CS 410 - Software Engineering - Exam 1

Adam Stammer

(Dated: October 15, 2020)

Abstract

The following document is a summary of specific sections of the third edition of "Object-Oriented Software Engineering Using UML, Patterns, and Java" by Bernd Bruegge and Allen H. Dutoit. It was written to prove understanding of the content material, and to be graded as an exam for CS 410 - Software Engineering (Fall 2020, Dr. Iyengar, Winona State University).

CHAPTER 1.1-1.5

Over time, commercial software systems have become increasingly complex, and expectations have increased too. Projects go far beyond simply writing code, to include the various engineering feats employed in other technical fields. Early on, projects were rarely planned ahead making communication between users and developers more convoluted, project scope poorly defined, and business expectations hard to quantify. All of these problems were made even more difficult to handle with the frequently changing goals of these projects. To lessen these issues and aid in software development, the concept of software engineering was born.

Software Engineering is all about well defined intentional design. Rather than starting to code, and updating that code as new issues arise and project goals shift, clearly design the entire project, taking into account as many potential problems as possible, and leave coding as the final step. As with any engineering project, it's all about solving a problem and/or completing a task. With any such task, learning is key. The more one understands about a problem, the better one is equipped to deal with that problem, and those like it. It's also important to understand why decisions were made, so as to not repeat mistakes, or forget a previous consideration.

Communicating an idea is a skill in and of itself, but communicating the concept of a system can be very challenging. Models allow us to represent systems in a way that is easy to communicate and easier to understand. Solving a problem follows the same general steps: identify, analyze, search/formulate, decide, specify. The specifics of this process may be unique to a given person or setting, but the process is definable, communicable and therefore can be standardized. Learning is a key part of life, and engineering is no different. The more one understands the better equipped they are to handle a problem. Context is very important too. A "good" idea applied to the wrong system, ceases to be a good idea. One must adapt their understanding and responses to the context of both the problem and

the solution.

As I said before, communication is aided by standardization. As such, we see standardization in these designs. Roles of project participants help us understand various responsibilities. Clients bring forth a problem, developers work to create the solution, users apply that solution to the problem in context, etc. Even these concepts must be thought of in the context of the project, like a business creating software for another business. Models should also be standardized, so those using them can understand them better and faster. UML is one such example. Projects are also broken into smaller more manageable pieces. A given activity comprises a multitude of tasks to be accomplished. They are often quantifiable and must be well defined to help in allocation of resources. Even the methodology of writing software within an organization should be standardized so as to be repeatable and more understandable to all involved.

Object-oriented design is one standardization of the design process itself. Requirement elicitation involves well defining what the solution must accomplish. Analysis involves producing models of a solution based on the use cases, actors, and flow of events. System design involves breaking the system into smaller subsystems, and defining strategies of development like which software/hardware tools and platforms to use. Object Design bridges the gap between the analysis model and the defined platforms to be used. Implementation is the actual translation of these models and design into a functional product. Testing is then used to make sure the solution functions as it was intended and to help find things that aren't a part of the model that should be.

Management is also a key part of any group-based project. Communication can be difficult because of the incredible variations in backgrounds, experience, and understandings of all involved, and establishing proper conventions of communication can vastly aid in this. Rationale is justifying decisions and must be handled in a way that doesn't leave everyone confused, or waste time. Decisions are questioned all the time, especially with changing criteria, and reevaluating solutions is much more effective when the rationale behind that decision is defined, available, and recorded. Software Configuration tracks and controls changes in the product. Since change is so common within the requirements, models, and tools used, establishing checkpoints and versioning of software has the potential to save a lot of time and effort in recovering from an unforeseen change. Project Management involves the oversight of the project as a whole. Oversight of activities can help in timely delivery of

a quality product while maintaining within budget. The Software Life Cycle as a whole can be very complex, but I hope it's clear at this point that communication can easily become a bottleneck, and that writing code is only a very tiny aspect of software development.

CHAPTER 2.1-2.3

UML (Unified Modeling Language) is a modeling notation that can be used by all object-oriented projects based on the best elements of precursor notations. It can be split into three different models: functional, object, and dynamic. Functional Models use use case diagrams to describe functionality of the system based on a user's perspective. Object Models use class diagrams to describe the overall structure of the system in terms of object and their various attributes and relationships. Dynamic Models use interaction, state, and activity diagrams to describe internal behavior, often sequentially.

Use Case Diagrams focus on what the system appears to be from a user's view. It's not about what the system is made of, or how it works, but rather how the system appears to behave. A function that can be used by an actor is defined using these use cases. These actors must be defined themselves, at least by name, and can be a person using the system, the environment, another system, etc. The actors are not a part of the boundary of the system, but the use cases themselves help define that boundary.

Class Diagrams help describe the structure of the system and how the objects relate to one another. Classes themselves abstract much of the functions and behavior of an object or set of objects and have various attributes including their relationship to other objects and classes. These relationships are described using connecting lines and numbers. This does not describe the functionality of behavior of those objects, just how they are related.

Interaction Diagrams formalize the behavior of the system and the internal communications. It helps describe the participating objects in a given use case so we better understand all that are involved. Sequential Diagrams are a special type of Interaction Diagram that describe an interaction over time and, as the name implies, the sequence of events involved.

State Machine Diagrams describe the behavior of just an individual object as various states and the transitions between these states. A given object can have any number of states. Unlike the Sequential Diagrams that focus on the interaction of multiple objects, these State Machine Diagrams just focus on the states and transitions of one object at a

time, even if they are based on external events.

Activity Diagrams display a given activity and the flow of events that it includes. It is most similar to a flow chart, in that it is not necessarily sequential but explains the various states of activity.

The design system as a whole can be broken into smaller and smaller pieces made up of objects, events, and relationships. Each subsystem becomes more specific in it's modeling and definitions making each parent subsystem an abstraction of what is beneath it. This can give a better overall picture of what a system is meant to accomplish, while still allowing one to focus on a subsystem when the system as a whole is too complex. In some ways that is the purpose of modeling; abstract what can be abstracted and focus on what must be focused on. Object-orientation follows a similar structure allowing for an individual or team to focus on one part of a larger system without worrying about exactly how the other parts work.

CHAPTER 3.1-3.3

Today, few projects can be done by just an individual. Most are too complex for one person to understand, and it would take too long for one person to finish a project even if they did. As such, communication is one of the hardest, yet most rewarding, aspect of software development. The notations of UML mentioned above are tools in the communication of systems but they still leave many unanswered questions that still need to be communicated. To further aid in this, projects are often broken apart into 4 pieces: Products, Schedules, Participants, and Tasks. The timeline of the project can also be broken apart to aid in communication and organization: Definition Phase, Start Phase, Steady State Phase, and Termination Phase.

The times when communication occur can be classified as scheduled or non-scheduled, and understanding how to use and handle both is important to being a good communicator. Meetings, project reviews, and inspections are some examples of scheduled communication. Questions, issue resolution, and coffee break talk, are examples of non-scheduled communication. Different types of information warrant different types of communication based on the timing and importance of that information.

Organization of a group can be quite difficult, but it is very important to understand

how one fits into the group to be an effective part of it. The interactions within a group can be broken apart into pieces of their own. Reporting is often one sided, in conveying the status of something. Decisions need to be made by those with the right authority, but the result needs to be understood by all involved for that decision to have any meaning. Communication is the exchanging of any other information, and is much more likely to be two sided than the previous two interaction types. Much of these more one-sided interactions follow a more hierarchical structure as information moves up or down a groups authority tier. Some information though must jump various tiers and follows a more liaison based flow of information.

Part of understanding communication of a group, is understanding where one fits into that group. Well defined roles are a part of that. Management roles focus on organization and execution of the project while staying true to the costs, constrains, and goals involved. Development roles rather focus on specifying, designing, and constructing the systems they are concerned with. Cross-functional roles are a bit of a mix of both in that they must organize much of what developers deal with across multiple teams. Consultant Roles focus on temporary support when and where it is needed.

Once one understands their role it's much easier to understand and focus on their own tasks. A task is a specific work assignment for a role. Group multiple related tasks together and it is called an activity. Any tangible item that results from one or more tasks is considered a work product. If these work products are something the client will see, they are considered deliverable work products, otherwise they are considered internal work products. A work package helps describe all that is involved in a given task, like the inputs and outputs.

Scheduling can be one of the most difficult aspects of project planning, but it remains an important expectation. One must estimate the time a given task will take and plan accordingly. Making these schedules available to all involved can help other in intelligently creating their own schedules. Formal structures like Gannt Charts and PERT Charts can make this communication much more fluid and save a lot of time. It's also very important to revisit schedules and update them accordingly. Some timelines are very strict and must be met in time to avoid negative consequences. Others are more fluid and can be changed around as priorities shift.

CHAPTER 4.1-4.3

Requirements are not only how we aim our efforts, but also how we evaluate our own progress and success. They provide the framework of accomplishment within a project on both an individual and group scale. As such, it is important that these requirements be well defined. This is so important that it can be considered a field of engineering on its own.

Those requirements can be further described as functional or nonfunctional. Functional requirements are the interactions between the system and its environment regardless of the specific implementation. These are the requirements that theoretically define the system and won't change if you re-implement the system in another language, or another computer for example.

Nonfunctional requirements, however, are dependent on the methods of implementation. These nonfunctional requirements can even be organized further as they effect usability, reliability, performance, supportability, etc. Some languages and hardware run faster than others; some are easier to modify, update, and support as time goes by; some systems interface far better than others. These things don't change the functional requirements, but they can vastly affect the nonfunctional requirements.

SELF EVALUATION

Looking back at the projects I've been a part of, I'd say that much of what is expected in software engineering I have covered in my own way. I may not have followed strict UML guidelines with my charts, or assigned tasks and roles as definitively as I should have, but I still handled them as I thought fit, but I know I can vastly improve. This last project especially, much of the networking system I chose to do myself, even though it was proportionally more work than what most people were assigned, simply because I understood it all and thought it was easier than trying to communicate that subsystem to my group members. Yet, throughout the whole project we talked as a group about how much smoother it would go once we better understood the formal definitions of various things, including proper usage of UML. Even our roles were more organic than ideal. I largely took over as an organizer of the group and acted as an ad hoc manager, scheduling meetings and assigning tasks, but this was never decided as a group; I just did it because no one else did.

In my own way I have been engineering software, but it's very clear that formal study of such a topic is well worth the investment, and investment into the engineering itself even more so.

One thing that I haven't really seen put well into words yet, in the book or otherwise, is the context of the project and how that affects engineering. Most of the projects I take on that are 'worth' engineering (i.e. group projects, projects that will be shared, projects that take longer than an hour to code), are school projects. These school projects are not met with the time or manpower necessary to produce a marketable product, and some more tedious aspects of the engineering side of things can become arguably invaluable. Every school project is expected to be buggy and incomplete. In essence, these assignments are prototypes at best, by design. This can make the engineering aspect of those projects seem superfluous and unnecessary. Why design redundancies, fail-safes, error recovery, etc. if it won't affect my grade anyway, as long as I understand that in a 'real' project they would be required and I will be expected to know how to handle them. This is very much a symptom of how little time we really spend in school, but it would be nice to formalize the importance of a project itself. Software Engineering is more important in life critical projects like spacecrafts and medical equipment than it is in noncritical projects like a digital clock or a "Board" Game assignment nobody is going to play anyway. Software Engineering is an investment into a project, but the return on that investment will change depending on the goals of the project. I've felt the varying returns on investment, and I'm sure I will continue to do so.

I'd like to give another example so my previous paragraph is not misunderstood. I do believe that software engineering is a worthy investment the vast majority of the time, but the importance of the project in question is going to change how much time I want to spend on that engineering. The other week I threw together a hardware and software project that was essentially a very simplistic thermostat. I barely preemptively engineered the system because it was very quick to build and not important. If I messed up and it didn't work correctly, nobody would be hurt, and I would realize the mistake when this thermostat disagreed with my others. I also didn't need to communicate this idea to anyone else because it was just a personal project. On the other hand, my current job is a hardware and software design project to build a complicated thermostat for a major radio telescope. I've spent a lot of time engineering that system because if I do mess up, my employers could

lose out on thousands of dollars to fix my mistake, and I could lose my job and damage my reputation, not to mention I'm being paid to do it. My system will be interfaced with a multi-million dollar telescope, so I know that I need to spend as much time as I possibly can engineering the system to avoid mistakes and meet all of the requirements I need to. This varying degree of return on investment - that investment being software engineering - is something that most people have an inherent 'feel' for. I hope that before the semester is done, though, we take a more formal look at that, by comparing various real world projects.