Software Development

5.1 Defining Software

Software is a set of instructions that is readable by a computer. Software is a generic term for organized collections of instructions. It is often broken into two major categories and one sub category: system software and application software. System software provides the soft infrastructure by which a computer operates while application software facilitates the specific tasks needed by users. The sub category is malicious software. This is software that is designed to hurt or harm a user or the computer they use. It can be considered a subset of application software in that it runs on the system infrastructure. At the same time though, it can take elements of system software. Malicious software can be written to manipulate system level processes and run hidden from the user. Malicious software will be discussed further in the chapter on security.

The first major category of software, system software, runs behind the scenes. It is responsible for controlling, integrating, and managing the individual components of a computer system so that other software and the users of the system see the entire system as a single cohesive unit. Users should not have to be concerned with low-level details such as transferring data from memory to disk, or rendering text onto a display. Rather, this should happen seamlessly by well-designed system software. Typically, system software consists of an operating system and some essential utilities such as Basic Input-Output Systems (BIOS), disk formatters, file managers, display managers, text editors, user authentication and management tools, and networking and device control software.

The second major category, application software, is used to accomplish user related tasks and not be concerned with how the system actually runs. Application software may consist of different packages and collections, including single programs, packages, suites, or systems. An example of a single program could be an image viewer, statistics analytical tool, or a video game. A software package (or small collection of programs that work closely together to accomplish a task) could take the form of a spreadsheet or text processing system. A package can be a larger collection which is often called a software suite. These are related but independent programs and packages that have a common user interface or shared data format. A good and widely used example of this is Microsoft Office. This suite consists of closely integrated word processor, spreadsheet, presentation software, and database packages. Finally, the largest grouping of software is called system. This could take the form of a database management system, Enterprise systems, or proprietary organization-wide system software. These are examples of software that are collections of fundamental programs, packages, and suites that service large scale organizations. Though it shares its name with the broader category of "system software," a "system" of application software is NOT the same as system software. It does not run at the system level of the computer; instead it runs as a large collection of applications. Since it encompasses the entire "system" of an organization, it utilizes that name.

5.2 Creating Software

The people who write these instructions are called programmers. The instructions are usually called code and they can take many different forms. These forms are most often known as programming languages. There are hundreds of programming languages. The most widely used in terms of the number of available programming jobs as of 2019 are Java, Javascript, C#, Python, C++, PHP, Swift, and Ruby/Rails[1].

Surprisingly, these languages are a tiny percentage of the actual code that is in use across the world. A 60 year old language called COBOL dominates this statistic. IBM's

[1] Data pulled from http://www.indeed.com/jobs

software group director of product delivery and strategy, Charles Chu, notes that there are over 220 billion lines of COBOL in existence; a figure which equates to about 80 per cent of the world's actively used code[2]. There are over a million COBOL programmers in the world. There are 200 times as many COBOL transactions that take place each day than Google searches.

When a programmer writes code with any of these languages, it is in a form that a human can understand but a computer cannot. There is a type of system software called a compiler that takes this human written code and transforms it into a language that a computer can understand. The compiler is expecting an exactly correct syntax with this code and will reject any code that has any errors. If the code was written correctly, it will "compile" the code into an executable format that can then be run by a computer.

5.3 Generations of Programming Languages

Programming languages have evolved dramatically since their inception in the 1940s. At that time, there was no abstraction between the hardware being programmed and the program itself. The language was the language of machines: 1s and 0s. Programming at this base level eventually became known as the first generation of programming. It was tedious and time consuming to develop the simplest of programs. As a result of this, the earliest programs were simple calculators and data sorters.

A famous early example of "machine language" first generation programs were the cryptanalysis programs written for the Colossus and Bombe computers in the early to mid 1940s in the United Kingdom. These programs helped defeat the German forces during World War II by decrypting messages sent using the Enigma Machine.

[2] https://www.theguardian.com/technology/2009/apr/09/cobol-internet-programming

Over time, the need for more sophisticated programs led to the development of the first programming language, Assembly. This was the first abstraction away from the hardware as the programmer now was telling the language what they wanted to do instead of telling the hardware directly what to do. From a modern perspective, this abstraction was on the minimal side. The developer was still directing very specific parts of the hardware, like registers and memory locations. Though this left a significant amount of power in the developer's hands, it also led to long development schedules. It took many lines of code to do simple things. Even the simple task of declaring a variable was multiple lines of code that could take time.

As the 1940s came to a close and businesses boomed in the 1950s, it became clear that further abstraction was needed to expedite development time. At the same time, there was pressure to limit abstraction, as high levels of abstraction did not allow developers to manipulate the hardware at a precise level. When computers were first introduced to governments and private organizations, memory and processing was extremely expensive. Software developers needed to optimize every aspect of the hardware to maximize its usage. One great modern example of this is code that is written for machines that NASA sends to Mars. The cost to send a kilogram of material to Mars is about 2.8 million dollars so hardware and even code has a cost attached to it. A single line of unnecessary code can add thousands of dollars to the cost of the project. Developers are under extreme pressure to be as efficient as possible with their coding.

As computers became more prevalent in organizations in the 1950s, the pressure for expedited software development continued to increase. This ultimately led to the introduction of programming languages in the late 1950s that were orders of magnitude more efficient and abstract than the 2nd generation. These languages became known as the

3rd generation of languages. The ability to specifically manipulate hardware components was more limited than the 3rd generation but the ability to quickly write applications was introduced. Languages like COBOL, Fortran, and Lisp were introduced and gave developers different options depending on what they needed to do. Organizations that needed large-scale data processing tended to rely on COBOL whereas scientists tended to use languages like Fortran and Lisp. They each had their strengths and weaknesses. Unlike the first generation, a compiler was necessary to turn the code into something the hardware could understand.

Over the ensuing decades, many 3rd generation languages emerged that filled a niche or need. In 1970, Pascal was created to provide a language that made learning programming easy. In 1972, C was created for programming on the new Unix operating system. It was such a versatile language that many other languages derivatives grew from C. These include C++, C#, Java, Javascript, Perl, PHP, and Python. As the web grew in the 1990s, web based languages like PHP, Ruby, Cold Fusion, and JavaScript became prevalent.

In the modern age, mobile development is at the forefront. Java is the official programming language of the Android platform. Apple's iOS platform uses a proprietary language called Swift. Swift has a similar syntax to the C family of languages but has streamlined (abstracted) much of the code. There are several indices that report the most popular programming languages as of 2019. There is some variation in these lists but generally, Java, C, Javascript, Python, and Swift make appearances.

Some make the argument that a 4th generation of programming languages already exists. The grey area involves defining what level of abstraction would constitute a break from the 3rd generation and into a new paradigm of programming. On one end of the argument, the 4th generation would constitute a level of abstraction so high that it is indistinguishable from natural language. In other words, a regular person that is not a programmer could just describe, in normal language, what they want in a program and it would be created. When Computer Assisted Software Engineering (CASE) tools were developed in the 1980s, some argued that this met that criterion. The reality of the situation was that the software that was created by CASE tools was generally of very poor quality and required significant revision by human programmers. In many cases, the organization would end up spending more man-hours fixing CASE created software than they would have if they had just developed the code from scratch.

On the other end of the argument for the 4th generation of programming languages, some argue that a moderate, iterative change in the way code is developed is sufficient for classification as a new generation. For example, many Database Management Systems (DBMS's) have built in tools to create reports based on the data. Since report generation is a very common task for programs in organizations, the fact that it is an automatically generated thing in DBMS's, some argue this is a form of 4th generation development. Other software like Crystal Reports, Hyperion, and Power BI provide similar functionality.

Though the concept of a 4th generation of programming has been discussed for more than 30 years, it is a difficult argument to make that report generators, Graphic User Interface (GUI) creators, or broken CASE tools made a paradigmatic change to the way software is developed. The changes that happened in software coding when developers were finally able to write code instead of directly manipulating the hardware (the move from the 1st to the 2nd generation) were massive. The same can be said when high level, 3rd generation languages debuted in the 1950s and into today. None of the technology that has sometimes been described as a 4th generation can make a similar claim. The vast majority of software development is still created with 3rd generation languages.

5.4 Systems Development Lifecycle (SDLC)

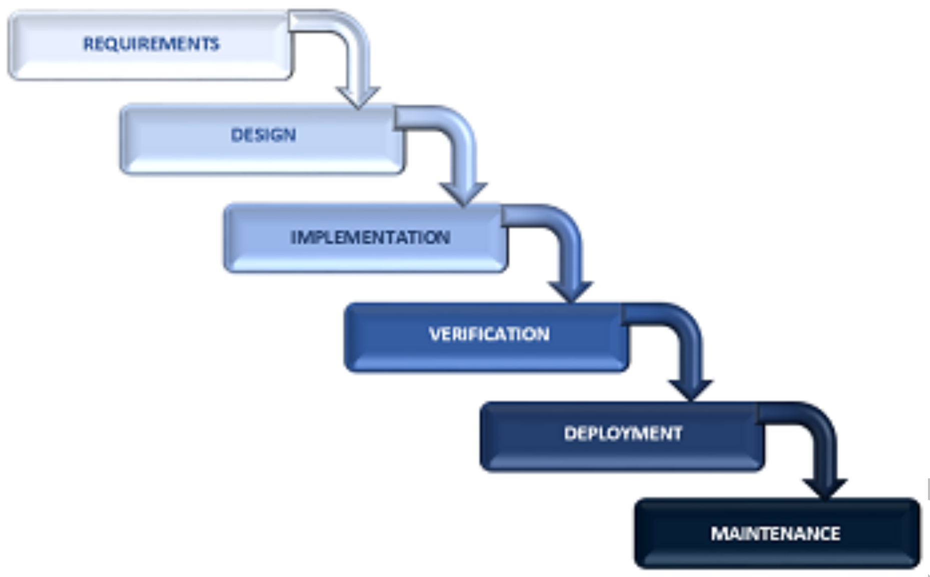
When software is developed, a set of steps is prescribed to maximize the chance of success in the development of the program. It is a well-known phenomenon in the software development world that many new Information Systems projects have some degree of significant failure. The Standish Group Chaos Report reports that only 29% of IT project implementations are successful, and 19 percent are considered utter failures[3]. Having a defined process to build new software projects is critical.

The de facto standard steps in program development are intuitive: planning, analysis, design, implementation, and maintenance. There are many variations and adjustments to these basic steps that compress or expand various core steps. For example, some consider testing of the new system a distinct step from implementation. Others do not include planning as that a purely managerial decision point regarding scheduling, personnel, fiscal viability, and technical viability. Yet others divide maintenance into security, enhancements, and fixes. Regardless, the principle of the core steps outlined can be found in any systems development lifecycle.

5.4.1 Development Methodologies

The core steps described above steps can be carried out in sequence, such as the venerated "waterfall" methodology. This approach, which had its origins in the construction and manufacturing sectors seemed to be a natural fit in the emerging computing world of the 1950s. In the waterfall approach, the development steps are rigidly placed one after the other as can be seen in the diagram below.

[3] https://standishgroup.com/



What developers and analysts failed to realize was that such rigidity made software development difficult in the real world. For example, in the sample waterfall approach displayed in the figure above, a major issue could be identified in the verification step. At this point, the project would likely be 6-12 months into the project. A major issue might require going back to the requirements stage, which this type of methodology does not easily enable. In a situation like that, what usually happens is that the project is deemed a "failure" and is scrapped. Sometimes, it is restarted as a new project with an awareness of the issue that came up the first time around. Despite its poor outcomes, the waterfall approach remained the gold standard for nearly four decades.

By the 1980s, the core weakness of the waterfall approach grew to a boiling point and alternate methodological approaches were sought after. The rigidity of the model is its core weakness so greater flexibility was the main need. One of the earliest alternate approaches was the spiral methodology. Given the inherent iterative nature of software with regular releases of updated code, a methodology with iteration at its core was a welcome change to the de facto standard of the waterfall method.

In the Spiral methodology, the same core steps are there but the underlying philosophy of development is different. As its name implies, the steps area arranged in a spiral where you start with planning and analysis but instead of ending with a 100% finished product, the goal from the start is a prototype. The prototype then informs the next cycle of planning, analysis, design, and implementation. The entire process is "maintenance" as a completed system is never the actual goal. The goal is continuous improvement of the system and growth through continued analysis and code adjustment.

Though it was significantly more flexible and adaptive than the waterfall approach, the spiral approach did not seem to affect the overall failure rates of information systems projects. Further flexibility was introduced with the contemporary methodology, agile development. Though officially credited with the 2001 book Manifesto for Agile Software Development, agile methods had been discussed and used up to 10 years prior with Rapid Application Development (RAD), the Unified Process (UP), Scrum, and Extreme Programming (XP).

Agile development is similar to the spiral method but is more focused on individuals and interactions over processes and tools, working software over documentation, customer Collaboration over contract negotiation, and responding to change over following a plan. Flexibility and adaptation are the core principles of the agile methodology. It also strives for adaptation over pretending to know the end point at the beginning.

As discussed in the preceding methodology subsection, there are core steps involved in the creation of any IS project. These are the Planning, Analysis, Design, Implementation, and Maintenance phases. The following three subsections will take a closer look at Analysis, Design, and Implementation. Planning takes place before a project begins and maintenance occurs after a project finishes. So, the actual core phases of the project itself are Analysis, Design, and Implementation.

5.4.2 Analysis

The point of the analysis phase is to gather and organize the business requirements that are driving the Information System. The questions answered here revolve around what the clients need. What should the system do within the organization? What are the users' expectations? What is the organization's expectation? How should the system interact with the users? What is the state of the current system? Is the new system intended to improve on the current system or completely reengineer it?

To answer these questions, data collection is the heart of the analysis phase. Interviews are conducted with end users. Observations are conducted of users using the current system. Meetings are conducted between all stakeholders including users, managers, developers, and systems analysts. The current system is analyzed through direct use and document review.

All of this data wouldn't be useful without significant organization. The best approach with this is to organize the data into a set of Use Cases. These are formal descriptions of tasks that users do at an organization. An important part of a Use Case is the Input-Process-Output (IPO) chart. This shows how the data flows through the organization and how the data will flow with the new system.

5.4.2 Design

The Use Cases created during Analysis inform the next phase, the Design phase. The intent of this phase is to link the data organized in the Analysis phase with the program that is to be created in the Implementation phase. This is generally done in a "top down" fashion, meaning that the big picture is designed first and the details follow from that.

The first "big picture" item completed is the structure chart which is literally a big picture of the entire system's components. The Systems Analyst breaks down the system into major components, then the sub components of the major components, then the smallest components of the sub components. This process is not always clear cut as the analyst has to determine how large to make each component and how much each should be broken down. If they are not broken down enough, then the component would be unwieldy and likely to contain difficult to fix errors. If they are broken down too much, then the system will be slowed down by excessive module calls.

The next step in the Design phase is to figure out the details of the algorithm for each module. This can be done with program flowcharts or pseudocode. Program flowcharts use diagrams to display the flow of logic. The flow is from top to bottom and each step is shown with a rectangle. Decision points are displayed with diamonds. Decision points are used with conditional statements. These types of statements allow for different logic to run depending on a condition. For example, if a user misses a required field, the logic should call for displaying the input screen again. Like conditional statements, looping statement starts with a decision point. Unlike conditional statements, looping statements return the flow of logic to the top of decision point. This way, the logic inside of the loop is run continuously until the condition is met. An alternative to program flowcharts is pseudocode. Pseudocode is logic that is written by manually but without the strict rules of a programming language's syntax.

5.4.3 Implementation

The implementation phase is where the system is actually created. This can mean writing the code in house, contracting an outside company to write the code, offshoring the coding to an overseas company, or purchasing an off the shelf solution. Each of these options has benefits and drawbacks. It all boils down to how much an organization wants to invest in the project and how much risk they want to incur. It also depends on the degree of in house talent the organization has on hand. If they have no developers or IT department, it would be too cost prohibitive to do it in house.

Assuming an organization has the resources (financial, personnel, and technology) and managerial foresight, the most effective option is to build the system in house. In this situation, all the stakeholders have a vested interest in the substantive success of the organization. They also are the most intimately familiar with the organization and its nuances. This typically translates to a well-tailored system.

If the pieces are not in place for an in house solution, an organization has to look externally for a solution. Finding a respected vendor is critical as there is added risk in implementing the project externally. Outsourcing can end up costing more than in house solutions so if the major issue is funding, organizations can take bids from offshore companies. A common area for this is India. There are many tech companies there that bid for projects in the US. Though the talent in India is extremely high, cultural differences can lead to issues. There is a very high failure rate for offshored projects. It is high enough that many US organizations have stopped offshoring.

A final possibility is to find a pre built system that the organization can license. This is a dangerous option as it is extremely unlikely that pre-existing code will fit the exact requirements for a given organization. What many organizations do is tweak the organization to fit the software. This can be disastrous as people are inherently resistant to change and forcing it can backfire on management. What can happen is that an organization's productivity can decline to the point of questioning the worthiness of the new system that is causing the problems.

5.5 Conclusion

Software development is at the core of Information Systems. Without software, the technical side of the IS formula would not be possible. Understanding how software is created is an important aspect of understanding Information Systems as a whole. From the taxonomy of software, to the languages used to develop software, to the process by which software development happens all paints a picture of how the code comes together to affect the people in the organization.