Applications of Computer Software Languages

Bill Norris

APPLICATIONS OF COMPUTER SOFTWARE LANGUAGES

APPLICATIONS OF COMPUTER SOFTWARE LANGUAGES

Bill Norris

Applications of Computer Software Languages
by Bill Norris

Copyright© 2022 BIBLIOTEX

www.bibliotex.com

All rights reserved. No part of this book may be reproduced or used in any manner without the prior written permission of the copyright owner, except for the use brief quotations in a book review.

To request permissions, contact the publisher at info@bibliotex.com

Ebook ISBN: 9781984664013

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

Contents

Chapter 1	Introduction	1
Chapter 2	Computer Software	20
Chapter 3	Computer-aided Software Engineering	30
Chapter 4	Unified Modeling Language	45
Chapter 5	Software Progrmming	85
Chapter 6	Computer Programming Language	98
Chapter 7	Measuring Programming Language Popularity	152

1

Introduction

The term computer language includes a wide variety of languages used to communicate with computers. It is broader than the more commonly-used term programming language. Programming languages are a subset of computer languages. For example, HTML is a markup language and a computer language, but it is not traditionally considered a programming language. Machine code is a computer language. It can technically be used for programming, and has been (e.g. the original bootstrapped for Altair BASIC), though most would not consider it a programming language.

Computer languages can be divided into two groups: high-level languages and low-level languages. High-level languages are designed to be easier to use, more abstract, and more portable than low-level languages. Syntactically correct programs in some languages are then compiled to low-level language and executed by the computer. Most

modern software is written in a high-level language, compiled into object code, and then translated into machine instructions.

Computer languages could also be grouped based on other criteria. Another distinction could be made between human-readable and non-human-readable languages. Human-readable languages are designed to be used directly by humans to communicate with the computer. Non-human-readable languages, though they can often be partially understandable, are designed to be more compact and easily processed, sacrificing readability to meet these ends.

TYPES OF COMPUTER LANGUAGES

Language can be categories broadly into three categories.

Machine Language

The most elementary and first type of computer, which was invented, was machine language. Machine language was machine dependent. A programme written in machine language cannot be run on another type of computer without significant alterations. Machine language is some times also referred as the binary language i-e, the language of 0 and 1 where 0 stands for the absence of electric pulse and i stands for the presence of electric pulse. Very few computer programs are actually written in machine language.

Assembly Language

As computer became more popular, it became quite apparent that machine language programming was simply too slow slow tedious for most programmers. Assembly languages are also called as low level language instead of

using the string of members programmers began using English like abbreviation to represent the elementary operation. The language provided an opportunity to the programmers to use English like words that were called MNEMONICS.

High Level Language

The assembly languages started using English like words,m but still it was difficult to learn these languages. High level languages are the computer language in which it is much easier to write a programme than the low level language. A programme written in high level language is just like gibing instruction to person in daily life. It was in 1957 that a high level language called FORTRAN was developed by IBM which was specially developed for scientist and engineers other high level languages are COBOL which is widely used for business data processing task.BASIC language which is developed for the beginners in general purpose programming language. you Can use C language for almost any programming task. PASCAL are other high level languages which has gained widespread acceptance.

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.

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.

2

Computer Software

Computer software, or just software, is a collection of computer programmes and related data that provide the instructions telling a computer what to do and how to do it. We can also say software refers to one or more computer programmes and data held in the storage of the computer for some purposes. In other words software is a set of programmes, procedures, algorithms and its documentation. Programme software performs the function of the programme it implements, either by directly providing instructions to the computer hardware or by serving as input to another piece of software. The term was coined to contrast to the old term hardware (meaning physical devices). In contrast to hardware, software is intangible, meaning it "cannot be touched". Software is also sometimes used in a more narrow sense, meaning application software only. Sometimes the term includes data that has not traditionally been associated

with computers, such as film, tapes, and records. Examples of computer software include:

- Application software includes end-user applications of computers such as word processors or video games, and ERP software for groups of users.
- Middleware controls and co-ordinates distributed systems.
- Programming languages define the syntax and semantics of computer programmes. For example, many mature banking applications were written in the COBOL language, originally invented in 1959. Newer applications are often written in more modern programming languages.
- System software includes operating systems, which govern computing resources. Today large applications running on remote machines such as Websites are considered to be system software, because the end-user interface is generally through a graphical user interface, such as a web browser.
- Testware is software for testing hardware or a software package.
- Firmware is low-level software often stored on electrically programmable memory devices. Firmware is given its name because it is treated like hardware and run ("executed") by other software programmes.
- Shrinkware is the older name given to consumer-purchased software, because it was often sold in retail stores in a shrink-wrapped box.
- Device drivers control parts of computers such as disk drives, printers, CD drives, or computer monitors.

- Programming tools help conduct computing tasks in any category listed above. For programmers, these could be tools for debugging or reverse engineering older legacy systems in order to check source code compatibility.

History

The first theory about software was proposed by Alan Turing in his 1935 essay *Computable numbers with an application to the Entscheidungsproblem (Decision problem).* The term "software" was first used in print by John W. Tukey in 1958. Colloquially, the term is often used to mean application software. In computer science and software engineering, software is all information processed by computer system, programmes and data. The academic fields studying software are computer science and software engineering. The history of computer software is most often traced back to the first software bug in 1946. As more and more programmes enter the realm of firmware, and the hardware itself becomes smaller, cheaper and faster as predicted by Moore's law, elements of computing first considered to be software, join the ranks of hardware. Most hardware companies today have more software programmers on the payroll than hardware designers, since software tools have automated many tasks of Printed circuit board engineers. Just like the Auto industry, the Software industry has grown from a few visionaries operating out of their garage with prototypes. Steve Jobs and Bill Gates were the Henry Ford and Louis Chevrolet of their times, who capitalized on ideas already commonly known before they

started in the business. In the case of Software development, this moment is generally agreed to be the publication in the 1980s of the specifications for the IBM Personal Computer published by IBM employee Philip Don Estridge. Today his move would be seen as a type of crowd-sourcing.

Until that time, software was *bundled* with the hardware by Original equipment manufacturers (OEMs) such as Data General, Digital Equipment and IBM. When a customer bought a minicomputer, at that time the smallest computer on the market, the computer did not come with Pre-installed software, but needed to be installed by engineers employed by the OEM. Computer hardware companies not only bundled their software, they also placed demands on the location of the hardware in a refrigerated space called a computer room. Most companies had their software on the books for 0 dollars, unable to claim it as an asset (this is similar to financing of popular music in those days). When Data General introduced the Data General Nova, a company called Digidyne wanted to use its RDOS operating system on its own hardware clone. Data General refused to license their software (which was hard to do, since it was on the books as a free asset), and claimed their "bundling rights".

The Supreme Court set a precedent called Digidyne v. Data General in 1985. The Supreme Court let a 9th circuit decision stand, and Data General was eventually forced into licensing the Operating System software because it was ruled that restricting the license to only DG hardware was an illegal *tying arrangement*. Soon after, IBM 'published' its DOS source for free, and Microsoft was born. Unable to

sustain the loss from lawyer's fees, Data General ended up being taken over by EMC Corporation. The Supreme Court decision made it possible to value software, and also purchase Software patents. The move by IBM was almost a protest at the time. Few in the industry believed that anyone would profit from it other than IBM (through free publicity). Microsoft and Apple were able to thus cash in on 'soft' products. It is hard to imagine today that people once felt that software was worthless without a machine. There are many successful companies today that sell only software products, though there are still many common software licensing problems due to the complexity of designs and poor documentation, leading to patent trolls. With open software specifications and the possibility of software licensing, new opportunities arose for software tools that then became the de facto standard, such as DOS for operating systems, but also various proprietary word processing and spreadsheet programmes. In a similar growth pattern, proprietary development methods became standard Software development methodology.

Application Software

Application software, also known as an application or an "app", is computer software designed to help the user to perform singular or multiple related specific tasks. Examples include enterprize software, accounting software, office suites, graphics software and media players. Many application programmes deal principally with documents. Application software is contrasted with system software and middleware, which manage and integrate a computer's

capabilities, but typically do not directly apply them in the performance of tasks that benefit the user. A simple, if imperfect, analogy in the world of hardware would be the relationship of an electric light bulb (an application) to an electric power generation plant (a system). The power station merely generates electricity, not itself of any real use until harnessed to an application like the electric light that performs a service that benefits the user. Application software applies the power of a particular computing platform or system software to a particular purpose. Some apps such as Microsoft Office are available in versions for several different platforms; others have narrower requirements.

Terminology

In information technology, an application is a computer programme designed to help people perform an activity. An application thus differs from an operating system (which runs a computer), a utility (which performs maintenance or general-purpose chores), and a programming language (with which computer programmes are created). Depending on the activity for which it was designed, an application can manipulate text, numbers, graphics, or a combination of these elements. Some application packages offer considerable computing power by focusing on a single task, such as word processing; others, called integrated software, offer somewhat less power but include several applications. User-written software tailors systems to meet the user's specific needs. User-written software include spreadsheet templates, word processor macros, scientific simulations, graphics and animation scripts. Even email filters are a kind of user

software. Users create this software themselves and often overlook how important it is.

The delineation between system software such as operating systems and application software is not exact, however, and is occasionally the object of controversy. For example, one of the key questions in the United States v. Microsoft antitrust trial was whether Microsoft's Internet Explorer web browser was part of its Windows operating system or a separable piece of application software. As another example, the GNU/Linux naming controversy is, in part, due to disagreement about the relationship between the Linux kernel and the operating systems built over this kernel. In some types of embedded systems, the application software and the operating system software may be indistinguishable to the user, as in the case of software used to control a VCR, DVD player or microwave oven. The above definitions may exclude some applications that may exist on some computers in large organizations. For an alternative definition of an app: *see Application Portfolio Management.*

Application Software Classification

Application software falls into two general categories; horizontal applications and vertical applications. Horizontal Application are the most popular and its widely spread in departments or companies. Vertical Applications are designed for a particular type of business or for specific division in a company. There are many types of application software:

- An *application suite* consists of multiple applications bundled together. They usually have related functions,

features and user interfaces, and may be able to interact with each other, e.g. open each other's files. Business applications often come in suites, e.g. Microsoft Office, OpenOffice.org and iWork, which bundle together a word processor, a spreadsheet, etc.; but suites exist for other purposes, e.g. graphics or music.

- *Enterprize software* addresses the needs of organization processes and data flow, often in a large distributed environment. (Examples include financial systems, customer relationship management (CRM) systems and supply-chain management software). Note that Departmental Software is a sub-type of Enterprize Software with a focus on smaller organizations or groups within a large organization. (Examples include Travel Expense Management and IT Helpdesk)
- *Enterprize infrastructure software* provides common capabilities needed to support enterprize software systems. (Examples include databases, email servers, and systems for managing networks and security.)
- *Information worker software* addresses the needs of individuals to create and manage information, often for individual projects within a department, in contrast to enterprize management. Examples include time management, resource management, documentation tools, analytical, and collaborative. Word processors, spreadsheets, email and blog clients, personal information system, and individual media editors may aid in multiple information worker tasks.

- *Content access software* is software used primarily to access content without editing, but may include software that allows for content editing. Such software addresses the needs of individuals and groups to consume digital entertainment and published digital content. (Examples include Media Players, Web Browsers, Help browsers and Games)
- *Educational software* is related to content access software, but has the content and/or features adapted for use in by educators or students. For example, it may deliver evaluations (tests), track progress through material, or include collaborative capabilities.
- *Simulation software* are computer software for simulation of physical or abstract systems for either research, training or entertainment purposes.
- *Media development software* addresses the needs of individuals who generate print and electronic media for others to consume, most often in a commercial or educational setting. This includes Graphic Art software, Desktop Publishing software, Multimedia Development software, HTML editors, Digital Animation editors, Digital Audio and Video composition, and many others.
- *Mobile applications* run on hand-held devices such as mobile phones, personal digital assistants and enterprize digital assistants : see mobile application development.
- *Product engineering software* is used in developing hardware and software products. This includes

computer aided design (CAD), computer aided engineering (CAE), computer language editing and compiling tools, Integrated Development Environments, and Application Programmer Interfaces.

- A command-driven interface is one in which you type in commands to make the computer do something. You have to know the commands and what they do and they have to be typed correctly. DOS and Unix are examples of command-driven interfaces.
- A graphical user interface (GUI) is one in which you select command choices from various menus, buttons and icons using a mouse. It is a user-friendly interface. The Windows and Mac OS are both graphical user interfaces.

3

Computer-aided Software Engineering

Computer-aided software engineering (CASE), in the field software engineering is the scientific application of a set of tools and methods to a software which results in high-quality, defect-free, and maintainable software products. It also refers to methods for the development of information systems together with automated tools that can be used in the software development process.

The term "computer-aided software engineering" (CASE) can refer to the software used for the automated development of systems software, i.e., computer code. The CASE functions include analysis, design, and programming. CASE tools automate methods for designing, documenting, and producing structured computer code in the desired programming language.

Two key ideas of Computer-aided Software System Engineering (CASE) are:

- Foster computer assistance in software development and or software maintenance processes, and
- An engineering approach to software development and or maintenance.

Typical CASE tools exist for configuration management, data modeling, model transformation, refactoring, source code generation.

INTEGRATED DEVELOPMENT ENVIRONMENT

An integrated development environment (IDE) also known as *integrated design environment* or *integrated debugging environment* is a software application that provides comprehensive facilities to computer programmersfor software development. An IDE normally consists of a:

- source code editor,
- compiler and/or interpreter,
- build automation tools, and
- debugger (usually).

IDEs are designed to maximize programmer productivity by providing tight-knit components with similar user interfaces. Typically an IDE is dedicated to a specific programming language, so as to provide a feature set which most closely matches the programming paradigms of the language.

MODELING LANGUAGE

A modeling language is any artificial language that can be used to express information or knowledge or systems in

a structure that is defined by a consistent set of rules. The rules are used for interpretation of the meaning of components in the structure. A modeling language can be graphical or textual. Graphical modeling languages use a diagram techniques with named symbols that represent concepts and lines that connect the symbols and that represent relationships and various other graphical annotation to represent constraints. Textual modeling languages typically use standardised keywords accompanied by parameters to make computer-interpretable expressions.

Example of graphical modelling languages in the field of software engineering are:

- Business Process Modeling Notation (BPMN, and the XML form BPML) is an example of a process modeling language.
- EXPRESS and EXPRESS-G (ISO 10303-11) is an international standard general-purpose data modeling language.
- Extended Enterprise Modeling Language (EEML) is commonly used for business process modeling across layers.
- Flowchart is a schematic representation of an algorithm or a stepwise process,
- Fundamental Modeling Concepts (FMC) modeling language for software-intensive systems.
- IDEF is a family of modeling languages, the most notable of which include IDEF0 for functional modeling, IDEF1X for information modeling, and IDEF5 for modeling ontologies.

- LePUS3 is an object-oriented visual Design Description Language and a formal specification language that is suitable primarily for modelling large object-oriented (Java,C++, C#) programs and design patterns.
- Specification and Description Language(SDL) is a specification language targeted at the unambiguous specification and description of the behaviour of reactive and distributed systems.
- Unified Modeling Language (UML) is a general-purpose modeling language that is an industry standard for specifying software-intensive systems. UML 2.0, the current version, supports thirteen different diagram techniques, and has widespread tool support.

Not all modeling languages are executable, and for those that are, using them doesn't necessarily mean that programmers are no longer needed. On the contrary, executable modeling languages are intended to amplify the productivity of skilled programmers, so that they can address more difficult problems, such as parallel computing and distributed systems.

PROGRAMMING PARADIGM

A programming paradigm is a fundamental style of computer programming, which is not generally dictated by the project management methodology (such as waterfall or agile). Paradigms differ in the concepts and abstractions used to represent the elements of a program (such as objects, functions, variables, constraints) and the steps that comprise a computation (such as assignations, evaluation, continuations, data flows). Sometimes the

concepts asserted by the paradigm are utilized cooperatively in high-level system architecture design; in other cases, the programming paradigm's scope is limited to the internal structure of a particular program or module.

A programming language can support multiple paradigms. For example programs written in C++ or Object Pascal can be purely procedural, or purely object-oriented, or contain elements of both paradigms.

Software designers and programmers decide how to use those paradigm elements. In object-oriented programming, programmers can think of a program as a collection of interacting objects, while in functional programming a program can be thought of as a sequence of stateless function evaluations.

When programming computers or systems with many processors, process-oriented programming allows programmers to think about applications as sets of concurrent processes acting upon logically shared data structures.

Just as different groups in software engineering advocate different *methodologies*, different programming languages advocate different *programming paradigms*. Some languages are designed to support one paradigm (Smalltalk supports object-oriented programming, Haskell supports functional programming), while other programming languages support multiple paradigms (such as Object Pascal, C++, C#, Visual Basic, Common Lisp, Scheme, Python, Ruby, and Oz).

Many programming paradigms are as well known for what methods they *forbid* as for what they enable. For

instance, pure functional programming forbids using side-effects;structured programming forbids using goto statements. Partly for this reason, new paradigms are often regarded as doctrinaire or overly rigid by those accustomed to earlier styles. Avoiding certain methods can make it easier to prove theorems about a program's correctness, or simply to understand its behavior.

Examples of high-level paradigms include:

- Aspect-oriented software development
- Domain-specific modeling
- Model-driven engineering
- Object-oriented programming methodologies
- Grady Booch's object-oriented design (OOD), also known as object-oriented analysis and design (OOAD). The Booch model includes six diagrams: class, object, state transition, interaction, module, and process.
- Search-based software engineering
- Service-oriented modeling
- Structured programming
- Top-down and bottom-up design
- Top-down programming: evolved in the 1970s by IBM researcher Harlan Mills (and Niklaus Wirth) in developed structured programming.

SOFTWARE FRAMEWORK

A software framework is a re-usable design for a software system or subsystem. A software framework may include support programs, code libraries, a scripting language, or other software to help develop and *glue together* the different

components of a software project. Various parts of the framework may be exposed via an API.

Software Development

Software development is the computer programming, documenting, testing, and bug fixing involved in creating and maintainingapplications and frameworks involved in a software release life cycle and resulting in a software product. The term refers to a process of writing and maintaining the source code, but in a broader sense of the term it includes all that is involved between the conception of the desired software through to the final manifestation of the software, ideally in a planned and structured process. Therefore, software development may include research, new development, prototyping, modification, reuse, re-engineering, maintenance, or any other activities that result in software products.

Software can be developed for a variety of purposes, the three most common being to meet specific needs of a specific client/business (the case with custom software), to meet a perceived need of some set of potential users (the case with commercialand open source software), or for personal use (e.g. a scientist may write software to automate a mundane task). Embedded software development, that is, the development of embedded software such as used for controlling consumer products, requires the development process to be integrated with the development of the controlled physical product. System software underlies applications and the programming process itself, and is often developed separately.

The need for better quality control of the software development process has given rise to the discipline of software engineering, which aims to apply the systematic approach exemplified in the engineering paradigm to the process of software development.

There are many approaches to software project management, known as software development life cycle models, methodologies, processes, or models. The waterfall model is a traditional version, contrasted with the more recent innovation of agile software development.

METHODOLOGIES

A software development methodology (also known as a software development process, model, or life cycle) is a framework that is used to structure, plan, and control the process of developing information systems. A wide variety of such frameworks have evolved over the years, each with its own recognized strengths and weaknesses. There are several different approaches to software development: some take a more structured, engineering-based approach to developing business solutions, whereas others may take a more incremental approach, where software evolves as it is developed piece-by-piece. One system development methodology is not necessarily suitable for use by all projects. Each of the available methodologies is best suited to specific kinds of projects, based on various technical, organizational, project and team considerations.

Most methodologies share some combination of the following stages of software development:

- Analyzing the problem

- Market research
- Gathering requirements for the proposed business solution
- Devising a plan or design for the software-based solution
- Implementation (coding) of the software
- Testing the software
- Deployment
- Maintenance and bug fixing.

These stages are often referred to collectively as the software development lifecycle, or SDLC. Different approaches to software development may carry out these stages in different orders, or devote more or less time to different stages. The level of detail of the documentation produced at each stage of software development may also vary. These stages may also be carried out in turn (a "waterfall" based approach), or they may be repeated over various cycles or iterations (a more "extreme" approach). The more extreme approach usually involves less time spent on planning and documentation, and more time spent on coding and development of automated tests. More "extreme" approaches also promote continuous testing throughout the development lifecycle, as well as having a working (or bug-free) product at all times. More structured or "waterfall" based approaches attempt to assess the majority of risks and develop a detailed plan for the software before implementation(coding) begins, and avoid significant design changes and re-coding in later stages of the software development life cycle planning.

There are significant advantages and disadvantages to the various methodologies, and the best approach to solving a problem using software will often depend on the type of problem. If the problem is well understood and a solution can be effectively planned out ahead of time, the more "waterfall" based approach may work the best. If, on the other hand, the problem is unique (at least to the development team) and the structure of the software solution cannot be easily envisioned, then a more "extreme" incremental approach may work best.

Software Development Activities

IDENTIFICATION OF NEED

The sources of ideas for software products are legion. These ideas can come from market research including the demographics of potential new customers, existing customers, sales prospects who rejected the product, other internal software development staff, or a creative third party. Ideas for software products are usually first evaluated by marketing personnel for economic feasibility, for fit with existing channels distribution, for possible effects on existing product lines, required features, and for fit with the company's marketing objectives. In a marketing evaluation phase, the cost and time assumptions become evaluated. A decision is reached early in the first phase as to whether, based on the more detailed information generated by the marketing and development staff, the project should be pursued further.

Students of engineering learn engineering and are rarely exposed to finance or marketing. Students of marketing

learn marketing and are rarely exposed to finance or engineering. Most of us become specialists in just one area. To complicate matters, few of us meet interdisciplinary people in the workforce, so there are few roles to mimic. Yet, software product planning is critical to the development success and absolutely requires knowledge of multiple disciplines.

Because software development may involve compromising or going beyond what is required by the client, a software development project may stray into less technical concerns such as human resources, risk management, intellectual property, budgeting, crisis management, etc. These processes may also cause the role of business development to overlap with software development.

PLANNING

Planning is an objective of each and every activity, where we want to discover things that belong to the project. An important task in creating a software program is extracting therequirements or requirements analysis.

Customers typically have an abstract idea of what they want as an end result, but do not know what *software* should do. Skilled and experienced software engineers recognize incomplete, ambiguous, or even contradictory requirements 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.

DESIGNING

Once the requirements are established, the design of the software can be established in a software design document. This involves a preliminary, or high-level design of the main modules with an overall picture (such as a block diagram) of how the parts fit together. The language, operating system, and hardware components should all be known at this time. Then a detailed or low-level design is created, perhaps with prototyping as proof-of-concept or to firm up requirements.

Implementation, Testing and Documenting

Implementation is the part of the process where software engineers actually program the code for the project.

Software testing is an integral and important phase of the software development process. This part of the process ensures that defects are recognized as soon as possible. In some processes, generally known as test-driven development, tests may be developed just before implementation and serve as a guide for the implementation's correctness.

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 anAPI, be it external or internal. The software engineering process chosen by the developing team will determine how

much internal documentation (if any) is necessary. Plan-driven models (e.g., Waterfall) generally produce more documentation than Agile models.

DEPLOYMENT AND MAINTENANCE

Deployment starts directly after the code is appropriately tested, approved for release, and sold or otherwise distributed into a production environment. This may involve installation, customization (such as by setting parameters to the customer's values), testing, and possibly an extended period of evaluation.

Software training and support is important, as software is only effective if it is used correctly. Maintaining and enhancing software to cope with newly discovered faults or requirements can take substantial time and effort, as missed requirements may force redesign of the software.

Subtopics

VIEW MODEL

A view model is a framework that provides the viewpoints on the system and its environment, to be used in thesoftware development process. It is a graphical representation of the underlying semantics of a view.

The purpose of viewpoints and views is to enable human engineers to comprehend very complex systems, and to organize the elements of the problem and the solution around domains of expertise. In the engineering of physically intensive systems, viewpoints often correspond to capabilities and responsibilities within the engineering organization.

Most complex system specifications are so extensive that no one individual can fully comprehend all aspects of the

specifications. Furthermore, we all have different interests in a given system and different reasons for examining the system's specifications. A business executive will ask different questions of a system make-up than would a system implementer. The concept of viewpoints framework, therefore, is to provide separate viewpoints into the specification of a given complex system. These viewpoints each satisfy an audience with interest in some set of aspects of the system. Associated with each viewpoint is a viewpoint language that optimizes the vocabulary and presentation for the audience of that viewpoint.

BUSINESS PROCESS AND DATA MODELLING

Graphical representation of the current state of information provides a very effective means for presenting information to both users and system developers.

- A business model illustrates the functions associated with the business process being modeled and the organizations that perform these functions. By depicting activities and information flows, a foundation is created to visualize, define, understand, and validate the nature of a process.
- A data model provides the details of information to be stored, and is of primary use when the final product is the generation of computer software code for an application or the preparation of a functional specification to aid a computer software make-or-buy decision.

Usually, a model is created after conducting an interview, referred to as business analysis. The interview consists of

a facilitator asking a series of questions designed to extract required information that describes a process. The interviewer is called a facilitator to emphasize that it is the participants who provide the information.

The facilitator should have some knowledge of the process of interest, but this is not as important as having a structured methodology by which the questions are asked of the process expert. The methodology is important because usually a team of facilitators is collecting information across the facility and the results of the information from all the interviewers must fit together once completed.

The models are developed as defining either the current state of the process, in which case the final product is called the "as-is" snapshot model, or a collection of ideas of what the process should contain, resulting in a "what-can-be" model. Generation of process and data models can be used to determine if the existing processes and information systems are sound and only need minor modifications or enhancements, or if re-engineering is required as a corrective action. The creation of business models is more than a way to view or automate your information process. Analysis can be used to fundamentally reshape the way your business or organization conducts its operations.

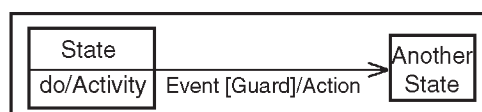
4

Unified Modeling Language

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.

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)

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).

5

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

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

PC-DOS and MS-DOS low memory

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

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

Command Shell

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

Programmes

- Desk Accessories

Software Characteristics

- 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)

6

Computer Programming Language

Computer programming (often shortened to programming or coding) is the process of designing, writing, testing, debugging / troubleshooting, and maintaining the source code of computer programmes. This source code is written in a programming language. The purpose of programming is to create a programme that exhibits a certain desired behaviour. The process of writing source code often requires expertise in many different subjects, including knowledge of the application domain, specialized algorithms and formal logic.

Definition

Hoc and Nguyen-Xuan define computer programming as "the process of transforming a mental plan in familiar terms into one compatible with the computer." Said another way, programming is the craft of transforming requirements into something that a computer can execute.

Overview

Within software engineering, programming (the *implementation*) is regarded as one phase in a software development process. There is an ongoing debate on the extent to which the writing of programmes is an art, a craft or an engineering discipline. In general, good programming is considered to be the measured application of all three, with the goal of producing an efficient and evolvable software solution (the criteria for "efficient" and "evolvable" vary considerably). The discipline differs from many other technical professions in that programmers, in general, do not need to be licensed or pass any standardized (or governmentally regulated) certification tests in order to call themselves "programmers" or even "software engineers."

However, representing oneself as a "Professional Software Engineer" without a license from an accredited institution is illegal in many parts of the world. However, because the discipline covers many areas, which may or may not include critical applications, it is debatable whether licensing is required for the profession as a whole. In most cases, the discipline is self-governed by the entities which require the programming, and sometimes very strict environments are defined (e.g. United States Air Force use of AdaCore and security clearance). Another ongoing debate is the extent to which the programming language used in writing computer programmes affects the form that the final programme takes. This debate is analogous to that surrounding the Sapir–Whorf hypothesis in linguistics, which postulates that a particular spoken language's nature influences the habitual thought of its speakers.

Different language patterns yield different patterns of thought. This idea challenges the possibility of representing the world perfectly with language, because it acknowledges that the mechanisms of any language condition the thoughts of its speaker community.

History

The Antikythera mechanism from ancient Greece was a calculator utilizing gears of various sizes and configuration to determine its operation, which tracked the metonic cycle still used in lunar-to-solar calendars, and which is consistent for calculating the dates of the Olympiads. Al-Jazari built programmable Automata in 1206. One system employed in these devices was the use of pegs and cams placed into a wooden drum at specific locations. which would sequentially trigger levers that in turn operated percussion instruments. The output of this device was a small drummer playing various rhythms and drum patterns. The Jacquard Loom, which Joseph Marie Jacquard developed in 1801, uses a series of pasteboard cards with holes punched in them. The hole pattern represented the pattern that the loom had to follow in weaving cloth. The loom could produce entirely different weaves using different sets of cards. Charles Babbage adopted the use of punched cards around 1830 to control his Analytical Engine. The synthesis of numerical calculation, predetermined operation and output, along with a way to organize and input instructions in a manner relatively easy for humans to conceive and produce, led to the modern development of computer programming. Development of computer programming accelerated through

the Industrial Revolution. In the late 1880s, Herman Hollerith invented the recording of data on a medium that could then be read by a machine. Prior uses of machine readable media, above, had been for control, not data. "After some initial trials with paper tape, he settled on punched cards..."

To process these punched cards, first known as "Hollerith cards" he invented the tabulator, and the keypunch machines. These three inventions were the foundation of the modern information processing industry. In 1896 he founded the *Tabulating Machine Company* (which later became the core of IBM). The addition of a control panel (plugboard) to his 1906 Type I Tabulator allowed it to do different jobs without having to be physically rebuilt. By the late 1940s, there were a variety of plug-board programmable machines, called unit record equipment, to perform data-processing tasks (card reading). Early computer programmers used plug-boards for the variety of complex calculations requested of the newly invented machines. The invention of the von Neumann architecture allowed computer programmes to be stored in computer memory.

Early programmes had to be painstakingly crafted using the instructions (elementary operations) of the particular machine, often in binary notation. Every model of computer would likely use different instructions (machine language) to do the same task. Later, assembly languages were developed that let the programmer specify each instruction in a text format, entering abbreviations for each operation code instead of a number and specifying addresses in symbolic form (e.g., ADD X, TOTAL). Entering a programme in assembly language is usually more convenient, faster,

and less prone to human error than using machine language, but because an assembly language is little more than a different notation for a machine language, any two machines with different instruction sets also have different assembly languages. In 1954, FORTRAN was invented; it was the first high level programming language to have a functional implementation, as opposed to just a design on paper. (A high-level language is, in very general terms, any programming language that allows the programmer to write programmes in terms that are more abstract than assembly language instructions, i.e. at a level of abstraction "higher" than that of an assembly language.) It allowed programmers to specify calculations by entering a formula directly (e.g. Y = X*2 + 5*X + 9). The programme text, or *source*, is converted into machine instructions using a special programme called a compiler, which translates the FORTRAN programme into machine language. In fact, the name FORTRAN stands for "Formula Translation". Many other languages were developed, including some for commercial programming, such as COBOL. Programmes were mostly still entered using punched cards or paper tape. By the late 1960s, data storage devices and computer terminals became inexpensive enough that programmes could be created by typing directly into the computers. Text editors were developed that allowed changes and corrections to be made much more easily than with punched cards. (Usually, an error in punching a card meant that the card had to be discarded and a new one punched to replace it.)

As time has progressed, computers have made giant leaps in the area of processing power. This has brought

about newer programming languages that are more abstracted from the underlying hardware. Although these high-level languages usually incur greater overhead, the increase in speed of modern computers has made the use of these languages much more practical than in the past. These increasingly abstracted languages typically are easier to learn and allow the programmer to develop applications much more efficiently and with less source code.

However, high-level languages are still impractical for a few programmes, such as those where low-level hardware control is necessary or where maximum processing speed is vital. Throughout the second half of the twentieth century, programming was an attractive career in most developed countries. Some forms of programming have been increasingly subject to offshore outsourcing (importing software and services from other countries, usually at a lower wage), making programming career decisions in developed countries more complicated, while increasing economic opportunities in less developed areas. It is unclear how far this trend will continue and how deeply it will impact programmer wages and opportunities.

History of programming languages

This article discusses the major developments in the history of programming languages. For a detailed timeline of events, see the timeline of programming languages.

Before 1940

The first programming languages predate the modern computer. At first, the languages were codes. The Jacquard loom, invented in 1801, used holes in punched cards to

represent sewing loom arm movements in order to generate decorative patterns automatically. During a nine-month period in 1842-1843, Ada Lovelace translated the memoir of Italian mathematician Luigi Menabrea about Charles Babbage's newest proposed machine, the Analytical Engine. With the article, she appended a set of notes which specified in complete detail a method for calculating Bernoulli numbers with the Engine, recognized by some historians as the world's first computer programme.

Herman Hollerith realized that he could encode information on punch cards when he observed that train conductors encode the appearance of the ticket holders on the train tickets using the position of punched holes on the tickets. Hollerith then encoded the 1890 census data on punch cards. The first computer codes were specialized for their applications. In the first decades of the 20th century, numerical calculations were based on decimal numbers. Eventually it was realized that logic could be represented with numbers, not only with words. For example, Alonzo Church was able to express the lambda calculus in a formulaic way. The Turing machine was an abstraction of the operation of a tape-marking machine, for example, in use at the telephone companies. Turing machines set the basis for storage of programmes as data in the von Neumann architecture of computers by representing a machine through a finite number. However, unlike the lambda calculus, Turing's code does not serve well as a basis for higher-level languages—its principal use is in rigorous analyses of algorithmic complexity. Like many "firsts" in history, the first modern programming language is hard to identify.

From the start, the restrictions of the hardware defined the language.

Punch cards allowed 80 columns, but some of the columns had to be used for a sorting number on each card. FORTRAN included some keywords which were the same as English words, such as "IF", "GOTO" (go to) and "CONTINUE". The use of a magnetic drum for memory meant that computer programmes also had to be interleaved with the rotations of the drum. Thus the programmes were more hardware-dependent. To some people, what was the first modern programming language depends on how much power and human-readability is required before the status of "programming language" is granted. Jacquard looms and Charles Babbage's Difference Engine both had simple, extremely limited languages for describing the actions that these machines should perform. One can even regard the punch holes on a player piano scroll as a limited domain-specific language, albeit not designed for human consumption.

The 1940s

In the 1940s, the first recognizably modern, electrically powered computers were created. The limited speed and memory capacity forced programmers to write hand tuned assembly language programmes. It was soon discovered that programming in assembly language required a great deal of intellectual effort and was error-prone. In 1948, Konrad Zuse published a paper about his programming language Plankalkül. However, it was not implemented in his lifetime and his original contributions were isolated

from other developments. Some important languages that were developed in this period include:

- 1943 - Plankalkül (Konrad Zuse), designed, but unimplemented for a half-century
- 1943 - ENIAC coding system, machine-specific codeset appearing in 1948
- 1949 - 1954 — a series of machine-specific mnemonic instruction sets, like ENIAC's, beginning in 1949 with C-10 for BINAC (which later evolved into UNIVAC). Each codeset, or instruction set, was tailored to a specific manufacturer.

The 1950s and 1960s

In the 1950s, the first three modern programming languages whose descendants are still in widespread use today were designed:

- FORTRAN (1955), the "FORmula TRANslator", invented by John Backus et al.;
- LISP [1958], the "LISt Processor", invented by John McCarthy et al.;
- COBOL, the COmmon Business Oriented Language, created by the Short Range Committee, heavily influenced by Grace Hopper.

Another milestone in the late 1950s was the publication, by a committee of American and European computer scientists, of "a new language for algorithms"; the *ALGOL 60 Report* (the "ALGOrithmic Language"). This report consolidated many ideas circulating at the time and featured two key language innovations:

- nested block structure: code sequences and associated declarations could be grouped into blocks without having to be turned into separate, explicitly named procedures;
- lexical scoping: a block could have its own private variables, procedures and functions, invisible to code outside that block, i.e. information hiding.

Another innovation, related to this, was in how the language was described:

- a mathematically exact notation, Backus-Naur Form (BNF), was used to describe the language's syntax. Nearly all subsequent programming languages have used a variant of BNF to describe the context-free portion of their syntax.

Algol 60 was particularly influential in the design of later languages, some of which soon became more popular. The Burroughs large systems were designed to be programmed in an extended subset of Algol. Algol's key ideas were continued, producing ALGOL 68:

- syntax and semantics became even more orthogonal, with anonymous routines, a recursive typing system with higher-order functions, etc.;
- not only the context-free part, but the full language syntax and semantics were defined formally, in terms of Van Wijngaarden grammar, a formalism designed specifically for this purpose.

Algol 68's many little-used language features (e.g. concurrent and parallel blocks) and its complex system of syntactic shortcuts and automatic type coercions made it

unpopular with implementers and gained it a reputation of being *difficult*. Niklaus Wirth actually walked out of the design committee to create the simpler Pascal language. Some important languages that were developed in this period include:

- 1951 - Regional Assembly Language
- 1952 - Autocode
- 1954 - FORTRAN
- 1954 - IPL (forerunner to LISP)
- 1955 - FLOW-MATIC (forerunner to COBOL)
- 1957 - COMTRAN (forerunner to COBOL)
- 1958 - LISP
- 1958 - ALGOL 58
- 1959 - FACT (forerunner to COBOL)
- 1959 - COBOL
- 1962 - APL
- 1962 - Simula
- 1962 - SNOBOL
- 1963 - CPL (forerunner to C)
- 1964 - BASIC
- 1964 - PL/I
- 1967 - BCPL (forerunner to C)

1967-1978: Establishing Fundamental Paradigms

The period from the late 1960s to the late 1970s brought a major flowering of programming languages. Most of the major language paradigms now in use were invented in this period:

- Simula, invented in the late 1960s by Nygaard and Dahl as a superset of Algol 60, was the first language designed to support object-oriented programming.
- C, an early systems programming language, was developed by Dennis Ritchie and Ken Thompson at Bell Labs between 1969 and 1973.
- Smalltalk (mid 1970s) provided a complete ground-up design of an object-oriented language.
- Prolog, designed in 1972 by Colmerauer, Roussel, and Kowalski, was the first logic programming language.
- ML built a polymorphic type system (invented by Robin Milner in 1973) on top of Lisp, pioneering statically typed functional programming languages.

Each of these languages spawned an entire family of descendants, and most modern languages count at least one of them in their ancestry. The 1960s and 1970s also saw considerable debate over the merits of "structured programming", which essentially meant programming without the use of Goto. This debate was closely related to language design: some languages did not include GOTO, which forced structured programming on the programmer. Although the debate raged hotly at the time, nearly all programmers now agree that, even in languages that provide GOTO, it is bad programming style to use it except in rare circumstances. As a result, later generations of language designers have found the structured programming debate tedious and even bewildering. Some important languages that were developed in this period include:

- 1968 - Logo
- 1969 - B (forerunner to C)
- 1970 - Pascal
- 1970 - Forth
- 1972 - C
- 1972 - Smalltalk
- 1972 - Prolog
- 1973 - ML
- 1975 - Scheme
- 1978 - SQL (initially only a query language, later extended with programming constructs)

The 1980s: Consolidation, Modules and Performance

The 1980s were years of relative consolidation in imperative languages. Rather than inventing new paradigms, all of these movements elaborated upon the ideas invented in the previous decade. C++ combined object-oriented and systems programming. The United States government standardized Ada, a systems programming language intended for use by defense contractors. In Japan and elsewhere, vast sums were spent investigating so-called fifth-generation programming languages that incorporated logic programming constructs. The functional languages community moved to standardize ML and Lisp. Research in Miranda, a functional language with lazy evaluation, began to take hold in this decade.

One important new trend in language design was an increased focus on programming for large-scale systems

through the use of *modules*, or large-scale organizational units of code. Modula, Ada, and ML all developed notable module systems in the 1980s. Module systems were often wedded to generic programming constructs—generics being, in essence, parameterized modules.

Although major new paradigms for imperative programming languages did not appear, many researchers expanded on the ideas of prior languages and adapted them to new contexts. For example, the languages of the Argus and Emerald systems adapted object-oriented programming to distributed systems.

The 1980s also brought advances in programming language implementation. The RISC movement in computer architecture postulated that hardware should be designed for compilers rather than for human assembly programmers. Aided by processor speed improvements that enabled increasingly aggressive compilation techniques, the RISC movement sparked greater interest in compilation technology for high-level languages. Language technology continued along these lines well into the 1990s. Some important languages that were developed in this period include:

- 1980 - C++ (as C with classes, name changed in July 1983)
- 1983 - Objective-C
- 1983 - Ada
- 1984 - Common Lisp
- 1985 - Eiffel
- 1986 - Erlang
- 1987 - Perl

- 1988 - Tcl
- 1989 - FL (Backus)

The 1990s: The Internet Age

The 1990s saw no fundamental novelty in imperative languages, but much recombination and maturation of old ideas. This era began the spread of functional languages. A big driving philosophy was programmer productivity. Many "rapid application development" (RAD) languages emerged, which usually came with an IDE, garbage collection, and were descendants of older languages. All such languages were object-oriented. These included Object Pascal, Visual Basic, and Java. Java in particular received much attention. More radical and innovative than the RAD languages were the new scripting languages. These did not directly descend from other languages and featured new syntaxes and more liberal incorporation of features.

Many consider these scripting languages to be more productive than even the RAD languages, but often because of choices that make small programmes simpler but large programmes more difficult to write and maintain. Nevertheless, scripting languages came to be the most prominent ones used in connection with the Web. Some important languages that were developed in this period include:

- 1990 - Haskell
- 1991 - Python
- 1991 - Visual Basic
- 1993 - Ruby
- 1993 - Lua

- 1994 - CLOS (part of ANSI Common Lisp)
- 1995 - Java
- 1995 - Delphi (Object Pascal)
- 1995 - JavaScript
- 1995 - PHP
- 1997 - Rebol
- 1999 - D

Current Trends

Programming language evolution continues, in both industry and research. Some of the current trends include:

- Mechanisms for adding security and reliability verification to the language: extended static checking, information flow control, static thread safety.
- Alternative mechanisms for modularity: mixins, delegates, aspects.
- Component-oriented software development.
- Constructs to support concurrent and distributed programming.
- Metaprogramming, reflection or access to the abstract syntax tree
- Increased emphasis on distribution and mobility.
- Integration with databases, including XML and relational databases.
- Support for Unicode so that source code (programme text) is not restricted to those characters contained in the ASCII character set; allowing, for example, use of non-Latin-based scripts or extended punctuation.
- XML for graphical interface (XUL, XAML).

Some important languages developed during this period include:

- 2001 - C#
- 2001 - Visual Basic .NET
- 2002 - F#
- 2003 - Scala
- 2003 - Factor
- 2006 - Windows Power Shell
- 2007 - Clojure
- 2007 - Groovy
- 2009 - Go

Prominent People in the History of Programming Languages

- John Backus, inventor of Fortran.
- Alan Cooper, developer of Visual Basic.
- Edsger W. Dijkstra, developed the framework for structured programming.
- James Gosling, developer of Oak, the precursor of Java.
- Anders Hejlsberg, developer of Turbo Pascal, Delphi and C#.
- Grace Hopper, developer of Flow-Matic, influencing COBOL.
- Kenneth E. Iverson, developer of APL, and co-developer of J along with Roger Hui.
- Bill Joy, inventor of vi, early author of BSD Unix, and originator of SunOS, which became Solaris.
- Alan Kay, pioneering work on object-oriented programming, and originator of Smalltalk.

- Brian Kernighan, co-author of the first book on the C programming language with Dennis Ritchie, coauthor of the AWK and AMPL programming languages.
- John McCarthy, inventor of LISP.
- John von Neumann, originator of the operating system concept.
- Dennis Ritchie, inventor of C (programming language). Unix Operating System, Plan 9 Operating System.
- Bjarne Stroustrup, developer of C++.
- Ken Thompson, inventor of B, Go Programming Language, Inferno Programming Language.
- Niklaus Wirth, inventor of Pascal, Modula and Oberon.
- Larry Wall, creator of Perl and Perl 6
- Guido van Rossum, creator of Python
- Yukihiro Matsumoto, creator of Ruby

MODERN PROGRAMMING

Quality Requirements

Whatever the approach to software development may be, the final programme must satisfy some fundamental properties. The following properties are among the most relevant:

- Efficiency/performance: the amount of system resources a programme consumes (processor time, memory space, slow devices such as disks, network bandwidth and to some extent even user interaction): the less, the better. This also includes correct disposal

of some resources, such as cleaning up temporary files and lack of memory leaks.

- Reliability: how often the results of a programme are correct. This depends on conceptual correctness of algorithms, and minimization of programming mistakes, such as mistakes in resource management (e.g., buffer overflows and race conditions) and logic errors (such as division by zero or off-by-one errors).
- Robustness: how well a programme anticipates problems not due to programmer error. This includes situations such as incorrect, inappropriate or corrupt data, unavailability of needed resources such as memory, operating system services and network connections, and user error.
- Usability: the ergonomics of a program: the ease with which a person can use the programme for its intended purpose, or in some cases even unanticipated purposes. Such issues can make or break its success even regardless of other issues. This involves a wide range of textual, graphical and sometimes hardware elements that improve the clarity, intuitiveness, cohesiveness and completeness of a program's user interface.
- Portability: the range of computer hardware and operating system platforms on which the source code of a programme can be compiled/interpreted and run. This depends on differences in the programming facilities provided by the different platforms, including hardware and operating system resources, expected behaviour of the hardware and operating system,

and availability of platform specific compilers (and sometimes libraries) for the language of the source code.

- Maintainability: the ease with which a programme can be modified by its present or future developers in order to make improvements or customizations, fix bugs and security holes, or adapt it to new environments. Good practices during initial development make the difference in this regard. This quality may not be directly apparent to the end user but it can significantly affect the fate of a programme over the long term.

Readability of Source Code

In computer programming, readability refers to the ease with which a human reader can comprehend the purpose, control flow, and operation of source code. It affects the aspects of quality above, including portability, usability and most importantly maintainability. Readability is important because programmers spend the majority of their time reading, trying to understand and modifying existing source code, rather than writing new source code.

Unreadable code often leads to bugs, inefficiencies, and duplicated code. A study found that a few simple readability transformations made code shorter and drastically reduced the time to understand it. Following a consistent programming style often helps readability.

However, readability is more than just programming style. Many factors, having little or nothing to do with the ability

of the computer to efficiently compile and execute the code, contribute to readability. Some of these factors include:

- Different indentation styles (whitespace)
- Comments
- Decomposition
- Naming conventions for objects (such as variables, classes, procedures, etc)

Algorithmic Complexity

The academic field and the engineering practice of computer programming are both largely concerned with discovering and implementing the most efficient algorithms for a given class of problem. For this purpose, algorithms are classified into *orders* using so-called Big O notation, *O(n)*, which expresses resource use, such as execution time or memory consumption, in terms of the size of an input. Expert programmers are familiar with a variety of well-established algorithms and their respective complexities and use this knowledge to choose algorithms that are best suited to the circumstances.

Methodologies

The first step in most formal software development projects is requirements analysis, followed by testing to determine value modeling, implementation, and failure elimination (debugging). There exist a lot of differing approaches for each of those tasks. One approach popular for requirements analysis is Use Case analysis.

Nowadays many programmers use forms of Agile software development where the various stages of formal software

development are more integrated together into short cycles that take a few weeks rather than years. There are many approaches to the Software development process. Popular modeling techniques include Object-Oriented Analysis and Design (OOAD) and Model-Driven Architecture (MDA). The Unified Modeling Language (UML) is a notation used for both the OOAD and MDA. A similar technique used for database design is Entity-Relationship Modeling (ER Modeling). Implementation techniques include imperative languages (object-oriented or procedural), functional languages, and logic languages.

Measuring Language Usage

It is very difficult to determine what are the most popular of modern programming languages. Some languages are very popular for particular kinds of applications (e.g., COBOL is still strong in the corporate data center, often on large mainframes, FORTRAN in engineering applications, scripting languages in web development, and C in embedded applications), while some languages are regularly used to write many different kinds of applications. Also many applications use a mix of several languages in their construction and use.

Methods of measuring programming language popularity include: counting the number of job advertisements that mention the language, the number of books teaching the language that are sold (this overestimates the importance of newer languages), and estimates of the number of existing lines of code written in the language (this underestimates the number of users of business languages such as COBOL).

Debugging

Debugging is a very important task in the software development process, because an incorrect programme can have significant consequences for its users. Some languages are more prone to some kinds of faults because their specification does not require compilers to perform as much checking as other languages. Use of a static analysis tool can help detect some possible problems.

Debugging is often done with IDEs like Eclipse, Kdevelop, NetBeans, Code::Blocks, and Visual Studio. Standalone debuggers like gdb are also used, and these often provide less of a visual environment, usually using a command line.

Programming Languages

Different programming languages support different styles of programming (called *programming paradigms*). The choice of language used is subject to many considerations, such as company policy, suitability to task, availability of third-party packages, or individual preference. Ideally, the programming language best suited for the task at hand will be selected. Trade-offs from this ideal involve finding enough programmers who know the language to build a team, the availability of compilers for that language, and the efficiency with which programmes written in a given language execute. Languages form an approximate spectrum from "low-level" to "high-level"; "low-level" languages are typically more machine-oriented and faster to execute, whereas "high-level" languages are more abstract and easier to use but execute less quickly. It is usually easier to code in "high-level" languages than in "low-level"

ones. Allen Downey, in his book *How To Think Like A Computer Scientist*, writes:

The details look different in different languages, but a few basic instructions appear in just about every language:

- input: Get data from the keyboard, a file, or some other device.
- output: Display data on the screen or send data to a file or other device.
- arithmetic: Perform basic arithmetical operations like addition and multiplication.
- conditional execution: Check for certain conditions and execute the appropriate sequence of statements.
- repetition: Perform some action repeatedly, usually with some variation.

Many computer languages provide a mechanism to call functions provided by libraries such as in .dlls. Provided the functions in a library follow the appropriate run time conventions (e.g., method of passing arguments), then these functions may be written in any other language.

Programmers

Computer programmers are those who write computer software. Their jobs usually involve:

- Coding
- Compilation
- Debugging
- Documentation
- Integration
- Maintenance

- Requirements analysis
- Software architecture
- Software testing
- Specification

Programming Language

A programming language is an artificial language designed to express computations that can be performed by a machine, particularly a computer. Programming languages can be used to create programmes that control the behavior of a machine, to express algorithms precisely, or as a mode of human communication. The earliest programming languages predate the invention of the computer, and were used to direct the behavior of machines such as Jacquard looms and player pianos. Thousands of different programming languages have been created, mainly in the computer field, with many more being created every year.

Most programming languages describe computation in an imperative style, i.e., as a sequence of commands, although some languages, such as those that support functional programming or logic programming, use alternative forms of description. A programming language is usually split into the two components of syntax (form) and semantics (meaning) and many programming languages have some kind of written specification of their syntax and/or semantics. Some languages are defined by a specification document, for example, the C programming language is specified by an ISO Standard, while other languages, such as Perl, have a dominant implementation that is used as a reference.

Definitions

A programming language is a notation for writing programmes, which are specifications of a computation or algorithm. Some, but not all, authors restrict the term "programming language" to those languages that can express *all* possible algorithms. Traits often considered important for what constitutes a programming language include:

- *Function and target:* A *computer programming language* is a language used to write computer programmes, which involve a computer performing some kind of computation or algorithm and possibly control external devices such as printers, disk drives, robots, and so on. For example PostScript programmes are frequently created by another programme to control a computer printer or display. More generally, a programming language may describe computation on some, possibly abstract, machine. It is generally accepted that a complete specification for a programming language includes a description, possibly idealized, of a machine or processor for that language. In most practical contexts, a programming language involves a computer; consequently programming languages are usually defined and studied this way. Programming languages differ from natural languages in that natural languages are only used for interaction between people, while programming languages also allow humans to communicate instructions to machines.
- *Abstractions:* Programming languages usually contain abstractions for defining and manipulating data

structures or controlling the flow of execution. The practical necessity that a programming language support adequate abstractions is expressed by the abstraction principle; this principle is sometimes formulated as recommendation to the programmer to make proper use of such abstractions.

- *Expressive power:* The theory of computation classifies languages by the computations they are capable of expressing. All Turing complete languages can implement the same set of algorithms. ANSI/ISO SQL and Charity are examples of languages that are not Turing complete, yet often called programming languages.

Markup languages like XML, HTML or troff, which define structured data, are not generally considered programming languages. Programming languages may, however, share the syntax with markup languages if a computational semantics is defined. XSLT, for example, is a Turing complete XML dialect. Moreover, LaTeX, which is mostly used for structuring documents, also contains a Turing complete subset.

The term *computer language* is sometimes used interchangeably with programming language. However, the usage of both terms varies among authors, including the exact scope of each. One usage describes programming languages as a subset of computer languages. In this vein, languages used in computing that have a different goal than expressing computer programmes are generically designated computer languages. For instance, markup languages are

sometimes referred to as computer languages to emphasize that they are not meant to be used for programming. Another usage regards programming languages as theoretical constructs for programming abstract machines, and computer languages as the subset thereof that runs on physical computers, which have finite hardware resources. John C. Reynolds emphasizes that formal specification languages are just as much programming languages as are the languages intended for execution. He also argues that textual and even graphical input formats that affect the behavior of a computer are programming languages, despite the fact they are commonly not Turing-complete, and remarks that ignorance of programming language concepts is the reason for many flaws in input formats.

Elements

All programming languages have some primitive building blocks for the description of data and the processes or transformations applied to them (like the addition of two numbers or the selection of an item from a collection). These primitives are defined by syntactic and semantic rules which describe their structure and meaning respectively.

Syntax

A programming language's surface form is known as its syntax. Most programming languages are purely textual; they use sequences of text including words, numbers, and punctuation, much like written natural languages. On the other hand, there are some programming languages which are more graphical in nature, using visual relationships between symbols to specify a programme.

The syntax of a language describes the possible combinations of symbols that form a syntactically correct programme. The meaning given to a combination of symbols is handled by semantics (either formal or hard-coded in a reference implementation). Since most languages are textual, this article discusses textual syntax. Programming language syntax is usually defined using a combination of regular expressions (for lexical structure) and Backus–Naur Form (for grammatical structure). Below is a simple grammar, based on Lisp:

expression ::= atom | list
atom ::= number | symbol
number ::= [+-]?['0'-'9']+
symbol ::= ['A'-'Z''a'-'z'].*
list ::= '(' expression* ')'

This grammar specifies the following:

- an *expression* is either an *atom* or a *list*;
- an *atom* is either a *number* or a *symbol*;
- a *number* is an unbroken sequence of one or more decimal digits, optionally preceded by a plus or minus sign;
- a *symbol* is a letter followed by zero or more of any characters (excluding whitespace); and
- a *list* is a matched pair of parentheses, with zero or more *expressions* inside it.

The following are examples of well-formed token sequences in this grammar: '12345', '()', '(a b c232 (1))'

Not all syntactically correct programmes are semantically correct. Many syntactically correct programmes are

nonetheless ill-formed, per the language's rules; and may (depending on the language specification and the soundness of the implementation) result in an error on translation or execution. In some cases, such programmes may exhibit undefined behavior. Even when a programme is well-defined within a language, it may still have a meaning that is not intended by the person who wrote it.

Using natural language as an example, it may not be possible to assign a meaning to a grammatically correct sentence or the sentence may be false:

- "Colorless green ideas sleep furiously." is grammatically well-formed but has no generally accepted meaning.
- "John is a married bachelor." is grammatically well-formed but expresses a meaning that cannot be true.

The following C language fragment is syntactically correct, but performs an operation that is not semantically defined (because p is a null pointer, the operations p->real and p->im have no meaning):

```
complex *p = NULL;
complex abs_p = sqrt (p->real * p->real + p->im * p->im);
```

If the type declaration on the first line were omitted, the programme would trigger an error on compilation, as the variable "p" would not be defined. But the programme would still be syntactically correct, since type declarations provide only semantic information.

The grammar needed to specify a programming language can be classified by its position in the Chomsky hierarchy. The syntax of most programming languages can be specified

using a Type-2 grammar, i.e., they are context-free grammars. Some languages, including Perl and Lisp, contain constructs that allow execution during the parsing phase. Languages that have constructs that allow the programmer to alter the behavior of the parser make syntax analysis an undecidable problem, and generally blur the distinction between parsing and execution. In contrast to Lisp's macro system and Perl's BEGIN blocks, which may contain general computations, C macros are merely string replacements, and do not require code execution.

Semantics

The term *semantics* refers to the meaning of languages, as opposed to their form (syntax).

Static Semantics

The static semantics defines restrictions on the structure of valid texts that are hard or impossible to express in standard syntactic formalisms. For compiled languages, static semantics essentially include those semantic rules that can be checked at compile time. Examples include checking that every identifier is declared before it is used (in languages that require such declarations) or that the labels on the arms of a case statement are distinct. Many important restrictions of this type, like checking that identifiers are used in the appropriate context (e.g. not adding a integer to a function name), or that subroutine calls have the appropriate number and type of arguments can be enforced by defining them as rules in a logic called a type system. Other forms of static analyses like data flow analysis may also be part of static semantics. Newer

programming languages like Java and C# have definite assignment analysis, a form of data flow analysis, as part of their static semantics.

Dynamic Semantics

Once data has been specified, the machine must be instructed to perform operations on the data. For example, the semantics may define the strategy by which expressions are evaluated to values, or the manner in which control structures conditionally execute statements. The *dynamic semantics* (also known as *execution semantics*) of a language defines how and when the various constructs of a language should produce a programme behavior. There are many ways of defining execution semantics. Natural language is often used to specify the execution semantics of languages commonly used in practice. A significant amount of academic research went into formal semantics of programming languages, which allow execution semantics to be specified in a formal manner. Results from this field of research have seen limited application to programming language design and implementation outside academia.

Type System

A type system defines how a programming language classifies values and expressions into *types*, how it can manipulate those types and how they interact. The goal of a type system is to verify and usually enforce a certain level of correctness in programmes written in that language by detecting certain incorrect operations. Any decidable type system involves a trade-off: while it rejects many incorrect programmes, it can also prohibit some correct, albeit unusual

programmes. In order to bypass this downside, a number of languages have *type loopholes*, usually unchecked casts that may be used by the programmer to explicitly allow a normally disallowed operation between different types. In most typed languages, the type system is used only to type check programmes, but a number of languages, usually functional ones, infer types, relieving the programmer from the need to write type annotations. The formal design and study of type systems is known as *type theory*.

Typed versus untyped languages: A language is *typed* if the specification of every operation defines types of data to which the operation is applicable, with the implication that it is not applicable to other types. For example, the data represented by "this text between the quotes" is a string. In most programming languages, dividing a number by a string has no meaning. Most modern programming languages will therefore reject any programme attempting to perform such an operation. In some languages, the meaningless operation will be detected when the programme is compiled ("static" type checking), and rejected by the compiler, while in others, it will be detected when the programme is run ("dynamic" type checking), resulting in a runtime exception. A special case of typed languages are the *single-type* languages. These are often scripting or markup languages, such as REXX or SGML, and have only one data type—most commonly character strings which are used for both symbolic and numeric data. In contrast, an *untyped language*, such as most assembly languages, allows any operation to be performed on any data, which are generally considered to be sequences of bits of various lengths. High-level languages

which are untyped include BCPL and some varieties of Forth. In practice, while few languages are considered typed from the point of view of type theory (verifying or rejecting *all* operations), most modern languages offer a degree of typing. Many production languages provide means to bypass or subvert the type system.

Static versus dynamic typing: In *static typing* all expressions have their types determined prior to the programme being run (typically at compile-time). For example, 1 and (2+2) are integer expressions; they cannot be passed to a function that expects a string, or stored in a variable that is defined to hold dates.

Statically typed languages can be either *manifestly typed* or *type-inferred*. In the first case, the programmer must explicitly write types at certain textual positions (for example, at variable declarations). In the second case, the compiler *infers* the types of expressions and declarations based on context.

Most mainstream statically typed languages, such as C++, C# and Java, are manifestly typed. Complete type inference has traditionally been associated with less mainstream languages, such as Haskell and ML. However, many manifestly typed languages support partial type inference; for example, Java and C# both infer types in certain limited cases.

Dynamic typing, also called *latent typing*, determines the type-safety of operations at runtime; in other words, types are associated with *runtime values* rather than *textual expressions*. As with type-inferred languages, dynamically

typed languages do not require the programmer to write explicit type annotations on expressions. Among other things, this may permit a single variable to refer to values of different types at different points in the programme execution. However, type errors cannot be automatically detected until a piece of code is actually executed, potentially making debugging more difficult. Ruby, Lisp, JavaScript, and Python are dynamically typed.

Weak and strong typing: Weak typing allows a value of one type to be treated as another, for example treating a string as a number. This can occasionally be useful, but it can also allow some kinds of programme faults to go undetected at compile time and even at runtime.

Strong typing prevents the above. An attempt to perform an operation on the wrong type of value raises an error. Strongly typed languages are often termed *type-safe* or *safe*.

An alternative definition for "weakly typed" refers to languages, such as Perl and JavaScript, which permit a large number of implicit type conversions. In JavaScript, for example, the expression 2 * x implicitly converts x to a number, and this conversion succeeds even if x is null, undefined, an Array, or a string of letters. Such implicit conversions are often useful, but they can mask programming errors. *Strong* and *static* are now generally considered orthogonal concepts, but usage in the literature differs. Some use the term *strongly typed* to mean *strongly, statically typed*, or, even more confusingly, to mean simply *statically typed*. Thus C has been called both strongly typed and weakly, statically typed.

Standard Library and Run-time System

Most programming languages have an associated core library (sometimes known as the 'standard library', especially if it is included as part of the published language standard), which is conventionally made available by all implementations of the language. Core libraries typically include definitions for commonly used algorithms, data structures, and mechanisms for input and output.

A language's core library is often treated as part of the language by its users, although the designers may have treated it as a separate entity. Many language specifications define a core that must be made available in all implementations, and in the case of standardized languages this core library may be required. The line between a language and its core library therefore differs from language to language. Indeed, some languages are designed so that the meanings of certain syntactic constructs cannot even be described without referring to the core library. For example, in Java, a string literal is defined as an instance of the java.lang.String class; similarly, in Smalltalk, an anonymous function expression (a "block") constructs an instance of the library's BlockContext class. Conversely, Scheme contains multiple coherent subsets that suffice to construct the rest of the language as library macros, and so the language designers do not even bother to say which portions of the language must be implemented as language constructs, and which must be implemented as parts of a library.

Design and Implementation

Programming languages share properties with natural

languages related to their purpose as vehicles for communication, having a syntactic form separate from its semantics, and showing *language families* of related languages branching one from another. But as artificial constructs, they also differ in fundamental ways from languages that have evolved through usage. A significant difference is that a programming language can be fully described and studied in its entirety, since it has a precise and finite definition. By contrast, natural languages have changing meanings given by their users in different communities. While constructed languages are also artificial languages designed from the ground up with a specific purpose, they lack the precise and complete semantic definition that a programming language has.

Many languages have been designed from scratch, altered to meet new needs, combined with other languages, and eventually fallen into disuse. Although there have been attempts to design one "universal" programming language that serves all purposes, all of them have failed to be generally accepted as filling this role. The need for diverse programming languages arises from the diversity of contexts in which languages are used:

- Programmes range from tiny scripts written by individual hobbyists to huge systems written by hundreds of programmers.
- Programmers range in expertise from novices who need simplicity above all else, to experts who may be comfortable with considerable complexity.
- Programmes must balance speed, size, and simplicity

on systems ranging from microcontrollers to supercomputers.

- Programmes may be written once and not change for generations, or they may undergo continual modification.
- Finally, programmers may simply differ in their tastes: they may be accustomed to discussing problems and expressing them in a particular language.

One common trend in the development of programming languages has been to add more ability to solve problems using a higher level of abstraction. The earliest programming languages were tied very closely to the underlying hardware of the computer.

As new programming languages have developed, features have been added that let programmers express ideas that are more remote from simple translation into underlying hardware instructions. Because programmers are less tied to the complexity of the computer, their programmes can do more computing with less effort from the programmer. This lets them write more functionality per time unit. Natural language processors have been proposed as a way to eliminate the need for a specialized language for programming.

However, this goal remains distant and its benefits are open to debate. Edsger W. Dijkstra took the position that the use of a formal language is essential to prevent the introduction of meaningless constructs, and dismissed natural language programming as “foolish”. Alan Perlis was similarly dismissive of the idea. Hybrid approaches have

been taken in Structured English and SQL. A language's designers and users must construct a number of artifacts that govern and enable the practice of programming. The most important of these artifacts are the language *specification* and *implementation*.

Specification

The specification of a programming language is intended to provide a definition that the language users and the implementors can use to determine whether the behavior of a programme is correct, given its source code. A programming language specification can take several forms, including the following:

- An explicit definition of the syntax, static semantics, and execution semantics of the language. While syntax is commonly specified using a formal grammar, semantic definitions may be written in natural language (e.g., as in the C language), or a formal semantics (e.g., as in Standard ML and Scheme specifications).
- A description of the behavior of a translator for the language (e.g., the C++ and Fortran specifications). The syntax and semantics of the language have to be inferred from this description, which may be written in natural or a formal language.
- A *reference* or *model* implementation, sometimes written in the language being specified (e.g., Prolog or ANSI REXX). The syntax and semantics of the language are explicit in the behavior of the reference implementation.

Implementation

An implementation of a programming language provides a way to execute that programme on one or more configurations of hardware and software. There are, broadly, two approaches to programming language implementation: *compilation* and *interpretation*. It is generally possible to implement a language using either technique. The output of a compiler may be executed by hardware or a programme called an interpreter. In some implementations that make use of the interpreter approach there is no distinct boundary between compiling and interpreting. For instance, some implementations of BASIC compile and then execute the source a line at a time. Programmes that are executed directly on the hardware usually run several orders of magnitude faster than those that are interpreted in software. One technique for improving the performance of interpreted programmes is just-in-time compilation. Here the virtual machine, just before execution, translates the blocks of bytecode which are going to be used to machine code, for direct execution on the hardware.

Usage

Thousands of different programming languages have been created, mainly in the computing field. Programming languages differ from most other forms of human expression in that they require a greater degree of precision and completeness. When using a natural language to communicate with other people, human authors and speakers can be ambiguous and make small errors, and still expect their intent to be understood. However, figuratively

speaking, computers "do exactly what they are told to do", and cannot "understand" what code the programmer intended to write. The combination of the language definition, a programme, and the program's inputs must fully specify the external behavior that occurs when the programme is executed, within the domain of control of that programme. A programming language provides a structured mechanism for defining pieces of data, and the operations or transformations that may be carried out automatically on that data. A programmer uses the abstractions present in the language to represent the concepts involved in a computation.

These concepts are represented as a collection of the simplest elements available (called primitives). *Programming* is the process by which programmers combine these primitives to compose new programmes, or adapt existing ones to new uses or a changing environment. Programmes for a computer might be executed in a batch process without human interaction, or a user might type commands in an interactive session of an interpreter. In this case the "commands" are simply programmes, whose execution is chained together. When a language is used to give commands to a software application (such as a shell) it is called a scripting language.

Measuring Language Usage

It is difficult to determine which programming languages are most widely used, and what usage means varies by context. One language may occupy the greater number of programmer hours, a different one have more lines of code,

and a third utilize the most CPU time. Some languages are very popular for particular kinds of applications. For example, COBOL is still strong in the corporate data center, often on large mainframes; FORTRAN in scientific and engineering applications; C in embedded applications and operating systems; and other languages are regularly used to write many different kinds of applications.

Various methods of measuring language popularity, each subject to a different bias over what is measured, have been proposed:

- counting the number of job advertisements that mention the language
- the number of books sold that teach or describe the language
- estimates of the number of existing lines of code written in the language—which may underestimate languages not often found in public searches
- counts of language references (i.e., to the name of the language) found using a web search engine.

Combining and averaging information from various internet sites, langpop.com claims that in 2008 the 10 most cited programming languages are (in alphabetical order): C, C++, C#, Java, JavaScript, Perl, PHP, Python, Ruby, and SQL.

Taxonomies

There is no overarching classification scheme for programming languages. A given programming language does not usually have a single ancestor language. Languages commonly arise by combining the elements of several

predecessor languages with new ideas in circulation at the time. Ideas that originate in one language will diffuse throughout a family of related languages, and then leap suddenly across familial gaps to appear in an entirely different family. The task is further complicated by the fact that languages can be classified along multiple axes.

For example, Java is both an object-oriented language (because it encourages object-oriented organization) and a concurrent language (because it contains built-in constructs for running multiple threads in parallel). Python is an object-oriented scripting language. In broad strokes, programming languages divide into *programming paradigms* and a classification by *intended domain of use*. Traditionally, programming languages have been regarded as describing computation in terms of imperative sentences, i.e. issuing commands. These are generally called imperative programming languages. A great deal of research in programming languages has been aimed at blurring the distinction between a programme as a set of instructions and a programme as an assertion about the desired answer, which is the main feature of declarative programming. More refined paradigms include procedural programming, object-oriented programming, functional programming, and logic programming; some languages are hybrids of paradigms or multi-paradigmatic. An assembly language is not so much a paradigm as a direct model of an underlying machine architecture. By purpose, programming languages might be considered general purpose, system programming languages, scripting languages, domain-specific languages, or concurrent/distributed languages (or a combination of these).

Some general purpose languages were designed largely with educational goals. A programming language may also be classified by factors unrelated to programming paradigm. For instance, most programming languages use English language keywords, while a minority do not. Other languages may be classified as being esoteric or not.

HISTORY

Early developments

The first programming languages predate the modern computer. The 19th century had "programmable" looms and player piano scrolls which implemented what are today recognized as examples of domain-specific languages. By the beginning of the twentieth century, punch cards encoded data and directed mechanical processing. In the 1930s and 1940s, the formalisms of Alonzo Church's lambda calculus and Alan Turing's Turing machines provided mathematical abstractions for expressing algorithms; the lambda calculus remains influential in language design. In the 1940s, the first electrically powered digital computers were created.

The first high-level programming language to be designed for a computer was Plankalkül, developed for the German Z3 by Konrad Zuse between 1943 and 1945. However, it was not implemented until 1998 and 2000. Programmers of early 1950s computers, notably UNIVAC I and IBM 701, used machine language programmes, that is, the first generation language (1GL). 1GL programming was quickly superseded by similarly machine-specific, but mnemonic, second generation languages (2GL) known as assembly

languages or "assembler". Later in the 1950s, assembly language programming, which had evolved to include the use of macro instructions, was followed by the development of "third generation" programming languages (3GL), such as FORTRAN, LISP, and COBOL. 3GLs are more abstract and are "portable", or at least implemented similarly on computers that do not support the same native machine code. Updated versions of all of these 3GLs are still in general use, and each has strongly influenced the development of later languages. At the end of the 1950s, the language formalized as ALGOL 60 was introduced, and most later programming languages are, in many respects, descendants of Algol. The format and use of the early programming languages was heavily influenced by the constraints of the interface.

Refinement

The period from the 1960s to the late 1970s brought the development of the major language paradigms now in use, though many aspects were refinements of ideas in the very first Third-generation programming languages:

- APL introduced *array programming* and influenced functional programming.
- PL/I (NPL) was designed in the early 1960s to incorporate the best ideas from FORTRAN and COBOL.
- In the 1960s, Simula was the first language designed to support *object-oriented programming*; in the mid-1970s, Smalltalk followed with the first "purely" object-oriented language.

- C was developed between 1969 and 1973 as a *system programming* language, and remains popular.
- Prolog, designed in 1972, was the first *logic programming* language.
- In 1978, ML built a polymorphic type system on top of Lisp, pioneering *statically typed functional programming* languages.

Each of these languages spawned an entire family of descendants, and most modern languages count at least one of them in their ancestry.

The 1960s and 1970s also saw considerable debate over the merits of *structured programming*, and whether programming languages should be designed to support it. Edsger Dijkstra, in a famous 1968 letter published in the Communications of the ACM, argued that GOTO statements should be eliminated from all "higher level" programming languages.

The 1960s and 1970s also saw expansion of techniques that reduced the footprint of a programme as well as improved productivity of the programmer and user. The card deck for an early 4GL was a lot smaller for the same functionality expressed in a 3GL deck.

Consolidation and Growth

The 1980s were years of relative consolidation. C++ combined object-oriented and systems programming. The United States government standardized Ada, a systems programming language derived from Pascal and intended for use by defense contractors. In Japan and elsewhere, vast sums were spent investigating so-called "fifth generation"

languages that incorporated logic programming constructs. The functional languages community moved to standardize ML and Lisp. Rather than inventing new paradigms, all of these movements elaborated upon the ideas invented in the previous decade.

One important trend in language design for programming large-scale systems during the 1980s was an increased focus on the use of *modules*, or large-scale organizational units of code. Modula-2, Ada, and ML all developed notable module systems in the 1980s, although other languages, such as PL/I, already had extensive support for modular programming. Module systems were often wedded to generic programming constructs.

The rapid growth of the Internet in the mid-1990s created opportunities for new languages. Perl, originally a Unix scripting tool first released in 1987, became common in dynamic websites. Java came to be used for server-side programming, and bytecode virtual machines became popular again in commercial settings with their promise of "Write once, run anywhere" (UCSD Pascal had been popular for a time in the early 1980s). These developments were not fundamentally novel, rather they were refinements to existing languages and paradigms, and largely based on the C family of programming languages.

Programming language evolution continues, in both industry and research. Current directions include security and reliability verification, new kinds of modularity (mixins, delegates, aspects), and database integration such as Microsoft's LINQ. The 4GLs are examples of languages which

are domain-specific, such as SQL, which manipulates and returns sets of data rather than the scalar values which are canonical to most programming languages. Perl, for example, with its 'here document' can hold multiple 4GL programmes, as well as multiple JavaScript programmes, in part of its own perl code and use variable interpolation in the 'here document' to support multi-language programming.

Programming Language Generations

Programming languages have been classified into several programming language generations. Historically, this classification was used to indicate increasing power of programming styles. Later writers have somewhat redefined the meanings as distinctions previously seen as important became less significant to current practice.

Historical View of First Three Generations

The terms "first-generation" and "second-generation" programming language were not used prior to the coining of the term "third-generation." In fact, none of these three terms are mentioned in an early compendium of programming language.The introduction of a third generation of computer technology coincided with the creation of a new generation of programming languages. The marketing for this generational shift in machines did correlate with several important changes in what were called high level programming languages, discussed below, giving technical content to the second/third-generation distinction among high level programming languages as well, and reflexively renaming assembler languages as first-generation.

First Generation

As Grace Hopper said about coding in machine language: "We were not programmers in those days. The word had not yet come over from England. We were coders.

The task of encoding an algorithm wasn't thought of as writing in a language any more than was the task of wiring a plug-board. But even by the early 1950s, the assembly languages were seen as a distinct "epoch". The distinguishing properties of these first generation programming languages are that:

- The code can be read and written by a programmer. To run on a computer it must be converted into a machine readable form, a process called assembly.
- The language is specific to a particular target machine or family of machines, directly reflecting their characteristics like instruction sets, registers, storage access models, etc., requiring and enabling the programmer to manage their use.
- Some assembler languages provide a macro-facility enabling the development of complex patterns of machine instructions, but these are not considered to change the basic nature of the language.

First-generation languages are sometimes used in kernels and device drivers, but more often find use in extremely intensive processing such as games, video editing, and graphic manipulation/rendering.

Second Generation

Second-generation programming languages, originally just called high level programming languages, were created to

simplify the burden of programming by making its expression more like the normal mode of expression for thoughts used by the programmer. They were introduced in the late 1950s, with FORTRAN reflecting the needs of scientific programmers, ALGOL reflecting an attempt to produce a European/ American standard view.

The most important issue faced by the developers of second-level languages was convincing customers that the code produced by the compilers performed well-enough to justify abandonment of assembler programming.In view of the widespread skepticism about the possibility of producing efficient programmes with an automatic programming system and the fact that inefficiencies could no longer be hidden, we were convinced that the kind of system we had in mind would be widely used only if we could demonstrate that it would produce programmes almost as efficient as hand coded ones and do so on virtually every job.The FORTRAN compiler was seen as a tour-de-force in the production of high-quality code, even including "... a Monte Carlo simulation of its execution ... so as to minimize the transfers of items between the store and the index registers Second-generation programming languages evolved through the decade.

FORTRAN lost some of its machine-dependent features, like access to the lights and switches on the operator console. Most second-generation languages employed a static storage model in which storage for data was allocated only once, when a programme is loaded, making recursion difficult, but Algol evolved to provide block-structured naming

constructs and began to expand the set of features made available to programmers, like concurrency management. In this way (Algol 68) began the movement into a new generation of programming languages.

Third Generation

The introduction of a third generation of computer technology coincided with the creation of a new generation of programming languages. The third-generation languages emphasized:

- expression of an algorithm in a way that was independent of the characteristics of the machine on which the algorithm would run.
- the rise of strong typing – by which typed languages deprecated or severely controlled access to the underlying storage representation of data. Complete prohibition of such access has never been a feature of major-use programming languages, which generally simply provide barriers to accidental access, e.g. coding them as "native" methods.
- block structure and automated management of storage with a stack – introduced in the Algol family of languages and adopted rapidly by most other major modular languages
- broad-spectrum applicability and greatly extended functionality – which was intended to service the needs of not only the previously separated commercial and scientific domains. The extended functionality often included concurrency features, creation and reference to non-stack data,

Re-characterization of First Three Generations

Since the 1990s, some authors have recharacterized the development of programming languages in a way that removed the (no longer topical) distinctions between early high-level languages like Fortran or Cobol and later ones, like

First Generation

In this categorization, a *first-generation programming language* is a machine-level programming language.

Originally, no translator was used to compile or assemble the first-generation language. The first-generation programming instructions were entered through the front panel switches of the computer system.

The main benefit of programming in a first-generation programming language is that the code a user writes can run very fast and efficiently, since it is directly executed by the CPU. However, machine language is a lot more difficult to learn than higher generational programming languages, and it is far more difficult to edit if errors occur. In addition, if instructions need to be added into memory at some location, then all the instructions after the insertion point need to be moved down to make room in memory to accommodate the new instructions. Doing so on a front panel with switches can be very difficult.

Second Generation

Second-generation programming language is a generational way to categorize assembly languages. The term was coined to provide a distinction from higher level third-generation

programming languages (3GL) such as COBOL and earlier machine code languages. Second-generation programming languages have the following properties:

- The code can be read and written by a programmer. To run on a computer it must be converted into a machine readable form, a process called assembly.
- The language is specific to a particular processor family and environment.

Second-generation languages are sometimes used in kernels and device drivers (though C is generally employed for this in modern kernels), but more often find use in extremely intensive processing such as games, video editing, graphic manipulation/rendering. One method for creating such code is by allowing a compiler to generate a machine-optimized assembly language version of a particular function. This code is then hand-tuned, gaining both the brute-force insight of the machine optimizing algorithm and the intuitive abilities of the human optimizer.

Third Generation

A *third-generation programming language* (*3GL*) is a refinement of a second-generation programming language. Whereas a second generation language is more aimed to fix logical structure to the language, a third generation language aims to refine the usability of the language in such a way to make it more user friendly. This could mean restructuring categories of possible functions to make it more efficient, condensing the overall bulk of code via classes (eg. Visual Basic). A third generation language improves over a second generation language by having more refinement on the

usability of the language itself from the perspective of the user.

First introduced in the late 1950s, FORTRAN, ALGOL and COBOL are early examples of this sort of language.

Most "modern" languages (BASIC, C, C++, C#, Pascal, and Java) are also third-generation languages.

Most 3GLs support structured programming.

Later Generations

"Generational" classification of these languages was abandoned after the third-generation languages, with the natural successors to the third-generation languages being termed object-oriented. C gave rise to C++ and later to Java and C#, Lisp to CLOS, ADA to ADA95, and even COBOL to COBOL2002, and new languages have emerged in that "generation" as well.

But significantly different languages and systems were already being called fourth and fifth generation programming languages by language communities with special interests. The manner in which these generations have been put forward tends to differ in character from those of earlier generations, and they represent software points-of-view leading away from the mainstream.

7

Measuring Programming Language Popularity

It is difficult to determine which programming languages are most widely used, and what usage means varies by context. One language may occupy the greater number of programmer hours, a different one have more lines of code, and a third utilize the most CPU time. Some languages are very popular for particular kinds of applications. For example, COBOL is still strong in the corporate data center, often on large mainframes; FORTRAN in engineering applications; C in embedded applications and operating systems; and other languages are regularly used to write many different kinds of applications. Various methods of measuring language popularity, each subject to a different bias over what is measured, have been proposed:

- counting the number of job advertisements that mention the language

- the number of books sold that teach or describe the language
- estimates of the number of existing lines of code written in the language—which may underestimate languages not often found in public searches
- counts of language references (i.e., to the name of the language) found using a web search engine.
- counting the number of projects in that language on SourceForge and FreshMeat.
- counting lines of code in a GNU/Linux Distribution

One organization tracking the popularity of programming languages is Tiobe. Their monthly *Programming Community Index* has been published since 2001, and shows the top 10 languages' popularity graphically, the top 20 languages with a rating and delta, and the top 50 languages' ratings. The numbers are based on searching the Web with certain phrases that include language names and counting the numbers of hits returned.

Combining and averaging information from various internet sites, langpop.com claims that in 2008 the 10 most cited programming languages are (in alphabetical order): C, C++, C#, Java, JavaScript, Perl, PHP, Python, Ruby, and SQL.

The *Language Popularity Index* is based on a similar approach, however in a totally transparent way: counts for all {search engine, language} pairs are published. A tool for grabbing counts from search engines is provided as well, so the rankings can be reproduced and verified.

The web site *Programming Language Popularity* aggregates statistics and charts on language popularity across a number of methodologies.

Programmer

A programmer, computer programmer or coder is someone who writes computer software. The term *computer programmer* can refer to a specialist in one area of computer programming or to a generalist who writes code for many kinds of software. One who practices or professes a formal approach to programming may also be known as a programmer analyst. A programmer's primary computer language (C, C++, Java, Lisp, Delphi etc.) is often prefixed to the above titles, and those who work in a web environment often prefix their titles with *web*. The term *programmer* can be used to refer to a software developer, software engineer, computer scientist, or software analyst. However, members of these professions typically possess other software engineering skills, beyond programming; for this reason, the term *programmer* is sometimes considered an insulting or derogatory oversimplification of these other professions. This has sparked much debate amongst developers, analysts, computer scientists, programmers, and outsiders who continue to be puzzled at the subtle differences in the definitions of these occupations.

Ada Lovelace is popularly credited as history's first programmer. She was the first to express an algorithm intended for implementation on a computer, Charles Babbage's analytical engine, in October 1842. Her work never ran, though that of Konrad Zuse did in 1941. The

ENIAC programming team, consisting of Kay McNulty, Betty Jennings, Betty Snyder, Marlyn Wescoff, Fran Bilas and Ruth Lichterman were the first working programmers.

In 2009, the government of Russia decreed a professional annual holiday known as Programmers' Day to be celebrated on September 13 (September 12 in leap year). It had also been an unofficial international holiday before that.

Nature of the Work

Some of this section is from the Occupational Outlook Handbook, 2006–07 Edition, which is in the public domain as a work of the United States Government. Computer programmers write, test, debug, and maintain the detailed instructions, called computer programmes, that computers must follow to perform their functions. Programmers also conceive, design, and test logical structures for solving problems by computer. Many technical innovations in programming — advanced computing technologies and sophisticated new languages and programming tools — have redefined the role of a programmer and elevated much of the programming work done today. Job titles and descriptions may vary, depending on the organization. Programmers work in many settings, including corporate information technology departments, big software companies, and small service firms.

Many professional programmers also work for consulting companies at client' sites as contractors. Licensing is not typically required to work as a programmer, although professional certifications are commonly held by programmers. Programming is widely considered a profession

(although some[who?] authorities disagree on the grounds that only careers with legal licensing requirements count as a profession).

Programmers' work varies widely depending on the type of business they are writing programmes for. For example, the instructions involved in updating financial records are very different from those required to duplicate conditions on an aircraft for pilots training in a flight simulator. Although simple programmes can be written in a few hours, programmes that use complex mathematical formulas whose solutions can only be approximated or that draw data from many existing systems may require more than a year of work. In most cases, several programmers work together as a team under a senior programmer's supervision.

Software Developer

A software developer is a person concerned with facets of the software development process. They can be involved in aspects wider than design and coding, a somewhat broader scope of computer programming or a specialty of project managing including some aspects of software product management. This person may contribute to the overview of the project on the application level rather than component level or individual programming tasks. Software developers are often still guided by lead programmers but also encompasses the class of freelance software developers. A person who develops stand-alone software (that is more than just a simple program) and got involved with all phases of the development (design and code) is a software developer.

Many legendary software people including Peter Norton (developer of *Norton Utilities*), Richard Garriott (Ultima-series creator), Philippe Kahn (Borland key founder), started as entrepreneurial individual or small-team software developers before they became rich and famous. Other names which are often used in the same close context are programmer, software analyst and software engineer. According to developer Eric Sink, the differences between system design, software development and programming are more apparent. Already in the current market place there can be found a segregation between programmers and developers, being that one who actually implements is not the same as the one who designs the class structure or hierarchy. Even more so that developers become systems architects, those who design the multi-leveled architecture or component interactions of a large software system. Aspects of developer's job may include:

- Software design
- Actual core implementation (programming which is often the most important portion of software development)
- Other required implementations (e.g. installation, configuration, customization, integration, data migration)
- Participation in software product definition, including Business case or Gap analysis
- Specification
- Requirements analysis
- Development and refinement of throw-away simulations or prototypes to confirm requirements

- Feasibility and Cost-benefit analysis, including the choice of application architecture and framework, leading to the budget and schedule for the project
- Authoring of documentation needed by users and implementation partners etc.
- Testing, including defining/supporting acceptance testing and gathering feedback from pre-release testers
- Participation in software release and post-release activities, including support for product launch evangelism (e.g. developing demonstrations and/or samples) and competitive analysis for subsequent product build/release cycles
- Maintenance

In a large company, there may be employees whose sole responsibility may consist of only one of the phases above. In smaller development environments, a few, or even a single individual might handle the complete process.

Separation of Concerns

In more mature engineering disciplines such as mechanical, civil and electrical engineering, the designers are separate from the implementers. That is, the engineers who generate design documents are not the same individuals who actually build things (such as mechanical parts, circuits, or roads, for instance). In software engineering, it is more common to have the architecture, design, implementation, and test functions performed by a single individual. In particular, the design and implementation of source code is commonly integrated. This resembles the early phases of

industrialization in which individuals would both design and built things. More mature organizations have separate test groups, but the architecture, design, implementation, and unit test functions are often performed by the same highly trained individuals.

Software Engineering

Software engineering (SE) is a profession dedicated to designing, implementing, and modifying software so that it is of higher quality, more affordable, maintainable, and faster to build. It is a "systematic approach to the analysis, design, assessment, implementation, test, maintenance and reengineering of software, that is, the application of engineering to software." The term *software engineering* first appeared in the 1968 NATO Software Engineering Conference, and was meant to provoke thought regarding the perceived "software crisis" at the time. The IEEE Computer Society's *Software Engineering Body of Knowledge* defines "software engineering" as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software. It is the application of Engineering to software because it integrates significant mathematics, computer science and practices whose origins are in Engineering.

Software development, a much used and more generic term, does not necessarily subsume the engineering paradigm. Although it is questionable what impact it has had on actual software development over the last more than

40 years, the field's future looks bright according to Money Magazine and Salary.com, which rated "software engineer" as the best job in the United States in 2006.

History

When the first modern digital computers appeared in the early 1940s, the instructions to make them operate were wired into the machine. Practitioners quickly realized that this design was not flexible and came up with the "stored programme architecture" or von Neumann architecture. Thus the first division between "hardware" and "software" began with abstraction being used to deal with the complexity of computing.

Programming languages started to appear in the 1950s and this was also another major step in abstraction. Major languages such as Fortran, ALGOL, and COBOL were released in the late 1950s to deal with scientific, algorithmic, and business problems respectively. E.W. Dijkstra wrote his seminal paper, "Go To Statement Considered Harmful", in 1968 and David Parnas introduced the key concept of modularity and information hiding in 1972 to help programmers deal with the ever increasing complexity of software systems. A software system for managing the hardware called an operating system was also introduced, most notably by Unix in 1969. In 1967, the Simula language introduced the object-oriented programming paradigm.

These advances in software were met with more advances in computer hardware. In the mid 1970s, the microcomputer was introduced, making it economical for hobbyists to obtain a computer and write software for it. This in turn led to the

now famous Personal Computer (PC) and Microsoft Windows. The Software Development Life Cycle or SDLC was also starting to appear as a consensus for centralized construction of software in the mid 1980s. The late 1970s and early 1980s saw the introduction of several new Simula-inspired object-oriented programming languages, including Smalltalk, Objective-C, and C++.

Open-source software started to appear in the early 90s in the form of Linux and other software introducing the "bazaar" or decentralized style of constructing software. Then the World Wide Web and the popularization of the Internet hit in the mid 90s, changing the engineering of software once again. Distributed systems gained sway as a way to design systems, and the Java programming language was introduced with its own virtual machine as another step in abstraction. Programmers collaborated and wrote the Agile Manifesto, which favored more lightweight processes to create cheaper and more timely software.

The current definition of *software engineering* is still being debated by practitioners today as they struggle to come up with ways to produce software that is "cheaper, better, faster". Cost reduction has been a primary focus of the IT industry since the 1990s. Total cost of ownership represents the costs of more than just acquisition. It includes things like productivity impediments, upkeep efforts, and resources needed to support infrastructure.

Profession

Legal requirements for the licensing or certification of professional software engineers vary around the world. In

the UK, the British Computer Society licenses software engineers and members of the society can also become Chartered Engineers (CEng), while in some areas of Canada, such as Alberta, Ontario, and Quebec, software engineers can hold the Professional Engineer (P.Eng)designation and/ or the Information Systems Professional (I.S.P.) designation; however, there is no legal requirement to have these qualifications.

The IEEE Computer Society and the ACM, the two main professional organizations of software engineering, publish guides to the profession of software engineering. The IEEE's *Guide to the Software Engineering Body of Knowledge - 2004 Version*, or SWEBOK, defines the field and describes the knowledge the IEEE expects a practicing software engineer to have. The IEEE also promulgates a "Software Engineering Code of Ethics".

Employment

In 2004, the U. S. Bureau of Labor Statistics counted 760,840 software engineers holding jobs in the U.S.; in the same time period there were some 1.4 million practitioners employed in the U.S. in all other engineering disciplines combined. Due to its relative newness as a field of study, formal education in software engineering is often taught as part of a computer science curriculum, and many software engineers hold computer science degrees.

Many software engineers work as employees or contractors. Software engineers work with businesses, government agencies (civilian or military), and non-profit organizations. Some software engineers work for themselves

as freelancers. Some organizations have specialists to perform each of the tasks in the software development process. Other organizations require software engineers to do many or all of them. In large projects, people may specialize in only one role. In small projects, people may fill several or all roles at the same time. Specializations include: in industry (analysts, architects, developers, testers, technical support, middleware analysts, managers) and in academia (educators, researchers). Most software engineers and programmers work 40 hours a week, but about 15 percent of software engineers and 11 percent of programmers worked more than 50 hours a week in 2008. Injuries in these occupations are rare. However, like other workers who spend long periods in front of a computer terminal typing at a keyboard, engineers and programmers are susceptible to eyestrain, back discomfort, and hand and wrist problems such as carpal tunnel syndrome.

Certification

The Software Engineering Institute offers certifications on specific topics like Security, Process improvement and Software architecture. Apple, IBM, Microsoft and other companies also sponsor their own certification examinations. Many IT certification programmes are oriented toward specific technologies, and managed by the vendors of these technologies. These certification programmes are tailored to the institutions that would employ people who use these technologies.

Broader certification of general software engineering skills is available through various professional societies. As of

2006, the IEEE had certified over 575 software professionals as a Certified Software Development Professional (CSDP). In 2008 they added an entry-level certification known as the Certified Software Development Associate (CSDA). In the U.K. the British Computer Society has developed a legally recognized professional certification called *Chartered IT Professional (CITP)*, available to fully qualified Members (*MBCS*). In Canada the Canadian Information Processing Society has developed a legally recognized professional certification called *Information Systems Professional (ISP)*. The ACM had a professional certification programme in the early 1980s, which was discontinued due to lack of interest. The ACM examined the possibility of professional certification of software engineers in the late 1990s, but eventually decided that such certification was inappropriate for the professional industrial practice of software engineering.

Impact of Globalization

The initial impact of outsourcing, and the relatively lower cost of international human resources in developing third world countries led to the dot com bubble burst of the 1990s. This had a negative impact on many aspects of the software engineering profession. For example, some students in the developed world avoid education related to software engineering because of the fear of offshore outsourcing (importing software products or services from other countries) and of being displaced by foreign visa workers. Although statistics do not currently show a threat to software engineering itself; a related career, computer programming does appear to have been affected. Nevertheless, the ability

to smartly leverage offshore and near-shore resources via the [follow-the-sun] workflow has improved the overall operational capability of many organizations. When North Americans are leaving work, Asians are just arriving to work. When Asians are leaving work, Europeans are arriving to work. This provides a continuous ability to have human oversight on business-critical processes 24 hours per day, without paying overtime compensation or disrupting key human resource sleep patterns.

Education

A knowledge of programming is a pre-requisite to becoming a software engineer. In 2004 the IEEE Computer Society produced the SWEBOK, which has been published as ISO/IEC Technical Report 19759:2004, describing the body of knowledge that they believe should be mastered by a graduate software engineer with four years of experience. Many software engineers enter the profession by obtaining a university degree or training at a vocational school. One standard international curriculum for undergraduate software engineering degrees was defined by the CCSE, and updated in 2004. A number of universities have Software Engineering degree programmes; as of 2010, there were 244 Campus programmes, 70 Online programmes, 230 Masters-level programmes, 41 Doctorate-level programmes, and 69 Certificate-level programmes in the United States.

In addition to university education, many companies sponsor internships for students wishing to pursue careers in information technology. These internships can introduce the student to interesting real-world tasks that typical

software engineers encounter every day. Similar experience can be gained through military service in software engineering.

Comparison with Other Disciplines

Major differences between software engineering and other engineering disciplines, according to some researchers, result from the costs of fabrication.

Sub-disciplines

Software engineering can be divided into ten subdisciplines. They are:

- Software requirements: The elicitation, analysis, specification, and validation of requirements for software.
- Software architecture: The elicitation, analysis, specification, definition and design, and validation and control of software architecture requirements.
- Software design: The design of software is usually done with Computer-Aided Software Engineering (CASE) tools and use standards for the format, such as the Unified Modeling Language (UML).
- Software development: The construction of software through the use of programming languages.
- Software testing
- Software maintenance: Software systems often have problems and need enhancements for a long time after they are first completed. This subfield deals with those problems.
- Software configuration management: Since software

systems are very complex, their configuration (such as versioning and source control) have to be managed in a standardized and structured method.

- Software engineering management: The management of software systems borrows heavily from project management, but there are nuances encountered in software not seen in other management disciplines.
- Software development process: The process of building software is hotly debated among practitioners; some of the better-known processes are the Waterfall Model, the Spiral Model, Iterative and Incremental Development, and Agile Development.
- Software engineering tools, see Computer Aided Software Engineering
- Software quality

Related Disciplines

Software engineering is a direct subfield of computer science and has some relations with management science. It is also considered a part of overall systems engineering.

Systems Engineering

Systems engineers deal primarily with the overall system design, specifically dealing more with physical aspects which include hardware design. Those who choose to specialize in computer hardware engineering may have some training in software engineering.

Computer Software Engineers

Computer Software Engineers are usually systems level (software engineering, information systems) computer science

or software level computer engineering graduates. This term also includes general computer science graduates with a few years of practical on the job experience involving software engineering.

serve as input to an interpreter. When a piece of computer hardware can interpret a programming language directly, that language is called *machine code*. A so-called *native code compiler* is one that compiles a programme into machine code. Actual compilation is often separated into multiple passes, like code generation (often in for of assembler language), assembling (generating native code), linking, loading and execution.

If a compiler of a given high level language produces another high level language it is called translator (source to source translation), which is often useful to add extensions to existing languages or to exploit good and portable implementation of other language (for example C), simplifying development. Many combinations of interpretation and compilation are possible, and many modern programming language implementations include elements of both. For example, the Smalltalk programming language is conventionally implemented by compilation into bytecode, which is then either interpreted or compiled by a virtual machine (most popular ways is to use JIT or AOT compiler compilation). This implementation strategy has been copied by many languages since Smalltalk pioneered it in the 1970s and 1980s.