { return iv; }
    void print() { cout << iv; }
private:
    int iv;
};

class Var : public Node
{
public:
    Var(char *s) : name(s) {}
    int eval() { return 0; }
    void print() { cout << name; }
private:
    char *name;
};

class Sum : public Node
{
public:
    Sum(Node *lhs, Node *rhs) : lhs(lhs), rhs(rhs) {}
    int eval() { return lhs->eval() + rhs->eval(); }
    void print();
private:
    Node *lhs;
    Node *rhs;
};
```

```

void Sum::print()
{
    lhs->print();
    cout << " + ";
    rhs->print();
}

class Prod : public Node
{
public:
    Prod(Node *lhs, Node *rhs) : lhs(lhs), rhs(rhs) {}
    int eval() { return lhs->eval() * rhs->eval(); }
    void print();
private:
    Node *lhs;
    Node *rhs;
};

void Prod::print()
{
    lhs->print();
    cout << " * ";
    rhs->print();
}

```

Here is a test program. Note that it is quite similar to the previous test, except that `eval(pp)` becomes `p->eval()`.

```

void main()
{
    Node *p = new Sum(
        new Var("x"),
        new Prod(
            new Const(2),
            new Var("y")));
    p->print();
    cout << endl << p->eval() << endl;
}

```

The inheritance hierarchy solution brings together the code for each kind of node. It is now easy to analyze the behaviour of, say, a `Var` node. On the other hand, the code for an operation, such as printing, is divided between the subclasses: it is scattered and fragmented. The recursive function solution and the inheritance hierarchy solution are complementary: the strengths of one are the weaknesses of the other.

### 5.3.3 The Visitor Solution

The **Visitor** pattern is intended to provide the advantage of the recursive function solution without the loose structuring and insecurity of that solution. It does this in the following way:

- ◇ Nodes of the structure are derived classes, as in the inheritance hierarchy solution. Nodes contain their own data.
- ◇ Each node type must be able to accept a “visiting” object (or “visitor”). It does this by implementing a function

```
void accept(Visitor &v);
```

- ◇ The **Visitor** object must be able to process a node of any kind. For our example, a **Visitor** would have functions such as `visitConst`, `visitVar`, and so on. The object being visited is passed to these functions. Consequently, we expect to see code like this:

```
class Var : public Node
{
public:
    void accept(Visitor &v) { v.visitVar(this); }
};
```

There is usually more than one kind of visitor. For our problem, there is a visitor that evaluates and another visitor that prints. Thus **Visitor** is actually the abstract base class of an inheritance hierarchy; the derived classes include **Evaluator** and **Printer**.

This scheme has some problems<sup>3</sup>

- ◇ The visitor function (e.g., `visitVar`) receives a pointer to a particular node. It will probably need to access data in the node to do its work. It can access data if the node provides accessor functions for all of its data (which weakens encapsulation) or the **Visitor** class is a **friend** of the **Node** class (which is the solution adopted here).
- ◇ There is easy way to communicate between visitor calls. In our example, it is difficult to evaluation work because values must be accumulated during the traversal, but there is no obvious place where the accumulation can take place.
- ◇ Each kind of visitor must know all of the different kinds of node. When we add a new kind of node, we must modify every class in the **Visitor** hierarchy.

Let’s overlook these difficulties for the moment and see what the solution looks like. We need some forward declarations to help the compiler.

```
class Node;
class Const;
class Var;
class Sum;
class Prod;
```

Then we can define the abstract base class of the **Visitor** hierarchy. Note that the argument passed to a `visit...` function is *not* a **Node** \*. These functions will be called by objects of particular kinds that pass their `this` pointers as arguments to the visitor; thus we know the exact type of the argument.

---

<sup>3</sup>Or so it seems to me — PG.

```

class Visitor
{
public:
    virtual void visitConst(Const *p) = 0;
    virtual void visitVar(Var *p) = 0;
    virtual void visitSum(Sum *p) = 0;
    virtual void visitProd(Prod *p) = 0;
};

```

Next, we can proceed to define the node classes.

- ◇ These are much the same as before, except that the `eval` and `print` functions are replaced by the single function `accept`.
- ◇ Note the `friend` declarations.
- ◇ In the `Sum` and `Prod` classes, we have to choose a sequence of calls in the visitor function. We have used infix (left subtree, current node, right subtree) here. This sequence is imposed on *all* visitors, which is a severe restriction. For our problem, we would like infix order for printing and postfix order for evaluation.

```

class Node
{
public:
    virtual void accept(Visitor &v) = 0;
};

class Const : public Node
{
    friend class Evaluator;
    friend class Printer;
public:
    Const(int i) : iv(i) {}
    void accept(Visitor &v) { v.visitConst(this); }
private:
    int iv;
};

class Var : public Node
{
    friend class Evaluator;
    friend class Printer;
public:
    Var(char *s) : name(s) {}
    void accept(Visitor &v) { v.visitVar(this); }
private:
    char *name;
};

class Sum : public Node
{

```

```

        friend class Evaluator;
        friend class Printer;
public:
    Sum(Node *lhs, Node *rhs) : lhs(lhs), rhs(rhs) {}
    void accept(Visitor &v);
private:
    Node *lhs;
    Node *rhs;
};

void Sum::accept(Visitor &v)
{
    lhs->accept(v);
    v.visitSum(this);
    rhs->accept(v);
}

class Prod : public Node
{
    friend class Evaluator;
    friend class Printer;
public:
    Prod(Node *lhs, Node *rhs) : lhs(lhs), rhs(rhs) {}
    void accept(Visitor &v);
private:
    Node *lhs;
    Node *rhs;
};

void Prod::accept(Visitor &v)
{
    lhs->accept(v);
    v.visitProd(this);
    rhs->accept(v);
}

```

Here are the visitors. The first is `Evaluator` which is supposed to find the value of the expression. Unfortunately, I cannot make it work properly so, in this version, the visitors simply print a message.

```

class Evaluator : public Visitor
{
public:
    void visitConst(Const *p) { cout << "Eval Const" << endl; }
    void visitVar(Var *p) { cout << "Eval Var" << endl; }
    void visitSum(Sum *p) { cout << "Eval Sum" << endl; }
    void visitProd(Prod *p) { cout << "Eval Prod" << endl; }
};

```

The other visitor is `Printer` which does work correctly, because printing depends entirely on side-effects and does not require communication.

```
class Printer : public Visitor
{
public:
    void visitConst(Const *p) { cout << p->iv;; }
    void visitVar(Var *p) { cout << p->name;; }
    void visitSum(Sum *p) { cout << " + "; }
    void visitProd(Prod *p) { cout << " * "; }
};
```

Finally, here is a little test to show that it works.

```
void main()
{
    Node *p = new Sum(
        new Var("x"),
        new Prod(
            new Const(2),
            new Var("y")));
    Evaluator e;
    p->accept(e);
    Printer k;
    p->accept(k);
    cout << endl;
}
```

The output produced by the test looks like this:

```
Eval Var
Eval Sum
Eval Const
Eval Prod
Eval Var
x + 2 * y
```

Figure 18 summarizes the three approaches.

Method	Add nodes	Add functions	Switches	Communicate
Recursive functions	hard	easy	yes	yes
Inheritance hierarchy	easy	hard	no	yes
Visitor pattern	hard	easy	no	no??

Figure 18: Comparison of traversal techniques

## 5.4 Strategy

The Strategy pattern hides a number of related algorithms behind a simple interface. Clients should be able to invoke a particular member function of an object without having to know which “strategy” the object is using. It should be possible to change strategies dynamically.

The example has a class called `Integrator` that can integrate functions using various algorithms. Before introducing it, we look at another useful “pattern” — the functional object. The abstract base class `Function` has a pure virtual function `f` that takes `doubles` to `doubles`.

```
class Function
{
public:
    virtual double f(double) = 0;
};
```

Here are two derived classes: the first provides the constant function,  $f(x) = 1$ , and the second provides the sine function,  $f(x) = \sin x$ .

```
class Unit : public Function
{
public:
    double f(double x) { return 1; }
};

class Sin : public Function
{
public:
    double f(double x) { return sin(x); }
};
```

The algorithms for integration are stored in another class hierarchy. The abstract base class is `Method` and it provides one pure virtual function `integrate`.

```
class Method
{
public:
    virtual double integrate(Function *fun,
        double lower, double upper, int steps) = 0;
};
```

Here are two subclasses of `Method`: class `Rectangular` uses the simplest integration strategy, approximating the function as a series of rectangles; and class `Simpson` uses the well-known approximation by Simpson.

```
class Rectangular : public Method
{
public:
    double integrate(Function *fun, double lower, double upper, int steps);
};
```

```

double Rectangular::integrate(Function *fun,
    double lower, double upper, int steps)
{
    double integral = 0;
    double gap = (upper - lower) / steps;
    for (int i = 0; i < steps; i++)
        integral += fun->f(lower + i * gap);
    return gap * integral;
}

class Simpson : public Method
{
public:
    double integrate(Function *fun, double lower, double upper, int steps);
};

double Simpson::integrate(Function *fun, double lower, double upper, int steps)
{
    if (steps % 2 != 0)
        steps++;
    // steps is even
    double gap = (upper - lower) / steps;
    double ends = fun->f(lower) + fun->f(upper);
    double odds = 0;
    for (int i = 1; i < steps; i += 2)
        odds += fun->f(lower + i * gap);
    double evens = 0;
    for (i = 2; i < steps; i += 2)
        evens += fun->f(lower + i * gap);
    return (gap / 3) * (ends + 4 * odds + 2 * evens);
}

```

When we construct an instance of class `Integrator`, we provide it with a pointer to an instance of the `Method` hierarchy. When the client calls `Integrator::integrate`, this pointer determines the method used. We have also provided a function `setMethod` so that the client can choose another strategy.

```

class Integrator
{
public:
    Integrator(Method *pm) : pm(pm) {}
    void setMethod(Method *pNew) { pm = pNew; }
    double integrate(Function *fun, double lower, double upper, int steps)
    {
        return pm->integrate(fun, lower, upper, steps);
    }
private:
    Method *pm;
}

```

```
};
```

Here is a small test in which one instance of `Integrator` uses two methods to evaluate an integral.

```
const double PI = 4.0 * atan(1);

void main()
{
    Integrator grate(new Rectangular);
    cout << grate.integrate(new Sin, 0, PI, 100) << endl;

    grate.setMethod(new Simpson);
    cout << grate.integrate(new Sin, 0, PI, 100) << endl;
}
```

Results:

```
1.99984
2
```

## 5.5 Composite

The Composite pattern is used when a class hierarchy contains both simple objects and multiple objects, and we want to hide the distinction from clients.

The example is a small generalization of the symbolic algebra example that we used to demonstrate the Visitor pattern in Section 5.3. We replace the `Sum` node, which had pointers to left and right subtrees, with a more general `Sum` node that has a list of terms. The interface of class `Node` hides the fact that there are simple nodes (`Constant` and `Var`) and composite nodes (`Sum`).

To keep things simple, we will discuss only one function: `print`. The example demonstrates another useful technique, namely, the use of dynamic binding of stream operators. If `pn` is a pointer to a node of any kind, we can print the node simply by writing:

```
cout << pn;
```

The base class `Node` overloads `operator<<` and also provides a pure virtual function `print`. The implementation of `operator<<` calls `Print` using a pointer to a node, ensuring that the appropriate version will be used, depending on the kind of node.

```
class Node
{
    friend ostream & operator<<(ostream & os, Node *pn);
public:
    virtual void print(ostream & os) = 0;
};

ostream & operator<<(ostream & os, Node *pn)
```

```

    {
        pn->print(os);
        return os;
    }

```

As before, we define subclasses of nodes for constants and variable names. Each one has its own implementation of `print`.

```

class Constant : public Node
{
public:
    Constant(int value) : value(value) {}
    void print(ostream & os) { os << value; }
private:
    int value;
};

class Var : public Node
{
public:
    Var(char *name) : name(name) {}
    void print(ostream & os) { os << name; }
private:
    char *name;
};

```

A sum is represented as a list of terms. We use instances of class `Link` to build the lists.

```

class Link
{
public:
    Link(Node *pNode, Link *pNext)
        : pNode(pNode), pNext(pNext) {}
    Node *getNode() { return pNode; }
    Link *getNext() { return pNext; }
private:
    Node *pNode;
    Link *pNext;
};

```

The `Sum` node has a constructor that creates an empty list and a function `addTerm` to add terms to the sum. Its `print` function walks down the list, printing each term in turn.

```

class Sum : public Node
{
public:
    Sum() { pTerms = NULL; }
    void addTerm(Node *pn) { pTerms = new Link(pn, pTerms); }
    void print(ostream & os);
};

```

```

private:
    Link *pTerms;
};

void Sum::print(ostream & os)
{
    Link *p = pTerms;
    while (p)
    {
        p->getNode()->print(os);
        p = p->getNext();
        if (p)
            os << " + ";
    }
}

```

The following example shows that simple and composite nodes behave in similar ways with respect to the function `print`. We could add other functions that also maintain the illusion that all nodes are similar.

```

void main()
{
    Sum *ps1 = new Sum;
    ps1->addTerm(new Var("x"));
    ps1->addTerm(new Constant(17));

    Sum *ps2 = new Sum;
    ps2->addTerm(new Var("h"));
    ps2->addTerm(new Constant(23));

    Sum *ps3 = new Sum;
    ps3->addTerm(ps1);
    ps3->addTerm(ps2);
    cout << ps3 << endl;
}

```

The example prints:

```
23 + h + 17 + x
```

## 5.6 Designing with Patterns

In this section, we describe the development of a simple accounting system using design patterns.

- ◇ The accounting system is real — actual, working, used software
- ◇ Patterns were not actually used in the design and are introduced here with hindsight

- ◇ Some features of the design may seem rather low level — although most developers would use existing code for these features, someone has to write the code

### 5.6.1 Application Overview

Input to the program consists of small business transactions: payments by cheque, VISA, or cash; deposits into the bank; and adjustments. Output from the program consists of a *transaction record* showing each transaction ordered by date and a *general ledger* showing activity in each account.

The accounting system uses *double entry bookkeeping*<sup>4</sup> This means that each entry into the system is stored once as a credit (positive entry into an account) and once as a debit (negative entry into an account). For manual bookkeeping, the total of all accounts must always be zero, which provides a useful quick check of correctness.

For example, suppose the business buys toner for the laser printer for \$100. The two entries are: debit account **Bank** by \$100 and credit account **Office Supplies** by \$100. What actually happens is slightly more complicated but follows the zero-sum principle. The cost of the toner, including tax, is \$115.03. Figure 19 shows how the entries are made.

Bank	−115.03
Office Supplies	100.00
Federal Tax	7.00
Provincial Tax	8.03
Total	0.00

Figure 19: Generalized double-entry

The user can enter transactions at any time and may also print reports at any time.

### 5.6.2 Design Overview

Goals of the design included:

- ◇ simplicity (important to ensure accounting integrity)
- ◇ uniformity of user interface
- ◇ use of object oriented techniques
- ◇ event-oriented architecture

The architecture was based on the following idea: the main program would consist essentially of a loop like this:

---

<sup>4</sup>The first generally accepted evidence for the application of double-entry bookkeeping derives from the communal account books of the City of Genoa in the year 1340; it is here stated, though, that the books had been kept *ad modum banchi* so that an older use of the method among private merchants may be assumed. But only during the 1380s did this method grow in popularity among Italian merchants. The earliest evidence of a journal, not a necessary but a rather important auxiliary book within the system of double-entry bookkeeping, in which the entries are formulated for assignment of the amounts to the specific accounts, dates from the 1390s. In 1383 Datini changes the bookkeeping of almost all Fondaci to the new system; but even at the end of the 14th century it had been introduced by the Bank of del Maino in Milan for a short period only. See <http://www.franzarlinghaus.de/Bookkeeping.htm>.

```

void display()           display the form and its fields
void clear()            set all fields to default values
Reply doKey(int key)    respond to a keystroke
bool process()          process the form and indicate success or failure
void setField(int field) select the given field

```

Figure 20: Functions of a Form

Form	Used to
Menu	select Forms; each Field is a Form
Parameter	set system parameters such as tax rates
Cheque	record a payment by cheque
Deposit	record a deposit into the bank
Adjustment	record a single double-entry transaction
MultiForm	record VISA and Petty Cash payments
Delete	delete a transaction
Process	generate reports etc.
Test	test form concepts during development

Figure 21: Various kinds of Form

```

while (true)
{
    char key = keyboard->getKey();
    currentForm->doKey(key);
}

```

A *Form* appears to the user as a collection of labelled items on the screen. The user can freely move the cursor around, updating and editing fields in any order. Nothing much happens until the user enters Y in a field labelled `Accept?` and then presses ENTER. At this point, the program validates the fields of the form and performs any necessary processing.

Internally, a *Form* is a set of `Fields`. Each `Field` object has screen coordinates, a title, and a place in which to store data. For example, a `Field` that stores an amount of money might have title "Net amount" and an `int` variable to store the amount.<sup>5</sup> Figure 20 shows the functions provided by a `Form` and Figure 21 shows various kinds of `Form`.

This suggests the *Strategy* pattern (one interface, many behaviours) and perhaps the *Composite* pattern (some forms — VISA, petty cash — are actually multiple forms).

As mentioned previously, each `Form` has a number of `Fields`. Figure 22 shows the functions provided by a `Field` and `fig-field-kind` shows various kinds of `Field`.

Once again, the common interface and varied behaviour suggest the *Strategy* pattern.

Only one instance of each `Form` is needed by the program. Each subclass of `Form` could have been implemented as a `Singleton` pattern, although in fact this was not considered necessary because the `Forms` needed are constructed once in the main program.

<sup>5</sup>Storing amounts of money as cents in an `int` variable works fine as long as the amounts are less than about \$21,474,836, which is likely to be true for a small business. Integer arithmetic also has the advantage of avoiding rounding errors, which are anathema to accountants.

```

void display()      display the current title and text
void select()      highlight the title and text
void deselect()    lowlight the title and text
Reply doKey(int key) process the given keystroke
void clear()       clear the text or set to default
bool validate()   validate the field
char *getText()   return current text
void setText(char *) set new text
bool isDef()      true if text present and validated

```

Figure 22: Functions of a Field

Field	Contents
Fixed	text that the user cannot alter
YesNo	one of {'Y'   'y'   'N'   'n'}
Numeric	digits only
Tax	percentage (e.g., 7.00)
Year	4-digit year
Money	digit { digit } [ '.' { digit } ]
Account	a 3- or 4-letter account code
Date	entered as 25 3 2 displayed as 25 Mar 2002
Accept	no text: entering ENTER indicates selection

Figure 23: Various kinds of Field

The `Form` and `Field` classes provide for data entry. What happens to the data?

When a `Form` has been accepted by the user, the program constructs a set of items corresponding to the transaction. The transaction of Figure 19, for example, shows that four items are needed to record the purchase of toner. These items could be written to a database but, for simplicity, are written to a file. For security reasons, the file is normally closed; when a transaction is accepted, the file is opened in append mode, the new items are appended to it, and the file is closed. This reduces the possibility of data loss, but does not eliminate it.

The file format is designed so that it can easily be read by people and programs. All fields other than the last field (the description of the item) have fixed width and fields are separated by blanks. Numeric fields, other than amounts of money, have leading zeroes to permit sorting. Here is a short excerpt from a transaction file. Note that the “double entry” is not obvious: only the last pair of items actually match (and were included in the example because they did).

```

gst  1997 07 22 01682 00003 00000 00038 00069 X 1      112 Chapters
bks  1997 07 22 01682 00003 00000 00038 00069 D 3      1600 Chapters
gst  1997 07 22 01682 00004 00000 00038 00070 X 1      173 Pneus St Pierre
qst  1997 07 22 01682 00004 00000 00038 00070 X 2      198 Pneus St Pierre
car  1997 07 22 01682 00004 00000 00038 00070 D 3      2472 Pneus St Pierre
reim 1997 07 22 01682 00004 00000 00038 00071 D 4       948 Car reimbursement
etc  1997 07 22 01682 00005 00000 00038 00072 D 3      14825 Transfer of balance
ca   1997 09 03 01684 00000 00000 00039 00073 D 4      -28680 VISA payment

```

```
rfy  1998 10 01 01725 00000 00000 00125 00217 D 4      18008 GST Refund
gst  1998 10 01 01725 00000 00000 00125 00218 D 4      -18008 GST Refund
```

### 5.6.3 Design Details

The foregoing description of the design makes it look fairly “clean” but there are some problems in practice.

There are some functions that apply to one kind of `Field` only. For example, there is a function `getAmount` which returns the amount of money in a `Money` `Field`. In principle, we should define this function only in class `Money` (which is derived from `Field`). In practice, this doesn’t work because there are places in the program where we need to write

```
Field *p;
....
p = .... // pointer to a \ff{Money} object
....
p->getAmount();
```

This doesn’t work because `getAmount` is not declared in `Field`. There are two solutions:

- ◇ Define a default virtual function `getAmount` in class `Field`. This was the solution adopted; it’s worse than it sounds because there are actually quite a number of functions like this. The default function reports an error if it is called for a class other than `Money`.
- ◇ Cast `p` to type `Money*`. At the time when the program was written, this was an unsafe solution and was therefore not adopted. C++ now has dynamic cast, which performs a run-time check, and would enable safe code to be written. However, the programmer does not have much choice about the error message.

At the top level, we mentioned the following loop:

```
while true
{
    char key = keyboard->getKey();
    currentForm->doKey(key);
}
```

This is a simplification because a `Menu` is a kind of `Form`. If the user opens a menu and selects an item from it, that item becomes the current `Form`. But when that `Form` has been processed, the display must revert to the `Menu`.

To implement this, the main program actually has a stack of `Forms`, with the current `Form` at the top of the stack. When the user makes a menu selection, the corresponding `Form` is pushed and becomes the new top of stack and current `Form`. When a `Form` has been processed, it is popped and the menu reappears. When the stack becomes empty, the program terminates.

The main program employs a form of the `Chain of Responsibility` pattern: each keystroke is sent to the top `Form` on the stack; this `Form` passes the keystroke on to the current `Field`. In the `Chain of Responsibility` pattern, a request is sent to a list of receivers, and any object on the list may choose to process it.

## 5.7 Assessing Design Quality

Parnas and Weiss (1987) identified eight requirements for good design. A good design should be

- ◇ ***well structured:*** modular, well-defined interfaces, consistent with chosen properties such as information hiding
- ◇ ***simple:*** to the extent of being “as simple as possible, but no simpler”
- ◇ ***efficient:*** providing functionality that can be provided within required time limits with the available resources
- ◇ ***adequate:*** fulfilling stated requirements
- ◇ ***flexible:*** able to accommodate changed requirements, especially the changes that are most likely to occur
- ◇ ***practical:*** interfaces provide neither more nor less than the facilities required by the design
- ◇ ***implementable:*** using resources available to the development team and clients
- ◇ ***standardized:*** using well-defined and familiar notation in all design documentation

Designs should be assessed by reviews.

A ***technical review*** should be performed by experienced designers and developers. It determines whether the design satisfies the eight criteria above. Technical review usually involves “mental execution” of the design to ensure that everything needed is present.

A ***management review*** assesses the resource implications of a design and answers questions such as:

- ◇ do we have the necessary resources to implement this design?
- ◇ will the implementation be completed within the budget?
- ◇ will the implementation be ready when the clients want it?

Although this discussion isolates design from the rest of development, it should not imply the use of the waterfall model. For example, design quality can be assessed with prototypes. A prototype might be a partial implementation of the design or a throw-away prototype of the complete system.

## 6 Varia

### 6.1 Automated Testing

- ◇ Go to [www.autotestco.com](http://www.autotestco.com)
- ◇ click Related Links
- ◇ click Test Life-Cycle Tools

### 6.2 Assertions increase test effectiveness

Assertions can be used to increase the value of testing in situations where *error masking* is likely (Jeffrey 1997).

Detecting an error requires three conditions to be met:

1. An input must cause a defect to be *executed*
2. The defect must cause the state to become corrupted, resulting in a *data state error*
3. The data state error must *propagate*, leading to erroneous output

If the first condition is unlikely to occur, assertions won't help much. But, if the second or third conditions are unlikely, assertions may help. In OO programs, defensive programming techniques tend to make these conditions unlikely. "OO systems provide clear reuse benefits but, for the same reasons, may allow catastrophic faults to remain hidden for longer."

Jeff Payne at Reliable Software Technologies (where Jeff Voas works) wrote two versions of an ATM machine, one in C and the other in C++. The C++ version made use of OO techniques: information hiding, abstract data types, inheritance, and polymorphism. It also showed, when tested with 102 system-level tests, slightly worse at propagating errors than the C version.

Analysis of the C++ code revealed 8 places where the code appeared to be particularly good at error masking: propagation test scores were between 0% and 15%. Adding assertions at these points increased propagation to 100%.

The conclusions of this and other studies are:

- ◇ Assertions are important for testing. An assertion may not increase propagation but it can never decrease it.
- ◇ Information hiding and encapsulation are detrimental to error propagation.
- ◇ Abstract data types have little effect on error propagation.
- ◇ Inheritance alone does not impede propagation, but the combination of inheritance and information hiding is "lethal". Unit testing costs increase with the depth of inheritance hierarchies.
- ◇ Polymorphism is difficult to test but tends to lead to larger faults, so lack of error propagation may not be so important.

### 6.3 Pisces Safety Net

Pisces Safety Net (PSN) is an implementation of a software analysis technique called *extended propagation analysis* (EPA) (Voas, Charron, McGraw, Miller, and Friedman 1997). EPA injects artificial faults — both hardware and software — into a system to test its tolerance for unusual events. EPA has two novel features:

- ◊ It is not concerned with correctness. Its only concern is output behaviour: the avoidance of dangerous outcomes.
- ◊ EPA does not require an oracle (that is, a source of correct results). Instead, the user identifies undesirable outcomes and EPA determines whether such outcomes can occur automatically.

The input for EPA is:

- ◊ a program
- ◊ an operational profile
- ◊ a description of safe or unacceptable outcomes

The output generated is a list of places in the source code where it has detected potential weaknesses on the basis of injected faults.

EPA works both by analyzing the source text and by executing the program. The analysis reveals places where failure is possible. EPA then injects faults and runs the program to see whether they cause the expected failure.

For example, suppose it is known that, when a particular input sensor fails, it transmits  $-1$  constantly (this is the kind of information given in the “operational profile” above). The software is supposed to check for this condition and respond appropriately. EPA would:

- ◊ look for places where suitable tests seem to be lacking
- ◊ create tests that set the sensor input to  $-1$  and execute the suspected code
- ◊ check the output for incorrect responses

EPA has a theoretical model.

$P$	=	the program
$x$	=	an input value
$\Delta$	=	the set of possible valid inputs
$Q$	=	the probability distribution of $\Delta$
$\lambda$	=	a program location in $P$
$i$	=	a particular iteration of $\lambda$ caused by input $x$
$A(i, \lambda, P, x)$	=	the state produced after executing $\lambda$ on iteration $i$ with data $x$
$N(x, \lambda)$	=	the number of times that $\lambda$ is executed by $x$

The set of states created by  $x$  is

$$A(\lambda, P, x) = \{ A(i, \lambda, P, x) \mid 0 \leq i \leq N(x, \lambda) \}$$

A *simulated infection* occurs when some data state’s variable that has an initial value or an undefined value is forced to have a new value. EPA employs simulated infections to let the user observe the effect of different classes of corrupted states and to observe whether the

system can recover from them. If the program is robust, it would move into a correct state after executing the infection.

The user defines a predicate `pred` that is true when the program behaves in an undesirable way. The predicate might depend on the value of a particular program variable or one of the program's outputs. EPA employs the following algorithm.

1.  $count = 0$
2. Randomly select input  $x$  using distribution  $Q$
3. Execute  $P$  on  $x$  until the current state is  $A(i, \lambda, P, x)$  or the program halts or times out
4.  $Z = A(i, \lambda, P, x)$
5. Alter the value of a variable  $a \in Z$  creating a new state  $Z'$
6. Execute the following code in state  $Z'$
7. If  $\lambda$  is executed more than once, alter  $a$  in each state following  $\lambda$
8. If `pred`, increment  $count$
9. Repeat steps 2 through 8  $N(x, \lambda)$  times
10.  $\psi = count/N(x, \lambda)$

Then  $1 - \psi$  is the *failure tolerance* of location  $\lambda$ .

In step 7, why is  $a$  altered repeatedly? This is because we are simulating a fault in which a variable is repeatedly corrupted, or a device which is stuck in a bad state.

The desired outcome is  $\psi = 0$  — that is, `pred` never becomes true — giving a failure tolerance of 1, or 100%. In all cases,  $0 \leq \psi \leq 1$ , with  $\psi = 1$  (giving fault tolerance 0) indicating that the code fails for every fault injected — that is, it is very fragile.

## Case Studies

1. A yaw damping controller for a Boeing 737 had a fault tolerance of 0.435. Further tests showed that this factor was almost entirely due to a function `lim()` that was supposed to limit the failure tolerance of the entire program.
2. An autopilot controller for the Boeing 737 had a fault tolerance of 0.95. The detailed results, giving the fault tolerance at different parts of the program, show that fault tolerance is 1 for most statements but that there are a few places where it is less — typically between 0.95 and 0.98.

EPA was then run with an inverted input distribution (a small number of likely inputs and a large number of unlikely inputs). The sections that previously failed more often, as expected. More usefully, the fault tolerance of one part, previously 1, dropped to 0.5.

3. A Magnetic Stereotaxis System is software for performing neurosurgery. It uses computer-controlled magnets to move a small metal seed through the brain along a precisely-defined trajectory. The machine becomes dangerous if the current in the magnets exceeds a certain value.

EPA showed that there were areas in the program where fault injection caused excessive current. These areas were preceded by a conditional statement that checked the current. Further tests showed that the current could be changed *after* the condition had been passed. This was a fairly subtle bug, but EPA detected it automatically.

4. Bay Area Rapid Transit (BART) uses Advanced Automatic Train Control (AATC) software. Head and tail cars of each train are equipped with spread-spectrum radios. Information obtained from the radios is used by a computer to determine safe speeds for the trains; these speeds are transmitted to the trains.

2000 locations in the program were selected for EPA. About 26 of them had fault tolerances less than 1. Of the 26, 14 caused core dumps; this is considered a good response, because the system is designed to cope with total failure of the commuter. Nevertheless, EPA analysis on further locations chosen on the basis of the earlier results enabled the developers to bring all locations to fault tolerance 1.

## 6.4 Do Standards Improve Quality?

This is a summary of a debate between Norman D. Schneidewind (for) and Norman Fenton (against) (Schneidewind and Fenton 1996).

### The case for standards

- ◇ Loral Space Information Systems has developed a process for developing shuttle software based on the *ANSI/AIAA Recommended Practice for Software Reliability*.
- ◇ WWW depends on the *Transmission Control Protocol* (MIL-STD 1778) and the *Internet Protocol* (MIL-STD 1777).
- ◇ LANs depend on the IEEE 802 LAN standards.

Standards can:

- ◇ educate and inform us about software engineering processes and current practices
- ◇ create processes for quality assurance programs

This *informal* use of standards is probably far more important than the traditional use of standards because it leads to cultural change in the workplace. Standards provide the information that people need when they move into a QA group.

By increasing understanding, standards improve quality.

Standards developed by professional organizations such as AIAA, IEEE, and ISO are developed by consensus in working groups. Consequently, diverse points of view are taken into account. IEEE standards must be reaffirmed by ballot every five years.

Standards are needed for portability: e.g., Posix.

Conclusion: customers receive higher quality software products when software standards are used in development.

### The case against standards

Norman Fenton works at the Centre for Software Reliability (CSR). Between 1990 and 1994, CSR developed methods for assessing the effectiveness of software engineering standards. They chose the definition:

A software standard is effective is, when used properly, it improves the quality of the resulting software products cost effectively.

“We found no evidence that existing standards are effective according to this criterion.”

The study included more than 250 standards that fell within the scope of software engineering. (“The nice thing about standards is that you have so many to choose from.”)

There are many standards dealing with safety-critical software. Some are based on formal methods, others totally ignore formal methods. There are many similar examples. This suggests that software engineering is not *sufficiently mature* for standardization.

Is any standard better than no standard?

- ◇ Software standards over-emphasize process and neglect the product. But there is no evidence or guarantee that a “good” process will lead to a good product.
- ◇ Many software standards are not standards. A standard is a *set of mandatory requirements*. A standard must be precise; if it isn’t, it’s a set of “codes” or “guidelines”. Software engineering does not make this distinction: many so-called “standards” are really guidelines.
- ◇ It is impossible to measure conformance to software standards.
- ◇ Many software standards prescribe, recommend, or mandate the use of technology that has not been validated objectively. The standards may therefore be prescribing techniques that are ineffective for high-quality systems.
- ◇ Many software standards are too big. Standards that embrace the entire life-cycle, for example, are typically too large to be manageable.

It is time that the software industry paid for the development of good, timely standards rather than relying on the efforts of individuals who volunteer their efforts to the standards-making organizations. Such contributions may be heroic and unsung, but they are also almost entirely *ad hoc*.

21

## 6.5 N-Version Programming

There is a school of thought that claims that reliability can be achieved by designing and coding many versions of a program and then running them in parallel, using a majority vote when outputs differ (Hatton 1997).

Software built in this way is called *N-channel software*. The channels may be identical, as in the shuttle, or different. Identical channels are presumably protecting against hardware failure, whereas different channels are protecting against software design errors. The Boeing 777 uses single-channel software but the European Airbus uses multi-channel software.

The rationale for *N-channel software* is:

- ◇ the channels are independent
- ◇ failures always lead to disagreement between channels
- ◇ the voting system works correctly

If these conditions are satisfied, the probability that at least two channels out of  $N$  agree is

$$p_{m \geq 2} = 1 - p_{m=1} - p_{m=0}$$

where  $m$  is the number of channels that agree and  $p$  is the probability of failure of any one channel. The probability of system failure is

$$\begin{aligned} 1 - p_{m \geq 2} &= p_{m=0} + p_{m=1} \\ &= p^n + n(1-p)p^{n-1} \end{aligned}$$

Thus the use of  $N$ -channels leads to a factor of improvement with respect to a single channel of

$$\begin{aligned} &\frac{p}{p^n + n(1-p)p^{n-1}} \\ &= \frac{1}{p^{n-1} + n(1-p)p^{n-2}} \end{aligned}$$

Here are some values of this function:

$n$	$p = 0.1$	$p = 0.01$	$p = 0.001$
3	3.57	33.56	333.56
4	27.02	2518.89	250187.62

These results show that diversity is much more effective if the system is already highly reliable. However, we must bear in mind the assumptions: channels fail independently and a failing channel gives a result that is both incorrect and different from other failing channels.

$N$ -channel systems in hardware have been used for a long time. The McDonnell-Douglas DC-10 uses three wires to control the rear surfaces (tailplane and rudder). If one or two cables break, the plane can be flown with the third one only. But when the baggage door of a DC-10 fell off after take-off from Paris in 1974, all three wires were broken and the plane crashed. This is called *common mode failure*.

One of the principle objections to  $N$ -channel software is that it is prone to common mode failure. The objection is based on the observation that programmers tend to think in the same way and therefore even independently written programs are likely to fail in the same way.

Another objection is based on cost. With hardware channels, there are likely to be economies of scale:  $N$  versions will cost less than  $N$  times one version. But with software,  $N$  versions will cost approximately  $N$  times the cost of one version.

There is a well-known study, by Knight and Leveson, on the effectiveness of  $N$ -version software. Hatton uses their data and his own reasoning; his results do not differ substantially from those of Knight and Leveson.

An application was written at two universities by 27 programmers of varying experience. The resulting code was thoroughly tested. The results for a 3-channel system, with the three channels chosen from the 27 versions, was:

- ◇ theoretical improvement for 3-channels: 833 times
- ◇ actual improvement for 3-channels: 45 times

The defect rate of software remains fairly flat. Typical programmers and testing teams produce software with about 3 defects/KLOC. An air-traffic control system which was developed with formal methods, 100% statement coverage during testing, and concurrent test plans obtained 0.7 defects/KLOC. In other words, if we do everything right at great expense, we produce a system that is 5 to 10 times more reliable than if we just muddle along.

These results make  $N$ -channel software look attractive: a factor of 45 is better than a factor of 5 to 10. But can we compare reliability measurements with defects/KLOC?

Suppose that we let  $p$  be the reliability of an average system and  $q$  be the reliability of a state of the art system. We have the choice of a single state of the art system or 3-channels of average quality. The improvement factor of the 3-channel system is

$$\frac{q}{p^3 + 3(1-p)p^2}$$

which, for  $p \ll 1$ , is approximately  $q/3p^2$ . Thus state of the art quality would have to improve at least as fast as the square of average quality if one-channel systems are going to keep ahead of 3-channel systems. Steady improvement seems to imply that  $N$ -channel systems will be increasingly worthwhile.

## 6.6 Testing for Reliability

Estimates of reliability, expressed by such measures as Mean Time To Failure (MTTF) are based on a theory of reliability. The theory is taken from the engineering of physical systems and applied to software systems. Dick Hamlet (Hamlet 1992) asks: is reliability theory valid for software systems? Is it even valid for physical systems?

There are two kinds of reliability model:

- ◇ A **reliability growth model** is used during debugging and models testing, failure, and correction. Managers can use reliability growth models to decide when to release software. (We have seen examples earlier in this course.)
- ◇ A **reliability model** are applied after testing is finished and when there are few or no known errors in the software. The reliability model predicts the MTTF.

It might seem that the reliability model is just the limiting case of the reliability growth model — as the number of remaining errors drops to zero. But there is an important difference: reliability growth models are based on **observed** errors but reliability models are based on **predicted** errors. To put it another way, it's fairly easy to predict the MTTF during testing, when it is short, but quite different to predict it after testing, when it should be infinite.

Software reliability is usually modelled by analogy to physical reliability. Physical reliability is concerned with collections of similar parts that differ slightly because of random variations in manufacturing, operating environment, or durability. The MTTF of a structure or machine is estimated in terms of the MTTF of its parts under normal working conditions.

In software, a single program is given different inputs during testing. For each program  $P$ , a **failure rate**  $\theta$  is assumed:  $\theta$  is the probability that  $P$  will fail when given an arbitrary input. Inputs are assumed to be drawn from a distribution that reflects the actual operating frequencies (cf. *extended propagation analysis* in Section 6.3).

The probability that a single test will fail is  $\theta$  and the probability that it will succeed is  $1 - \theta$ . The probability that  $N$  independent tests will succeed is  $(1 - \theta)^N$ . Reliability requires that this probability be greater than a certain constant —  $\alpha$ , say. Thus we require

$$(1 - \theta)^N > \alpha$$

which we can solve for  $\theta$  giving

$$\theta < 1 - \alpha^{1/N}$$

For example, with one million test points ( $N = 10^6$ ) and a confidence level of 99% ( $\alpha = 0.99$ ), we get  $\theta \approx 4.6 \times 10^{-6}$  which corresponds to an MTTF of about 220,000 runs.

How reliable is this result?

**Random variables differ** If the software is intended to run continuously (e.g., a telephone switch) starting it and waiting for failure is analogous to testing a physical object. The random variable is *execution time* and you can form a collection of results by giving the program different inputs. The MTTF is in fact an estimate of mean *time* to failure.

But if the program is used in batch mode, or interactively by people, the random variable is the *number of runs* and the MTTF is an estimate of the mean *number of uses* before failure.

Thus the significance of the predicted MTTF depends on the assumptions used in its calculation.

**Runs are not independent** The statistical theory depends on the fact that failure modes are independent of one another. It is not intended to take into account (in a physical system) the possibility that wear in one component might cause failure in another.

Programs accumulate state and this has the effect of compromising independence. In an extreme case, one fault might put the system into a state where it could not accept any more input. Parnas has proposed that safety-critical programs should initialize themselves whenever possible to avoid bad state build-up.

Physical systems fail because defects appear as the components are used. (Design errors sometimes occur, as when the Kansas City Hyatt-Regency Hotel walkway collapsed, but most physical systems just wear out.) The errors in programs are there from the beginning and there is no reason to believe that they are independent or purely random.

The theory will be valid only if inputs are *uncorrelated with respect to defects*. But there is no way of checking this, since we do not know where the defects are. In fact, the opposite is likely to be true: some defects will affect many inputs, and other defects will never be detected.

**No operational profile** For some applications, the operational profile is easy to determine. Since telecoms keep detailed statistics of telephone calls and can predict the usage patterns of switches precisely. But for other applications it may be very hard to collect such data.

**No failure rate** The theory assumes that the probability of failure is independent of time: the system is just as likely to fail after 1 minute of operation as it is after 1 year. This may not be true for software: every programmer has had the experience of debugging a program that never crashes before it has run for an hour or more. (This kind of fault is typically caused by small memory leaks, counters overflowing, etc.)

**Defects per KLOC** The theory predicts that MTTF does not depend on program size. But observation suggests that MTTF varies inversely with program size — the larger the program, the more often it crashes. This is consistent with a constant ratio of defects/KLOC.

Thus conventional reliability theory is deficient in two ways:

- ◇ it relies on an operational profile that may not exist
- ◇ it makes incorrect assumptions about sampling independence

What would a better theory of reliability look like? It is not good enough to make vague predictions such as

- ◇ random testing should replace unit partition testing
- ◇ inspections are better than testing

A reliability theory requires a *sampling basis* to infer the probability of failure. Program input is *not* an appropriate sampling technique. This is because there is no simple relation between input data and causes of failure.

Failures are caused by errors in the *text* of the program. This suggests that reliability measurements should be based on source text rather than different inputs. (This is perhaps one reason for the success of inspections in detecting defects.)

Another possibility is to sample the state space of the program. Since the state space is the Cartesian product of all the values of all the variables, it is likely to be large. But there may be ways of taming its size.

- ◇ *Program slicing* extracts parts of the program that have particular effects. Slices relate parts of the state space to parts of the code. (Jürgen Rilling is a researcher in this area.)
- ◇ The state “fans in” and “fans out”. An assignment statement such as `x = 0;` is an extreme fan-in because it reduces the number of states to one ( $x = 0$ ), at least as far as `x` is concerned. An input statement such as `cin >> x` is the ultimate fan-out because it leaves `x` completely undetermined. Fan-in and fan-out can be used to make state space analysis more manageable.

## Summary

- ◇ When we are testing for reliability, tricky partition methods may be no better than random testing.
- ◇ Analogies to reliability of physical systems re misleading.
- ◇ More research is needed on state space analysis.
- ◇ It may be possible to characterize programs according to “testability”.
- ◇ If testing for quality is the goal, we must solve the oracle problem: if we need a million tests, how do we check the output for every one?

## 6.7 Aspect-Oriented Programming

The first aspect-oriented programming language is AspectJ, based on Java, and developed at Xerox PARC ([www.aspectj.org](http://www.aspectj.org)).

Traditional exception-handling code suffers from any problems, most importantly that it is not modular: exception cross module boundaries. This is sometimes inevitable but more often because programmers treat exception handling carelessly, sprinkling handlers through the code at the last minute. Reuse of exception handlers is rare and abstraction techniques are rarely applied to handlers. The only kind of abstraction that is often applied to error handlers is to catch many different kinds of error: but this leads to unreliable and error-prone code.

The core of the problem is that it is difficult to write disciplined exception handling code using conventional object oriented techniques. Features, such as exceptions, that cross module boundaries are called *cross cutting concerns*. Cross cutting concerns lead to code that is unreliable, hard to understand, and hard to maintain.

In AOP, programmers write their programs in a simple way in a base programming language. The programmers can also define *aspects* that cut across modules of the program. Aspects can observe objects and respond to their behaviour.

In some ways, aspects are the opposite of inheritance. With inheritance, you choose to subsume some of the behaviour of an object. With aspects, you choose what behaviour the objects will subsume.

AspectJ is Java with four added constructs: aspects, join points, advice, and code introduction. We illustrate AspectJ by showing how it helps with exception handling problems.

**Aspects** An *aspect*, like a class, is a typed entity that contains functionality. Aspects are intended to address crosscutting concerns and they may contain programming features that classes cannot contain. A programmer might define an aspect that encapsulates a system-wide exception-handling capability, or a more specialized aspect that handles a particular class of exceptions. Another possibility would be to write an aspect that supplements every raised exception by logging it.

**Join Points** Objects cannot be crosscut at arbitrary places. AspectJ allows crosscutting only at certain well-defined places, such as at object construction time, method entry time, and variable access points. These points are called *join points*. A join point is introduced by a *pointcut declaration*.

**Advice** After you have defined the join points at which an aspect is to be applied, you have to define the behaviour that you want. This behaviour is called *advice*. The advice consists of more or less arbitrary Java code. There is *before advice* and *after advice*; as the names suggest, the qualifiers determine when the advice should be taken.

**Code Introduction** Programmers can add variables and methods into existing classes to enrich their behaviour. An exception handling application would be to maintain a class-specific count of errors raised.

**Example** The aspect below prints all uncaught exceptions in the current package. The pointcut `allMethods` cuts all executions of any function in the package. The keyword `executions` specifies that we wish to cut method executions and its argument is a wildcard expression saying that we want to cut methods with any signatures. A different expression here might specify that we want to cut only methods with integer arguments, for example.

The first `*` indicates that the method might return anything and the second indicates that any name matches. The `(..)` indicates that there can be any number of arguments.

```
aspect ExceptionPrinter
{
    pointcut allMethods() : executions(* *(..));
```

```
static after() throwing (Exceptions e) : allMethods()
{
    System.out.println("Uncaught exception: " + e);
}
}
```

## References

- Beck, K. (2000). *Extreme Programming Explained: Embrace Change*. Addison-Wesley.
- Beizer, B. (1997, March/April). Cleanroom process model: a critical examination. *IEEE Software* 14(2), 14–16. See also correspondence in the July/August issue.
- Brettschneider, R. (1989, July). Is your software ready for release? *IEEE Software* 6(4), 100–8.
- Chillarege, R., I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong (1992, November). Orthogonal defect classification — a concept for in-process management. *IEEE Transactions on Software Engineering* 18(11), 943–956.
- Cusumano, M. A. and R. W. Selby (1995). *Microsoft Secrets*. The Free Press.
- Ellis, M. A. and B. Stroustrup (1990). *The Annotated C++ Reference Manual*. Addison-Wesley.
- Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- Gibbs, W. W. (1994, September). Software’s chronic crisis. *Scientific American* 271(3), 86–95.
- Gosling, A. (1998). *The Java Programming Language* (second ed.). Addison-Wesley.
- Hamlet, D. (1992, July). Are we testing for true reliability? *IEEE Software* 9(4), 21–27.
- Hatton, L. (1997, November/December). *N*-version design versus one good design. *IEEE Software* 14(6), 71–75.
- Humphrey, W. S. (1997). *Introduction to the Personal Software Process*. Addison-Wesley.
- Jeffrey (1997, March/April). How assertions can increase software effectiveness. *IEEE Software* 14(2), 118–122.
- Jones, C. (1991). *Applied Software Measurement: Assuring Productivity and Quality*. McGraw-Hill.
- Lakos, J. (1996). *Large-Scale C++ Software Design*. Professional Computing Series. Addison-Wesley.
- Lethbridge, T. C. and R. Laganière (2001). *Object-Oriented Software Engineering*. McGraw-Hill.
- Mills, H., M. Dyer, and R. Linger (1987, September). Cleanroom software engineering. *IEEE Software* 4(5), 19–25.
- Parnas, D. and D. Weiss (1987). Active design review: principles and practice. *Journal of Systems and Software* 7, 259–65.
- Poore, J. H. and C. J. Trammell (1998). Engineering preactices for statistical testing. Technical report, Software Engineering Technology.
- Pope, A. (1711). *An Essay on Criticism*, Volume 1. Lewis.
- Schneidewind, N. D. and N. Fenton (1996, January). Do standards improve quality? *IEEE Software* 13(1), 22–24.
- Stavely, A. M. (1999). *Toward Zero-Defect Programming*. Addison-Wesley.
- Voas, J., F. Charron, G. McGraw, K. Miller, and M. Friedman (1997, July/August). Predicting how badly ‘good’ software can behave. *IEEE Software* 14(4), 73–83.

Yeh, H.-T. (1993). *Software process quality*. McGraw-Hill. QA 76.76 D47Y44 1993.