REPUBLIC OF THE PHILIPPINES
POLYTECHNIC UNIVERSITY OF THE PHILIPPINES
STA. MESA, MANILA
COLLEGE OF ENGINEERING
COMPUTER ENGINEERING DEPARTMENT

CMPE30074

SOFTWARE DESIGN

INSTRUCTIONAL MATERIAL

ENGR. JULIUS CANSINO

Welcome
to
Software Design

This course focuses on techniques used throughout the software development process. The software lifecycle and modeling techniques for requirements specification and software design are emphasized. Both traditional and object-oriented approaches are addressed. A group project gives students hands on experience developing a software requirements specification, analysis documentation, design specifications and a working prototype. This is a project-based class where students are expected to start from a narrative of the problem, and then specify output reports, analyze the problem using special data modeling techniques (entity-relationship, relational, object-oriented), design data structures, and then follow through with any Programming Language. Nine workshops are scheduled to allow teams to work on the project.

Team Project Description

This is a project-based course that focuses on the life cycle in software engineering: prototyping, requirements, analysis, design, implementation, and testing. The Module lectures cover topics needed to understand the development process and include additional information about object-oriented and function-oriented methodologies. The project follows these steps. Students are expected to start from a selected project idea and then specify output reports, analyze the problem using special data modeling techniques (entity-relationship, relational, object-oriented), design data structures, and then follow through with any Programming language prototype. Nine workshops are scheduled to allow teams to work on the project.

The purpose of this group project is to use the concepts and tools that you will learn in this course in a simulated software development environment. This effort will include the creation of planning documents, project management plan, the capture and documentation of requirements, design, coding, and testing. You will be building a variety of models during this process.

You will accomplish this project working in teams. This will require considerable coordination. To complete all the work involved, it will be necessary that team members carry their fair share of the load. It is recommended that your team clearly allocate responsibilities and coordinate all deliverables.

Since the Software Solution project is a simulation of typical software engineering activities, the requirements have not been fully documented. It is your responsibility to gather the requirements and implement them into a prototype (the final product). At the end of the class, your team will also need to present and demonstrate your simulated project prototype to the class.

Project Grading

The Team Project is worth 70% of your total grade and is broken up into the following deliverables/percentages. Refer to the course syllabus for specific due dates.

Deliverables	Percent of Total Grade
Requirements Specification	5%
Analysis Document	20%
Design Specification	20%
Prototype	20%
Class Presentation	5%
Total Project:	**70%**

Table of Contents

Module One

Objectives:

- Software Development Life Cycles
- Phases of Systems Development and the Software Process
- Strategies for Systems Analysis and Problem Solving
- Define the Team Project and Requirements.

Assignments/Activities:

- No deliverables
- Teams (three students each) will be formed

Lecture Notes: Part 1: Software Development Life Cycles, Phases of Systems Development, and the Software Process

Introduction

"Although managers and practitioners alike recognize the need for a more disciplined approach to software development, they continue to debate the way discipline is to be applied. Many individuals and companies still develop software haphazardly, even as they build systems to service the most advanced technologies of the day. Many professionals and students are unaware of modern methods. And as a result, the quality of the software that we produce suffers and bad things happen. In addition, debate and controversy about the true nature of the software engineering approach continue." Roger Pressman, 2001

Software Engineering is about:

- Quality improvement
- Reliability
- Maintainability

Within the framework of:

- Time
- Money

- Customer requirements.

The goal is to bring to the software development process the same rigor and discipline associated with traditional engineering methods. Software Engineering evolved during the late 1970's and as Dr. Pressman's quote above attests, we are still struggling to apply these principles and practices today. This course introduces you to these various software engineering principles and practices regarded collectively as a software methodology or software process.

In your academic careers thus far, you most likely have been coding single applications or programs. Not much formality of process is required. You are the sole analyst, coder, and tester. You wouldn't have much need for software engineering. However, when developing large-scale, multi-functional software systems requiring dozens or hundreds of software engineers to construct, more formal and rigorous software engineering practices are necessary to ensure their success and delivery of a quality product.

The Early Years

The first computer built in the early 1950's doesn't remotely resemble the computing devices that we use today. However, the basic computing principles on which it was built formed the basic foundation for all the advancements that have followed.

These early devices relied on vacuum tubes to store and transmit data. These vacuum tubes failed at the rate of 1 every 3 seconds. When a tube failed, the process it was computing typically failed as well making it necessary to restart and possibly reprogram the process. It required teams of engineers to diagnose which tubes failed, correct the problem immediately by inserting a new tube, and repeatedly restart the failed process. At that rate, it could take days to complete a single computing task.

There were not any programming "languages" during this period. Computer programs were written in "machine code" or the manufacturer's "assembly language." These programs were tightly coupled to the hardware on which they ran. A program written on one computer could not be run on another computer of a different manufacturer or even the same manufacturer of a different model.

The Second Era

The Second Era of computing evolved in the 1960's. During this time, computers were still very large (mainframe), very expensive, yet they were markedly more reliable than their predecessors. IBM and Honeywell began manufacturing and marketing computers to large-scale businesses: banking, insurance, etc. The

early business intent for the computers was to store and process data at a rate faster than human resources were able to do the job. Thus, saving big businesses big bucks. In addition, the ability to process more data at a faster rate also meant more revenue.

Software languages began to evolve: Basic, Pascal, Fortran, and later COBOL. This made programming computers easier. However, Programs were still tightly coupled to the hardware on which they were written.

The Third Era

The creation of the microprocessor revolutionized the computing industry in the mid to late 1970s. Minicomputers entered the market. These computers, in most respects, were more powerful than most of their mainframe counterparts of the previous era. They were certainly more affordable. Now, most businesses recognized that they needed to utilize the power of the computer to compete. Providers could not manufacture them fast enough. Demand exceeded supply.

Operating systems were becoming more standardized amongst manufacturers. New software languages evolved, i.e., C, C++, Ada, etc. A new market developed: prefabricated, off-the-shelf software. Prior to this era, if you needed a problem solved by the computer, you had to write software to accomplish the task. Now, many common business functions such as accounting practices, payroll, etc. were developed and marketed as off-the-shelf items. Business users had a choice: build it or buy it.

The early computer adopters during the Second Era had, by this time, amassed quite an investment in their mainframe technology and had uncountable numbers of computer programs written in older languages that were tightly coupled to their hardware technology. This represented the first dilemma of the software industry.

- Do they start over and rewrite all their applications?
- Do they try to "re-engineer" them by manipulating the existing software so that it would run on the newer technology?
- Do they determine which applications can be replaced by products that are now available off-the-shelf?

They tried them all. The result was that some were successful, and others were not, but in either case all were quite expensive. Though they have made some progress, these large, early adopters are still wrestling with this dilemma today.

During the Third Era, two-year colleges began to develop "computer programming" curricula to satisfy the need for business-minded programmers who could relate to the business problems they were attempting to solve. Though, these new programmers did relate more to the business, they lacked

thorough understanding of the hardware they were using to program. These new programmers were not as adept at designing computing solutions or troubleshooting complex computing problems. The traditional four-year universities were developing Computer Science curricula that concentrated on producing the new hardware and software language pioneers.

The Fourth Era

The Fourth Era is mainly defined by the introduction of the personal desktop computer. Though the business community took early PCs seriously, there was not much software available for them. The programmers of mainframe and mid-tier computing systems did not understand much about the technology. Early PCs came with an operating system (that was an unknown to the users), a few games, rudimentary word processing, and a crude spreadsheet application. They were not very fast or powerful. They were largely viewed as toys by the business world.

However, by the end of the fourth era, advances in microprocessors and the availability of off-the-shelf PC software for business applications caused the business community to make large investments in desktop computing. Today, nearly every white-collar employee has a PC or has access to a PC to perform major portions of their daily work.

The Internet was also introduced late in the fourth era and has revolutionized marketing and sales strategies for most businesses.

The Next Era

It is hard to predict what will come next. This past year was filled with technology markets trying to capitalize on the Internet. Market analysts failed to see any real value add in most of these products and stock prices fell. As hardware continues to become faster, better, and less expensive, our software advances will continue to follow. Wireless technologies and voice recognition have just begun to make inroads in the market.

Complexity of System Design

In the early days of mainframe computers, limitations of the technology made for few engineering choices. The end user terminal contained no intelligence. You had one computer, the mainframe that performed all functions. Few choices for database structure were available. Networking was performed through phone lines via modem devices at short range, storage was a concern, and software languages were COBOL and perhaps FORTRAN or BASIC. End user input was

via a computer terminal or submitted on paper for input by data entry operators onto tape or disk storage for batch input.

During the third era of client-server, mid-tier devices and intelligent desktop computers system design became a little more complex. Terms such as "fat client" and "thin client" emerged. If the designer chose to put most of the computing load on the desktop it was termed "fat client." If the designer chose to put most of the computing on the server or host server with less on the client, it was termed "thin client." Early desktops were not designed to handle heavy computing loads and the "fat clients" did not perform well.

How could these systems be in a network together? Wide area networks were emerging, expensive, and yet they worked. Mid-tier database technologies were being perfected. There were many questions that emerged, as disparate systems became part of the same network. For instance, should we offload the database access (I/O) from the mainframe to the mid-tier? Much was learned through trial and error using the client-server interaction and now entering the 21st century even more technology has entered the engineering mix to form a highly complex computing environment.

It is rare today that a computer system stands alone. It is typically integrated with other computer systems to form a web of complex data transfers and synchronization schedules, networking challenges and security concerns. Larger and larger teams of engineers are required to maintain them, compounding the complexity of the system.

Software Engineering

Engineering is a relatively well-known term and is applied to all sorts of professions: mechanical engineer, aerospace engineer, construction engineer, domestic engineer, and so on.

Engineering is a systematic and disciplined approach to problem solving for complex solutions. When it is applied, it produces economical, efficient and reliable products.

Hence, software engineering is the application of engineering principles to software development. Software engineering was first introduced over 20 years ago as an attempt to provide a framework and bring order to an inherently chaotic activity. Software engineering is still evolving and much debate and controversy still continues within the discipline. The engineer needs to determine the requirements of the system being designed and select the appropriate platforms or combinations of platforms on which to implement the designed solution. The term "right-sizing" was coined more than a decade ago to indicate that each of the platforms - personal/desktop computer, mid-range and

mainframe computer each have their strengths and weaknesses that the engineer must take into account.

The term software engineer is widely used to encompass the various disciplines involved in software design: data administrators, software architects, security administrators, programmers, quality assurance specialists and many others necessary to deliver a final software product.

Engineering physical constructs

When an engineer is faced with the task of building a bridge, he/she would not consider beginning construction until the requirements of the customer were fully understood, blueprints were drawn and approved by the customer. However, most software engineers today begin construction of systems before the requirements are fully understood or before a workable design (blueprint) is developed. Taking these "shortcuts" has been caused by:

- People unfamiliar with quality software engineering practices.
- Shortened delivery cycles within the software market.
- Increased demand for software products that exceeds supply.

Building software, by nature, is different than building a bridge or a house. It is a logical construct rather than a physical structure. It is more difficult to depict software requirements and design in terms of what a customer wants it to do, rather than a house or bridge in customer terms of what they would like it to look like. We have centuries of experience building bridges and houses to determine what quality factors go into the construction. We have comparatively little experience to draw on when determining software quality factors.

What is a System Development Life Cycle?

A system development life cycle (SDLC) is a logical process by which systems analysts, software engineers, programmers, and end-users build information systems and computer applications to solve business problems and needs. It is sometimes called an application development life cycle. The SDLC usually incorporates the following general-purpose problem solving steps:

Planning - identify the scope and boundary of the problem, and plan the development strategy and goals.

Analysis - study and analyze the problems, causes, and effects. Then, identify and analyze the requirements that must be fulfilled by *any* successful solution.

Design - if necessary, design the solution not all solutions require design.

Implementation - implement the solution.

Support - analyze the implemented solution, refine the design, and implement improvements to the solution. Different support situations can thread back into the previous steps.

Example SDLC: The Waterfall Model

The Waterfall model of software development, formed in the 1970s, as an attempt to bring order to the chaos of the early “code-and-fix” model -- write some code and fix the problems -- of software development. The primary problems with the code and fix model lay in the lack of overall foresight and structure of complex software engineering projects resulting in expensive fixes, poor match to users’ needs, and expensive redevelopment and/or failure of the product. The Waterfall model encourages software development to proceed in successive stages including operational specifications, coding specifications, coding, parameter testing, assembly testing, shakedown and system evaluation. This model has become the basis for most software acquisition standards in government and industry.

Example SDLC: FAST Model

Both the Waterfall and *FAST* are commonly used in today's software development environment. *FAST*, like most modern commercial methodologies is base on a combination of techniques called best practices. All methodologies have similarities and common ground. It is up to the development team to decide on which techniques and practices to use. Most development is done using a combination of several SDLC models.

The Life Cycle Phases

Each of the various SDLC models contain a number of different phases. These phases may be independent, inter-related, or iterated many times. Nevertheless there are a number of typical phases, which can be identified as existing (in one form or another) in most models. These are analysis, design, implementation, testing, and maintenance. These phases are most visible in the waterfall model (or its variants) but still exist in the other models. Each of these will be discussed in turn.

Analysis (Waterfall Model = Concept Phase + Software Requirements Phase)

The system to be developed is analyzed and specified. This involves considering the environment in which the system operates, the systems services, constraints and goals, the inputs that are available, and the outputs that must be produced.

This is formalized in a specification of the system. The specification should specify what the system must do, and NOT how it is to be done. The input to this phase will be the knowledge of the users and their requirements, and the knowledge of the problem domain. The outputs will be the requirements specification (which specifies what the system should do, along with other factors such as cost, timeframe etc.), acceptance testing specifications and possibly a draft user manual.

Design (Waterfall Model = Design Phase)

The specification is used to design the system. The specifications are partitioned to either hardware or software and the structure of the system is established. The software is typically partitioned into increasingly small components until the method for implementing each component can be readily outlined. The input to this phase will be the requirements specification. The output will be the design documentation, and often component test specifications.

Implementation (Waterfall Model = Implementation Phase)

The actual code is written, using the design documentation as a guide. This will probably also include low-level testing of the individual components that are developed and integration of these components to produce the final working system. The input to this phase is the design documentation, and the output is the program code.

Testing (Waterfall Model = Integration and Test Phase)

The system is tested to ensure that it adheres to the original user requirements. This will also test the appropriateness of the whole system, and identify any deficiencies (as opposed to errors) that need to be rectified. The input will be the program code, the specification and the acceptance test requirements. The output will be test reports and the final working system.

Operation and Maintenance (Waterfall Model = Maintenance Phase)

As the system is used, deficiencies may be uncovered, or changes in the environment or requirements will mean that changes to the software need to be made. The software is progressively modified to keep it up to date and relevant.

What *is* a Methodology?

A methodology is the physical implementation of the logical life cycle that incorporates (1) step-by-step activities for each phase, (2) individual and group roles to be played in each activity, (3) deliverables and quality standards for each activity, and (4) tools and techniques to be used for each activity. A true methodology should encompass the entire system's development life cycle. Most

modern methodologies incorporate the use of several development tools and techniques.

Why Do Companies use Methodologies?

There are three reasons why companies use Methodologies.

(1) Methodologies ensure that a consistent, reproducible approach is applied to all projects.
(2) Methodologies reduce the risk associated with shortcuts and mistakes.
(3) Methodologies produce complete and consistent documentation from one project to the next.

The Systems Analyst - A Key Resource

Many organizations consider information systems and computer applications as essential to their ability to compete or gain competitive advantage. Information has become a management resource equal in importance to property, facilities, employees, and capital. All workers need to participate in the development of these systems and applications – not just the computer and information specialists. But one specialist plays a special role in systems and applications development, the *systems analyst*. A systems analyst(s) *facilitates* the development of information systems and computer applications. The systems analyst performs *systems analysis and design*.

> Systems analysis is the study of a business problem domain for the purpose of recommending improvements and specifying the business requirements for the solution.
>
> Systems design is the specification or construction of a technical, computer-based solution for the business requirements identified in a systems analysis. (Note: Increasingly, the design takes the form of a working prototype.).

Why do businesses need Systems Analysts?

The system analyst bridges the communications gap between those who need the computer and those who understand the technology.

What is a Systems Analyst?

Systems analysts are people who understand both business and computing. Systems analysts study business problems and opportunities and then transform business and information requirements of the business into the computer-based

information systems and computer applications that are implemented by various technical specialists including computer programmers.

A formal definition:

A systems analyst facilitates the study of the problems and needs of a business to determine how the business system and information technology can best solve the problem and accomplish improvements for the business. The *product* of this activity may be improved business processes, improved information systems, or new or improved computer applications, and frequently all three.

When *information technology* is used, the systems analyst is responsible for:

- The efficient capture of data from its business source the flow of that data to the computer
- The processing and storage of that data by the computer
- The flow of useful and timely information back to the business and its people

What is Information Technology?

Information technology is a contemporary term that describes the combination of computer technology (hardware and software) with telecommunications technology (data, image, and voice networks). The role of systems analyst is changing into two distinct positions or roles, *business analyst* and *application analyst*.

> A business analyst is a systems analyst that specializes in business problem analysis and technology-independent requirements analysis.
>
> An application analyst is a systems analyst that specializes in application design and technology-dependent aspects of development. A synonym is *system* or *application architect*.

What Does A System Analyst Do?

A system analyst is a system-oriented problem solver.

> System problem solving is the act of studying a problem environment in order to implement corrective solutions that take the form of new or improved systems. Most systems analysts use some variation of a system problem solving approach called a system development life cycle.

System analysts are responsible for other aspects of a system including:

People, including managers, users, and other developers – and including the organizational behaviors and politics that occur when people interact with one another.

Data, including capture, validation, organization, storage, and usage.

Processes, both automated and manual, that combine to process data and produce information.

Interfaces, both to other systems and applications, as well to the actual users (e.g., reports and display screens).

Networks, which effectively distribute data, processes, and information to the people.

What is a user?

A user is a person, or group of persons, for whom the systems analyst builds and maintains business information systems and computer applications. A common system is client. There are at least two specific user/customer groups: *system users* and *system owners.*

System users are those individuals who either have direct contact with an information system or application or they use information generated by a system.

System owners provide sponsorship of information systems and computer applications. In other words, they pay to have the systems and applications developed and maintained.

The are two types of system users:

Traditionally, most system users were internal users, that is employees of the business for which a system or application is designed. Today's user community includes external users as businesses seek to make their information systems and applications interoperate with other businesses and the consumer. Information technology managers and system analysts are making a demonstrated attempt to get closer to their customers by forming a partnership.

Lecture Notes: Part 2: Strategies for Systems Analysis and Problem Solving, and Definition Phases of Systems Analysis.

Modern Structured Analysis

Structured analysis was one the first formal strategies developed for systems analysis of information systems and computer applications. It is a process-centered technique that is used to model business requirements for a system. The models are structured pictures that illustrate the processes, inputs, outputs, and files required to respond to business events. Structured analysis introduced an overall strategy that has been adopted by many of the other techniques – *model-driven development*. A model is a representation of reality. Just as "a picture is worth a thousand words," most models use pictures to represent reality. Model-driven development techniques emphasis the drawing of models to define business requirements and information system designs. The model becomes the design blueprint for constructing the final system. Modern structured analysis is simple in concept. Systems and business analysts draw a series of process models called *data flow diagrams* that depict the essential processes of a system along with inputs, outputs, and files. Because these pictures represent the *logical* business requirements of the system independent of any *physical*, technical solution, the models are said to be a *logical design* for the system.

Information Engineering (IE)

Today, many organizations have evolved from a structured analysis approach to an information engineering approach. Information engineering is a data-centered, but process-sensitive technique that is applied to the organization as a whole (or a significant part therefore – such as a division), rather than on an ad-hoc, project-by-project basis (as in structured analysis). The basic concept of information engineering is that information systems should be engineered like other products. The phases are the following:

Information Strategy Planning (ISP) applies systems analysis methods to examine the business as a whole for the purpose of defining an overall plan and architecture for subsequent information systems development. Based on the strategic plan, business areas are 'carved out' and prioritized. A business area is a collection of cross-organizational business processes that should be highly integrated to achieve the information strategy plan (and business mission).

A Business Area Analysis (BAA) uses systems analysis methods to study the business area and define the business requirements for a highly streamlined and integrated set of information systems and computer applications to support that business area. Based on the business area requirements analysis,

information system applications are 'carved out' and prioritized. These applications become projects to which other systems analysis *and* design methods are applied to develop production systems. Information engineering is said to be a data-centered paradigm. Since information is a product of data, that data must be planned first! Data models are drawn first. In addition to data models, information engineers also draw process models similar to those drawn in structured analysis.

Prototyping

Prototyping is an engineering technique used to develop partial, but functional versions of a system or applications. When extended to system design and construction, a prototype can evolve into the final, implemented system. Two 'flavors' of prototyping are applicable to systems analysis:

> Feasibility prototyping is used to test the feasibility of a specific technology that might be applied to the business problem.
>
> Discovery prototyping (sometimes called *requirements prototyping*) is used to 'discover' the users' business requirements by having them react to a 'quick-and-dirty' implementation of those requirements.

Joint Application Development (JAD)

Joint application development (JAD) uses highly organized and intensive workshops to bring together system owners, users, analysts, designers, and builders to jointly define and design systems. Synonyms include *joint application design and joint requirements planning*. A JAD-trained systems analyst usually plays the role of facilitator for a workshop. A JAD workshop will typically run from three to five full working days. This workshop may replace months of traditional interviews and follow-up meetings.

Business Process Redesign (BPR)

Business process redesign (also called *business process reengineering*) is the application of systems analysis (and design) methods to the goal of dramatically changing and improving the fundamental business processes of an organization, independent of information technology. BPR projects focus almost entirely on non-computer processes. Each process is studied and analyzed for bottlenecks, value-returned, and opportunities for elimination or streamlining. Once the business processes have been redesigned, most BPR projects conclude by examining how information technology might best be applied to the improved business processes. This creates new application development projects.

Object-Oriented Analysis (OOA)

Data and the processes that act upon that data are combined or *encapsulated* into things called *objects.* The only way to create, delete, change, or use the data in an object (called *properties*) is through one of its encapsulated processes (called *methods*). Object-oriented analysis (OOA) techniques are used to:

(1) Study existing objects to see if they can be reused or adapted for new uses, and to
(2) Define new or modified objects that will be combined with existing objects into a useful business computing application.

Module One Team Activities:

- Form Teams of three
- Select a Topic for the Development Project
- Discuss Project Design and Requirements
- Assign Team Roles and Responsibilities
- Address any Concerns and/or Team Logistics.

Module Two

Objectives:

- The Engineering Process
- Standards and Documentation
- Requirements Specifications

Assignments/Activities:

- Project Topics Due

Lecture Notes: Part 1: The Engineering Process

The generic software engineering lifecycle involves:

- Planning activities
- Analysis of the customer requirements
- Software design
- Software construction (coding)
- Testing
- Maintenance

Software maintenance is sometimes included in the software lifecycle but most discussions of software engineering do not include these activities. Additionally, some software engineers do not consider the planning activities part of the software engineering lifecycle. Again, this is part of the controversy amongst software engineers.

Generally, any software development process begins with the "definition phase." In this phase the software designer/developer gathers and analyzes the customer's requirements and begins to develop an overall system design to satisfy those requirements.

During the "development phase" the software designer/developer refines the "definition phase" design to a working software model, perhaps even constructing a prototype for customer review. Through iterations of the design model refinement, code development, and testing, a working product eventually emerges. The product is delivered to the customer and put into production. Over time, the customer may request modifications to the product, or sound engineering practice may dictate modifications to maintain the product's technological stability or currency. On-going service enhancements to the

product are generically referred to as the "maintenance phase." Experience over the past 20 years of software development has shown that maintenance activities consume anywhere from 30 - 60% of the overall IT resource budget. Much research has been conducted over the past decade to discover ways to reduce this expenditure to 20-30% of the IT budget.

As the generic software development life cycle moves through definition, development, and maintenance phases, there are "umbrella" activities that begin the first day the customer requests a service or product and continue throughout the entire software development life cycle. These activities are:

1. Project tracking and control
2. Formal technical reviews
3. Software quality assurance
4. Software configuration management
5. Document preparation and production
6. Measurement/metrics collection
7. Risk management

The Software Development Life Cycle (SDLC), described above in generic terms, is referred to by many names, including:

- Software Process Models
- Process Models
- Software Methodologies
- Life Cycle Processes
- Life Cycle Methodologies
- And most commonly, Software Development Life Cycle (SDLC)

These terms all refer to the same flow of software definition, development, maintenance, and all umbrella activities in between to deliver quality software products.

How SDLCs Work

The requirements phase contains all those activities in which you determine what the system or software is supposed to do. This is where you begin to analyze what the client asked for, versus what they really want, versus what they need. Requirements analysis deals with defining in a very specific manner, system or software capabilities. Design analysts then take over and use those requirements to formulate the system or software design. During the design phase requirements are allocated and traced to various parts of the design. Designers come up with the blueprints for system and software products. This information is then passed down to the programmers in the coding phase. Programmers take the initial designs, in conjunction with the requirements and begin to "make it happen". In most organizations, the programmers are given the flexibility in

regard to how they implement the designs. Designs are rarely written to a level of detail in which they tell the programmers exactly what to do. During the coding phase programmers are responsible for unit testing and first level integration tests. It is generally accepted that the coding phase ends when the programmers have completed a series of their own tests. At this point the system or software product is then handed off to a test team. The test phase is concerned with testing the reliability of what has been created. Testing usually involves a series of progressive steps that takes the product through integration, alpha, beta, and acceptance tests. Test teams also take the initial requirements and trace them down to the testing activities. In addition to making sure the product works, test teams are concerned with ascertaining whether all of the requirements were implemented as stated.

Programming can be viewed as a part of a set of activities within a life cycle model. In software engineering we talk about life cycle models in terms of a system or a software development effort. System development life cycle (SDLC) models are more global in context and are comprised of several other models or development efforts. For example, an SDLC for an accounting system would have several development efforts for the payroll module, an accounts receivable/payable module, inventory module, etc. Each of these modules could have their own software development life cycle. A system model can also contain life cycles for hardware. Phases in a system model concentrate on additional concerns such as corporate strategy, return on investment, and long-term planning. Both system and software development life cycle (also known as SDLC) models contain a series of activities broken down into phases called requirements, design, coding, testing, delivery and maintenance. There can be several other phases depending on the model and it's author, but generally we talk in terms of the six major phases.

After a successful acceptance test the product is deployed or delivered. The delivery phase contains those activities in which the product is delivered to the users. For some products the delivery phase can entail several releases or timed events where the product is put into use over a period of time. Once the product is in use the maintenance phase begins. Maintenance involves correcting software errors known as defects, as well as making enhancements to the system. The maintenance phase also includes activities for configuration management, testing, and subsequent releases or upgrades of the product.

Although there are six major phases in a life cycle model, the activities that occur in each one, and how they occur are dependent on the type of model in use. The industry recognizes several different software development models, each with its own advantages and disadvantages.

Software

Most people think of software as computer programs or code. This is a true enough statement. But, for purposes of this course, the term software will be expanded to include the executable code (programs), data structures, and documentation.

> "*Software is instructions (computer programs) that when executed provide desired function and performance, data structures that enable the programs to adequately manipulate information, and documents that describe the operations of the programs."* Roger Pressman.

Executable code in itself is not a saleable or deliverable product. Without data structures and documentation to support it, rarely would it function and it would most certainly be difficult to maintain.

Software = code + data structures + documentation

Software is an information transformer—producing, managing, acquiring, modifying, displaying, and transmitting. This information is what the customer sees as the end product, not the software. However, the software engineer sees the quality software as the end product. Software is the vehicle to deliver the information product. Software engineers must take care to always keep the customer's view of the information end product in mind when designing the software end product.

After 30+ years of developing software, early adopters of computers during the second and early third era have amassed great investments in their software inventories. The U.S. Government, large insurance companies, and large banking institutions are still dependent on some of these older, mission-critical software systems. These older systems are referred to as legacy systems. They have gone through multiple generations of changes, they were not well designed in the first place, and the original developers have either left the company or retired. There typically is not any documentation for these systems or it is so out of date that no one relies on it anymore.

Pressman (see recommended textbook) refers to this as the Software "Chronic Affliction." So what can these companies do to solve their problem? Doing nothing is certainly an option and one that has been practiced by some companies for many years. However, this practice increases the risk of complete system failure, an occurrence that could cost millions of dollars a day. Another option is to re-engineer the software, i.e., that the original software would be modified to operate on newer technology. There are many forms of re-engineering, however, re-engineering is an expensive process. Another option is to completely start over and rewrite the software. Completely re-writing these complex applications is also extremely expensive and time consuming. Yet

another alternative is to buy the application off the shelf. Off-the-shelf or pre-written software products are available for nearly every business application imaginable. These products need to be evaluated for their adaptability to the individual business' requirements. In many cases, modifications are necessary. Modifying pre-written software is not a task to be taken lightly. First come the modifications, then the documentation, and then reapplying these modifications after each subsequent upgrade of the pre-written software. More modifications are always desirable. The maintenance cycle can be daunting.

Need for Increased Software Quality

Society has become increasingly more dependent on software and computer systems, in general. Failures can generate losses of millions of dollars in revenues or even the loss of human lives. Applying software engineering methods to the software development lifecycle clearly increases the quality and reliability of the software produced. Methods for developing higher quality, more reliable software are still evolving.

Lecture Notes: Part 2: Standards and Documentation

Most of the SDLS models utilize some set of standards and documentation. For a long time the military and industry used Mil-Std 498. This body of standards and documentation was canceled a couple of years ago. The Mil-Std 498 was document intensive and followed the traditional Waterfall model of software development. It has been replaced by a series of new standards such as the International Standards Organization (ISO) 9000 and the Capability Maturity Model (CMM).

The ISO 9000 is a series of five standards that apply to a range of industries that includes software engineering. The standards focus on documenting the process and measuring the results of implementing those processes. It is very similar to CMM, but it concentrates on a different set of attributes.

CMM is concerned with the level of maturity exhibited in a software development organization. The model is broken down into five levels; each level contains a set of key practices and attributes that are characteristic of software organizations at that level of maturity. CMM is process oriented. The model views success in software development as being dependent on what processes you follow and how well they are managed. Success is a result of qualified individuals followed a defined process as opposed to success as the result of last minute efforts and the heroics of key individuals. If you are not familiar with this model please go to the Software Engineering Institute at www.sei.cmu.edu and browse the information they have concerning CMM.

The type of standards and documents used in Mil-Std 498, ISO 9000 and CMM are basically the same in respecting to coding. Generally, programmers are expected to following a set of coding standards in an organization. Development efforts consisting of more than a few programmers cannot afford to have several people coding differently. Standards are necessary to ensure that everyone uses the same naming conventions, programming constructs, etc. Consistency in programming reduces maintenance costs. Documentation in programming usually consists of a software requirements document or specification (SRD or SRS), a software design document (SDD), a software version description document (SVD) as well as unit test plans and a wide range of other documentation. Software development organizations view program documentation in a more formal manner.

What you produce for documentation is really dependent on the context in which you're writing a program. At a minimum the source code for all programs should be documented internally with sufficient contents to allow another programmer to understand what you did. Most people begin a source code file with a header block of comments that describes what the code is going to do, who wrote it, the date, etc. Throughout the source file you should comment groups of statements to add clarity to your code. Additionally, you should create a separate document that explains where the code is located, what it is used for, how to run it, and how it is compiled. This level of documentation is appropriate for individual programs that you write for your own use. Programs written for others and those that are part of a development effort require documentation far beyond what you do for your own use. The amount of documentation and the quality of information greatly impacts the cost of the maintenance phase in a development life cycle.

Lecture Notes: Part 3: Example Software Requirements Speciation Document

The following is an example of a Software Requirements Specification Document. The first deliverable for the team project is a document similar to this one but designed for your team's software development project. You may use this document as a template, use another template of our choice, or design your own format. The idea is to include the type of information (and more if possible) that is contained in the following example.

SOFTWARE REQUIREMENTS DOCUMENT (SRD)

1. Scope
The SRD covers those requirements pertaining to the interface, internal operation, and reporting functions of the XYZ program.

1.1 Identification
This document covers version 1.0 of the XYZ program. At this time a formal name has not been identified for the program module.

1.2 System Overview
The XYZ program is designed to provide DAP personnel, managers and software developers with a means to ascertain if specific address records are contained in the XYZ loadables.

2 Referenced Documents
None at this time.

3 Requirements

3.1 General

- The program shall be developed as a 32-bit Windows application using Visual C++ version 6.0.
- The program architecture shall support the multiple document interface metaphor.
- The program shall contain a context sensitive Help facility.
- Context sensitive help shall be provided at the screen and menu levels. It is not necessary to provide context sensitive help at the data element input level.
- The program shall store user define parameters in a parameter file that can be modified by any ASCII text editor.
- Upon startup the program shall attempt to read and make use of the parameter file.
- The program shall validate entries in the parameter file and report to the user if an existing parameter is no longer valid.

3.2 User Interface - Parameter Settings

- The program shall make use of a tab property sheet metaphor accessible from a drop down menu for user defined program settings.
- A "General" property sheet tab shall be created to allow the user to define and store directory settings for basic program operation that include local XYZ directory location and output log/report directories. An example of this type of property sheet is available in the Appendix, labeled Screen Shot #1.
- A "Network" property sheet tab shall be created to allow the user to define and store network settings, user name and password for access to the XYZ loadables, and the NDSS location of XYZ loadables. An example of this type of property sheet is available in the Appendix, labeled Screen Shot #2.

3.3 User Interface - Refreshing Directories

- The program shall utilize a dialog box accessible from a drop down menu that provides the user with a means of selecting how the local refresh on their PC is to occur. An example of this dialog box is available in the Appendix, labeled Screen Shot #3.
- The dialog box shall provide the user with an option of refreshing from the NDSS platform, a local drive/directory, or from a networked drive/directory.
- All refresh options are to present confirmation dialog boxes to the user before the action takes place.
- All refresh options must first determine whether the destination has enough available free space before the action takes place.
- The option to refresh from the NDSS must echo to the user the TCP/IP address of the NDSS machine and the directory on the machine of where the XYZ loadables are located. This information is to come from the parameter settings supplied by the user in previous program options.
- Due to inconsistent FTP problems, the refresh option may have to include a means of transferring data from the NDSS to the VAX Cluster via DECNet "COPY" commands, then an FTP session from the VAX Cluster to the user's local machine.
- The dialog box shall display the target locations of the users local machine of where the information will be located once a refresh occurs.

3.4 Address Record Lookup Functions

- The program shall provide the means for the user to specify whether they are looking for a POB, RRT of SFB record.
- The program shall provide a further depth of options for SFB records that include Firm, Street and Building.
- Input for POB records shall include a distinct field for box numbers.
- RRT record lookup shall allow the user to specify whether they are looking for a Rural Route or a Highway Contract record.
- Input for RRT record lookup shall include a distinct field for route or highway contract numbers.
- SFB record lookups shall provide distinct input fields for house/street number, pre-directional, suffix and post-directional data, as well as the street, building or firm name field.
- Street, building and firm names are to be normalized before they are sent for lookup by using keyword routines from the Normalization Phase of XYZ Directory Generation.

3.5 Output & Report Requirements

TBD

3.6 Directory Test Functions

TBD

3.7 Internal Program Specific Requirements

- The user interface portion of the program shall be separated from the processing or lookup function of the program.
- The directory access and lookup functions of the program shall be written in standard ANSI C.
- The directory access and lookup functions of the program shall be written in a manner that will allow the code to be ported from a Windows based environment to a VAX Alpha based environment.

3.8 Security and Privacy Protection Requirements

There are no specific security and privacy requirements outside of those normal requirements governing use of XYZ data and systems.

3.9 Computer Resources.

- The XYZ program shall be designed to run on an INTEL 586 or greater machine with an NT 4.0 operating system.
- The platform must contain at least 128 megabytes of RAM.
- There should be sufficient free space (over 400 megabytes) to hold the XYZ loadable directories.
- The machine should have sufficient additional free space in order for swap space settings to be increased to 200 megabytes.

3.10 Software Quality Factors

- All code written shall adhere to ESGI's internal coding standards.
- The Windows front end of the program shall follow Microsoft Windows programming standards and conventions.
- All error messages must be clear and concise.
- All error messages must provide the user with information on a possible solution to the problem.
- Error checking must be performed on all input and output actions. Specifically, the program must verify the existence of input/output directories and data files as well as their accessibility regarding READ, WRITE, and UPDATE.

3.11 Design & implementation Constraints

[Describe any existing constraints on the design and implementation of the application (e.g. standards, growth, or changes in technology).]

3.12 Personnel Related Requirements

There are no specific personnel related requirements.

3.13 Training Related Requirements

There are no specific training related requirements.

3.14 Logistics Related Requirements

[Describe requirements on system maintenance, software maintenance, system transportation modes, supply-system requirements, impact on existing facilities, and impact on existing equipment.]

3.15 Other Requirements

No other requirements have been identified at this time.

3.16 Packaging Requirements

- The XYZ program shall be delivered via an installation kit created by the latest version of Install Shield Express.
- The installation kit shall be formed as a single self-extracting .EXE file.
- The installation kit shall be produced on CD-ROM and shall include a sample set of XYZ loadables. The XYZ shall not be part of the self-extracting .EXE file.
- The installation media shall include a "README" text file that instructs the user how to install the software and what the installation kit contains.
- The installation kit shall present the user with a default set of installation directories and parameters.
- The installation kit shall allow the user to change the default program installation location.

3.17 Precedence and Criticality of Requirements

At this time the precedence and criticality of requirements has not been ascertained.

4 Notes

[Include any pertinent general information such as background information, glossary, or rationale. This includes acronyms, abbreviations, terms and definitions.]

5 Appendices

Screen Shot #1
Screen Shot #2
Screen Shot #3

Module Two Team Activities:

- Work with your team to develop a Software Requirements Specification Document.

Module Three

Objectives:

- Data Modeling and Flow Diagrams
- Entity Relationship Models
- Mapping Analysis to Design
- Analysis Document Requirements

Assignments/Activities:

- Requirements Specification Document Due

Lecture Notes: Part 1: Data Modeling and Flow Diagrams

Data Modeling

A way to structure unstructured problems is to draw models. A model is a representation of reality. Just as a picture is worth a thousand words, most system models are pictorial representations of reality. Models can be built for existing systems as a way to better understand those systems, or for proposed systems as a way to document business requirements or technical designs.

What are *Logical Models*?

> Logical models show what a system 'is' or 'does'. They are implementation-independent; that is, they depict the system independent of any technical implementation. As such, logical models illustrate the *essence* of the system.

What are *Physical Models*?

> Physical models show not only *what* a system 'is' or 'does', but also how the system is physically and technically implemented. They are implementation-dependent because they reflect technology choices, and the limitations of those technology choices.

Systems analysts use logical system models to depict business requirements, and physical system models to depict technical designs. Systems analysis activities tend to focus on the logical system models for the following reasons: Logical models remove biases that are the result of the way the current system is

implemented or the way that any one person thinks the system might be implemented. Logical models reduce the risk of missing business requirements because we are too preoccupied with technical details. Logical models allow us to communicate with end-users in non-technical or less technical languages.

What is *Data Modeling*?

Data modeling is a technique for defining business requirements for a database. It is a technique for organizing and documenting a system's data. Data modeling is sometimes called database modeling because a data model is usually implemented as a database. It is sometimes called *information modeling*. Many experts consider data modeling to be the most important of the modeling techniques.

Why is data modeling considered crucial?

Data is viewed as a resource to be shared by as many processes as possible. As a result, data must be organized in a way that is flexible and adaptable to unanticipated business requirements – and that is the purpose of data modeling. Data structures and properties are reasonably permanent – certainly a great deal more stable than the processes that use the data. Often the data model of a current system is nearly identical to that of the desired system. Data models are much smaller than process and object models and can be constructed more rapidly. The process of constructing data models helps analysts and users quickly reach consensus on business terminology and rules.

System Concepts

Most systems analysis techniques are strongly rooted in *systems thinking*. Systems thinking is the application of formal systems theory and concepts to systems problem solving. There are several notations for data modeling, but the actual model is frequently called an entity relationship diagram (ERD). An ERD depicts data in terms of the entities and relationships described by the data.

Entities

All systems contain data. Data describes 'things.' A concept to abstractly represent *all* instances of a group of similar 'things' is called an entity. An entity is something about which we want to store data. Synonyms include *entity type* and *entity class*. An entity is a class of persons, places, objects, events, or concepts about which we need to capture and store data. An entity instance is a single occurrence of an entity.

Attributes

The pieces of data that we want to store about each instance of a given entity are called attributes. An attribute is a descriptive property or characteristic of an entity. Synonyms include *element*, *property*, and *field*. Some attributes can be logically grouped into super-attributes called compound attributes. A compound attribute is one that actually consists of more primitive attributes. Synonyms in different data modeling languages are numerous: *concatenated attribute*, *composite attribute*, and *data structure*.

Attribute Domains

The values for each attribute are defined in terms of three properties: data type, domain, and default. The data type for an attribute defines what class of data can be stored in that attribute. For purposes of systems analysis and business requirements definition, it is useful to declare logical (non-technical) data types for our business attributes. An attribute's data type determines its domain. The domain of an attribute defines what values an attribute can legitimately take on. Every attribute should have a logical default value. The default value for an attribute is that value which will be recorded if not specified by the user.

Relationships

Conceptually, entities and attributes do not exist in isolation. Entities interact with, and impact one another via relationships to support the business mission. A relationship is a natural business association that exists between one or more entities. The relationship may represent an event that links the entities, or merely a logical affinity that exists between the entities. A connecting line between two entities on an ERD represents a relationship. A verb phrase describes the relationship. All relationships are implicitly bi-directional, meaning that they can be interpreted in both directions. Each relationship on an ERD also depicts the complexity or degree of each relationship and this is called *cardinality*. Cardinality defines the minimum and maximum number of occurrences of one entity for a single occurrence of the related entity. Because all relationships are bi-directional, cardinality must be defined in both directions for every relationship.

Strategic Data Modeling

Many organizations select application development projects based on strategic information system plans. Strategic planning is a separate project. This project produces an information systems strategy plan that defines an overall vision and architecture for information systems. Almost always, the architecture includes an enterprise data model. An enterprise data model typically identifies only the most fundamental of entities. The entities are typically defined (as in a dictionary) but they are not described in terms of keys or attributes. The enterprise data model may or may not include relationships (depending on the planning methodology's

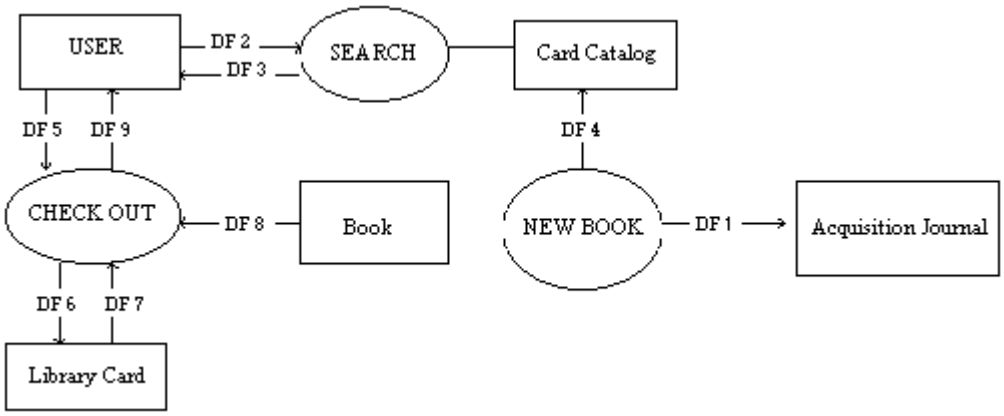
standards and the level of detail desired by executive management). If relationships are included, many of them will be *non-specific.* The enterprise data model is usually stored in a corporate repository.

Data Modeling During Systems Analysis

The data model for a single system or application is usually called an application data model. Logical data models have a date focus and a system user perspective. Logical data models are typically constructed as deliverables of the study and definition phases of a project. Logical data models are not concerned with implementation details or technology; they may be constructed (through *reverse engineering*) from existing databases. Data models are rarely constructed during the survey phase of systems analysis. Data modeling is rarely associated with the study phase of systems analysis. Most analysts prefer to draw process models to document the current system. Many analysts report that data models are far superior for the following reasons: (1) Data models help analysts to quickly identify business vocabulary more completely than process models. (2) Data models are almost always built more quickly than process models. (3) A complete data model can be fit on a single sheet of paper. Process models often require dozens of sheets of paper. (4) Process modelers too easily get hung up on unnecessary detail.

Applying the Concepts

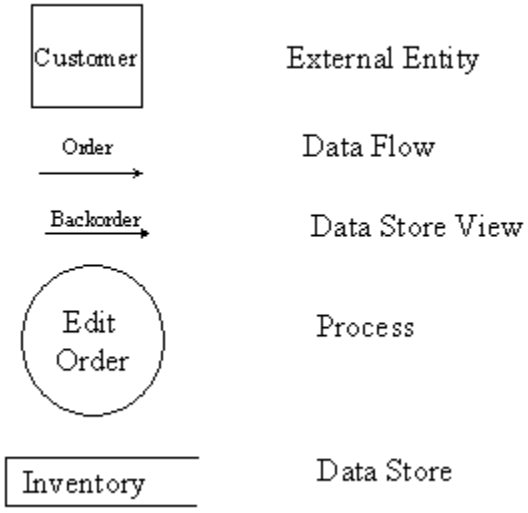
A Data Flow Diagram (DFD) Level 0 is a single process bubble with all inputs to, and outputs from, the system represented. Commonly called a 'Context Diagram', it includes providers of data to the desired system and users of the information formulated from that data as external entities. A sample Context Diagram is shown below.



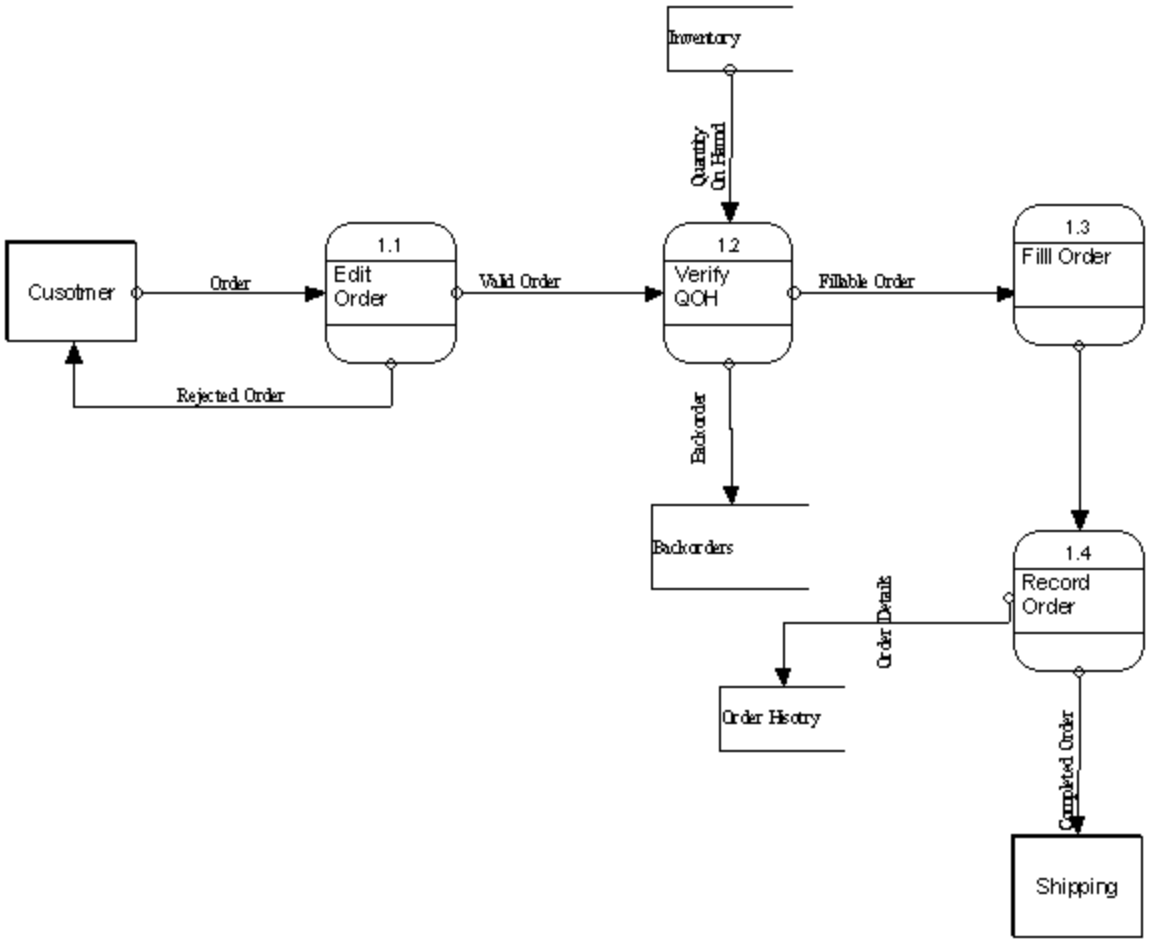
Data Flow Diagrams (DFDs)

The DFD is a tool that allows us to model the relationships among processes in terms of the data that is passed between them. The DFD is the most popular tool used for process modeling with traditional systems.

The components of the DFD are as follows:



While the Data Flow arrow and the Data Store arrows look alike, they can be differentiated by the way they are connected to the other components. The Data Store View arrows always interface at one end or the other with a Data Store as shown in this example. The Data Flow arrows always interface between External Entities or Processes. Data Store Views represent either the requirement of a process to retrieve stored data in order to accomplish its function, or the capture of data that will be needed later by some other process. It is important to understand that the sum of the content of the Data Store Views correlates directly to the requirements for data in the ERD, and ultimately the database itself.



DFDs are built in 'leveled sets' that begin with a single process and are progressively portioned until the required detail is documented. Sometimes this means literally thousands of processes. For now, let's assume that the diagrams we built covered the entire scope of our system. If this were true, it would be considered a 'level one' diagram. A context diagram also represents the entire system, but shows it as a single process with all of the inputs and outputs interacting with this single process. Often, but not always, the analyst will begin with the context diagram and build hierarchically from there.

The Context Diagram represents our contract with the user. Essentially we are saying that the user wants a system that creates a given set of outputs using a given set of inputs. If our system accomplishes this, the user should be happy.

Process Specifications

Data Flow Diagrams identify the processes and how they are related in the context of the data that is shared among them. However, at this point the details necessary to build the system or to understand just how these processes can be performed is not available. These process details are contained in process

specifications. Each process bubble on a DFD is related to a formal process specification that provides more details about that process. The sum of all of the process specifications for a system provides all of the details about that system.

Process specifications are sometimes referred to as transformation descriptions since they effectively describe the transformation of data that occurs in a process bubble. Every process on a DFD should have a corresponding process description. The nature of these specifications will vary depending on whether the process it describes represents a primitive level process or some higher-level process (anything above the primitive level).

At all levels above the primitive level the process specification is a general description that enables the reader to interpret the diagram and navigate their way to lower level processes.

Levels of Process Specification

A distinction may be made between Process Descriptions at the lowest, or primitive level of a set of DFDs, and those of all higher levels. At the Primitive Level, the Process Specification must provide all of the details about the processes necessary for the designers and coders to build the software. The amount of detail provided must be sufficient to allow a person to accurately create the outputs from the project if they were given a copy of each of the inputs to the process.

Above the primitive level the Process Specifications are more general in nature and are intended to help the reader navigate their way through the set of DFDs and find the particular Primitive Level Processes of interest. The large hierarchy that is commonly created when large systems are modeled necessitates them.

Process Specifications can be documented in a variety of ways. For the higher levels simple narrative is most commonly used. For Primitive Levels, where specific business logic must be clearly spelled out, it is often appropriate to use tools such as flow charts, decisions tables, and structured English.

Consider a DFD that contains a process called ‘Process Student Registration.’ Assume that there is also a lower level diagram that represents the details of this process - processes such as "Verify Registration On Time,’ ‘Verify Prerequisites Met,’ ‘Verify Slot Available,’ ‘ Place on Standby List’ and ‘Record Registration’. Since ‘Process Student Registration’ is not a primitive level process, the associated process description could contain a general description, such as the following:

PROCESS STUDENT REGISTRATION

This process receives registration requests from students, ensures that they are valid, and records them in the database.

This description lets us know what it going on in the process and enables us to interpret the process model of which it is a part. However, it would not be sufficient to enable us to actually perform this process. Provided lower level processes and descriptions accomplish this goal, it is not a problem.

Once we get to the primitive level the specification becomes much more precise. Software developers use the specifications from this level to design and code the system. All of the business rules for transforming the data must be documented here. The best guideline for writing a primitive level process specification is as follows: The specification should be written such that if an individual were given the process specification and an instance of each input to the process, they would be able to create each of the outputs correctly.

The implications of this guidance are significant. Historically it is very difficult to get analysts to even approach this level of detail. It is often argued that achieving such detail is simply too labor intensive. The counter argument is that the developer is trying to build a system that uses computer software to accomplish some function. Computers know only what we tell them, so we must tell them everything in great detail. Since the software engineer is writing this software, we must tell them everything as well. If we don't document the details in the process specification, where will they come from? How will the software engineer be able to complete the coding process? Failure to provide this level of detailed specification is one of the contributors to software project failure.

Our business rule for Registration Requests is that they must be submitted two weeks before the start of the class. Here is a sample Process Description for 'Verify Registration On Time':

Verify Registration On Time

- Get the Class Start Date from the Class table.
- Subtract 14 from the Class Start Date to determine Latest Registration Date.
- Get the Date the Registration Request was received by the Registrar from the Registration Request.
- If this date is not on the request, return it as incomplete.
- If the Date the Registration Request was received is later than the Latest Registration Date, reject the Registration Request as late.

This description is clear and detailed. It clarifies that the date we are concerned with is the date the Registration Request is received, not the date the Request is filled out or any other date associated with the Request. It clarifies the often-sticky issue of whether a request that two weeks is exactly early will be accepted (it would). It also clarifies what to do if part of the required data is missing.

If we were to hand someone a stack of Registration Requests, give them access to the data in the Class table, and provide them with this Process Specification, they should be able to accurately determine which requests should be accepted and which should not.

Let's look at a less precise process description for calculating gross pay in a payroll system:

Calculate Gross Pay

- Gross Pay is equal to Hours multiplied by Pay Rate.
- If Hours are Overtime Hours, pay 50% premium.
- If Hours are for Evening Hours pay 10% premium.

This looks pretty clear, and pretty detailed. However, what constitutes Overtime Hours? Is it hours over 40? What are Evening Hours? When does the Evening start? Actually, if we did a good job of defining our data in the Data Dictionary we could probably find answers to both of these questions. But how much should we pay if some of the hours are both Evening and Overtime? Is it normal pay + 50% + 10%? Or perhaps it is normal pay times 150% times 110%. Or, perhaps the business rule is simply to give them the greater of the two premiums. If we were to give someone this Process Specification and a sample transaction with this condition they might not know what to do. Worse yet, two different people given the same information might do two different things. We can't afford this type of ambiguity.

The bottom line for process specifications is that if you provide a narrative that describes very specifically what you want from the system, you will get a system that does very specifically what you want it to do. If you provide narratives that describe in general what you want from the system, you are likely to get a system that generally does what you want it to. The choice should be clear.

Structured English

Structured English (sometimes called Pseudocode) resembles a simple narrative, but 'structures' conditional logic in such a way as to make it clear. The advantage of Structured English is that it is easily understood by a variety of users while providing the specificity that software developers require. For example:

When a student tries to register for a class, the following process is followed:

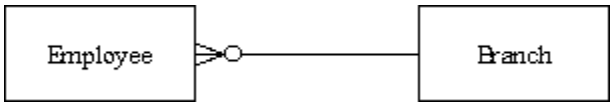
```
For each Form 17
If Class Date is not two weeks or more away
        Reject as too late
Else
        Check to see if Student has prerequisites for Course
        If Prerequisites not met
                Reject as Prerequisites not met
        Else
                Check for Available Slots
                If Slots not Available
                        Reject for No Room
                Else
                        Register Student
```

Lecture Notes: Part 2: Entity Relationship Models

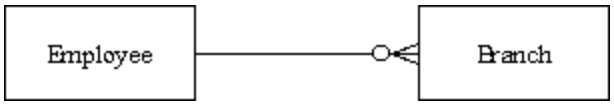
Entity Relationship Diagrams (ERDs) are the primary tool for representing the data associated with a software application development effort. The ERD represents all data that is input, stored, transformed, and produced within a software application. ERDs consider data independent of the processing that transforms that data. They represent data in a logical way that can be translated into any number of physical structures.

Building ERD's in Analysis

Assume we are told that an employee works for one branch (part of a company) and that each branch can have many employees. The relationship between employee and branch would look like this:



However, we may have employees who are fire marshals. These fire marshals are responsible for performing periodic inspections on their own and other branches to make sure they are practicing proper fire safety and to make sure that they follow proper procedures during fire drills. An employee can be a fire marshal for several branches, and each branch can have only one fire marshal. We would model this relationship like this:



So which relationship is correct? Actually, both are correct, and both need to be named and modeled if the relationship has some relevance to the system we are implementing. (Perhaps we have a requirement that a 'Fire Marshal report' be produced). The main point here is that the relationships are not always obvious and that it is the role of the analyst to identify all relevant relationships and model them appropriately.

Consider the relationship between a teacher and a student at a University. How many relationships are there? Teachers teach students. Teachers can be advisors for students. Teachers can be the parents of students (and therefore eligible for discounts on tuition).

Do we model each of these relationships on our ERD? Not necessarily. We only model the relationship if it is relevant to the processes we plan to implement. This is what we are trying to accomplish during the analysis stage of software development. Identify all of the entities, and relationships among those entities, that are relevant to the system we are going to build.

Relationship of ERDs to data stores on the DFD

When we build our DFD's we include a symbol called the Data Store. Conceptually, the data represented by the Data Stores in a DFD and the data in an ERD are the same. However, the ERD representation is superior because it focuses on the natural structure of the data.

The individual Data Stores in a DFD are tied to specific processes and represent a more limited, process oriented view. They are necessary so that we can represent how the process interacts with the stored data of the system and understand those processes fully. However, any database structures we build later will be built from the ERDs, not the DFD's. Sometimes analysts can get bogged down trying to show the structure of the data on the DFD. That isn't necessary, as we have already captured this structure on the ERD.

In fact, it is not an effective use of time to focus too closely on the specific Data Stores at all. Name them quickly to make the diagrams readable and use your time elsewhere. Focus instead on the content of the Data Store Views. The individual view for each process may tie together multiple files - or entities. This can be documented in the definition of the Data Store View by including each of the specific elements that are retrieved and the key that is used to retrieve that data. We can then look at the various entities where this data is stored and determine what the relationship is for that particular process.

For example, let's look at a simple inquiry process. In this process an Advisor submits an inquiry that contains only a Student ID. In response they want to get the name and address of the student, along with the name of each of the courses the student has taken, the date of the class, and the name and phone number of the teacher who taught this class.

This data would be spread out over several entities, including Student, Course, Class (an occurrence of a Course), and Instructor. When building our DFD we want to model the relationship between the 'Process Advisor Inquiry' process and the stored data of the system. Do we need to show several Data Stores? No. We can simply show one Data Store View and define it with a structure that shows the key and the related data. Here is how that might look for a Data Store View that we chose to call Student Inquiry Details:

Student Inquiry Details
*Student ID (key)
Student Name
Student Address
Course Name
Class Date
Instructor Name
Instructor Phone Number

We show this arrow leaving the 'Process Advisor Inquiry' process bubble, but where does it go? The answer is a Data Store, but which Data Store? We don't need to show four different Data Stores and how they are related. Just make up a Data Store name that makes the diagram readable (Student Inquiry Data, or simply Stored Data, may work). Let the structure of the date remain in the ERD where it belongs.

Commonly, the ERD is developed to support multiple applications. In these cases it is not uncommon to see data on an ERD that is not referenced by a particular set of DFD's. However, it is a problem if the data required by any given DFD process is not on the ERD. The best way to determine if all of the required data is present is to verify that the data on each of the Data Store Views can be retrieved. It is the Data Store Views that provide the link between our data and process models.

This step involves identifying the key for each of the Data Store Views and checking to make sure that the particular key structure exists in the ERD and can be used to retrieve any required data elements. This is a tedious and time consuming process if done manually, but is virtually automatic when using a robust CASE tool.

Data Flows and Data Store Views

Like processes, the data components on DFD's (data flows and data store views) simply represent the data - they do not describe precisely the data that is represented. To determine this we must look beyond the DFD to the data definitions. Every Data Flow and Data Store View must be defined in terms of its data elements. These definitions are stored in the Data Dictionary.

If we look back at our previous DFD example we will find a Data Flow entitled 'Time Card.' The Time Card is submitted at the end of the pay period so the employee may be paid. It must identify who is being paid and how many hours they worked. While we may have some idea of what this means just from the name Time Card, it is important that we define it precisely if we are to use it in constructing software.

If we consider the purpose of the Time Card we could conclude that it would normally include the Employee ID (who are we paying), a Date (what pay period are we paying them for), and the number of Hours (how much work will we be paying them for).

Our initial Data Flow Definition might look like this:

Time Card
Employee ID
Date
Hours

To define this data flow for inclusion in our formal specification we need to be precise in our naming. It should be clear what each data element is and what it describes. We should preclude the potential that data elements may be confused with one another, particularly in very large systems.

The name 'Employee ID' is pretty good. It is the ID (or Identifier) of the Employee. We won't need to change this one.

But what about 'Date?' What date are we referring to? What does this date describe? Is this the date the Time Card was submitted or the date the date they pay period ends? Will we be able to distinguish this date from all of the other dates in the system? This date happens to be the last date of the pay period, so let's make that clear by renaming it as 'Pay Period End Date.'

Now what about 'Hours?' We can probably figure out that this is the number of Hours that the employee worked, but why take a chance that someone might misinterpret it? Let's make it clear by renaming this element 'Employee Hours Worked.'

Names sometimes get pretty long when we start to describe them as we have above, especially in large systems where many transactions contain similar data elements. However, this is a small price for effectively communicating our requirement.

Time Card might now be defined as:

TIME CARD
Employee ID
Pay Period End Date
Employee Hours Worked

One additional item that we might add to this definition is an indication of the 'key.' We will have many Time Cards flowing into our system. What is it that makes each Time Card unique? Your first thought might be the 'Employee ID.' This would be unique for each employee, but over the course of time we are going to pay each employee many times (every week, every two weeks, every month, or whatever the pay cycle turns out to be). To uniquely identify every occurrence of this data flow we would need to use the combination of Employee ID and Pay Period End Date. We should never have two transactions with this same combination of values.

The key is often indicated with an asterisk before the data element names such as this:

TIME CARD
*Employee ID
*Pay Period End Date
*Employee Hours Worked

Keys are important, especially for Data Store Views. It is the key that identifies how we want to access the data and can be used to build SQL-like statements for data access. All of our Data Flows and Data Store Views must be defined in this fashion.

This is just the start of our data definition effort. Next we must make sure that every data element is itself defined. There are several components to the data element name. At a minimum we should include the length, possible values or validation criteria, data format, any aliases, and a brief narrative description.

Let's do this for Employee Hours Worked.

Employee Hours Worked would be a two position numeric field. Can it contain any numeric value? Probably not. We might not want any Time Cards with a value of zero, so we could set a lower limit of 1.

We might have an upper limit as well. If this is a weekly Time Card there are only 168 hours (24*7) in the week. However, a more reasonable limit might be 80 or 90, since no one can really work 168 hours. Of course, if we set the limit too low and someone works an incredible number of hours in a heroic effort to complete a task on time it would be a shame for his or her Time Card to reject. So what do we do?

The answer is that we follow the business rule that the company follows. Validation criteria need to be documented in the requirements specification, but they are defined first by the user of the system. Talk to the customer, find out what the business rules are, and capture them in the data definition. The purpose of the specification is to communicate business rules between the customer and the software engineer.

As for the narrative, this allows us to say more about the data element than we could reasonably put in the name itself. In this case we might simply say: "The number of hours an employee works in a given pay period." This is simple, clear, and absolutely essential for large complex systems.

Finally, we mentioned something about an alias. An alias is simply another name that can be used to reference a data element. There are two types of aliases.

Functional aliases are names that are used in the business world. It may be that the end user commonly refers to Employee Hours Worked as Payroll Hours. That's OK. They don't have to use our name, and we probably couldn't get them to change if we tried. But we do want to capture their name as an alias and enter it into our automated dictionary tool. That way if someone is researching the definition they will be sure to find it. Alias support is a common function of any robust data dictionary tool.

Technical aliases are names that are used in the implementation of software. The programming language that we use may have restrictions that prohibit the use of long names (some early languages limited data names to 8 characters). Or, we may simply find it cumbersome to enter lengthy names during the coding process (not to mention the increased opportunity for typos that long names allow). Technical aliases allow us to define any name we like to reference a data element. Provided we enter it as an alias in the data dictionary our alias support feature will always tie us back to the proper definition and business rules.

As you might guess, uniqueness is an issue for both the official name of a data element and all of its aliases. If we find ourselves trying to use the same name to refer to two different things, no automated tool is going to be able to help us sort out the confusion. A good dictionary will not allow you to enter duplicates.

If you follow the steps above you will come up with pretty good definitions for all of your data. You will capture the user's requirements completely. You will also

capture all of the information that the developer needs to create the databases and program code.

Lecture Notes: Part 3: Mapping Analysis to Design

Architecture

Architecture associates the system capabilities identified in the requirements specification with the system components that will implement them. Architecture is an often-overused term that has many definitions in software design. In general there are three reasons to explore architecture. First, we must analyze the effectiveness of the design in meeting its state requirements. Second we must consider architectural alternatives. Finally, we must reduce the risk associated with the construction of the software.

Modularity

Modularity is the organization of software into separately named and addressable components. Modularity is the single attribute of software that allows a program to be intellectually manageable.

Functionally Independent Module

Modules with single-minded functions, with simple interfaces, that are easy to develop and maintain, and with reduced error propagation and increased reusability are considered functionally independent. Ideally all modules within a design should be functionally independent, although this goal is unrealistic.

Transform Analysis

Transform Analysis describes one method of translating processes on a DFD to a design structure. In Transform Analysis the source processes represent steps in the transformation of a single transaction. Generally, Transform Analysis occurs at a lower level of design from Transaction Analysis.

Transaction Analysis

Transaction Analysis describes a second method of translating processes on a DFD to a design structure. In Transaction Analysis the source processes represent discrete transaction and lead to a design that organizes around these transactions. Generally, Transaction Analysis precedes Transform Analysis.

Design Patterns

A Design Pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable design. A Pattern must identify the components, their roles, and their relationships. A Design Pattern leads to reusability of software by allowing previously developed structures to be used to solve similar, but new, problems.

Requirements Traceability

Requirements Traceability describes the ability to specifically identify where each of the lowest level requirements are implemented within a design. Essentially, primitive level processes can be mapped to the specific module where that process is implemented. Requirements Traceability supports change management throughout the development process.

Boss Module

A boss module is a management module within a structure chart that does not execute the functionality of the module, but rather controls the activities of other modules. The higher levels of modules within a design structure tend to be boss modules. A goal of an effective design should be to separate work and management and to limit the functionality of boss modules strictly to the control of lower level processes.

Control Couple

A control couple is an item of information that is passed between two modules that controls the activities of a module or passes information that describes what has occurred in a module in response to a call from another module. Control couples are a direct reflection of the cohesiveness of the design. A certain level of control passing is necessary for any system, but should be minimized.

Data Couple

A data couple is a data item that is passed from one module to another so that the receiving module can use the data to accomplish its function. One goal of an effective design is to limit the passing of data among modules to only that data that is specifically needed by the receiving module. Modules within a design are commonly reorganized to minimize data coupling.

Verification

The term Verification refers to the concept of determining that we are building the product correctly. It assumes that the requirement is stated and interpreted properly and focuses on the way we have implemented that requirement.

Validation

The term Validation means to determine whether we are building the correct product. Validation ascertains that we correctly understand the requirement.

Lecture Notes: Part 4: Analysis Document Specifications

The following is an example of an Analysis Document. The second deliverable for the team project is a document similar to this one but designed for your team's software development project. You may use this document as a template, use another template of our choice, or design your own format. The idea is to include the type of information (and more if possible) that is contained in the following template.

Your final analysis document should be one complete document, not separate files. You can import to MS Word from other software packages. There must be a table of contents and the pages must be numbered. Your analysis document needs to include the following Modules and content:

1. Introduction

   The Introduction Module should be written so that anyone reading it can have a good idea of what the project is intended to accomplish. This entire Module is normally completed in narrative format (no models required).

   Specific Project Objectives

   You will be asked to create data and process models for a project with a fairly broad scope. It would not be reasonable to code the entire project during this term. Therefore, you will be asked to develop a working prototype for this project. Your objectives statement should clearly describe the objectives for both the documentation of the broad project deliverable and the narrower working prototype.

Overall Requirement Description

A general description of what the software is to do and a list of the high level requirements (taken from your Requirements Specification). This Module does not need to include detailed business rules. Rather, it should summarize the requirements at a high level so that the general scope of the project is clear.

Project Constraints & Assumptions

A list of constraints that may limit what you are able to accomplish or how you can accomplish it. The fact that you have a limited amount of time to complete the project would be considered a constraint.

Create a list of assumptions that you are making about the project. It is not practical, or necessary, to clarify every point about the project with the instructor. For minor points, you are welcome to make assumptions, provided that the assumptions are reasonable and do not change the purpose of the assignment. For example, if the project required that you calculate discounts for customer orders over a certain dollar threshold, and the threshold was not specified, it would be acceptable to 'make up' a threshold. When you make such assumptions, document them in this Module.

2. Functional Requirements

Entity Relationship Diagram (ERD)

ERD showing all entities involved with the entire scope of the project, not just the prototype. The database delivered with the prototype must be consistent with the ERD. Entity Descriptions in narrative format that clarify what each entity represents to someone who might not be familiar with the business area being modeled. Attribute Definitions for each of the attributes in each entity. Should include length, data type, possible values, edit criteria, and a unique data element name.

Data Flow Diagram (DFD)

Must cover the scope of the entire project. Must include a Context Level diagram and one or more lower level diagrams as needed to represent a Process Bubble for each major process within the entire proposed system. Must include a Primitive Level process bubble for every process in the working prototype. Consistency of data between levels must be maintained. No more than nine processes should be present on any given diagram.

Process Descriptions

Every process on every DFD must have a corresponding process description. Process Descriptions for primitive level processes must be supported by descriptions that meet the criteria of "enabling the transformation of the inputs into outputs". Primitive Level Transform descriptions should be documented in Structured English, Pseudocode, or some comparable mechanism such that all business logic is unambiguous.

Design Specification

You need to include a design architecture description. This is the identification of the development tools to be used and the hardware and software platforms that the system will operate on. Keep this Module brief. The next deliverable is the actual Design Specification Document.

Performance & Reliability Requirements

Identification of relevant performance requirements, such as the number of transactions that will be processed during any given time period and the expected response time. (This will require assumptions on your part as to what reasonable performance requirements might be for this type of application.)

Component/Deployment Diagrams

You need to provide representation of the hardware and software components in a UML format.

Test Plan

Test Case Grid in specified format. Grid should identify at least 15 test transactions. Content of each transaction, including specific input values, must be identified. Predicted results should be included.

Module Three Team Activities:

- Work as a team to develop the Analysis Document

Module Four

Objectives:

- Traditional versus OO Development
- Object Oriented Analysis
- Unified Modeling Language (UML)

Assignments/Activities:

- No deliverables

Lecture Notes: Part 1: Traditional versus OO Development

The move from traditional to OO development represents an incremental change for data and a radical change for processes. The Entity identified in traditional development corresponds roughly to Classes in an OO system. However, the processes that are traditionally organized into hierarchies to accomplish specific functions are arranged much differently in an OO world. These processes are associated directly with objects as 'operations' that must collaborate to accomplish the same functions that are recognized occur as single processes in traditional systems.

While the tools and languages we have used to build software have evolved over time, the fundamental way we went about organizing our software has remained essentially the same. We organized processes into hierarchical structures (systems, subsystems, programs, modules) and these process structures interacted with separate data structures (databases).

This fundamental organizing principle changes when we build OO software. An object includes both the data and the operations that manage that data. In building objects we essentially wrap the processing around the data to form the object. This 'encapsulation' restricts action against data to the operations that are part of the object.

When building OO systems our software engineering focus is on identifying these objects and their operations. The structure is no longer hierarchical, but is based on objects and their interactions. For simple 'get' and 'set' operations this is a pretty straightforward process. But what about a function like 'Registering a student for a class.' Is this an operation of the student or the class? Or, is it an operation of an object called a 'Registration?'

To resolve these issues we use different tools than we did in traditional software development. Instead of DFDs and ERDs we use UML models Class Diagrams, Sequence Diagrams, and Use Case Scenarios. The steps in the process we follow are different as well. Analysis and design overlap much more than they do in traditional development. The same models are often used throughout the software engineering process.

The change to OO development is an evolutionary change from the perspective of the data, as can be seen in the similarities between class diagrams and ERDs. However, the change for processes is dramatic. Processes are no longer modeled in hierarchies independently of the data, but become one with the objects of which they are a part.

New terms for the phases of software engineering have been introduced. Instead of analysis, design, coding, and testing, we have inception (defining the business rationale for the project), elaboration (high level analysis and design and the construction of a plan that drives construction), construction (the rough equivalent of coding), and transition (includes not only testing, but also training and implementation).

Since the same models are used in various phases of the process we differentiate not by the models we use, but by the perspective we take in building those models. There are three perspectives: conceptual, specification, and implementation.

In the conceptual phase we represent the concepts in the domain of study. In a training application we might model the student as a concept, representing the student as a conceptual class related to other conceptual classes. We also capture the basic operations that the student will carry out without worrying about how other classes will invoke those operations.

As we move to the specification perspective we document the interfaces among classes. We are not yet ready to implement the operations of the student class, but we will define the specific interface that other classes must use to invoke those operations. This enables us to move forward with the construction of other classes that use those operations.

Finally, with the implementation perspective we model the specifics of how attributes and operations will be implemented in our final solution. These models can be used to construct the object itself.

So, while the goal of software engineering is the same for OO and traditional systems the way we go about achieving that goal differs. Many of the differences are subtle, while others are quite dramatic.

Lecture Notes: Part 2: Object Oriented Analysis

The following are common object oriented terms and concepts:

Objects

An object is like a module that contains both data and the instructions that operate on those data. Data are stored in instance variables; the instructions for their manipulation are defined inside methods. Methods are explained in the "Messages and methods" Module below. An object in programming can be compared to a real world object. For example, a desk is an object that has the following characteristics:

- color
- dimensions
- parts (e.g. drawers)
- etc...

and we can perform the following operations on those characteristics:

- add part
- remove part
- change color
- etc...

Messages and Methods

A message is a request to an object to act in a specified way. Whenever an object-oriented program executes, objects exchange messages and data among them. The message may contain information to clarify the request, but the sender does not need to know how the receiver will carry out the request; it only need to know that it is carried out. For example, if a document object receives a print request, it may send a message to the printer object; the printer object may in turn send a message to the document object requesting print data, and so on.

A method is a procedure or function that determines how the object will respond to the request. Methods provide the only way to manipulate the instance variables of the object. They may also send messages to other objects requesting action or information.

Classes and Instances

A class is a description of almost identical objects. It contains data and methods that determine the common characteristics of a set of objects. It is really an

abstract definition of an object. Sending a creation message to the class creates an object. This new object then uses the data and methods of its parent class. We can now say that an object is an instance of a class. A class can also be used to define new classes. However, this falls under inheritance, which is described in detail in the next Module.

Object Oriented Analysis

The object-oriented programming paradigm has become so popular that now virtually every new programming language or environment labels itself as "object-oriented" or "object-based." What does it really mean for a language to be object-oriented? What are the important differences between object-oriented languages? Object based programming involves four general design concepts and can be remembered by thinking of "**A PIE**."

- **A**bstraction
- **P**olymorphism
- **I**nheritance
- **E**ncapsulation

Abstraction

Abstraction provides a way to decompose a complex system into a bunch of smaller more manageable parts, simplifying the development process. We can think of a computer as counting many levels of abstraction. We generally think of a computer as a "box", but the CPU case contains many components that work together in order for us to use it. That is one level of abstraction. Beyond that, each computer component, like a video card, has circuits and chips which provide yet another level of abstraction. Most people (nowadays) know how to use a computer, but as we move to lower and lower levels of abstraction, we see more complexity and probably a decreased understanding on the part of these same people.

Polymorphism

Objects are smart enough to be dynamic. Specifically, they have the ability to react differently depending on the situation. This is important because it makes the API (application programming interface) dependable and easy to understand for the incoming developer. Consider this example: Suppose you have a method called "print()." The first question you might ask is, "what does this method print: pictures, text, or what?" Using "polymorphism," an object can be made to handle any scenario with the exact same method name. Depending on what the object is asked to print, it will be able to print it. Instead of having separate methods like printPicture(), printLetters(), and printNumbers() you have one unified print() and the object itself determines how to handle the different types of situations.

Inheritance

Reusability of code is always been a matter of concern in programming. In imperative languages, it is not just a matter of reusing; it is also a matter of altering part of the code to adapt to a new situation. This usually requires copying and pasting the original code to a new module and performing necessary changes to it. However, if the original code is large, there are equally large chances of making errors; and it often leads to having identical data and procedures from more than one file, producing identical object modules for linking to the final executable. Clearly a waste of resources.

In OO languages, effective reusability of code is succeeded using the concept of inheritance. An existing class can be used to derive a subclass, which will have all the functionality of the original class. This new class can then have new features added, unnecessary features removed and existing features altered, without of course affecting the original class.

This may sound like simply creating a copy of the old class, just as we would do with a module in an imperative language. However, this is not what happens internally. What really happens is that the implementation of the old class is NOT duplicated! Instead, the new class knows that the implementation already exists in its parent, so it simply uses it from there. In other words, there is a binding between the two classes, thus avoiding any compilation of code that need only be compiled once.

As we can see, inheritance plays an important role in software reusability. Instead of having to solve every new problem from scratch, we can simply build on previous solutions and adapt them to the new problem. It is similar to having several related modules, but which are implemented by describing their differences from a common module rather than rewritten. Why should we do what we have already done?

Encapsulation

Objects are "little black boxes of functionality." Nobody but the object itself needs to know anything about how its properties and methods are defined and implemented. Is the list of items stored in an array or a vector? How is sorting handled, with a quick sort or a bubble sort? How is a selection marked and how do you handle multiple selections? Encapsulation means that the answers to all these questions are private, known only by the object itself.

The benefit is that if you want to use a "Select Box," you do not need to deal with all of the complex code that handles all of the functionality of a select box. Instead, you just put the self-contained select box object in your application and use it. This is an incredibly useful concept because it means that it is far easier to

modify and understand code because you only need to deal with small pieces of code at any one time. As a developer, you do not need to deal with the intricacies of the select box functionality you just use the thing! It is also a good metaphor for the real world that can be thought of as being made up of encapsulated objects. Consider the computer you are using to read this. Do you know how the CPU works? Most likely you don't. But that is fine. It works regardless. And the fact that you don't have to spend time learning electrical engineering means that you are free to spend your time building things "using" the CPU.

Using Non Object-Oriented Languages

It's also possible to use objects and messages in plain old non-object-oriented languages. This is done via function calls, which look ordinary, but which have object-oriented machinery behind them. Suppose we added a "plus" function to a C program:

```
int plus(int arg1, int arg2)
{
  return (arg1 + arg2);
}
```

This hasn't really bought us anything yet. But suppose instead of doing the addition on our own computer, we automatically sent it to a server computer to be performed:

```
int plus(int arg1, int arg2)
{
  return server_plus(arg1, arg2);
}
```

The function server_plus() in turn creates a message containing arg1 and arg2, and sends this message, via a network, to a special object that sits on a server computer. This object executes the "plus" function and sends the result back to us. It's object-oriented computing via a back-door approach!

This example is not very fancy, and, of course, it's easier to simply add two numbers directly. But as the example illustrated, there's no limit to the complexity of an object. A single object can include entire databases, with millions of pieces of information. In fact, such database objects are common in client-server software.

Object-Oriented Programming (OOP)

In object-oriented programming, objects of the program interact by sending messages to each other. The previous Modules have already introduce some

"object-oriented" concepts. However, they were applied in a procedural environment or in a verbal manner. In this Module we investigate these concepts in more detail and give them names as used in existing object-oriented programming languages.

Abstract Data Types (ADT)

An abstract data type (ADT) is a data type defined only in terms of the operations that may be performed on objects of the type. Users (programmers) are allowed to examine and manipulate objects using only these operations and they are unaware of how the objects are implemented in the programming language. The programmer defines the data type - its values and operations - without referring to how it will be implemented. Applications that use the data type are oblivious to the implementation: they only make use of the operations defined abstractly. In this way the application, which might be millions of lines of code, is completely isolated from the implementation of the data type. If we wish to change implementations, all we have to do is re-implement the operations. No matter how big our application is, the cost in changing implementations is the same. In fact often we do not even have to re-implement all the data type operations, because many of them will be defined in terms of a small set of basic core operations on the data type. Object-oriented programming languages must allow implementing these types. Consequently, once an ADT is implemented you have a particular representation of it available.

Consider the ADT Integer. Programming languages such as Pascal, C, Java and others already offer an implementation for it. Sometimes it is called *int* or *integer*. Once you've created a variable of this type you can use its provided operations.

For example, you can add two integers:

```
int i, j, k;    /* Define three integers */

i = 1;          /* Assign 1 to integer i */
j = 2;          /* Assign 2 to integer j */
k = i + j;      /* Assign the sum of i and j to k */
```

Let's play with the above code fragment and outline the relationship to the ADT Integer. The first line defines three instances i, j and k of type Integer. Consequently, for each instance the special operation constructor should be called. In our example, this is internally done by the compiler. The compiler reserves memory to hold the value of an integer and "binds" the corresponding name to it. If you refer to i you actually refer to this memory area which was "constructed" by the definition of i. Optionally, compilers might choose to initialize the memory, for example, they might set it to 0 (zero).

The next line

```
i = 1;
```

sets the value of i to be 1. Therefore we can describe this line with help of the ADT notation as follows: Perform operation set with argument 1 on the Integer instance i. This is written as follows: i.set(1).

We now have a representation at two levels. The first level is the ADT level where we express everything that is done to an instance of this ADT by the invocation of defined operations. At this level, pre- and post-conditions are used to describe what actually happens. In the following example, these conditions are enclosed in curly brackets.

```
{ Precondition: i = n where n is any Integer }
i.set(1)
{ Postcondition: i = 1 }
```

Don't forget that we are currently talking about the ADT level. Consequently, the conditions are mathematical conditions.

The second level is the implementation level, where an actual representation is chosen for the operation. In C the equal sign "=" implements the set () operation. The ADT operation set is implemented. Let's stress these levels a little bit further and have a look at the line

```
k = i + j;
```

Obviously, "+" was chosen to implement the add operation. We could read the part "i + j" as "add the value of j to the value of i", thus at the ADT level this results in

```
{ Precondition: Let i = n1 and j = n2 with n1, n2 particular Integers }
i.add(j)
{ Postcondition: i = n1 and j = n2 }
```

The post-condition ensures that i and j do not change their values. Please recall the specification of add. It says that a new Integer is created the value of which is the sum. Consequently, we must provide a mechanism to access this new instance. We do this with the set operation applied on instance k:

```
{ Precondition: Let k = n where n is any Integer }
k.set(i.add(j))
{ Postcondition: k = i + j }
```

As you can see, some programming languages choose a representation that almost equals the mathematical formulation used in the pre- and post-conditions. This makes it sometimes difficult to not mix up both levels.

Programmer-Defined Classes

A class is an actual representation of an ADT. It therefore provides implementation details for the data structure used and operations. Lets play with the ADT Integer and design our own class for it:

```
class Integer {
attributes:
  int i

methods:
  setValue(int n)
  Integer addValue(Integer j)
}
```

In the example above as well as in examples that follow we use a notation that is not programming language specific. In this notation class {...} denotes the definition of a class. Enclosed in the curly brackets are two Modules attributes: and methods: which define the implementation of the data structure and operations of the corresponding ADT. Again we distinguish the two levels with different terms: At the implementation level we speak of "attributes" which are elements of the data structure at the ADT level. The same applies to "methods" which are the implementation of the ADT operations.

In our example, the data structure consists of only one element: a signed sequence of digits. The corresponding attribute is an ordinary integer of a programming language. We only define two methods setValue() and addValue() representing the two operations set and add.

A class is the implementation of an abstract data type (ADT). It defines attributes and methods that implement the data structure and operations of the ADT, respectively. Instances of classes are called objects. Consequently, classes define properties and behavior of sets of objects. Here is another definition of a class:

```
class date
{
public:
    void changeDate(int month, int day, int year);
```

```
    void displayDate();
private:
    int month;
    int day;
    int year;
};
date payDay;
```

Objects

Consider the concept of employees working for a company. You can talk of instances of abstract employees. These instances are actual "examples" of an abstract employee, hence, they contain actual values to represent a particular employee. We call these instances objects.

Objects are uniquely identifiable by a name. Therefore you could have two distinguishable objects with the same set of values. This is similar to "traditional" programming languages where you could have, say two integers i and j both of which equal to "2". Please notice the use of "i" and "j" in the last sentence to name the two integers. We refer to the set of values at a particular time as the state of the object.

So by definition we can say that an object is an instance of a class. It can be uniquely identified by its name and it defines a state that is represented by the values of its attributes at a particular time. The state of the object changes according to the methods that are applied to it. We refer to this possible sequence of state changes as the behavior of the object. The behavior of an object is defined by the set of methods that can be applied on it.

We now have two main concepts of object-orientation introduced, class and object. Object-oriented programming is therefore the implementation of abstract data types or, in more simple words, the writing of classes. At runtime instances of these classes, the objects, achieve the goal of the program by changing their states. Consequently, you can think of your running program as a collection of objects. The question arises of how these objects interact? We therefore introduce the concept of a message in the next Module.

Messages

A running program is a pool of objects where objects are created, destroyed and interacting. This interacting is based on messages that are sent from one object to another asking the recipient to apply a method on itself. To give you an understanding of this communication, let's come back to the class Integer presented above. In our pseudo programming language we could create new objects and invoke methods on them. For example, we could use

```
Integer i;          /* Define a new integer object */
i.setValue(1);      /* Set its value to 1 */
```

to express the fact, that the integer object i should set its value to 1. This is the message "Apply method setValue with argument 1 on yourself." sent to object i. We notate the sending of a message with ".". This notation is also used in C++; other object-oriented languages might use other notations, for example "-".

Sending a message asking an object to apply a method is similar to a procedure *call* in "traditional" programming languages. However, in object-orientation there is a view of autonomous objects that communicate with each other by exchanging messages. Objects react when they receive messages by applying methods on themselves. They also may deny the execution of a method, for example if the calling object is not allowed to execute the requested method.

In our example, the message and the method which should be applied once the message is received have the same name: We send "setValue with argument 1" to object i which applies "setValue(1)".

By definition, we can say that a message is a request to an object to invoke one of its methods. A message therefore contains

the name of the method and
the arguments of the method.

Consequently, invocation of a method is just a reaction caused by receipt of a message. This is only possible, if the method is actually known to the object. A method is associated with a class. An object invokes a method as a reaction to receipt of a message.

Conclusion

How does this concepts help us developing software? To answer this question let's recall how we have developed software for procedural programming languages. The first step was to divide the problem into smaller manageable pieces. Typically these pieces were oriented to the procedures that were taken place to solve the problem, rather than the involved data. As an example consider your computer. Especially, how a character appears on the screen when you type a key. In a procedural environment you write down the several steps necessary to bring a character on the screen:

1. wait, until a key is pressed.
2. get key value

3. write key value at current cursor position
4. advance cursor position

You do not distinguish entities with well-defined properties and well-known behavior. In an object-oriented environment you would distinguish the interacting objects key and screen. Once a key receive a message that it should change its state to be pressed, its corresponding object sends a message to the screen object. This message requests the screen object to display the associated key value.

Lecture Notes: Part 3: Unified Modeling Language (UML)

Unified Modeling Language

A variety of modeling languages have evolved over time in support of traditional systems. We previously looked at some of the most common of these. In the OO world a single unified standard is winning out as the universal syntax for modeling. That syntax is defined by the Unified Modeling Language (UML).

UML is a modeling syntax, not a development method. It was originally defined by Booch, Jacobson, and Rumbauch and has been adopted as an OMG standard.

Stages in OO Development

OO development stages are often organized differently from the standard traditional stages: analysis, design, coding, testing. These stages are often given different names. Perhaps the most common are those recognized in the Rational Unified Process (RUP):

- Inception: Defining the business rationale for the project.
- Elaboration: Performing high-level analysis and design and creating products that drive construction.
- Construction: Building the software.
- Transition: Testing, along with training and implementation activities.

Types of UML Diagrams

The seven categories of UML diagrams are:

1. Use Case
2. Class Diagram
3. Interaction Diagram (including both the Sequence Diagram and the Collaboration Diagram)
4. State Diagram
5. Activity Diagram
6. Component Diagram
7. Deployment Diagram

Use Case Diagram

A Use Case diagram represents an interaction between an external entity (end user or other system) and the target software. The Use Case Diagram is the primary tool of the OO elaboration phase of OO development. Use Case diagrams do not correlate directly to classes, but lead us to the identification of classes and methods. They are the primary mechanism for capturing the users requirements.

In the context of a Use Case diagram, the Extend syntax represents a link to another Use Case that extends the functionality of the parent Use Case. The Use Case that invokes it only sometimes references a Use Case referenced by an Extend. Extend can be used to simplify complex Use Cases by extracting a part of the use case and setting it ups as a separate component.

In the context of a Use Case diagram, the Include syntax represents a link to another Use Case that is always included in the parent Use Case, but may also be included in other Use Cases. The Use Case that invokes it always references a Use Case referenced by an Include. Include can be used both to simplify complex Use Cases, and also to increase reusability. Reusability is increased because the Included Use Case can be used anywhere the same functionality is required.

Use Case Scenarios

A Use Case simply represents an interaction between an external entity and the target system. The processing represented by that Use Case is defined in a Use Case Scenario. A Use Case Scenario focuses on a single Use Case and a specific event associated with on occurrence of that Use Case. It lists the individual steps involved to accomplish the function represented. If the steps would be different under different conditions, a different Use Case Scenario would be developed for each potential set of conditions.

Use Case Scenarios can be contrasted to Process Specifications for traditional systems that are more generalized descriptions that address all possible

combination of conditions in one narrative. Scenarios address each combination separately.

Deployment Diagram

The Deployment Diagram shows the configuration of run-time processing nodes and the components that 'live' on these nodes. They address the Static Deployment view of the physical architecture. They are commonly used to depict how the hardware of the target system will be configured physically across locations.

Component Diagram

A Component Diagram shows the organization into, and dependency among, a set of components. The diagrams address the Static Implementation View of a system. The Component Diagram is commonly used to represent the organization of software components across hardware nodes.

Interaction Diagrams

The Interaction Diagram addresses the Dynamic View of the system. The diagrams represent Objects and relationships – including the messages that are passed among those objects. They show how objects communicate in the context of accomplishing some function. They normally focus on only one Use Case and one or more scenarios.

Sequence Diagram

The Sequence Diagram is an Interaction Diagram that focuses on time ordering of messages. The sequence diagram shows classes arranged in columns and depict messages passed between operations in the sequence necessary to carry out an activity. Sequence diagrams are the most common OO diagrams for depicting dynamic activities.

Collaboration Diagram

A Collaboration Diagram is an Interaction Diagram that focuses on the structural organization of objects exchanging messages. A Collaboration Diagram can be used to show the same interactions as a Sequence Diagram. However, instead of focusing on the time ordering of messages, it focuses on which objects interact with other objects - information that can be easily lost in the complexity of a Sequence Diagram.

State Diagram

Objects can behave differently in response to the same stimulus depending on the state of the object when it receives that stimulus. For example, pressing on the accelerator of an automobile has a quite different effect if the automobile is in drive than if it is in park. If the target software is sensitive to these states such that it is important to clearly understand them, a State Diagram may be drawn. A State Diagram describes all of the possible states that a particular object can experience and how the object's state changes as a result of events that reaches that object.

Activity Diagram

An Activity Diagram is a special type of State Diagram that shows flow from activity to activity. They are useful for describing workflow and behavior that involves parallel processing. Activity Diagrams are not always created, but are available when it is considered important to show this flow across activities. They are also very useful in modeling concurrent activities.

Module Four Team Activities:

- Work as a team to develop the Analysis Document

(This page left blank for your notes)

Module Five

Objectives:

- Object Oriented Design
- Coupling and Cohesion
- Design Specification Requirements

Assignments/Activities:

- Analysis Document Due

Lecture Notes: Part 1: Object Oriented Design

Software design consists of two components, modular design and packaging. Modular design is the decomposition of a program into modules. A module is a group of executable instructions with a single point of entry and a single point of exit. Packaging is the assembly of data, process, interface, and geography design specifications for each module. Ed Yourdon and Larry Constantine developed structured design**.** This technique deals with the size and complexity of a program by breaking up the program into a hierarchy of modules that result in a computer program that is easier to implement and maintain. Structured design requires that data flow diagrams (DFDs) first be drawn for the program. Processes appearing on the logical, elementary DFDs may represent modules on a structure chart. Logical DFDs need to be revised to show more detail in order to be used by programmers. The following revisions may be necessary:

- Processes appearing on the DFD should do one function.
- Processes need to be added to handle data access and maintenance.
- DFDs must be revised to include editing and error handling processes, and processes to implement internal controls.

Transform Analysis

One approach used to derive a program structure chart from program DFD is transform analysis. Transform analysis is an examination of the DFD to divide the processes into those that perform input and editing, those that do processing or data transformation (e.g., calculations), and those that do output. The portion consisting of processes that perform input and editing is called the afferent. The portion consisting of processes that do actual processing or transformations of data is called the central transform. The portion consisting of processes that do output is called the efferent. The strategy for identifying the afferent, central

transform, and efferent portions begins by first tracing the sequence of processing for each input. There may be several sequences of processing. A sequence of processing for a given input may actually split to follow different paths. Once sequence paths have been identified, each sequence path is examined to identify a process along that path that is a different process. The steps are as follows:

Step 1 - Beginning with the input data flow, the data flow is traced through the sequence until it reaches a process that does processing (transformation of data) or an output function.

Step 2 **-** Beginning with an output data flow from a path, the data flow is traced backwards through connected processes until a transformation processes is reached (or a data flow is encountered that first represents output).

Step 3 **-** All other processes are then considered to be part of the central transform!

Once the DFD has been partitioned, a structure chart can be created that communicates the modular design of the program.

Step 1 - Create a process that will serve as a "commander and chief" of all other modules. This module manages or coordinates the execution of the other program modules.

Step 2 - The last process encountered in a path that identifies afferent processes becomes a second-level module on the structure charts.

Step 3 - Beneath that module should be a module that corresponds to its preceding process on the DFD. This would continue until all afferent processes in the sequence path are included on the structure chart.

Step 4 - If there is only one transformation process, it should appear as a single module directly beneath the boss module. Otherwise, a coordinating module for the transformation processes should be created and located directly above the transformation process.

Step 5 - A module per transformation process on the DFD should be located directly beneath the controller module.

Step 6 - The last process encountered in a path that identifies efferent processes becomes a second-level module on the structure chart.

Step 7 - Beneath the module (in step 6) should be a module that corresponds to the succeeding process appearing on the sequence path. Likewise any process immediately following that process would appear as a module beneath it on the structure chart.

Transaction Analysis

An alternative structured design strategy for developing structure charts is called transaction analysis. Transaction analysis is the examination of the DFD to identify processes that represent transaction centers. A transaction center is a process that does not do actual transformation upon the incoming data (data flow); rather, it serves to route the data to two or more processes. You can think of a transaction center as a traffic cop that directs traffic flow. Such processes are usually easy to recognize on a DFD, because they usually appear as a process containing a single incoming data flow to two or more other processes. The primary difference between transaction analysis and transform analysis is that transaction analysis recognizes that modules can be organized around the transaction center rather than a transform center.

Packaging Program Specification

As a systems analyst, you are responsible for packaging that set of design documentation into a format suitable for the programmer. This package is called a technical design statement. The technical design statement should include all data, process, interface, and geography building block specifications developed by the designer.

Object-Oriented Design

Design **Objects**

The objects that represented actual data within the business domain in which the user was interested in storing were called Entity Objects. Objects that represent a means through which the user will interface with the system are called Interface Objects. Objects that hold application or business rule logic are called Control Objects. A system's functionality is distributed across all three types of objects. This practice makes the maintenance, enhancement, and abstraction of objects simpler and easier to manage. The three object types correlate well with the client-server model. The client is responsible for the application logic (control

objects) and presentation method (interface objects). The server is responsible for the repository (entity objects).

Entity Objects

Entity objects usually correspond to items in real life and contain information known as attributes that describe the different instances of the entity. They also encapsulate those behaviors that maintain its information or attributes. An entity object is said to be persistent meaning the object typically outlives the execution of a use case (or program). An entity object exists between use case executions because the information about that entity object is typically stored in a database (allowing for later retrieval and manipulation).

Interface Objects

It is through interface objects that the users communicate with the system. The responsibility of the interface object is two fold:

- It translates the user's input into information that the system can understand and use to process the business event.
- It takes data pertaining to a business event and translates the data for appropriate presentation to the user.

Each actor or user needs its own interface object to communicate with the system. In some cases the user may need multiple interface objects.

Control Objects

Control objects contain behavior that does not naturally reside in either the interface or entity objects. Such behavior is related to the management of the interactions of objects to support the functionality of the use case. Controller objects serve as the "traffic cop" containing the application logic or business rules of the event for managing or directing the interaction between the objects. Controller objects allow the scenario to be more robust and simplifies the task of maintaining that process once it is implemented. As a general rule of thumb, within a use case, a control object should be associated with one and only one actor.

Object Responsibilities

In object-oriented design it is important to recognize an object has responsibility. An object responsibility is the obligation that an object has to provide a service when requested and thus collaborating with other objects to satisfy the request if required. Object responsibility is closely related to the concept of objects being able to send and/or respond to messages.

Object Reusability

Many companies achieve their highest level of reuse by exploiting object frameworks. An object framework is a set of related, interacting objects that provide a well-defined set of services for a accomplishing a task. By using object frameworks, developers can concentrate on developing the logic that is new or unique to the application, thus reducing the overall time required to build the entire system.

Object oriented design includes the following activities

- Refining the use case model to reflect the implementation environment
- Modeling object interactions and behavior that support the use case scenario

---

Lecture Notes: Part 2: Coupling and Cohesion

Coupling and Cohesion are the two primary measure of the quality of design for both OO and traditional systems. Coupling is a measure of the interdependence between modules - how much data and control needs to be passed to enable our overall design to function. Coupling should be minimized. Cohesion is a measure of the strength of association of the components within a module - did we package code that belongs together. Cohesion should be maximized. In traditional design there are six levels of coupling:

1. Content
2. Common
3. External
4. Control
5. Stamp
6. Data

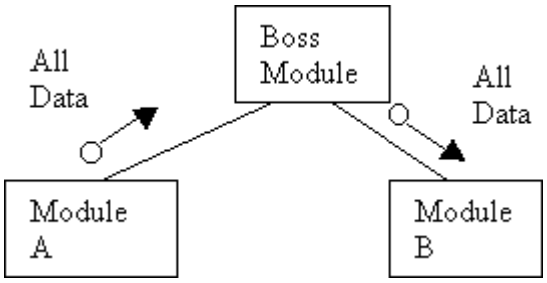
Content Coupling

In Content coupling one module directly references the content of another. This cannot be accomplished in many programming languages. However, it can be done in machine languages and assembly languages - languages that are tied closely to the internal structure of the machine. Content coupling was also possible in the early days of Cobol using the ‘alter’ verb. When Content coupling is used it is generally where the efficiency of the code is of critical importance. In content coupling a module can actually change a code statement in another

module. Imagine the maintenance problems trying to debug a problem in a program when you are not even certain what the code looked like at the time it executed. Content coupling should always be avoided. There is not even a convention for modeling Content coupling using a structure chart.

Common Coupling

Common coupling is where two modules have access to the same global data. Again, it is not generally found with most modern programming languages. Normally you can define data as being either local or global. When data is defined as local only the local modules can modify it. This makes maintenance easier because when we have a bug we can identify one or a few modules that could have modified the data and created the bug. With Common coupling it is as though all of the data was defined as global. Every parameter is passed from module to module. If a bug occurs we have no way of knowing where it was introduced because the data could have been modified anywhere within the design. If we were going to model common coupling it would look like the following:

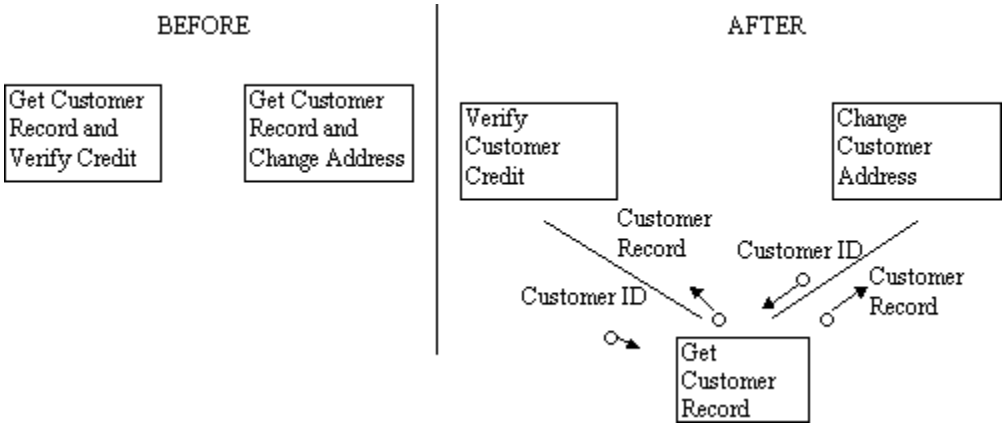


Common coupling is common in COBOL programs. In COBOL each program contains a Data Division and a Procedure Division. All data in the Data Division is available to all of the modules in the Procedure Division - which leads to common coupling. This is one of the reasons that COBOL programs are so difficult to maintain. Regardless of what language you are using Common coupling should always be avoided. Even where the data structures support common coupling, such as in COBOL, one should take care within the code itself not to reference data that is not needed by a module.

External Coupling

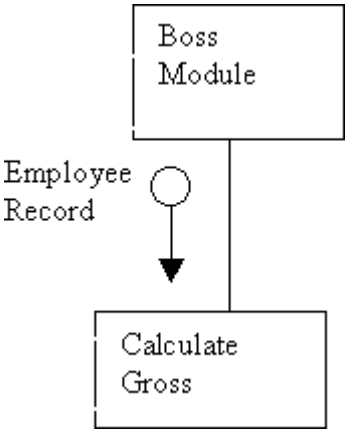
External coupling describes a module's communication with the outside environment. Read and Write statements to physical files, as well as displaying or receiving data from the computer screen, are examples of external couples. Obviously, it is essential that any program be able to read and write data - if we can't bring data in or send it out the module can serve no useful purpose. Therefore, we must have external coupling. Our goal should be to manage and minimize such coupling.

What it means to minimize external coupling is that we should use a single module for the actual reading and writing of data. If multiple modules are reading from the same file, factor out the actual read statement from any other functionality and create a reusable module that does the external coupling and only the external coupling - nothing else. The modules under 'Before' in the following examples contain external coupling along with other functionality, such as editing and updating. The change to factor out the external coupling into its own module is shown under 'After'



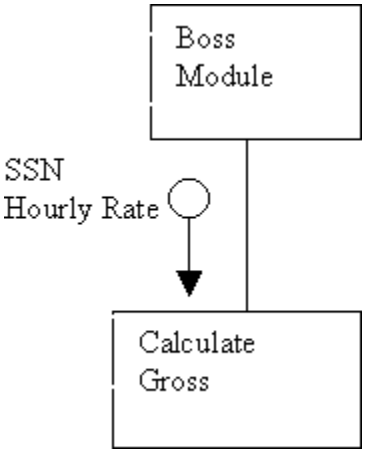
Stamp Coupling

Stamp couples are a much less dramatic form of common coupling. Stamp coupling is the passing of a data structure where not all elements in the structure are used. The following example shows the passing of the entire employee record from the boss module to Calculate Gross.



The Payroll Record contains SSN, Name, Address, Org Code, Date of Birth, and Hourly Rate. The Calculate Gross only needs the SSN and Hourly Rate to do its job. The passing of the extra elements makes this a stamp couple. Stamp couples are very common and are not generally a problem. However, the potential exists that the extra data could lead to additional maintenance if an

unused element was changed, leading to a change in a module that does not even use that data. Stamp couples can be eliminated simple by changing the design to pass only what is needed.



This reduction in coupling makes a module that is easier to maintain and more likely to be reused.

Data Coupling

Data coupling is the least intrusive form of coupling. In Data Coupling every argument is either a simple argument or a data structure in which all elements are used by the called module. The example above that factored required data out of the stamp couple represents data coupling where only the data needed by Calculate Gross is passed.

Cohesion

Just as there are levels of Coupling, there are levels of Cohesion, in this case, seven.

1. Functional
2. Informational
3. Communicational
4. Procedural
5. Temporal
6. Logical
7. Coincidental

Functional Cohesion

The ideal form of cohesion is Functional Cohesion. In Functional Cohesion each of the components within a module belong together and contribute towards the

performance of a single function. They perform one action or achieve a single goal. You can generally get a hint about the cohesion of a module from its name.

> Example: Calculate Gross Pay.
>
> The simple verb/object name is an indication that it is probably a functionally cohesive module. It is performing one simple function (calculate) against one data structure (Gross Pay).

Functionally Cohesive modules are more likely to be reusable by other software. Maintenance and testing are simpler because the module is not altered unless that specific function needs to be changed.

Informational (or Sequential) Cohesion

Informational cohesion exists when output from one element in the module serves as input for some other element. All elements are part of a series of actions and all of the actions contribute to the same general function, however, the complete function is not accomplished.

> Example: Calculate Gross Pay and Calculate State Tax.
>
> Both of these functions contribute to a single, more complete process (calculate payroll) and are sequential steps in that process.

Elements in a module with this type of cohesion are organized with function in mind and provide a better isolation of functionality than in communicational cohesion (the next level on our list). Reuse is somewhat limited, because the module is only usable in an identical environment. For example, if there is a situation where there is a need to Calculate Gross Pay, but not Calculate Sales Tax, we would not be able to use this module. Informational cohesion is considered an acceptable form of cohesion, but can be improved simply by breaking the module into two components - in our example one to calculate gross pay and a second to calculate State Tax.

Communicational Cohesion

In communicational cohesion the module performs more than one function, and these functions are all performed on the same body of data. All of the data is passed among the elements of the module.

> Example: Update the record in database and record the record in the audit trail. These components use the same data but perform two different functions.

Communicational cohesion is better than the procedural cohesion (which follows) because the module's focus is organized around the problem being solved rather than process. Such modules tend to be more reusable than procedural modules, but less reusable than informational and functional modules. Because the module is performing more than one function it is more difficult to maintain. If any of the functions within the module are changed the other functions are subjected to potential errors. All functions must be retested to guard against inadvertent error. The potential for reuse is somewhat limited as well.

Procedural Cohesion

In procedural cohesion a module performs more than one function, and these functions are only related to a general procedure affected by the software.

> Example: Plot graph R2 on plotter and print report XYZ. Where the information on graph R2 and report XYZ are not related.

Although the relationship among the functions is not strong, there is at least some relationship based on the type of procedure that is performed. However, this re relationship is based on the program (design) procedure rather than the problem (requirement) procedure. Reusability is adversely impacted whenever we focus on procedure as a basis for organizing modules.

Temporal Cohesion

In temporal cohesion a Module performs more than one function, and the functions are related by the fact that they must occur within the same time span.

> Example: Create year-end tax statements and post Cost of Living increase for next year. These two modules are related because they both occur at the end of the calendar year. The designer may decide to put them together in the same module (or program) based on this relationship. This is a weak relationship that decreases the potential for reuse and increases the cost of maintenance and testing.

Modules performing program initialization, termination, house keeping and clean up are common candidates for temporal cohesion. Temporal cohesion is considered better than logical cohesion because all of the elements are executed at each invocation of the module. However, this is still not considered a good basis for combining modules.

Logical Cohesion

Under Logical Cohesion a module performs more than one function and these are only related 'logically' based on some perceived commonality of process.

> Example: A module that adds New Employees, New Customers, and New Inventory Items.

Such a module is performing the same general type of process to different records. This is a very weak relationship, based solely on the type of process performed, even though the specific functional problems being solved are totally unrelated. There are very few advantages to this type of module, although it is considered better than simple coincidental cohesion. Changes to this type of module may cause a ripple effect necessitating the change to components of the module that are not using the new functionality. Because of the lack of strong relationship among the functions within the module, these ripple effects may be common and considerable.

Coincidental Cohesion

In coincidental Cohesion functions are grouped within a module pretty much at random. There is thought regarding how or why they are combined into the same module. Such cohesion reflects a total lack of consideration for the design of the software.

> Example: Update New Customer, Pay Employees, and Calculate Bowling Scores.

Clearly in this case changes to one function could have a ripple effect on other completely unrelated functions. Such inadvertent impacts would be totally unpredictable. Data and Control Coupling would be increased and reusability would be nonexistent.

Lecture Notes: Part 3: Design Specification Requirements

The following is an example of a Design Specification Document. The third deliverable for the team project is a document similar to this one but designed for your team's software development project. You may use this document as a template, use another template of our choice, or design your own format. The idea is to include the type of information that is contained in the following example. Keep in mind that the design specification document is the blueprint for the programming stage of development. This document is handed to the system programmers and is used as a guide to write the source code.

Software Design Specification

1.0 Introduction

This Module provides an overview of the entire design document. This document describes all data, architectural, interface and component-level design for the software.

1.1 Goals and objectives

Overall goals and software objectives are described.

1.2 Statement of scope

A description of the software is presented. Major inputs, processing functionality and outputs are described without regard to implementation detail.

1.3 Software context

The software is placed in a business or product line context. Strategic issues relevant to context are discussed. The intent is for the reader to understand the “big picture.”

1.4 Major constraints

Any business or product line constraints that will impact the manner in which the software is to be specified, designed, implemented or tested are noted here.

2.0 Data design

A description of all data structures including internal, global, and temporary data structures.

2.1 Internal software data structure

Data structures that are passed among components the software are described.

2.2 Global data structure

Data structured that are available to major portions of the architecture are described.

2.3 Temporary data structure

Files created for interim use are described.

2.4 Database description

Database(s) created as part of the application is (are) described.

3.0 Architectural and component-level design

A description of the program architecture is presented.

3.1 Program Structure

A detailed description of the program structure chosen for the application is presented.

3.1.1 Architecture diagram

A pictorial representation of the architecture is presented.

3.1.2 Alternatives

A discussion of other architectural styles considered is presented. Reasons for the selection of the style presented in Module 3.1.1 are provided.

3.2 Description for Component n

A detailed description of each software component contained within the architecture is presented. Module 3.2 is repeated for each of n components.

3.2.1 Processing narrative for component n

A processing narrative for component n is presented.

3.2.2 Component n interface description.

A detailed description of the input and output interfaces for the component is presented.

3.2.3 Component n processing detail

A detailed algorithmic description for each component is presented. Module 3.2.3 is repeated for each of n components.

3.2.3.1 Interface description

3.2.3.2 Algorithmic model

3.2.3.3 Restrictions/limitations

3.2.3.4 Local data structures

3.2.3.5 Performance issues

3.2.3.6 Design constraints

3.3 Software Interface Description

The software's interface(s) to the outside world are described.

3.3.1 External machine interfaces

Interfaces to other machines (computers or devices) are described.

3.3.2 External system interfaces

Interfaces to other systems, products, or networks are described.

3.3.3 Human interface

An overview of any human interfaces to be designed for the software is presented. See Module 4.0 for additional detail.

4.0 User interface design

A description of the user interface design of the software is presented.

4.1 Description of the user interface

A detailed description of user interface including screen images (if available) is presented.

4.1.1 Screen images

Representation of the interface from the user's point of view.

4.1.2 Objects and actions

All screen objects and actions are identified.

4.2 Interface design rules

Conventions and standards used for designing/implementing the user interface are stated.

4.3 Components available

GUI components available for implementation are noted.

4.4 Development system description

The user interface development system is described.

5.0 Restrictions, limitations, and constraints

Special design issues that impact the design or implementation of the software are noted here.

6.0 Testing Issues

Test strategy and preliminary test case specification are presented in this Module.

6.1 Classes of tests

The types of tests to be conducted are specified, including as much detail as is possible at this stage. Emphasis here is on black box and white box testing.

6.2 Expected software response

The expected results from testing are specified.

6.3 Performance bounds

Special performance requirements are specified.

6.4 Identification of critical components

Those components that are critical and demand particular attention during testing are identified.

7.0 Appendices

Presents information that supplements the design specification.

7.1 Requirements Traceability matrix

A matrix that traces stated components and data structures to software requirements is developed.

7.2 Packaging and installation issues

Special considerations for software packaging and installation are presented.

7.3 Supplementary information (as required)

Module Five Team Activities:

- Work as a team to develop the Design Document

Module Six

Objectives:

- Midterm Exam Prep and Review
- Risk Management

Assignments/Activities:

- Midterm Exam

Lecture Notes: Risk Management

Overview of Risk Management

If everything on a project remained the same from the preliminary analysis and estimates, then every project would be successful. It would deliver the expected product, on time, and within budget. Reality, though, is that nothing remains the same. Change is certain. These changes are risks to project success.

It is the Project Managers' job to identify these changes or risks at the very beginning of the project. What is likely to change? What will the impact to the project be? This proactive approach is called Risk Management.

Typically though, people operate in a reactive mode. When changes begin to occur, the team is surprised and caught off guard. This typical project behavior is referred to in the industry as "fire fighting." The changes occur, fires breakout, and even routine project tasks fall behind schedule while team members try to "put out the fires."

Risk Management Process

The proactive approach to reacting to project changes - Risk Management involves the following steps:

- Risk Identification - assess probability and impact
- Communicate Risks - to the project team, management team, and customers. Set expectations.

- Monitor Risks - throughout the project life cycle at the weekly team meeting. Sometimes these risks give off early warning signals that they are going to occur before they actually do
- Risk Mitigation - some identified risks can be prevented before they occur. Perhaps we have identified a key team member that may be likely to be reassigned before the project is finished. Putting a partner with this key member who can step into his role if he's reassigned would be a way to mitigate the risk.
- Risk Contingency Planning - if you can't mitigate a risk or prevent it from occurring, then you must put together a plan for how the team is to deal with the change when it does occur. Having a plan prepared ahead of time will prevent fire fighting.

Larger projects inherently have more risk associated with them. More team members lead to a higher likelihood of communication failures. Longer projects increase the likelihood that changes will be required. Consequently, Risk Management is a MUST for larger projects. Project Managers prepare a Risk Management plan referred to as a Risk Management, Mitigation, and Monitoring (RMMM) plan that formally identifies the risks and outlines contingency plans.

Risk Management is important for smaller projects too. However, the process is less formal. The Project Manager goes through all the same steps as outlined above, but they are more abbreviated and typically are not formally documented.

Predictable Risks

Three predictable risks on any software development project are:

- Poor communication - identify where the weak communication links exist.
- Staff turnover - someone will be reassigned, leave the company, or have some personal emergency.
- Requirement changes - on every project!

As stated earlier, larger projects bear more of these risks than smaller projects. However, smaller projects also have these three risks. The Project Manager must identify and assess the likely communication problems, the stability of team members, and which requirements are likely to change. Mitigation or contingency plans must be developed and these risk areas must be monitored throughout the project.

Identifying Risks

The Project Manager begins to identify risks by looking at the following situations.

- What is the product size? Bigger product means bigger risks.

- What is the business impact? A software product impacting revenue, marketing or sales ability of the organization will bear more risk than a software product providing reporting or monitoring abilities.
- What are the characteristics of the customer? Some customers are technically savvy or just plain picky. They will likely present more changes during product development than other customers.
- What is the development environment? If the development team does not have the proper tools to construct the product, the risks will be higher and there will be increased productivity loss.
- What technology is being delivered? Building for newer, cutting edge technology that the team is not familiar with will increase the risk.
- Staff size and experience? The larger the staff the larger the risk. As the experience levels decreases, the risk level increases.

Strategic Risk

- Are the project objectives aligned with the overall business objectives?
- To what extent is the business structure likely to change/reorganize during the project lifecycle?
- To what extent is senior management committed to the project's outcome? If not fully committed, the team may not get the proper support when they need it the most.

Financial Risk

- How secure is the project funding?
- Are the cost/benefits clearly defined?

Project Management Risk

- Is the PM experienced? Is there a mentor assigned?
- Has the process methodology been defined? Or are the project team members going to develop the product any way they can?
- If the process methodology has been defined, are the team members comfortable delivering in the manner expected? Do they understand the selected methodology?
- Is the software development team located in the same building? City? Country? A widely dispersed development team bears more risk of communication failure.

Technology Risk

- How thoroughly have the technology options been evaluated?
- Is there an experienced architect on the team?
- How familiar is the team with the selected technology?
- What's the complexity of the engineering and integration?

Risk Prioritization

Once the Project Manager has identified all the possible risks, assessed their likelihood of occurring, and assessed the impact to the project if they do occur, then the risks should be prioritized. Those that are most likely to occur and have the highest impact to the project should be of highest priority. The top priority risks (perhaps 10) will either be mitigated or contingency plans prepared. These top 10 risks are the ones the PM will monitor most closely for the duration of the project. It is not practical to develop contingency plans and mitigation strategies for every risk to a project, but all risks should be monitored occasionally.

The Pareto rule (sometimes called the 80-20 rule) can be applied here. This says that 80% of the causes of project failure can be traced to 20% of the risk factors. Hopefully, our list represents that 20%.

The RMMM plan outlines and reports these findings. Though this report is largely in textual format, the Air Force has developed a visual risk grid (Pressman, page 150) that shows the risks along the Y-axis and the project impact for each risk along the X-axis. This is a good supplement to be included in the RMMM report and for the PM to use as a guide for monitoring.

Risk Management Example

The following example addresses more specifically the steps that should be followed to create a Risk Management plan. Let's look at the steps involved:

Step 1: Identify risks

Brainstorming sessions involving all of the personnel on the project are often conducted to identify candidate risks. We can also look at past history to see what risks have been considered in the past and what problems have occurred. For this activity, let's consider the idea of going for a day in the Sonora Desert outside Tucson. Before starting out it might be a good idea to consider what risks we are taking and what we could do about them. What risks do you think would be involved?

One risk could be that 'We might run out of water.' However, while this simple statement identifies an area of risk it is not stated in such a way as to make the actual risk clear. It doesn't pass the 'So what?" test. That is, what will happen if we run out of water? A better statement would be 'We might run out of water and suffer extreme dehydration.' This statement identifies both what happens and what the consequence might be. One of the most common mistakes made in risk assessments is failing to define the risk statement adequately.

A complete risk statement that passes the ‘So What?’ test is easier to assess in terms of its probability and impact. It also makes it possible to develop focused mitigation strategies and contingency plans.

Risk identification is not complete until you have carefully considered all of the possible risks to your project.

Step 2: Assess the probability and impact of each risk

While it might be nice to formally manage all of our risks, this takes time and money. So, our next step is to determine which risks are significant enough that they need to be formally managed. In this step we will assign a number from 1 to 9 to each risk to indicate how likely it is to happen and how significant the impact would be if it does happen. The higher number indicates a higher probability and greater impact. Then you multiply the two numbers to calculate a risk index and identify those risks that will be formally managed.

How likely is it that we will run out of water? It depends on how long the hike is, how hot it is, and how much water we carry. Making an accurate judgment about this likelihood is easier if we have prior experience hiking in the desert. Experience is a critical factor in all risk assessment. You should always involve people who are experienced in the process to help you assess probability and impact.

For our purposes we are going to say that it is moderately likely that we will run out of water. Therefore we will give this a 5. What is the impact? Again, experience is helpful, but I think we would all agree that the impact of this is pretty high. Let’s give it a 9.

We can now calculate our risk index as 45 (5*9).

By calculating the risk index for each risk we can identify those risks that need to be managed further. All risks above a specified risk index value (we will use 28, a common value in practice) are formally managed.

On real world projects it is common to have a variety of people provide estimates of probability and impact. If they are all pretty close to the same value, we can average out the estimates and use the result to calculate our risk index. However, if the numbers diverge (some people rating the likelihood a 1 and others a 9) it is an indication that our risk is not clearly defined. When this happens you should revisit your risk definition and try again.

Step 3: Develop Mitigation Strategies and Contingency plans for each risk above the specified risk index value.

Each risk that exceeds our specified risk index value (over 28 in our case) will be formally managed. This means that we will develop a mitigation strategy and contingency plan. The mitigation strategy addresses how we will reduce the probability that the risk will actually occur. What could be done to reduce the probability of running out of water? We could carry more water than we think we will need. We could also research the locations of fresh water supplies near our trail.

Which strategies we consider and which we finally choose to employ is a judgment call. However, whatever we come up with for our mitigation strategy should be something we plan to implement. Contingency plans lead to additional tasks on our work breakdown structures and have impacts on our cost and schedule.

The contingency plan addresses what we do if, despite our attempts to mitigate the risk, it still occurs. What happens if we start to run out of water? We could begin rationing immediately. We could also head for the nearest fresh water supply. Finally, we could use our cell phone to call for help.

But wait a minute, we might not have a cell phone with us - at least not if we didn't consider that as one of our mitigation strategies. This goes to the very purpose of risk assessments - thinking about what might go wrong so we can reduce the probability that it will happen, and be ready to react if it does. Let's add carry a cell phone to our mitigation strategies.

Step 4: Document the Risk Assessment.

Once you have captured all of this information you need to document it in a readable format. The following is a sample Risk Assessment for our desert hike.

There are have been three risks assessed:

1. We might be bitten by a snake and die
2. We might develop sore feet and not be able to complete the hike.
3. We might run out of water and suffer extreme dehydration.

Risk	*Probability*	*Impact*	*Wtd Risk*	*Mitigation*	*Contingency*
1.	1	9	9	Stay on the path	Use a snakebite kit
2.	5	6	30	Practice hikes	Shorten the trip
3.	6	9	54	Cary more water	Ration Water

Note: Normally, you would have more than one Mitigation and Contingency for each risk. This is a simplified example.

Step 5: Risk Monitoring

If a risk does not exceed our threshold risk index, we do not develop mitigation strategies and contingency plans. However, we do continue to monitor those risks, along with our other risks. By reassessing the probability and impact of risks as the project progresses we ensure that we aren't caught by surprise and become a 'reactive' risk manager.

Module Six Team Activities:

- Work as a team to develop the Design Document

(This page left blank for your notes)

Module Seven

Objectives:

- Java Programming
- GUI and Prototype Design

Assignments/Activities:

- No deliverables
- Java Programming Lab

Lecture Notes: There will be no new Software Engineering concepts introduced in this Module. Java programming basics will be review in class. You will also have an opportunity to work on your team project prototypes and trouble shoot any problems that you might be having with them, etc.

Module Seven Team Activities:

- Work as a team to develop the Design Document
- Work as a team to develop a Working Prototype

(This page left blank for your notes)

Module Eight

Objectives:

- Software Metrics
- Quality Assurance
- Configuration Management

Assignments/Activities:

- No deliverables
- Java Programming Lab

Lecture Notes: Part 1: Software Metrics

Early Failures of Metric Programs

Metric programs sprang up everywhere during the 1980s with the popularity of the Total Quality Management (TQM) movement. These metrics programs promised if software development activities could measure variations, properly interpret those results, and adjust processes accordingly, the resulting software product quality would significantly increase. Everyone was interested in these kinds of results and invested heavily in sending software professionals to metric training classes and purchasing metric collection tools. However, the management teams driving these programs were not committed to the long-term investments necessary in the collection and analysis of data over periods of time nor did they have the proper training or experience to interpret the resulting data to take the appropriate corrective actions. Consequently, most metric programs during the 1980s resulted in large investments wasted on failed metric programs.

Metrics and the CMM

The Software Engineering Institute's Capability Maturity Model discussed in Module two also stresses the importance of metrics in an overall continuous process improvement program.

- Level 1 - Initial - ad hoc, chaotic.
- Level 2 - Repeatable - initializing basic Project Management, Configuration Management, Quality Assurance, and Requirements Management practices.

- Level 3 - Defined - refining these practices and implementing them across the entire organization.
- Level 4 - Managed - define and collect metrics.
- Level 5 - Optimizing - interpreting metrics to refine processes for continuous process improvement.

There has been debate in the industry over the cost/benefit of moving past CMM Level 3. There is no argument that implementing basic PM, CM, QA, and Requirements Management practices yield significant quality benefits. However, there is speculation that achieving CMM Level 3 produces somewhere near an 80% product quality improvement. The additional investment required moving to Level 4 and Level 5 may only yield an additional 20% product quality improvement. There is some doubt whether the 20% quality improvement is worth the investment. Establishing workable, sustainable metric programs that yield continuous process improvements is VERY expensive.

Process Metrics

Process metrics are strategic. These measurements tell the management team whether they are meeting their overall business goals. To determine these objectives metrics are collected across several projects. Investments and commitments must be made for long-term metrics collection to amass enough historical data over time to identify and analyze trends. Those organizations that are mature enough to make these commitments are reaping the benefits.

Project Metrics

Project metrics are tactical. These measurements tell the Project Manager and management team whether the project they are managing is on track. Is it producing deliverables on time and within budget? These metrics have immediate as well as long-term benefits. Most organizations gather project metrics during project execution for this very reason. However, few have methods of collecting this data over long periods to further analyze trends to be beneficial to overall process improvement.

Lines Of Code Metrics

The most common metrics used for size estimation is the Line of Code (LOC). A LOC is simply that. One line of code is counted as one; the next LOC is counted as two, and so forth. However, controversy on what constitutes an LOC abounds. Should a comment line in a computer program be counted as a LOC? Some argue yes. It takes effort (therefore cost) to produce it and maintain it. The comment line represents an asset just as a line of executable code does. Others argue no. Whatever method is used to count LOC is not as important as consistency. Determine what to count and consistently count it for the entire inventory.

LOC metrics does us some relative size factor when determining effort required to maintain it or in estimating future projects of similar size. However, LOC is language specific. 1000 lines of COBOL code will take much less effort to maintain that 1000 lines of C code. Further, by assigning value to software based on its size in LOC, we are effectively rewarding inefficient software development. Also, accurate LOC counts are not available until late in the development life cycle, long after key decision based on size need to be made.

Function Points

The Function Point (FP) is also considered a sizing metric. It yields language independent data, and provides information on the relative complexity of the program being evaluated. The program is evaluated using scales that determine relative complexity values for such processing events as: I/O required, inquiries, numbers of external interfaces, use of graphical user interfaces, and other criteria. The program is assessed, the values are applied, and a resulting FP for the program generated. Consequently, in our example above - 1000 lines of COBOL code may yield a FP value of 10, whereas the 1000 lines of C code may yield a FP value of 100. This data is more relevant in predicting maintenance levels and estimating future projects.

FP estimates are more easily estimated during analysis and design phases. Software engineers can estimate the relative FP of the resulting product early in the lifecycle using analysis and design models. This data is immensely useful in estimating further development efforts.

There are automated tools that will scan source code inventories and count LOC. However, the tools available for scanning source code to produce FP have not become very popular. The process for deriving FP by manually scanning source code has not been an investment that many developers have been willing to make. For these reasons, LOC continues to be the more widely used size metric for legacy inventories.

Roger Pressman provides examples of algorithms used in determining FP calculations in the recommended textbook. There is interesting data in this textbook that shows how many LOC per language it would take to equal one Function Point. It would take an estimated 320 lines of Assembly Language Code to equal one FP. While, it would take only 32 lines of Visual Basic code to equal one FP. Other languages are evaluated in his example.

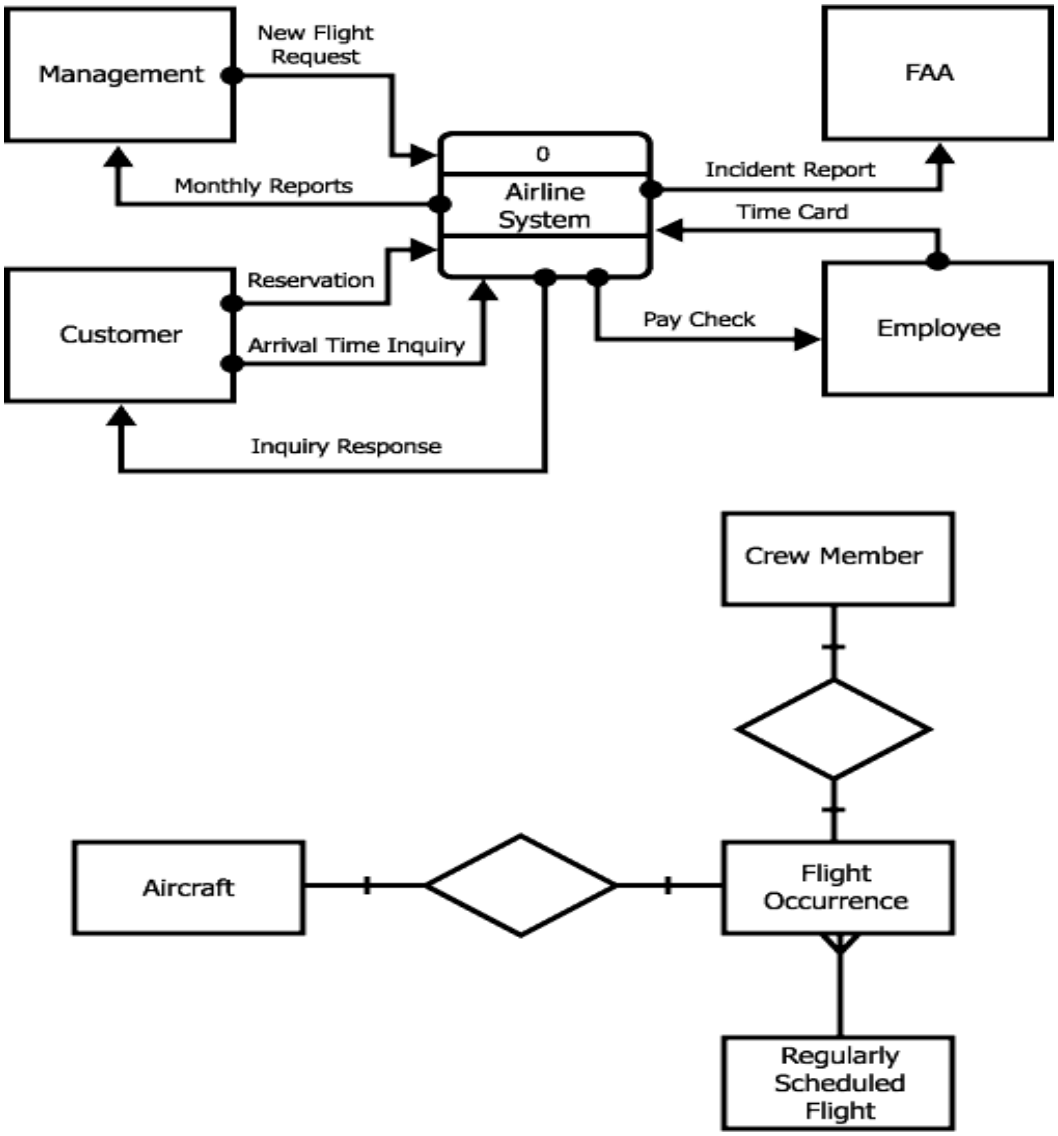
Function Point Example

Function Point counts are driven primarily by the number of Inputs, Outputs, Inquiries, External Interfaces, and Files for the target system. These counts are assigned a weighting factor and the final count is adjusted for complexity. The

process for calculating function points is discussed on pages 89-91 of Pressman. Here we will walk through this process step by step to see just how it works.

In this activity we will calculate the function points for a sample system. To do this we will reference a context level Data Flow Diagram and an Entity Relationship Diagram for that system.

Below are the context level DFD and ERD for an imaginary airline company. They have, of course, been simplified for use in this exercise.



Any arrow pointing towards the process bubble in the center of the DFD is an input. Any arrow flowing away from the process is an output. Pretty simple. However, for purposes of counting Function Points some of these inputs and

outputs are counted as inquiries and external interfaces. These must not be double counted.

Inquiries

The ‘Arrival Time Inquiry’ from the Customer is simply a request to retrieve the time that a flight will arrive from the system. This inquiry and the associated ‘Inquiry Response’ arrow are counted as one inquiry for Function Point purposes. So that we don’t double count them we will ignore these same two arrows later when we count inputs and outputs. Inquiries are counted separately because they are generally simpler to implement than other inputs and outputs and the Function Point counting process has been structured to reflect this.

External Interfaces

External Interfaces are more difficult to implement than other inputs and outputs so they need to be counted separately as well. ‘Incident Report’ going to the FAA is an external interface because the data is flowing between our system and another automated system, rather than to a human being. The airline system must send electronic transactions to the FAA, something much more difficult to implement than using a screen interface.

In contrast, the Reservation uses a GUI interface screen, which we control. While it is true that this is an interface with something ‘external’ to our system, it is not what we mean by the term ‘external interface’ when counting function points.

Again, you would only be able to make these distinctions if you had more detailed knowledge of the actual requirements. When Function Points are counted for real world systems the counter must be familiar enough with the requirements to make this sort of distinction. When you do the function point count for your team project you can make your own assumptions based on your knowledge of that requirement. In any case, we will count Incident Report as an external interface and not as an output. Incident Report is the only external interface on our diagram.

Inputs and Outputs

The remaining arrows (including the aforementioned ‘reservation’) are all either inputs (if flowing into the process bubble) or outputs (if flowing out). This leaves us with a count of 3 inputs (Time Card, New Flight Request and Reservation), 2 outputs (Pay Check and Monthly Report), one Inquiry and one External Interface. This count is one less than the number of arrows on the diagram because the two arrows associated with the inquiry count as only one.

Files

All we have left to count are the files. For our purposes we will simply use the number of entities on our ERD. As you can see, there are 4. While the counts above will work for this course, the actual process is much more complex. In the real world trained certified Function Point counters are often called upon to perform the counting while working with experts in the functionality of the process.

Applying Weighting Factors to Compute Base Function Point Count

Our next step is to plug these counts into the table on the top of page 90 in Pressman. To use this table we must decide if each of our counts is simple, average, or complex. Let's assume that everything is average except for the Pay Check, which we will treat as complex, and the Aircraft file, which we will treat as simple. Do the math and determine your own count before proceeding.

We came up with a count of 73 as follows:

Inputs:	(2*5) + (1*7) = 17
Outputs:	2*4 = 8
Inquiries:	1*4 = 4
Files:	(1*7) + (3*10) = 37
External Interfaces:	1*7 = 7

Total = 8 + 17 + 4 + 37 + 7 = 72

This is our base function point count.

Applying Complexity Adjustment Factors

The final step in our process is to adjust this count based on the overall complexity of our target system. If we are implementing the system in a very simple environment we should adjust the count down. If the environment is particularly complex, we should adjust upwards. To make this adjustment we must answer the 14 questions listed at the top of page 91. Again, knowledge of the target application is essential to do this properly.

Let's look at a couple of these questions. Number 1 asks; 'Does the system require reliable backup and recovery?' Well of course it does. Every system does. But we need to answer with a numeric value from 0 to 5, where 0 is not important and 5 is absolutely essential. If this were a banking application we would probably give it a 5 - we certainly don't want to lose records of our customer withdrawals. If the system simply calculated handicaps for our local bowling league we might rate this a 1 or 2. What would it be for an airline system?

I would think at least a 4 or 5. Let's go with 4.

Continue this process for each of the 14 questions. Make whatever assumptions you feel are appropriate for an airline system. For our purposes, let's assume that the rest of our answers are all 4s. If we add the values of all of the 14 answers it would give us a sum of 56 (4*14). Therefore, our complexity adjustment value is 56.

Final Function Point Count

To complete our calculation we plug the complexity adjustment value (S (Fi)) and the base function point count (count total) into the formula:

FP = count total * [0.65 + 0.01 * S (Fi)].

Give this formula a try before going forward.

We should have arrived at a final FP of 81.03.

This number identifies the size of the project in Function Points. It provides a means of comparing various projects and of making size and budget estimates based on past history. While it is not an exact process, it has proven to be more meaningful than any other common method for estimating project size.

Quality Metrics

Quality - though some may argue that quality is in the eye of the customer, quality measurements are usually gathered in terms of numbers of errors produced during the software development process or numbers of defects reported by the customer after product delivery. Finding errors is good - we want to find as many errors as possible in the product before it is delivered to the customer. A good software tester will find many errors. A poor software tester will find few. Finding defects is bad - these are errors that we failed to find during the software development process and were delivered to the customer in the final product delivery.

We will discuss ways in which we can find errors in the software artifacts early in the software development process even before the code is produced later in this module. Suffice it to say at this point in time that the errors should be documented, tracked, and analyzed at the end of the production cycle to determine root causes and ways to improve the process so as not to produce the errors in the first place. A useful metric is Defect Removal Efficiency - number of errors found before delivery compared to defects found after delivery - DRE = E/(E+D).

There are quality measures that the software engineer may consider that are not specified by the customer, but rather are deemed necessary in constructing a sound product.
Examples of such quality factors are:

- Maintainability - mean-time-to-change (time it takes to analyze, implement, test, and distribute the change).

- Integrity - measures of attacks and threats averted.

Effort Metrics

Effort measurements will tell us how long it took to develop a product and the relative resource costs. Effort is measured by the project schedule. At the beginning of the project, the project schedule is established with the original estimates of each task and duration. During the project's execution, the project manager will track the actual duration of each task. The resulting estimates and actual results will be evaluated at the conclusion of the project and assessed for process improvement. This data, if captured in a repository can be evaluated collectively with other projects to form trend analysis.

Tips for establishing a successful software metrics program

- NEVER use metrics to assess, reward, or punish individual performance.
- The individual software engineers collect metrics during the software development process. The consistency and honesty in reporting these metrics ensures quality metrics. If individual contributors feel that the data is being used to reward or punish, they will naturally distort the figures reported.
- Get top-level management commitment to the metrics program including budget, staff, and authority to ensure consistent collection practices and procedures.
- Provide metrics training for everyone involved in the metrics program in the software engineering process.
- Evaluate the use of automated tools in the metrics collection and reporting process. These tools can be expensive, however, they can save time and money if applied properly to meet the organization's overall goals.
- KISS - "keep it simple stupid." Most metrics training programs provide a wide variety of metrics training needs. These metrics techniques need to be measured against the overall organizational goals and needs and tailored accordingly.

In the end, the skill and motivation of people has been shown to be the single most influential factor in quality and performance.

Lecture Notes: Part 2: Quality Assurance

Quality assurance tries to respond to the questions of what is quality and how will we know when we have it? Quality Control (QC) is an SQA activity that refers to the act of inspecting, reviewing, and testing. It is a general misconception that QC is synonymous with SQA. SQA refers to the strategic planning and management of ALL quality activities of which QC is one function.

Quality Management

The problem of quality management is not what people don't know about it. The problem is what they think they do know. The prevalent attitude in the software industry assumes that everyone already knows how to build quality software; therefore, no formal programs are required. This attitude could not be further from reality. Building quality software is not a talent one is born with; it is a learned skill and requires sound management practices to ensure delivery.

Quality does not mean the same thing to all people. It is important that each software development organization form a common definition of software quality practices for their group. Will the customer tolerate a certain defect level? As customers of software, we all have. You can purchase any piece of software off the shelf and you will find defects in the product. Not a single Microsoft product has been released without defects as it is rushed to market in an effort to beat competitors. We make our purchase and wait for the next release to fix the errors and introduce new functionality with new defects. These defects can range from merely annoying to life threatening. Consider, for example, the consequences of defects in aerospace software or in software that supports surgical procedures. Defects can cost billions of dollars if they are in financial software. How much quality testing is enough? One must further ask, how much would an error cost the customer? Testing practices should be as rigorous as the financial impact of the defect.

If we are producing a piece of desktop software for sale on the market at a retail price of $29.99, and an error would not be more than an irritant to the customer, it would not be cost effective to spend the time and money to ensure that the product was completely error free.

Some software developers continue to believe that software quality is something you begin to worry about after code has been generated. This attitude too, obstructs the production of quality software. A typical software development project is already running late by the time the coding begins and the whole team is frantically trying to make up for lost time. If most of the software testing is

reserved for the later part of the project, it typically is cut short to deliver the product on time. Consequently, defects are delivered to the customer.

Software quality is an "umbrella" activity and should be practiced throughout the entire software development lifecycle. It is not just a testing activity that is performed before the software is shipped. Software quality is not just a job for the Software Quality Assurance (SQA) group or team; EVERY member of the software engineering team performs quality assurance.

History of Quality Programs and their Application in the Software Industry

During the 1940's the U.S. was heavily entrenched in product manufacturing. Little importing or exporting occurred at this time. Therefore, the manufacturing firms had a captive and eager U.S. marketplace in which to sell its products. These products were not of the highest quality. Regardless, the manufacturer could not make the goods fast enough. Demand exceeded supply.

Dr. W. Edward Deming began to lecture U.S. manufacturers on ways to improve the quality of their products through the use of metrics and continuous process improvement techniques. In a market with little competition and in which demand exceeded supply, U.S. manufacturers showed little interest in Deming's ideas.

During this time, Japan was rebuilding after WWII and was very interested in competing in the world market. Dr. Deming took his quality message to Japan where it was well received. They implemented his programs and were able to produce higher quality products. The U.S. began to ease import restrictions and during the 60's and 70's these higher quality products appeared in the U.S. marketplace. The U.S. manufacturers quickly lost market share to the Japanese.

When Dr. Deming returned to the U.S. in the 1970's to deliver his quality improvement message, the U.S. manufacturers were ready to listen. During the 1980's these programs were known as Total Quality Management (TQM). However, the U.S. wanted a quick fix and did not fully comprehend the paradigm shift necessary to implement TQM. Many manufacturers invested large budgets in training programs, but failed to commit to real change. Even today, products bearing the name of SONY, Mitsubitsi, Fuji, Toyota, and Honda represent quality in the U.S. marketplace.

Many software development organizations attempted to implement Dr. Deming's TQM programs in the construction of software. Many of the practices and principles of manufacturing do apply to software development. However, the processes were different enough that the software development community realized they needed to expand the principles to establish models for software quality management. The Software Engineering Institute borrowed much from Dr.

Deming's TQM methods to establish the Capability Maturity Model (CMM). The CMM is the foundation for establishing sound, continuous, quality-improvement models for software development.

Quality Programs

Many major U.S. software development firms (IBM, TRW, Bell Labs, etc.) are serious about software quality and have implemented permanent quality programs. Other companies have yet to begin to determine what software quality is or whether it merits time and money investments for their business.

The CMM is more widely practiced and accepted in the U.S., the international standard for software quality is known as ISO 9001. The CMM and ISO both provide criteria for assessment certification and their standards are almost identical. An important distinction between the two is that the Software Engineering Institute provides certification at various levels (1-5), while ISO provides certification only when all standards are met. ISO 9001 provides 20 requirements for the Software Engineering industry in the areas of:

- Management,
- Quality systems,
- Contract review,
- Design control,
- Document and data control,
- Product identification and trace ability,
- Process control,
- Inspection and testing,
- Corrective and preventive action,
- Internal quality audits, and
- Training.

To receive ISO certification, a group must establish policies and procedures defined by the Standards Board and demonstrate that the procedures are being followed. Achieving certification informs potential customers that a firm is committed to implementing quality programs.

Fundamental Steps to Establish a Software Quality Program

1. Define what software quality means (in specific terms) for your organization or team.
2. Identify a set of activities that will filter errors out of the work products before they are passed on to the next activity.
3. Perform these activities.
4. Use audits and reviews to ensure these activities are being performed.

5. Use metrics to develop strategies for improving your software process and the quality of the end product.

Lecture Notes: Part 3: Configuration Management

Configuration: the arrangement of the parts or elements of a whole.

Management: the practice of tracking and controlling an activity

Configuration Management (CM): the practice of tracking and controlling the arrangement of parts of a whole product.

Configuration Management is an "umbrella" activity that is performed throughout the entire software development lifecycle. CM is usually assigned as a management responsibility of the Software Quality Assurance (SQA) group/team. As with all SQA activities, everyone on the project team participates in CM.

Software Configuration Items (SCIs), sometimes referred to as artifacts, are computer programs (code), documentation, models, data schemas, and any other component produced during the software development process. These artifacts are typically stored and tracked in a repository using an automated CM tool. Working with the project manager, the SQA group identifies which work products must be managed closely as they change what will come under Configuration Management. These will be identified in the SQA Plan.

The first law of system engineering is that "No matter where you are in the system life cycle, the system will change, and the desire to change it will persist throughout the lifecycle." Change is inevitable.

Failure to properly analyze changes or to assess, record, and report impacts creates confusion and even chaos. Configuration Management practices were established to control change throughout the software development lifecycle.

Configuration Management activities include:

- Identification – specify which artifacts will come under CM
- Version Control – establish baseline procedures and identify tools
- Change Control – establish procedures and reviews required to made changes to base-lined artifacts
- Configuration Audit – identify when applying changes to one artifact. For instance, when applying a change to a design model, will it affect the analysis model, code, and other pertinent documentation?
- Status Reporting – document what changes were made? By whom? When was the change made? What was the impact?

Once an artifact has undergone formal technical review and is ready to pass onto the next development phase, it is "base-lined" and placed in the CM repository. This product will be referred to as version 1.0. Once the artifact is base-lined, it must go through a formal change process to be modified.

This process is referred to as Version Control. Version Control enables the team to trace changes and provide an audit trail later in the product lifecycle. The Project Plan is usually one of the first artifacts to be base-lined. The original estimates and scope for the project are preserved. As the project scope expands and estimates are adjusted, incremental versions of the plan are generated.
Likewise, as analysis models are reviewed and base-lined and requirement changes are introduced during the coding phase, new versions of the models are generated to reflect the changes preserving the older versions for trace ability. It is not uncommon that changes that were made are inadvertently removed and the older version is restored.

Many companies maintain multiple versions of their software. Version 1.0 may be installed at one customer site while other customers are implementing the latest release version 2.0. It is very important that the company keep track of which versions of which artifacts go with which software product releases.

Some companies will also manage the software tool sets used to produce their software products as configuration items. For instance, which version of the tool used to produce the analysis and design models was used for which versions? It is very important to capture information such as the use of a specific tool to produce a product. If we used version 3.3 of the modeling tool to produce analysis model version 1.0 through 3.0 and then upgraded the modeling tool to version 3.4 to produce analysis models version 4.0 and higher, we might have a problem. Analysis models 1.0 through 3.0 might not be readable by the newer version of the modeling tool.

It is the job of the SQA team to make sure that everyone on the project understands which artifacts are to come under CM, which tools are being used, and that everyone receives training in the proper use of those tools. The SQA team will conduct audits during the development process to ensure that procedures are being followed.

Module Eight Team Activities:

- Work as a team to develop the Design Document
- Work as a team to develop a Working Prototype

(This page left blank for your notes)

Module Nine

Objectives:

- Software Maintenance
- Debugging and Testing

Assignments/Activities:

- Design Specification Due
- Java Programming Lab

Lecture Notes: Part 1: Software Maintenance

Out of all of the phases in a life cycle, maintenance is the most costly in relation to the other phases. Data gathered in the late 1970s and early 1980s by Boehm and others showed that maintenance costs represent 67% of total life cycle costs. Maintenance is not only the most costly part of a project; it is usually the longest phase in a project. Although the length of the maintenance phase is out of our control, we can manage the costs associated with this phase. Maintenance costs can be reduced by several techniques such as creating an appropriate level of documentation of sufficient quality, and by writing quality code.

As time goes on costs increase and knowledge disappears. The disappearance of knowledge is best dealt with through documentation. Increased levels of documentation reduce the need for maintenance programmers to reverse engineer and figure out what's going on in the code. Reducing this time reduces the costs associated with maintenance. Quality also plays an important factor in maintenance. In maintenance, quality is viewed in respect to the maintainability of code. Code maintainability consists of the following characteristics: testable, understandable, modifiable, portable, reliable, efficient, and usable. These characteristics can judge through quality checklists, quality tests, and quality metrics. Quality can also be concerned with the adherence to coding standards and conventions.

Lecture Notes: Part 2: Debugging and Testing

One means of reducing maintenance costs and increasing quality is through a rigorous set of debugging and testing. Debugging is what you do while you're writing code. During the debugging process you are working out the details of what you've written, trying to ensure that it is correct and does not blow up. Debugging is not the same as testing. Testing is designed to ensure that the program is reliable, performs correctly, and does what it's supposed to do.

Most programmers begin testing by making sure the program performs correctly. Checking to make sure a program performs correctly is not the same thing as checking the reliability of a program. Reliability is concerned with the program performing under all uses. For example, is the program able to add two real numbers? What happens if the user types erroneous data such as alpha characters for the first input value? Does the program blow up or print out an error message? A program that works correctly and does not blow up is not necessarily a program that does what it's supposed to do. Tracing requirements to the test process ensures that the product meets the expectations of the client and user community. An integral part of testing is making sure that requirements have been met.

Depending on the size of your organization and the degrees of formality in your software development life cycle, testing can be the responsibility of the programmer, a Quality Assurance (QA) team or an Independent Verification and Validation team. No matter what's in place at your organization, testing begins with the programmer.

Unit testing is what a programmer does to ensure that everything they coded is correct. This type of testing begins when the programmer has decided that they are done coding and debugging. In a formal environment, the programmer is expected to document unit tests by recording test cases. A test case consists of input, expected output and the results of the specific test case. It is also common for the programmer to map requirements to the test cases. This mapping ensures that the product does what its required to do by the client or designers. Once the programmer is finished testing the product is ready for alpha and/or beta testing.

Whether you conduct alpha and/or beta testing is really dependent on the size of the product and the size of the user community. For example, the QA organization at USPS uses both alpha and beta testing techniques. After successful completion of unit testing the product is ready for an "alpha" test. Alpha testing consists of releasing the product to one or two key users. This initial release provides testers with additional feedback on the product and it also provides a level of user testing that is difficult to duplicate in-house. Users always seem to use a product in manner in which the developers did not intend. At the conclusion of alpha testing the tests are expanded to a greater number of sites or

users during beta testing. This last set of tests is designed to ensure that the product is reliable and does meet the needs of the user community by performing in a real-world environment.

Independent Testing

Personnel who are not part of the project team perform Independent Testing. That is, they are not accountable for delivering the product on time or within budget. Independent testing is important because the project team may unwittingly compromise testing in order to meet budget and schedule pressures. An Independent Testing Team is not accountable for meeting the project dates and can focus solely on the quality of the product without this conflict of interest. Independent Testers normally report to senior management or to the customer, never to the Project Manager.

Debugging

If an error is found during a test and it is determined to be a result or an error (or bug) in the software, the problem must be fixed. The act of debugging is not part of testing per se, but occurs in response to errors identified through testing. Debugging includes identifying and repairing bugs and retesting the software.

Regression Testing

Whenever software is modified to correct an error it must be retested. However, it is always possible that the repaired software may introduce problems in areas that have already been tested. Regression Testing refers to the process of retesting of the software to prove that a problem has been fixed and that no new problems have been introduced. Full Regression Testing that would retest all prior conditions is not feasible. Instead, analysis is done to determine what areas are likely to have been impacted and need to be retested. The use of automated software to re-execute prior conditions and compare the new results and previous results is essential to robust Regression Testing.

Testing Types

There are a number of categories of testing that can be accomplished for software. Each category addresses the software from a unique perspective. The four basic tests that are accomplished for almost all systems are Unit, Integration, Validation, and System Testing. There are a variety of other tests as well that may be performed as part of, or in addition to, these four basic tests. The more common categories of testing are listed below.

Unit Testing

Unit Testing refers to the testing of individual modules to ensure that they operate correctly independently of other modules. This is the first level of testing and is normally accomplished by the developer. Unit Testing is White Box testing techniques.

Integration Testing

Integration Testing ensures that all modules interoperate properly and can be integrated successfully into a functioning system. The development team performs integration testing once Unit Testing is complete and prior to Validation Testing.

Validation Testing

Validation Testing is Black Box testing that ensures the customers requirement is met. The end user or representatives of the end user normally perform Validation Testing. Validation Testing is the most important testing phase from the perspective of user acceptance of the system.

System Testing

System Testing follows Validation Testing and ensures that the software will operate properly within the overall target environment, including other existing systems.

Acceptance Testing

Acceptance Testing is a formal approach to Validation Testing where the test becomes the official basis for user acceptance of the completed software. It is commonly used when a formal contract is in place for the development of software. Once the customer signs off on the Acceptance Test they are certifying that the contract is complete.

Beta Testing

An end user of a finished product performs Beta Testing in the end user's environment. The end user is commonly given unrestricted use of the software in exchange for providing information about software defects. Beta Testing is commonly used for software that operates on a workstation; it is provided to users who are likely to apply the software in a variety of ways. This approach is helpful when it is difficult to completely test software using formal test cases and predicted results.

Performance Test

A Performance Test specifically focuses on the ability of the software to meet specified performance requirements. It may be completed as a part of some other testing phase. During a performance test the response time and execution time is carefully monitored in response to controlled or measured input conditions.

Stress Test

A Stress Test is a form of Performance Testing in which the software is subjected to large volumes of transactions in an effort to find the breaking point. The Stress Test indicates the volume the software will be able to handle. It also provides input for future planning when the performance requirements are met but the user base is likely to grow over time.

Security Test

A Security Test investigates the capacity of the software to withstand security attacks. Testers intentionally try to violate security procedures established for the system. Security Testing is commonly conducted in conjunction with some other phase of testing.

Implementation Test

An Implementation Test is a formal test of the implementation procedures for the software. An implementation team will attempt to implement the software on the target platform using only the formally defined implementation procedures. This testing uncovers weaknesses or omissions in the implementation procedures.

Parallel Test

During a Parallel Test the new software is executed in parallel with existing software. This provides a baseline for comparison of products of the new system to ensure that they are the same as in the old (or that any differences can be explained in terms of new requirements). It also provides a fall back system in the event the new software is proven to be inadequate.

Initial Operational Test

Initial Operational Testing may occur for a specified period after the software is put into production. This is not testing in the strictest sense, but rather implies that a formal predefined scrutiny will be applied to all products from the software until verification that the software is functioning properly is obtained.

Environmental Test

In an Environmental Test the software is operated within the actual environment in which the software will operate in production. This is necessary when the test environment does not fully represent the production environment - which is commonly the case.

Recovery Test

A Recovery Test is executed to test the recovery procedures for the software. When the software fails the specified recovery procedures are executed to ensure that they achieve the appropriate result. Recovery Testing is often accompanied by efforts to force the software to fail.

Module Nine Team Activities:

- Work as a team to develop a Working Prototype

Module Ten

Objectives:

- Final Exam Prep
- Team Presentations

Assignments/Activities:

- Final Exam
- Prototypes Due
- Peer Evaluations

Lecture Notes: There will be no new Software Engineering concepts introduced in this Module. Each team will present their project to the class and demonstrate their prototype.

Module Ten Team Activities:

- As a team, present your project to the class.

You should be prepared to discuss the project objectives, scope, methods of attaining the appropriate outcomes, lessons learned and any strengths or weaknesses of the project. You should be prepared to answer any questions the instructor or other participants might have regarding your deliverables.

Team Member Peer Evaluations are to be used to provide input on your teammate's participation on the project. These should be delivered directly to the instructor. Your input will not be shared with other team members.

(This page left blank for your notes)