CMPT-370
INFORMATION SYSTEMS DESIGN

INSTRUCTOR: RUSSELL TRONT

Semester 02-1

1. INTRODUCTION

Information systems, and most text books on information systems analysis and design, are oriented largely toward business information systems. This is largely because most information systems have, or are being created to support business information needs. Examples are the computer systems to support functions like inventory, payroll, accounts receivable, etc. Though each of these functions are becoming fairly well understood, nonetheless each company has slightly different ways of doing things and thus the design and/or customization of software for a particular situation requires considerable care. Since even users and corporate management (let alone computing science majors without a minor in business), have a hard time determining the requirements, it is still a very costly and error prone process to develop such systems. And thrown on top of this is the new move to implementation via distributed, client/ server architected systems.

Also, in the ever diversifying applications of computer systems, strange new applications which are basically information systems are showing up every day, even though they are not necessarily 'business' information systems. For instance, just because one has to interface with radar, and use sine and cosine to track aircraft, an national air traffic control system is nonetheless basically an distributed information system! So are systems that keep track of an airline's maintenance, so are loading gate reservation systems at airports for the airport authority, or systems which handle baggage routing for all the airlines behind the scene at an airport. In addition, often these various partially-related systems need to interact via the exchange of information!

Computing Science 370 is basically a course in the analysis and design of information systems. In this regard, it is an expansion of that part of the Cmpt 275 course.

In the past in Cmpt 370, we studied and contrasted several different methods and methodologies of analysis and design (in particular, structured, information engineering, and object methodologies). It was pointed out where they differ, where they are similar, and which kind of project each is most appropriate to. Starting in Semester 02-1, we will be more closely concentrating on the Unified Modelling Language and its associated methodology. And we will look at the concept of Design Patterns.

In addition to Analysis, Design (including User Interface Design), we will also be looking at system deployment, at the importance of a corporate information strategy, and at the problems of re-engineering existing systems.

1.1 Course Overview

1.1.1 Objective/Description

This course is intended to provide an integrated overview of the analysis and design of information systems while taking into account the realities of the software lifecycle. It points out the value and longevity of enterprise-wide data, and shows how to model and design a good application around this data. The role of specification techniques and methodologies is examined, along with design patterns, and will examine the opportunity for some automatic code generation by CASE tools, data base management systems, and 4GLs. A series of assignments focus the students on some of the techniques, and on the advantages and complexities of using sophisticated CASE tools.

1.1.2 Topics

- Analysis and Role of Analysts:
  - look, gather, and check before you leap.
- Design and the Role of Designers
- The Value and Longevity of Enterprise-Wide Data:
  - why systems should be structured around retained data objects
  - corporate technical and political realities.
- Gathering Analysis Information: - read, ask, sample, record
- Basic Data Modelling:
  - synthesizing a model of the present system, and designing the nature of the new system.
  - objects, relationships, normalization, and formalization.
- Unified Modelling Language
- The 4 Models of an Application System:
  - WHAT: data modelling via EER diagrams,
  - WHERE: application message architecture using Object Communication Diagrams,
  - WHEN: dynamic modelling - control specs using events, state machines, and process activation.
  - HOW: DFDs.
- Major Classes of Analysis and Design Methodologies:
  - Structured Analysis, Information Engineering, Object-Oriented Analysis: how they are similar and how they relate to the models
- Advanced Data Base Design
  - unusual structures, space usage, safety, location, and performance.
- External Design:
  - the draft user manual: input, output, functions and exceptions.
- System Architectural - bridging the gap to implementation.
- Code Generation:
  - mostly automatic if the analysis and architectural models are definitive
- Installation, Training, and Cut-over to the New System.

1.1.3 Cmpt 370 Course Administrative Details

a) Prerequisites: CMPT 275 and CMPT 354.
   - A minimum grade of C- is required in all prerequisite courses.

b) Grading: Assignment 30%, Midterm 20%, Final 50%.
   - Students must attain an overall passing grade on the weighted average of the exams in the course, in order to obtain a C or better.

c) Assignments: Will likely be entail the creation of a Requirements Spec, a UI design, an Architectural Design using UML, and object state machines for an example application. Short essays may also be required.
   - This will require the use of a good wordprocessor, and a course-supplied CASE tool.
   - The assignments will likely be done in groups of 4, although the exact size of the groups has not been decided yet.

d) Textbook: "Requirements Analysis and System Design: Developing Information Systems with UML" by Leszek Maciaszek, Addison Wesley, 2001. This is not a perfect book, but it nicely encompasses both information systems design and also UML. You should read Chapters 1 and 3 immediately.

e) Extensive lecture notes and other assigned readings will be used.
   - Lecture notes will be available from the course web site:
     `http://www.cs.sfu.ca/CC/370/tront/`
   - There will be a lot of reading in this course. Some of it will be from chapters in various other books that will be put on reserve in the library, or from magazine articles which will be put in the 'Cmpt 370 Extra Reading Binder' in the Cmpt 370 section of the reserve room.
   - Books put on reserve are designated as being for special short term loans of 2 hours, 4 hours, 24 hours, or 3 days. Note that 4 hour loans can be taken out overnight after 6 P.M.

f) You will need 4 things to do the assignments:

   1) A campus Unix computer system account (ID and Password). These are available to every student registered in the university. You can pick them up at Academic Computing Services in the Strand administration building, Room 1001.

   2) You will need a home directory on the Computing Science Instructional Laboratory CPUs (e.g. gemini.csil.sfu.ca). This should

be set up automatically after you activate your campus account. Note that we will be using the CSIL Sun Workstations to run a CASE tools called Rational Rose. More information on the CSIL Unix machines is available at www.cs.sfu.ca/CC/Labs. Note that if you have never used a Sun Workstation, you should seriously consider going to some of the tutorials on various Unix tools that will be provided by the School of Computing Science in the next few weeks.

3) You will need a security access card to the lab where the Sun Workstations are located. More information is available at http://www.cs.sfu.ca/CC/Labs/

4) Most students prefer to use the CSIL network printers rather than the Academic Computing Services (ACS) ones, so you must add money to a Computing Resource Charges (CRC) account using a form available from the Computing Science department office.

Get your Unix ID right away. This will give you access to important e-mail that I send to the whole class. Documentation on how to use e-mail is also available in Strand building room 1001, and online.

Information on how to get the other things is mostly online. See: http://www.cs.sfu.ca/CC/Labs/ and look for CSIL UNIX LAB.

There will be more information on the Rational Rose UML CASE tool available in the lab and the library. There will be online manuals and help from within the CASE tool.

In both the Reserver Room of the SFU Library, and also cabled to the work table in the lab will be copies of a book called "Visual Modeling with Rational Rose and UML" by Terry Quatrani [Quatrani98].

1.1.4 Contacts

a) The Instructor's office is ASB 10840.

b) My email is tront@cs.sfu.ca

c) Please feel free to drop by or call anytime (office: 291-4336 - includes answering machine). If you want to be sure to reach me, either make an appointment by phone or email, or drop by my regular office hours:
   -
   -
   -
   -
   -
   -

d) I will later announce the Teaching Assistant's
   - Name:
   - Office Hours:
   - Office Hours Location: ASB 9804 (CSIL Unix).
   - E-mail:

The TA will mark most of the assignments, part of the exams, and help you with any detailed problems with the software systems used in the course. I also can help, and would like to know about any problems you have with the software that need reporting to the developer.

1.1.5 Cheating and Abuse of Computer Privileges:

- Academic Honesty plays a key role in our efforts to maintain a high standard of academic excellence and integrity. Unfortunately, we are regularly provided opportunities to demonstrate our ability to detect cheating and identify violators. Students are advised that ALL acts of intellectual dishonesty are subject to disciplinary action by the School and that serious infractions will be referred to the President for further sanctions. Students are encouraged to obtain a copy of the School's "Statement on Intellectual Honesty".
- Any abuse of computer privileges can result in revocation of those privileges, even if an assignment is due the next day!

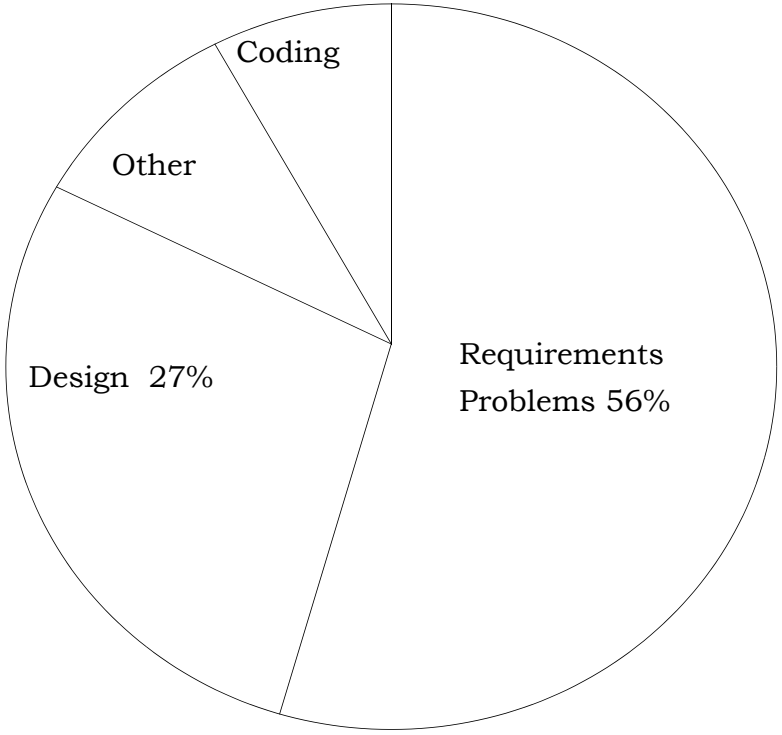
1.2 Analysis and Analysts

In this section, analysis will be explained and justified. Then, since it is a unique task, we will describe the difficulties of the special role played by the analyst.

1.2.1 Why Do We Need To Do Analysis?

This is easy to answer. Based on studies given in [Demarco 82] (and others), the source of errors/problems with large software systems can be broken down as follows:

FIGURE 1. - Source of Errors/Problems

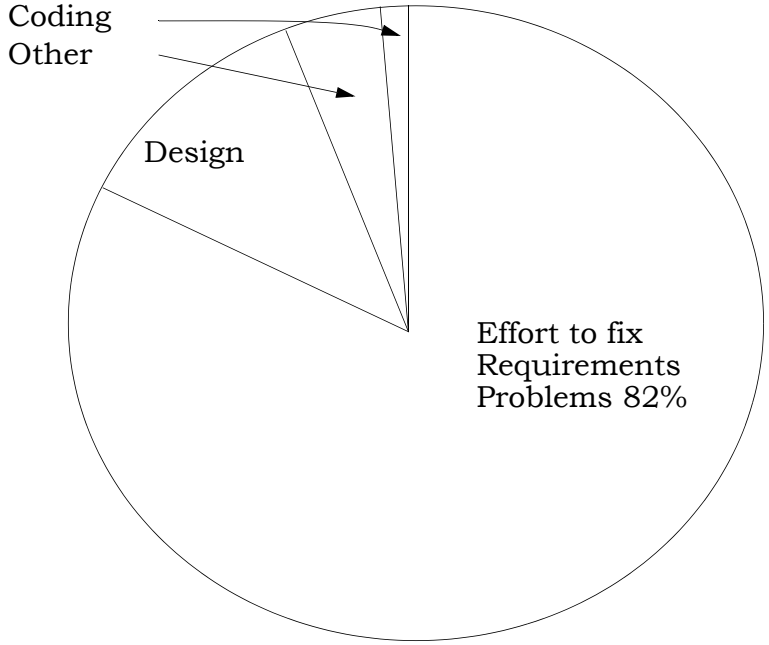


This is a stunning result which shows that requirements errors cause more problems that ALL OTHER TYPES OF ERRORS COMBINED!

Well, we know that coding errors are tough, but they are usually easy to fix once we find them. Design errors can

require a major re-structuring effort if a major change is made. But if we didn't even develop a system with the right goals, a very large amount of effort is required to revise the nature of the exact system functions. If we look at the industry distribution of 'effort', as measured by labor (person-hours) required to fix the nature of the system, [Demarco 82] concluded the following:

FIGURE 2. - Effort Needed to Fix Various Sources of Problems



From Cmpt 275, you will recall that small systems are easy to specify and build. The above statistics are for the definition and construction of **BIG** systems. But don't forget, that most systems turn out bigger that you expect, ... especially *after* the users have seen the first release, and tell you what needs to be added and enhanced!

So, analysis is needed to make sure someone takes the time and effort to CAREFULLY and DEFINITIVELY document what is needed from a proposed software system for an unfamiliar (to the programmer) application. For new systems, this requires researching and postulating ways that an automated system can improve the functioning of a business or institution, then documenting that definitively. Alternatively, when setting about to '***re-engineer***' a '***legacy***' system, this requires determining the possibly customized current functioning of the existing automated information system, specifying how a newer, better one should function, and specifying the existing data will be converted over for use by a new system.

1.2.2 The Difficulties of Analysis

There are several reasons for the pervasiveness of the difficulties caused by requirements problems:

- The subject area which the information system automates may be COMPLETELY UNFAMILIAR to the analysts and programmers.
  - What the users assume from day-to-day working knowledge ("Oh, in that special case we ...), the analysts and programmers must find out.
  - Also, terminology of some applications is completely foreign to analysts and programmers (e.g. in an air traffic control system, do you know the difference between an 'altitude' and a 'flight level'?)
- The customers themselves may not know what they need/want.
  - all the system developers are told is "something must be done to make our shipping department work more smoothly and track items better", or
  - if the customers involved in the specification process are managers, they may possibly not be experts in that field of the business (they are just are general managers), or
  - if they are users, they may understand some of the details but not the inter-relation of the needs to other operations/departments (i.e. the full set of features required).
- The customers may not be sophisticated enough to envision, in full detail, all that will need to be handled by the system. e.g.
  - the details of the data attributes and relations,
  - the details of the operations needed, and
  - any data or operation exception handling needed.
- The customers, if they are busy managers, may not feel they have the time to think about the future system and specify their needs in detail.
  - They may feel this month's big sale or big disaster needing handling is more important.
  - They may express only short-sighted, rather than corporate-wide, or long term goals for the function of the system.

- The managers of either the customer or the software developer may not consider it worthwhile to write down the requirements specifications, or document them in enough detail.
- A large requirements specification can have quite a few inconsistencies, even within itself.

The result is either a non-existent, incomplete, short-sighted, or inconsistent specification of what is required of the system to be developed (or modified).

In the last 15 years, it has slowly been recognized that expending useful effort in producing a good analysis of the requirements will pay back with a significantly reduced effort needed to handle the requirements problems which would have otherwise occurred. It is felt that on big projects, the extra analysis effort is more than compensated by a large reduction in requirements problem costs. This is basically the maxim of software engineering: Do an up-front, organized job (i.e. "Pay a little now to save more later").

Unfortunately, particularly in North American business, many middle and senior managers are more concerned about the next quarter fiscal report on sales and profits, than on long term cost reduction. They all say they are interested in long term cost reduction, but on a day-to-day basis they often make decisions which sacrifice the long for the short term. But there have been years of promises of the advantages of computer systems, but many failures in the design, implementation, and integration of these systems into corporate operations. Remember from Cmpt 275 that 15% of all software projects are cancelled (after spending an average of 75% of their budget). If this were true of corporate factory construction projects, there would be an uproar!

Note: In Cmpt 370 (93-2) there were 3 students who just returned from coop placements in a huge project (1500 database tables, 100 personnel) which was cancelled after spending $30,000,000. These students were kind of in shock, still recovering from the magnitude of it all, for a number of weeks.

So, corporations are beginning to learn computers systems take a fair amount of effort to specify and implement. There has been slow improvement over the last 15 years in the appreciation of this in corporations and institutions which have large information systems. Many books now advocate the 'fact finding' effort required to determine requirements. Specification methodologies have been designed. Diagramming techniques which facilitate the efficient and definitive communication and review of analysis models are showing up on overhead projectors during design reviews often. Computer Aided Software Engineering (CASE) tools for analysis and design are being written to make the analysis process more efficient.

Interestingly, when I was at a half-day promotional seminar in 1990 on a particularly vendor's CASE products, I managed to talk to one of the original founders of that company. He said the main impediment to the sales of their CASE tools was not that they were of poor quality, expensive, not useful, or unavailable. It was that:

a) Most development managers who would make the decisions to buy such tools did not have modern computer science degrees, and thus were unappreciative of the advantages of, and unfamiliar with the methodologies.

b) Most existing software developers were unfamiliar with the specification and diagramming methodologies, and would have to be taken off important projects to be trained. This is a rather shortsighted but typical viewpoint of project managers, whose reponsibilities are to getting a project finished, rather than to ensure strategic training investments are made to obtain future productivity gains.

Nonetheless, many continuing studies and undergraduate courses such as this one have been created to slowly help alleviate this skill deficiency!

1.2.3 The Software Life Cycle

The software life cycle is a description of the evolution that a software system moves through as it is specified, developed, used, and retired. It is important to review this evolution, not just to show where analysis fits in, but more importantly, to show that analysis is a small part of the effort in a system's life and it is aimed at significantly reducing the effort needed in the many later phases. The typical steps are:

1) Request For Proposal (RFP)
   - Someone realizes that something needs to be done to improve things. An RFP briefly describing the problem is issued internally or advertised externally requesting others respond estimating the time and cost to properly analyze (and in some cases implement a solution to) the problem.
2) Analysis
   - A contract or work order is issue to an analysis or analysis/development firm.
   - The problem is analyzed, and a requirement specification is written.
3) External Design
   - A draft of the user manual is written, complete with:
     * exact input screen and data formats,
     * exact output specifications and report formats,
     * description of each operation performed and exceptions.
4) Architectural Design
   - The structure of the software subsystems, and how control will thread through the subsystems, is planned. Analysis objects are assigned to individual subsystems.
5) Internal Interface Design
   - The details of the client-server interaction mechanism between modules is worked out, whether it be procedure calls, interrupts, inter-process or inter-thread synchronization, or network communications. Usually procedure names, parameter names, and parameter types are selected. Often at this point, with a strongly-type language, the interfaces (i.e. definition modules) can be compiled and tested for compatibility.

6) Detailed Design
- The actual algorithms and data structures they work on are selected and coded. This is the only phase where any serious programming takes place.

7) Unit Testing
- Depending on the developer's software development process, compilation and unit testing may be a separate step, or simply part of detailed design.

8) Integration and Integration Testing
- In a large system, the modules are not normally just linked together to see if the whole system will run (it usually has so many problems that it is difficult to isolate their source). Particularly when using older languages which do not strongly-type check procedure parameter numbers and types, incremental integration using either the top-down method with stubs, or bottom-up method with drivers, is needed.

9) System Testing

10) Installation, Cut-over, and Training
- Since some systems have to cut-over without any down time from the old system and data, to the new system (working on either the old or new versions of the data), considerable planning and care is required. At Easter'94, I had a friend who along with much of the Gemini company staff which maintains Air Canada's reservation system, worked the entire 4 day weekend doing a cut-over that was 2 years in planning. If it didn't work, they were prepared to either return the old system, or try and make instant fixes to the new system!
- Some widely used systems require whole departments devoted just to developing course materials (manuals, videos, and on-line tutorials) and giving on-site training to customers (e.g. airline reservations system training for internal Air Canada employees and for travel agents).

11) Maintenance (memorize the 3 parts)
- Fixing the software
- Enhancing the software
- Porting the software to new underlying hardware, operating systems, database management systems, and communications systems.

12) Retirement
- Phase-out normally requires the conversion of data to formats required by the new system, or sometimes the modification of the old system to permit the parallel running of both the old and new systems on both the old and new data formats, until everyone is sure the new system is running good enough.

1.2.4 The Role of the Analyst

The analyst plays the same role as the architect in building construction. Mainly he gathers requirements, drafts a description of how the building will look, and passes that on in a well documented manner to a civil engineer to do the structural design.

There are occasions when the analyst works for the customer developing a detailed requirements specification which is issued to developers as part of an RFP. Or he may work on behalf of the contractor to enhance a brief RFP into a full fledged specification. Sometimes on a big project a whole team of analysts are needed just to do the analysis, leaving the designers and civil engineers to lay out the structural plans. Other times, on small projects, the analyst can if properly trained and educated, do the design himself. In this course, when we use the term analyst, we will mean the person or persons who does the analysis only.

Now, you can't just say to an building architect "build me a new and better factory". An architect would immediately say "tell me more detail, or tell me where I can get more detail". She will become a scout and go on a requirements finding mission to:

1) Examine any current factory that the company has (e.g. lay-out of production lines).
2) Refer to any operations manuals that describe the workings of the factory, and how it interfaces to the product development, shipping, and marketing departments.
3) Research, or even conduct measurements of, volume of storage needed, rate of production of each product, percentage of product needing shipment via truck, rail, air, etc.).
4) Interview the management and workers about the problems and advantages of the current plant.
5) Analyze any plans for future expansion that the company has (e.g. different products, higher productions rates, or space needed different production equipment expected to be installed).
6) Investigate proposed sites for the new factory (e.g. how much area, is the land on a slope, which side has road access for the loading bay, is it adjacent to a railway track?).

Only after doing the above would the architect begin to consider the design of the new factory. And even before the design was formally drawn, she would show preliminary sketches to senior and factory management. Such a review checks:

a) Whether her understanding of the problem is correct.
b) Possibly gives management a chance to notice something missing (e.g. no elevated loading dock, or no elevator for handicapped people).
c) Provides her an opportunity to ask further questions she has thought of since she started her scouting and done her sketches.

Additionally and importantly, the analyst acts as a single interface who asks all the questions once, and documents

them properly for referral by the designers and programmers whenever they need information about the system to be develop. Since up to 100 designers and programmers may subsequently be working on a project, they would otherwise all be regularly phoning up the customer and asking a tremendous number of redundant questions. This would drive the customer crazy.

So this is what an analyst does:

1) Look at the present,
2) Gather requirements facts and measures,
3) Create a model of what the new system features should be, and
4) Communicate it efficiently and definitively via diagrams to both customer and designers for review, feedback, and subsequent implementation. Remember the maxim: "a picture is worth a thousand words".
5) Write a coherent and definitive description which will provide both:
   - an interface between the customer's needs and the designers/programmers, and
   - which may form the basis of the future development contract.

Finally, your should realize that the analyst, as a result of her analysis, may feel that the business is organized and being run the wrong way! It could be that certain departments wrongly have write authority on certain data, or choose new part numbers for products which conflict. Or it could be that she recommends changing to a system where all parts are bar-code stamped for easy tracking. In essence, she may suggest 're-engineering' the business, not just the computer system.

1.2.5 "Analyst Wanted, Programmers Need Not Apply"

The title of this section is taken from [Kendall 87], where the point is made that analysts need a different skill set compared to programmers.

Programmers notoriously think totally in terms of 'how' the system should be implemented (e.g. "Would I use a variant record, or just a regular record and leave some fields blank?"). Unfortunately, once you have been trained to understand how things are built, it is incredibly hard to get away from picturing in your mind how something will be implemented, and instead just think only about 'what' the system needs to do. Inappropriately concentrating on the 'how' aspect has two negative effects:

1) It unneccessarily clogs your mind with extra considerations that do not need to be considered at this early stage. Huge information systems have requirements that compose several binders full of analysis (e.g. Some coop students who recently took this course had worked on a project which had 1500 relational tables/entities in it!) Always, in dealing with complex systems, it is important to abstract away unnecessary detail. (Research has proven that humans can only handle up to approximately 7 things at once, before starting to risk making mistakes. The mistakes are usually either:
   - oversight,
   - inconsistency, or
   - mis-understanding of the interface/relationship between objects.
2) Concentrating on the 'how' can also unfortunately bias you against features which you may (possibly wrongly) think would be hard to implement. To propose the best possible system, the analyst should be a 'lateral thinker',

unifying the best parts of many somewhat wild alternate solutions independent of cost, employment layoffs, or the 'old ways of doing things'.

Generally, an analyst never even talks to a programmer, nor does he or she need to know how to program. Systems analysts can be used to analyze how any system works and suggest changes or improvements. For instance, a time and movement analyst could be asked to go on an aircraft carrier and figure out why there are problems getting airplanes, fuel, and bombs up onto the deck quickly. His job would be to study and suggest ways to more efficiently use bottlenecks such as the aircraft elevators. Should all 4 elevators be devoted at particular times to one type of cargo, or should you dedicate one permanently to do only the fuel and weapons transfers to the deck? And what are the tradeoffs of speed versus safety against explosion?

So, the analyst typically has the traits of a curious scout, a lateral thinker, a solution constructionist, and a pretend user. His main interface is between the customer and the designer (not the programmer). Just as the architect must enlist the help of the civil engineer to help with the design, the analyst's job is mostly that of a conceptualist who determines requirements and conceives solution features (rather than the design in detail).

Nonetheless, the designs often follow the solutions closely, so the analyst and the designer often have considerable understanding in common. But it is the analyst who tells/ illustrates to the designer what data must be stored and manipulated. If the designer can envision the kinds of problems users have, has the determination and people skills to ferret out requirements from sometimes reluctant sources, is a good lateral thinker, and can propose a better organization and control authority for data and operations, then he can also take on the role of an analyst. Similarly, a good analyst, if she is versed in data normalization, user

interface mechanisms, and distributed system design, may be capable of subsequently evolving her analysis models into a design.

Unfortunately, many job ads today ask for programmer-analysts. This either skips design or encompasses all 3 fields. It is also the job title many programmers get promoted to if they do a good job programming, even if they know nothing about analysis!

1.3 Design and Designers

In this section, design will be explained and justified. Then, since it is a unique task, we will describe the skill set needed for the special role played by the designer.

1.3.1 Design

Simply put, design is the choosing among alternative designs. Design is not using the first implementation method that pops into your head!

Typically, there are always several ways to implement things. e.g.

- brick vs. wood house construction
- batch processing vs. interactive
- prompt-driven vs. interrupt driven
- buy a database management system to build upon vs. write your own
- use tree-based structures vs. linear ones (or hybrids for fast random and sequential access)
- GUI vs. text screen
- C++ vs. ADA 9X

Unfortunately, rarely does one of the two choices have no drawbacks and thus rarely is there a glaringly-obvious preferred choice. Usually, each of the choices has different advantages and disadvantages that must be weighted.

How is this relative evaluation done? Usually on the basis of several factors (in no particular order):

- feasibility
- performance
- memory space usage (RAM and DISK). c.f. VM quotas
- ease of use or other benefit to the customer
- ease of implementation (i.e. development cost)
- safety/reliability
- cost

The evaluation of the above can often be done in a quantitative manner. And since this is computer science, you will be expected to be able to use analytical techniques to make quantitative comparisons.

Unfortunately, the relative merits of the above factors is not clear (e.g. 1 MB of extra memory use will get you 15% faster performance. Should you choose this design?). Often, the designer must use his judgement, and/or consult with the requirement specification (or the analyst or customer) regarding these decisions.

Design must be done before coding for several reasons:

a) To prevent the system being thrown together without consideration for its overall lay-out, or its ease of future enhancement or porting. These three important factors are easily forgotten if you just sit down and start coding.

b) To prevent the programmers from implementing a design that does not look like what the customer wants (i.e. external design was not approved first). This oversight could result in the need for a tremendous amount of re-work to change the system user interface and possibly the related underlying control mechanism; much more work than would have been required vs. simply wordprocessing a few changes to the draft user manual.

c) To keep programmers from slowing down to consider how the system should work or appear to the user.

d) To break the system up into work units suitable for one person to do in a week, yet retain confidence the interfaces between the units will be well defined and efficient.

e) To allow the use of novice (coop?) programmers to work on important projects. This allows cheaper labor, and provides an excellent training ground for staff.

1.3.2 The Skill Set of a Designer

A designer is really the design engineer for the system. He or she is an experienced programmer who:

a) Has experience working on the maintenance of systems, and knows from harsh experience how difficult it is to fix, enhance, and port poorly designed and documented systems.

b) Has worked on at least two big projects previously, and thus has seen the design process in action.

c) Usually has a good knowledge of some or all of the:
   - application subject.
   - the operating system or GUI.
   - the database management system on top of which the system must be designed to function.
   - the network.

d) Has had some analysis experience which has put him more in touch with computer-illiterate users, and with fickle customers (or his own fickle marketing department). He thus understands the importance of designing in a way that will be somewhat tolerant to specification changes in mid-project.

e) Has some project management experience, so s/he understands the best way to break the system into optimum units for development personnel scheduling, for testing, enhancement, and future re-use. He will also understand configuration management, so that the design is structured so changes will hopefully only require changes of a few units.

f) Finally, he must be able to train his staff to become future designers, by allowing them to design parts, and conducting design reviews to improve their skills (and they his!).

The designer will thus need a rather broad skill set spanning familiarity with user needs, to project management, to programming.

Recently, I read and interesting analogy. It suggested that the designer must understand, and the methodologists must provide descriptive modelling tools to allow reasoning and review of the features of whole systems (10 million lines of code and 3 volume manuals) down in scale to individual

lines of code. This is 7 orders of magnitude. If this range of scale were applied to building design, it would require the designer to understand the use of building materials from large girders down to also being a molecular engineer!

1.4 Purchase/Modify/Develop Decisions

Sometimes the analyst, and sometimes the designer, must make a judgement as to whether the best (i.e. most suitable and preferably most economical) change would be to either:

a) buy a new application which does the job, or
b) modify the existing application system, or modify a highly configurable, newly-purchased system, or
c) write a new application from scratch.

The trade offs are as follows:

a) A purchased system is usually:
   - cheaper because the developer of the system was able to spread the development costs over several customers,
   - better because it is both well designed and tested on previous customers, and more flexibly designed for use by several customers,
   - faster operating as better optimized for performance than what a small development firm could kludge together,
   - faster to get operational as it is already developed,
   - needs less support as it is maintained by someone else,
   - but is difficult to fit exactly into the local environment,
   - and finding the correct software package to purchase is not easy.
b) A modification of an existing system is often:
   - cheaper
   - but may not be as good as a custom written application,
   - and major modifications can be expensive.
c) A custom system development is:
   - good as it is exactly tailored to the desired application problem,
   - but may be very costly to develop,
   - and may take a long time to get operational.

The above decision options are very hard to judge. The decision depends both on the results of the analysis, and additionally on the judgement of a designer. The designer must understand the present system's advantages and weaknesses, and be able to estimate time and costs to modify the present (or a purchased system), vs. developing a new one from scratch.

1.5 References

[Demarco 82] "Software Systems Development", Tom Demarco, Prentice-Hall, New York, 1982.

[Kendall 87] "Introduction to Systems Analysis and Design" Penny A. Kendall, Wm. C. Brown Publishers, 1987.

[Maciaszek01] "Requirements Analysis and System Design: Developing Information Systems with UML" by Leszek Maciaszek, Addison Wesley, 2001.

[Montgomery94] "Object-Oriented Information Engineering" by Stephen Montgomery, Harcort Brace, 1994.

[Quatrani98] "Visual Modeling with Rational Rose and UML" by Terry Quatrani

[Shlaer 92] "Object Lifecycles: Modelling the World in States", Sally Shlaer and Stephen Mellor, Prentice-Hall, 1992.