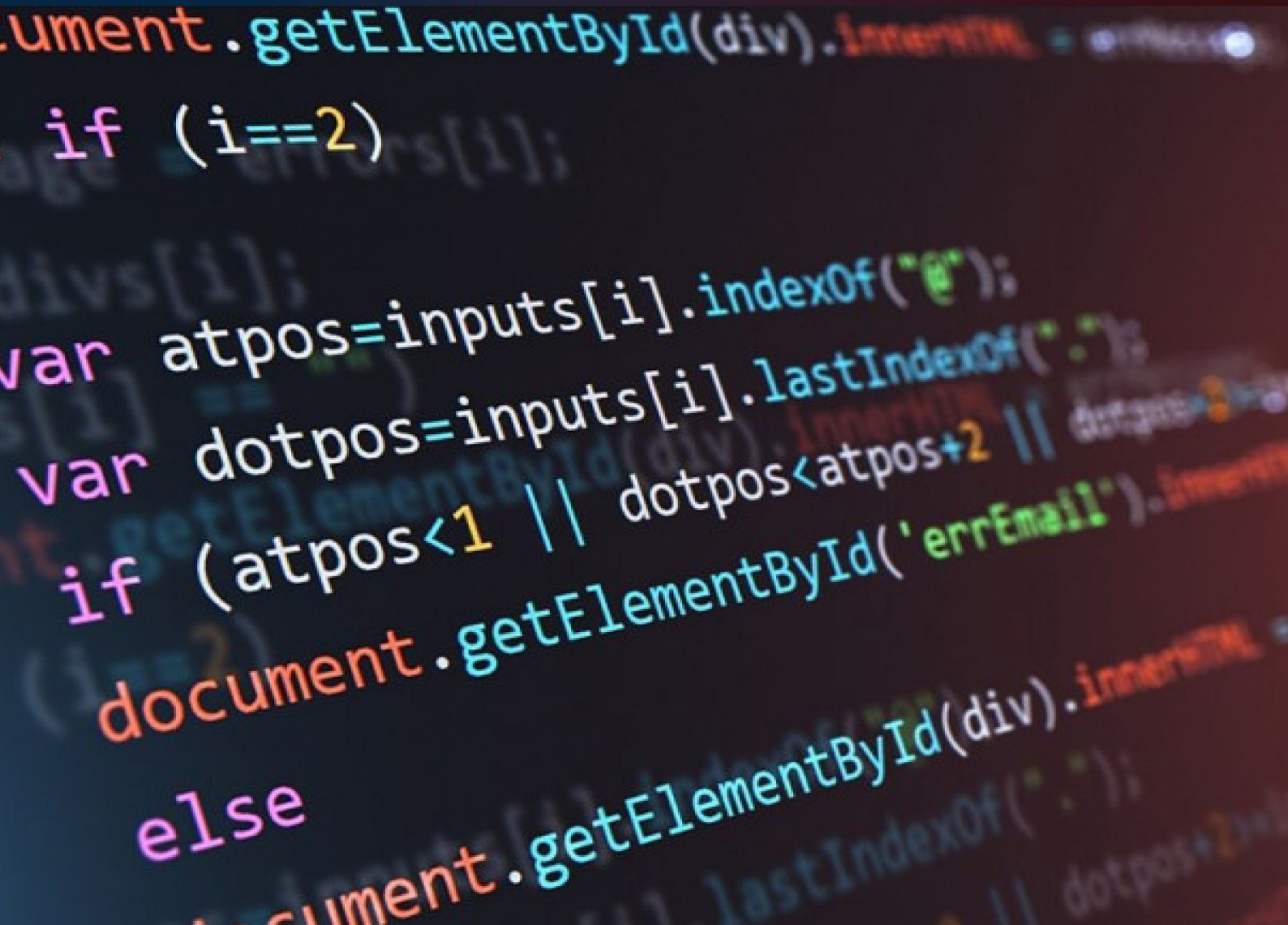
# Software Engineering in Configuration Management

Wesley Ryan



# SOFTWARE ENGINEERING IN CONFIGURATION MANAGEMENT

# SOFTWARE ENGINEERING IN CONFIGURATION MANAGEMENT

Wesley Ryan

BIBLIOTEX
Digital Library

Software Engineering in Configuration Management
by Wesley Ryan



www.bibliotex.com



Ebook ISBN: 9781984663986

Published by:
Bibliotex
Canada
Website: www.bibliotex.com

# Contents

# 1

# Introduction

## Concept

There hardly existed any specific documentation, system design approach and related documents etc. These things were confined to only those who developed hardware systems. Software development plans and designs were confined to only concepts in mind. Even after number of people jumped in this field, because of the lack of proper development strategies, documentations and maintenance plans, the software system that was developed was costlier than before, it took more time to develop the entire system (even sometimes, it was next to impossible to predict the completion date of the system that was under development), the lines of codes were increased to a very large number increasing the complexity of the project/software, as the complexity of the software increased it also increased the number of bugs/

problems in the system. Most of the times the system that was developed, was unusable by the customer because of problems such as late delivery (generally very very very late) and also because of number of bugs, there were no plans to deal with situations where in the system was needed to be maintained, this lead to the situation called 'Software Crisis'. Most of software projects, which were just concepts in brain but had no standard methodologies, practices to follow, experienced failure, causing loss of millions of dollars.

'Software Crisis' was a situation, which made people think seriously about the software development processes, and practices that could be followed to ensure a successful, cost-effective system implementation, which could be delivered on time and used by the customer. People were compelled to think about new ideas of systematic development of software systems.

This approach gave birth to the most crucial part of the software development process, this part constituted the most modern and advanced thinking and even the basics of any project management, it needed the software development process be given an engineering perspective thought. This approach is called as 'Software Engineering'. Standard definition of 'Software Engineering' is 'the application of systematic, disciplined, quantifiable, approach to the development, operation and maintenance of software *i.e.* the application of engineering to software.'

The Software Engineering subject uses a systematic approach towards developing any software project. It shows how systematically and cost-effectively a software project can be handled and successfully completed assuring higher success rates.

Software Engineering includes planning and developing strategies, defining time-lines and following guidelines in order to ensure the successful completion of particular phases, following predefined Software Development Life-Cycles, using documentation plans for follow-ups etc. In order to complete various phases of software development process and providing better support for the system developed.

Software Engineering takes an all-round approach to find out the customer's needs and even it asks customers about their opinions hence proceeding towards development of a desired product. Various methodologies/practices such as 'Waterfall Model', 'Spiral Model' etc. Are developed under Software Engineering which provides guidelines to follow during software development ensuring on time completion of the project.

These approaches help in dividing the software development process into small Tasks/phases such as requirement gathering and analysis, system design phase, coding phase etc. That makes it very much easy to manage the project. These methods/approaches also help in understanding the problems faced (which occur during the system development process and even after the deployment of the system at customer's site) and strategies to be followed to take care of all the problems and providing a strong support for the system developed (for example: the problems with one phase are resolved in the next phase, and after deployment of the product, problems related to the system such as queries, bug that was not yet detected etc. which is called support and maintenance of the system.

## Software Progrmming

Software Engineering is an approach to developing software that attempts to treat it as a formal process more like traditional engineering than the craft that many programmers believe it is. We talk of crafting an application, refining and polishing it, as if it were a wooden sculpture, not a series of logic instructions. Manufacturers cannot build complex life-critical systems like aircraft, nuclear reactor controls, medical systems and expect the software to be thrown together.

They require the whole process to be thoroughly managed, so that budgets can be estimated, staff recruited, and to minimize the risk of failure or expensive mistakes. In safety critical areas such as aviation, space, nuclear power plants, medicine, fire detection systems, and roller coaster rides the cost of failure can be enormous as lives are at risk. A divide by zero error that brings down an aircraft is just not acceptable.

### Cad Engineering

Enormous design documents- hundreds or thousands of pages long are produced using C.A.S.E. (Computer Aided Software Engineering) tools then converted into Design Specification documents which are used to design code.

C.A.S.E suffers from the “not quite there yet” syndrome. There are no systems that can take a set of design constraints and requirements then generate code that satisfies all the requirements and constraints. Its far too complex a process. So the available C.A.S.E. systems manage parts of the lifecycle process but not all of it. One distinguishing

feature of Software Engineering is the paper trail that it produces. Designs have to be signed off by Managers and Technical Authorities all the way from top to bottom and the role of Quality Assurance is to check the paper trail. Many Software Engineers would admit that their job is around 70% paperwork and 30% code. It's a costly way to write software and this is why avionics in modern aircraft are so expensive.

## Basic Software Components

Software can be further divided into seven layers. Firmware can be categorized as part of hardware, part of software, or both. The seven layers of software are (top to bottom): Programmes; System Utilities; Command Shell; System Services; User Interface; Logical Level; and Hardware Level. A Graphics Engine stradles the bottom three layers.

Strictly speaking, only the bottom two levels are the operating system, although even technical persons will often refer to any level other than programmes as part of the operating system (and Microsoft tried to convince the Justice Department that their web browser application is actually a part of their operating system). Because this technical analysis concentrates on servers, Internet Facilities are specifically separated out from the layers.

Human users normally interact with the operating system indirectly, through various programmes (application and system) and command shells (text, graphic, etc.), The operating system provides programmes with services thrrough system programmes and Application Programme Interfaces (APIs).

## Network and Internet Services

- Internet
- TCP/IP
- Server choices
- Tuning web servers
- DHCP
- Print serving
- File serving
- FTP
- SAMBA
- Mail Transport Agents (e-mail servers)
- Majordomo
- Application serving

## Hardware Level of Operating System

## Basics of Computer Hardware

- Processor
- Arithmetic and logic
- Control
- Main storage
- External storage
- Input/output overview
- Input
- Output

## Processors

- CISC
- RISC
- DSP
- Hybrid

## Processes and Jobs

- General information
- Linking
- Loading
- Run/execute

## Buses

- Kinds of buses
- Bus standards

## Memory

- Main storage
- External storage
- Buffers
- Absolute addressing
- Overlay
- Relocatable software
- Demand paging and swapping
- Programme counter relative
- Base pointers
- Indirection, pointers, and handles
- OS memory services

## Memory Maps

- PC-DOS and MS-DOS memory map
- MS-DOS TSR memory map
- Mac Plus memory map
- Mac Plus video memory locations
- Mac Plus sound memory locations

## Low Memory

## PC-DOS and MS-DOS low memory

- BIOS Communication Area
- Reserved
- Inter-Application (User) Communication Area
- DOS Communication Area

## Character codes

## Logical Level of Operating System

- File systems
- Files
- Resource Manager
- Cut and paste

## Graphics Engine

- Font Management

## User Interface

- Command line user interfaces
- Graphic user interfaces
- Aqua
- Common Desktop Environment
- IRIX Interactive Desktop
- Macintosh Toolbox
- Motif
- Visual User Environment
- Workbench
- XFree86
- Spoken user interfaces

- Screen shots
- Event Management
- Windows
- Controls
- Menus
- Text Display and Editing
- Dialog Boxes
- Alerts

### System Services

### Command Shell

- Command line command shells
- DCL
- DOS
- JCL
- UNIX shells
- Scripting
- Graphic command shells
- Screen shots

### System Utilities

### Programmes

- Desk Accessories

## Software Characteristics

### Software requirement

- Microsoft Windows 98 SE, Me, NT4 (sp5+), 2000 or XP,

- Word processing software (optional),
- Spell checker (optional),
- Spreadsheet (optional), Microsoft Excel is necessary to generate analysis reports
- Web browser (optional), Internet Explorer 5 or Netscape 6 or above,
- Adobe Acrobat (optional).

## Hardware requirement

- PC compatible computer (Pentium II or compatible),
- CD-ROM Drive,
- SVGA or XGA (1024×768) graphic screen and card,
- Floppy drive (optional, for Ethnos input transfer),
- Printer port (parallel port RS232).

## Text Analysis

- Minimum size advised for a text: less than 1 page (1 Kb),
- Maximum size advised for a single text: 5,000 pages (50 Mb),
- Average analysis throughput: from 20,000 words/second (Pentium III 733 MHz) to 80,000 words/second (Pentium IV 3.2 GHz, HT) on local Web pages, for a single processor.

## Semantic Search Engine

- Automatic generation of hierarchical keywords,
- Automatic information filtering (based on a pertinence treshold),
- Massive data analysis and information cartography (text-mining),

- Search improvement for the references (nouns, trademarks and proper names),
- Maximum numbers of text databases: unlimited,
- Average indexing throughput: from 1 Gb/hour (Pentium III 733 Mhz) to 4 Gb/hour (Pentium IV 3.2 GHz, HT) on local Web pages, for a single processor.

**features**

- File formats converted by our linguistic softwares (Tropes, Zoom and Index): Adobe Acrobat, ASCII, ANSI, HTML, Macromedia Flash, Microsoft Excel, Microsoft Powerpoint, Microsoft Word, Microsoft WordML (Word XML), RTF, XML, SGML and Macintosh texts
- Automatic extraction of Microsoft Outlook messages via an external utility (Zoom Semantic Search Engine)
- Automatic exportation of the results towards other software (Zoom Semantic Search Engine)
- Indexing engine in batch mode (Acetic Index)
- Win32 Application Programming Interface (Acetic Index)
- Real time XML output interface (Acetic Index)
- Distributed fault tolerant and load-balancing Interface (CORBA, Acetic Index)
- Runtime, operation on Intranet, HTML generation (contact us)
- Some features (for example, very large Text Mining) may require the use of an additional statistics software, of data mining software and/ or a RDBMS

# Software Features

## Software products

- Successful software
- Provides the required functionality
- Is usable by real (*i.e.* naive) users
- Is predictable, reliable and dependable
- Functions efficiently
- Has a "life-time" (measured in years)
- Provides an appropriate user interface
- Is accompanied by complete documentation
- May have different configurations
- Can be "easily" maintained

## Software Consumer

- Cheap to buy
- Easy to learn
- Easy to use
- Solves the problem
- Reliable
- Powerful
- Fast
- Flexible
- Available

## Requirement of Software producer

- Cheap to produce
- Well-defined behaviour
- Easy to "sell"
- Easy to maintain

- Reliable
- Easy to use
- Flexible
- Available (quick to produce)

### Issue of measurement

- The issue is…how to measure these things
- Why measure at all?
- Human subjective perception is notoriously inaccurate (how many shark attacks in the last 200 years?)
- Numbers give us a way of comparing, controlling and predicting
- Measurements give us a way of tracking progress (and rescheduling if necessary)
- Also provide an assessment of product quality
- Measurement is the difference between "craft" and "engineering"

## Metric

- "A quantitative measure of the degree to which a system component or process possesses a given attribute (IEEE)
- Hence, for each metric, we require…
- A measurable property
- A relationship between that property and what we wish to know
- A consistent, formal, validated expression of that relationship
- *For example*: who is the greatest actor of all time?

## Good Metric

- Simple and computable
- Persuasive
- Consistent/objective
- Consistent in use of units/dimensions
- Programming language independent
- Gives useful feedback

## Process metrics

- Measures of attributes of a process
- Attributes may relate to people (*e.g.* "person-hours")...
- Or technology (*e.g.* "megaLOCs")...
- Or the product (*e.g.* "total cost to date")

## Measurement

- Effort, time and capital spent on various related activities
- Number of functionalities implemented
- Number of errors remediated (of various severities)
- Number of errors *not* remediated (during development process)
- Conformance to delivery schedule
- Benchmarks (speed, throughput, error-rates, etc)

## Hard Measure

- Abstract desiderata
- Usability
- Efficiency
- Reliability
- Maintainability
- Quality

## Standard Code of Metrics

- Lines of code (LOC)
- Cyclomatic complexity (McCabe)
- Function/feature points (Albrecht/Jones)

## Lines of Code

- A size-oriented metric
- Easy to measure
- Easy to compare
- Easy to differentiate wrt time, cost, etc.
- Programming language dependent (*e.g.* 1 OO-LOC = 3 3GL-LOC = 9 assembler-LOC)
- Meaningless in isolation
- Penalize efficient design and coding

## Object-oriented Metrics

- Measures of...
- Classes
- Encapsulation
- Modularity
- Inheritance
- Abstraction

## Some Object-oriented Metrics

- Chidamber and Kemerer
- Lorenz and Kidd

## Chidamber and Kemerer's

- Class-oriented metrics (*Proc. OOPSLA*)...
- Weighted methods per class (number of methods weighted by static complexity)

- Depth of inheritance tree (number of ancestral classes)
- Number of children (number of immediate subclasses)
- Degree of coupling (how many other classes rely on the class, and vice versa)
- Response (number of public methods)
- Method cohesion (degree to which data members shared by two or more methods)

## Software Crisis

Indeed, the problem of trying to write an encyclopedia is very much like writing software. Both running code and a hypertext/encyclopedia are wonderful turn-ons for the brain, and you want more of it the more you see, like a drug. As a user, you want it to do everything, as a customer you don't really want to pay for it, and as a producer you realise how unrealistic the customers are. Requirements will conflict in functionality vs affordability, and in completeness vs timeliness.

### different Types of Crisis

### Chronic Software Crisis

By today's definition, a "large" software system is a system that contains more than 50,000 lines of high-level language code. It's those large systems that bring the software crisis to light. If you're familiar with large software development projects, you know that the work is done in teams consisting of project managers, requirements analysts, software engineers, documentation experts, and programmers.

With so many professionals collaborating in an organized manner on a project, what's the problem? Why is it that the team produces fewer than 10 lines of code per day over the average lifetime of the project? And why are sixty errors found per every thousand lines of code? Why is one of every three large projects scrapped before ever being completed? And why is only 1 in 8 finished software projects considered "successful?"

- The cost of owning and maintaining software in the 1980's was twice as expensive as developing the software.
- During the 1990's, the cost of ownership and maintenance increased by 30% over the 1980's.
- In 1995, statistics showed that half of surveyed development projects were operational, but were not considered successful.
- The average software project overshoots its schedule by half.
- Three quarters of all large software products delivered to the customer are failures that are either not used at all, or do not meet the customer's requirements.

Software projects are notoriously behind schedule and over budget. Over the last twenty years many different paradigms have been created in attempt to make software development more predictable and controllable.

While there is no single solution to the crisis, much has been learned that can directly benefit today's software projects.

*It appears that the Software Crisis can be boiled down to two basic sources*:

1. Software development is seen as a craft, rather than an engineering discipline.
2. The approach to education taken by most higher education institutions encourages that "craft" mentality.

## Software Development

Software development today is more of a craft than a science. Developers are certainly talented and skilled, but work like craftsmen, relying on their talents and skills and using techniques that cannot be measured or reproduced. On the other hand, software engineers place emphasis on reproducible, quantifiable techniques–the marks of science. The software industry is still many years away from becoming a mature engineering discipline. Formal software engineering processes exist, but their use is not widespread. A crisis similar to the software crisis is not seen in the hardware industry, where well documented, formal processes are tried and true, and ad hoc hardware development is unheard of. To make matters worse, software technology is constrained by hardware technology. Since hardware develops at a much faster pace than software, software developers are constantly trying to catch up and take advantage of hardware improvements.

Management often encourages ad hoc software development in an attempt to get products out on time for the new hardware architectures. Design, documentation, and evaluation are of secondary importance and are omitted or completed after the fact. However, as the statistics show, the ad hoc approach just doesn't work. Software developers have

classically accepted a certain number of errors in their work as inevitable and part of the job. That mindset becomes increasingly unacceptable as software becomes embedded in more and more consumer electronics. Sixty errors per thousand lines of code is unacceptable when the code is embedded in a toaster, automobile, ATM machine or razor (let your imagination run free for a moment).

## Computer Science and Product Orientation

Software developers pick up the ad hoc approach to software development early in their computer science education, where they are taught a "product orientation" approach to software development. In the many undergraduate computer science courses I took, the existence of software engineering processes was never even mentioned.

Computer science education does not provide students with the necessary skills to become effective software engineers. They are taught in a way that encourages them to be concerned only with the final outcome of their assignments–whether or not the programme runs, or whether or not it runs efficiently, or whether or not they used the best possible algorithm. Those concerns in themselves are not bad. But on the other hand, they should not be the focus of a project. The focus should be on the complete process from beginning to end and beyond. Product orientation also leads to problems when the student enters the work force–not having seen how processes affect the final outcome, individual programmers tend to think their work from day to day is too "small" to warrant the application of formal methods.

## Fully Supported Software

As we have seen, most software projects do not follow a formal process. The result is a product that is poorly designed and documented. Maintenance becomes problematic because without a design and documentation, it's difficult or impossible to predict what sort of effect a simple change might have on other parts of the system. Fortunately there is an awareness of the software crisis, and it has inspired a worldwide movement towards process improvement. Software industry leaders are beginning to see that following a formal software process consistently leads to better quality products, more efficient teams and individuals, reduced costs, and better morale.

Ratings range from Maturity Level 1, which is characterized by ad hoc development and lack of a formal software development process, up to Maturity Level 5, at which an organization not only has a formal process, but also continually refines and improves it. Each maturity level is further broken down into key process areas that indicate the areas an organization should focus on to improve its software process (*e.g.* requirement analysis, defect prevention, or change control).

Level 5 is very difficult to attain. In early 1995, only two projects, one at Motorola and another at Loral (the on-board space shuttle software project), had earned Maturity Level 5. Another study showed that only 2% of reviewed projects rated in the top two Maturity Levels, in spite of many of those projects placing an extreme emphasis on software process improvement.

Customers contracting large projects will naturally seek organizations with high CMM ratings, and that has prompted increasingly more organizations to investigate software process improvement. Mature software is also reusable software. Artisans are not concerned with producing standardized products, and that is a reason why there is so little interchangeability in software components.

Ideally, software would be standardized to such an extent that it could be marketed as a "part", with its own part number and revision, just as though it were a hardware part. The software component interface would be compatible with any other software system. Though it would seem that nothing less than a software development revolution could make that happen, the National Institute of Standards and Technology (NIST) founded the Advanced Technology Programme (ATP), one purpose of which was to encourage the development of standardized software components.

# 2

# Software Architectural Design

Design principles are not necessarily right or wrong but should be an accurate reflection of the fundamentals that guide decision making in an enterprise. The following should therefore not be seen as design principles fixed in concrete but rather as examples of business principles.

Best practice is to define the design principle in terms of its Benefits and rationale as well as the implication to the enterprise and the counter argument expressing the potential negative impact of the design principle.

### Design principle

### Description

All management information and business intelligence will be sourced from a single consolidated source of information

## Benefits

- A central source of management information will provide the enterprise with a wide breath of reporting and analysis without being constraint by the organisations functional structuring.
- Users will become used to a single interface to management information allowing managers to become familiar with the infrastructure and extracting maximum benefit from all information available in the organisation.
- The central information will eliminate contradicting information sources and ensure accurate reporting of current affairs and identification of issues and opportunities.
- Increase the flexibility and manageability of providing information rapidly and effectively to support business decisions.
- Use best of breed analytic functionality to support management decision making
- Increased security in managing access to The enterprise's management information

## Implications

- Information must be sourced to the central information infrastructure from all the various operational applications as close to real time as possible
- No additional analytical modules are required for transactional applications.
- The interface to management information and training should be rolled out to all decision makers to effectively access information.

## Counter Argument

- The central management information might not be adequate in situations where real time analytics of transactional information is needed
- Information in the central information source might not be structured for a specific requirement and the development time might be too long to provide the information in time for a once off request.
- The assumption cannot be made that a single tool will satisfy all the information requirements. The information infrastructure will therefore consist of a variety of integrated tools.

## Internationalisation

## Description

Information must be structured for global deployment in various cultures and support multi-currency, multi-language and multi platforms

## Benefits

- Flexibility to enter into other global markets
- Consistency of being able to deploy a proven business model and then adapt to local conditions

## Implications

- Applications should be much more flexible to accommodate differences defined by different countries
- Current applications should be evaluated in terms of internationalisation requirements.

### Counter Argument

- Applications designated for use locally in South-Africa only does not have to comply to the internationalisation principle
- The enterprise might strategise to enter only into markets that have a certain set of international commonality which make the rigid application of this principle unnecessary
- It might be too expensive to change or replace legacy systems to adhere to the internationalisation requirement.

## Single contact database

### Description

A single contact database for all business contacts *e.g.* policyholders, intermediaries and service providers

### Benefits

- A central source of information to manage relationships effectively
- Have a more comprehensive view of interrelationships between business contacts.
- More effective marketing campaign design and management

### Implication

- High levels of data integration with transactional systems updating contact information.
- Transactional systems updating information must be assigned with levels of trust.

- Central contact information should not.complicate functional requirements to only view specific relationships.

**Counter Argument**

- Some functional units might not want to share contact information for the fear that it might be used wrongly or out of context by other parts of the organisation.

**Single point of authentication**

**Description**

Access for to information should be constraint through a single point of authentication infrastructure

**Benefits**

- Improved security of information

**Implication**

- Central management of user access to information.

**Counter Argument**

- The current infrastructure might not be mature enough to implement the principle.

**Data quality measurement**

**Description**

Data quality will be measured both in quantitative and qualitative terms eg. Audit procedures and Data quality questionnaires

## Benefits

- Improved data quality
- Accurate usage of data
- Improved management information
- Increased operational efficiency

## Implication

- Bi-annually measure data through with a data quality survey
- Audit data ownership procedures annually
- Build in data quality measures into applications
- Connect data quality results with performance incentives

## Counter Argument

- Regular audits and subjective measurement techniques *e.g.* surveys might be time consuming.

## Formalised data exchange/enrichment

## Description

All data exchange/enrichment activities are managed and approved by the appointed data strategist and the exchange of information must be subjected to a standardised methodology for information exchange/enrichment.

## Benefits

- Control costs associated with data exchange and enrichment
- Protect operational data
- Improve data quality and value
- Data sharing to allow for better detection of fraud

## Implication

- Development of a formal policy and methodology for data exchange and enrichment
- Data strategist must be responsible for approving data exchange/enrichment efforts and minimise cost.
- Identification and management of organisations that can enrich and/or validate the enterprise data.

## Counter Argument

- The formalised methodology should not become a constraint to enrich and improve data quality.
- The enterprise might not always be in a position to demand compliance from external parties to comply with data sharing standards.
- Current lack of industry standards might make it difficult to implement the principle.

## Central repository of data naming standards

## Description

Data names and field content must be standardised through a central reference repository and must be accessible to the business *e.g.* Street rather than str. is used to reference a street.

## Benefits

- Consistency of information across all business processes
- Usability of information increases across the organisation.

### Implication

- Alignment of all applications to support the standardised naming standards

### Counter Argument

- Difficulty to implement naming standardisation in some applications

## Align information requirements with data model

### Description

All information requirements must be aligned with the corporate data model before requesting changes to the information architecture:

### Benefits

- Integrity of transactional data model stays in tact.
- Prevent duplication of information.

### Implication

- The corporate data model must be maintained to be up to date at all times.
- In any application development life cycle it is a condition to align the application with the corporate data model.

### Counter Argument

Data owners might not understand the data model to update it with changes.

## Information governance on all data elements

### Description

All information elements must be subjected to information architecture governance

### Benefits

- Sustain data quality.
- Improve operational efficiency.

### Implication

- Design the business process application of the data element.
- Assign Applications sourcing the information.
- Align with corporate data model, rules, validations, naming standard.
- Define data management policies *e.g.* security, back-up, archiving/retrieval.
- Assign data ownership.

### Counter Argument

- Lack training to adhere to information governance.
- Information governance is not adequately communicated.

## Data privacy and legality

### Description

Client privacy must be respected and legal requirements must be complied with, in any event of data exchange or commerce.

**Benefit**

- Maintain good relationships with clients and protect premium income
- Avoid legal costs due to mismanagement of information resulting in lawsuits.

**Implication**

- The enterprise must be up to date with laws relating to information
- Communicating The enterprise's data policy to clients
- Legal department needs to be up to date with laws governing information usage, commerce and distribution

**Counter Argument**

- Uncertainty of what constitutes data privacy might make it difficult to implement this principle.

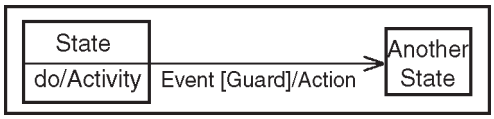
*Bottom line*: Design principles for enterprise architecture must provide a decision framework based on a clear rationale defined in terms of the benefits, implications and counter argument relating to the design principle

## Unified Modeling Language (UML)

The Unified Modeling Language or UML is is a mostly graphical modelling language that is used to express designs. It is a standardized language in which to specify the artefacts and components of a software system. It is important to understand that the UML describes a notation and not a process. It does not put forth a single method or process of design, but rather is a standardized tool that can be used in a design process.

## State Diagram

The state diagram shows the change of an object through time. Based upon events that occur, the state diagram shows how the object changes from start to finish.




States are represented as a rounded rectangle with the name of the state shown. Optionally you can include an activity that represents a longer running task during that state. Connecting states together are transitions. These represent the events that cause the object to change from one state to another. The guard clause of the label is again mutually exclusive and must resolve itself to be either true or false. Actions represent tasks that run causing the transitions.

Actions are different from activities in that actions cannot be interrupted, while an activity can be interrupted by an incoming event. Both ultimately represent an operation on the object being studied. For example, an operation that sets an attribute would be considered an action, while a long calculation might be an activity. The specific separation between the two depends on the object and the system being studied.

## Architectural patterns

Patterns for system architecting are very much in their infancy. They have been introduced into TOGAF essentially to draw them to the attention of the systems architecture community as an emerging important resource, and as a placeholder for hopefully more rigorous descriptions and

references to more plentiful resources in future versions of TOGAF. They have not (as yet) been integrated into TOGAF. However, in the following, we attempt to indicate the potential value to TOGAF, and to which parts of the TOGAF Architecture Development Method (ADM) they might be relevant.

## Background

*A "pattern" has been defined as*: "an idea that has been useful in one practical context and will probably be useful in others" [*Analysis Patterns - Reusable Object Models*]. In TOGAF, patterns are considered to be a way of putting building blocks into context; for example, to describe a re-usable solution to a problem. Building blocks are what you use: patterns can tell you how you use them, when, why, and what trade-offs you have to make in doing so. Patterns offer the promise of helping the architect to identify combinations of Architecture and/or Solution Building Blocks (ABBs/SBBs) that have been proven to deliver effective solutions in the past, and may provide the basis for effective solutions in the future.

## Content of a Pattern

Several different formats are used in the literature for describing patterns, and no single format has achieved widespread acceptance. However, there is broad agreement on the types of things that a pattern should contain. The headings which follow are taken from *Pattern-Oriented Software Architecture: A System of Patterns*. The elements described below will be found in most patterns, even if different headings are used to describe them.

## Name

A meaningful and memorable way to refer to the pattern, typically a single word or short phrase.

## Problem

A description of the problem indicating the intent in applying the pattern - the intended goals and objectives to be reached within the context and forces described below (perhaps with some indication of their priorities).

## Context

The preconditions under which the pattern is applicable - a description of the initial state before the pattern is applied.

## Forces

A description of the relevant forces and constraints, and how they interact/conflict with each other and with the intended goals and objectives. The description should clarify the intricacies of the problem and make explicit the kinds of trade-offs that must be considered. (The need for such trade-offs is typically what makes the problem difficult, and generates the need for the pattern in the first place.) The notion of "forces" equates in many ways to the "qualities" that architects seek to optimize, and the concerns they seek to address, in designing architectures.

*For example*:

- Security, robustness, reliability, fault-tolerance
- Manageability
- Efficiency, performance, throughput, bandwidth requirements, space utilization
- Scalability (incremental growth on-demand)

- Extensibility, evolvability, maintainability
- Modularity, independence, re-usability, openness, composability (plug-and-play), portability
- Completeness and correctness
- Ease-of-construction
- Ease-of-use
- etc....

A description, using text and/or graphics, of how to achieve the intended goals and objectives. The description should identify both the solution's static structure and its dynamic behaviour - the people and computing actors, and their collaborations. The description may include guidelines for implementing the solution. Variants or specializations of the solution may also be described.

## Resulting Context

The post-conditions after the pattern has been applied. Implementing the solution normally requires trade-offs among competing forces. This element describes which forces have been resolved and how, and which remain unresolved. It may also indicate other patterns that may be applicable in the new context. (A pattern may be one step in accomplishing some larger goal.) Any such other patterns will be described in detail under Related Patterns.

## Examples

One or more sample applications of the pattern which illustrate each of the other elements: a specific problem, context, and set of forces; how the pattern is applied; and the resulting context.

## Rationale

An explanation/justification of the pattern as a whole, or of individual components within it, indicating how the pattern actually works, and why - how it resolves the forces to achieve the desired goals and objectives, and why this is "good".

The Solution element of a pattern describes the external structure and behaviour of the solution: the Rationale provides insight into its internal workings.

## Related Patterns

The relationships between this pattern and others.

These may be predecessor patterns, whose resulting contexts correspond to the initial context of this one; or successor patterns, whose initial contexts correspond to the resulting context of this one; or alternative patterns, which describe a different solution to the same problem, but under different forces; or co-dependent patterns, which may/must be applied along with this pattern.

## Known Uses

Known applications of the pattern within existing systems, verifying that the pattern does indeed describe a proven solution to a recurring problem. Known Uses can also serve as Examples.

Patterns may also begin with an Abstract providing an overview of the pattern and indicating the types of problems it addresses. The Abstract may also identify the target audience and what assumptions are made of the reader.

## Low Level Design

The low level design document should contain a listing of the declarations of all the classes, non-member-functions, and class member functions that will be defined during the implementation stage, along with the associations between those classes and any other details of those classes (such as member variables) that are firmly determined by the low level design stage. The low level design document should also describe the classes, function signatures, associations, and any other appropriate details, which will be involved in testing and evaluating the project according to the evaluation plan defined in the project's requirements document.

More importantly, each project's low level design document should provide a narrative describing (and comments in your declaration and definition files should point out) how the high level design is mapped into its detailed low-level design, which is just a step away from the implementation itself. This should be an English description of how you converted the technical diagrams (and text descriptions) found in your high level design into appropriate class and function declarations in your low level design. You should be especially careful to explain how the class roles and their methods were combined in your low level design, and any changes that you decided to make in combining and refining them.

### Description

Control systems elements like Advanced Metering Infrastructure (AMI) networks fully field wireless sensors and controls outside a utility's physical security perimeter, placing them at a high risk of compromise. System attackers have

every opportunity to damage, sniff, spoof, or tamper communications hardware platforms for malicious, hobbyist, or incidental reasons.

This paper demonstrates the relevance of common control systems communications hardware vulnerabilities that lead to direct control systems compromise. The paper describes several enabling vulnerabilities exploitable by an attacker, the design principles that causing them to arise, the economic and electronic design constraints that restrict their defence, and ideas for vulnerability avoidance.

Topics include design induced vulnerabilities such as the extraction and modification of communications device firmware, man-in-the-middle attacks between chips of a communications devices, circumvention of protection measures, bus snooping, and other attacks. Specific examples are identified in this report, ranked by attack feasibility. Each attack was investigated against actual IEEE 802.15.4 radio architectures.

## Embedded System Architecture

Standard wireless embedded implementation technologies such as IEEE 802.15.4 are generally designed to serve specific market needs. Therefore, the market offers components that translate such standards to mass producible designs. Embedded wireless technologies typically, but not always, have relatively low power consumption, component cost, computational power requirements, design cost, and implementation cost.

Commodity variants of components that implement wireless technology generally have higher individual reliability

than custom designs, and a ready and willing engineer talent pool to integrate them. Almost all such components are designed to leverage or integrate with existing mass production components and subcomponents such as microcontrollers, RAM chips, ROM chips, and others.

All of the above is highly desirable. As with all such technologies that have the potential to achieve economy of scale in design and implementation, vulnerability generally follows or surpasses all cost optimizations and design trade-offs unless specifically mitigated. Such optimizations and economies of scale can serve to broaden the impact of overlooked security flaws, turning their advantage into a weakness.

This paper does not attempt to cover all potential aspects for such wireless technology implementations, much less the entire range of implementation issues for a single technology. We present security vulnerabilities for typical components found in specific IEEE 802.15.4 implementations; and abstract them to help translate real-world tactical security vulnerabilities as recognizable design classes requiring consideration for mitigation. This paper does not educate the reader in the many nuances of RF design. For RF design and implementation issues, see individual standards such as and engineering references including, but not limited to. We present an abstraction of monolithic vulnerable aspects of a typical commodity IEEE 802.15.4 platform, the Telos-B development kit.

While the Telos-B is a basic user-programmable development kit, its architecture is close enough to most typical applications to be considered general. This

abstraction is intended to give the reader a repeatable context as a starting point when looking at other platform architectures.

The RF physical, media access, link layer, and sometimes network layers will be offloaded onto an RF component such as the pictured CC2420. Breaking up the design lets designers implement the RF portion of the application with the best possible RF module for the lowest time to market while targeting host applications to the optimal host processor. Most standalone communications modules will be linked to their host processor by a trivial board-level serial bus such as SPI or I2C. In some designs the host processor also contains the RF stack implementation, eliminating the board-level serial bus. Components such as microcontrollers or host processors rarely fully implement the analog portion of an RF module.

Antennae, inbound and outbound amplifiers, RF switches, and various filters are generally integrated separately as their requirements vary widely across potential applications. Due to their application orientation, host processors will have external timing means.

In general external oscillators reduce processor chip cost and allow the designer to scale the system to the cheapest clock source meeting application requirements. Though typically not used, many 802.15.4 RF modules have a means to slave a host's clock to the RF module to further reduce design cost. Power is supplied to the devices as required by the module, though often platform power requirements are aligned to reduce component count and subsequent cost.

### Confidentiality

- Snooping Bus Traffic
- Extracting Firmware for Vulnerability Analysis
- Extracting Stored Information
- Snooping Side Channels

### Integrity

- Tampering Bus Traffic
- Replacing Hardware Components
- Modifying Existing Components
- Bypassing Hardware Components
- Disrupting or Distorting Normal Hardware Operation
- Bypassing Software Components

### Availability

- Jamming or Shrouding
- Alert/Condition Flooding
- Run Battery Down

### PHY, Link Transceiver

The subcomponent that deals with PHY, MAC, and LINK layer issues. Potentially executes link layer cryptography algorithms.

*Key Subcomponents*: Registers, RAM, other storage, boot loader, internal programme storage, internal timing source, and architecture specific functionality

*Key External Dependencies*: RF Front End, NET & App Controller, data bus to NET & App Controller, external timing source, power supply, RF/EM environment, temperature environment

*Potentially Vulnerable to*: DoS, Disruption, Distortion, Spoofing, Snooping, live code injection, serial Bus tampering, reconfiguration, firmware analysis, firmware tampering, snooping side channels, environmental tampering, etc.

## NET & APP Controller

The subcomponent that primarily focuses on executing any higher layer network functionality. This is generally an independent microprocessor or microcontroller that may also run the application.

*Key Subcomponents*: Registers, RAM, other storage, boot loader, internal programme storage, internal timing source, and architecture specific functionality

*Key External Dependencies*: PHY, Link Transceiver, external buses, data bus to the PHY, Link Transceiver, external timing source, power supply, RF/EM environment, temperature environment, external storage

*Potentially Vulnerable to*: DoS, Disruption, Distortion, Spoofing, Snooping, live code injection, serial Bus Tampering, reconfiguration, flash/RAM snooping, flash/RAM tampering, firmware analysis, firmware tampering, snooping side channels, environmental tampering, tampering of external flash, etc.

## Low-Level Document

On PC-class hardware, there are two basic mechanisms for sending rendering commands to the graphics device: PIO/MMIO (see glossary for specific definitions) and DMA. The architecture described in this document is designed around DMA-style hardware, but can easily be extended to accommodate PIO/MMIO-style hardware.

- *Client* is a user-space X11 client which has been linked with various modules to support hardware-dependent direct rendering. Typical modules may include:
  - libGL.so, the standard OpenGL (or Mesa) library with our device and operating system independent acceleration and GLX modifications.
  - libDRI.so, our device-independent, operating-system dependent driver.
  - libHW3D.so, our *device-dependent* driver.
- *X server* is a user-space X server which has been modified with *device and operating-system independent* code to support DRI. It may be linked with other modules to support hardware-dependent direct rendering.

  *Typical modules may include*:
  - libDRI.so, our *device-independent, operating-system dependent* driver.
  - libH2D.so, our *device-dependent* driver. This library may provide hardware-specific 2D rendering, and 3D initialization and finalization routines that are not required by the client.
- *Kernel Driver* is a kernel-level device driver that performs the bulk of the DMA operations and provides interfaces for synchronization. [Note: Although the driver functionality is hardware-dependent, the actual implementation of the driver may be done in a generic fashion, allowing all of the hardware-specific details to be abstracted into libH3D.so for loading into the Kernel Driver at DRI initialization time. An

implementation of this type is desirable since the Kernel Driver will not then have to be updated for each new graphics device. The details of this implementation are discussed in an accompanying document, but are mentioned here to avoid later confusion.]

- PROTO is the standard X protocol transport layer (*e.g.*, a named pipe for a local client).
- SAREA is a special shared-memory area that we will implement as part of the DRI. This area will be used to communicate information from the X server to the client, and may also be used to share state information with the kernel. This area should not be confused with DMA buffers. This abstraction may be implemented as several different physical areas.
- DMA BUFFERS are memory areas used to buffer graphics device commands which will be sent to the hardware via DMA. These areas are not needed if memory-mapped IO (MMIO) is used exclusively to access the hardware.
- IOCTL is a special interface to the kernel device driver. Requests can be initiated by the user-space programme, and information can be transfered to and from the kernel. This interface incurs the overhead of a system call and memory copy for the information transfered. This abstract interface also includes the ability of the kernel to signal a listening user-space application (*e.g.*, the X server) via I/O on a device (which may, for example, signal the user-space application with the SIGIO signal).

- MMIO is direct memory-mapped access to the graphics device.

## Initialization Analysis

The X server is the first application to run that is involved with direct rendering. After initializing its own resources, it starts the kernel device driver and waits for clients to connect. Then, when a direct rendering client connects, SAREA is created, the XFree86-GLX protocol is established, and other direct rendering resources are allocated. This section describes the operations necessary to bring the system to a steady state.

## X Server Initialization

When the X server is started, several resources in both the X server and the kernel must be initialized if the GLX module is loaded. Obviously, before the X server can do anything with the 3D graphics device, it will load the GLX module if it is specified in the XFree86 configuration file. When the GLX module (which contains the GLX protocol decoding and event handling routines) is loaded, the device-independent DRI module will also be loaded. The DRI module will then call the graphics device-dependent module (containing both the 2D code and the 3D initialization code) to handle the resource allocation outlined below.

## X Resource Allocation Initialization

Several global X resources need to be allocated to handle the client's 3D rendering requests. These resources include the frame buffer, texture memory, other ancillary buffers, display list space, and the SAREA.

## Frame 3Buffer

*There are several approaches to allocating buffers in the frame buffer*: static, static with dynamic reallocation of the unused space, and fully dynamic. Static buffer allocation is the approach we are adopting in the sample implementation for several reasons that will be outlined below.

*Static allocation*. During initialization, the resources supported by the graphics device are statically allocated. For example, if the device supports front, back and depth buffers in the frame buffer, then the frame buffer is divided into four areas. The first three are equal in size to the visible display area and are used for the three buffers (front, back and depth). The remaining frame buffer space remains unallocated and can be used for hardware cursor, font and pixmap caches, textures, pbuffers, etc.

## Texture memory

Texture memory is shared among all 3D rendering clients. On some types of graphics devices, it can be shared with other buffers, provided that these other buffers can be "kicked out" of the memory. On other devices, there is dedicated texture memory, which might or might not be sharable with other resources. Since memory is a limited resource, it would be best if we could provide a mechanism to limit the memory reserved for textures. However, the format of texture memory on certain graphics devices is organized differently (banked, tiled, etc.) than the simple linear addressing used for most frame buffers. Therefore, the "size" of texture memory is device-dependent. This complicates the issue of using a single number for the size of texture memory.

Another complication is that once the X server reports that a texture will fit in the graphics device memory, it must continue to fit for the life of the client (*i.e.*, the total texture memory for a client can never get smaller). Therefore, at initialization time, the maximum texture size and total texture memory available will need to be determined by the device-dependent driver. This driver will also provide a mechanism to determine if a set of textures will fit into texture memory.

### Other Ancillary Buffers

All buffers associated with a window (*e.g.*, back, depth, and GID) are preallocated by the static frame-buffer allocation. Pixmap, pbuffers and other ancillary buffers are allocated out of the memory left after this static allocation.

During X server initialization, the size off-screen memory available for these buffers will be calculated by the device-dependent driver. Note that pbuffers can be "kicked out" (at least the old style could), and so they don't require virtualization like pixmaps and potentially the new style pbuffers.

### Display Lists

For graphics devices that support display lists, the display list memory can be managed in the same way as texture memory. Otherwise, display lists will be held in the client virtual-address space.

### SAREA

The SAREA is shared between the clients, the X server, and the kernel. It contains four segments that need to be shared: a per-device global hardware lock, per-context information, per-drawable information, and saved device state information.

- *Hardware lock segment*. Only one process can access the graphics device at a time. For atomic operations that require multiple accesses, a global hardware lock for each graphics device is required. Since the number of cards is known at server initialization time, the size of this segment is fixed.
- *Per-context segment*. Each GLXContext is associated with a particular drawable in the per-drawable segment and a particular graphics device state in the saved device state segment. Two pointers, one to the drawable that the GLXContext is currently bound and one to the saved device state is stored in the per-context segment. Since the number of GLXContexts is not known at server start up time, the size of this segment will need to grow. It is a reasonable assumption to limit the number of direct rendering contexts so the size of this segment can be fixed to a maximum. The X server is the only process that writes to this segment and it must maintain a list of available context slots that needs to be allocated and initialized.
- *Per-drawable segment*. Each drawable has certain information that needs to be shared between the X server and the direct rendering client:
  - Buffer identification (*e.g.*, front/back buffer) (int32)
  - Window information changed ID
  - Flags (int32)

The window information changed ID signifies that the user has either moved, unmapped or resized the window, or the clipping information has changed and needs to be

communicated to the client via the XFree86-GLX protocol. Since OpenGL clients can create an arbitrary number of GLXDrawables, the size of this segment will need to grow. As with the per-context segment, the size of this segment can be limited to a fixed maximum. Again, the X server is the only process that writes to this segment, and it must maintain a list of available drawable slots that needs to be allocated and initialized.

- *Saved device state segment*. Each GLXContext needs to save the graphics hardware context when another GLXContext has ownership of the graphics device. This information is fixed in size for each graphics device, but will be allocated as needed because it can be quite large. In addition, if the graphics device can read/write its state information via DMA, this segment will need to be locked down during the request.

## Kernel Initialization

When the X server opens the kernel device driver, the kernel loads and initializes the driver. See the next section for more details of the kernel device driver.

## Double Buffer Optimizations

*There are typically three approaches to hardware double buffering*:

1. *Video Page Flipping*: The video logic is updated to refresh from a different page. This can happen very quickly with no per pixel copying required. This forces the entire screen region to be swapped.
2. *Bitblt Double Buffering*: The back buffer is stored in offscreen memory and specific regions of the screen

can be swapped by coping data from the offscreen to onscreen. This has a performance penality because of the overhead of copying the swapped data, but allows for fine grain independent control for multiple windows.

2. *Auxillary Per Pixel Control*: An additional layer contains information on a per pixel basis that is used to determine which buffer should be displayed. Swapping entire regions is much quicker than Bitblt Double Buffering and fine grain independed control for multiple windows is achieved. However, not all hardware or modes support this method.

If the hardware support Auxillary Per Pixel Control for the given mode, then that is the preferred method for double buffer support. However, if the hardware doesn't support Auxillary Per Pixel Control, then the following combined opproach to Video Page Flipping and Bitblt Double Buffering is a potential optimization.

- Initialize in a *Bitblt Double Buffering* mode. This allows for X Server performance to be optimized while not double buffering is required.
- Transition to a *Video Page Flipping* mode for the first window requiring double buffer support. This allows for the fastest possible double buffer swapping at the expense of requiring the X Server to render to both buffers. Note, for the transition, the contents of the front buffer will need to be copied to the back buffer and all further rendering will need to be duplicated in both buffers for all non-double buffered regions while in this mode.

- Transition back to *Bitblt Double Buffering* mode when additional double buffering windows are created. This will sacrifice performance for the sake of visual accuracy. Now all windows can be independently swapped.

In the initial SI, only the Bitblt Double Buffering mode will be implemented.

### Kernel Driver Initialization

When the kernel device driver is opened by the X server, the device driver might not be loaded. If not, the module is loaded by kerneld and the initialization routine is called. In either case, the open routine is then called and finishes initializing the driver.

### Kernel DMA Initialization

Since the 3D graphics device drivers use DMA to communicate with the graphics device, we need to initialize the kernel device driver that will handle these requests. The kernel, in response to this request from the X server, allocates the DMA buffers that will be made available to direct rendering clients.

### Kernel Interrupt Handling Initialization

Interrupts are generated in a number of situations including when a DMA buffer has been processed by the graphics device. To acknowledge the interrupt, the driver must know which register to set and to what value to set it. This information could be hard coded into the driver, or possibly a generic interface might be able to be written. If this is possible, the X server must provide information to

the kernel as to how to respond to interrupts from the graphics device.

### Hardware Context Switching

Since the kernel device driver must be able to handle multiple 3D clients each with a different GLXContext, there must be a way to save and restore the hardware graphics context for each GLXContext when switching between them. Space for these contexts will need to be allocated when they are created byglXCreateContext(). If the client can use this hardware context (*e.g.*, for software fallbacks or window moves), this information might be stored in the SAREA.

### Client DMA wait Queues

Each direct rendering context will require a DMA wait queue from which its DMA buffers can be dispatched. These wait queues are allocated by the X server when a new GLXContext is created (glXCreateContext()).

### Client Initialization

This section examines what happens before the client enters steady state behaviour. The basic sequence for direct-rendering client initialization is that the GL/GLX library is loaded, queries to the X server are made (*e.g.*, to determine the visuals/FBConfigs available and if direct rendering can be used), drawables and GLXContexts are created, and finally a GLXContext is associated with a drawable. This sequence assumes that the X server has already initialized the kernel device driver and has pre-allocated any static buffers requested by the user at server startup (as described above).

## Library Loading

When a client is loaded, the GL/GLX library will automatically be loaded by the operating system, but the graphics device-specific module cannot be loaded until after the X server has informed the DRI module which driver to load (see below). The DRI module might not be loaded until after a direct rendering GLXContext has been requested.

## Client Configuration Queries

During client initialization code, several configuration queries are commonly made. GLX has queries for its version number and a list of supported extensions. These requests are made through the standard GLX protocol stream. Since the set of supported extensions is device-dependent, similar queries in the device-dependent driver interface (in the X server) are provided that can be called by device-independent code in GLX.

One of the required GLX queries from the client is for the list of supported extended visuals (and FBConfigs in GLX 1.3). The visuals define the types of colour and ancillary buffers that are available and are device-dependent. The X server must provide the list of supported visuals (and FBConfigs) via the standard protocol transport layer (*e.g.*, Unix domain or TCP/IP sockets). Again, similar interfaces in the device-dependent driver are provided that can be called by the device-independent code in GLX. All of this information is known at server initialization time (above).

## Drawable creation

The client chooses the visual (or FBConfig) it needs and creates a drawable using the selected visual. If the drawable

is a window, then, since we use a static resource allocation approach, the buffers are already allocated, and no additional frame buffer allocations are necessary at this time. However, if a dynamic resource allocation approach is added in the future, the buffers requested will need to be allocated.

Not all buffers need to be pre-allocated. For example, accumulation buffers can be emulated in software and might not be pre-allocated. If they are not, then, when the extended visual or FBConfig is associated with the drawable, the client library will need to allocate the accumulation buffer. In GLX 1.3, this can happen withglXCreateWindow(). For earlier versions of GLX, this will happen when a context is made current (below).

## Pixmaps and Buffers

GLXPixmaps are created from an ordinary X11 pixmap, which is then passed to glXCreatePixmap(). GLXPbuffers are created directly by a GLX command. Since we are using a static allocation scheme, we know what ancillary buffers need to be created for these drawables. In the initial SI, these will be handled by indirect rendering or software fallbacks.

## GLXContext creation

The client must also create at least one GLXContext. The last flag to glXCreateContext() is a flag to request direct rendering. The first GLXContext created can trigger the library to initialize the direct rendering interface for this client. Several steps are required to setup the DRI. First, the DRI library is loaded and initialized in the client and X server. The DRI library establishes the private communication mechanism between the client and X server (the XFree86-

GLX protocol). The X server sends the SAREA shared memory segment ID to the client via this protocol and the client attaches to it. Next, the X server sends the device-dependent client side 3D graphics device driver module name to client via the XFree86-GLX protocol, which is loaded and initialized in the client.

The X server calls the kernel module to create a new WaitQueue and hardware graphics context corresponding to the new GLXContext. Finally, the client opens and initializes the kernel driver (including a request for DMA buffers).

## Making a GLXContext current

The last stage before entering the steady state behaviour occurs when a GLXContext is associated with a GLXDrawable by making the context "current". This must occur before any 3D rendering can begin. The first time a GLXDrawable is bound to a direct rendering GLXContext it is registered with the X server and any buffers not already allocated are now allocated. If the GLXDrawable is a window that has not been mapped yet, then the buffers associated with the window are initialized to size zero. When a window is mapped, space in the pre-allocated static buffers are initialized, or in the case of dynamic allocation, buffers are allocated from the available offscreen area (if possible).

For GLX 1.2 (and older versions), some ancillary buffers (*e.g.*, stencil or accumulation), that are not supported by the graphics device, or unavailable due to either resource constraints or their being turned off through X server config options (see above), might need to be allocated.

At this point, the client can enter the steady-state by making OpenGL calls.

### Steady-state Analysis

The initial steady-state analysis presented here assumes that the client(s) and X server have been started and have established all necessary communication channels (*e.g.*, the X, GLX and XFree86-GLX protocol streams and the SAREA segment). In the following analysis, we will impose simplifying assumptions to help direct the analysis towards the main line rendering case. We will then relax our initial assumptions and describe increasingly general cases.

### Single 3D Client (1 GLXContext, 1 GLXWindow), X Server Inactive

Assume: No X server activity (including hardware cursor movement). This is the optimized main line rendering case. The primary goal is to generate graphics device specific commands and stuff them in a DMA buffer as fast as possible. Since the X server is completely inactive, any overhead due to locking should be minimized.

### Processing rendering requests

In the simplest case, rendering commands can be sent to the graphics device by putting them in a DMA buffer. Once a DMA buffer is full and needs to be dispatched to the graphics device, the buffer can be handed immediately to the kernel via an ioctl.

The kernel then schedules the DMA command buffer to be sent to the graphics device. If the graphics device is not busy (or the DMA input queue is not full), it can be

immediately sent to the graphics device. Otherwise, it is put on the WaitQueue for the current context.

In hardware that can only process a single DMA buffer at a time, when the DMA buffer has finished processing, an IRQ is generated by the graphics device and handled by the kernel driver.

In hardware that has a DMA input FIFO, IRQs can be generated after each buffer, after the input FIFO is empty or (in certain hardware) when a low-water mark has been reached. For both types of hardware, the kernel device driver resets the IRQ and schedules the next DMA buffer(s).

A further optimization for graphics devices that have input FIFOs for DMA requests is that if the FIFO is not full, the DMA request could be initiated directly from client space.

## Synchronization

GLX has commands to synchronize direct rendering with indirect rendering or with ordinary X11 operations. These include glFlush(), glFinish(), glXWaitGL() and glXWaitX() synchronization primitives. The kernel driver provides several ioctls to handle each of the synchronization cases. In the simplest case (glFlush()), any partially filled DMA buffer will be sent to the kernel.

Since these will eventually be processed by the hardware, the function call can return. WithglFinish(), in addition to sending any partially filled DMA buffer to the kernel, the kernel will block the client process until all outstanding DMA requests have been completely processed by the graphics device. glXWaitGL() can be implemented using glFlush(), glXWaitX() can be implemented with XSync().

## Buffer Swaps

Buffers swaps can be initiated by glXSwapBuffers(). When a client issues this request, any partially filled DMA buffers are sent to the kernel and all outstanding DMA buffers are processed before the buffer swap can take place. All subsequent rendering commands are blocked until the buffer has been swapped, but the client is not blocked and can continue to fill DMA buffers and send them to the kernel.

If multiple threads are rendering to a GLXDrawable, it is the client's responsibility to synchronize the threads. In addition, the idea of the *current* buffer (*e.g.*, front or back) must be shared by all GLXContexts bound to a given drawable. The X double buffer extension must also agree.

## Kernel-driver Buffer Swap Ioctl

When the buffer swap ioctl is called, a special DMA buffer with the swap command is placed into the current GLXContext's WaitQueue. Because of sequentiality of the DMA buffers in the WaitQueue, all DMA buffers behind this are blocked until all DMA buffers in front of this one have been processed. The header information associated with this buffer lets the scheduler know how to handle the request.

*There are three ways to handle the buffer swap*:

1. *No vert sync*: Immediately schedule the buffer swap and allow subsequent DMA buffers in the WaitQueue to be scheduled. With this policy there will be tearing. In the initial SI, we will implement this policy.
2. *Wait for vert sync*: Wait for the vertical retrace IRQ to schedule the buffer swap command and allow subsequent DMA buffers in the WaitQueue to be

scheduled. With this policy, the tearing should be reduced, but there might still be some tearing if a DMA input FIFO is present and relatively full.

3. *No tearing*: Wait for vertical retrace IRQ and all DMA buffers in the input FIFO to be processed before scheduling the buffer swap command. Since the buffer swap is a very fast bitblt operation, no tearing should be present with this policy.

## Software Fallbacks

Not all OpenGL graphics primitives are accelerated in all hardware. For those not supported directly by the graphics device, software fallbacks will be required. Mesa and SGI's OpenGL SI provide a mechanism to implement these fallbacks; however, the hardware graphics context state needs to be translated into the format required by these libraries. The hardware graphics context state can be read from the saved device state segment of SAREA. An implicit glFinish() is issued before the software fallback can be initiated to ensure that the graphics state is up to date before beginning the software fallback. The hardware lock is required to alter any device state.

## Image Transfer Operations

Many image transfer operations are required in the client-side direct rendering library. Initially these will be software routines that read directly from the memory mapped graphics device buffers (*e.g.*, frame buffer and texture buffer). These are device-dependent operations since the format of the transfer might be different, though certain abstractions should be possible (*e.g.*, linear buffers). An optimization is to allow

the client to perform DMA directly to/from the client's address space. Some hardware has support for page table translation and paging. Other hardware will require the ability to lock down pages and have them placed contiguously in physical memory. The X server will need to manage how the frame and other buffers are allocated at the highest level. The layout of these buffers is determined at X server initialization time.

### Texture Management

Each GLXContext appears to own the texture memory. In the present case, there is no contention. In subsequent cases, hardware context switching will take care of texture swapping as well (see below).

For a single context, the image transfer operations described above provides the necessary interfaces to transfer textures and subtextures to/from texture memory.

### Display List Management

Display lists initially will be handled from within the client's virtual address space. For graphics devices that supports display lists, they can be stored and managed the same as texture memory.

### Selection and Feedback

If there is hardware support for selection and feedback, the rendering commands are sent to the graphics pipeline, which returns the requested data to the client. The amount of data can be quite large and are usually delivered to a collection of locked-down pages via DMA. The kernel should provide a mechanism for locking down pages in the client address space to hold the DMA buffer.

## Queries

Queries are handled similarly to selection and feedback, but the data returned are usually much smaller. When a query is made, the hardware graphics context state has to be read. If the GLXContext does not currently own the graphics device, the state can be read from the saved device state segment in SAREA. Otherwise, the graphics pipeline is temporarily stalled, so that the state can be read from the graphics device.

## Events

GLX has a "pbuffer clobbered" event. This can only be generated as a result of reconfiguring a drawable or creating a new one. Since pbuffers will initially be handled by the software, no clobbered events will be generated. However, when they are accelerated, the X server will have to wrap the appropriate routine to determine when the event needs to be generated.

## Single 3D Client (1 GLXContext, 1 GLXWindow), X Server can Draw

*Assume*: X server can draw (*e.g.*, 2D rendering) into other windows, but does not move the 3D window. This is a common case and should be optimized if possible. The only significant different between this case and the previous case, is that we must now lock the hardware before accessing the graphics device directly directly from the client, X server or kernel space.

The goal is to minimize state transitions and potentially avoid a full hardware graphics context switch by allowing the X server to save and restore 3D state around its access for GUI acceleration.

## Hardware Lock

Access to graphics device must be locked, either implicitly or explicitly. Each component of the system requires the hardware lock at some point. For the X server, the hardware lock is required when drawing or modifying any state. It is requested around blocks of 2D rendering, minimizing the potential graphics hardware context switches.

In the 3D client, the hardware lock is required during the software fallbacks (all other graphics device accesses are handled through DMA buffers). The kernel also must request the lock when it needs to send DMA requests to the graphics device. The hardware lock is contained in the *Hardware lock segment* of the SAREA which can be accessed by all system components. A two-tiered locking scheme is used to minimize the process and kernel context switches necessary to grant the lock. The most common case, where a lock is requested by the last process to hold the lock, does not require any context switches. See the accompanying locks.txt file for more information on two-tiered locking (available late February 1999).

## Graphics Hardware Context Switching

In addition to locking the graphics device, a graphics hardware context switch between the client and the X server is required. One possible solution is to perform a full context switch by the kernel (see the "multiple contexts" section below for a full explanation of how a full graphics hardware context switch is handled). However, the X server is a special case since it knows exactly when a context switch is required and what state needs to be saved and restored.

For the X server, the graphics hardware context switch is required only (a) when directly accessing the graphics device and (b) when the access changes the state of the graphics device. When this occurs, the X server can save the graphics device state (either via a DMA request or by reading the registers directly) before it performs its rendering commands and restore the graphics device state after it finishes.

Three examples will help clarify the situations where this type of optimization can be useful. First, using a cfb/mi routine to draw a line only accesses the frame buffer and does not alter any graphics device state. Second, on many vendor's cards changing the position of the hardware cursor does not affect the graphics device state. Third, certain graphics devices have two completely separate pipelines for 2D and 3D commands. If no 2D and 3D state is shared, then they can proceed independently (but usually not simultaneously, so the hardware lock is still required).

## Single 3D Client (1 GLXContext, 1 GLXWindow), X Server Active

*Assume*: X server can move or resize the single 3D window. When the X server moves or resizes the 3D window, the client needs to stop drawing long enough for the X server to change the window, and it also needs to request the new window location, size and clipping information. Current 3D graphics devices can draw using window relative coordinates, though the window offset might not be able to be updated asynchronously (*i.e.*, it might only be possible to update this information between DMA buffers). Since this is an infrequent operation, it should be designed to have minimal impact on the other, higher priority cases.

## X Server Operations

On the X server side, when a window move is performed, several operations must occur. First, the DMA buffers currently being processed by the graphics device must be completely processed before proceeding since they might associated with the old window position (unless the graphics device allows asynchronous window updates). Next, the X server grabs the hardware lock and waits for the graphics device to become quiescent.

It then issues a bitblt to move the window and all of its associated buffers. It updates the window location in all of the contexts associated with the window, and increments the "Window information changed" ID in the SAREA to notify all clients rendering to the window of the change. It can then release the hardware lock.

Since the graphics hardware context has been updated with the new window offset, any outstanding DMA buffers for the context associated with the moved window will have the new window offset and thus will render at the correct screen location. The situation is slightly more complicated with window resizes or changes to the clipping information.

When a window is resized or when the clipping information changes due to another window popping up on top of the 3D window, outstanding DMA buffers might draw outside of the new window (if the window was made smaller). If the graphics device supports clipping planes, then this information can be updated in the graphics hardware context between DMA buffers.

However, for devices that only support clipping rectangles, the outstanding DMA requests cannot be altered with the new clipping rects.

*To minimize this effect, the X server can*:

- Flush the DMA buffers in all contexts' WaitQueues associated with the window,
- Wait for these DMA buffers to be processed by the graphics device. However, this does not completely solve the problem as there could be a partially filled DMA buffer in the client(s) rendering to the window (see below).

## 3D Client Operations

On the client side, during each rendering operation, the client checks to see if it has the most current window information. If it does, then it can proceed as normal. However, if the X server has changed the window location, size or clipping information, the client issues a XFree86-DRI protocol request to get the new information.

See the accompanying XFree86-DRI.txt file for more information on the XFree86-DRI protocol implementation. This information will be mainly used for software fallbacks. Since there could be several outstanding requests in the partially filled "current" DMA buffer, the rendering commands already in this buffer might draw outside of the window. The simplest solution to this problem is to send an expose event to the windows that are affected.

*This could be accomplished as follows*:

- Send the partially filled DMA buffer to the kernel,
- Wait for it to be processed,
- Generate a list of screen-relative rectangles for the affected region,
- Send a request to the X server to generate an expose event in the windows that overlap with that region.

On graphics devices that do not allow the window offset to be updated between DMA buffers, the situation described above will also occur for window moves. The "generate expose events" solution also will be used to solve the problem. It is not known at this time if any graphics devices of this type exist.

## Multiple 3D Clients

*Assume*: There are now multiple 3D clients, each of which has their own GLXContext(s). As with the previous case, multiple GLXContexts are actively used in rendering, and this case can be handled the same as the previous one.

## Finalization Analysis

This section examines what happens after exiting steady state behaviour via destroying a rendering surface or context, or via process termination. Process suspension and switching virtual consoles are special cases and are dealt with in this section.

## Destroying a Drawing Surface

If the drawing surface is a window, it can be destroyed by the window manager. When this occurs, the X server must notify the direct rendering client that the window was destroyed. However, before the window can be removed, the X server must wait until all outstanding DMA buffer requests associated with the window have been completely processed in order to avoid rendering to the destroyed window after it has been removed. When the client tries to draw to the window again, it recognizes that the window is no longer valid and cleans up its internal state associated with the

window (*e.g.*, any local ancillary buffer), and returns an error. GLX 1.3 uses glXDestroyWindow() to explicitly notify the system that the window is no longer associated with GLX, and that its resources should be freed.

### Destroying a GLXContext

Since there are limited context slots available in the per-context segment of SAREA, a GLXContext's resources can be freed by calling glXDestroyContext()when it is no longer needed. If the GLXContext is current to any thread, the context cannot be destroyed until it is no longer current. When this happens, the X server marks the GLXContext's per-context slot as free, frees the saved device state, and notifies the kernel that the WaitQueue can be freed.

### Destroying Shared Resources

Texture objects and display lists can be shared by multiple GLXContexts. When a context is destroyed in the share list, the reference count should be decremented. If the reference count of the texture objects and/or display lists is zero, they can be freed as well.

### Process Finalization

When a process exits, its direct rendering resources should be freed and returned to the X server.

### Graceful Termination

If the termination is expected, the resources associated with the process are freed. The kernel reclaims its DMA buffers from the client. The X server frees the GLXDrawables and GLXContexts associated with the client. In the process

of freeing the GLXContexts, the X server notifies the kernel that it should free any WaitQueues associated with the GLXContexts it is freeing. The saved device state is freed. The reference count to the SAREA is decremented. Finally, any additional resources used by the GLX and XFree86-GLX protocol streams are freed.

### Unexpected Termination

Detecting the client death is the hardest part of unexpected process termination. Once detected, the resources are freed as in the graceful termination case outlined above. The kernel detects when a direct rendering client process dies since it has registered itself with the kernel exit procedure. If the client does not hold the hardware lock, then it can proceed as in the graceful termination case. If the hardware lock is held, the lock is broken. The graphics device might be in an unusable state (*e.g.*, waiting for data during a texture upload), and might need to be reset. After reset, the graceful termination case can proceed.

### Process Suspension

Processes can suspend themselves via a signal that cannot be blocked, SIGSTOP. If the process holds the hardware lock during this time, the SIGSTOP signal must be delayed until the lock is freed. This can be handled in the kernel. As an initial approximation, the kernel can turn off SIGSTOP for all direct rendering clients.

### Switching Virtual Consoles

XFree86 has the ability to switch to a different virtual console when the X server is running. This action causes

the X server to draw to a copy of the frame buffer in the X server virtual address space. For direct rendering clients, this solution is not possible. A simple solution to use in the initial SI is to halt all direct access to the graphics device by grabbing the hardware lock.

In addition to switching virtual consoles, XFree86 can be started on multiple consoles (with different displays). Initially, only the first display will support direct rendering.

## Future Enhancements

### MMIO

This architecture has been designed with MMIO based 3D solution in mind, but the initial SI will be optimized for DMA based solutions. A more complete MMIO driven implementation can be added later. Base support in the initial SI that will be useful for an MMIO-only solution is unprivileged mapping of MMIO regions and a fast two-tier lock. Additional optimizations that will be useful are virtualizing the hardware via a page fault mechanism and a mechanism for updating shared library pointers directly.

### Device-specific Kernel Driver

Several optimizations (mentioned above) can be added by allowing a device-specific kernel driver to hook out certain functions in the generic kernel driver.

### Other Enhancements

*We should consider additional enhancements including*:

- Multiple displays and multiple screens
- More complex buffer swapping (cushion buffering, swap every N retraces, synchronous window swapping)

## Glossary

### MMIO

Memory-Mapped Input-Output. In this document, we use the term MMIO to refer to operations that access a region of graphics card memory that has been memory-mapped into the virtual address space, or to operations that access graphics hardware registers via a memory-mapping of the registers into the virtual address space (in contrast to PIO).

Note that graphics hardware "registers" may actually be pseudo-registers that provide access to the hardware FIFO command queue.

### PIO

Programmed Input-Output. In this document, we use the term PIO to refer specifically to operations that *must* use the Intel in and out instructions (or equivalent non-Intel instructions) to access the graphics hardware (in contrast to using memory-mapped graphics hardware registers, which allow for the use ofmov instructions).

# 3

# Software Testing

A primary purpose for testing is to detect software failures so that defects may be uncovered and corrected. The scope of software testing often includes examination of code as well as execution of that code in various environments and conditions as well as examining the quality aspects of code: does it do what it is supposed to do and do what it needs to do. We test software because developers are unable to build defect free software. If the development processes were perfect, meaning no defects were produced, testing would not be necessary. Testing by the individual who developed the work has not proven to be a substitute to building and following a detailed test plan.

*The disadvantages of a person checking their own work using their own documentation are as follows*:

- Misunderstandings will not be detected, because the checker will assume that what the other individual heard from him was correct.

- Improper use of the development process may not be detected because the individual may not understand the process.
- The individual may be "blinded" into accepting erroneous system specifications and coding because he falls into the same trap during testing that led to the introduction of the defect in the first place.
- Information services people are optimistic in their ability to do defect-free work and thus sometimes underestimate the need for extensive testing.
- Without a formal division between development and test, an individual may be tempted to improve the system structure and documentation, rather than allocate that time and effort to the test.

Testing unveils design defects as well as data defects of any product. All testing focuses on discovering and eliminating defects or variances from what is expected.

*Testers need to identify these two types of defects:*

1. *Variance from Specifications*: A defect from the perspective of the builder of the product.
2. *Variance from what is Desired*: A defect from a user (or customer) perspective.

## Background and Objectives

Software testing is an integral and important activity in every software development environment. Software seems to have has permeated almost every equipment that we use in our daily lives. Companies that produce embedded systems for use in health care, transportation, and other critical segments of our society have embraced model based software

testing by integrating them into their development environments.

- Software Testing is designed to establish that the software is working satisfactorily as per the requirements.
- Software Testing is a process designed to prove that the programme is error free.
- Software The job of testing is to certify that the software does its job correctly and can be used in production.

Because, with these as the guidelines, one would tend to operate the system in a normal manner to see if it works and one would unconsciously choose such normal/correct test data as would prevent the system from failing. Besides, it is any way not possible to certify that a software has no errors, simply because it is almost impossible to detect all errors. In a way, we can say that software testing is basically a task of locating errors. From the objective point of view, testing can be done in two ways:

## Positive Testing

Operate application or software as it should be operated. Use proper variety of test data, including data values at boundries to test if it fails.

*Check actual test results with the expected and see*:

- Does it behave normally?
- Are results correct?
- Does the software function correctly?

## Negative Testing

Test for abnormal operations. Test with illegal/ abnormal test data. Intentionally attempt to make things go wrong and to discover/ detect and see

- Does the system fail/ crash?
- Does the programme do what it should not?
- Does it fail to do what it should?

## Positive view of Negative Testing

The job of testing is to discover errors before the user does. A good tester is one who is successfull in making the system fail. Mentality of the tester has to be destructive – opposite to that of the creator/ developer which should be constructive.

This chapter is designed to enable a clear understanding and knowledge of the foundations, techniques, and tools in the area of software testing and its practice in the industry. The course will prepare students to be leaders in software testing. Whether you are a developer or a tester, you must test software. This course is a unique opportunity to learn strengths and weaknesses of a variety of software testing techniques.

Applications of testing techniques in health care industry (*e.g.* pacemaker), nuclear industry (*e.g.* plant control), aerospace industry (*e.g.* Mars Polar Lander), security (*e.g.* smart card), automobile industry (*e.g.* automotive control systems), and others will be considered.

*The chapter will focus on:*

- Test process and continuous quality improvement
- Test generation from requirements
- Modeling techniques: UML: FSM and Statecharts, Combinatorial design; and others.

- Test generation from models.
- Test adequacy assessment.
- Industrial applications.

Discussion oriented lectures by the instructor, in-class group presentations by teams, laboratory exercises using advanced testing tools, and invited talks by experts from the industry will be the primary mechanisms for learning and the dissemination of knowledge.

## Chapter description

Fundamentals of software testing; software test proces and continuous quality improvement; Test generation using finite state models, Combinatorial design, and others; Test adequacy assessment using black box and white box criteria; Industrial applications of model based testing. Students will be required to form small teams of three or four, preferably interdisciplinary, and make presentations to the class.The work of each team will be reviewed by the instructor and other teams.

## unit testing

These type of tests are usually written by developers as they work on code (white-box style), to ensure that the specific function is working as expected. One function might have multiple tests, to catch corner cases or other branches in the code. Unit testing alone cannot verify the functionality of a piece of software, but rather is used to assure that the building blocks the software uses work independently of each other.

**testing in software**

Unit testing is a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinized for proper operation.This testing mode is a component of software development that takes a meticulous approach to building a product by means of continual testing and revision.

Once all of the units in a programme have been found to be working in the most efficient and error-free manner possible, larger components of the programme can be evaluated by means of integration testing. Unit testing must be done with an awareness that it may not be possible to test a unit for every input scenario that will occur when the programme is run in a real-world environment.

## Rules

- Write the test first
- Never write a test that succeeds the first time
- Start with the null case, or something that doesn't work
- Don't be afraid of doing something trivial to make the test work
- Loose coupling and testability go hand in hand
- Use mock objects. A mock object is an object that pretends to be a particular type, but is really just a sink, recording the methods that have been called on it
- A test is not a pure unit test if: It talks to the database
  - It communicates across the network
  - It touches the file system

- It can't run at the same time as any of your other unit tests
- You have to do special things to your environment (such as editing config files) to run it. Tests that do these things should be kept aside from the regular unit test suit to run the test cases faster whenever we make changes.

Unit testing deals with testing a unit as a whole. This would test the interaction of many functions but confine the test within one unit. The exact scope of a unit is left to interpretation. Supporting test code, sometimes called scaffolding, may be necessary to support an individual test. This type of testing is driven by the architecture and implementation teams. This focus is also called black-box testing because only the details of the interface are visible to the test. Limits that are global to a unit are tested here.

In the construction industry, scaffolding is a temporary, easy to assemble and disassemble, frame placed around a building to facilitate the construction of the building. The construction workers first build the scaffolding and then the building. Later the scaffolding is removed, exposing the completed building. Similarly, in software testing, one particular test may need some supporting software.

This software establishes an environment around the test. Only when this environment is established can a correct evaluation of the test take place. The scaffolding software may establish state and values for data structures as well as providing dummy external functions for the test. Different scaffolding software may be needed from one test to another test. Scaffolding software rarely is considered part of the

system. Sometimes the scaffolding software becomes larger than the system software being tested. Usually the scaffolding software is not of the same quality as the system software and frequently is quite fragile. A small change in the test may lead to much larger changes in the scaffolding.

Internal and unit testing can be automated with the help of coverage tools. A coverage tool analyses the source code and generates a test that will execute every alternative thread of execution. It is still up to the programmer to combine these test into meaningful cases to validate the result of each thread of execution. Typically, the coverage tool is used in a slightly different way.

First the coverage tool is used to augment the source by placing informational prints after each line of code. Then the testing suite is executed generating an audit trail. This audit trail is analysed and reports the per cent of the total system code executed during the test suite. If the coverage is high and the untested source lines are of low impact to the system's overall quality, then no more additional tests are required.

The idea behind unit testing is elegant and simple, but can be expanded to enable sophisticated series of tests for code validation and regression testing. A unit test is strictly something that 'exercises' or runs the code under test. Many developers manually perform unit testing on a regular basis in the course of working on a segment of code. In other words, it can be as simple as *'I know the code should perform this task when I supply this input; I'll try it and see what happens.'* If it doesn't behave as expected, the developer would likely modify the code and repeat this iterative process

until it works. The problem with doing this manually is that it can easily overlook large ranges of values or different combinations of inputs and it offers no insight into how much of the code was actually executed during testing. Additionally, it does not help us with the important task of proving to someone else that it worked and that it worked *correctly*.

The cost and time required is compounded by the reality that one round of testing is rarely enough; besides fixing bugs, any changes that are made to code later in the development process may require additional investment of time and resources to ensure it's working properly.

Large projects typically augment manual procedures with tools such as the Framework to automate and improve this process. Automation mitigates risk of undetected errors, saves costs by detecting problems early, and saves time by keeping developers focused on the task of writing the software, instead of performing the tests themselves.

The idea behind unit testing is that once you have a unit that you think works, you set up a test case where you specify some input to the unit and compare the result of your unit with your expected result. You know about the expected result, because you know what your unit is doing (or you should know it).

Well, you better know what your unit is supposed to do, or else you should not do programming in the first place...;) Then you run the tests and the CakePHP/«insert your framework here» test suite tells you if they passed or if not, with some graceful message where the error occurred.

Now you have that testcase. Now you add testcases for t h a t input and this one. The advantage of this is that once you

wrote the tests down they are there (cool, huh?) and you can hold on to them. There is no need anymore for you to open the browser and test everything manually again when you change your system. Instead, you add functionality, run the automated unit tests again, if they pass you are good to go, if they don't pass you broke something. Well, what if you broke something but your tests don't catch it? That's something that UT cannot do for you. You must make sure you have a good test coverage

Typically, the order of the running of the tests should not matter. There might be special cases, but in well over 90% it does not. This should also be your goal, too, to have two different problems if two test cases fail. Keep them all isolated and you will sleep well. For most tests there is also not much configuration to be done. You specify the input, your expected result, crank the handle and evaluate how well you have done. You should typically be able to group tests together, too. When you run these groups you can get a good overview over large components of your system.

## Testing Phase

The first test in the development process is the unit test. The source code is normally divided into modules, which in turn are divided into smaller units called units. These units have specific behaviour. The test done on these units of code is called unit test. Unit test depends upon the language on which the project is developed. Unit tests ensure that each unique path of the project performs accurately to the documented specifications and contains clearly defined inputs and expected results.

**Table. The Testing Phase: Improve Quality.**

| Phase | Deliverable |
|---|---|
| Testing | Regression Test |
| | Internal Testing |
| | Unit Testing |
| Application Testing | |
| | Stress Testing |

Simply stated, quality is very important. Many companies have not learned that quality is important and deliver more claimed functionality but at a lower quality level. It is much easier to explain to a customer why there is a missing feature than to explain to a customer why the product lacks quality. A customer satisfied with the quality of a product will remain loyal and wait for new functionality in the next version. Quality is a distinguishing attribute of a system indicating the degree of excellence.

In many software engineering methodologies, the testing phase is a separate phase which is performed by a different team after the implementation is completed. There is merit in this approach; it is hard to see one's own mistakes, and a fresh eye can discover obvious errors much faster than the person who has read and re-read the material many times. Unfortunately, delegating testing to another team leads to a slack attitude regarding quality by the implementation team. Alternatively, another approach is to delegate testing to the the whole organization. If the teams are to be known as craftsmen, then the teams should be responsible for establishing high quality across all phases. Sometimes, an attitude change must take place to guarantee quality.

Regardless if testing is done after-the-fact or continuously, testing is usually based on a regression technique split into several major focuses, namely internal, unit,application, and stress.The testing technique is from the perspective of the system provider.

Because it is nearly impossible to duplicate every possible customer's environment and because systems are released with yet-to-be-discovered errors, the customer plays an important, though reluctant, role in testing. As will be established later in the thesis, in the Water Sluice methodology this is accomplished in the alpha and beta release of the system.

## Uses

Forget for a moment that there is something called XP (Extreme Programming) that coined the Unit Test term. The most of the projects developed today are always under tight development schedules and usually have only its developers as the tester of their code. By writing the unit tests themselves they can have a head start towards bug-free and quality code.

One will argue that if the developer is writing all the unit tests, it is quite possible to get the set of unit tests that are passable, because these unit tests are developed based either on the foreknowledge of application code or the assumptions made in the application code. However, do not be fooled with this, imagine what will happen if developer decides to change the application, her old test cases will break. That will force her to either re-think her changes or re-write the unit tests.

The application architect or analyst can write all the unit test cases upfront (Not what XP recommend, but we are not worried about it) and test the developed code against these

cases and functionalities. The advantage is well defined deliverable for the developer and more quantifiable progress. A developer can also use this to disciple their work habits *e.g.* she can write a set of unit test that she wants to accomplish in a days work. Once tests ready, she can start developing the application and check her progress against the unit test. Now she has a metre to check her progress.

## NUnit Framework

NUnit framework is port of JUnit framework from java and Extreme Programming (XP). This is an open source product. You can download it from http://www.nunit.org. The NUnit framework is developed from ground up to make use of.NET framework functionalities. It uses an Attribute based programming model. It loads test assemblies in separate application domain hence we can test an application without restarting the NUnit test tools. The NUnit further watches a file/assembly change events and reload it as soon as they are changed. With these features in hand a developer can perform develop and test cycles sides by side.

*Before we dig deeper, we should understand what NUnit Framework is not:*

- It is not Automated GUI tester.
- It is not a scripting language, all test are written in.NET supported language *e.g.* C#, VC, VB.NET, J# etc.
- It is not a benchmark tool.
- Passing the entire unit test suite does not mean software is production ready.

# 4

# Software Requirements Specification in Engineering Process

There are many good definitions of System and Software Requirements Specifications that will provide us a good basis upon which we can both define a great specification and help us identify deficiencies in our past efforts. There is also a lot of great stuff on the web about writing good specifications. The problem is not lack of knowledge about how to create a correctly formatted specification or even what should go into the specification. The problem is that we don't follow the definitions out there.

We have to keep in mind that the goal is not to create great specifications but to create great products and great software. Can you create a great product without a great specification? Absolutely! You can also make your first million through the lottery – but why take your chances? Systems and software these days are so complex that to embark on the design before

knowing what you are going to build is foolish and risky. The IEEE is an excellent source for definitions of System and Software Specifications. As designers of real-time, embedded system software, we use IEEE STD 830-1998 as the basis for all of our Software Specifications unless specifically requested by our clients. Essential to having a great Software Specification is having a great System Specification. The equivalent IEEE standard for that is IEEE STD 1233-1998. However, for most purposes in smaller systems, the same templates can be used for both.

## Benefits of SRS

Establish the basis for agreement between the customers and the suppliers on what the software product is to do. The complete description of the functions to be performed by the software specified in the SRS will assist the potential users to determine if the software specified meets their needs or how the software must be modified to meet their needs

Reduce the development effort. The preparation of the SRS forces the various concerned groups in the customer's organization to consider rigorously all of the requirements before design begins and reduces later redesign, recoding, and retesting. Careful review of the requirements in the SRS can reveal omissions, misunderstandings, and inconsistencies early in the development cycle when these problems are easier to correct.

Provide a basis for estimating costs and schedules. The description of the product to be developed as given in the SRS is a realistic basis for estimating project costs and can be used to obtain approval for bids or price estimates. Provide a baseline for validation and verification. Organizations can

develop their validation and Verification plans much more productively from a good SRS. As a part of the development contract, the SRS provides a baseline against which compliance can be measured.

Facilitate transfer. The SRS makes it easier to transfer the software product to new users or new machines. Customers thus find it easier to transfer the software to other parts of their organization, and suppliers find it easier to transfer it to new customers.

Serve as a basis for enhancement. Because the SRS discusses the product but not the project that developed it, the SRS serves as a basis for later enhancement of the finished product. The SRS may need to be altered, but it does provide a foundation for continued production evaluation.

## Characteristics

*An SRS should be*:

- Correct
- Unambiguous
- Complete
- Consistent
- Ranked for importance and/or stability
- Verifiable
- Modifiable
- Traceable
  - *Correct*: This is like motherhood and apple pie. Of course you want the specification to be correct. No one writes a specification that they know is incorrect. We like to say - "Correct and Ever Correcting." The discipline is keeping the

specification up to date when you find things that are not correct.

- *Unambiguous*: An SRS is unambiguous if, and only if, every requirement stated therein has only one interpretation. Again, easier said than done. Spending time on this area prior to releasing the SRS can be a waste of time. But as you find ambiguities - fix them.
- *Complete*: A simple judge of this is that is should be all that is needed by the software designers to create the software.
- *Consistent*: The SRS should be consistent within itself and consistent to its reference documents. If you call an input "Start and Stop" in one place, don't call it "Start/Stop" in another.
- *Ranked for Importance*: Very often a new system has requirements that are really marketing wish lists. Some may not be achievable. It is useful provide this information in the SRS.
- *Verifiable*: Don't put in requirements like - "It should provide the user a fast response." Another of my favorites is - "The system should never crash." Instead, provide a quantitative requirement like: "Every key stroke should provide a user response within 100 milliseconds."

## System and Specification

*Important issues are not defined up front and Mechanical, Electronic and Software designers do not really know what their requirements are*:

- Define the functions of the system
- Define the Hardware/ Software Functional Partitioning
- Define the Performance Specification
- Define the Hardware/ Software Performance Partitioning
- Define Safety Requirements
- Define the User Interface (A good user's manual is often an overlooked part of the System specification. Many of our customers haven't even considered that this is the right time to write the user's manual.)
- Provide Installation Drawings/Instructions.
- Provide Interface Control Drawings (ICD's, External I/O)

One job of the System specification is to define the full functionality of the system. In many systems we work on, some functionality is performed in hardware and some in software. It is the job of the System specification to define the full functionality and like the performance requirements, to set in motion the trade-offs and preliminary design studies to allocate these functions to the different disciplines (mechanical, electrical, software).

Another function of the System specification is to specify performance. For example, if the System is required to move a mechanism to a particular position accurate to a repeatability of ± 1 millimeter, that is a System's requirement. Some portion of that repeatability specification will belong to the mechanical hardware, some to the servo amplifier and electronics and some to the software. It is the job of the System specification to provide that requirement and to set

in motion the partitioning between mechanical hardware, electronics, and software. Very often the System specification will leave this partitioning until later when you learn more about the system and certain factors are traded off (For example, if we do this in software we would need to run the processor clock at 40 mHz.

However, if we did this function in hardware, we could run the processor clock at 12 mHz). However, for all practical purposes, most of the systems we are involved with in small to medium size companies, combine the software and the systems documents. This is done primarily because most of the complexity is in the software. When the hardware is used to meet a functional requirement, it often is something that the software wants to be well documented.

Very often, the software is called upon to meet the system requirement with the hardware you have. Very often, there is not a systems department to drive the project and the software engineers become the systems engineers. For small projects, this is workable even if not ideal. In this case, the specification should make clear which requirements are software, which are hardware, and which are mechanical.

## Design and Requirement

SRS should not include any design requirements. However, this is a difficult discipline. For example, because of the partitioning and the particular RTOS you are using, and the particular hardware you are using, you may require that no task use more than 1 ms of processing prior to releasing control back to the RTOS.

Although that may be a true requirement and it involves software and should be tested – it is truly a design requirement and should be included in the Software Design Document or in the Source code. Consider the target audience for each specification to identify what goes into what documents.

### Marketing/Product Management

Creates a product specification and gives it to Systems. It should define everything Systems needs to specify the product

### Systems/Software

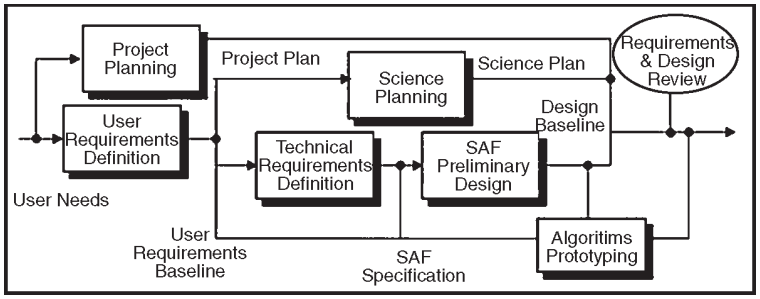
Creates a Software Specification and gives it to Software. It should define everything Software needs to develop the software. Thus, the SRS should define everything explicitly or (preferably) by reference that software needs to develop the software. References should include the version number of the target document. Also, consider using master document tools which allow you to include other documents and easily access the full requirements.

## Requirement Engineering Process

Based on assessed user needs, the SAF User Requirements are established and implemented into a Technical Specification and Design baseline, in line with scientific assessments and plans.

Software requirements engineering is the process of determining what is to be produced in a software system. In developing a complex software system, the requirements engineering process has the widely recognized goal of

determining the needs for, and the intended external behaviour, of a system design.




This process is regarded as one of the most important parts of building a software system: " The hardest single part of building a software system is deciding what to build. No other part of the conceptual work is as difficult a establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems.

Tracing the emergence of significant ideas in software development over the years, one can observe that in the '60s the attention was on coding, in the '70s on design and in the '80s on specification. However, in the process of requirements engineering it is often difficult to state the real 'what' level of a system because one person's 'how' may be another person's 'what' and conversely. In this perspective, the requirements engineer faces a complex problem, in meeting the needs of the customer and at the same time meeting the needs of the designer.

*The four specific steps in software requirements engineering are*:

1. Requirements elicitation
2. Requirements analysis
3. Requirements specification
4. Requirements validation

Although they seem to be separate tasks, these four processes cannot be strictly separated and performed sequentially. Some of the requirements are implicit in the working practices, while others may only arise when design solutions are proposed.

## Inquiry based requirements

The Inquiry-Based Requirements Analysis Model views the analysis process as essentially inquiry-based "a series of questions and answers designed to pinpoint where information needs come from and when". The Inquiry Cycle Model, a "formal structure for describing discussion about requirements", addresses the case of mass-market-driven product development, for which there may be no clear customer authority.

The term used in this model is "stakeholder", anyone who shares information about the system, its implementation constraints or problem domain. The model consists of an integration of three phases, where the stakeholders write down their proposed requirements, challenge them by attaching typed annotations and then refine the requirements when change requests are approved.

These requirements are derived from many sources and in many formats, hence a tremendous amount of complex raw data comprise the source material for a given system."AMORE is interested in modeling those vast amounts of raw source material as requirements, and provides access to knowledge about the problem domain, as well as tools for the capture, modeling, analysis and manipulation of raw requirements data.

## User-development interaction

The most important aspect of user-development interaction is the mutual learning and cooperation among them. Some methodologies assume that the transfer of knowledge between users and designers can be achieved in the environment of a meeting room. At the same time, other methodologies (*e.g.* PD) foster the full collaboration of stakeholders through a process where users are directly faced with the designers' work situation and conversely, and by the end of the elicitation process everyone learned about real needs of users and technical possibilities.

In this context, success in meeting the real needs of the software system is contingent upon the ability of users to clearly specify what their requirements are. For this reason, requirements definition needs close interaction between developers and end-users of the software.

It is critical that requirements engineering tools must support collaborative development of the software requirements negotiation. Requirements definition should be an iterative process where, through reflection and experience, users become familiar with the technology and developers become familiar with the work. For example, scenarios, prototypes or mock-ups which provide the opportunity for the users to "experience" the new technology and for the developers to "experience" the work practice.

## TEAM ROOMS

The groupware system called Tearooms provides an electronic equivalent of a team room for groups that are either co-located or at a distance. More about Tearooms as a Group Kit application may be found in Roseman and Greenberg. It

is implemented using an extended version of the groupware toolkit GroupKit. Facilities offered by the GroupKit's Application Programming Interface are preserved in Team Rooms. This enabled the developers to move the existing GroupKit applications to TeamRooms and rapidly create new ones. It combines the rich applications and interfaces found in the existing real-time groupware applications, providing a persistent work space suitable for both synchronous and asynchronous collaboration. It encapsulates both structured and unstructured work through its applications and also takes into account individual and group work.

Apples are special-purpose applications, designated for more specific needs of a group. Team Rooms supports any type of application which can be constructed in Group Kit, such as meeting tools, drawing tools, text editors, card games and so on. When a user starts up the system, he or she is prompted for a user name and a password. If he is among the work group permitted to use the system, he or she will be connected to the Team Rooms central server.

## Elicitation

Using an elicitation method can help in producing a consistent and complete set of security requirements. However, brainstorming and elicitation methods used for ordinary functional (end-user) requirements usually are not oriented towards security requirements and do not result in a consistent and complete set of security requirements. The resulting system is likely to have fewer security exposures when security requirements are elicited in a systematic way.

## Elicitation Evaluation Criteria

The following are example evaluation criteria that may be useful in selecting an elicitation method, but certainly there are other criteria that you could use. The main point is to use criteria and to have a common understanding of what they mean.

- *Adaptability*: The method can be used to generate requirements in multiple environments. For example, the elicitation method works equally as well with a software product that is near completion as with a project in the planning stages.
- *Computer-aided software engineering (CASE) tool*: The method includes a CASE tool. (The Software Engineering Institute defines a CASE tool as "a computer-based product aimed at supporting one or more software engineering activities within a software development process")
- *Stakeholder acceptance*: The stakeholders are likely to agree to the elicitation method in analyzing their requirements. For example, the method isn't too invasive in a business environment.
- *Easy implementation*: The elicitation method isn't overly complex and can be properly executed easily.
- *Graphical output*: The method produces readily understandable visual artifacts.
- *Quick implementation*: The requirements engineers and stakeholders can fully execute the elicitation method in a reasonable length of time.
- *Shallow learning curve*: The requirements engineers and stakeholders can fully comprehend the elicitation method within a reasonable length of time.

# Data Flow Diagrams

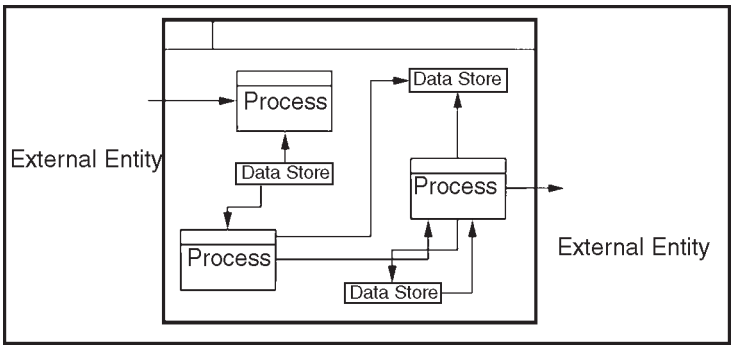
## Introduction

A data flow diagram (DFD) is a significant modeling technique for analyzing and constructing information processes. DFD literally means an illustration that explains the course or movement of information in a process. DFD illustrates this flow of information in a process based on the inputs and outputs.

A DFD can be referred to as a Process Model. Additionally, a DFD can be utilized to visualize data processing or a structured design. A DFD illustrates technical or business processes with the help of the external data stored, the data flowing from a process to another, and the results. A designer usually draws a context-level DFD showing the relationship between the entities inside and outside of a system as one single step. This basic DFD can be then disintegrated to a lower level diagram demonstrating smaller steps exhibiting details of the system that is being modeled.

## Uses

The technique starts with an overall picture of the business and continues by analyzing each of the functional areas of interest. This analysis can be carried out to precisely the level of detail required. The technique exploits a method called top-down expansion to conduct the analysis in a targeted way.




The result is a series of diagrams that represent the business activities in a way that is clear and easy to communicate. A business model comprises one or more data flow diagrams (also known as business process diagrams). Initially a context diagram is drawn, which is a simple representation of the entire system under investigation. This is followed by a level 1 diagram; which provides an overview of the major functional areas of the business. Don't worry about the symbols at this stage, these are explained shortly. Using the context diagram together with additional information from the area of interest, the level 1 diagram can then be drawn.

The level 1 diagram identifies the major business processes at a high level and any of these processes can then be analysed further - giving rise to a corresponding level 2 business process diagram. This process of more detailed analysis can then continue ýÿC through level 3, 4 and so on. However, most investigations will stop at level 2 and it is very unusual to go beyond a level 3 diagram.

Identifying the existing business processes, using a technique like data flow diagrams, is an essential precursor to business process re-engineering, migration to new technology, or refinement of an existing business process.

However, the level of detail required will depend on the type of change being considered. The process model is typically used in structured analysis and design methods. Also called a data flow diagram (DFD), it shows the flow of information through a system. Each process transforms inputs into outputs.

### process

The process shows a part of the system that transforms inputs into outputs; that is, it shows how one or more inputs are changed into outputs. Some systems analysts prefer to use an oval or a rectangle with rounded edges, as shown in Figure below; still others prefer to use a rectangle, as shown in Figure. The differences between these three shapes are purely cosmetic, though it is obviously important to use the same shape consistently to represent all the functions in the system. Throughout the rest of this book, we will use the circle or bubble.

In some cases, the process will contain the name of a person or a group of people (*e.g.*, a department or a division of an organization), or a computer, or a mechanical device. That is, the process sometimes describes who or what is carrying out the process, rather than describing what the process is.

### The Flow

A flow is represented graphically by an arrow into or out of a process. The flow is used to describe the movement of chunks, or packets of information from one part of the system to another part. For most of the systems that you model as a systems analyst, the flows will indeed represent data, that

is, bits, characters, messages, floating point numbers, and the various other kinds of information that computers can deal with. But DFDs can also be used to model systems other than automated, computerized systems; we may choose, for example, to use a DFD to model an assembly line in which there are no computerized components.

In such a case, the packets or chunks carried by the flows will typically be physical materials; an example is shown in Figure below. For many complex, real-world systems, the DFD will show the flow of materials and data.

The flows in Figures are *named*. The name represents the meaning of the packet that moves along the flow. A corollary of this is that the flow carries only one type of packet, as indicated by the flow name. The systems analyst should not name a dataflow Apples and organes and widgets and various other thing.

*However, we will see in Part III, that there are exceptions to this convention*: it is sometimes useful to consolidate several elementary dataflow into a consolidated flow. Thus, one might see a single dataflow labeled vegetables instead of several different dataflow labeled potatoes, brussel sprouts, and peas. whether data (or material) are moving into or out of a process (or doing both). The flow shown in Figure, for example, clearly shows that a telephone number is being sent into the process labeled Validate Phone Number.

Dataflows can diverge and converge in a DFD; conceptually, this is somewhat like a major river splitting into smaller tributaries, or tributaries joining together. However, this has a special meaning in a typical DFD in which packets of data are moving through the system: in the case of a diverging

flow, it means that duplicate copies of a packet of data are being sent to different parts of the system, or that a complex packet of data is being split into several more elementary data packets, each of which is being sent to different parts of the system, or that the dataflow pipeline carries items with different values (*e.g.*, vegetables whose values may be "potato," "brussel sprout," or "lima bean") that are being separated. Conversely, in the case of a converging flow, it means that several elementary packets of data are joining together to form more complex, aggregate packets of data.

## The Terminator

The next component of the DFD is a *terminator*; it is graphically represented as a rectangle, as shown in Figure. Terminators represent external entities with which the system communicates. Typically, a terminator is a person or a group of people, for example, an outside organization or government agency, or a group or department that is *within* the same company or organization, but *outside*the control of the system being modeled.

In some cases, a terminator may be another system, for example, some other computer system with which your system will communicate.


**Fig.** Graphical Representation of a Terminator.

*There are three important things that we must remember about terminators*:

1. They are *outside* the system we are modeling; the flows connecting the terminators to various processes (or

stores) in our system represent the interface between our system and the outside world.

2. As a consequence, it is evident that neither the systems analyst nor the systems designer are in a position to change the contents of a terminator or the way the terminator works. In the language of several classic textbooks on structured analysis, the terminator is outside the domain of change. What this means is that the systems analyst is modeling a system with the intention of allowing the systems designer a considerable amount of flexibility and freedom to choose the best (or most efficient, or most reliable, etc.) implementation possible.
3. Any relationship that exists *between* terminators will not be shown in the DFD model. There may indeed be several such relationships, but, by definition, those relationships are not part of the system we are studying. Conversely, if there *are*relationships between the terminators, and if it is essential for the systems analyst to model those requirements in order to properly document the requirements of the system, then, by definition, the terminators are actually part of the system and should be modeled as processes.

## Entity Relationship Diagrams

### Introduction

An entity can be thought of as a class of data. Each entity has a name, a definition, a type. In addition, each entity has a set of attributes that describe the various characteristics of the entity. Each attribute also has a name, a definition, a

type and constraints. The attribute types are text, numeric, binary and date types. Field and attributes are different name for the same thing. Entity and table are different name for the same thing. In the context of relationship diagrams, the words entity and attributes are used. In the context of physical database work, the words table and field are used.

An Entity-Relationship (E-R) Diagram (or E-R Model) visually depicts an organization's entities, the entities' relationships to each other, and the business rules (*i.e.*, cardinality and dependency) associated with the relationships. The E-R Diagram is the picture used to represent and test the knowledge obtained from Data Modelling.

*The output of Data Modelling includes*:

- E-R Diagram,
- Descriptions of entities and their relationships,
- Dttributes and their descriptions,
- Edit rules,
- Business rules,
- Volumetrics.

The last step in creating entity relationship diagrams is the specification of the relationships among the entities. Just as every object in the real world has some kind of relationship to one or more objects so too the entities in a database are related to other entities.

The nature of relationships between entities is usually implied in the very definition of the entity. Despite the obviousness of these relationships, it is important to review all entities and specify how they relate to each other.

*There are at least three types of relationships possible*:

1. One to one, where one entity corresponds to exactly another entity. For example, a table about patient's death has a one to one relationship with the table "Person."
2. One to many, where one instance of one entity can be repeatedly used by another. For example the look up table Gender may be repeatedly used in the table "Patient."
3. Many to many where one instance of both entities can be repeatedly used by another. For example, tables "Patient" and "Clinician" have many to many relationships as a clinician may have many patient and a patient may have many clinicians.

Sometimes, the relationship between two entities is not clear. The most common cause is that a third entity is missing. This often occurs when two entity have many to many relationship. For example, the entity Patient and the entity Clinician have, as mentioned earlier, many to many relationship. It is difficult to show these relationships inside a database in a way that can easily be manipulated.

An alternative is to show a new table that links these two tables to each other and has one to many relationship to each of the tables. For example, we can make a new table called Visit. Within a visit a patients is diagnosed. Both the patients and the clinicians identity are kept in the visit table. The Visit table has one to many relationship with either patient or clinician table. Sometimes, as we specify the relationships among entities, a new entity must be defined.

Linkages between entities are part of the business rules that databases should capture. In our example, the business rule for the linkage between a Clinician and a Patient is that a clinician may have zero, one, or, more patients.

The business rule for the linkage between the Patient and the Clinician is that a patient may have one, or, more clinicians. Note that these are the business rules that someone may have specified. In a different information system someone could decide that a patient can only have one clinician at a time, or that the number of clinicians dealing with a patient must always be 3, or some other similar rule. The important point is that entities can be linked to each other, and that the nature of the linkage is part of the business rules of the system.

In Access, a database, The line shows the relationship between the two tables and the shared field shows the nature of the relationship.

The arrow shows if the relationship is one to many, with the many side shown by the direction of the arrow. As with the specification of the entities discussed at the beginning of this lecture, the documentation of the relationships is part of the logical information model.

The format for documenting the linkages among entities includes the name of both entities, the verb phrase that describes the semantics of the linkage and the cardinality of the linkage (*i.e.* whether one to one, one to many or many to many). The statement of the cardinality can be made plain English. All relationships must be documented before proceeding to the physical design of the database.

## Components of Entity-Relationship Diagram

### Entities

An entity is a person, place, thing, event, or concept of interest to the business or organization about which data is likely to be kept. For example, in a school environment possible entities might be Student, Instructor, and Class.

Entity type refers to a generic class of things such as Company. Entity is the short form of entity-type. Entity occurrence refers to specific instances or examples of a type. For example, one occurrence of the entity Car is Chevrolet Cavalier. An entity usually has attributes (*i.e.*, data elements) that further describe it. Each attribute is a characteristic of the entity. An entity must possess a set of one or more attributes that uniquely identify it (called a primary key).

Identifying entities is the first step in Data Modelling. Start by gathering existing information about the organization. Use documentation that describes the information and functions of the subject area being analysed, and interview subject matter specialists (*i.e.*, end-users). Derive the preliminary entity-relationship diagram from the information gathered by identifying objects (*i.e.*, entities) for which information is kept. Entities are easy to find. Look for the people, places, things, organizations, concepts, and events that an organization needs to capture, store, or retrieve information about.

### Types of Entities

Different types of entities are required to provide a complete and accurate representation of an organization's

data and to enable the analyst to use the Entity-Relationship Diagram as a starting point for physical database design.

*Types of entities include*:

- Fundamental where the entity is a base entity that depends on no other for its existence. A fundamental entity has a primary key that is independent of any other entity and is typically composed of a single attribute. Fundamental entities are real-world, tangible objects, such as, Employee, Customer, or Product.
- Attributive where the entity depends on another for its existence, for example, Employee Hobby depends on Employee. An attributive entity depends on another entity for parts of its primary key. It can result from breaking out a repeating group, the first rule of normalization, or from an optional attribute.
- Associative where the entity describes a connection between two entities with an otherwise many-to-many relationship, for example, assignment of Employee to Project (an Employee can be assigned to more than one Project and a Project can be assigned to more than one Employee). If information exists about the relationship, this information is kept in an associative entity. For example, the number of hours the Employee worked on a particular Project is an attribute of the relationship between Employee and Project, not of either Employee or Project. An associative entity is uniquely identified by concatenating the primary keys of the two entities it connects.

The common data elements are put in the supertype entity and the specific data elements are placed with the subtype to which they apply. For example, Employee (supertype) may contain three subtypes, Permanent Employee, Part-time Employee, and Temporary Employee.

All data elements of the supertype must apply to all subtypes. Each subtype contains the same key as the supertype.

Relationships between an entity supertype and its subtypes are always described as "is a." For example, Employee is a Permanent Employee, Employee is a Part-time Employee.

### Identifying Entity Supertypes/Subtypes

Entity supertypes/subtypes involve classes of entities that are truly different, but at the same time, significantly similar.

*When identifying supertypes/subtypes, look for*:

- Entity types that have the same attributes,
- Entity types that participate in the same relationships,
- Occurrences of an entity that do not participate in all the relationships in which the entity type participates,
- Occurrences of an entity that do not have all the attributes that the entity type has.

## *CATEGORIES OF ENTITIES*

*There are different general categories of entities*:

- Physical entities are tangible and easily understood. They generally fall into one of the following categories:
- People, for example, doctor, patient, employee, customer,

- Property, for example, equipment, land and buildings, furniture and fixtures, supplies,
- Products, such as goods and services.
- Conceptual entities are not tangible and are less easily understood. They are often defined in terms of other entity-types.

  *They generally fall into one of the following categories*:
  - organizations, for example, corporation, church, government,
  - agreements, for example, lease, warranty, mortgage,
  - abstractions, such as strategy and blueprint.
  - Event/State entities are typically incidents that happen. They are very abstract and are often modelled in terms of other entity-types as anassociative entity. Examples of events are purchase, negotiation, service call, and deposit. Examples of states are ownership, enrollment, and employment.

## Imposter Entities

## When an Entity is not an Entity

There are a number of things that may appear to be entities about which facts are kept, but which should not be defined as such.

*These include*:

- Processes,
- Calculations,
- Reports,
- Facts about entities.

### Processes

Processes may actually perform actions on entities but are not, themselves, entities. Examples are:

- Payroll deduction,
- Budgeting (an action on an organization unit).

### Calculations

Calculations are derived from the attributes of an entity. *Examples are*:

- Inventory level,
- Average age,
- Net worth.

### Reports

Reports present facts about one or more entities. *Examples are*:

- Project schedule,
- Income statement.

### Facts About Entities

Facts about entities describe characteristics of an entity and should be modelled as attributes. *Examples are*:

- Telephone number,
- Date of hire.

### Attributes Define Entities

Collectively, attributes define an entity. An attribute is meaningless by itself. For example, date of birth comes from the context of the entity to which it is assigned, for example, date of birth of an employee. Attributes are not shown on the Entity-Relationship Model but are recorded in the

underlying data dictionary which contains the definitions of attributes for all entities and relationships identified in the model. An attribute should not have facts recorded about it. In practice, however, there are exceptions.

For example, you might wish to show address as an attribute of Customer. Address is not significant enough to be modelled as an entity in its own right and would typically be shown as an attribute of Customer. However, at the detailed level, it may itself have attributes such as an indicator for mailing address or home address.Attributes do not have to be recognized and defined during the early stages of entity definition. Entity definition is an iterative process, and it is unlikely that a completely satisfactory Entity-Relationship Model will be obtained on the first iteration.

## Identifying Attributes

*To identify entity attributes, examine*:

- All external entities from the Context Diagram,
- The data flows passed by the external entities,
- Existing automated data,
- Each entity (*i.e.*, generate a list of entity attributes that describe the entity).

## Attributes Versus Data Elements

Attributes have a looser description than data elements. For instance, whereas an attribute may have only a descriptive name, a data element needs:

- A size and range,
- A format and length,
- An accurate and detailed description,
- valid values,

- Defined edit rules.

Some attributes may be converted into many data elements.

*For instance, the attribute "address" may become four data elements representing*:

1. Street Address,
2. City/Town,
3. State/Province,
4. Postal or Zip Code.

Additional data elements may also be defined as a result of customer requirements. For example, the customer may require a list of all companies by county. For the purposes of Data Modelling, attributes and data elements are often considered identical because attributes in the data model typically become data elements in the database.

## Relationships

A relationship is an association that exists between two entities. For example, Instructor teaches Class or Student attends Class. Most relationships can also be stated inversely. For example, Class is taught by Instructor. The relationships on an Entity-Relationship Diagram are represented by lines drawn between the entities involved in the association. The name of the relationship is placed either above, below, or beside the line.

# 5

# Software Life Cycle Models

## Waterfall model

The least flexible of the life cycle models. Still it is well suited to projects which have a well defined architecture and established user interface and performance requirements. The waterfall model *does* work for certain problem domains, notably those where the requirements are well understood in advance and unlikely to change significantly over the course of development. Software products are oriented towards customers like any other engineering products. It is either driver by market or it drives the market. Customer Satisfaction was the main aim in the 1980's. Customer Delight is today's logo and Customer Ecstasy is the new buzzword of the new millennium. Products which are not customer oriented have no place in the market although they are designed using the best technology. The front end of the

product is as crucial as the internal technology of the product. A market study is necessary to identify a potential customer's need. This process is also called as market research. The already existing need and the possible future needs that are combined together for study. A lot of assumptions are made during market study. Assumptions are the very important factors in the development or start of a product's development.

## Advantages

- Simple and easy to use.
- Easy to manage due to the rigidity of the model – each phase has specific deliverables and a review process.
- Phases are processed and completed one at a time.
- Works well for smaller projects where requirements are very well understood.

## Disadvantages

- Adjusting scope during the life cycle can kill a project
- No working software is produced until late during the life cycle.
- High amounts of risk and uncertainty.
- Poor model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Poor model where requirements are at a moderate to high risk of changing.

## Extreme programming (XP)

Is the latest incarnation of Waterfall model and is the most recent software fad. Most postulates of Extreme programming

are pure fantasy. It has been well known for a long time that *big bang* or waterfall models don't work well on projects with complex or shifting requirements. The same is true for XP. Too many shops implement XP as an excuse for not understanding the user requirements. XP try improve classic waterfall model by trying to start coding as early as possible but without creating a full-fledged prototype as the first stage. In this sense it can be considered to be variant of evolutionary prototyping (see below). Often catch phase "Emergent design" is used instead of evolutionary prototyping.

It also introduces a very questionable idea of pair programming as an attempt to improve extremely poor communication between developers typical for large projects. While communication in large projects is really critical and attempts to improve it usually pay well, "pair programming" is a questionable strategy.

*There are two main problems here:*

1 In a way it can be classified as a hidden attempt to create one good programmer out of two mediocre. But in reality it is creating one mediocre programmer from two or one good. No senior developer is going to put up with some jerk sitting on his lap asking questions about every line. It just prevents the level of concentration needed for high quality coding. Microsoft's idea of having a tester for each programmer is more realistic: one developer writes tests.

2 The actual code to be tested. This forces each of them to communicate and because tester has different priorities then developer such communication brings the developer a new and different perspective on his

code, which really improves quality. This combination of different perspectives is a really neat idea as you can see from the stream of Microsoft Office products and operating systems.

## Spiral model

The spiral model is a variant of "dialectical spiral" and as such provides useful insights into the life cycle of the system. Can be considered as a generalization of the prototyping model. That why it is usually implemented as a variant of prototyping model with the first iteration being a prototype.

The spiral model is similar to the incremental model, with more emphases placed on risk analysis. The spiral model has four phases: Planning, Risk Analysis, Engineering and Evaluation.

A software project repeatedly passes through these phases in iterations (called Spirals in this model). The baseline spiral, starting in the planning phase, requirements are gathered and risk is assessed.

Each subsequent spirals builds on the baseline spiral. Requirements are gathered during the planning phase. In the risk analysis phase, a process is undertaken to identify risk and alternate solutions. A prototype is produced at the end of the risk analysis phase. Software is produced in the engineering phase, along with testing at the end of the phase.

## Advantages

- High amount of risk analysis
- Good for large and mission-critical projects.
- Software is produced early in the software life cycle.

## Disadvantages

- Can be a costly model to use.
- Risk analysis requires highly specific expertise.
- Project's success is highly dependent on the risk analysis phase.
- Doesn't work well for smaller projects.

## Evolutionary prototyping model

This is kind of mix of Waterfall model and prototyping. Presuppose gradual refinement of the prototype until a usable product emerge. Might be suitable in projects where the main problem is user interface requirements, but internal architecture is relatively well established and static. In this case system first is coded in a scripting language and then gradually critical components are rewritten in the lower language.

## OSS development model

It is the latest variant of evolutionary prototype model. The "waterfall model" was probably the first published model and as a specific model for military it was not as naive as some proponents of other models suggest. The model was developed to help cope with the increasing complexity of aerospace products. The waterfall model followed a documentation driven paradigm.

Prototyping model was probably the first realistic of early models because many aspects of the syst4m are unclear until a working prototype is developed. A better model, the "spiral model" was suggested by Boehm in 1985. The spiral model is a variant of "dialectical spiral" and as such provides useful insights into the life cycle of the system. But it also

presuppose unlimited resources for the project. No organization can perform more then a couple iterations during the initial development of the system. the first iteration is usually called prototype. Prototype based development requires more talented managers and good planning while waterfall model works (or does not work) with bad or stupid managers works just fine as the success in this model is more determined by the nature of the task in hand then any organizational circumstances.

Like always humans are flexible and programmer in waterfall model can use guerilla methods of enforcing a sound architecture as manager is actually a hostage of the model and cannot afford to look back and re-implement anything substantial. Because the life cycle steps are described in very general terms, the models are adaptable and their implementation details will vary among different organizations.

The spiral model is the most general. Most life cycle models can in fact be derived as special instances of the spiral model. Organizations may mix and match different life cycle models to develop a model more tailored to their products and capabilities.

There is nothing wrong about using waterfall model for some components of the complex project that are relatively well understood and straightforward. But mixing and matching definitely needs a certain level of software management talent.

## V-Shaped Model

Just like the waterfall model, the V-Shaped life cycle is a sequential path of execution of processes. Each phase must

be completed before the next phase begins. Testing is emphasized in this model more so than the waterfall model though. The testing procedures are developed early in the life cycle before any coding is done, during each of the phases preceding implementation.

Requirements begin the life cycle model just like the waterfall model. Before development is started, a system test plan is created. The test plan focuses on meeting the functionality specified in the requirements gathering.

The high-level design phase focuses on system architecture and design. An integration test plan is created in this phase as well in order to test the pieces of the software systems ability to work together. The low-level design phase is where the actual software components are designed, and unit tests are created in this phase as well. The implementation phase is, again, where all coding takes place.

## Advantages

- Simple and easy to use.
- Each phase has specific deliverables.
- Higher chance of success over the waterfall model due to the development of test plans early on during the life cycle.
- Works well for small projects where requirements are easily understood.

## Disadvantages

- Very rigid, like the waterfall model.
- Little flexibility and adjusting scope is difficult and expensive.

- Software is developed during the implementation phase, so no early prototypes of the software are produced.
- Model doesn't provide a clear path for problems found during testing phases.

## Incremental Model

The incremental model is an intuitive approach to the waterfall model. Multiple development cycles take place here, making the life cycle a "multi-waterfall" cycle. Cycles are divided up into smaller, more easily managed iterations.

Each iteration passes through the requirements, design, implementation and testing phases. A working version of software is produced during the first iteration, so you have working software early on during the software life cycle. Subsequent iterations build on the initial software produced during the first iteration.

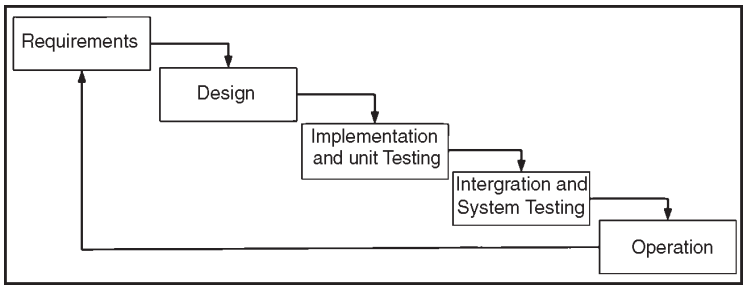



**Fig.** Incremental Life Cycle Model.

## Advantages

- Generates working software quickly and early during the software life cycle.
- More flexible – less costly to change scope and requirements.
- Easier to test and debug during a smaller iteration.

- Easier to manage risk because risky pieces are identified and handled during its iteration.
- Each iteration is an easily managed milestone.

## Disadvantages

- Each phase of an iteration is rigid and do not overlap each other.
- Problems may arise pertaining to system architecture because not all requirements are gathered up front for the entire software life cycle.

# 6

# Process of Software Engineering

## process

Software engineering process and practices are the structures imposed on development of a software product. There are different models of software process (software lifecycle is a synonym) used in different organizations and industries. RAL has identified three levels of software process for its projects. These levels balance the different needs of different types of projects. Scaling the process to the project is vital to its success, too much process can be as problematic as too little; too much process can slow down a purely R&D exploration, too little process can slow down a large development project with hard deliverables. The levels are briefly identified as follows:

*Level* 1: R&D

- No software products delivered, pure research

- Minimal software process

*Level* 2: Research system

- Larger development team, informal software releases
- Moderate software process

*Level* 3: Delivered system

- Large software development team, formal software releases
- More formal software process

For example, the Juneau, Alaska Winds Project has evolved from a Level 1 to a Level 3 project over multiple years. It started as a purely R&D effort (Level 1), expanded to a field programme in Juneau (Level 2), and is currently running in the field as a Operational Prototype (Level 3).

The software process and software engineering practices have become more formalized and more structured as the project proceeded through the different levels. RAL has evolved a set of software engineering best practices that implement the three software process levels.

These include: source code control, nightly code builds, writing reusable code, using different team models, commitment to deadlines, design and code reviews, risk management, bug tracking, software metrics, software configuration management, requirements management.

Software configuration management (SCM) is a step up in formality and reproducibility from source code control and includes controlling and versioning of software releases. Source code control is a software engineering best practice used with RAL Level 2 and Level 3 projects. SCM is a best practice used on a number of RAL Level 3 projects.

## Description

## Software Engineering Process

*The elements of a software engineering process are generally enumerated as*:

- Marketing Requirements
- System-Level Design
- Detailed Design
- Implementation
- Integration
- Field Testing
- Support

No element of this process ought to commence before the earlier ones are substantially complete, and whenever a change is made to some element, all dependent elements ought to be reviewed or redone in light of that change. It's possible that a given module will be both specified and implemented before its dependent modules are fully specified — this is called advanced development or research.

It is absolutely essential that every element of the software engineering process include several kinds of *review:* peer review, mentor/management review, and cross-disciplinary review. Software engineering elements (whether documents or source code) must have version numbers and auditable histories. "Checking in" a change to an element should require some form of review, and the depth of the review should correspond directly to the scope of the change.

## Marketing Requirements

The first step of a software engineering process is to create a document which describes the target customers and their

reason for needing this product, and then goes on to list the features of the product which address these customer needs. The Marketing Requirements Document (MRD) is the battleground where the answer to the question "What should we build, and who will use it?" is decided.

In many failed projects, the MRD was handed down like an inscribed stone tablet from marketing to engineering, who would then gripe endlessly about the laws of physics and about how they couldn't actually build that product since they had no ready supply of Kryptonite or whatever. The MRD is a joint effort, with engineering not only reviewing but also writing a lot of the text.

## System-Level Design

This is a high-level description of the product, in terms of "modules" (or sometimes "programmes") and of the interaction between these modules. The goals of this document are first, to gain more confidence that the product could work and could be built, and second, to form a basis for estimating the total amount of work it will take to build it. The system-level design document should also outline the system-level testing plan, in terms of customer needs and whether they would be met by the system design being proposed.

## Detailed Design

The detailed design is where every module called out in the system-level design document is described in detail. The interface (command line formats, calling API, externally visible data structures) of each module has to be completely determined at this point, as well as dependencies between

modules. Two things that will evolve out of the detailed design is a PERT or GANT chart showing what work has to be done and in what order, and more accurate estimates of the time it will take to complete each module.

Every module needs a unit test plan, which tells the implementor what test cases or what kind of test cases they need to generate in their unit testing in order to verify functionality. Note that there are additional, nonfunctional unit tests which will be discussed later.

## Implementation

Every module described in the detailed design document has to be implemented. This includes the small act of coding or programming that is the heart and soul of the software engineering process. It's unfortunate that this small act is sometimes the only part of software engineering that is taught (or learned), since it is also the only part of software engineering which can be effectively self-taught.

A module can be considered implemented when it has been created, tested, and successfully used by some other module (or by the system-level testing process). Creating a module is the old edit-compile-repeat cycle. Module testing includes the unit level functional and regression tests called out by the detailed design, and also performance/stress testing, and code coverage analysis.

## Integration

When all modules are nominally complete, system-level integration can be done. This is where all of the modules move into a single source pool and are compiled and linked and packaged as a system. Integration can be done

incrementally, in parallel with the implementation of the various modules, but it cannot authoritatively approach "doneness" until all modules are substantially complete.

Integration includes the development of a system-level test. If the built package has to be able to install itself (which could mean just unpacking a tarball or copying files from a CD-ROM) then there should be an automated way of doing this, either on dedicated crash and burn systems or in containerized/simulated environments. Sometimes, in the middleware arena, the package is just a built source pool, in which case no installation tools will exist and system testing will be done on the as-built pool. Once the system has been installed (if it is installable), the automated system-level testing process should be able to invoke every public command and call every public entry point, with every possible reasonable combination of arguments.

If the system is capable of creating some kind of database, then the automated system-level testing should create one and then use external (separately written) tools to verify the database's integrity. It's possible that the unit tests will serve some of these needs, and all unit tests should be run in sequence during the integration, build, and packaging process.

## Field Testing

Field testing usually begins internally. That means employees of the organization that produced the software package will run it on their own computers. This should ultimately include all "production level" systems — desktops, laptops, and servers.

The statement you want to be able to make at the time you ask customers to run a new software system (or a new version of an existing software system) is "we run it ourselves." The software developers should be available for direct technical support during internal field testing. Ultimately it will be necessary to run the software externally, meaning on customers' (or prospective customers') computers. It's best to pick "friendly" customers for this exercise since it's likely that they will find a lot of defects — even some trivial and obvious ones — simply because their usage patterns and habits are likely to be different from those of your internal users.

The software developers should be close to the front of the escalation path during external field testing. Defects encountered during field testing need to be triaged by senior developers and technical marketers, to determine which ones can be fixed in the documentation, which ones need to be fixed before the current version is released, and which ones can be fixed in the next release (or never).

## Support

Software defects encountered either during field testing or after the software has been distributed should be recorded in a tracking system. These defects should ultimately be assigned to a software engineer who will propose a change to either the definition and documentation of the system, or the definition of a module, or to the implementation of a module. These changes should include additions to the unit and/or system-level tests, in the form of a regression test to show the defect and therefore show that it has been fixed

(and to keep it from recurring later). Just as the MRD was a joint venture between engineering and marketing, so it is that support is a joint venture between engineering and customer service. The battlegrounds in this venture are the bug list, the categorization of particular bugs, the maximum number of critical defects in a shippable software release, and so on.

## Software Quality Attribute

### high quality software

Developing high quality software is hard, especially when the interpretation of term "quality" is patchy based on the environment in which it is used. In order to know if quality has been achieved, or degraded, it has to be measured, but determining what to measure and how is the difficult part. Software Quality Attributes are the benchmarks that describe system's intended behaviour within the environment for which it was built.

The quality attributes provide the means for measuring the fitness and suitability of a product. Software architecture has a profound affect on most qualities in one way or another, and software quality attributes affect architecture. Identifying desired system qualities before a system is built allows system designer to mold a solution (starting with its architecture) to match the desired needs of the system within the context of constraints (available resources, interface with legacy systems, etc). When a designer understands the desired qualities before a system is built, then the likelihood of selecting or creating the right architecture is improved.

## Statements

Both statements are useless as they provide no tangible way of measuring the behaviour of the system. The quality attributes must be described in terms of scenarios, such as "when 100 users initiate 'complete payment' transition, the payment component, under normal circumstances, will process the requests with an average latency of three seconds." This statement, or scenario, allows an architect to make quantifiable arguments about a system.

A scenario defines the source of stimulus (users), the actual stimulus (initiate transaction), the artifact affected (payment component), the environment in which it exists (normal operation), the effect of the action (transaction processed), and the response measure (within three seconds). Writing such detailed statements is only possible when relevant requirements have been identified and an idea of components has been proposed.

## Qualities

Scenarios help describe the qualities of a system, but they don't describe how they will be achieved. Architectural tactics describe how a given quality can be achieved. For each quality there may be a large set of tactics available to an architect. It is the architect's job to select the right tactic in light of the needs of the system and the environment.

For example, a performance tactics may include options to develop better processing algorithms, develop a system for parallel processing, or revise event scheduling policy. Whatever tactic is chosen, it must be justified and documented.

## Software Qualities

It would be naïve to claim that the list below is as a complete taxonomy of all software qualities – but it's a solid list of general software qualities compiled from respectable sources. Domain specific systems are likely to have an additional set of qualities in addition to the list below. System qualities can be categorized into four parts: runtime qualities, non-runtime qualities, business qualities, and architecture qualities.

Each of the categories and its associated qualities are briefly described below. Other articles on this site provide more information about each of the software quality attributes listed below, their applicable properties, and the conflicts the qualities.

## types of softare qualities

*It defines six software quality attributes, also called quality characteristics*:

1. *Functionality*: Are the required functions available, including interoperabilithy and security
2. *Reliability*: Maturity, fault tolerance and recoverability
3. *Usability*: How easy it is to understand, learn, operate the software system
4. *Efficiency*: Performance and resource behaviour
5. *Maintainability*: How easy is it to modify the software
6. *Portability*: Can the software easily be transferred to another environment, including installability

## Product Revision

*The product revision perspective identifies quality factors that influence the ability to change the software product, these factors are*:

- Maintainability, the ability to find and fix a defect.
- Flexibility, the ability to make changes required as dictated by the business.
- Testability, the ability to Validate the software requirements.

## Product Transition

*The product transition perspective identifies quality factors that influence the ability to adapt the software to new environments*:

- Portability, the ability to transfer the software from one environment to another.
- Reusability, the ease of using existing software components in a different context.
- Interoperability, the extent, or ease, to which software components work together.

## Product Operations

*The product operations perspective identifies quality factors that influence the extent to which the software fulfils its specification*:

- Correctness, the functionality matches the specification.
- Reliability, the extent to which the system fails.
- Efficiency, system resource (including cpu, disk, memory, network) usage.
- Integrity, protection from unauthorized access.
- Usability, ease of use.

# 7

# Configuration in Computer Networking

Setting up network components for FTP is not trivial for use outside your LAN (Local Area Network). Since so many firewalls and routers exist, it is impractical to give detailed step-by-step instructions suitable for every user. It is important to understand the basics of the FTP protocol in order to configure FileZilla and the routers and/or firewalls involved. This documentation describes the history of the FTP and how some aspects of the protocol work. Reading it carefully will save you a lot of trouble setting up FTP.

## Background

In the fast living world of the internet, the File Transfer Protocol is not just old, it's ancient. Early drafts of the protocol go back as far as 1971, and the current specifications are from 1985. The protocol might even be older than you!

Back then, the Internet was mainly used by universities and research centres. The community was small, many users knew each other and all were collaborating together. The internet was a friendly, trusting place. Security was not much of a concern. A lot has changed since then. The Internet is now ubiquitous, with millions of users communicating with each other in many different ways. It is also a more hostile place. The availability and openness has attracted malicious users who exploit design limitations, incomplete implementations, bugs, and the inexperience of other users. A well-known software company located in Redmond, WA certainly played a part in this.

*Several attempts have been made to address these problems:*

- NAT (Network Address Translation) routers. Many hosts and routers on the internet use the IPv4 protocol. The number of hosts connected to the internet has reached IPV4's design limit for the number of addresses (IPv6 is designed to relieve this). NAT routers allow multiple systems within a LAN to connect to the outside world with one external IP address.
- Personal firewalls try to protect personal computers from attacks by malicious users.

Unfortunately, both NAT and personal firewalls conflict with FTP more often than not. To make things worse, some are themselves flawed, causing additional problems regarding FTP.

## Technical Background

What distinguishes FTP from most other protocols is the use of secondary connections for file transfers. When you

connect to an FTP server, you are actually making two connections. First, the so-called *control connection* is established, over which FTP commands and their replies are transferred. Then, in order to transfer a file or a directory listing, the client sends a particular command over the control connection to establish the *data connection*.

The data connection can be established two different ways, using *active mode* or *passive mode*.

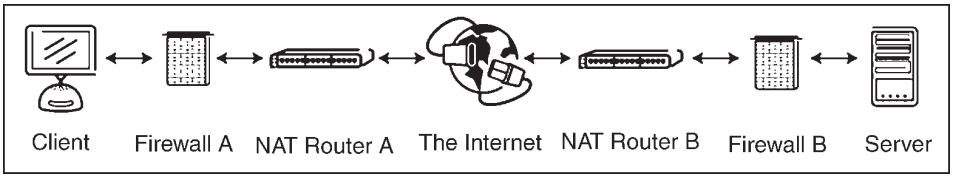
In passive mode, which is recommended, the client sends the PASV command to the server, and the server responds with an address. The client then issues a command to transfer a file or to get a directory listing, and establishes a secondary connection to the address returned by the server.

In active mode, the client opens a socket on the local machine and tells its address to the server using the PORT command. Once the client issues a command to transfer a file or listing, the server will connect to the address provided by the client.

In both cases, the actual file or listing is then transferred over the data connection.

Generally, establishing outgoing connections requires less configuration on the routers/firewalls involved than establishing incoming connections. In passive mode, the connection is outgoing on the client side and incoming on the server side and in active mode this is reversed. Note that the only differences are in establishing a connection. Once established, the connection can be used for uploads or downloads.

*A common network setup might look like this:*

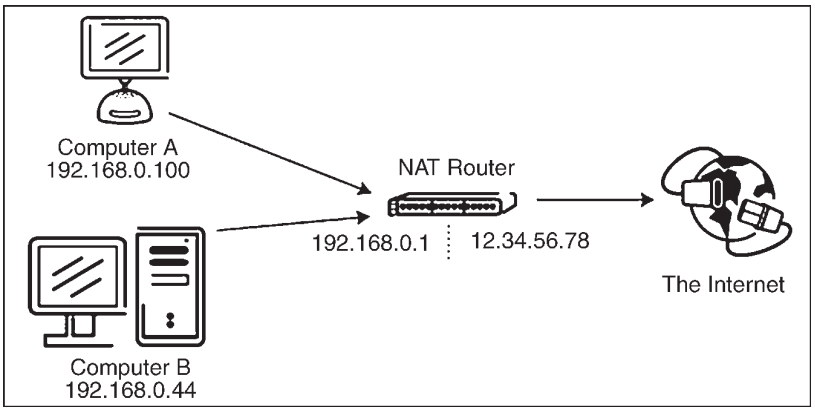



In passive mode, the router and firewall on the server side need to be configured to accept and forward incoming connections. On the client side, however, only outgoing connections need to be allowed (which will already be the case most of the time). Analogously, in active mode, the router and firewall on the client side need to be configured to accept and forward incoming connections. Only outgoing connections have to be allowed on the server side.

Since in most cases one server provides a service for many users, it is much easier to configure the router and firewall on the server side once for passive mode than to configure the client's router/firewall for each individual client in active mode. Therefore, passive mode is recommended in most cases.

## NAT Routers

Most broadband users will have a NAT (Network Address Translation) router between their computer and the internet. This may be a standalone router device (perhaps a wireless router), or be built into a DSL or cable modem. In a NAT environment, all systems behind the NAT router form a *Local Area Network (LAN)*, and each system in the LAN has a local IP address (recognizable as four small numbers separated by dots). The NAT router itself has a local IP address as well. In addition, the NAT router also has an external IP address by which it is known to the Internet. An example system might look like this:




The internal IP addresses are only valid inside the LAN, since they would make little sense to a remote system. Think about a server behind a NAT router. Imagine what might happen if a client requests passive mode, but the server doesn't know the external IP address of the NAT router. If the server sends its internal address to the client, two things could happen:

- If the client is not behind a NAT, the client would abort since the address is invalid.
- If the client is behind a NAT, the address given by the server might be the same as a system in the client's own LAN.

Obviously, in both cases passive mode would be impossible.

So if a server is behind a NAT router, it needs to know the external IP address of the router in passive mode. In this case, the server sends the router's external address to the client. The client then establishes a connection to the NAT router, which in turn routes the connection to the server.

## Firewalls

*Personal firewalls* are installed on many systems to protect users from security vulnerabilities in the operating system or applications running on it. Over the internet, malware

such as worms try to exploit these flaws to infect your system. Firewalls can help to prevent such an infection. However, firewalls and other security applications can sometimes interfere with non-malicious file transfers.

Especially if using FTP, firewall users might occasionally see messages like this from their firewall:

```
Trojan Netbus blocked on port 12345 used by FileZilla.exe
```

In many cases, this is a false alarm. Any program can choose any port it wants for communication over the internet. FileZilla, then, might choose a port that is coincidentally also the default port of a trojan or some other malware being tracked by your firewall. FileZilla is clean of malware *as long as it is downloaded from the official website*.

## TCP/IP Protocol Architecture

While there is no universal agreement about how to describe TCP/IP with a layered model, it is generally viewed as being composed of fewer layers than the seven used in the OSI model. Most descriptions of TCP/IP define three to five functional levels in the protocol architecture.

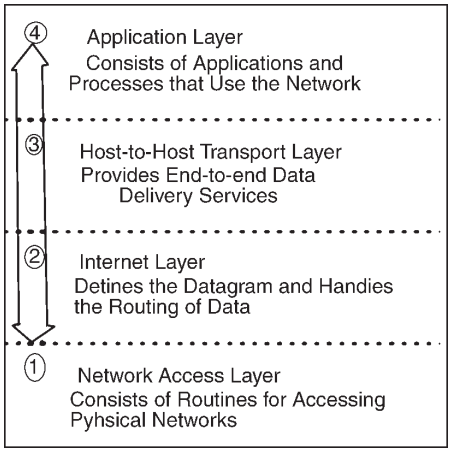



**Figure** : Layers in the TCP/IP Protocol Architecture.

As in the OSI model, data is passed down the stack when it is being sent to the network, and up the stack when it is being received from the network. The four-layered structure of TCP/IP is seen in the way data is handled as it passes down the protocol stack from the Application Layer to the underlying physical network. Each layer in the stack adds control information to ensure proper delivery. This control information is called a *header* because it is placed in front of the data to be transmitted. Each layer treats all of the information it receives from the layer above as data and places its own header in front of that information. The addition of delivery information at every layer is called *encapsulation*. When data is received, the opposite happens. Each layer strips off its header before passing the data on to the layer above. As information flows back up the stack, information received from a lower layer is interpreted as both a header and data.

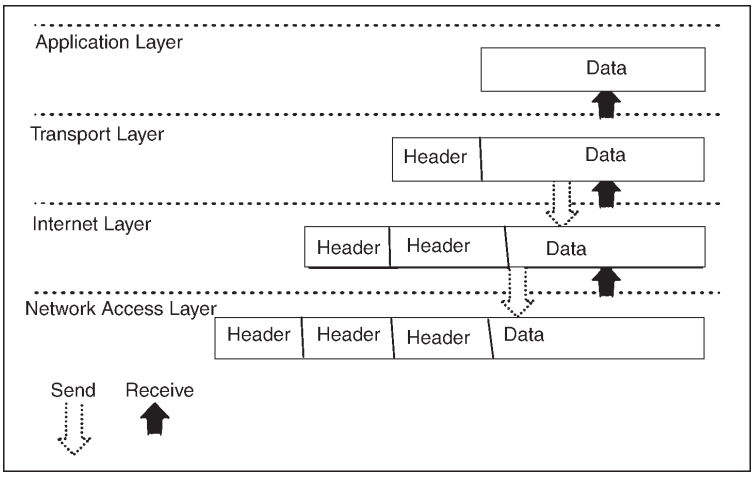



**Figure** : Data Encapsulation.

Each layer has its own independent data structures. Conceptually, a layer is unaware of the data structures used by the layers above and below it. In reality, the data structures

of a layer are designed to be compatible with the structures used by the surrounding layers for the sake of more efficient data transmission. Still, each layer has its own data structure and its own terminology to describe that structure.

The terms used by different layers of TCP/IP to refer to the data being transmitted. Applications using TCP refer to data as a *stream*, while applications using the User Datagram Protocol (UDP) refer to data as a *message*. TCP calls data a *segment*, and UDP calls its data structure a *packet*. The Internet layer views all data as blocks called *datagrams*. TCP/IP uses many different types of underlying networks, each of which may have a different terminology for the data it transmits. Most networks refer to transmitted data as *packets* or *frames*.

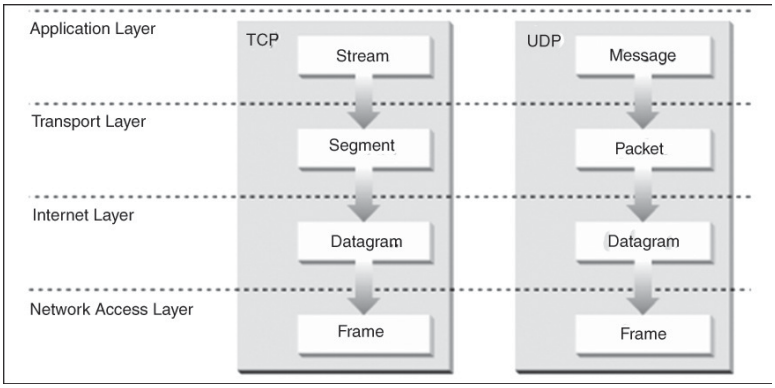



**Figure** : Data Structures.

## Network Access Layer

The *Network Access Layer* is the lowest layer of the TCP/IP protocol hierarchy. The protocols in this layer provide the means for the system to deliver data to the other devices on a directly attached network. It defines how to use the network to transmit an IP datagram.

Unlike higher-level protocols, Network Access Layer protocols must know the details of the underlying network (its packet structure, addressing, etc.) to correctly format the data being transmitted to comply with the network constraints. The TCP/IP Network Access Layer can encompass the functions of all three lower layers of the OSI reference Model (Network, Data Link, and Physical).

The Network Access Layer is often ignored by users. The design of TCP/IP hides the function of the lower layers, and the better known protocols (IP, TCP, UDP, etc.) are all higher-level protocols. As new hardware technologies appear, new Network Access protocols must be developed so that TCP/IP networks can use the new hardware. Consequently, there are many access protocols- one for each physical network standard.

Functions performed at this level include encapsulation of IP datagrams into the frames transmitted by the network, and mapping of IP addresses to the physical addresses used by the network. One of TCP/IP's strengths is its universal addressing scheme. The IP address must be converted into an address that is appropriate for the physical network over which the datagram is transmitted.

Two examples of RFCs that define network access layer protocols are:

- RFC 826, *Address Resolution Protocol (ARP)*, which maps IP addresses to Ethernet addresses
- RFC 894, *A Standard for the Transmission of IP Datagrams over Ethernet Networks*, which specifies how IP datagrams are encapsulated for transmission over Ethernet networks.

## Internet Layer

The layer above the Network Access Layer in the protocol hierarchy is the *Internet Layer*. The Internet Protocol, RFC 791, is the heart of TCP/IP and the most important protocol in the Internet Layer. IP provides the basic packet delivery service on which TCP/IP networks are built. All protocols, in the layers above and below IP, use the Internet Protocol to deliver data. All TCP/IP data flows through IP, incoming and outgoing, regardless of its final destination.

## Internet Protocol

The Internet Protocol is the building block of the Internet. Its functions include:

- Defining the datagram, which is the basic unit of transmission in the Internet
- Defining the Internet addressing scheme
- Moving data between the Network Access Layer and the Host-to-Host Transport Layer
- Routing datagrams to remote hosts
- Performing fragmentation and re-assembly of datagrams.

Before describing these functions in more detail, let's look at some of IP's characteristics. First, IP is a *connectionless protocol*. This means that IP does not exchange control information (called a "handshake") to establish an end-to-end connection before transmitting data. In contrast, a *connection-oriented protocol* exchanges control information with the remote system to verify that it is ready to receive data before any data is sent. When the handshaking is successful, the systems are said to have established a

*connection*. Internet Protocol relies on protocols in other layers to establish the connection if they require connection-oriented service.

IP also relies on protocols in the other layers to provide error detection and error recovery. The Internet Protocol is sometimes called an *unreliable protocol* because it contains no error detection and recovery code. This is not to say that the protocol cannot be relied on-quite the contrary. IP can be relied upon to accurately deliver your data to the connected network, but it doesn't check whether that data was correctly received. Protocols in other layers of the TCP/IP architecture provide this checking when it is required.

## The Datagram

The TCP/IP protocols were built to transmit data over the ARPANET, which was a *packet switching network*. A *packet* is a block of data that carries with it the information necessary to deliver it-in a manner similar to a postal letter, which has an address written on its envelope. A packet switching network uses the addressing information in the packets to switch packets from one physical network to another, moving them towards their final destination. Each packet travels the network independently of any other packet.

The *datagram* is the packet format defined by Internet Protocol. A pictorial representation of an IP datagram. The first five or six 32-bit words of the datagram are control information called the *header*. By default, the header is five words long; the sixth word is optional. Because the header's length is variable, it includes a field called *Internet Header Length (IHL)* that indicates the header's length in words. The header contains all the information necessary to deliver the packet.

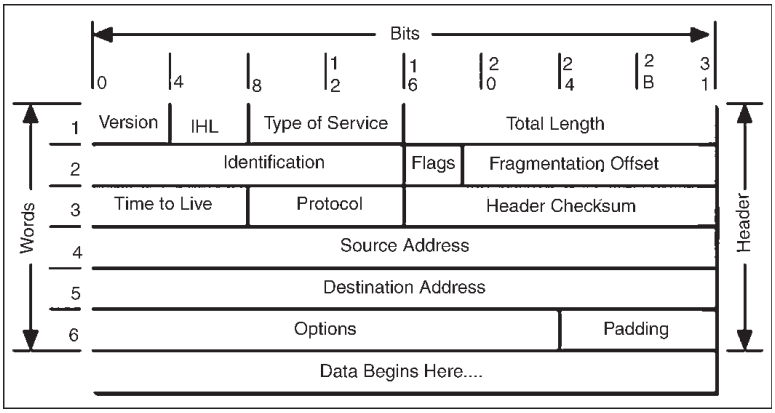



**Figure** : IP Datagram Format.

The Internet Protocol delivers the datagram by checking the *Destination Address* in word 5 of the header. The Destination Address is a standard 32-bit IP address that identifies the destination network and the specific host on that network. If the Destination Address is the address of a host on the local network, the packet is delivered directly to the destination. If the Destination Address is not on the local network, the packet is passed to a gateway for delivery. *Gateways* are devices that switch packets between the different physical networks. Deciding which gateway to use is called *routing*. IP makes the routing decision for each individual packet.

## Malicious Routers, Firewalls and Data Sabotage

Some routers and firewalls pretend to be smart. They analyse connections and, if they think they detect FTP, they silently change the data exchanged between client and server. If the user has not explicitly enabled this feature, this

behaviour is essentially data sabotage and can cause various problems. For an example, imagine a client behind a NAT router trying to connect to the server. Let's further assume that this client does not know it is behind a NAT and wants to use active mode. So it sends the PORT command with the user's local, un-routable IP address to the server:

```
PORT 10,0,0,1,12,34
```

This command tells the server to connect to the address 10.0.0.1 on port 12*256+34 = 3106

The NAT router sees this and silently changes the command to include the external IP address. At the same time, the NAT router will also create a temporary port forwarding for the FTP session, possibly on a different port even:

```
PORT 123,123,123,123,24,55
```

The above command tells the server to connect to the address 123.123.123.123 on port 24*256+55 = 6199

With this behaviour, a NAT router allows an improperly configured client to use active mode.

So why is this behaviour bad? Essentially, it can cause a number of problems if it is enabled by default, without explicit user consent. The FTP connections in their most basic form appear to work, but as soon as there's some deviation from the basic case, everything will fail, leaving the user stumped:

- The NAT router blindly assumes some connection uses FTP based on criteria like target ports or the initial server response:
  (a) The used protocol is detected as FTP, yet there is no guarantee that this is true (a *false positive*). Though unlikely, it is conceivable that a future revision of the FTP protocol might change the

syntax of the PORT command. A NAT router modifying the PORT command would then silently change things it does not support and thus break the connection.

(B) The router's protocol detection can fail to recognize an FTP connection (a *false negative*). Say the router only looks at the target port, and if it is 21, it detects it as FTP. As such, active mode connections with an improperly configured client to servers running on port 21 will work, but connections to other servers on non-standard ports will fail.

- Obviously, a NAT router can no longer tamper with the connection as soon as an encrypted FTP session is used, again leaving the user clueless why it works for normal FTP but not for encrypted FTP.
- Say a client behind a NAT router sends "PORT 10,0,0,1,12,34". How does the NAT router know the client is improperly configured? It is also possible that the client is properly configured, yet merely wants to initiate an FXP (server-to-server) transfer between the server it is connected to and another machine in the server's own local network.

Therefore, having protocol specific features enabled in a NAT router by default can create significant problems. The solution to all this, then, is to know your router's settings, and to know the configuration abilities of a router before you set it up. A good NAT router should always be fully protocol-agnostic. The exception is if you as the user have explicitly enabled this feature, knowing all its consequences. The

combination of a NAT router on the client side with active mode, the same applies to a server behind a NAT router and the reply to the PASV command.

## Setting up FileZilla Client

If you're running FileZilla 3, it's recommended you run the network configuration wizard. It will guide you through the necessary steps and can test your configuration after set-up.

Obviously, if you want to connect to any server, you need to tell your firewall that FileZilla should be allowed to open connections to other servers. Most normal FTP servers use port 21, SFTP servers use port 22 and FTP over SSL/TLS (implicit mode) use port 990 by default. These ports are not mandatory, however, so it's best to allow outgoing connections to arbitrary remote ports. Since many servers on the internet are misconfigured and don't support both transfer modes, it's recommended that you configure both transfer modes on your end.

## Passive Mode

In passive mode, the client has no control over what port the server chooses for the data connection. Therefore, in order to use passive mode, you'll have to allow outgoing connections to all ports in your firewall.

## Active Mode

In active mode, the client opens a socket and waits for the server to establish the transfer connection.

By default, FileZilla Client asks the operating system for the machine's IP address and for the number of a free port.

This configuration can only work if you are connected to the internet directly without any NAT router, and if you have set your firewall to allow incoming connections on all ports greater than 1024.

If you have a NAT router, you need to tell FileZilla your external IP address in order for active mode connections to work with servers outside your local network:

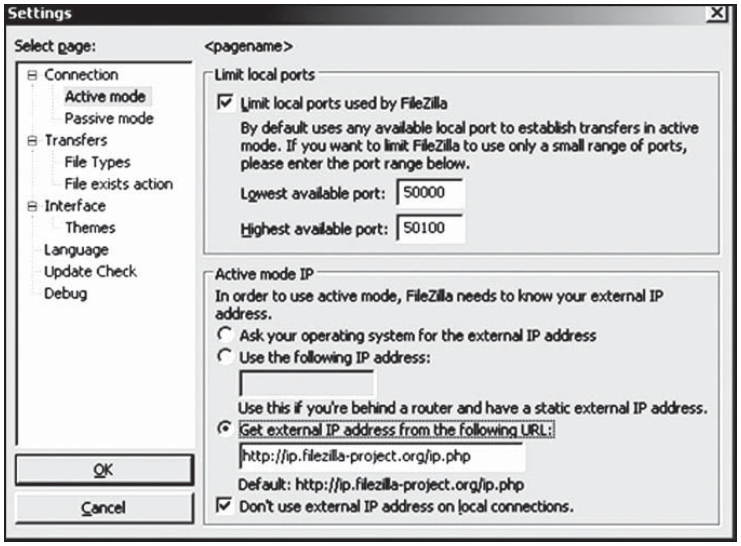
- If you have a fixed external IP address, you can enter it in FileZilla's configuration dialog.
- If you have a dynamic IP address, you can authorize FileZilla to obtain your external IP address from a special website. This will occur automatically each time FileZilla is started. No information will be submitted to the website (regardless of FileZilla version).

If in doubt, use the second option.

If you do not want to allow incoming connections on all ports, or if you have a NAT router, you need to tell FileZilla to use a specific range of ports for active mode connections. You will have to open these ports in your firewall. If you have a NAT router, you need to forward these ports to the local machine FileZilla is installed on. Depending on your router model, you can either forward a range of ports or you need to forward all ports individually.

Valid ports can be from 1 to 65535; however, ports less than 1024 are reserved for other protocols. It is best to choose ports greater than or equal to 50000 for active mode FTP. Due to the nature of TCP (the underlying transport protocol), a port cannot be reused immediately after each connection. Therefore, the range of ports should not be too small to

prevent the failure of transfers of multiple small files. A range of 50 ports should be sufficient in most cases.




## Setting up and testing FileZilla Server

Setting up the server is very similar to setting up the client, with the main difference being that the roles of active and passive mode are reversed.

A common mistake, especially by users with NAT routers, is in testing the server. If you are within your local network, you can only test using the local IP address of the server. Using the external address from the inside will probably fail, and one of the following may happen:

- It actually works (surprisingly- and it probably means something else is wrong...)
- The router blocks access to its own external address from the inside, due to identifying it as a possible attack
- The router forwards the connection to your ISP, which then blocks it as a possible attack

Even if the test works, there is no guarantee that an external user can really connect to your server and transfer files. The only reliable way to test your server is to try connecting from an external system, *outside* of your LAN.

## Active Mode

Make sure FileZilla Server is allowed to establish outgoing connections to arbitrary ports, since the client controls which port to use.

On the local end of the connection, FileZilla Server tries to use a port one less than that of the control connection (*e.g.* port 20 if server is listening on port 21). However, this is not always possible- so don't rely on it.

## Passive Mode

The server configuration is very similar to client configuration for active mode. In passive mode, the server opens a socket and waits for the client to connect to it.

By default, FileZilla Server asks the operating system for the machine's IP address, and for a free port number. This configuration can only work if you are connected to the internet directly without any NAT router and if you have set your firewall to allow incoming connections on all ports greater than 1024.

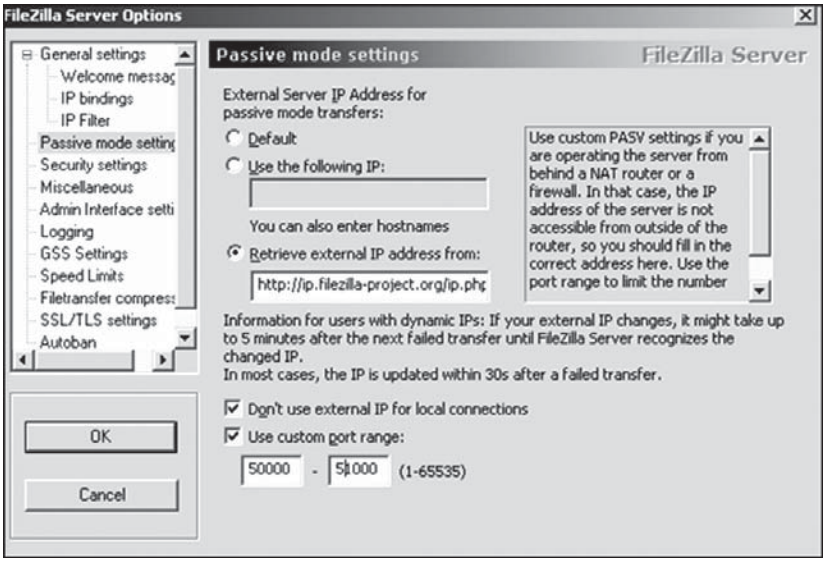
If you have a NAT router, you need to tell FileZilla Server your external IP address or passive mode connections will not work with clients outside your local network:

- If you have a fixed external IP address, you can enter it in the configuration dialog of FileZilla Server.
- If you have a dynamic IP address, you can let FileZilla Server obtain your external IP address from

a special website automatically. Except your version of FileZilla Server, no information will be submitted to that website.

If in doubt, use the second option. If you do not want to allow incoming connections on all ports, or if you have a NAT router, you need to tell FileZilla Server to use a specific range of ports for passive mode connections. You will have to open these ports in your firewall. If you have a NAT router, you need to forward these ports to the local machine FileZilla Server is installed on. Depending on your router model, you can either forward a range of ports or you need to forward all ports individually.

Valid ports can be from 1 to 65535, however ports less than 1024 are reserved for other protocols. It is best to choose ports >= 50000 for passive mode FTP. Due to the nature of TCP (the underlying transport protocol), a port cannot be reused immediately after each connection. Hence the range of ports should not be too small or transfers of multiple small files can fail. A range of 50 ports should be sufficient in most cases.



## Troubleshooting

The following are a few troubleshooting suggestions:

Unfortunately, many personal firewalls and consumer routers are flawed or in some cases, even actively sabotage FTP (*e.g.* SMC Barricade V1.2). First of all, as with all software, you should keep everything updated. This includes the firewall software as well as the firmware version of your router.

If that does not help, you might want to try to uninstall your firewall to see what happens. Simply disabling your firewall might not work, as some firewalls cannot be fully disabled.

If possible, try to connect directly to the internet without a router.

If you are trying to setup a server and it works fine within your LAN but is not reachable from the outside, try changing the listening port. Some ISPs don't like their customers to host servers and they may block ports with numbers under 1024. Another issue may occur if you are hosting an FTP server on default port 21. There might be a firewall at the ISP side of your connection which can do odd things like changing the port for PASV commands. Try using another non-default port for your FTP server.

If you encounter "cannot open data connection" on a random basis (*i.e.*, the ftp client can connect to the ftp server without problem for many connections until it encounters this problem), one possible reason may be that your client PC anti-virus software is configured to block outgoing connections on certain ranges of ports. When your ftp connections are running in pasv mode, the client-side outgoing ports are selected randomly and some of those

randomly selected ports may be blocked by the anti-virus software. To identify this problem, read your anti-virus log on the client. In general, any software that can block certain ranges of outgoing ports (such as PC firewalls) can cause similar FTP grief.

### Timeouts on Large Files

If you can transfer small files without any issues, but transfers of larger files end with a timeout, a broken router and/or firewall exists between the client and the server and is causing a problem.

FTP uses two TCP connections: a control connection to submit commands and receive replies, and a data connection for actual file transfers. It is the nature of FTP that during a transfer the control connection stays completely idle.

The TCP specifications do not set a limit on the amount of time a connection can stay idle. Unless explicitly closed, a connection is assumed to remain alive indefinitely. However, many routers and firewalls automatically close idle connections after a certain period of time. Worse, they often don't notify the user, but just silently drop the connection. For FTP, this means that during a long transfer the control connection can get dropped because it is detected as idle, but neither client nor server are notified. So when all data has been transferred, the server assumes the control connection is alive and it sends the transfer confirmation reply. Likewise, the client thinks the control connection is alive and it waits for the reply from the server. But since the control connection got dropped without notification, the reply never arrives and eventually the connection will timeout.

In an attempt to solve this problem, the TCP specifications include a way to send *keep-alive packets* on otherwise idle TCP connections, to tell all involved parties that the connection is still alive and needed. However, the TCP specifications also make it very clear that these keep-alive packets should not be sent more often than once every two hours. Therefore, with added tolerance for network latency, connections can stay idle for up to 2 hours and 4 minutes.

However, many routers and firewalls drop connections that have been idle for less than 2 hours and 4 minutes. This violates the TCP specifications (RFC 5382 makes this especially clear). In other words, all routers and firewalls that are dropping idle connections too early cannot be used for long FTP transfers. Unfortunately manufacturers of consumer-grade router and firewall vendors do not care about specifications... all they care about is getting your money (and only deliver barely working lowest quality junk). To solve this problem, you need to uninstall affected firewalls and replace faulty routers with better-quality ones.

## Setting up FileZilla Server with Windows Firewall

If you are having problems with setting up FileZilla Server to run behind Windows Firewall (specifically, it fails on "List" and the client receives a "Failed to receive directory listing" error), you must add the FileZilla Server application to Windows Firewall's Exceptions list. To do this, follow these steps:

1. Open Windows Firewall under Control Panel.
2. If using Vista, click "Change Settings"

3. Select the “Exceptions” tab.
4. Click “Add program...”
5. Do NOT select “FileZilla Server Interface” from the list, instead click on “Browse...”
6. Locate the directory you installed FileZilla Server to (normally “C:\Program Files\FileZilla Server\”)
7. Double click or select “FileZilla server.exe” and press open (Once again, NOT “FileZilla Server Interface.exe”)
8. Select “FileZilla server.exe” from the list and click “Ok”
9. Verify that “FileZilla server.exe” is added to the exceptions list and that it has a check mark in the box next to it
10. Press “Ok” to close the window

Passive mode should now work. If you are still having problems connecting (from another computer or outside the network), check your router settings or try to add the port number in the Windows Firewall settings located in the Exceptions tab.

# 8

# Configuration Management

Configuration management is a process for maintaining computer systems, servers, and software in a desired, consistent state. It's a way to make sure that a system performs as it's expected to as changes are made over time.

Managing IT system configurations involves defining a system's desired state—like server configuration—then building and maintaining those systems. Closely related to configuration assessments and drift analyses, configuration management uses both to identify systems to update, reconfigure, or patch.

**Why manage configurations?** Configuration management keeps you from making small or large changes that go undocumented. Misconfigurations like these were identified in our State of Kubernetes Security report as a leading cause of security incidents among containerized or Kubernetes-orchestrated environments.

Misconfigurations can lead to poor performance, inconsistencies, or noncompliance and negatively impact business operations and security. When undocumented changes are made across many systems and applications, it adds to instability and downtime.

Manually identifying systems that require attention, determining remediation steps, prioritizing actions, and validating completion are too complicated to perform in large environments.

But without documentation, maintenance, and a change control process, system administrators and software developers could end up not knowing what's on a server or which software has been updated.

Configuration management systems let you consistently define system settings, as well as build and maintain those systems according to those baseline settings. Configuration management helps users and administrators know where certain services exist and what the current state of applications are.

Proper configuration management tools:

- Classify and manage systems by groups and subgroups.
- Centrally modify base configurations.
- Roll out new settings to all applicable systems.
- Automate system identification, patches, and updates
- Identify outdated, poor performing, and noncompliant configurations.
- Prioritize actions.
- Access and apply prescriptive remediation.

## Configuration management benefits

Think of it like this. If you keep up with the small things, you can avoid more complicated, expensive repairs in the future. Configuration management is about preventing issues so you don't have to deal with as many problems later.

For example, you can make sure that your test and production environments match. That way, you'll have fewer problems with applications once they've been deployed than you would if these environments weren't exactly the same.

With configuration management, you can accurately replicate an environment with the correct configurations and software because you know what exists in the original environment.

## Automating configuration management

The role of configuration management is to maintain systems in a desired state. Traditionally, this was handled manually or with custom scripting by system administrators. Automation is the use of software to perform tasks, such as configuration management, in order to reduce cost, complexity, and errors.

Through automation, a configuration management tool can provision a new server within minutes with less room for error. You can also use automation to maintain a server in the desired state, such as your standard operating environment (SOE), without the provisioning scripts needed previously.

## Configuration management tools

They also help you to keep track of the state of your resources, and keep you from repeating tasks, like installing the same package twice.

Improve system recovery after a critical event with automated configuration management. If a server goes down for an unknown reason, you can deploy a new one quickly and have a record of any changes or updates that occurred so you can identify the source of the problem.

Your configuration management tools can also help you to run an audit of your system so you can more quickly identify where the problem is coming from.

### Git

Git is the industry-leading version control system to track code changes. Adding configuration management data alongside code in a Git repository provides a holistic version control view of an entire project. Git is a foundational tool in higher-level configuration management. The following list of other configuration management tools is designed to be stored in a Git repository and leverage Git version control tracking.

### Docker

Docker introduced containerization that is an advanced form of configuration management -- like a configuration lockdown. Docker is based on configuration files called Dockerfiles, which contain a list of commands that are evaluated to reconstruct the expected snapshot of operating system state. Docker creates containers from these Dockerfiles that are snapshots of a preconfigured application.

Dockerfiles are committed to a Git repository for version tracking and need additional configuration management to deploy them to infrastructure.

### Terraform

Terraform is an open source configuration management platform by HasiCorp. Terraform uses IaC to provision and manage clusters, cloud infrastructure, or services. Terraform supports Amazon Web Services (AWS), Microsoft Azure, and other cloud platforms. Each cloud platform has its own representation and interface for common infrastructure components like servers, databases, and queues. Terraform built an abstraction layer of configuration tools for cloud platforms that enable teams to write files that are reproducible definitions of their infrastructure.

### Ansible, Salt Stack, Chef, Puppet

Ansible, Salt Stack, and Chef are IT automation frameworks. These frameworks automate many traditional system administrators' processes. Each framework uses a series of configuration data files -- usually YAML or XML -- that are evaluated by an executable.

The configuration data files specify a sequence of actions to take to configure a system. The actions are then run by the executable. The executable differs in language between the systems -- Ansible and Salt Stack are Python based and Chef is Ruby. This workflow is similar to running ad-hoc shell scripts but offers a more structured and refined experience through the respective platforms ecosystems. These tools are what will bring enable the automation needed to achieve CI/CD.

# How configuration management fits with DevOps, CI/CD and agile

Configuration data has historically been hard to wrangle and can easily become an afterthought. It's not really code so it's not immediately put in version control and it's not first-class data so It isn't stored in a primary database. Traditional and small scale system administration is usually done with a collection of scripts and ad-hoc processes. Configuration data can be overlooked at times, but it is critical to system operation. The rise of cloud infrastructures has led to the development and adoption of new patterns of infrastructure management. Complex, cloud-based system architectures are managed and deployed through the use of configuration data files. These new cloud platforms allow teams to specify the hardware resources and network connections they need provisioned through human and machine readable data files like YAML. The data files are then read and the infrastructure is provisioned in the cloud. This pattern is called infrastructure as code (IaC).

## DevOps configuration management

In the early years of internet application development, hardware resources and systems administration were primarily performed manually. System administrators wrangled configuration data while manually provisioning and managing hardware resources based on configuration data.

Configuration management is a key part of a DevOps lifecycle. DevOps configuration is the evolution and automation of the systems administration role, bringing automation to infrastructure management and deployment.

DevOps configuration also brings system administration responsibility under the umbrella of software engineering. Enterprises today utilize it to empower software engineers to request and provision needed resources on demand. This removes a potential organizational dependency bottleneck of a software development team waiting for resources from a separate system administration team.

## CI/CD configuration management

CI/CD configuration management utilizes pull request-based code review workflows to automate deployment of code changes to a live software system. This same flow can be applied to configuration changes. CI/CD can be set up so that approved configuration change requests can immediately be deployed to a running system. A perfect example of this process is a GitOps workflow.

## Agile configuration management

Configuration management enables agile teams to clearly triage and prioritize configuration work. Examples of configuration work are chores and tasks like:

- Update the production SSL certificates
- Add a new database endpoint
- Change the password for dev, staging, and production email services.
- Add API keys for a new third-party integration

Once a configuration management platform is in place, teams have visibility into the work required for configuration tasks. Configuration management work can be identified as dependencies for other work and properly addressed as part of agile sprints.

# 9

# Software Development Process

A software development process, also known as a software development lifecycle, is a structure imposed on the development of a software product. Similar terms include software life cycle and *software process*. There are several models for such processes, each describing approaches to a variety of tasks or activities that take place during the process. Some people consider a lifecycle model a more general term and a software development process a more specific term. For example, there are many specific software development processes that 'fit' the spiral lifecycle model.

## Overview

The large and growing body of software development organizations implement process methodologies. Many of them are in the defense industry, which in the U.S. requires a rating based on 'process models' to obtain contracts. The international standard for describing the method of selecting,

implementing and monitoring the life cycle for software is ISO 12207. A decades-long goal has been to find repeatable, predictable processes that improve productivity and quality.

Some try to systematize or formalize the seemingly unruly task of writing software. Others apply project management techniques to writing software. Without project management, software projects can easily be delivered late or over budget. With large numbers of software projects not meeting their expectations in terms of functionality, cost, or delivery schedule, effective project management appears to be lacking. Organizations may create a Software Engineering Process Group (SEPG), which is the focal point for process improvement. Composed of line practitioners who have varied skills, the group is at the center of the collaborative effort of everyone in the organization who is involved with software engineering process improvement.

## Software Development Activities

### Planning

The important task in creating a software product is extracting the requirements or requirements analysis. Customers typically have an abstract idea of what they want as an end result, but not what software should do. Incomplete, ambiguous, or even contradictory requirements are recognized by skilled and experienced software engineers at this point. Frequently demonstrating live code may help reduce the risk that the requirements are incorrect. Once the general requirements are gathered from the client, an analysis of the scope of the development should be determined and clearly stated. This is often called a scope

document. Certain functionality may be out of scope of the project as a function of cost or as a result of unclear requirements at the start of development. If the development is done externally, this document can be considered a legal document so that if there are ever disputes, any ambiguity of what was promised to the client can be clarified.

## Implementation, Testing and Documenting

Implementation is the part of the process where software engineers actually programme the code for the project. Software testing is an integral and important part of the software development process. This part of the process ensures that defects are recognized as early as possible. Documenting the internal design of software for the purpose of future maintenance and enhancement is done throughout development. This may also include the writing of an API, be it external or internal. It is very important to document everything in the project.

## Deployment and Maintenance

Deployment starts after the code is appropriately tested, is approved for release and sold or otherwise distributed into a production environment. Software Training and Support is important and a lot of developers fail to realize that. It would not matter how much time and planning a development team puts into creating software if nobody in an organization ends up using it. People are often resistant to change and avoid venturing into an unfamiliar area, so as a part of the deployment phase, it is very important to have training classes for new clients of your software. Maintaining and enhancing software to cope with newly discovered problems

or new requirements can take far more time than the initial development of the software. It may be necessary to add code that does not fit the original design to correct an unforeseen problem or it may be that a customer is requesting more functionality and code can be added to accommodate their requests. If the labor cost of the maintenance phase exceeds 25% of the prior-phases' labor cost, then it is likely that the overall quality of at least one prior phase is poor. In that case, management should consider the option of rebuilding the system (or portions) before maintenance cost is out of control.

## Software Development Models

Several models exist to streamline the development process. Each one has its pros and cons, and it's up to the development team to adopt the most appropriate one for the project. Sometimes a combination of the models may be more suitable.

## Waterfall Model

The waterfall model shows a process, where developers are to follow these phases in order:

1. Requirements specification (Requirements analysis)
2. Software Design
3. Integration
4. Testing (or Validation)
5. Deployment (or Installation)
6. Maintenance

In a strict Waterfall model, after each phase is finished, it proceeds to the next one. Reviews may occur before moving to the next phase which allows for the possibility of changes

(which may involve a formal change control process). Reviews may also be employed to ensure that the phase is indeed complete; the phase completion criteria are often referred to as a "gate" that the project must pass through to move to the next phase. Waterfall discourages revisiting and revising any prior phase once it's complete. This "inflexibility" in a pure Waterfall model has been a source of criticism by supporters of other more "flexible" models.

## Spiral Model

The key characteristic of a Spiral model is risk management at regular stages in the development cycle. In 1988, Barry Boehm published a formal software system development "spiral model", which combines some key aspect of the waterfall model and rapid prototyping methodologies, but provided emphasis in a key area many felt had been neglected by other methodologies: deliberate iterative risk analysis, particularly suited to large-scale complex systems. The Spiral is visualized as a process passing through some number of iterations, with the four quadrant diagram representative of the following activities:

1. formulate plans to: identify software targets, selected to implement the programme, clarify the project development restrictions;
2. Risk analysis: an analytical assessment of selected programmes, to consider how to identify and eliminate risk;
3. the implementation of the project: the implementation of software development and verification;

Risk-driven spiral model, emphasizing the conditions of options and constraints in order to support software reuse,

software quality can help as a special goal of integration into the product development. However, the spiral model has some restrictive conditions, as follows:

1. The spiral model emphasizes risk analysis, and thus requires customers to accept this analysis and act on it. This requires both trust in the developer as well as the willingness to spend more to fix the issues, which is the reason why this model is often used for large-scale internal software development.
2. If the implementation of risk analysis will greatly affect the profits of the project, the spiral model should not be used.
3. Software developers have to actively look for possible risks, and analyze it accurately for the spiral model to work.

The first stage is to formulate a plan to achieve the objectives with these constraints, and then strive to find and remove all potential risks through careful analysis and, if necessary, by constructing a prototype. If some risks can not be ruled out, the customer has to decide whether to terminate the project or to ignore the risks and continue anyway. Finally, the results are evaluated and the design of the next phase begins.

## Iterative and Incremental Development

Iterative development prescribes the construction of initially small but ever larger portions of a software project to help all those involved to uncover important issues early before problems or faulty assumptions can lead to disaster. Iterative processes are preferred by commercial developers because it allows a potential of reaching the design goals of a customer who does not know how to define what they want.

## Agile Development

Agile software development uses iterative development as a basis but advocates a lighter and more people-centric viewpoint than traditional approaches. Agile processes use feedback, rather than planning, as their primary control mechanism. The feedback is driven by regular tests and releases of the evolving software.

There are many variations of agile processes:

- In Extreme Programming (XP), the phases are carried out in extremely small (or "continuous") steps compared to the older, "batch" processes. The (intentionally incomplete) first pass through the steps might take a day or a week, rather than the months or years of each complete step in the Waterfall model. First, one writes automated tests, to provide concrete goals for development. Next is coding (by a pair of programmers), which is complete when all the tests pass, and the programmers can't think of any more tests that are needed. Design and architecture emerge out of refactoring, and come after coding. Design is done by the same people who do the coding. (Only the last feature — merging design and code — is common to *all* the other agile processes.) The incomplete but functional system is deployed or demonstrated for (some subset of) the users (at least one of which is on the development team). At this point, the practitioners start again on writing tests for the next most important part of the system.
- Scrum

## Process Improvement Models

### Capability Maturity Model Integration

The Capability Maturity Model Integration (CMMI) is one of the leading models and based on best practice. Independent assessments grade organizations on how well they follow their defined processes, not on the quality of those processes or the software produced. CMMI has replaced CMM.

### ISO 9000

ISO 9000 describes standards for a formally organized process to manufacture a product and the methods of managing and monitoring progress. Although the standard was originally created for the manufacturing sector, ISO 9000 standards have been applied to software development as well. Like CMMI, certification with ISO 9000 does not guarantee the quality of the end result, only that formalized business processes have been followed.

### ISO 15504

ISO 15504, also known as Software Process Improvement Capability Determination (SPICE), is a "framework for the assessment of software processes". This standard is aimed at setting out a clear model for process comparison. SPICE is used much like CMMI. It models processes to manage, control, guide and monitor software development. This model is then used to measure what a development organization or project team actually does during software development. This information is analyzed to identify weaknesses and drive improvement. It also identifies strengths that can be continued or integrated into common practice for that organization or team.

## Formal Methods

Formal methods are mathematical approaches to solving software (and hardware) problems at the requirements, specification and design levels. Examples of formal methods include the B-Method, Petri nets, Automated theorem proving, RAISE and VDM. Various formal specification notations are available, such as the Z notation. More generally, automata theory can be used to build up and validate application behaviour by designing a system of finite state machines.

Finite state machine (FSM) based methodologies allow executable software specification and by-passing of conventional coding. Formal methods are most likely to be applied in avionics software, particularly where the software is safety critical. Software safety assurance standards, such as DO178B demand formal methods at the highest level of categorization (Level A). Formalization of software development is creeping in, in other places, with the application of Object Constraint Language (and specializations such as Java Modeling Language) and especially with Model-driven architecture allowing execution of designs, if not specifications.

Another emerging trend in software development is to write a specification in some form of logic (usually a variation of FOL), and then to directly execute the logic as though it were a programme. The OWL language, based on Description Logic, is an example. There is also work on mapping some version of English (or another natural language) automatically to and from logic, and executing the logic directly. Examples are Attempto Controlled English, and Internet Business Logic,

which does not seek to control the vocabulary or syntax. A feature of systems that support bidirectional English-logic mapping and direct execution of the logic is that they can be made to explain their results, in English, at the business or scientific level. The Government Accountability Office, in a 2003 report on one of the Federal Aviation Administration's air traffic control modernization programmes, recommends following the agency's guidance for managing major acquisition systems by

- establishing, maintaining, and controlling an accurate, valid, and current performance measurement baseline, which would include negotiating all authorized, unpriced work within 3 months;
- conducting an integrated baseline review of any major contract modifications within 6 months; and
- preparing a rigorous life-cycle cost estimate, including a risk assessment, in accordance with the Acquisition System Toolset's guidance and identifying the level of uncertainty inherent in the estimate.

## Microarchitecture

In computer engineering, microarchitecture (sometimes abbreviated to μarch or uarch), also called computer organization, is the way a given instruction set architecture (ISA) is implemented on a processor. A given ISA may be implemented with different microarchitectures. Implementations might vary due to different goals of a given design or due to shifts in technology. Computer architecture is the combination of microarchitecture and instruction set design.

## Relation to Instruction Set Architecture

The ISA is roughly the same as the programming model of a processor as seen by an assembly language programmer or compiler writer. The ISA includes the execution model, processor registers, address and data formats among other things. The microarchitecture includes the constituent parts of the processor and how these interconnect and interoperate to implement the ISA. The microarchitecture of a machine is usually represented as (more or less detailed) diagrams that describe the interconnections of the various microarchitectural elements of the machine, which may be everything from single gates and registers, to complete arithmetic logic units (ALU)s and even larger elements. These diagrams generally separate the data path (where data is placed) and the control path (which can be said to steer the data).

Each microarchitectural element is in turn represented by a schematic describing the interconnections of logic gates used to implement it. Each logic gate is in turn represented by a circuit diagram describing the connections of the transistors used to implement it in some particular logic family. Machines with different microarchitectures may have the same instruction set architecture, and thus be capable of executing the same programmes. New microarchitectures and/or circuitry solutions, along with advances in semiconductor manufacturing, are what allows newer generations of processors to achieve higher performance while using the same ISA. In principle, a single microarchitecture could execute several different ISAs with only minor changes to the microcode.

## Aspects of Microarchitecture

The pipelined datapath is the most commonly used datapath design in microarchitecture today. This technique is used in most modern microprocessors, microcontrollers, and DSPs. The pipelined architecture allows multiple instructions to overlap in execution, much like an assembly line. The pipeline includes several different stages which are fundamental in microarchitecture designs. Some of these stages include instruction fetch, instruction decode, execute, and write back. Some architectures include other stages such as memory access. The design of pipelines is one of the central microarchitectural tasks. Execution units are also essential to microarchitecture. Execution units include arithmetic logic units (ALU), floating point units (FPU), load/store units, branch prediction, and SIMD. These units perform the operations or calculations of the processor. The choice of the number of execution units, their latency and throughput is a central microarchitectural design task. The size, latency, throughput and connectivity of memories within the system are also microarchitectural decisions. System-level design decisions such as whether or not to include peripherals, such as memory controllers, can be considered part of the microarchitectural design process. This includes decisions on the performance-level and connectivity of these peripherals. Unlike architectural design, where achieving a specific performance level is the main goal, microarchitectural design pays closer attention to other constraints. Since microarchitecture design decisions directly affect what goes into a system, attention must be paid to such issues as:

- Chip area/cost

- Power consumption
- Logic complexity
- Ease of connectivity
- Manufacturability
- Ease of debugging
- Testability

## Microarchitectural Concepts

In general, all CPUs, single-chip microprocessors or multi-chip implementations run programmes by performing the following steps:

1. Read an instruction and decode it
2. Find any associated data that is needed to process the instruction
3. Process the instruction
4. Write the results out

Complicating this simple-looking series of steps is the fact that the memory hierarchy, which includes caching, main memory and non-volatile storage like hard disks, (where the programme instructions and data reside) has always been slower than the processor itself. Step (2) often introduces a lengthy (in CPU terms) delay while the data arrives over the computer bus. A considerable amount of research has been put into designs that avoid these delays as much as possible. Over the years, a central goal was to execute more instructions in parallel, thus increasing the effective execution speed of a programme. These efforts introduced complicated logic and circuit structures. Initially these techniques could only be implemented on expensive mainframes or supercomputers due to the amount of circuitry

needed for these techniques. As semiconductor manufacturing progressed, more and more of these techniques could be implemented on a single semiconductor chip. What follows is a survey of micro-architectural techniques that are common in modern CPUs.

## Instruction Set Choice

Instruction sets have shifted over the years, from originally very simple to sometimes very complex (in various respects). In recent years, load-store architectures, VLIW and EPIC types have been in fashion. Architectures that are dealing with data parallelism include SIMD and Vectors. Some labels used to denote classes of CPU architectures are not particularly descriptive, especially so the CISC label; many early designs retroactively denoted "CISC" are in fact significantly simpler than modern RISC processors (in several respects). However, the choice of instruction set architecture may greatly affect the complexity of implementing high performance devices.

The prominent strategy, used to develop the first RISC processors, was to simplify instructions to a minimum of individual semantic complexity combined with high encoding regularity and simplicity. Such uniform instructions were easily fetched, decoded and executed in a pipelined fashion and a simple strategy to reduce the number of logic levels in order to reach high operating frequencies; instruction cache-memories compensated for the higher operating frequency and inherently low code density while large register sets were used to factor out as much of the (slow) memory accesses as possible.

## Instruction Pipelining

One of the first, and most powerful, techniques to improve performance is the use of the instruction pipeline. Early processor designs would carry out all of the steps above for one instruction before moving onto the next. Large portions of the circuitry were left idle at any one step; for instance, the instruction decoding circuitry would be idle during execution and so on. Pipelines improve performance by allowing a number of instructions to work their way through the processor at the same time. In the same basic example, the processor would start to decode (step 1) a new instruction while the last one was waiting for results. This would allow up to four instructions to be "in flight" at one time, making the processor look four times as fast.

Although any one instruction takes just as long to complete (there are still four steps) the CPU as a whole "retires" instructions much faster and can be run at a much higher clock speed. RISC make pipelines smaller and much easier to construct by cleanly separating each stage of the instruction process and making them take the same amount of time — one cycle. The processor as a whole operates in an assembly line fashion, with instructions coming in one side and results out the other. Due to the reduced complexity of the Classic RISC pipeline, the pipelined core and an instruction cache could be placed on the same size die that would otherwise fit the core alone on a CISC design. This was the real reason that RISC was faster. Early designs like the SPARC and MIPS often ran over 10 times as fast as Intel and Motorola CISC solutions at the same clock speed and price. Pipelines are by no means limited to RISC designs. By

1986 the top-of-the-line VAX implementation (VAX 8800) was a heavily pipelined design, slightly predating the first commercial MIPS and SPARC designs. Most modern CPUs (even embedded CPUs) are now pipelined, and microcoded CPUs with no pipelining are seen only in the most area-constrained embedded processors. Large CISC machines, from the VAX 8800 to the modern Pentium 4 and Athlon, are implemented with both microcode and pipelines. Improvements in pipelining and caching are the two major microarchitectural advances that have enabled processor performance to keep pace with the circuit technology on which they are based.

## Cache

It was not long before improvements in chip manufacturing allowed for even more circuitry to be placed on the die, and designers started looking for ways to use it. One of the most common was to add an ever-increasing amount of cache memory on-die. Cache is simply very fast memory, memory that can be accessed in a few cycles as opposed to "many" needed to talk to main memory. The CPU includes a cache controller which automates reading and writing from the cache, if the data is already in the cache it simply "appears," whereas if it is not the processor is "stalled" while the cache controller reads it in. RISC designs started adding cache in the mid-to-late 1980s, often only 4 KB in total. This number grew over time, and typical CPUs now have at least 512 KB, while more powerful CPUs come with 1 or 2 or even 4, 6, 8 or 12 MB, organized in multiple levels of a memory hierarchy. Generally speaking, more cache means more performance,

due to reduced stalling. Caches and pipelines were a perfect match for each other. Previously, it didn't make much sense to build a pipeline that could run faster than the access latency of off-chip memory. Using on-chip cache memory instead, meant that a pipeline could run at the speed of the cache access latency, a much smaller length of time. This allowed the operating frequencies of processors to increase at a much faster rate than that of off-chip memory.

## Branch Prediction

One barrier to achieving higher performance through instruction-level parallelism stems from pipeline stalls and flushes due to branches. Normally, whether a conditional branch will be taken isn't known until late in the pipeline as conditional branches depend on results coming from a register. From the time that the processor's instruction decoder has figured out that it has encountered a conditional branch instruction to the time that the deciding register value can be read out, the pipeline needs to be stalled for several cycles, or if it's not and the branch is taken, the pipeline needs to be flushed. As clock speeds increase the depth of the pipeline increases with it, and some modern processors may have 20 stages or more. On average, every fifth instruction executed is a branch, so without any intervention, that's a high amount of stalling. Techniques such as branch prediction and speculative execution are used to lessen these branch penalties. Branch prediction is where the hardware makes educated guesses on whether a particular branch will be taken. In reality one side or the other of the branch will be called much more often than the other. Modern designs have rather complex statistical prediction systems, which

watch the results of past branches to predict the future with greater accuracy. The guess allows the hardware to prefetch instructions without waiting for the register read. Speculative execution is a further enhancement in which the code along the predicted path is not just prefetched but also executed before it is known whether the branch should be taken or not. This can yield better performance when the guess is good, with the risk of a huge penalty when the guess is bad because instructions need to be undone.

## Superscalar

Even with all of the added complexity and gates needed to support the concepts outlined above, improvements in semiconductor manufacturing soon allowed even more logic gates to be used. In the outline above the processor processes parts of a single instruction at a time. Computer programmes could be executed faster if multiple instructions were processed simultaneously. This is what superscalar processors achieve, by replicating functional units such as ALUs. The replication of functional units was only made possible when the die area of a single-issue processor no longer stretched the limits of what could be reliably manufactured. By the late 1980s, superscalar designs started to enter the market place. In modern designs it is common to find two load units, one store (many instructions have no results to store), two or more integer math units, two or more floating point units, and often a SIMD unit of some sort. The instruction issue logic grows in complexity by reading in a huge list of instructions from memory and handing them off to the different execution units that are idle at that point. The results are then collected and re-ordered at the end.

## Out-of-order Execution

The addition of caches reduces the frequency or duration of stalls due to waiting for data to be fetched from the memory hierarchy, but does not get rid of these stalls entirely. In early designs a *cache miss* would force the cache controller to stall the processor and wait. Of course there may be some other instruction in the programme whose data *is* available in the cache at that point. Out-of-order execution allows that ready instruction to be processed while an older instruction waits on the cache, then re-orders the results to make it appear that everything happened in the programmed order. This technique is also used to avoid other operand dependency stalls, such as an instruction awaiting a result from a long latency floating-point operation or other multi-cycle operations.

## Register Renaming

Register renaming refers to a technique used to avoid unnecessary serialized execution of programme instructions because of the reuse of the same registers by those instructions. Suppose we have two groups of instruction that will use the same register. One set of instructions is executed first to leave the register to the other set, but if the other set is assigned to a different similar register, both sets of instructions can be executed in parallel.

## Multiprocessing and Multithreading

Computer architects have become stymied by the growing mismatch in CPU operating frequencies and DRAM access times. None of the techniques that exploited instruction-level parallelism within one programme could make up for the

long stalls that occurred when data had to be fetched from main memory. Additionally, the large transistor counts and high operating frequencies needed for the more advanced ILP techniques required power dissipation levels that could no longer be cheaply cooled. For these reasons, newer generations of computers have started to exploit higher levels of parallelism that exist outside of a single programme or programme thread. This trend is sometimes known as *throughput computing*. This idea originated in the mainframe market where online transaction processing emphasized not just the execution speed of one transaction, but the capacity to deal with massive numbers of transactions. With transaction-based applications such as network routing and web-site serving greatly increasing in the last decade, the computer industry has re-emphasized capacity and throughput issues. One technique of how this parallelism is achieved is through multiprocessing systems, computer systems with multiple CPUs. Once reserved for high-end mainframes and supercomputers, small scale (2-8) multiprocessors servers have become commonplace for the small business market. For large corporations, large scale (16-256) multiprocessors are common. Even personal computers with multiple CPUs have appeared since the 1990s.

With further transistor size reductions made available with semiconductor technology advances, multicore CPUs have appeared where multiple CPUs are implemented on the same silicon chip. Initially used in chips targeting embedded markets, where simpler and smaller CPUs would allow multiple instantiations to fit on one piece of silicon. By 2005,

semiconductor technology allowed dual high-end desktop CPUs *CMP* chips to be manufactured in volume. Some designs, such as Sun Microsystems' UltraSPARC T1 have reverted back to simpler (scalar, in-order) designs in order to fit more processors on one piece of silicon.

Another technique that has become more popular recently is multithreading. In multithreading, when the processor has to fetch data from slow system memory, instead of stalling for the data to arrive, the processor switches to another programme or programme thread which is ready to execute. Though this does not speed up a particular program/thread, it increases the overall system throughput by reducing the time the CPU is idle. Conceptually, multithreading is equivalent to a context switch at the operating system level. The difference is that a multithreaded CPU can do a thread switch in one CPU cycle instead of the hundreds or thousands of CPU cycles a context switch normally requires. This is achieved by replicating the state hardware (such as the register file and programme counter) for each active thread. A further enhancement is simultaneous multithreading. This technique allows superscalar CPUs to execute instructions from different programmes/threads simultaneously in the same cycle.

## Operating System

An operating system (OS) is software, consisting of programmes and data, that runs on computers and manages computer hardware resources and provides common services for efficient execution of various application software. For hardware functions such as input and output and memory allocation, the operating system acts as an intermediary

between application programmes and the computer hardware, although the application code is usually executed directly by the hardware and will frequently call the OS or be interrupted by it. Operating systems are found on almost any device that contains a computer—from cellular phones and video game consoles to supercomputers and web servers. Examples of popular modern operating systems for personal computers are (in alphabetical order): GNU/Linux, Mac OS X, Microsoft Windows and Unix

## Types of Operating Systems

Real-time Operating System: It is a multitasking operating system that aims at executing real-time applications. Real-time operating systems often use specialized scheduling algorithms so that they can achieve a deterministic nature of behaviour. The main object of real-time operating systems is their quick and predictable response to events. They either have an event-driven or a time-sharing design. An event-driven system switches between tasks based on their priorities while time-sharing operating systems switch tasks based on clock interrupts. Multi-user and Single-user Operating Systems: The operating systems of this type allow a multiple users to access a computer system concurrently. Time-sharing system can be classified as multi-user systems as they enable a multiple user access to a computer through the sharing of time. Single-user operating systems, as opposed to a multi-user operating system, are usable by a single user at a time. Being able to have multiple accounts on a Windows operating system does not make it a multi-user system. Rather, only the network administrator is the real user. But for a Unix-like operating system, it is possible

for two users to login at a time and this capability of the OS makes it a multi-user operating system. Multi-tasking and Single-tasking Operating Systems: When a single programme is allowed to run at a time, the system is grouped under a single-tasking system, while in case the operating system allows the execution of multiple tasks at one time, it is classified as a multi-tasking operating system. Multi-tasking can be of two types namely, pre-emptive or co-operative. In pre-emptive multitasking, the operating system slices the CPU time and dedicates one slot to each of the programmes. Unix-like operating systems such as Solaris and Linux support pre-emptive multitasking. Cooperative multitasking is achieved by relying on each process to give time to the other processes in a defined manner. MS Windows prior to Windows 95 used to support cooperative multitasking.

Distributed Operating System: An operating system that manages a group of independent computers and makes them appear to be a single computer is known as a distributed operating system. The development of networked computers that could be linked and communicate with each other, gave rise to distributed computing. Distributed computations are carried out on more than one machine. When computers in a group work in cooperation, they make a distributed system. Embedded System: The operating systems designed for being used in embedded computer systems are known as embedded operating systems. They are designed to operate on small machines like PDAs with less autonomy. They are able to operate with a limited number of resources. They are very compact and extremely efficient by design. Windows CE and Minix 3 are some examples of embedded operating systems.

## Summary

Early computers were built to perform a series of single tasks, like a calculator. Operating systems did not exist in their modern and more complex forms until the early 1960s. Some operating system features were developed in the 1950s, such as monitor programmes that could automatically run different application programmes in succession to speed up processing. Hardware features were added that enabled use of runtime libraries, interrupts, and parallel processing. When personal computers by companies such as Apple Inc., Atari, IBM and Amiga became popular in the 1980s, vendors added operating system features that had previously become widely used on mainframe and mini computers. Later, many features such as graphical user interface were developed specifically for personal computer operating systems. An operating system consists of many parts. One of the most important components is the kernel, which controls low-level processes that the average user usually cannot see: it controls how memory is read and written, the order in which processes are executed, how information is received and sent by devices like the monitor, keyboard and mouse, and decides how to interpret information received from networks. The user interface is a component that interacts with the computer user directly, allowing them to control and use programmes. The user interface may be graphical with icons and a desktop, or textual, with a command line. Application programming interfaces provide services and code libraries that let applications developers write modular code reusing well defined programming sequences in user space libraries or in the operating system itself. Which features are considered

part of the operating system is defined differently in various operating systems. For example, Microsoft Windows considers its user interface to be part of the operating system, while many versions of Linux do not.

## History

In the early 1950s, a computer could execute only one programme at a time. Each user had sole use of the computer and would arrive at a scheduled time with programme and data on punched paper cards and tape. The programme would be loaded into the machine, and the machine would be set to work until the programme completed or crashed. Programmes could generally be debugged via a front panel using toggle switches and panel lights. It is said that Alan Turing was a master of this on the early Manchester Mark 1 machine, and he was already deriving the primitive conception of an operating system from the principles of the Universal Turing machine.

Later machines came with libraries of software, which would be linked to a user's programme to assist in operations such as input and output and generating computer code from human-readable symbolic code. This was the genesis of the modern-day operating system. However, machines still ran a single job at a time. At Cambridge University in England the job queue was at one time a washing line from which tapes were hung with different colored clothes-pegs to indicate job-priority.

## Mainframes

Through the 1950s, many major features were pioneered in the field of operating systems, including batch processing,

input/output interrupt, buffering, multitasking, spooling, runtime libraries, link-loading, and programmes for sorting records in files. These features were included or not included in application software at the option of application programmers, rather than in a separate operating system used by all applications. In 1959 the SHARE Operating System was released as an integrated utility for the IBM 704, and later in the 709 and 7090 mainframes. During the 1960s, IBM's OS/360 introduced the concept of a single OS spanning an entire product line, which was crucial for the success of the System/360 machines. IBM's current mainframe operating systems are distant descendants of this original system and applications written for OS/360 can still be run on modern machines. In the mid-'70s, MVS, a descendant of OS/360, offered the first implementation of using RAM as a transparent cache for data. OS/360 also pioneered the concept that the operating system keeps track of all of the system resources that are used, including programme and data space allocation in main memory and file space in secondary storage, and file locking during update. When the process is terminated for any reason, all of these resources are re-claimed by the operating system.

The alternative CP-67 system for the S/360-67 started a whole line of IBM operating systems focused on the concept of virtual machines. Other operating systems used on IBM S/360 series mainframes included systems developed by IBM: COS/360 (Compatabililty Operating System), DOS/360 (Disk Operating System), TSS/360 (Time Sharing System), TOS/360 (Tape Operating System), BOS/360 (Basic Operating System), and ACP (Airline Control Programme), as well as a

few non-IBM systems: MTS (Michigan Terminal System) and MUSIC (Multi-User System for Interactive Computing). Control Data Corporation developed the SCOPE operating system in the 1960s, for batch processing. In cooperation with the University of Minnesota, the KRONOS and later the NOS operating systems were developed during the 1970s, which supported simultaneous batch and timesharing use. Like many commercial timesharing systems, its interface was an extension of the Dartmouth BASIC operating systems, one of the pioneering efforts in timesharing and programming languages. In the late 1970s, Control Data and the University of Illinois developed the PLATO operating system, which used plasma panel displays and long-distance time sharing networks. Plato was remarkably innovative for its time, featuring real-time chat, and multi-user graphical games. Burroughs Corporation introduced the B5000 in 1961 with the MCP, (Master Control Programme) operating system. The B5000 was a stack machine designed to exclusively support high-level languages with no machine language or assembler, and indeed the MCP was the first OS to be written exclusively in a high-level language – ESPOL, a dialect of ALGOL. MCP also introduced many other ground-breaking innovations, such as being the first commercial implementation of virtual memory. During development of the AS400, IBM made an approach to Burroughs to licence MCP to run on the AS400 hardware. This proposal was declined by Burroughs management to protect its existing hardware production. MCP is still in use today in the Unisys ClearPath/MCP line of computers.

UNIVAC, the first commercial computer manufacturer, produced a series of EXEC operating systems. Like all early

main-frame systems, this was a batch-oriented system that managed magnetic drums, disks, card readers and line printers. In the 1970s, UNIVAC produced the Real-Time Basic (RTB) system to support large-scale time sharing, also patterned after the Dartmouth BC system. General Electric and MIT developed General Electric Comprehensive Operating Supervisor (GECOS), which introduced the concept of ringed security privilege levels. After acquisition by Honeywell it was renamed to General Comprehensive Operating System (GCOS). Digital Equipment Corporation developed many operating systems for its various computer lines, including TOPS-10 and TOPS-20 time sharing systems for the 36-bit PDP-10 class systems. Prior to the widespread use of UNIX, TOPS-10 was a particularly popular system in universities, and in the early ARPANET community. In the late 1960s through the late 1970s, several hardware capabilities evolved that allowed similar or ported software to run on more than one system. Early systems had utilized microprogramming to implement features on their systems in order to permit different underlying architecture to appear to be the same as others in a series. In fact most 360s after the 360/40 (except the 360/165 and 360/168) were microprogrammed implementations. But soon other means of achieving application compatibility were proven to be more significant. The enormous investment in software for these systems made since 1960s caused most of the original computer manufacturers to continue to develop compatible operating systems along with the hardware. The notable supported mainframe operating systems include:

- Burroughs MCP – B5000, 1961 to Unisys Clearpath/MCP, present.

- IBM OS/360 – IBM System/360, 1966 to IBM z/OS, present.
- IBM CP-67 – IBM System/360, 1967 to IBM z/VM, present.
- UNIVAC EXEC 8 – UNIVAC 1108, 1967, to OS 2200 Unisys Clearpath Dorado, present.

## Microcomputers

The first microcomputers did not have the capacity or need for the elaborate operating systems that had been developed for mainframes and minis; minimalistic operating systems were developed, often loaded from ROM and known as *Monitors*. One notable early disk-based operating system was CP/M, which was supported on many early microcomputers and was closely imitated in MS-DOS, which became wildly popular as the operating system chosen for the IBM PC (IBM's version of it was called IBM DOS or PC DOS), its successors making Microsoft. In the '80s Apple Computer Inc. (now Apple Inc.) abandoned its popular Apple II series of microcomputers to introduce the Apple Macintosh computer with an innovative Graphical User Interface (GUI) to the Mac OS. The introduction of the Intel 80386 CPU chip with 32-bit architecture and paging capabilities, provided personal computers with the ability to run multitasking operating systems like those of earlier minicomputers and mainframes. Microsoft responded to this progress by hiring Dave Cutler, who had developed the VMS operating system for Digital Equipment Corporation. He would lead the development of the Windows NT operating system, which continues to serve as the basis for Microsoft's operating systems line. Steve Jobs, a co-founder of Apple Inc., started NeXT Computer Inc., which

developed the Unix-like NEXTSTEP operating system. NEXTSTEP would later be acquired by Apple Inc. and used, along with code from FreeBSD as the core of Mac OS X.

The GNU project was started by activist and programmer Richard Stallman with the goal of a complete free software replacement to the proprietary UNIX operating system. While the project was highly successful in duplicating the functionality of various parts of UNIX, development of the GNU Hurd kernel proved to be unproductive. In 1991, Finnish computer science student Linus Torvalds, with cooperation from volunteers collaborating over the Internet, released the first version of the Linux kernel. It was soon merged with the GNU user space components and system software to form a complete operating system. Since then, the combination of the two major components has usually been referred to as simply "Linux" by the software industry, a naming convention that Stallman and the Free Software Foundation remain opposed to, preferring the name GNU/Linux. The Berkeley Software Distribution, known as BSD, is the UNIX derivative distributed by the University of California, Berkeley, starting in the 1970s. Freely distributed and ported to many minicomputers, it eventually also gained a following for use on PCs, mainly as FreeBSD, NetBSD and OpenBSD.