CS446

Richard H. Bartels

September 12, 1996

Contents

Chapter 1

General Information

This is likely to be the last term in which CS446 will be given in its present form. The department is taking the same material and spreading it over three courses, one devoted to *Requirements*, one devoted to *Implementation* and *Testing*, and one devoted to *Maintenance*, three basic components of the *Software Development Life Cycle*. In attempting to touch on these three in a single term, our coverage of any single aspect of Software Engineering will have to be very light.

1.1 Staffing and Scheduling

instructor: Richard Bartels, rhbartel@watcgl, DC 2108, ext. 4528

teaching assistant Stuart Gill, sggill@neumann, DC 2131, ext. 3352

news: uw.cs.cs446

www: `http://www.undergrad.math.uwaterloo.ca/~cs446/info.html`

classes: MWF 2:30-3:30 MC 4058(MW), MC 4063(F)

text: Software Engineering, Fifth Edition,
Ian Sommerville, Addison-Wesley, 1996

library reserve materials: 1993, 1613, and 1614 (all for overnight usage)

environment: SunOS, GP development tools (enhancements to RCS, imake, man), g++, Microsoft Project, standard Unix tools

manual: ~cs446/Formatted/Manual.ps

1.2 Course Goals

The goals of this course will be to

- simulate project-oriented software development (e.g. as might be seen in a small company),
- provide exposure to some major issues in large-scale software development,

- make us aware of the certain areas of concern and investigation in software engineering,
- provide an appreciation for the needs and motivations that drive software engineering,
- sample a few techniques in current use and (possibly) assess their strengths and weaknesses.

We will attempt to reach these goals by forming equal-sized teams (as nearly as can be done) and carrying out the design, implementation, documentation, and maintenance of a small software system. Each team will work on its own version of the system; i.e., each team will act for most of the term as a company in competition against other companies in getting a product to market. In a final phase of work, systems will be reassigned to different teams, and each team will act as the maintenance department in a larger corporation assigned to enhance the product in a certain way. In order for this to be a success, standardization will be required (just as in real life). In practical terms for this course this will mean

- a required *programming language*,
- a required *compiler*,
- a required *coding style*,
- a required *make* facility,
- a required *version control* facility,
- a required *code documentation system*,
- required *documents*.

All work delivered to the "customer" and to the "maintenance department" must conform to required standards.

1.3 Project Goals

The product to be developed is to be a cross-country flight planner for a small airplane. Object-oriented design will be used, with design stress to be laid upon the use of abstraction, encapsulation, dynamic binding, and loose coupling to provide clean structure and maintainability. Documentation at three levels will be provided. One level will be planning documents. A second level will be development/maintenance documentation in the form of code comments and man pages. The third level will be user documentation. (Students will maintain a fourth level: an accounting of their individual time and activities.)

The final product will include the legacy of code, testing structure, documentation, and archives. Each team will also deliver an honest time accounting for its development efforts, from which a final "price" for the product can be determined.

The development effort will follow several phases, consisting of:

1. consultation with the customer, establishing requirements;
2. planning implementation from requirements, presentation, submission for review;

3. planning testing strategy, presentation, submission for review;
4. work schedule planning, presentation, submission for review;
5. implementation, testing, documentation
6. delivery of system for evaluation
7. receipt of a similar system for enhancement (maintenance)
8. enhancement, testing, reporting
9. final delivery of all materials

In these phases there will be planning documents to be delivered, on which a team member must sign off, indicating that the quality assurance of the work was their personal responsibility. Class presentations of team material for review may be required, for which an assessment of personal responsibility will also be made. These factors will contribute to a person's individual grade.

The team as a whole will will also be assessed through the quality of the system material delivered. This will be judged in two ways. Firstly, by the "customer's" (instructor's) evaluation of its organization, style, documentation, testing, careful version control, and correct execution. Secondly, by the "maintainer's" (another team's) evaluation of its organization, style, documentation, testing, and ease of enhancement.

Additionally, before teams are chosen, individuals will be responsible for some short assignments to provide environment training and personal calibration.

A individual's final grade will be given as

- Individual grade
  - Personal responsibilities: 40%
  - Team's collective grade: 60%

where the former will be composed of

- Personal responsibilities
  - Short assignments: 20%
  - Sign-off responsibilities: 40%
  - Evidence and quality of team participation: 40%

And the latter will be composed of

- Team's collective grade
  - Implementation design: 15%
  - Testing design: 15%
  - Team scheduling: 15%
  - Requirements satisfaction: 30%

- Enhancement results: 15%
- Management (team selection, version control, work efficiency, etc.): 10%

In the real world, errors found later in the development life cycle are more expensive than those found earlier. To simulate this, the points counted off against a group or individual will escalate, as the stages of the project progress.

1.4 Note This Well

There will be no examinations. You will be expected to have software experience and maturity, personal maturity, and organizational and communication skills. You will be expected to work cooperatively with the other people in your team: able to manage them where necessary, be managed by them where necessary, and reliably handling your share of all responsibilities set by the team to achieve commonly chosen goals. This should **NOT** be the first 400-level course you take. **You have been warned!**

1.5 Job Offer

Newsgroups: can.jobs
From: David Murray <dmurray@prodigy.bc.ca>
Subject: Analyst - Vancouver
Sender: news@prodigy.bc.ca
Organization: Prodigy Technologies Corporation (Macintosh)
Date: Thu, 15 Jul 1993 21:22:18 GMT
Systems Analyst
A product focused organization using object-oriented technology, is seeking a Systems Analyst. The candidate will become a member of a dynamic group of young software professionals dedicated to applying advanced technology to solve today's business problems.
Your experience as an Analyst using structured or object methodologies in "big iron" or client/server architectures will permit you to hit the ground running. **Knowledge and practice of OOA and OOD will be considered a strong asset.**
You must have experience facilitating small software development projects and be able to recognize and set priorities in order to complete deliverables on time, on budget. Excellent written and verbal communication skills are a must. Experience with client presentations and end-user training an asset.
Mail or fax resume and salary expectations:
Prodigy Technologies Corporation
Re: Systems Analyst
1190 Hornby St. 11th. Floor
Vancouver, BC, V6Z 2K5
Fax: 604-687-1671.

Chapter 2

Overview

This chapter expands on most of the resource and organizational components of the course.

2.1 Teams

All work for the term will be initiated in teams of N individuals (with the possible exception of one larger team to handle any remainder). N will be chosen (and announced) based upon the size of the class and the number of teams we feel we can handle. No team will be permitted to form initially with fewer than N members. All team members must meet together, formally, for review and planning, no less than once every two weeks. You are to read the material on choosing team mates that appears in Chapter 3 and take it very strongly into consideration when forming your team. Each team will be expected to have at least one person who has experience in C++ programming and one person with good English communication skills. A team's overall success is strongly connected to its strength in these two areas.

2.2 Dismissal

If your efforts are not satisfactory to the rest of your team, your team members have the right to dismiss you. Such a dismissal will take place in two stages. Firstly, your team members must collectively inform you, in writing, of the nature of their dissatisfaction and provide a copy of their complaints to the Instructor. If you are not able to correct the cause of your team members' dissatisfaction within one week, your team members will inform the Instructor collectively, in writing, that you are no longer a member of the team. If you have been dismissed from a team, you will be required to fulfill some portion of the remaining work for the term by yourself, and the team will be given a reduced set of goals to achieve. *However*, a team must not take dismissal lightly, as it may reflect badly on a team's management grade. Chapter 3 has more words on this subject.

2.3 Newsgroup, Files, Web Pages

Everyone is to subscribe to the course newsgroup and is responsible for reading it frequently. All important announcements will be made via the newsgroup. Everyone is encouraged to post questions, answers, and tips of **general interest** to this newsgroup. Questions of interest only to a particular team or individual may be submitted through e-mail to the Instructor.

At various times during the term, any files containing useful or necessary items that may be needed will be placed into directories to be found under `/u/cs446` on the MFCF undergraduate computers. The availability of such items will be announced through the newsgroup.

Talk slides and interesting supplementary material available through the course web pages. The web pages will also accumulate, in ever more precise detail, the requirements of the system you are to produce.

2.4 Documents

Each team must submit a number of documents

- implementation plan
- testing plan
- scheduling plan
- user guide
- maintenance report and effort accounting

along with well-documented, stylistically-acceptable code. One team member acts as "presenter and individual responsible for the final review" of each document. The responsible person should "sign off" on the document; that is, his/her name should be clearly indicated on the cover page of the document and his/her signature should be on the cover page. No team member may act more than once in this capacity until all team members have signed off on a document at least once. The "signer off" will receive an individual grade based upon the quality of the document's *form*. In particular, such things as poor English, shoddy formatting, haphazard organization, unclear exposition, poor or incomplete presentation, and lack of neatness will be counted against the him/her. The grade for the *content* of the document, as opposed to its form, will be assigned to the team as a whole.

Time constraints will probably prevent each team from making a presentation on each of their planning documents, but any signer off must be prepared to make a presentation of approximately 15 minutes on the main points of the document's content, and the Instructor reserves the right to call upon any signer off to make such a presentation without prior notice.

If teams are large, there may not be enough documents for everyone to sign off a document. Team members who are left out may acquire a grade of equal weight by submitting a diary of their activity in the team (see Section 2.8). These submitted diaries will be judged on their form and comprehensiveness in the same way as the quality judgment applied to planning documents and reports.

2.5 Version Control

All team software is to be managed using *rcs* version control as provided by the GP environment. Team members must make use of the *gpinsertsector*, *gpextractsector*, *make checkin*, *make checkout*, and *make revision number=x.y* as appropriate to manage stages of development, release, maintenance, and re-release. Failure to make sensible use of this protocol will be reflected in the team's grade. In order for records to be taken accurately by *rcs* you **must** work under your own userid and in your own account. You will be penalized if this is not done.

2.6 Style

The GP environment mandates a particular header-file style, which every member of the team must employ. Use of this style will result in *man* page documentation for all sections of the code, but it does not mandate any style for code or commenting for the implementation code itself. Each team is to fix its own uniform style and submit a statement of that style with the code itself. All code prepared by team members must be commented and all code and comments from a team must be in the team's uniform style.

2.7 System Environment

The MFCF undergraduate *Unix* environment will be used as a common resource location. Team members who do any development off line (eg. on a home machine) must strive to keep their work shadowed on the MFCF machines as needed by the rest of the team. All work must ultimately compile and execute from a "cold start" in the GP environment on the MFCF machines. By this it is meant that the commands

gpmkmf; make install

alone, executed in the

`GP/src/Classes`

directory with empty

`GP/include` and `GP/lib`

directories, will compile all infrastructure code for the team's system, and the commands

gpmkmf; make

alone, executed in the

`GP/src/Applications/<team's-system-name-here>`

directory, will compile, link-load, and produce an executable for the team's system. No extra shell scripts or elaborate *make* constructions should be needed or used.

Documents are to be formatted in a document preparation system of your choosing in cooperation with **at least** a spelling checker and **ideally** also a grammar checker. You **must** use at least a spelling checker.

Any diagrams being provided should be prepared on a diagram preparation system, ideally one included with the document system, or at least one whose products can be included digitally into the documents (rather than physically cut and pasted).

Code delivery will be accomplished by removing *all executables, libraries, and links* from the team's version of its `src/Applications/*` and `src/Classes/*` directories (*make realclean* is designed to accomplish this) and mailing a *uuencoded*, *compressed*, *tar* file of that directory and its subdirectories to the Instructor. (Do not use MIME inclusion; mail the *uuencoded* file as plain text.) Successful delivery will require that the Instructor can

- un*tar* the team's `src` under his own `GP` root directory,
- execute the *gpmkmf* and *make* commands described above

and have a functioning system as a result.

2.8 Diary

Individuals should acquire and keep up to date a small, bound, laboratory-style notebook to serve as a diary of activities carried out and time spent for the software development work. The entries must provide brief information for at least the following:

a. time spent, date, agenda, and decisions for meetings held;

b. time spent obtaining and learning technical information;

c. time spent learning, formulating, or clarifying requirements;

d. time spent in software/data-structure design;

e. time spent in unit/integration/verification test design;

f. time spent in management/project planning;

g. time spent coding;

h. time spent debugging;

i. time spent testing;

j. time spent learning/reviewing code to be maintained;

k. time spent in maintenance coding/debugging/testing;

l. time spent preparing documentation;

The time information for each of the above categories must be summed up and be included at the end of the diary if it is handed in. This diary is for your own benefit, so choose the format and style that suits you best. If you have not served as signer off for a team document, you will be asked to submit your diary. You will be graded on evidence of care, neatness, and completeness. Whether or not you hand the diary in, it is through the diary that you will gain an appreciation of where and how much effort is expended in the various aspects of a software project. The diary is also the means by which your team will be able to report on its development costs.

Chapter 3

Partner Choice and Team Management

You will work in teams. Each team may simulate, at various times, a complete company or a department within a larger company. Each individual may act, at various times, as an employee or as a manager. A significant percentage of your grade will depend on the performance of your team. As in real life, the individual employee/manager suffers if the company goes bankrupt; and companies with competent, productive, cooperative employees/managers tend to go bankrupt less frequently. It is important that you avoid difficulties as much as possible in advance and that you pay attention to the health of your team as the term progresses.

3.1 Some tips on selecting a team.

Forget who your friends are. If your friends do not fit the requirements below, they won't be your friends after you have tried to work with them on a team.

1. **Choose partners who have similar work habits to your own.**

   Do you work best in the mornings, but your partners work better in the evening? Do you like to get started on assignments quickly, but your partners usually procrastinate? Are you neat and precise, but your partners are more laid back about how they work? Are you organized, but your partners are chaotic?

2. **Choose partners who have similar abilities.**

   You may be tempted to choose a partner who is vastly better than you in some critical skill relating to the project. Unless you all are very controlled about sharing the work, the skilled partner will do everything, and hate the rest of you. You, on the other hand, will learn nothing, may do poorly, and find yourself subject to dismissal. There are only two cautionary notes on this: you must make sure that at least one individual can serve as a knowledgeable reference on C++ and at least one can serve as an effective editor for English. Code must be good and documentation must be good, and someone must be able to review both. If all team members are competent in both areas, so much the better. If not, remaining team members must be willing trade their services in other areas in return for these abilities, if necessary, so that the workload is balanced.

3. **Choose partners who have similar goals and expectations.**

   Do you want to do the best project in all of history, but your partners are happy with a passing grade? Do you want to learn as broad a sweep of course material as you can, but your partners would rather focus on a few issues in depth? Do you want to apply as many techniques as you can to the work, but your partners want to specialize on one or two useful tools.

4. **Choose partners who have similar workloads and resources.**

   Can you allocate five hours a week to the course work, but you have a partner who is loaded twelve hours a day and struggling to keep up? Do you have to travel home to Calgary every Wednesday, but your partners' parents live in Waterloo? Do you have to come to the Math Building to accomplish all your computing, but your partners have PC's, modems, and printers?

5. **Pay attention to interpersonal dynamics.**

   Make sure there are no potential personality conflicts. Two people with dominant personalities, for example, and each with a need to control, provide a recipe for team stress and discord. Be aware of other possible incompatibilities as well. People with different attitudes to responsibility and with different styles of thoroughness or precision may also come into conflict. Ask yourself whether you can really enjoy associating with these other people throughout a long and difficult term.

Please note well: part of the grade assigned to your team will be based on the Instructor's assessment of whether the team was competently and carefully chosen!

3.2 Team Logistics and Meetings

You will be **required** to meet as a full team at least once every two weeks. Each meeting is to have an agenda and protocol precise enough and small enough that it can be expected to last under two hours. The team must keep a common record of the agendas together with what was proposed, discussed, decided, accomplished. The Instructor reserves the right to ask to see this record.

Apart from these meetings, you should:

- exchange full names, phone numbers, e-mail ids, addresses;
- establish a method for between-meeting communication and consultation;
- organize a common bulletin-board file to record items to do, milestones, tasks, assignments, progress, etc.;
- establish a firm, meeting schedule at which everyone can, and is expected to be, present;

- spend time getting to know each other and share any concerns you have about the team, the arrangements, and the course;
- agree upon and establish a written guide for a uniform implementation code style.

A useful form for meetings is one in which one person is responsible for keeping things tightly on schedule, one person keeps summary notes, and some effort is made to cover:

- an update on each person's progress in their assigned tasks,
- an update on the progress toward each goal,
- a review of the progress chart and any necessary revisions,
- an analysis of workloads and schedules with any necessary revisions and redistributions,
- the setting of deadlines and goals for the next meeting,
- an evaluation of how well the team is functioning and an effort to air and resolve difficulties,
- a review of any past decisions that appear not to be working and need to be revised.

3.3 Management Issues

Understand the rights and responsibilities of the individual in carrying out his/her tasks. Determine a viable mode of making decisions and establishing team control that all can live with. Strive to discover problems and resolve them before they become serious. Learn to manage as well as to be managed. And finally, when all else fails, follow careful and established procedures for dismissal.

3.3.1 Rights and Responsibilities

During the course of the term, the team will accomplish its work by delegating portions of it to each member. Once work is delegated, each member has **responsibilities:**

- to provide the interfaces agreed upon,
- implement the internals with methods of good quality in code of the style agreed upon,
- provide documentation,
- provide adequate testing structure,
- meet the deadlines established.

The member also has the **right** to be free of "micro-management," able to make her/his own decision on the finer details of how the task is accomplished without unnecessary interference by others in the team.

3.3.2 Decision Strategies

Decision strategies may vary from team to team, but each team should establish early, and firmly, how it will make its important decisions. Since it is unlikely that all members will agree with all decisions, some mechanism must be established to make decisions in the face of conflicting ideas. Three schemes that have been tried with success are:

1. **Decision by Majority**
   in which a vote by more than half the members of the team carries the decision (more difficult in small teams, especially with an even number of members);

2. **Decision by Unanimous Consent**
   in which discussion proceeds, and the decision is revised, until all are satisfied that the decision is truly the best choice (which may take a significant amount of time on occasions);

3. **Decision by Consensus**
   in which the team discussion is open and responsive to the views of all participants and the decision, while not necessarily exactly what everyone finds optimal, is still understood by all, was influenced by all, and can be accepted by all (which requires a significant degree of honesty, openness, and maturity from all participants).

In making decisions it is well to remember that a function of the regular meetings should be any necessary re-assessment of past decisions in the light of experience. Never make a decision that is not open to adjustment.

3.3.3 Problem Avoidance

Problems will arise, and it is important that modes of interaction are established that help in addressing and resolving them.

- Be open. Be honest with praise and be diplomatic and constructive with criticism. Do not let the good or the bad go without notice or remark.

- Communicate. Listen carefully; don't interrupt; ask for clarification, if you don't understand; feel free to disagree constructively, and be prepared to accept disagreement calmly.

- Get agreements, responsibilities, and task assignments clearly made and recorded; don't assume that your understanding is the same as someone else's unless it has been discussed explicitly and noted in writing (e.g. on a central bulleton-board file).

- Get to know each other. Discover the person behind the team member. Learn to accept likes, dislikes, strengths, weaknesses, work habits and motivations. Learn to balance loads so that strengths are utilized and weaknesses avoided or corrected.

- Encourage participation. Make each member feel comfortable and willing to contribute to discussions and decisions.

- Compromise.

- Stay on schedule.

- Allocate sufficient time to do a good job.

3.3.4 Managing and Being Managed

Each body of work handed in before the final project is delivered is to be "presented and signed off" by a team member, and each team member is to act in this capacity no more than once during the term until all team members have acted at least once in this role. This does not mean that the team member must do the work, although she or he may participate in it, but it is the team member's responsibility to manage activities and resources so that the work does get done. And it is a further responsibility that the team member check the work over and verify with her/his signature that it is up to reasonable standards of quality, both in form and content.

The handing in of a body of work may be accompanied by the requirement that the work be presented for review by the instructor and the class. The responsible team member is to do this alone.

3.3.5 Dismissal

It has happened in the past that some teams are so poorly structured that they cannot accomplish their work. Sometimes this comes from the deficiencies of a single individual; e.g. a member that does not fulfill responsibilities, does shoddy work, is unreliable. Single individuals can be dismissed from a team only after a meeting has been held at which the individual is informed of problems and given a week to correct the situation. The instructor is to be advised of the meeting, and its reasons, in writing. If the situation does not improve, The instructor is to be informed, again in writing, that the individual has been discharged from the team. The instructor, after having conferred with those involved, will reduce the extent of the project that the team must accomplish and determine what work the individual should carry out on his/her own.

It may also happen that a team splits into factions that are unable to work productively together. Any team suffering this problem is to schedule a meeting with the instructor to negotiate the termination of the team's existence and an arrangement of work to be accomplished separately.

It should be understood, however, that the responsibility for the continued success of a team lies with **all** the individuals in it. Dismissal and team termination cannot help but reflect negatively to some degree in the grades of all those involved. The dismissal of an individual or the termination of a team is not a decision to be taken until all other avenues of repair have been explored.

Chapter 4

Documents

During the course of completing the project, each team will be required to go through several stages of planning and preparation. The completion of each stage will be evidenced by the submission of a report document, signed off by a team member, and the possible presentation of the report's material by the signing team member.

Since the approach chosen for this term involves building a system in C++, some of the terminology in one finds in the text or other software engineering literature may be inappropriate. Classical software engineering books are mainly oriented around the view that a system is built with a *functional* (*procedural*) language, for which functional decomposition is the appropriate means of top-down design and the *module* is the basic, encapsulated functional unit. In an object oriented language such as C++, functional decomposition into modules is only one aspect of design. Classes, class inheritance, entity relationships, class clusters, class inclusion of objects from other classes, polymorphic code, class interface, class services, object creation, object activity, and communication are also important.

In a functional language, data is data and function is embodied in procedures. Data and procedures bundled together with a specified interface for usage constitute a module. (In conventional C, a module is usually provided by a pair of files, a `.h` file that the user includes, which declares publicly available elements, and a `.c` file that implements these elements (possibly using private, hidden, `static` elements). The elements provided to the user are simple data types (data without process) and procedures (process applicable to simple data types).

In C++, data and function may be bundled together in yet another form, as a *class*. Depending on your point of view, a class is data enriched by associated processes, or it is a group of cooperating processes enriched by shared data. Indeed, classes may be designed to take on the role of any conventional program element, from a data element or function through an entire subsystem. Data, furthermore, as used in this description, is not simple. The data of one class can consist of objects of other classes: bundles of data and process in their own right, and so on, extensively. Classes, moreover, may exist in hierarchies, groups of related classes that progress from the simple to the more enriched in a progression of related features. Code may be designed using only the properties of the base classes of these hierarchies yet be executed polymorphically on any object from any of the derived classes. Finally, clusters of classes (together with conventional procedures that use them) can still be bundled, in C fashion, into conventional modules.

Thus, the challenge throughout the planning process is to arrange form and function into units that have a coherent organization *structure*, a clearly defined set of *responsibilities*, and a well-designed, clean *interface*, and (in the case of a hierarchy), contains code written for objects from classes as near to the root of the hierarchy as possible. Such units, from here on, will be referred to as **entities**.

The report documents to be submitted by each team are the *Design Plan*, the *Testing Plan*, the *Scheduling Plan*, the *Maintenance and Cost Accounting Report*. The Design document will be submitted at least twice, once before implementation begins and once, updated to reflect the final version of the system, upon project submission. An initial planning exercise in software development commonly results in the *Requirements Document*. In the interests of time, requirements will have to be kept somewhat informal. All common understandings as to what requirements are important will be added to a *Requirements Page* on the course web material.

4.1 Requirements Document

When one is developed or provided, such a document is a specification of the way in which the system is to perform with as little commitment as possible to how this performance is to be accomplished. Behavior, information input/output, and information organization are stressed in terms precise enough so that:

- the customer and the company understand what is intended;
- the customer agrees to what is promised;
- the implementer understands what should be achieved by the code;
- the tester has a precise statement of what must be validated;
- the maintainer has a recognized base of correctness to which to refer.

4.2 Design Plan

This is a high-level design of implementation, specifying entities with enough detail so that individual programmers could complete them independently yet without so much detail that there is nothing left to do but transcribe code. In the conventional Waterloo practice of code design, this will correspond to something like the second or third level of refinement in a top-down design. For C++ classes or class hierarchies, this will correspond to an overview of the class interfaces and entity relationships without detail about the implementation or stress on the private components. It will also provide an overview of control flow and of the creation, activity, and destruction of objects.

4.3 Testing Plan

This is a plan for the testing of entities, their integration with other entities, and the final validation that the system as a whole satisfies the requirements. The document will begin with a statement of the priority order in which the entities should be implemented, paying attention both to the dependencies among entities and to the staging of implementation so that a partially tested and completed system could be delivered in the event of difficulties or delays. The document will include a declaration of what testing philosophy will be used (top down, bottom up, mix, whatever), what entities will be treated in what order, what test, integration, and validation cases will be applied, and what extra facilities (scripts, debug sections, scaffolds, stubs, whatever) will be built, and (most importantly) what will constitute success in each phase of the process. This document will also include a statement of the order in which the entities should be unit-tested, integrated, and validated.

4.4 Scheduling Plan

This document begins with an estimate of the times required to implement and to test all fundamental units of the system. The units correspond to *tasks* and *milestones* on the way to delivering the system. It follows with a plan, including task network, time line chart, resource allocation table as produced by Microsoft Project, that organizes what will be accomplished, when, and by whom. This document should also pay attention to the priorities of tasks and entities established in the Testing Plan so that, if schedules slip, rational fallback positions can be assumed that will result in a successful *partial* fulfillment of the project.

4.5 Project Delivery

On an established delivery date your team will provide a complete set of its *source files*, *archive files*, *testing files*, *script and imake files*, and any *documentation files* you choose to include. You should also include an *executable file* for your system. This will typically be requested in the form of a final, *cleaned* snapshot of a designated portion of the development environment in which you will be *required* to do your work. Along with your files, you should submit a revision of your Design Document that has been updated to be consistent with the system being submitted.

4.6 Maintenance Report

This document will consist of a possible list of bugs (in code), deficiencies (in documentation and users' guide), and inadequacies (in testing legacy), that were encountered and the steps taken to correct them. It should contain, as well, an honest assessment of the strengths and weaknesses of design and implementation in the system, as you see them. Thereafter should come an abbreviated implementation design for the enhancement requested and an addition to the users' guide corresponding to the enhancement. In the event that the system provided for maintenance and enhancement was too flawed to be saved, the report will expand the "honest assessment" mentioned above to give a full accounting of the flaws.

4.7 Users' Guide

A brief guide to using the system, in any format you choose, provided that it will be clear and instructive to any member of the class.

4.8 Cost Accounting

Finally, during the project phases starting with the establishment of your team and ending with the delivery of your project, every team member should have been keeping a simple diary of time spent on various activities. A team summary of that material should be prepared and handed in with before the end of the term so that an approximate "price" for your system can be computed. The cost-accounting document will contain the total times spent, in hours, for each of the activity categories given in 2.8, summed over all team members. It will also include a bottom-line total of all hours spent by all members of the team in all activities related to the software development of the system. These figures are only to include the original system *development* and should not take any of the *enhancement/maintenance* activities into account.

4.9 Assignments

The chapters following provide specific assignments for individual "calibration" tasks and for the Design, Testing, and Scheduling Plans, Maintenance Report, Cost Accounting, Users' Guide, and Deliveries.

Chapter 5

Programming Calibration

This assignment will provide the Instructor with an estimation of your strengths in C++ programming and the use of *Unix* utilities.

5.1 Description

You are to prepare and submit, in C++

1. an implementation of the abstract data type(s) described below
2. a short program example illustrating the use of the type(s)

All development is to be carried out in a directory named `ADTypes`. The classes that implement the type(s) are to be be contained in `.h`/`.cc` files directly in this directory. The example is to be provided by files (`.cc`/`.input`/`.output`) to be found in the subdirectory `ADTypes/Examples`. In this subdirectory will also be found a `README` file describing what is illustrated by running the program with the input to produce the output. No other subdirectories are to be present.

The directories `ADTypes` and `ADTypes/Examples` will each have a `Makefile`. While in directory `ADTypes`, the command *make* will cause all code for the classes to be compiled, producing a library (`.a`) file in the `ADTypes` directory. While in the subdirectory `Examples` the command *make* will cause the example program to be compiled and linked to the library file, producing an executable named `example` in the `Examples` subdirectory. In this subdirectory, with such an executable, the command *make verify* will cause the executable to be run, with the `.input` file as data, following which the output of the execution will be compared with the `.output` file using *diff*.

In `ADTypes` the command *make clean* will remove all `.o`/`.a` files in *both* `ADTypes` and `Examples`. In `Examples` the command *make clean* will only remove the `.o` files from the `Examples` subdirectory.

5.2 Submission

Submission is to take place not later than the specified deadline by

1. executing *make clean* in `ADTypes`,

2. preparing a *tar* file named `ADTypes.tar` of `ADTypes` and all its contents,

3. using *compress* to produce the file `ADTypes.tar.Z`

4. using *uuencode* to produce a file `ADTypes.uu` that will be mailed to the Instructor.

The Instructor *must* be able to use the command

uudecode ADTypes.uu

to successfully recreate the file `ADTypes.tar.Z`.

5.3 Abstract Data Type(s)

We abstract the notion of a *FiniteStateMachine* and the related notions of a *Stimulus*, a *Transition*, an *Action*, and a *State*.

FiniteStateMachine

A *FiniteStatMachine* maintains a set of *State*s, one of which is designated as the *current State*. A *FiniteStatMachine* provides the following services:

addState

a given *State* is added to the machine;

initiate

an already given *State* is designated as the current one;

addTransition

a *Transition* that links two already given *State*s (or an already given *State* with itself) is defined for a given *Stimulus* and a given *Action*, one *State* is the beginning of the *Transition* and the other is the end;

fire

a *Stimulus* is given to the *FiniteStateMachine*, and if the current *State* is the beginning of a *Transition* with that *Stimulus*, the current *State* is changed to the end *State* of the *Transition* and the *Action* of the *Transition* is carried out.

Stimulus

A *Stimulus* is a key that selects a *Transition* based upon the *current State*.

State

A *State* is an entity that provides the current focus for a *FiniteStateMachine*; pairs of *State*s (or a single *State* repeated twice) serve as the beginning and ending of a *Transition*.

Action

An *Action* provides a single service:

doIt

a procedure with no input that carries out some algorithm (for the purpose of this assignment, it can merely print a message).

As a test that your *Finite State Machine* can do something useful, write a simple example program that uses it to simulate a coin-operated turnstyle. The turnstyle is initially "locked." In a locked state it cannot be "rotated." Upon "deposit" of a coin, it becomes "unlocked." While in an unlocked state, it will not accept deposit of a coin. After being rotated (i.e., when a person passes through), it returns to the locked state.

5.4 Objectives

The grade assigned will be based upon the following considerations:

1. a correctly uuencoded, compressed, tar file is submitted
2. the submitted directory structure is correct
3. the directories have been cleaned
4. there are no permissions problems
5. all *make* targets work as required
6. compilation is successful for classes and example
7. library file and executable are provided as specified
8. code style and commenting is good
9. code efficiency is good
10. memory is managed correctly
11. data type(s) and operation(s) are correctly implemented
12. errors are handled reasonably
13. example program clearly verifies correctness
14. `README` file is sufficiently informative

Chapter 6

Installing Calibration

In this assignment we establish that you can correctly install the GP environment and make use of some of its main features. This will also give you the incentive to become familiar with the GP manual and raise any questions you may have about it or about the environment.

6.1 Description

You are to take a copy of the GP environment (available as the file `GP.uu` under `/u/cs446`) and, following the instructions in the GP manual, install a copy in your own user directory space. This will involve uudecoding, uncompressing, and tar extracting `GP.uu`. Further, you will need to edit some files, and set your `PATH` to include your version of `GP/bin`. The files to be edited are `GP/bin/LOCATIONS` and (possibly) `GP/config/DEFINES`. The commands *g++configure* and *c++configure* in `GP/bin` will assist with the `LOCATIONS` file. Further details are in the manual under the chapter on project installation. Feel free to ask your Instructor for clarifications.

In an appropriate location for class development you are to use *gpnewsector ADType* to create an `ADType` directory. Within this directory use *gpnewclass* to provide the models for newer versions of your `ADType` codes from Chapter 5. Within the `ADType/Examples` subdirectory use *gpnewexample* to provide the model for a newer version of your `ADType` example program.

Transcribe your previous `ADType` codes (`.cc` contents) and interface specifications (`.h` contents) into the corresponding newer models provided by *gpnewclass* and *gpnewexample*. You may have to remove unnecessary elements as well as add other material to provide your `ADTypes` with the standard GP functionality for the input, output, and equals operators, as well as for run-time type checking, `get/set` conventions, constructors, and destructor.

All classes that you wrote to implement your `ADType`, *must* be brought up to conformity with GP conventions. Furthermore, you are to convert to using GP standard `Imakefiles` (`Makefiles` of your own design are not allowed), and you are to edit all comments in your `.h` files so that they provide appropriate *man* pages. Any member data or methods generated by *gpnewclass* that are not necessary for your classes should be *removed.*

6.2 Submission

Use *gpinsertsector* to install your new `ADType`. Use *make realclean* to clean the GP-conforming `ADType` directory tree. A uuencoded, compressed, tar file of the cleaned, GP-conforming `ADType` directory tree is to be mailed to the Instructor not later than the designated deadline, as was done for Chapter 5.

6.3 Objectives

The grade assigned will be based upon the following considerations:

1. permissions, tar extraction, directory structure, etc. are good
2. `Imakefiles` generate workable `Makefiles` via *gpmkmf*
3. modifications to the generated `Imakefiles` are minimal
4. no hand-crafted `Makefiles` are used
5. all classes conform to GP conventions on input/output
6. all classes support run-time type checking
7. all classes conform to GP conventions on the equals operator
8. all classes conform to GP constructor/destructor conventions
9. all classes generate appropriate *man* pages
10. GP *make* targets work as expected
11. class installation (*make install*) conforms to GP conventions
12. code style is uniform throughout
13. example program provides a useful *make verify* demonstration
14. example program is informatively commented
15. `README` file is instructive
16. *RCS* archives are present
17. *RCS rlog* information is informative

Chapter 7

Requirements

This assignment sketches out the basic requirements for the system your team is to deliver. Feel free to ask the "customer" for clarifications at any time.

7.1 Description

Simply stated, your system must be able to compute the performance, in terms of time and fuel consumption, for a cross country flight in a light plane, under VFR conditions. The flight will consist of

- taxi and run-up
- takeoff and climb to altitude
- progress to set heading point
- progress through one or more waypoints
- progress to descent point
- entry into pattern, landing, taxi

The time and fuel cost of each stage must be computed, and the legally mandated surplus fuel must be included. The computations must correctly account for wind, temperature, and pressure along the route. Totals for the trip must be provided as well incrementals for each leg. (Some examples will be provided for you, and a reference book will be put on reserve.)

7.2 Environment

Your system must be produced as a GP *application* (in the sense of the command *gpnewapplicaton*) and provided under `GP/src/Applications`. All classes (save possibly a few very specific ones with no general applicability) are to be developed in conformance to GP conventions and installed from `GP/src/Classes`. When delivered, it must be possible to bootstrap your system's executable from a *cold start*. By this it is meant that your *cleaned* `GP/src` tree can by hung onto a GP root directory that has no `src` directory, and that executing *gpmkmf; make compile; make install*

within `GP/src/Classes` and executing

gpmkmf; make compile; make

within `GP/src/Applications` will result in a correctly functioning executable.

7.3 Submission

Only one submission is to come from each team. Clean your `GP/src` directories. A uuencoded, compressed, tar file of the cleaned `GP/src` directory tree is to be mailed to the Instructor before the designated deadline. In hard copy you must also submit a user guide and a description of the uniform coding style that has been followed.

7.4 Objectives

The grade assigned will be based upon the following considerations:

1. GP conventions have been followed throughout
2. GP utilities have been used exclusively and appropriately for development
3. project installs from a cold start
4. code style is uniform throughout
5. code commenting is good
6. code is well crafted, concise, efficient, readable
7. all man pages are complete and informative
8. no extraneous *gpnew** elements remain
9. broad and good use of abstractions
10. abstractions effectively used for polymorphic code
11. implementations are hidden
12. coupling is limited and weak
13. class cohesion is high and classes have a limited, strong focus
14. public interfaces are well planned
15. protected interfaces are well planned
16. design emphasis on modifiability is evident

17. user interface is clean and easy to use
18. all expected aspects of the project are covered
19. calculations and results are acceptably correct
20. *RCS* archives are present and have been used effectively
21. *RCS rlog* information is informative
22. test legacy is comprehensive and complete
23. example programs have a tutorial and informative nature
24. user documentation is concise, helpful, well organized
25. an updated design document is provided

Chapter 8

Design

The first step in creating a good system is a good design, which has been carefully thought through in *advance* of any major efforts of implementation. This chapter provides guidelines for the first team assignment to be delivered during the course: the Design Plan.

8.1 Description

The text reminds us that design typically progresses from the informal and general to the formal and detailed, generally by a process involving cycles of analysis, proposal, review, and revision. The ultimate goal is to have a description of the system detailed enough so that individual programmers can take its parts and implement them independently without inconsistencies. The detail, however, should not be so extensive that the only work left to be done is a mere transcription into some programming language. This means that interface and data conventions must be clearly decided, and that functionality has to be understood. It does not mean that implementation has been written in stone.

Each system to be developed will have static aspects of design (subsystems, modules, etc.) and dynamic dynamic aspects of design (control flow, data management, etc.). In a top-down approach, each of these aspects will be structured according to some chosen models and principles. Static decomposition will produce subsystems, modules, data types. Dynamic decomposition will produce functional units, data flow, operations on types, data management. Object-oriented design may express the static and dynamic aspects together, with static detail given in terms of classes, hierarchies, and *is-a*/*has-a* relationships and dynamic detail expressed by methods and *uses-a* relationships. At the bottom level of design, attention will be paid to recognizing a consistent and complete infrastructure of data types and algorithms that can be reused, providing a common base of support for subsystems, modules, and functional units.

Design is considered good when cohesion is strong *within* a static entity (subsystem, module, hierarchy, class); that is, when data held in common is logically chosen, is streamlined, and is shared using effective, consistent functionality. Design is considered good when coupling is weak *between* static entities; that is, there are few interactions between them and these interactions exchange a minimum of data and information. Finally, striving toward smaller numbers of entities, smaller numbers of entity relationships, and more streamlined dynamic design is usually considered good.

The object of this assignment is to deliver the best design your team is capable of in the time allotted. The deliverable is the Design Plan document, consisting of text and illustrated by diagrams as necessary. Two aspects of this design will be looked at, the *form* and the *content*. The form will be judged by how presentable the document is, how well the document presents the design, and how well it indicates the reasoning behind choices made for static and dynamic structures. The content will be judged by how well the design itself appears to have been conceived, how well it compares with other designs delivered, and what prospects it appears to have of yielding a tight, efficient, adaptable, and maintainable system.

8.2 Submission

Hard copy is to be handed in at the beginning of the designated class. The pages are to be neatly and securely stapled together at the upper left hand corner. No special cover is necessary. The cover page should contain the names and IDs of all team members, with the responsible team member clearly designated.

8.3 Objectives

Grades will be assigned with consideration given to the objectives listed below. Note that there are objectives for *form* and objectives for *content*.

1. content: good component reuse; duplication avoided
2. content: effective infrastructure planning
3. content: static structuring shows good thought
4. content: dynamic/control structuring shows good thought
5. content: attention to simplicity and streamlining
6. content: entity interfaces minimal, logical, consistent
7. content: loose coupling between entities
8. content: good cohesion within entities
9. content: good naming conventions
10. content: classes/objects sensibly and effectively chosen
11. content: class hierarchies (is-a relationships) well designed
12. content: class composition (has-a relationships) clean
13. content: class interaction (uses-a relationships) sensible
14. form: formatting and layout good
15. form: good diagrams and effective use of diagrams

16. form: entities traceable; information indexed
17. form: readable, clear, concise
18. form: system structuring model(s)/principle(s) clearly indicated and presented
19. form: control structuring model(s)/principle(s) clearly indicated and presented
20. form: entities clearly indicated and presented
21. form: data-flow presentation, if any, contributes to understanding
22. form: processing narratives are clear, concise, understandable

Chapter 9

Testing

System implementation is most securely accomplished if the tests to be applied for verifying code correctness are designed integrally with the design of the code itself; that is, in *advance* of the actual writing of code. The strategy of implementation should always be a strategy that involves the coding of an entity, the testing of the entity in isolation, the integration of the entity with others, and the testing of the result. All tests should have been carefully planned to cover all aspects of functionality, and planning a test involves clearly establishing what the input should be and what results are correct for it. This chapter provides guidelines for the second team assignment: the Test Plan.

9.1 Description

The usual discussions of testing distinguish between *unit* tests and *integration* tests, between *verification* tests and *validation* tests, between *white box* and *black box* testing. The discussions also distinguish the strategies of *top down* and *bottom up* testing.

Unit testing is the testing of a code entity in isolation; for example, a single method of a class or a single, free-standing procedure. When the functionality of a larger entity depends on the working of two or more such units, the testing of the combination is called an integration test. This covers the very real possibility that two units can function quite correctly in isolation but fail in combination; for example, due to a flaw in the exchange of information and/or services between them. An entity is tested as a black box if the tests are merely designed to observe that representative input cases produce the expected results. A white box test, on the other hand, "looks inside" the entity. Such tests are design to exercise the various control paths and computations of the code itself. (Clearly, such tests have to be planned much later in the development process than would be true for black box tests, since the actual implementation code must be known.) All testing described above verifies that the code does what it was written to do. This may not be what the requirements demanded. It is sometimes easy to lose track of the original demands when one is several months along in the implementation. Thus a validation test is one that takes a finished system and ensures that it will satisfy the requirements.

The strategies of testing, top down or bottom up, have the same meaning as in design. In top down testing, the entire system in simple outline is written, with most or all subsystems represented by *stubs*. Little by little, functionality is added to the system by replacing stubs with functioning code entities (which, for a while, may have sub-entity stubs). Little by little the system is fleshed out from the large to the small. In bottom up testing, the fundamental entities of infrastructure are written first and tested in *scaffolding* code, which serves as temporary system glue. The system is built from the small to the large as more and more entities are integrated together through the inplementation of the final system code. These two strategies are rarely found in pure form. Parts of the system may be built from the bottom up and others from the top down.

Another term is often seen in discussions: *big bang testing*. Essentially this means that no testing is done on individual entities, only on the system as a whole (or perhaps, on major subsystems). This will be considered a bad thing in this course.

9.2 Submission

Hard copy is to be handed in at the beginning of the designated class. The pages are to be neatly and securely stapled together at the upper left hand corner. No special cover is necessary. The cover page should contain the names and IDs of all team members, with the responsible team member clearly designated.

9.3 Objectives

In addition to reasonable observation of formatting, layout, effective use of diagrams and text, the grade will depend on the clear presentation of the following issues of content:

1. testing proceeds in definite stages
2. the details of each stage are presented
3. the strategy for each stage is clear
4. input and expected output are given for black box tests
5. any white box testing is precisely described
6. unit tests and integration tests are distinct
7. validation tests are present for requirements
8. testing is comprehensive

Chapter 10

Scheduling

In order to complete a project under a deadline, a team must guage its *resources* (time and skills of team members, availability of computer services, etc.), it must estimate the time required to complete all necessary *milestones*, and then it must *allocate* its resources to the attainment of its milestsones and to the project's completion. Of equal importance, a team must *prioritize* its milestones so that, in the event of difficulty, a meaningful partial completion of the project is possible. A team must achieve its milestones through the completion of *tasks* that are sensibly designed and carried out under *review*. Areas of *responsibility* must be clear and cover all activities, and it is wise to make sure that someone can serve in a *backup* capacity for any assignment.

This chapter gives a few guidelines for the third team assignment: the Schedule Plan. Since this will be accomplished with the help of Microsoft Project, much of its format will be out of your hands, and we will only concentrate on general considerations.

10.1 Considerations

First, determine your resources. You have a team of only N people, and you have to get the job done by the due date. Establish with your team what considerations are to be taken into account. Does anyone have co-op interviews, midterms, family visits, projects in other courses, etc? Establish any and all constraints that limit your ability to work 24 hours a day, 7 days a week exclusively on the project.

Next, lay out the subgoals (tasks and milestones) that will lead to the completion of the project. These will be such things as the implementation of an entity, completion of a scaffolding, review of a phase of work, revision of a document, etc.

Prioritize your tasks so that you are assured that the most important goals will be met, if you find you won't be successful in meeting all your goals. Order tasks so that the important ones are done by the most appropriate individuals and so that there is a backup person for every critical task. Assign people to tasks and milestones.

Next, estimate times required for each task. Use summaries from past terms, pertinent historical material from the book, your recollection of how long assignments typically took, and educated guesses. You might select an estimation technique from the book that seems most relevant (e.g. so and so many person hours per functional unit, best-worst-median-time average, etc.). The outcome of this exercise should be a list of the expected duration for each task implied by the Design and Test Plans.

Finally, you must make assignments of work to people. How much in estimated time is supplied by each team member? Who is assigned to do what? What team members fill what roles? Who is the final authority on testing? On implementation? On scheduling? On style and form? On documenting? The time-task-person-resource data is the basic input to Microsoft Project (or any similar scheduling tool). The outcome is a time-dependency chart and layout of time lines for each task and individual that provides the schedule for project completion. An important byproduct of these charts is a *critical path* that is threaded through the tasks whose slippage will impact the final completion of the project. All tasks and milestones on the critical path are to be monitored very closely.

After charts are completed and tasks and milestones are ordered in time, you will need to establish tracking and control mechanisms What regular schedule of meeting and review will you use? What communications channels will you use to keep each other up to date on progress? What provisions are made for emergencies?

10.2 Comments

I'll provide a general sketch of the best and worst characteristics of schedule plan documents from past terms.

Best

Some careful estimate of times, perhaps using something from the book. Very thorough layout of each team member's available times. Selection of milestones so that integration, testing, communication and review, and documentation goes hand in hand with development, thereby keeping the development honest and well-tracked. There is a meeting schedule planned, including an indication of the agenda to be used. Task chart rich, detailed (but not excessively detailed), and with cross dependencies among separate members so that there is careful checkpointing. Prioritization of tasks so that lesser ones can be sacrificed. Diffuse development, which avoids "killer" nodes that bottleneck the development and ruin things if they fail. Careful assessment of risks. Thorough charts and timelines, explicitly stating dates. Managerial establishment of means of communication and a designated central directory on the Unix machines to facilitate up-to-the-minute sharing. Responsibilities assigned according to individual strengths and availabilities.

Worst

Design consists of a small number of very large tasks or a huge number of picky little tasks. Everybody gets an equal number of tasks and no justification is given for the assignments. An unreasonably large number of hours is assigned to each person, and task-time estimation is poorly thought out. Everybody goes off and does their own thing independently without any clear lines of oversight or management responsibility. No files are communally available. There is a sort of a meeting every week in which each team member is supposed to "report progress", but (since everybody is acting independently and no testing and verification is done until the end) progress can't be reliably assessed. Everything is expected to come magically together in one great big wonderful "integration" step a couple of days before the due date.

10.3 Submission

Hard copy is to be handed in at the beginning of the designated class. The pages are to be neatly and securely stapled together at the upper left hand corner. No special cover is necessary. The cover page should contain the names and IDs of all team members, with the responsible team member clearly designated. The hard copy will be produced by Microsoft Project and will consist of a critical path chart and time lines for individual team members. Also to be included are a summary of the team resources, of the management, responsibility, and control mechanisms decided upon, and of the prioritization considerations that will provide fallback positions for meaningful partial fulfillment of the project.

10.4 Objectives

The document should include a presentation of the issues of resource assessment, task breakdown, prioritization and fallback, responsibility and authority, communication and coordination, review, backup. It should then provide the PERT, Gantt, and Resource Usage charts from MicroSoft Project.

1. team member strengths/skills assesses
2. team member time/availability inventoried
3. team member time resources not expected in excess
4. task size/granularity is reasonable
5. task assignments justified
6. task times sensibly estimated
7. management/review responsibilities clear and complete
8. testing tasks included
9. implementation tasks included
10. preparation/infrastructure tasks included
11. communication/coordination/review tasks included
12. documentation tasks included
13. activities carried on reasonably in parallel
14. backup provided
15. tasks structured and prioritized
16. partial system completion evident and possible
17. task sequencing accounts for priority and partial completion

18. uniformity is enforced and controlled
19. reasonable milestones at reasonable intervals
20. justifications are given, are reasonable, and show thought

Chapter 11

Project Delivery and Maintenance

The fourth team assignment is, simply, a completed system, consisting of well documented code, and a short user manual in hard copy. This system must compile and execute successfully from a "cold start" in the Instructor's GP environment on an SGI machine. The objectives to be considered are as stated at the end of Chapter 7.

11.1 Delivery

When the finished projects have been received, the Instructor will provide a simple enhancement to be made, and he will issue projects to random teams for this maintenance task. The Maintenance Document, which is the last team assignment to be submitted, is to be a concise sequence of items reports on

- any problems discovered in the system received
- the steps taken to correct these problems
- the steps takn to achieve the enhancements
- the validation that the enhancement has been achieved

together with a professional commentary on the quality of the system being maintained and how well your team believes it has met the objectives of Chapter 7. The modified system, which must also compile and execute successfully from a cold start, is to be returned to the Instructor along with the Maintenance Document. This constitutes the fifth and last team assignment.

11.2 Objectives

The objectives to be achieved in the maintenance phase are succinct:

1. a fair, professional, mature commentary is given for the system maintained
2. the maintenance goals are achieved (or a convincing case is presented why they cannot be)
3. the sequence of items mentioned above are presented completely and concisely
4. the modified system works as required from a cold start (if modification was possible)

Chapter 12

Things Go Wrong

A Case History

Date: Tue, 26 Apr 94 08:42:32 EDT
From: Jerry Leichter <leichter@lrw.com>
Subject: Stress Analysis of a Software Project [long]

The following, which claims to be an internal Silicon Graphics memo, has already seen fairly broad network distribution. I have no way of verifying that it is what it claims to be, but (a) I'm told by someone with close dealings with SGI that it fits with what he's heard; (b) if it's a fake, someone put a huge amount of effort into producing it.

I forward it to RISKS as a wonderful record of what goes wrong with large software projects, and why. It would be as useful if all the names, including the company and product names, were removed. This memo should not be seen as an indictment of SGI, which is hardly unique. There is good evidence that Sun, for example, had very similar problems in producing Solaris; and I watched the same thing happen with the late, unlamented DEC Professional series of PC's, and something like it almost happen with firmware for DEC terminals a number of years back.

I hope that Tom Davis's position hasn't been badly hurt by the broad distribution of his memo - but based on the traditional reaction to bearers of bad news, especially when the bad news becomes widely known, I can't say I'm sanguine about it.
– Jerry

Software Usability II
October 5, 1993
Tom Davis

Last May, I published my first report on software usability, which Rocky Rhodes and I presented to at Tom Jermoluk's staff meeting (with Ed, but without Tom). Subsequently, I made it available to quite a few other people.

This sequel is to satisfy all those people who have urged me to bring it up to date. I begin with a summary; details follow.

Please read at least the summary.

SUMMARY

Release 5.1 is a disappointment. Performance for common operations has dropped 40% from 4.0.5, we shipped with 500 priority 1 and 2 bugs, and a base Indy is much more sluggish than a Macintosh. Disk space requirements have increased dramatically.

The primary cause is that we attempted far too much in too little time. Management would not cut features early, so we were forced to make massive cuts in the final weeks of the release.

What shall we do now? Let's not look for scapegoats, but learn from our mistakes and do better next time.

A December release of 5.1.2 is too early to fix much – we'll spend much more time on the release process than fixing things. Allow enough time for a solid release so we don't get: 5.1.2.1, 5.1.2.2, 5.1.2.3, ...

Let's decide ahead of time exactly what features are in 5.1.2. If we pick a reasonable set we'll avoid emergency feature cuts at the end.

Nobody knows what's wrong – opinions are as common as senior engineers. The software environment is so convoluted that at times it seems to rival the US economy for complexity and unpredictability. I propose massive code walk-throughs and design reviews to analyze the software. We'll be forced to look closely at the code, and fresh reviewers can provide fresh insights.

For the long term, let's change the way we do things so that the contents and scheduling of releases are better planned and executed. Make sure marketing and engineering expectations are in agreement.

INTRODUCTION

We've addressed some of the problems presented in the original May report, but not enough. Most of the report's warnings and predictions have come true in 5.1. If we keep doing the exact same thing, we'll keep getting the exact same results.

I'm preparing this report in ASCII to make it widely available. It's easy to distribute via news and mail, and everyone can read it.

An ASCII version of the May 12 report can be found in:

```
bedlam.asd:/usr/tmp/report.text
```

The included quotations are not verbatim. Although the wordings are inexact, I believe they capture the spirit of the originals.

BLOAT UPDATE

> "Do you want to be a bloat detective? It's easy; just pick any executable. There! You found some!"
>
> – Rolf van Widenfelt

In the May report, I listed a bunch of executable sizes, and pointed out that they were unacceptable if we intended to run without serious paging problems on a 16 megabyte system. Between May and the 5.1 release, many have grown even larger. IRIX went up from 4.8 megabytes to 8.1 megabytes, and has a memory leak that causes it to grow. Within a week, my newly-booted 5.1 IRIX was larger than 13.8 megabytes – a big chunk of a 16 megabyte system. It's wrong to require our users to reboot every week.

There are too many daemons. In a vanilla 5.1 installation with Toto, there are 37 background processes.

DSOs were supposed to reduce physical memory usage, but have had just the opposite effect, and their indirection has reduced performance.

Programs like Roger Chickering's "Bloatview" based on Wiltse Carpenter's work make some problems obvious. The news reader "xrn", starts out small, but leaks memory so badly that within a week or so it grows to 9 or 10 megabytes, along with plenty of other large programs. But what's really embarrassing is that even the kernel leaks memory that can't be recovered except by rebooting!

Showcase grew from 3.2 megabytes to 4.0 megabytes, and the master and status gizmos which are run by default occupy another 1.7 megabytes. Much of this happened simply by recompiling under 5.1 – not because of additional code.

The window system (Xsgi + 4Dwm) is up from 3.2 MB to 3.6 MB, and the miscellaneous stuff has grown as well. As I type now, I have the default non-toto environment plus a single shell and a single text editor, jot. The total physical memory usage is 21.9 megabytes, and only because I rebooted IRIX yesterday evening to reduce the kernel size. Luckily, I'm on a 32 megabyte system without Toto, or I'd be swamped by paging.

Much of the problem seems to be due to DSOs that load whole libraries instead of individual routines. Many SGI applications link with 20 or so large DSOs, virtually guaranteeing enormous executables.

In spite of the DSOs, large chunks of Motif programs remain unshared, and duplicated in all Motif applications.

PERFORMANCE UPDATE

> "Indy: an Indigo without the 'go' ".
> – Mark Hughes (?)
>
> "X and Motif are the reasons that UNIX deserves to die."
> – Larry Kaplan

The performance story is just as bad. I was tempted to write simply, "Try to do some real work on a 16 megabyte Indy. Case closed.", but I'll include some details.

In May, I listed some unacceptable Motif performance measurements. Just before 5.1 MR, someone reran my tests and discovered that the performance had gotten even worse. Some effort was expended to tune the software so that instead of being intolerable, it was back to merely unacceptable performance.

We no longer report benchmark results on our standard system. The benchmarks are not done with the DSO libraries; they are all compiled non-DSO so that the performance in 5.1 has not declined too much.

Before I upgraded from 4.0.5 to the MR version of 5.1, I ran some timings of some everyday activities to see what would happen. These timings were all made with the wall clock, so they represent precisely what our users will see. I run a 32 megabyte R4000 Elan.

```
Test                 4.0.5          5.1          % change
----                 -----          ---          --------


C compile of a       25 sec         35 sec          40%
small application
```

C++ compile of a small application	68 sec	105 sec	54%
Showcase startup, May report file	13 sec	18 sec	38%
Start a shell	<2 sec	~3 sec	~50%
Jot 2 MB file	<2 sec	~3 sec	~50%

What's most frightening about the 5.1 performance is that nobody knows exactly where it went. If you start asking around, you get plenty of finger-pointing and theories, but few facts. In the May report, I proposed a "5% theory", which states that each little thing we add (Motif, internationalization, drag-and-drop, DSOs, multiple fonts, and so on) costs roughly 5% of the machine. After 15 or 20 of these, most of the performance is gone.

Bloating by itself causes problems. There's heavy paging, there's so much code and it's so scattered that the cache may as well not be there. The window manager and X and Toto are so tangled that many minor operations like moving the mouse or deleting a file wake up all the processes on the machine, causing additional paging, and perhaps graphics context swaps.

But bloat isn't the whole story. Rocky Rhodes recently ran a small application on an Indy, and noticed that when he held the mouse button down and slid it back and forth across the menu bar, the (small) pop-up menus got as much as 25 seconds behind. He submitted a bug, which was dismissed as paging due to lack of memory. But Rocky was running with 160 megabytes of memory, so there was no paging. The problem turned out to be Motif code modified for the SGI look that is even more sluggish than regular Motif. Perhaps the problem is simply due to the huge number of context swaps necessary for all the daemons we're shipping.

The complexity of our system software has surpassed the ability of average SGI programmers to understand it. And perhaps not just average programmers. Get a room full of 10 of our best software people, and you'll get 10 different opinions of what's causing the lousy performance and bloat. What's wrong is that the software has simply become too complicated for anyone to understand.

WHAT WENT WRONG IN 5.1?

The one sentence answer is: we bit off more than we could chew. As a company, we still don't understand how difficult software is.

We planned to make major changes in everything – a new operating system, new compilers, a new user environment, new tools, and lots of new features in the multi-media area. Not only that, but the new stuff was promised to do everything the old software had done, and with major enhancements. (Early warning: version 6.0 promises to be even more disruptive.)

About 9 months ago, Rocky and I pointed out the impossibility of what we were attempting. Rather than reduce the scope of the projects, a decision was made to hire a couple of contractors (who know nothing about our system) to handle the worst user interface problems in the Roxy project. In addition, promises were obtained from various executives that a significant effort would be made to improve software performance.

Management was basically afraid to cut any features, so we continued to work on a project that was far too large. The desperate attempt to do everything caused programmers to cut corners, with disastrous effects on the bug count. And the bug count was high simply because 5.1 was so big.

Only when the situation was beyond hope of repair did we start to do something. Features and entire products were removed wholesale from the release, and hundreds of high-priority bugs were classified as exceptions, so that we could ship with "no priority 1 and 2 bugs". We did, however, ship with over 500 "exceptions". The release was deemed too crummy to push to all our machines, but was restricted to the Indys, the high-end machines, and a few others where new hardware required the new software. Due to the massive bug count, virtually no performance tuning was done.

When the schedule is impossible as it was in 5.1, the release process itself can get in the way. The schedule imposes a code freeze long before the software is stable, and fixing things becomes much more difficult. If you know you're going to be late, slip before the code freeze, not after. We're trying to wrap up the box before the stuff inside is finished, and then trying to fix things inside the box without undoing the wrapping – it has to be less efficient.

Management Issues:

There was never an overall software architect, and there still is not, and until Way Ting was given the job near the end, there was no manager in charge of the 5.1 release, either.

I wrote a note in sgi.bad-attitude about the "optimist effect", which I believe is mostly true. In condensed form:

> Optimists tend to be promoted, so the higher up in the organization you are, the more optimistic you tend to be. If one manager says "I can do that in 4 months", and another only promises it in 6 months, the 4 month guy gets the job. When the software is 4 months late, the overall system complexity makes it easy to assign blame elsewhere, so there's no way to judge mis-management when it's time for promotions.
>
> To look good to their boss, most people tend to put a positive spin on their reports. With many levels of management and increasing optimism all the way up, the information reaching the VPs is very filtered, and always filtered positively.

The problem is that the highly filtered estimates are completely out of line with reality (at least in recent software plans here at SGI), and there are no reality checks back from the VPs to the engineers on the bottom. I think it's great to have aggressive schedules where you try to get things out 20% or so faster than you'd expect. The problem is that in 5.1, the engineers were expected to get things out 80% faster, and it was clearly impossible, so many just gave up.

We certainly didn't win any morale prizes among the engineers with 5.1. It's the first release here at SGI where most of the engineers I talked to are ashamed of the product. There are always a few, but this time there were many. When engineers were asked to come in over the weekends before the 5.1 release to fix show-stopper bugs, I heard a comment like: "Why bother? SGI's going to release it anyway, whether they're fixed or not."

I'm not blaming the engineers. Most of them worked their hearts out for 5.1, and did the best they could, given the circumstances. They'll be happy to buy into a plan where there's a 20% stretch, but not where there's an 80% stretch. They figure: "It's hopeless, and I'll be late anyway, and I'm not going to get rewarded for that, so why kill myself?"

Marketing - Engineering Disconnect

"Marketing – where the rubber meets the sky."

– Unknown

There's a disconnect between engineering and marketing. It's not surprising – marketing wants all the whiz-bang features, it wants to run in 16 megabytes, and it wants it yesterday. Although engineering would like the same things, it is faced with the reality of time limits, fixed costs, and the laws of nature.

It's great to have pressure from marketing to do a better job, but at SGI, we often seem to have deadlocks that are simply not resolved. Marketing insists that Indy will work in 16 megabytes and engineering insists that it won't, but both continue to make their plans without resolving the conflict, so today we're shipping virtually useless 16 MB systems. Similarly for feature lists, reliability requirements, and deadlines.

Well, at least we met the deadline.

WHAT TO DO – SHORT TERM (5.1.2)

> "We should sell 'bloat credits', the way the government sells pollution credits. Everybody's assigned a certain amount of bloat, and if they go over, they have to purchase bloat credits from some other group that's been more careful."
>
> – Bent Hagemark

There are problems in both performance and bugs, and we'd like to fix both. In addition, the first thing we should do is decide exactly what's going into release 5.1.2.

If we are serious about a December all-platforms release, there may be very little we can do other than keep stumbling along as we have been. Three months isn't much time to do anything, considering the overhead of a release, where perhaps half of the time will be spent in "code freeze". After 5.1, many engineers are exhausted, and it's unreasonable to expect them to start hard work immediately. 500 outstanding priority 1 and 2 bugs is a huge list, and we haven't even begun to hear about customer problems yet.

What Should be in Release 5.1.2:

I'm afraid the answer is going to be "everything that didn't make it into 5.1". I know that won't be the case, but I hope that we will carefully select what goes in now, rather than hack things out in a panic in December. The default should be "not included", and we should require a good reason to include things. Let's make sure that there's a minimal, solid, working set before we start adding frills.

Improving Performance:

> "SGI software has a cracked engine block, and we're trying to fix it with a tune-up."
>
> – Mark Segal

As stated above, we don't even know exactly what's wrong. We probably never will, but we should start doing things that will have as much of an impact on the problem as possible. I don't think we have time to study the problem in detail and then decide what to do – we've got to mix the research with doing something about it.

Before we begin, we should have definite performance goals – lose less than 5% wall-clock time on compiles of some known program over 4.0.5, have shells come up as fast as in 4.0.5, or whatever.

Some people claim that we need new software debugging tools to look at the problem, and that may be true, but it's not a short-term solution, and it runs the risk of causing us to spend all our time designing performance measurement tools, rather than fixing performance.

In fact, I don't really believe that simple "tuning" will make a large dent. To get things to run significantly faster, we've got to make significant changes. And we can't beat the "5% rule" by just speeding up all the systems by 5% – if everything is exactly 5% faster, the overall system will be exactly 5% faster.

There's a strong tendency to look for the "quick fix". "Get the code re-arranger to work", or "Put all the non-modifiable strings in shared code space", for example. These ideas are attractive, since they promise to speed up all the code, and they should probably be pursued, but I think we're not going to make a lot of progress until we identify the major software architectural problems and do some massive simplification. Remember that DSOs were the last "quick fix".

There's got to be more to it than tuning; there must be some amazingly bad software architecture – from a novice's point of view, a 4 MB Macintosh runs a far more efficient, interesting system than a 16 MB Indy. The Mark Segal quote above sums it up.

Code walk-throughs and design reviews are in order for most of our software. The attendees should include not only people working in the same area, but a small cross-section of experienced engineers from other areas. Get a pool of, say, 20 experienced engineers and perhaps 3 at a time would sit in on code reviews together with the other people working in that area.

Code reviews will help in many ways – the engineer presenting the code will have to understand it thoroughly to present it, others will learn about it, and outside observers will provide different ways to look at the problems.

The most important thing should be the focus – we're trying to make the code better and faster, not to make it more general, or have new features, or be more reusable, or better structured.

For complex problems, the walk-through should also include some general design review. Are these daemons really necessary? Do we really need this feature? And so on.

Fixing Bugs:

The code walk-throughs will obviously tend to turn up some bugs, so they'll serve a dual purpose.

With 500 or so priority 1 and 2 bugs, we must prioritize these as well. A bug that causes a system crash only on machines with some rare hardware configuration is properly classified priority 1, but it's probably less important than a bug in a popular program like Showcase that causes you to lose your file every tenth time, which would normally rank as priority 2. The effort involved in the fix should also be taken into account. For bugs of equal frequency of occurrence, it's probably better to fix 20 priority 2 bugs than 1 priority 1 bug if the priority 2 bugs are 20 times easier to fix.

A bunch of bugs can be eliminated by getting rid of features. Let's have the courage to cut some of the fat.

WHAT TO DO – LONG TERM

"Software quality is not a crime."

– Unknown (seen on a poster in building 7)

It's easy to go on forever here, but I'll try to limit it to a few key ideas. We don't have to do all these at once, but we'd better start.

- Have an overall SGI software plan. Let's get an architect, or at least a small group of highly technical people, not just managers, to agree on plans for releases. In fact, since the release is a company-wide project, there ought to be company-wide participation in the decisions of what's in a release. The group should include marketing, documentation, engineering, and management and should come up with a compromise that's reasonable to all.

- In every case, some attempt must be made to check reasonableness all the way to the bottom. There's a long series of excuses, "Well, that's what my junior VPs told me.", or "That's what my directors/managers/lead engineers/engineers told me." We get killed by the optimist effect, and a disinclination to listen seriously to anyone but our direct reports. Try to imagine the guts it takes for an engineer to go to his director and say: "My manager's out of his mind – I can't possibly do what he's promised."

- Let's try to concentrate on performance and quality, not on new features, especially for the 5.1.2 release. I know from my own experience that when I write good code, I spend 10% of the time adding features, and 90% debugging and tuning them. It's the only way to make quality software. In SGI's recent releases, the opposite proportions are often the rule. It's much easier to add 100 really neat features that don't work than to speed up performance by 1%.

- Aim for simplicity in design, not complexity. Make a few things work really well; don't have 1000 flaky programs.

- Be willing to cut features; who's going to be more pissed off: a customer who was promised a feature that doesn't appear, or the same customer who gets the promised feature, and after months of struggling with it, discovers he can't make it work?

- Get better agreement between the top level VPs and the lowest engineers that a given schedule is reasonable.

- For new development, continue the formal design reviews and code walk-throughs. These shouldn't just happen once in the development cycle – things are bound to change, and code reviews can be very valuable, even for our experienced programmers.

ACKNOWLEDGEMENTS

I take full responsibility for the opinions contained herein, but I'd like to thank Mark Segal, Rosemary Chang, Mary Ann Gallager, Jackie Neider, Sharon Fischler, Henry Moreton, and Jon Livesey for suggestions and comments.

Date: Fri, 29 Apr 1994 19:35:15 GMT
From: wombat@kilimanjaro.engr.sgi.com (Joan Eslinger)
Subject: Re: Stress Analysis of a Software Project (Davis/Leichter R-15.80)

The memo Jerry Leichter posted was an actual Silicon Graphics memo. However, life for Silicon Graphics and Tom Davis is not quite so bleak as some might think. Tom Davis wrote the original memo to point out problems and ask everyone to help fix them.

It was very effective. I installed a beta version of the new 5.2 release on my Indy in January, and only rebooted the machine a couple of weeks ago because I was moving to another building. Sure, I had to add another swap file on-the-fly about once a month because my emacs processes grew so large :-), but the system did not crash. And performance is quite snappy. "Watch the skies."

Since the memo has been popping up all over the net, Tom has written a reply to it, included below. There isn't really a RISKS tie-in, unless you count the risk of having only the "bad" half of a story get wide distribution.

Joan Eslinger / Silicon Graphics / wombat@sgi.com

-+-

I am the author of the original memo below, which was intended for internal Silicon Graphics use only, and was not for anyone outside the company. But since it has been leaked to the net, and is beginning to be used by competitor's sales people, I feel a response is required.

I don't believe that these problems are unique to Silicon Graphics. From discussions with friends who are insiders in many different companies, I am certain that similar memos could be written about the software of each of our competitors.

What I like about working for Silicon Graphics is that at least here, something is being done about it – I worked for companies in the past where the response would have been to stick our heads in the sand in hopes that the problems would just fix themselves. If I hadn't thought that the memo would catalyze some change here, I wouldn't have written it.

The details appear as comments to my original article below. Luckily, the article is 6 months old, and we have had a chance to make some significant progress.

Typically, what happens is that each faster generation of hardware is followed by software that more than compensates for the increased speed, but as a result of this memo, Silicon Graphics has been able to skip one of the slowing software cycles, making, instead, a performance and quality based release. The next release is going to be similar, and in the meantime, we get an extra hardware boost from the faster R4600 processors.

– Tom Davis
Silicon Graphics

General comments:

As a fairly direct result of this memo, SGI decided not to continue "business as usual" in software development. The approach we took to the problem was the following:

With the 4.0.5abcdefghi... fiasco, and the fact that the 5.* releases were still for specific machines, our developers were desperate for an all-platforms release. We decided to make such a release relatively soon – and 5.2 actually MRed in February. The 5.2 release had two goals – primarily, all-platform, and given that it went out in February, do as much performance-tuning and bug-fixing as time allowed.

In that period, the performance on 16MB systems was essentially doubled, which improved performance on larger systems as well, but to a lesser degree. Significant numbers of bugs were fixed as well.

Some people hoped that a few quick fixes would bring back all the performance in 5.2, but a little investigation indicated that was a list of things to be done, and that another quality release would be required.

The 5.3 release, not officially scheduled, but which should be MRed around October or November is that quality (performance and bug-fix) release. We'll add a few new features, but they will be the exception rather than the rule. The longer time before the 5.3 release should give us time to do a thorough job of solving our problems.

For 5.3, there's also time to set up solid performance and bug-fixing goals, and these are already being discussed.

And most important – the worst problems were with 16 MB systems that paged their brains out. They are better now, but not great. But we don't sell them. One of the 5.3 goals is to improve performance (or reduce sizes enough) that it will be acceptable on a 16 MB machine.

The kernel memory leaks are all fixed, and many of the important programs have been reduced in size. For 5.2, 5 or 6 of our most heavily-used programs were subjected to close scrutiny to find out where the performance went, and many were significantly improved.

A lot more work is planned for 5.3 to reduce the sizes of the executables.

Work is continuing on the DSOs to split them up properly so that they don't all have to be loaded, and to improve their performance and start-up time. We're working to make "quick-starting" happen more automatically.

PERFORMANCE UPDATE

I don't think it's unusual to do benchmarks with non-standard compiler settings. Both we and our competition have done so for a long time. We do ship all the libraries, et cetera, necessary to duplicate these results so customers for whom speed is the only objective can pay the cost of larger executables in exchange for the added speed.

Unfortunately, I can't re-run some of these tests, but 5.2 is definitely better than 5.1.

I think the 5.1 fiasco has caused a lot of our management to see the light, and in conversations with people at all levels, it's clear that nobody wants to see anything like it happen again. The 5.2 and future 5.3 releases seem to be steps in the right direction.

But there's still a lot of work to do, and we in engineering can use every minute between now and the 5.3 release to improve things.

The 5.3 release is being planned with reasonable beta-cycles, and with enough time between now and "code freeze" to make significant improvements.

Management Issues:

I think this sort of disconnect is not too unusual – there is always enormous pressure to announce a very low entry price-point, and the 16MB system provided that. Everybody does this with the full knowledge that on a minimum system, you won't be able to run many interesting applications, and almost everyone will have to purchase a bit more memory. It's just that in the case of Indy, there were so many new features that the proposed minimal system was embarrassingly slow.

The "fix" is simply not to ship the 16MB systems which will insure that everyone will get a very usable machine. All we really lose is our low entry price point, and the gain is that we won't have to deal with the few irate customers who bought a minimal system.

Although some of our performance loss is due to more complicated features, the vast majority is due to the fact that more memory is required, and without it, the systems page with a consequent dramatic reduction in performance. The 4.0.X -> 5.X change on our large machines was measurable, but not nearly so noticeable as on the smaller ones.

We're still not completely there (as far as I can tell) with respect to better software management. The good thing is that many of our higher-level managers are acutely aware of the problem now – Forest Baskett and Tom Jermoluk are extremely concerned, for example.

It's too bad it took a shock like 5.1 to make everybody take notice, but at least they did, and we're doing the right sorts of things to correct it.