Final Paper: Software Requirements and Analysis Concept Map & Paper

Alicia Piavis

CST 304: Software Requirements & Analysis

Amr Elchouemi

11/30/2019

Software Requirements and Analysis Concept Map & Paper

When engaging in the software development life cycle (SDLC), it is important to consider guidance by the core set of software development principles. According to Pressman & Maxim (2015), these principles "establish a philosophical foundation that guides a software team as it performs framework and umbrella activities, navigates the process flow, and produces a set of software engineering work products." The principles serve as shared values and rules that ensure both the development team and stakeholders have a common goal, and software engineers are able to "analyze a problem, design a solution, implement and test the solution, and ultimately deploy the software in the user community" (Pressman & Maxim, 2015). These guiding principles direct each phase of the SDLC including communication, planning, modeling, coding, testing, and deployment. The software development principles for practice and process help facilitate an effective software development life cycle, and ultimately, customer satisfaction.

There are a number of general software development principles including: 1) Keep it simple; 2) Provide value to end users; 3) Employ a high level of abstraction; 4) Maintain the vision of the project and product; 5) Recognize that others consume what you produce; 6) Be open to the future; 6) Plan ahead for reuse; and 7) Think! (Pressman & Maxim, 2015). While these are useful, they are very high level, and may be better broken down into principles that guide process, and principles that guide practice (Refer to Figure 1.3). The first principle that guides process is "be agile." This means keeping the approach simple and emphasizing economy of action (Pressman & Maxim, 2015). In evaluating similarities between the software development process and the instructional design process, researchers Adnan & Ritzhaupt (2018) describe agile as "an approach to develop software by empowering and trusting people, encouraging continuous feedback, and recognizing change as a process." The second principle

guiding process is to focus on quality. In other words, the end-goal for every activity, action and task should be quality. The third principle is "adapt." This is important as changes in the problem, project, requirements, and people come along. The fourth principle is to build an effective team that self-organizes and has mutual trust and respect. The fifth principle is establishing communication and collaboration expectations. The sixth principle of process is to manage change. It is important to determine how changes will be requested, assessed, approved, and implemented. The next principle is "assess risk." This includes the creation of contingency plans. And finally, the last principle is to provide value to others (Pressman & Maxim, 2015). If requirements are ambiguous or have omissions, it is difficult to provide a product that stakeholders find valuable.

The other set of over-arching principles in software development are principles that guide practice. According to Pressman & Maxim (2015), "Software engineering practice has a single overriding goal—to deliver on-time, high-quality, operational software that contains functions and features that meet the needs of all stakeholders." The first principle the guides practice is "divide and conquer." This entails breaking a large problem down into smaller chunks and employing separation of concerns. The next principle is the use of abstraction to simplify complex system elements in order to improve communication. The third principle that guides practice is to strive for consistency. This is because a recognizable context makes software easier to use. The fourth principle is the focus on the transfer of information. This involves paying special attention to the design and implementation of interfaces. The next principle is to employ modularity. This means that each component of the project should focus on one well-defined part of the system. The sixth principle requires looking for patterns. Design patterns can be used for recurring problems to reduce work and enhance communication. The next principle

involves approaching the problem from different perspectives, since this will lead to greater insight and potentially expose errors. Finally, software should be developed with maintenance in mind, as maintenance is easier when the previous principles of practice and process have been followed (Pressman & Maxim, 2015).

While the principles of process and practice guide the SDLC at a very high level, there are more specific principles that guide the software development process at a lower level. These principles guide activities such as communication, planning, modeling, construction, testing, and deployment (Refer to Figure 1.3). Principles that guide communication activities involve listening, preparing, having a facilitator, meeting face-to-face, taking notes and documenting, striving for collaboration, staying focused on the present topic, trying a picture as an alternative to a verbal description, moving on, and negotiating (Pressman & Maxim, 2015). Since communication is one of the biggest causes of project failure, it is important to be conscientious of these guiding principles throughout the development process. Principles in the planning phase include understanding the scope of the project, involving stakeholders, planning iteratively, making estimates based on what you know, considering risk, being realistic, adjusting granularity of tasks, defining how quality assurance will be managed, having protocols for change, and tracking daily progress (Pressman & Maxim, 2015). It is important that stakeholders are involved throughout planning activities. As Adnan and Ritzhaupt (2018) state, "The user is an active participant in the decision-making process and collaborative decision making is important in an agile environment."

The communication, planning, and modeling phases of the SDLC constitute part of what is known as requirements engineering. Requirements engineering is an important step in the software development lifecycle, as it aims to "establish a solid foundation for design and

construction” (Pressman, & Maxim, 2015). The requirements resulting from this process provide direction for a software project. Additionally, it is important to focus on quality requirements, as the failure of software projects is often attributed to inadequate software gathering techniques (Lane, O'Raghallaigh, & Sammon, 2016). The activities involved with requirements engineering are typically carried out during the communication and modeling phases of a software project, and the process involves software developers and stakeholders, such as customers, managers, and end users (Refer to Figure 1.3). The process is iterative in nature, since requirements may evolve as a system is built and stakeholders and developers acquire a better understanding of a system, as well as customer needs. According to Bee Bee, Bernardo, and Verner (2010), “Ensuring requirements that are complete and correct is important for the software development team so that the correct system can be properly developed in accordance with users’ wants and needs.” Requirements engineering is important because it helps ensure that developers and stakeholders are working towards a common goal, that the right solution is being developed, and that unnecessary use of resources, such as time and cost are avoided (Refer to Figure 1.3).

The steps involved in requirements engineering involve inception, elicitation, elaboration, negotiation, specification, validation, and management (Refer to Figure 1.3). Inception typically begins when a business need is identified. Elicitation involves eliciting information from stakeholders in order to establish objectives for a project. As *Collaborative Practices for Software Requirements Gathering in Software Startups* states, “collaborative practices may enhance the requirements gathering process since entrepreneurs rarely envision all aspects that should be taken into account themselves” (Chanin, Pompermaier, Sales, & Prikladnicki, 2019). Requirements elicitation should be a collaborative process, since this is when stakeholders

discuss and prioritize their wants and needs. These wants and needs will later translate into requirements that guide the project. A number of different activities can be used to elicit requirements. For example, sometimes stakeholders will use a voting system based on priority points to rank requirements. According to Pressman & Maxim (2015), the job of a requirements engineer is to identify areas of commonality and areas of conflict or inconsistency between stakeholders. Further methods that guide requirements gathering include meetings, quality function deployment (QFD), and developing usage scenarios, or use cases.

Quality function deployment (QFD) is a method used to help translate customer wants and needs into software requirements (Pressman & Maxim 2015). This process is important to reduce ambiguity. For example, Spezialetti states in *Thinking About Asking: Encouraging a Questioning Approach to Requirements Gathering and Problem Solving* (2016) that, "real-world problems are frequently ill-defined, filled with ambiguity and open-ended. Successfully translating these problems into computing solutions involves both recognizing the issues that need to be addressed and asking questions to resolve the unknowns." QFD involves three types of requirements: normal requirements (stated requirements that align with the goals of the project), expected requirements (requirements that are expected, but not explicitly stated), and exciting requirements (those that exceed the customers' expectations). Elicitation also typically utilizes usage scenarios, or use cases. These describe how the system will be used from a user's perspective. The end goal of the elicitation phase is to develop a set of elicitation products that include: 1) "a statement of need and feasibility"; 2) "a bounded statement of scope for the system or product"; 3) "a list of customers, users, and other stakeholders who participated in requirements elicitation"; 4) "a description of the system's technical environment"; 5) "a list of requirements (preferably organized by function) and the domain constraints that applies to each";

6) “a set of usage scenarios that provide insight into the use of the system or product under different operating conditions”; and 7) “any prototypes developed to better define requirements” (Pressman & Maxim 2015).

Effective use cases can serve as a powerful tool for establishing requirements and guiding the development process. While some individuals believe that interviews with stakeholders are enough to gather requirements for a project, researchers Bee Bee, Bernardo, and Verner (2010) argue that additional activities and methods are required, “although interview technique is no doubt a popular, and well-known requirements elicitation technique, unless appropriate approaches are used with the technique, the effect on requirements gathering will be incomplete.” Use cases are stories that describe how a single type of user will interact with a system under certain conditions, or in a certain scenario. These stories can be recorded as a narrative, an outline, a template, or a diagram (Pressman & Maxim 2015). Effective use cases describe the system from the user’s perspective, and focus on a simple system requirement. If user stories are too large or complex, they may be broken down into smaller stories or tasks. User stories are helpful for reducing ambiguity, as effective stories provide as much detail as possible. Some of the details that user stories include are the use case, primary actor, the goal in context, preconditions, the trigger, scenario, exceptions, priority, when the feature/function will be available, frequency of use, channel to the actor, secondary actors, channels to secondary actors, and open issues (Pressman & Maxim 2015).

After software requirements are communicated and the development process is planned, the next activity in the SDLC is modeling. High level principles that guide software modeling include: 1) The software team’s primary objective is to build software, not models; 2) Minimize the number of models you create; 3) Create the simplest model that will accomplish the task; 4)

Make models modifiable; 5) Make sure each model has an explicit purpose; 6) Adapt the models to the current system; 7) Don't worry about the models being perfect; 8) Don't over-emphasize correct syntax for the model; 9) Follow your instincts if it feels like a model is not right; and 10) Get feedback as soon as possible (Pressman & Maxim, 2015). Investigators Watts and Kellner (1989) also describe characteristics of effective process models. According to their research, effective models, "Represent the way the work is actually (or is to be) performed, provide a flexible and easily understandable, yet powerful framework for representing and enhancing the process" and are "refinable to whatever level of detail is needed."

Models can be divided into two classes, each with their own guiding principles. The two classes of models are requirements models and design models. The principles that guide requirements modeling are: 1) "The information domain of a problem must be represented and understood'; 2) "The functions that the software performs must be defined"; 3) "The behavior of the software (as a consequence of external events) must be represented"; 4) "The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical fashion)"; and 5) "The analysis task should move from essential information toward implementation detail" (Pressman & Maxim, 2015). These principles help ensure that requirement models enhance the plan proposed to deliver the intended requirements of a software project. The principles that guide design modeling are: 1) "The design should be traceable to the requirements model"; 2) "Always consider the architecture of the system to be built"; 3) "Design of data is as important as design of processing functions"; 4) "Interfaces (both internal and external) must be designed with care"; 5) "User interface design should be tuned to the needs of the end users. However, in every case, it should stress ease of use."; 6) "Component-level design should be functionally independent"; 7) "Components should be

loosely coupled to one another and to the external environment"; 8) "Design representations (models) should be easily understandable"; 9) "The design should be developed iteratively"; and 10) "Creation of a design model does not preclude an agile approach" (Pressman & Maxim, 2015). The requirements are important, as design has the potential to heavily affect end users. As Pressman and Maxim (2015) state, "a poor interface design often leads to the perception that the software is 'bad.'"

Requirements modeling is part of the elaboration phase of requirements engineering, and involves refining the requirements analysis models. The purpose of these models is to illustrate the informational, functional, and behavioral requirements of the system (Pressman & Maxim 2015). The models should be flexible, because requirements may change as the project evolves and developers and stakeholders gain a better understanding of the system and customer needs. Analysis models are comprised of scenario-based, class-based, and behavioral elements. Scenario-based elements are derived from use cases, and focus on developing requirements from the user's point of view. This part of the model is often developed first. Class-based elements utilize objects that will be manipulated by user interactions with the system. Behavioral elements illustrate how the system behaves in response to events. Each model shows the system from a different perspective, supporting the opportunity to generate requirements that are clear and purposeful. The goal of these models is to bridge the gap between the system description (acquired during inception and elicitation), and the design models. According to Pressman & Maxim (2015), requirements models serve three purposes: 1) "to describe what the customer requires"; 2) "to establish a basis for the creation of a software design"; and 3) "to define a set of requirements that can be validated once the software is built." These models are often the first technical representation of a system, and they typically involve the use of UML (unified

modeling language) diagrams for consistency in communication. As Filipova and Nikiforova (2019) state, "Large system modelling enables software developers to understand its structure, behaviour and its core elements. For these purposes, spatial placement of Unified Modelling Language (UML) diagram elements is used to determine how well the software system will be understood."

A commonly used scenario-based UML model is the use case diagram. Use case diagrams depict a system from a user, or actor's point of view. According to *Definition of the Criteria for Layout of the UML Use Case Diagrams* (2019), "Use cases represent what the customer wants system to do, in other words, customer system requirements. At the high level of abstraction, use cases show which purposes the developed system is meant for." A use case diagram is a simple diagram that provides an overview of the interactions that will occur between actors and the system (Pressman & Maxim, 2015). The use cases in the diagram come from functional and behavioral requirements that are established during the inception and elicitation phase of the requirements engineering process. Figure 1.1 (Wells, 2009) provides an example of a use case diagram. The ovals represent specific use cases, and the stick figures represent different actors that will interact with the system. In Figure 1.1, a simple banking system is represented in which there are three actors, a customer, a cashier, and a bank manager. The customer is associated with four use cases: depositing money, withdrawing money, checking their account balance, and taking out a loan. A cashier needs to be able to update an account balance, deposit money, withdraw money, and check the account balance. A bank manager needs to be able to check an account balance and assist a customer in taking out a loan (Wells, 2009). The arrows show which use cases a specific actor needs to carry out. Use case diagrams

provide a big picture perspective of a system and do not include details specific to each use case (Pressman & Maxim, 2015).

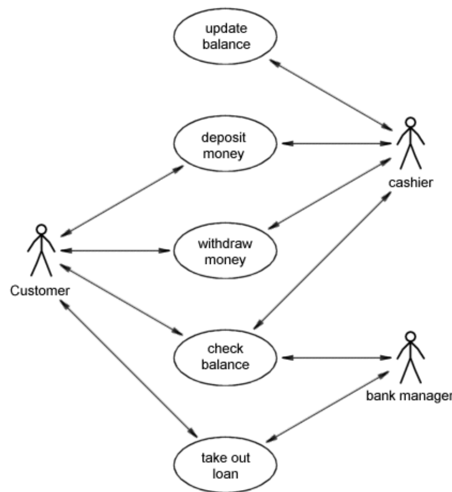

Figure 1.1 UML Use Case Diagram Example (Wells, 2009)

Activity diagrams typically supplement use case diagrams in that they provide detail regarding the flow of interactions for a specific use case. They are similar to flow charts, but can also depict concurrent flows (Pressman & Maxim, 2015). Researchers Lima, Tavares, and Nogueira discuss their effectiveness in developing both high and low-level designs for a system. They also state that, “the activity diagram is one the most used among all UML diagrams. It is a behavioral diagram focused on the description of a coordinated dynamics with emphasis on the sequence of actions and conditions” (2019). Activity diagrams are different than class and use case diagram in that they are dynamic, since they represent the flow of control within a system. In Figure 1.2 (Ankit, n.d.), the solid black dot represents the beginning of flow in the system, while the black dot with an addition circle around it represents the final node, and the end of the activity. This diagram depicts the flow of control for a use case in which a user interacts with an application that provides customized music and wallpaper, based on their current emotional state. The flow of control begins when the user opens the application. This rounded rectangle

represents an action node. Action nodes corresponds to tasks that the system performs. The arrows depict the direction of flow. An activity node may not be initiated until the node before it has been completed. A hollow diamond represents a decision node. The flow of control here depends on specific conditions being met. In the example provided, the system will either allow a user to take a photo with their webcam, or direct them back to the sign in depending on whether or not their credentials were verified. In a second decision node, after the app detects a user's emotional state from the photo, the system will either retrieve music or set the wallpaper to something reflecting that emotion, or possibly both. A fork represents concurrent activities. There are no forks in Figure 1.2, but a fork is represented as a solid horizontal black bar. Forks show the separation of activities that will execute concurrently (Pressman & Maxim, 2015). A join is the opposite of a fork, in that it represents the flow of control coming back together after concurrent activities are carried out. Solid horizontal lines in an activity diagram can be used to show the execution of parallel activities (Pressman & Maxim, 2015). Together, use-case diagrams and activity diagrams are used for scenario-based modeling.

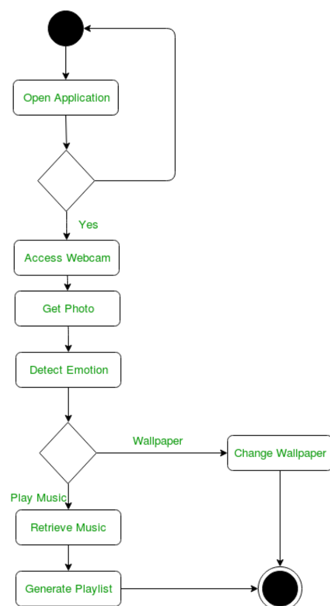

Figure 1.2 UML Activity Diagram Example (Ankit, n.d.)

Class-based methods involve UML models that are static and provide a structural view of the system. Class UML diagrams shows the relationships between objects and classes, as well as the attributes and operations that classes possess. According to *Smart Bound Selection for the Verification of UML/OCL Class Diagrams* (2019), "UML class diagrams are arguably the most commonly used models for describing the specification of a software system." Class-based methods are important, as they represent objects in a system from a customer's point of view. Customers can also evaluate UML class-diagrams and provide useful feedback. The elements of class-based methods include classes, objects, attributes, operations, class-responsibility-collaborator (CRC) models, collaboration diagrams, and packages. The main components of a class diagram are boxes, which represent classes and interfaces. Each box is divided into three sections to show the name of the class, the attributes, and the operations. Attributes are characteristics that describe a class, while operations are behaviors that a class exhibits (Refer to Figure 1.3).

Often times, a method called CRC (class-responsibility-collaborator) modeling is used to help identify classes, objects, attributes, and operations (Refer to Figure 1.3). Emphasizing the importance of user input throughout the requirements engineering process, Ambler (1995) states, "I am a strong proponent of active user participation in the system development process. To that end, I have always found that CRC (class responsibility collaborator) modeling is an incredibly effective technique for getting users involved in the OO analysis process." The CRC modeling activity usually involves index cards where the name of a class is written at the top, responsibilities of the class are listed on the left-hand side, and collaborators are listed on the right-hand side. According to Pressman & Maxim (2015), responsibilities are "the attributes and operations that are relevant for the class." Classes can achieve their responsibilities in two ways:

1) by using their own operations to influence their own attributes, or 2) through collaboration with other classes. There are five guidelines used to determine responsibilities for a class: 1) the system should be distributed across classes so that objects only do a few things (this increases system cohesiveness); 2) responsibilities should be stated as generally as possible to take advantage of the principle of inheritance; 3) data, and the operations related to it, should be packaged within a class to take advantage of the principle of encapsulation; 4) information about one class should not be distributed across the system (this makes it difficult to test); and 5) classes should share responsibilities when appropriate (Pressman & Maxim, 2015). By determining responsibilities strategically during the requirements engineering process, more effective design patterns and development strategies can be implemented later on in the software development life cycle.

Once potential classes, objects, attributes and operations have been identified, UML class-diagrams are constructed to verify the requirements. In the class-based UML diagram provided in Figure 10.5 from Pressman & Maxim (2015), the system is comprised of four classes: Wall, WallSegment, Window, and Door. The Wall class is connected to WallSegment, Window, and Door by solid lines. These lines represent relationships called associations. Associations are classes that are related to each other in some way. In this case, the association is a generalization, since Wall is a more generalized class that WallSegment, Window, and Door are part of. In addition, the relationships shown in Figure 10.5 have multiplicities (Refer to Figure 1.3). Multiplicities illustrate the number of objects of a class associated with another class, and are represented by numbers on either side of an association (Pressman & Maxim, 2015). In Figure 10.5, the multiplicity closest to the WallSegment Class, "1..*", illustrates that an instance of Wall can have one or more WallSegments (the asterisk represents an unlimited

bound). The number closest to the Wall in the same relationship is "1", meaning that an instance of WallSegment can only be associated with one Wall. In the relationship between the Window class and the Wall class, "0..*" means that an instance of Wall can have 0 or more windows. The "1" on the other side of the relationship means that an instance of Window can only be associated with one Wall. The multiplicities in the Wall and Door relationship mirror that of the Wall and Window relationship.

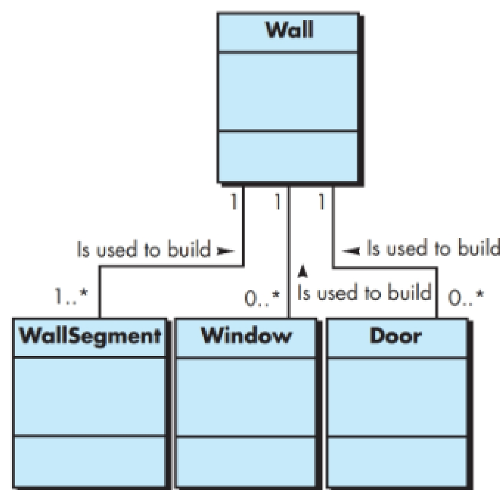

Figure 10.5 Multiplicity (Pressman & Maxim, 2015)

UML class diagrams can also show dependency relationships. Figure 10.6 from Pressman & Maxim (2015) below shows a dependency relationship between the DisplayWindow class and the Camera class in a SafeHome home security system. A dependency is a relationship in which changes to one class may result in changes to its associated class (Pressman & Maxim). In addition, a dependency can mean that the behavior of one class depends on another class. The relationship between the DisplayWindow and the Camera class is a dependency since the Camera requests a password from the user, and as long as the password is verified, the Camera provides access to the Camera view from a DisplayWindow on the security system. Therefore,

DisplayWindow depends on Camera, because Camera needs to request and verify a password before DisplayWindow will be updated.

FIGURE 10.6

Dependencies

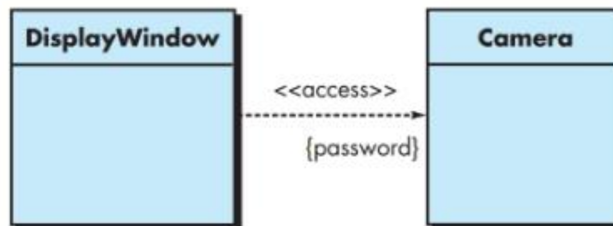

Figure 10.6 Dependencies (Pressman & Maxim, 2015)

Finally, it is important to consider the use of analysis packages when implementing class-based methods. According to IBM, analysis packages are utilized for two reasons: to organize a model so that it is easier to understand and navigate, and to model the architecture of a system and show different layers or subsystems (2014). Analysis packages allow use cases and analysis classes with commonalities to be grouped together (Pressman & Maxim, 2015). Figure 10.7 provided by Pressman & Maxim (2015) provides an example of three analysis packages for a video game system. One package is named Environment, and contains classes that relate to the game environment. Another package is utilized to group classes related to the rules of the game. A third package is used to organize classes that relate to the characters in the game. Grouping and organizing classes in this way makes complex systems more manageable. Analysis packages share some notations with attributes and operations in that a plus, minus, or hash symbol before the name of an analysis packages represents that a package is either publicly visible, private, or accessible only to elements within the same package (Pressman & Maxim,

2015). Packaging similar use cases and analysis classes together is an example of encapsulation in object-oriented programming.

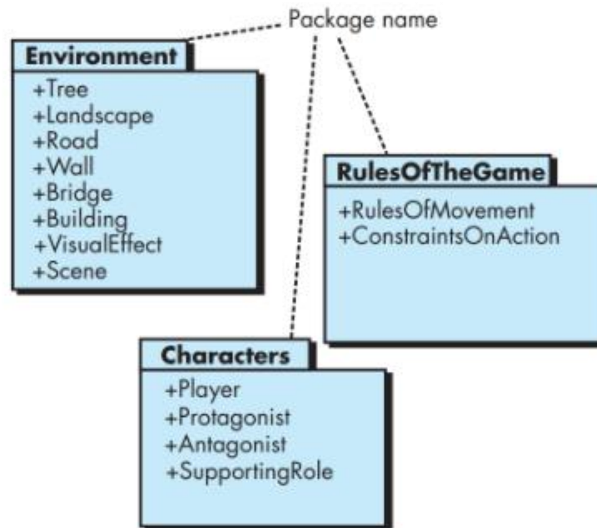

Figure 10.7 Packages (Pressman & Maxim, 2015)

While scenario-based methods and class-based methods are used to represent a system from the user's perspective and a structural perspective respectively, behavioral UML models are utilized to depict state changes within a system dynamically (Refer to Figure 1.3). These models supplement scenario-based and class-based methods, and are used to represent the state of a class, as well as the state of a system. State diagrams are used to illustrate state changes within a specific class. The state of an entire system, rather than an individual class, is usually represented with a sequence diagram. Pattern-based methods can also be implemented in the requirements modeling process. These methods take advantage of design and architectural patterns that already exist within the same domain, or exist within other domains, but have overlapping functionalities. Implementing design patterns makes use of existing code to reduce

the cost and time for a project. Lastly, special consideration must be made when designing for web and mobile apps. These models are specifically adapted to address requirements concerning content, interactivity, functionality, navigation, and configuration.

A state diagram is one type of behavioral UML diagram, which illustrates the dynamic behavior of a system. State diagrams are used to represent how events can cause the state of specific objects to change. *An Approach to Transforming UML Model to FSM Model for Automatic Testing* (2008) states, “As one of the behavioral models, state diagram is often used to model the life cycle of certain object. Many elements are involved.” The state of a class can be described in two ways: active or passive. Active state describes an object during transition or processing (Pressman & Maxim, 2015). Passive state is like a snap-shot in time of an object’s attributes. State diagrams are composed of active states, events, guards, and actions (Refer to Figure 1.3). Figure 11.1 from Pressman and Maxim (2015) provides an example of a state diagram for the ControlPanel class in the SafeHome security system. In the example provided, the system has four active states: reading, comparing, locked, and selecting. The events in the diagram are “password entered” and “activation successful.” These events trigger changes in state within the system. The state diagram for the ControlPanel class also utilizes guards and actions. A guard is a conditional check preventing a transition from occurring until a specific condition is met (Refer to Figure 1.3). In the case of ControlPanel, a password needs to be verified before a user can select the next option. An action occurs simultaneously with, or as a result of, a state transition. An example of an action in Figure 11.1 is validatePassword(). This may be an operation of the ControlPanel object.

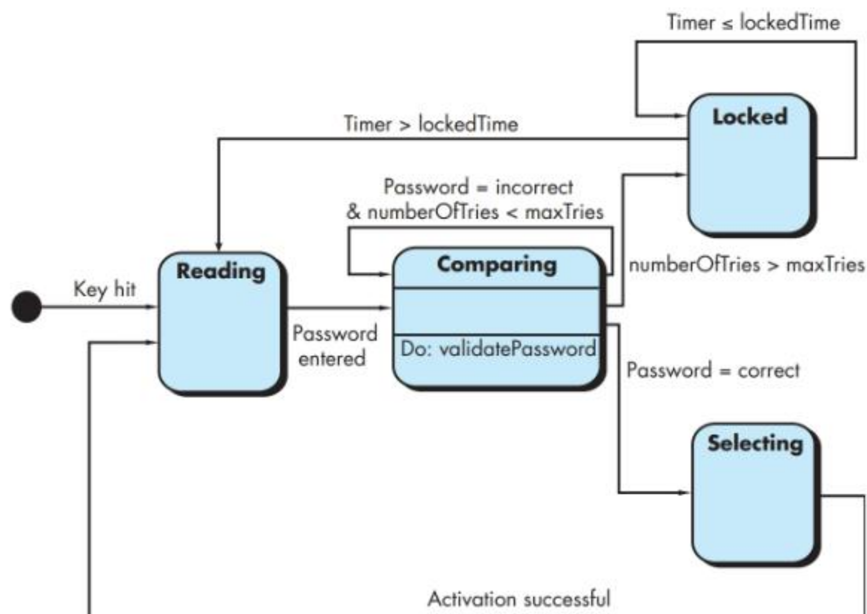

Figure 11.1 State diagram for the ControlPanel class (Pressman & Maxim, 2015)

While state diagrams focus on state changes within an individual class, sequence diagrams show the behavior of a system as a whole. In a way, sequence diagrams are a shorthand way of depicting use cases. Sequence diagrams are helpful because they can be used to generate input and output events, which are then used to drive system design (Refer to Figure 1.3). These diagrams are also dynamic, and show how transitions occur between objects. Transitions in a sequence diagram are shown relative to time. Sequence diagrams are composed of objects, arrows (which represent events), narrow vertical rectangles (which represent processing time for an activity), a vertical timeline, events, and active states. In Figure 11.1 provided by Pressman and Maxim (2015), the initial state of the system is "system ready". The first class the system employs in the sequence diagram is Homeowner. The Homeowner class triggers the event "Password entered" when a user enters their password. This causes the state of the ControlPanel to change to "reading". At this stage, a new event is triggered, "Request lookup". This event is passed to the System object to perform the password lookup in the

database. The System object then returns the result of the lookup to the ControlPanel. At this point, the active state of the ControlPanel is changed to “comparing”. If the password is correct, the “Password = correct” event is sent to the System. This causes the System to pass the “Request activation” event to the Sensors object, which sends the “Activation successful” event back to the Homeowner so that they can activate the sensors in the SafeHome system.

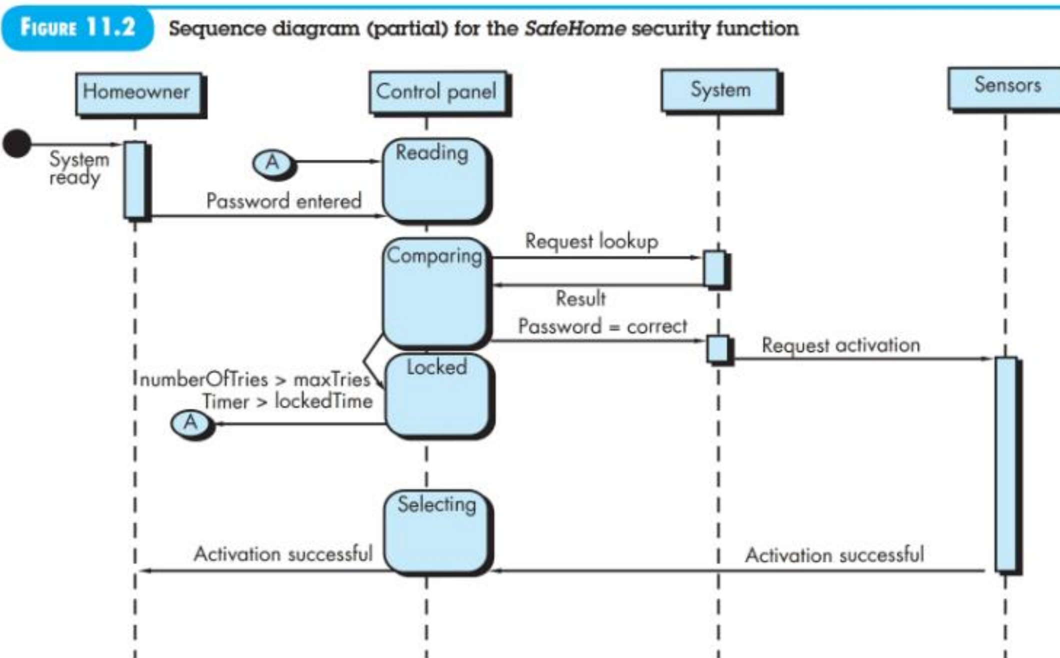

Figure 11.2 Sequence diagram (partial) for the SafeHome security function (Pressman & Maxim, 2015)

Patterns can also be utilized in UML requirements models. Analysis patterns allow developers to reduce the time and cost on a project by reusing code. Typically, a domain analysis will be conducted in which developers research similar solutions or components that have previously been used within the same domain (ex. healthcare, education, business), or outside of a domain (Refer to Figure 1.3). Utilizing analysis patterns takes advantage of existing domain knowledge in order to reapply knowledge when a new, but similar problem arises (Pressman & Maxim, 2015). Analysis patterns are not created, but discovered upon developing a

solution to a problem. Documenting an analysis pattern involves describing the general problem to which the pattern applies, the proposed solution, assumptions and constraints of the pattern, the motivation behind using the pattern, advantages and disadvantages of implementing the pattern, and examples of practical applications of the pattern. Repositories are often used to store patterns, so that they are easily accessible and searchable for future use. An example of an analysis pattern is the sensor-actuator pattern described by Pressman and Maxim (2015), in which sensors and actuators in an embedded system are identified.

Additionally, it is important to take special consideration when modeling the requirements for web and mobile apps. Models for these types of applications are specially adapted to account for content, interactivity, functionality, navigation, and configuration (Refer to Figure 1.3). The depth of the requirements models concerning these applications typically depend on five factors related to size: 1) "the size of the company and the application increment"; 2) "the number of stakeholders"; 3) "the size of the app development team"; 4) "the degree to which members of the team have worked together before"; and 5) "the degree to which the organization's success is directly dependent on the success of the application" (Pressman & Maxim, 2015). In addition, five main classes of models are used for web and mobile apps: content models, interaction models, functional models, navigation models, and configuration models. Content models illustrate all types of content that will be displayed and used by the app. Interaction models depict how users will interact with the app. Functional models show how operations will be utilized to manipulate data. They also illustrate functions that are independent of content, but will affect the user experience. Navigation models depict how a user will navigate through the application. Lastly, configuration models represent the computing environment and the infrastructure of the application (Pressman & Maxim, 2015).

As mentioned previously, representing a system from multiple perspectives is ideal for generating quality requirements and setting the stage for an effective design process. Behavioral models allow stakeholders and developers to view a system dynamically, by showing state changes within a class in responses to events, as well as by showing how different objects within a system interact with one another in response to events that trigger actions. As Pressman and Maxim (2015) state, "Your insight into software requirements grows in direct proportion to the number of different requirements modeling dimensions." Through the use of behavioral UML models, developers and stakeholders can evaluate whether or not requirements have been properly specified, and whether or not they align with the overall objectives for the project. The goal of utilizing behavioral methods is to generate graphical representations of a system, which leads to a better understanding of the requirements and paves the way for an effective design process.

Once scenario-based, class-based, and/or behavioral requirements models are initiated, the requirements engineering process moves into the negotiation phase. During this time, stakeholders and developers assess the cost, risk, and internal conflicts associated with requirements. Negotiation is an iterative approach that ensures compromises are made and everyone walks away happy with the agreed upon requirements. Often, negotiations include discussions about requirements monitoring and validation. Requirements monitoring is a way of observing the system to support continuous validation. Monitoring often involves five tasks: 1) "distributed debugging uncovers errors and determines their cause"; 2) "run-time verification determines whether software matches its specification"; 3) "run-time validation assesses whether the evolving software meets user goals"; 4) "business activity monitoring evaluates whether a system satisfies business goals"; 5) "evolution and codesign". These tasks help validate whether

or not there are inconsistencies, omissions, or ambiguity in the requirements, if requirements conflict with one another, and whether or not the requirements are testable (Pressman & Maxim 2015). After requirements are specified and validated, the final stage in requirements engineering in management. This step tracks and controls the requirements and determines how changes to requirements will be handled as the project proceeds.

After requirements have been negotiated and specified, the next step in the SDLC is construction. This process involves preparing to code, coding, validating the code, and testing the software. Principles that guide the preparation phase include: understanding the problem and basic design principles, picking a programming language, and writing unit tests (Pressman & Maxim, 2015). Picking a programming language is an important step of the process because the language needs to support the software and the environment, and provide tools that will make the construction process easier. In *Sharing and Preserving Coding Best Practices through Programmer Data Analytics* (2018), Bashani and Perera describe the many benefits of following coding guidelines. They state that the use of coding best practices can, “Reduce the number of unnecessary code churn, reduce obvious defects in the code, make the code easy to maintain and understand, reduce the time to market the software, improve the quality of the product and many more.” The principles that guide coding are: 1) selecting efficient algorithms; 2) considering pair programming; 3) selecting advantageous data structures; 4) understanding the architecture and interfaces; 5) using simple conditional logic; 6) making nested loops testable; 7) strategically naming variables and following standards and conventions; 8) writing self-documenting code; and 9) creating a readable layout through the use of whitespace (Pressman & Maxim, 2015). Three principles are used to guide the validation process: conducting code walkthroughs, performing unit tests and fixing errors, and refactoring the code (Pressman & Maxim, 2015).

Following construction in the SDLC is testing. The testing process has its own guiding principles: 1) "All tests should be traceable to customer requirements"; 2) "Tests should be planned long before testing begins"; 3) "The Pareto principle applies to software testing" (this principle states that "80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components"); 4) "Testing should begin 'in the small' and progress toward testing 'in the large'"; 5) "Exhaustive testing is not possible"; 6) "Apply to each module in the system a testing effort commensurate with its expected fault density"; 7) "Static testing techniques can yield high results"; 8) "Track defects and look for patterns in defects uncovered by testing"; and 9) "Include test cases that demonstrate software is behaving correctly" (Pressman & Maxim, 2015). On the benefits of static testing, Bashani and Perera (2018) state, "Using static analysis tools for automating code inspections can be beneficial for software engineers in various ways. Such tools can make finding bugs or software defects, faster and cheaper than manual inspections."

The final step in the SDLC is deployment. During this phase, increments of the project are released, and user feedback is gathered. These incremental releases are milestones for software development projects. Principles that guide deployment are: 1) "Customer expectations for the software must be managed"; 2) "A complete delivery package should be assembled and tested"; 3) "A support regime must be established before the software is delivered"; 4) "Appropriate instructional materials must be provided to end users"; and 5) "Buggy software should be fixed first, delivered later (Pressman & Maxim, 2015). In *Software engineering: A practitioner's approach (8th ed.)* (2015), Pressman and Maxim emphasize the importance of deploying quality software: "Customers will forget you delivered a high-quality product a few days late, but they will never forget the problems that a low-quality product caused them. The

software reminds them every day." It is important to follow deployment guidelines in order to make a good impression on stakeholders and acquire quality feedback for future iterations of the product.

There are many principles that guide software development. Higher level principles include those that guide software development process and practice. Lower level principles guide specific activities involving communication, planning, modeling, construction, testing, and deployment. Figure 1.1 illustrates these guiding principles. Beginning at the left side of the concept map, the general principles of software development are outlined. Working in a clockwise fashion, the general principles are organized into more specific principles that guide process and then practice. As the concept map continues in a clockwise direction, guiding principles for specific steps in the SDLC are illustrated--those for communication, planning, modeling, construction (coding and validation), testing, and deployment. Branching out to the right from communication, planning, and modeling, the requirements engineering process is specified. Then, scenario-based, class-based, and behavioral models are illustrated as results of the elaboration phase of requirements engineering. Finally, characteristics of patterns and web/mobile app modeling is illustrated. It is important to consider that software development is an iterative process. While these guidelines have been described in a linear fashion, they are actually repeated in cycles, or iterations, to ensure that requirements are met and that a quality product is delivered to stakeholders.

Understanding effective requirements engineering practices is valuable for producing quality software solutions and establishing positive relationships with customers. However, it is also beneficial to understand common pitfalls associated with development projects: featuritis (when the focus is on quantity of features being released, rather than quality), flexibilitis (when a

system provides so many pathways and hooks for additional features and adaptations that it is no longer easy to test and manage), and performitis (when performance is prioritized above maintainability, reliability, and security). By avoiding these pitfalls and following the seven steps in the requirements engineering process (inception, elicitation, elaboration, negotiation, specification, validation, and management), developers and stakeholders are more likely to create a solution that everyone is satisfied with. In addition, requirements will be easier to modify and refine as a project evolves. According to *Collaborative Practices for Software Requirements Gathering in Software Startups* (2019), "the software development process constantly evolves and changes, especially in relation to software which works with market uncertainties that force it to make rapid and constant changes." Therefore, it is important to remember that requirements engineering is an iterative process that encourages collaboration between stakeholders and developers.

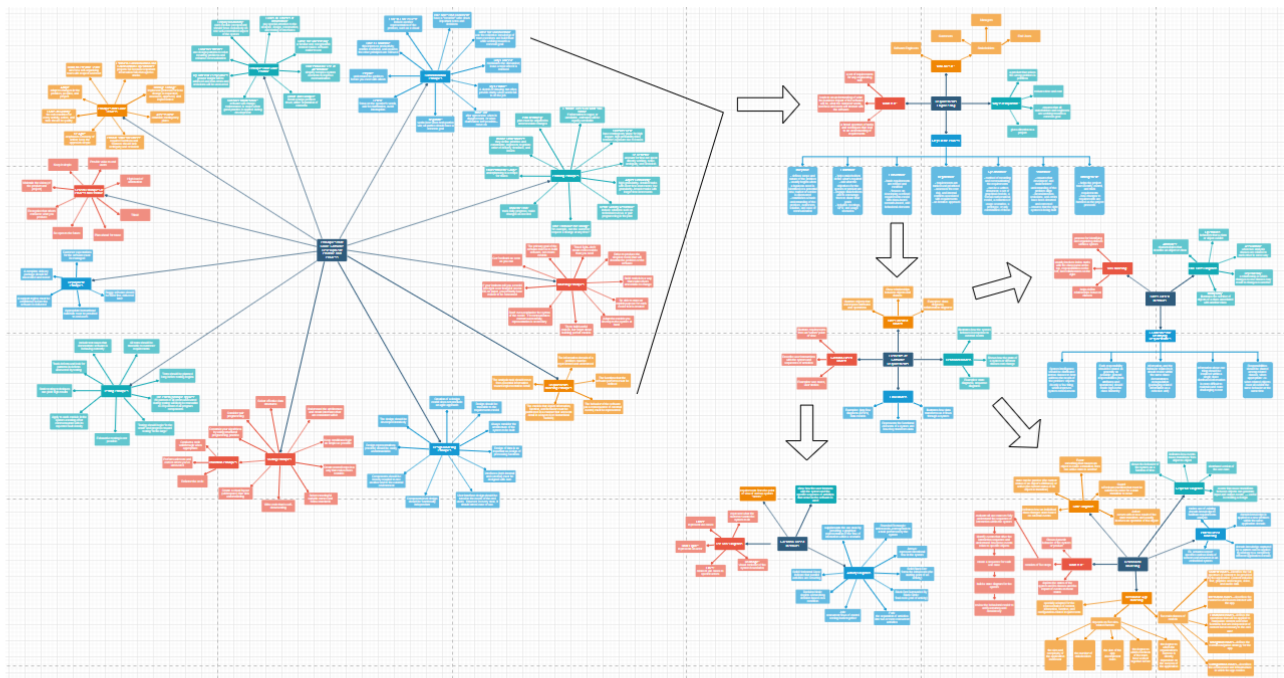

Figure 1.3 Software Requirements and Analysis

Click here to view an interactive version of this concept map.

Resources

Adnan, N. H., & Ritzhaupt, A. D. (2018). Software Engineering Design Principles Applied to Instructional Design: What can we Learn from our Sister Discipline? TechTrends: Linking Research & Practice to Improve Learning, 62(1), 77–94. https://doi-org.proxy-library.ashford.edu/10.1007/s11528-017-0238-5

Ambler, S. (1995). The case of OOCASE in design process. Computing Canada, (n26). Retrieved from http://search.ebscohost.com.proxy-library.ashford.edu/login.aspx?direct=true&db=edsbig&AN=edsbig.A17942908&site=eds-live&scope=site

Ankit, J. (n.d.). Unified Modeling Language: Activity Diagrams. GeeksForGeeks. Retrieved November 15, 2019 from https://www.geeksforgeeks.org/unified-modeling-language-uml-activity-diagrams/

Bashani, S., & Perera, I. (2018). Sharing and Preserving Coding Best Practices through Programmer Data Analytics. 2018 18th International Conference on Advances in ICT for Emerging Regions (ICTer), Advances in ICT for Emerging Regions (ICTer), 2018 18th International Conference On, 392. https://doi-org.proxylibrary.ashford.edu/10.1109/ICTER.2018.8615547

Bee Bee, C., Bernardo, D.V., Verner, J. (2010). Understanding the Use of Elicitation Approaches for Effective Requirements Gathering. 2010 Fifth International Conference on Software Engineering Advances, Software Engineering Advances (ICSEA), 2010 Fifth International Conference On, 325. https://doi-org.proxy-library.ashford.edu/10.1109/ICSEA.2010.89

Chanin, R., Pompermaier, L., Sales, A., Prikladnicki, R. (2019). Collaborative Practices for Software Requirements Gathering in Software Startups. 2019 IEEE/ACM 12th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), Cooperative and Human Aspects of Software Engineering (CHASE), 2019 IEEE/ACM 12th International Workshop On, 31. https://doi-org.proxy-library.ashford.edu/10.1109/CHASE.2019.00014

Clariso, R., Gonzalez, C.A., Cabot, J. (2019). Smart Bound Selection for the Verification of UML/OCL Class Diagrams. IEEE Transactions on Software Engineering, Software Engineering, IEEE Transactions on, IIEEE Trans. Software Eng, (4), 412. https://doi-org.proxy-library.ashford.edu/10.1109/TSE.2017.2777830

Filipova, O., & Nikiforova, O. (2019). Definition of the Criteria for Layout of the UML Use Case Diagrams. Applied Computer Systems, (1), 75. https://doi-org.proxy-library.ashford.edu/10.2478/acss-2019-0010

Humphrey, W.S., & Kellner, M. (1989). Software Process Modeling: Principles Of Entity Process Models. (1989). 11th International Conference on Software Engineering, Software Engineering, 1989. 11th International Conference On, 331. https://doi-org.proxylibrary.ashford.edu/10.1109/ICSE.1989.714444

IBM Knowledge Center. (2014). Packages. Retrieved November 23, 2019 from https://www.ibm.com/support/knowledgecenter/SS8PJ7_9.1.1/com.ibm.xtools.modeler.doc/topics/cpkg.html?pos=2

Lane, S., O'Raghallaigh, P., & Sammon, D. (2016). Requirements gathering: the journey. Journal of Decision Systems, 25, 302–312. https://doi-org.proxy-library.ashford.edu/10.1080/12460125.2016.1187390

Lima, L., Tavares, A., & Nogueira, S. C. (2019). A framework for verifying deadlock and nondeterminism in UML activity diagrams based on CSP. Retrieved from http://search.ebscohost.com.proxy-library.ashford.edu/login.aspx?direct=true&db=edsarx&AN=edsarx.1910.13638&site=eds-live&scope=site

Pressman, R. S., & Maxim, B. R. (2015). Software engineering: A practitioner's approach (8th ed.). Retrieved from https://www.vitalsource.com

Spezialetti, M. (2016). Thinking about asking: Encouraging a questioning approach to requirements gathering and problem solving. 2016 IEEE Frontiers in Education Conference (FIE), Frontiers in Education Conference (FIE), 2016 IEEE, 1. https://doi-org.proxy-library.ashford.edu/10.1109/FIE.2016.7757595

Wells, C. (2009). An Introduction to OOAD. www.technologyuk.net. Retrieved November 15, 2019 from http://www.technologyuk.net/computing/software-development/systems-analysis/intro-to-ooad.shtml

Xi, W., Liang, G., Huaikou, M. (2008) An Approach to Transforming UML Model to FSM Model for Automatic Testing. 2008 International Conference on Computer Science and Software Engineering, Computer Science and Software Engineering, 2008 International Conference On, 251. https://doi-org.proxy-library.ashford.edu/10.1109/CSSE.2008.700