FINAL ASSIGNMENT: Software Engineering as a Process

Alicia Piavis

CST301: Software Technology & Design

Robert Key

6/2/2019

FINAL ASSIGNMENT: Software Engineering as a Process

Table of Contents

Part I. Software Engineering Processes and Models……………………………………………3

Part II. Software Development Methods and Requirements Engineering………………………..5

Part III: System Modeling and Architectural Design……………………………………………10

Part IV: Object-Oriented Design, Implementation, and Testing………………………..…………13

Part V: Software Evolution and Maintenance…………………………………………...…..16

Part VI: Reflection…………………………………………………………………………20

Part VII. Concept Map………………………………………………………………………21

Resources……………………………………………………………………………………22

Part I. Software Engineering Processes and Models

Software engineering is a complex process involving all phases of software production from the initial need and design, all the way through implementation, operation, and maintenance. Some of the elements involved in software engineering include “applying theories, methods, and tools where these are appropriate” in order to meet customer and user requirements (Sommerville, 2016). Software engineering is much more than the process of developing software. According to Software Engineering (10th ed.) (2016), software engineering involves “practical cost, schedule, and dependability issues, as well as the needs of software customers and producers.” Software is not only the code written to build a computer program, but also the documentation required to make the program effective (Sommerville, 2016). There are four fundamental activities involved in software engineering: specification, development, validation, and evolution. Specification encompasses the process of gathering requirements and defining the software and constraints. Development describes the process of designing and programming the software. Validation is the process of verifying that the product meets customer requirements. Evolution describes the final step, which includes modifying the software to adapt to changing user needs (Sommerville, 2016).

There are various software development models, which describe differing ways to approach the software development process. Four of the basic models are waterfall, incremental development, integration and configuration, and Rational Unified Process (RUP). The waterfall model approaches specification, development, validation, and evolution in a sequential way. All specifications are defined and documented before any software is developed. Then the process cascades from one phase to another in the following order of steps: requirements definition, system and software design, implementation and unit testing, integration and system testing, and

operation and maintenance. The best use of this model is in embedded systems, critical systems, and "large software systems that are part of broader engineering systems developed by several partner companies" (Sommerville, 2016). The incremental development model has crossover between the different stages of the software development life cycle. This model emphasizes rapid feedback from users, and involves small increments in which increasing functionality is added. Th incremental model is beneficial when requirements are likely to change during development. In addition, benefits include reduced cost to implement changes, more frequent customer feedback, and more rapid delivery of a product to the customer. Integration and configuration is a model founded on reusing software components. The most common components that get reused are stand-alone application systems, collections of objects, and web services. The stages in this model include requirements specification, software discovery and evaluation, requirements refinement, application system configuration, and component adaptation and integration (Sommerville, 2016). The model is ideal when time is limited, and when cost is of great consideration. The final model, RUP, combines elements of all of the aforementioned models. This approach involves four phases: inception, elaboration, construction, and transition.

Specification, also known as requirements engineering, is the most critical step in the software development life cycle (Sommerville, 2016). This is because errors made in this stage can trickle down to other phases later on in the process. In some cases, companies will carry out market studies to see if developing the desired product is feasible and necessary. During the requirements engineering process, requirements are gathered and constraints are identified. The goal of this step is to create documentation that will be agreed upon between the company and the client to ensure that the clients' needs are met. The requirements are usually outlined at two

different levels—a high level for the users, and detailed version for the developers (Sommerville, 2016). There are three fundamental activities involved during the requirements engineering process: requirements elicitation and analysis, requirements specification, and requirements validation. The first step defines the process of gathering requirements through observation, discussion, and task analysis. The second step involves translating the gathered requirements into documentation. The final step ensures that requirements are evaluated for "realism, consistency, and completeness" (Sommerville, 2016).

It is important to note that part of software engineering includes choosing the right method for the type of system being built. For example, Sommerville (2016) describes an insulin pump control system, which analyzes information from the blood collected by a sensor, and then controls the dose of insulin being administered to an individual with diabetes. This is considered a safety-critical system, since the patient's life depends on the reliability of the system. In this case, the design process should be determined by the requirements of the system, which include delivering insulin when required, and correctly administering differing levels of insulin, depending on readings from the sensor. This system would benefit most from the waterfall model of development, since safety is critical, and all specifications should be thoroughly documented and tested before being used on a human. Another system that would benefits from the waterfall model is embedded control systems in automobiles, since these systems are expensive to change, and safety if of high priority (Sommerville, 2016).

Part II. Software Development Methods and Requirements Engineering

Agile development is an incremental approach to the software development life cycle that emphasizes rapid development and delivery. Features are developed in increments, and delivered to customers every two to three weeks. This process allows for increased customer

feedback to ensure the product is meeting user needs and expectations. The agile method emphasizes design and implementation over documentation, and encourages the direct involvement of stakeholders in the design and development process (Sommerville, 2016). Contrary to plan-based methods, agile development supports rapid software development projects, reduces formal documentation, involves interleaved stages of development, employs iteration across activities, and supports less formalized testing processes.

A common theme in software development is that the system type determines the method selected for the development process. For example, a safety-critical system would benefit from a plan-based approach involving heavy front-end planning and documentation. On the other hand, agile methods are beneficial in dynamic business environments where rapid delivery of a product is of utmost importance (Sommerville, 2016). Agile methods are used for application development, where system requirements change constantly throughout development. Furthermore, agile methods support small to medium-sized development products built by software development companies, as well as "custom system development within an organization, where there is a clear commitment from the customer to become involved in the development process" (Sommerville, 2016).

Agile development methods are facilitated through a specialized project management process called Scrum. According to Sommerville (2016), scrum is "a framework for organizing agile projects." The process involves short sprints (two to three weeks each) that are determined by the product backlog. The backlog involves user features, requirements, and engineering improvements that have not yet been implemented. These components initially come from requirements documents and user stories. A user story is "a scenario of use that might be experienced by a system user" (Sommerville, 2016). The product owner on the Scrum team

reviews the backlog and prioritizes items. Then the whole Scrum team decides which high-priority items are feasible to complete during the sprint. The team determines how to distribute the work, and then the sprint proceeds with daily Scrum meetings to review progress (Sommerville, 2016). This process provides external visibility since it often involves a Scrum board to track progress and assign tasks. In addition, since visibility is improved, collaboration and morale typically increase.

One of the original agile approaches introduced in the 1990's is known as Extreme Programming (XP). This model pushed the limits of incremental development to the extreme, even resulting in new versions of a system being developed and tested within a single day. This model is based on user stories, which determine requirements. A user story is, "a scenario of use that might be experienced by a system user" (Sommerville, 2016). This story involves the direct interaction of a customer with the development team, and results in a story card. The card is then broken down into tasks, which are prioritized and then implemented. Programmers often work in pairs to accomplish the tasks. The basic pillars of extreme programming include collective ownership, continuous integration, incremental planning, on-site customer, pair programming, refactoring, simple design, small releases, sustainable pace, and test-first development (Sommerville, 2016). These methods are beneficial when rapid development and delivery is essential.

While agile methods can be beneficial in many scenarios, the agile approach involves a few challenges. For example, it can be difficult to propose the adaptation of agile methods when a company has been using formal, plan-based approaches since its conception. In addition, agile methods involve user stories, which can be vague and hard to analyze for completeness (Sommerville, 2016). Regarding the incremental methods employed, increments often involve

finding workarounds to problems, which slowly degrades the code base over time and requires refactoring. The agile approach can also be considered too informal for use at large organizations that are highly regulated. Furthermore, agile methods are intended to be employed by small teams of developers located in the same geographic area (Sommerville, 2016). Considering the fact that many development teams and stakeholders are now distributed, this poses a challenge in the collaboration process.

Scaling agile methods while maintaining the fundamentals of agile development can be challenging. Agile methods can be scaled up and out. Scaling up involves scaling for larger systems. Scaling out refers to the process of making the methods used by specialized teams more widespread within an organization. Often, the scaling of agile methods involves multiple agile teams working on components of one larger project. One of the best ways to approach scaling agile methods is to integrate them with plan-driven methods. This allows developers to employ the rapid software development characteristic of agile methods, while satisfying the documentation and regulatory requirements of larger organizations and projects. The balance between agile and plan-based is determined by the type of system, the development team, and the organization desiring the system. When employing Scrum for large-scale development, multiple Scrum teams abide by the following characteristics: role replication, product architects, release alignment, and Scrum of Scrums (Sommerville, 2016).

Requirements engineering is one of the key components of the software development life cycle. This process involves defining what a system should do, along with any system constraints. Requirements documentation is written at two levels: user requirements, and system requirements. User requirements are defined at a high level, and written in natural language that is easy to understand. System requirements are much more detailed, and can use technical

language to guide developers and system architects in the implementation process (Sommerville, 2016). There are two major categories of requirements: functional and non-functional. Functional requirements state the services the system should provide, or what the system should do. These requirements are written in natural language and describe how the system should react in specific situations (Sommerville, 2016). An example of a functional requirement might be that a nurse needs to be able to see the history of drugs that have been administered to a patient in a single ER visit, within the context of a health information system. Functional requirements can also define what a system should not do. Non-functional requirements emphasize constraints on the system. Constraints can include budget and time limitations. These requirements usually pertain to the whole system, rather than a unique part of it (Sommerville, 2016). An example of a non-functional requirement is reliability. If an insulin delivery pump system is not reliable, then it should not be implemented.

The process of requirement elicitation, specification, validation, and change is iterative. This means that requirements can change throughout the development process, and the process should lend itself to change. The aim of the elicitation process is to understand how stakeholders might use a system to accomplish their work. However, additional insight can be gained with each iteration of the software development process. Therefore, the requirements elicitation process needs to be revisited. Somerville (2016) describes this iterative process as a spiral that starts with requirements discovery and ends with documentation. However, throughout each iteration, requirements can change as "people involved develop a better understanding of what they want the software to do; the organization buying the system changes; and modifications are made to the system's hardware, software, and organizational environment" (Sommerville, 2016). There are a number of ways that requirements can be elicited including interviews, ethnographic

observation, and acquiring stories, which provide a high-level picture of how the system will be used. Requirement specification refers to the formal documentation process, which can involve natural language, structured natural language, graphical notations, and/or mathematical specifications. Finally, requirements go through a validation process, which ensures that the requirements meet the customer needs of the system. This process involves validity checks, consistency checks, completeness checks, realism checks, and verifiability. In addition, there are multiple validation techniques that can be employed including requirements reviews, prototyping, and test-case generation (Sommerville, 2016).

Part III. System Modeling and Architectural Design

Architectural models are used throughout the software development life cycle to show an abstract perspective of a system. These models can show the context, interactions, structure, and behavior of a system. A contextual view models the environment surrounding a system. An interaction view models interactions between the system and its environment. Examples of interaction models include case modeling and sequence diagrams. A structural view represents the architecture and organization of a system, such as in a class diagram. Lastly, a behavioral view shows a system in execution, and how it responds to stimuli in the environment (Sommerville, 2016). Before the software development process begins, it is important that engineers develop an architectural design, since the system design can affect performance, security, and availability (Sommerville, 2016). There are a number of Unified Modeling Language (UML) diagrams that can be used to build architectural models, which aid in the design process. An activity diagram shows the steps involved in a process. A use case diagram shows interactions between the system and its environment under specific conditions. A sequence diagram shows interactions between users and the system, or between components

within a system. A class diagram shows object classes and their associations. A state diagram illustrates how a system responds to internal and external events, and supports events-driven modeling (Sommerville, 2016).

There are two main categories of behavioral modeling: data-driven modeling and event-driven modeling (Sommerville, 2016). Data-driven models illustrate end-to-end processing and involve a sequence of actions from input to output. A common data-driven diagram used when stakeholders are involved is the data-flow-diagram. These diagrams are simple and intuitive. When more detail is required, a sequence diagram might be used. Sequence diagrams tend to be more beneficial for engineers, who have the technical knowledge required to interpret and implement them. An example of a process that can be illustrated with a data-driven model is an insulin pump system, which involves data flow in between processing steps. For example, steps in the system include: get sensor value, compute sugar level, calculate insulin delivery, calculate pump commands, and control pump. The data flowing between these steps include: blood sugar sensor, sensor data, blood sugar level, insulin requirement, pump control commands, and the insulin pump (Sommerville, 2016).

Event-driven modeling on the other hand shows how internal and external events affect a system. This model is based on the fact that a system exhibits different states, and transitions between these states occur in response to stimuli in the environment. Event-driven models are beneficial for real-time systems. While they show states and events, they do not show data flow within a system. A table may be used in conjunction with a state-diagram to provide more detail to the model (Sommerville, 2016). Event-driven modeling might be used to illustrate the different states a patient exhibits as they transition through different steps in a clinical admissions process (these states might align with data in the health information system being utilized). For

example, a patient's file might be tagged in an "appointment" state when they have not yet checked in. They might transition into a "waiting" state once they check in and are waiting to be seen. They may exhibit other states as they have their vitals checked by a nurse, they wait to be seen by a practitioner, they receive a diagnosis, they pick up their prescription, and they are discharged.

While architectural models can provide an abstraction of a system from different views, architectural patterns are also beneficial in the design process. Architectural patterns allow engineers to reuse design ideas from generic system architectures within particular domains. A few common architectural patterns include layered, client-server, and application architecture. A layered architectural pattern organizes functionality into separate, independent layers. The lowest layer is typically the system support software. The next layer is the application layer. Then there is the user interface management layer. The top layer is the user interface (Sommerville, 2016). A client-server pattern involves a set of servers, a set of clients, and a network, which allows clients to access different services from distributed servers. Finally, application architecture is used to meet business needs and allows for the reuse of design principals across applications within the same sector.

Examples of application architectures include transaction processing systems and language processing systems (Sommerville, 2016). Transaction processing is commonly used for applications that interact with a database to process user requests. For example, an ATM uses transaction processing in that it involves I/O when the user interacts with the system, it incorporates application logic to check an account for sufficient funds, it uses a transaction manager that send commands to the ATM to disburse cash, and it involves a database, which is updated after the transaction between the use and the ATM is fully complete (Sommerville,

2016). Language-processing systems are another type of application architecture. These systems can either be used to translate from a formal programming language into machine language, or to translate from one natural language into another, such as English to French, or French to English. These systems interpret and process languages, and the best-known example of this type of system is a compiler. Compilers often involve a generic architecture that involves a lexical analyzer, a symbol table, a syntax analyzer, a syntax tree, a semantic analyzer, and a code generator. All of these components work together to translate one language into another (Sommerville, 2016).

The Model-View-Controller (MVC) architectural pattern is most commonly used when there are various ways to view data. In this pattern, three components interact together: the model, the view, and the controller. The model manages the data and associated operations. The view determines how data is presented. The controller manages interactions such as mouse clicks, key hits, etc., and then passes them to the view and model. A web application using this pattern would involve a controller that interprets mouse and keyboard clicks, and processes HTTP requests as a user interacts with a web page through a browser. The web application would also involve a model that uses business logic and interacts with a database. Finally, the view in the MVC architectural pattern would generate the web page and manage forms (Sommerville, 2016).

Part IV. Object-Oriented Design, Implementation, and Testing

According to Sommerville (2016), the object-oriented design process includes 5 major activities: defining the system context and interactions, designing the system architecture, identifying the principal objects in the system, developing design models, and specifying interfaces. Defining the system context and interactions is the first step in the design process.

This stage determines how system functionalities will be achieved and how the system will be structured. Boundaries are also defined in this step (Sommerville, 2016). Models can be created for both the system context, as well as system interactions. A system context model illustrates other systems in the environment. An interaction model dynamically portrays interactions between the system and its environment (Sommerville, 2016). A context model can be as simple as a block diagram that shows associations between entities. Use case models can represent system interactions. After the system context and interactions have been defined, the next step in the design process is architectural design. This step involves defining the major components and interactions of a system (Sommerville, 2016). Next, the principal objects in the system are identified. According to *Software Engineering (10th ed.)* (2016), "use case description helps to identify objects and operations in the system." Once objects are identified, design models are developed. Structural models are used to illustrate objects and their relationships in order to give structure to the system. Dynamic models represent the system in execution. Finally, interfaces are specified. Sommerville (2016) describes interface specification as, "specifying the detail of the interface to an object or to a group of objects."

Another important phase in the software development life cycle is testing. The testing process reveals whether or not software meets its intended requirements, and aims to discover bugs before software is released. Testing is one of the methods through which software validation and verification is achieved. Another method used to validate and verify software is code inspection. Inspection is a static validation and verification process. This means that the software can be verified without needing to execute it (Sommerville, 2016). Inspection can be used at different stages in software development and specifically checks "system requirements, design models, the program source code, and even proposed test systems (Sommerville, 2016).

This process focuses on source code, and has a few advantages over testing. For example, errors can mask other errors in the testing process. Since inspection does not rely on system execution, inspection can reveal errors that might otherwise have been hidden. In addition, code can be inspected even if a system is unfinished. On the contrary, some testing requires a system to be complete, which requires additional time and money. Furthermore, the inspection process not only locates bugs, but it can also reveal "unexpected interactions between different parts of a program, timing problems, or problems with system performance" (Sommerville, 2016).

There are three major stages in the software testing process: development testing, release testing, and user testing. Development testing occurs during the development process, and is often carried out by the design and development team. This phase of testing has three stages of its own: unit testing, component testing, and system testing. Unit testing is the first step, and involves testing individual objects and program units. The goal of this step is to test every possible state of an object. The next step in development testing is component testing. Components are the result of interacting objects. This step determines whether or not component interfaces are behaving as expected. System testing is the last stage of development testing, and tests how the system behaves when components are integrated together. This step may reveal expected or unexpected emergent behavior (Sommerville, 2016). After development testing is complete, release testing is performed by a separate testing team, and evaluates a system version before it is released. Finally, user testing employs real users to test the system in their own environment and determine whether or not the system meets their expectations.

User testing can include Alpha, Beta, and acceptance testing. Alpha testing refers to a process in which a small, specific group of users is selected to interact with the development team in order to test early releases of the product and provide rapid feedback. Alpha testing is

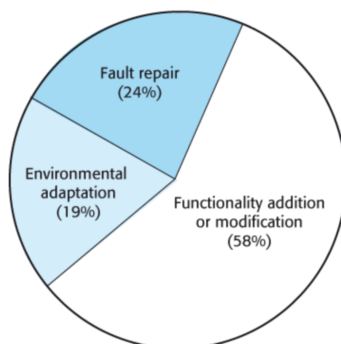
commonly used during the development of applications. Beta testing is when a larger group of users are allowed to use the software with the goal of experimenting and communicating issues that arise. A Beta version may be an incomplete version of the final product. This method of user testing is often employed for products that are used in many settings. Releasing a Beta version can also serve as marketing for a business. Finally, acceptance testing is the process through which a customer decides whether or not the product is acceptable in the desired environment. This testing process involves real data, and involves six steps: 1) define acceptance criteria; 2) plan acceptance testing; 3) derive acceptance tests; 4) run acceptance tests; 5) negotiate test results; 6) reject/accept a system (Sommerville, 2016). Once these steps are complete, the customer decides whether or not the system is accepted. If the system is accepted, there is a transaction of payment.

Part V. Software Evolution and Maintenance

Business needs are constantly changing. In order to stay competitive, business need to be adaptable, and allow for changes in their software in order to continue receiving value from it. While many believe that software engineering involves a single process that takes an input (requirements from a customer) and produces an output (the product a client desires), software engineering is much more than that. Sommerville (2016) describes software engineering as a spiral process that occurs throughout the lifetime of a system. This process includes requirements, design, implementation, and testing. While the first iteration of that spiral is the initial development process, subsequent iterations can describe software evolution, servicing, and finally the retirement of a system.

Once a system has been delivered, changes will need to be made. Changes arise as a result of code errors, the need to correct design errors, the desire to add new features, or to

correct specification errors (Sommerville, 2016). There are three major types of software maintenance: 1) fault repairs; 2) environmental adaptation; and 3) functionality addition. Fault repairs refer to maintenance activities that resolve bugs and reduce vulnerabilities. The cheapest type of errors to fix are coding errors. The most expensive are requirements errors. Design errors fall somewhere in the middle. Environmental adaptation refers to system maintenance as a response to new platforms, hardware, software, etc. Finally, functionality addition is maintenance that revolves around the addition of new features to the software and responds to new system requirements (Sommerville, 2016). The distribution of effort required by these forms of maintenance has been evaluated by Davidsen and Krogstie (2010), and the results of their study are found below. The figure illustrates that, "new or changed requirements generally consumes most maintenance effort" (Sommerville, 2016).



(Figure 9.12, Sommerville, 2016)

Software development is the process of building or modifying software to meet system requirements. According to Software engineering (10th ed.). (2016), "The only difference between initial development and evolution is that customer feedback after delivery has to be considered when planning new releases of an application". Once the development process is complete, the next step in the software life cycle for business systems is evolution. This phase

entails making large changes to the software architecture and functionality (Rajlich & Bennett, 2000). Changes are prompted by stakeholders after they use the product, and the evolution phase involves implementation of these changes. The next phase in the cycle is software servicing. During this phase, only small, essential changes are made, as system maintenance decreases and changes become more costly. When the cost of making changes exceeds the expected benefit of the change, the software is retired (Sommerville, 2016).

A general model of software evolution involves a number of activities. First, there is a proposed change to the system. The software evolution process is often integrated with change management. Therefore, the proposed change is evaluated, and impact analysis is executed in order to weigh the costs and benefits of the change. The proposed change is then accepted, rejected, or modified. Once a change is accepted, the release of the new software version is planned. This step involves considering the type of change, and how it will be implemented. There are three basic types of software changes: 1) fault repair, which is basically debugging; 2) adaptation, which requires modifying the software in response to environmental changes such as updates in hardware or software related to the system; and 3) new functionality, where a new functionality is added to the software. Once planning is complete, the change is implemented, and then the new system is released (Sommerville, 2016).

Because technology has evolved significantly since businesses began relying on computers for operation in the 1960's, many systems are now outdated. Legacy systems are those that are built on old software and/or hardware, but are still in use because they play a critical role in business functions (Sommerville, 2016). Often, legacy systems employ languages and technology that are no longer in use in modern development projects. Some systems may even depend on older hardware, and they are too expensive or risky to replace. When a business

relies on a legacy system, it is important to evaluate the quality and value that the system or system components bring to the business. Once considerations are made, there are four strategic options that a business can pursue: scrap the system, leave the system unchanged and continue with regular maintenance, reengineer the system, or replace all or part of the system with a new one (Sommerville, 2016). One method of evaluating systems before pursuing a strategy utilizes a graphical representation that clusters the systems relative to quality and business value coordinates.

The four system clusters that this graphical representation includes are: 1) low quality, low business value; 2) low quality, high business value; 3) high quality, low business value; and 4) high quality, high business value (Sommerville, 2016). A system that falls into low quality, low business value will be expensive and provide a low rate of return. The recommended strategy for systems in this quadrant is that they are scrapped completely. A system that falls into the low quality, high business value quadrant adds significant value to the business, but is expensive to maintain. These types of systems should be reengineered through re-documentation, refactoring, translation of the old programming language to a more modern one, or updating the structure and values of data in the system (Sommerville, 2016). A system that falls into the quadrant for high quality, low business value doesn't necessarily provide significant value to the business, but is relatively cheap to maintain, so normal system maintenance should continue unless expensive changes arise. The final quadrant is reserved for systems that are high quality and provide high business value. These systems are essential to the business and relatively cheap to maintain, so normal system maintenance should continue (Sommerville, 2016).

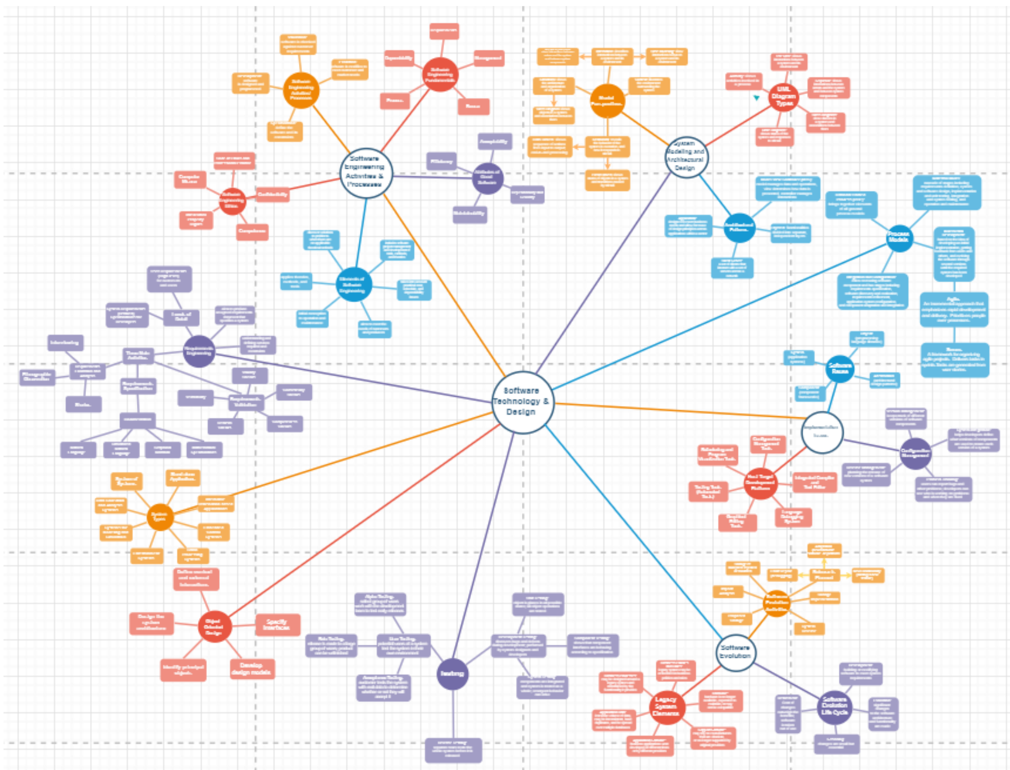
Occasionally, the best strategy for a legacy system is reengineering. This process makes a legacy system easier to understand and maintain, so that necessary changes can be implemented. It may seem that the simplest solution to a legacy system is replacement, but reengineering actually reduces both risk and cost. Reducing risk is important when legacy systems are essential to the success of a business. The software reengineering process involves a number of steps. One step in the process is code translation. This involves a tool that translates an old programming language to a more modern version of the same language, or to a different language altogether. Another step in the process is reverse engineering. This process analyzes and extracts information from a program to document organization and functionality. Program structure improvement analyzes and modifies the control structure so that it's easier to understand. Program modularization refers to the process of grouping related parts of a program and eliminating redundancies. Finally, data reengineering redefines database schemas and converts databases to a new structure. This process also involves cleaning up data, and it can be very long and expensive, especially for legacy systems that have vast quantities of data (Sommerville, 2016). While software reengineering is a complex process, it is capable of taking a legacy system and transforming it into an improved version of itself.

Part VI. Reflection

Software engineering concepts are vital in developing and maintaining software efficiently. While many individuals view software development as the process of designing and building the initial software product, it is much more than that. In fact, software engineering involves all phases of software production from the initial need and design, all the way through implementation, operation, and maintenance. As a future software developer, it will be important for me to know and understand the types of systems that I will be working with. For

example, is the system, new, embedded, critical, custom, small, large, legacy, etc.? What does the documentation for the system look like? How can both the current and new systems be modeled? What development methods does the team currently use? What has been used in the past? Are agile methods or Scrum involved, or is the development of the system plan-based? Will open source tools be utilized? Are they appropriate? What types of testing are currently in use? Is there a way to automate testing or other processes? How are changes implemented and communicated among the team? Asking these questions will help me be an asset to future development teams and a contributing member who uses critical thinking to employ the best software engineering methods possible for the system at hand.

Part VII. Concept Map



Click here to view the map in your browser.

Resources

Davidsen, M. G., and J. Krogstie. 2010. "A Longitudinal Study of Development and Maintenance." Information and Software Technology 52 (7): 707–719. doi:10.1016/j.infsof.2010.03.003.

Rajlich, V. T., and K. H. Bennett. 2000. "A Staged Model for the Software Life Cycle." IEEE Computer 33 (7): 66–71. doi:10.1109/2.869374.

Raymond, E. S. 2001. The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. Sebastopol. CA: O'Reilly & Associates.

Sommerville, I. (2016). Software engineering (10th ed.). Retrieved from https://www.vitalsource.com/