Supplier's guide for communication in software projects

Master's Thesis
University of Turku
Department of Future Technologies
Software Engineering
2019
Joni Rämö

UNIVERSITY OF TURKU
Department of Information Technology

JONI RÄMÖ: Supplier's guide for communication in software projects

Master's Thesis, 107 p., 0 app. p.
Software Engineering
June 2019

In the modern age of digitalization an increasing number of companies is stepping into the world of software development. This also means companies of various technical skills levels. To create successful software systems, there needs to be adequate skill but also sufficient trust between the buyer of software development and the supplier. This is why the software development procurement process is very important. New buyers rarely want to take a lot of risk with software projects, which is why a lot of bad practices have stuck in the industry. There should be enough information and trust already in the procurement process, where the first mistakes are made. For this purpose special buyer's guides have emerged to help new software buyers navigate this unknown ground and learn about software development and what creates value. The intent behind this thesis is to act as a so-called "supplier's guide" that teaches software companies how to improve their communication strategies towards a better procurement and development process. The result is an improved communication model consisting of a list of guidelines and directions to their application. This improved model gives advice on where to concentrate communication during the procurement and development process and how to communicate in the best way. Finally, this improved model is validated in the writer's own professional network and adjustments to the model are presented.

Keywords: software, procurement, development, communication, buyer, supplier, guide

Contents

1 Introduction

In the modern age of digitalization as countless new companies are creating their own presence on the web, software applications are no longer the exclusivity of large corporations. Almost every business is expected to have an application or at the very least a website. Because of this large influx of new projects in the field of software, there is a growing need for stronger guidelines for how to communicate about software projects. Many traditional non-technical companies just want an app or a website to beat their competition and have no idea how one goes about making one. This means there is a greater responsibility for the software companies to educate their customers in software development. The proper education of both parties is paramount for the success of each project.

Before a software project can be started, the customer must decide which software company will have the privilege of developing the project. In Finland this is usually done with a tender. A tender is a bidding competition where multiple software companies offer the customer to do the project with a certain team, technology, schedule and costs. The customer will then choose the winner from this group of supplier companies. The problem in the IT industry tends to be that the greatest focus is usually on costs. The company that does the same project for less will often get the job, because this approach has much less perceived risk. However, this is detrimental both to the software product and the entire industry, because low costs do not necessarily predict lower risk and greater results.

Companies that pride themselves for making quality software products have a hard time competing with companies that negotiate their costs unrealistically low. This often leads to estimates that are too optimistic and that have been streamlined so far so that the project cannot realistically be completed within the estimate. This in turn leads to projects that are considered ”failures” because their costs went far above the unrealistic estimates.

There should be a shift in the industry, even for the betterment of software quality in general. Software companies should educate customers to scrutinize the development team’s capabilities in relation to the cost and raise questions if there is a high variance in this metric between companies.

To shift the industry to value skills more than purely costs, there must be an increased understanding of software and especially software projects across all fronts. This thesis serves as a guide to software developers and management in software companies to understand software projects better and integrate a knowledge-sharing approach into their interactions with the customer. The customer organization, or even more importantly the customer representative, must be continuously evaluated and educated, starting from the first meeting. Do they know software? Do they know their own organization? Do they understand what they want? How to help them get what they need?

What this thesis wants to answer is: **How to communicate with the customer during the lifespan of the software project to lay the groundwork for a successful software product?** By ”lifespan of the software product” I mean both the procurement process before the project has been started and also the development process itself. It is a waste of time and resources for both the customer and the software company when a project fails to get initiated after possibly multiple meetings and lots of communication. It is a better choice to stop the negotiations as early as possible if a force majeure is encountered, not to waste any more resources on a project that will not come into fruition. An even worse situation is encountered when a software project begins successfully but is be cancelled later because of disagreements or a lack of funds. The root of these problems is poor communication and the challenges of pricing the software development.

Billing software projects is difficult as the natural instinct for new customers is to use fixed prices to avoid overspending. However, fixing prices often causes also rigidity in the development process as common sense dictates one must have a solid plan how to make the development effort correspond to the fixed price. This of course fights against the idea of iterative development that is common for modern agile software development. Not only this, it usually causes overspending regardless because of poor development practices that come with sequential development. This conflict is caused by the imbalance of risk between the customer and the supplier. It is not always clear for the customer how the development effort is creating value.

The IT industry requires common guidelines to think and communicate about software development and procurement in an iterative way. Software projects should be billed as *software development* not as a *software product*. The problem may exist because iterative development is still so young or because so many new companies of various technical skill levels are now embracing software. Does it simply require time so the billing practices catch up with the iterative development? This question is difficult to answer but the purpose of this thesis is to contribute to solutions for this question.

1.1 Thesis structure

In the year 2019 there are many processes for creating software that are used to increase the efficiency and chance of success of software projects. In the IT industry programmers and other people in technical roles have to constantly be learning about new concepts and tools. This makes it easy for people to become complacent and want to stick to old proven ways. This means unsuitable methods may be used for the complete lifespan of the project. It is very important for software teams to understand their tools to create successful software products. It is also important to state the reasons for a specific technology to the customer. More often than not the customer does not completely understand the logic behind certain tool choices, so this has to be understood well within the team to be communicated to the customer effectively.

This is why Chapter 2 goes through a brief history of software development, starting from the 1970s with sequential development and eventually ending up in the 1990s with the advent of iterative development methods. In Chapter 3 we go through two of the most popular modern software development processes to understand what software development is like today. I also highlight certain aspects of the process models that need special consideration in customer communication.

When the basic concepts of software development are clear, it is easier to talk about the procurement process. Software is an organic and a challenging product to create, so it must be created incrementally. Once the project requirements have been determined, the approach can be decided based on the budget and schedule of the customer. Does the customer have only a small budget or is there virtually no cap at all? Does the project have to be created in a short couple of months or is the focus more on quality? There are many factors to consider when negotiating the terms of the procurement. The most important piece of paper in the process is the proposal that the software company presents to the customer. This is the first concrete outline of the project: members of the development team, the schedule, the price and the initial software requirements. This is the ”golden ticket” that the customer representative takes to the executive group. So it must be well drafted to succeed. The theory for the procurement process, contracts and the participating actors is outlined in Chapter 4.

Software buyer’s guides are booklets created for the purpose of introducing newcomers to the world of software development. As the pricing of software development is not intuitive for new buyers, these guides provide basic concepts of modern software development, how to buy software safely and how to think about software in terms of value. In Chapter 5 I analyze a software buyer’s guide made by a well-known Finnish software company to demonstrate the value these guides bring and if there are any places for improvement.

Based on the previous chapters in Chapter 6 I draft a model for customer communication especially in the beginning stages of a software procurement process. This model consists of gathered communication guidelines and directions to their application. I also validate my conclusions by presenting the guidelines gathered in Chapter 3 and 5 to my professional network to see their preference on the most important ones.

Throughout the body of work there will be one main theme: What pieces of information are important for communication for a successful software project? Software process models are the heart of software development and even when they are essential processes for software developers, certain aspects are also really important to the customer. As there is a lot of theory to cover, the key principles and practices are highlighted and their usefulness is constantly compared to customer communication.

2 Evolution of software

Before figuring out how to communicate software concepts to the customer, it should be clear how software is created. Naturally it varies from project to project but usually it is a big undertaking that requires many different kinds of expertise: user interface development, data layer integration, database management, secured connections... the list goes on. It is simply too much for a single person to grasp, so a software team is required where everyone has a certain expertise. Everyone should be on the same page how the software is roughly supposed to work, but the nuances of each technology is up to the corresponding expert.

To understand the architectural and technological decisions in a software project, it is important to understand the life cycle of software development. What kind of processes are used and why? Customers buying software projects do not necessarily need to understand how to create a user interface with the latest front-end technologies. However, it is important for them to understand how the processes of software development work so they can understand the way software must be billed, how the procurement and development processes should ideally work and what kind of problems the project may encounter.

We should look into the past to see why software is the way it is today and how certain aspects of past have shaped it. It is not as important to focus on the technologies themselves but rather the overlying processes that govern the process of software development. These are called *software process models*. They create structure around the different facets of engineering a software product. They increase clarity to the development process and provide vital metrics of progression. A software process model determines the order of the phases involved in software development and establishes the transition criteria for moving from one phase to another. Thus it plays an important role in the life cycle of a software product. A software process model answers two questions: What will we do next and how long will we keep doing it? It is different from a *software methodology* which asks a pair of more technical questions: How to navigate through each phase and how to represent the products of each phase? [1]

Programming was very different before the 1970s and the dawn of personal computers. The first software process models were born in the 1960s but they were still very immature as software development was still taking its baby steps. There have been many software process models in the history of software development but we will only give a greater consideration for two of them: the waterfall model and the spiral model.

2.1 Waterfall model

When software development was new, it was not clear how software should be developed efficiently. The very first ”software process model” was the *Code-and-fix model.* Although it can be hardly called a model as there are only two phases: produce program code and fix problems in the code when necessary. It was soon clear that a more robust structure was required around the development.

So there had to be a design to base the development on predefined rules instead of the whims of a developer. To create this design, there had to be a set of requirements to determine what is the wanted outcome. So naturally to start things off people used previously acquired knowledge from manufacturing and construction industries. In these industries, changes in design meant much higher costs sooner in the development process. The design would become harder and eventually impossible to change the longer the project went on.

This is not the case in software industry. The design and implementation can be changed quite quickly on the fly because the end product costs only brain power to produce. But because the only related processes that were known at the time were manufacturing and construction, there was virtually no other way of thinking about it. This meant the development process for software started off very rigid. This is where the *waterfall model* was born. [1, 2, 3]

Software development naturally gravitated towards a model like the waterfall model because no better alternative yet existed. It resembled the process models from manufacturing and construction as the waterfall model expects no change in requirements or specifications after they are once defined. The first formal description of the waterfall model was by Dr. Winston Royce in 1970 in his article *Managing the development of large software systems*. Although Royce did not coin the term ”waterfall” in his paper, he described the model as an example of a flawed, non-working, software development model. In the model Royce presented in his article there are 7 different phases of the waterfall development model: system requirements, software requirements, analysis, program design, coding, testing and operations (as seen in Figure 2.1). Whilst the amount of these phases vary based on the way the concepts are grouped, the contents are essentially the same.

As we go through the concepts of the waterfall model, the primary source I will use is the previously mentioned article by Winston Royce [4]. I will also use parts of another article, *The waterfall model in large-scale development* from 2009 by Kaj Petersen et al. [5] that describes a more modern approach to the familiar waterfall model. It is important to keep in mind that while the model itself is antiquated by modern standards, the phases it contains are still very much a part of modern software development.

According to Royce the essential steps to develop any program is *analysis* and *coding*. This means analyzing the problem or wanted outcome and actually implementing it through program code. This is enough when the required effort is sufficiently small and

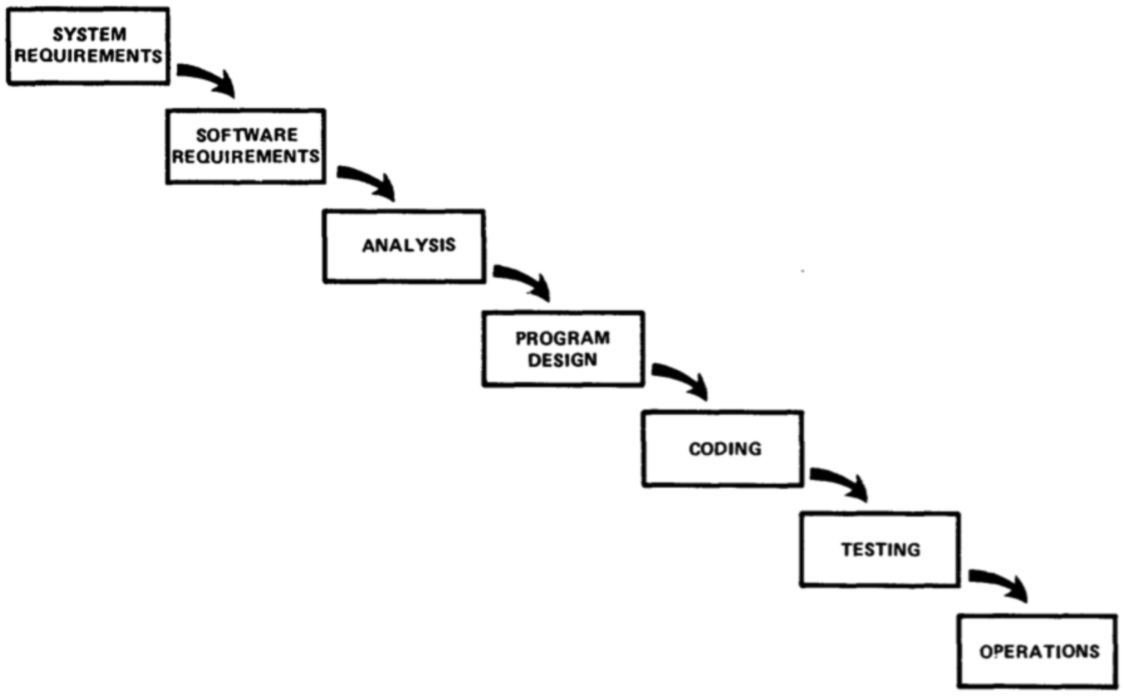


Figure 2.1: Waterfall model as described by Royce (1970) [4]

the product is to be operated by those who built it. Even though customers are happy to pay only for analysis and coding because they directly contribute to the usefulness of the product, for bigger systems other steps are needed to manage the complexity of the system and to provide accurate results.

In this model the analysis step is preceded by *system requirements* and *software requirements*. These can be unified as a single phase called ”requirements” as it is in many descriptions of this model. During these phases the requirements of the software product are defined. These consist of the needs of the customer for the product to fulfill their business goals and the design of the software itself.

In their description of a modern approach to waterfall, Petersen et al. described a so-called Quality Door between each major phase (as visualized in Figure 2.2). This is essentially a checklist to verify that it is safe to proceed to the next phase. The Quality Door for the requirements phases could be to verify that the requirements were understood correctly, agreed upon and documented. It is also important that relevant stakeholders are identified and they are satisfied that the solution supports their business strategy.

The next phase after the requirements is the analysis. However, in more streamlined descriptions of waterfall model the analysis phase is included in the *design* phase. The phase is called ”program design” in Royce’s article most likely because in 1970 computer systems were very monolithic machines meant to crunch numbers instead of having fancy front ends. [6] In other words, as the problem or wanted outcome is analyzed and defined, the design of the software naturally follows. This design will detail the steps to reach the wanted outcome of the project. In actuality this means documenting the architecture of the software system and the design of the user interface. The architecture details the elements of the software, how the elements are connected to one another and how they communicate with each other. The design of the software details the layout of the user interface and how certain views work with other views.

After the design is complete, the implementation i.e. *coding* follows. The development team follows the architecture and design that was created in the previous phase. They usually conduct basic unit testing before moving onto the actual testing phase. Before moving to the next phase the architecture should be evaluated and whether there is deviation from the previously defined requirements, timeline, effort or product scope. This evaluation can be thought of as a Quality Door.

In the *testing* phase the system’s functionality and quality is tested. Also metrics of its performance are collected to determine whether the system can be deployed. In the scope of this thesis we only examine software products so the testing is directed solely on the software components. However, if the company provided complete solutions that also includes hardware, the testing would have to be even more extensive, targeted for a variety of hardware and software configurations. As the Quality Door of the testing phase the outcome of the phase should be verified by comparing it to the system and the outcome of the previous phase for deviations in quality and time. It should also be verified that the project handout is defined according to company guidelines and that the outcome

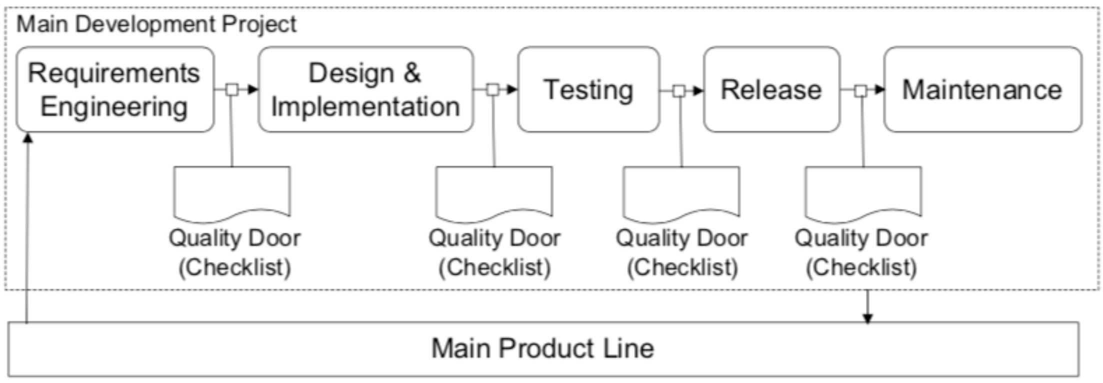


Figure 2.2: A modern take on waterfall as described by Petersen et al. (2009) [5]

of the project meets the customer's requirements.

The next phase is called *operations* in Royce's article because back in 1970 the software went straight to use after the testing phase, usually by the very team that built it. In modern systems the system must usually be deployed to a server for it to be used by the customers. In the *deployment* or *release* phase the software is brought to a state that it is ready to be shipped for usage. The documentation for installation instructions and user guides are finalized and the product is evaluated together with the customer. The important aspects of Quality Door evaluation are whether the product meets the customer's requirements, whether customer has accepted the outcome of the project and whether the final project was presented in time and fulfilled its quality requirements. These aspects determine the success of the project. The release phase is also a good time to execute a post-mortem analysis of the project.

A more modern approach to waterfall can be seen in Figure 2.2. It includes the Quality Doors between each phase and the phases have been streamlined to encompass all requirements under one phase and also combining design and implementation phase. From here we can also see that after the release phase, there is one last phase called *maintenance*. In modern software systems this phase is very common. This phase is usually included in the project so the customer can get help with fixing bugs that may appear or add additional features when needs for such arise. [5]

2.1.1 Pitfalls of waterfall

There seems to be a consensus in the IT industry about the pitfalls of the waterfall software development model. It does not cope well with change. It requires approval of many documents so changing these documents requires a lot of effort and creates costly rework. It also leads to unpredictable software quality because the testing phase is after the implementation. Problems are also too easily pushed to later phases. [5]

For an example, see Figure 2.3. When going through the waterfall process, there is an iteration between each step's preceding step and following step but rarely with the more remote steps. The good part about this is that the possible changes are kept manageable. But the bad part is encountered when something further back the process has to be managed. Consider a critical bug is found during the testing phase. This bug unearths a need for a major design change in the software. This change could be so disruptive that the very requirements the software is based on must be changed as well. Suddenly the development process is at the starting point again and the schedule and costs go up. [4]

There have been certain reasons identified with failures regarding the waterfall model. The large scope of the project is hard to manage all at once. This leads to requirements not being managed well and at the end of the project customer's current needs are not addressed. This leads to a situation where many of the implemented features are not being used because there has been a change in needs during the process or because there has not been an opportunity to clarify misunderstandings. [5]

When this model was common back in the day, the system requirements were specified by the programmers themselves without much input from stakeholders. In this environment a waterfall-based approach works because users cannot give feedback that could alter the system's requirements. Today, however, software systems are closer to the users than ever and if the users' feedback is not heeded they will not use it. Waterfall model

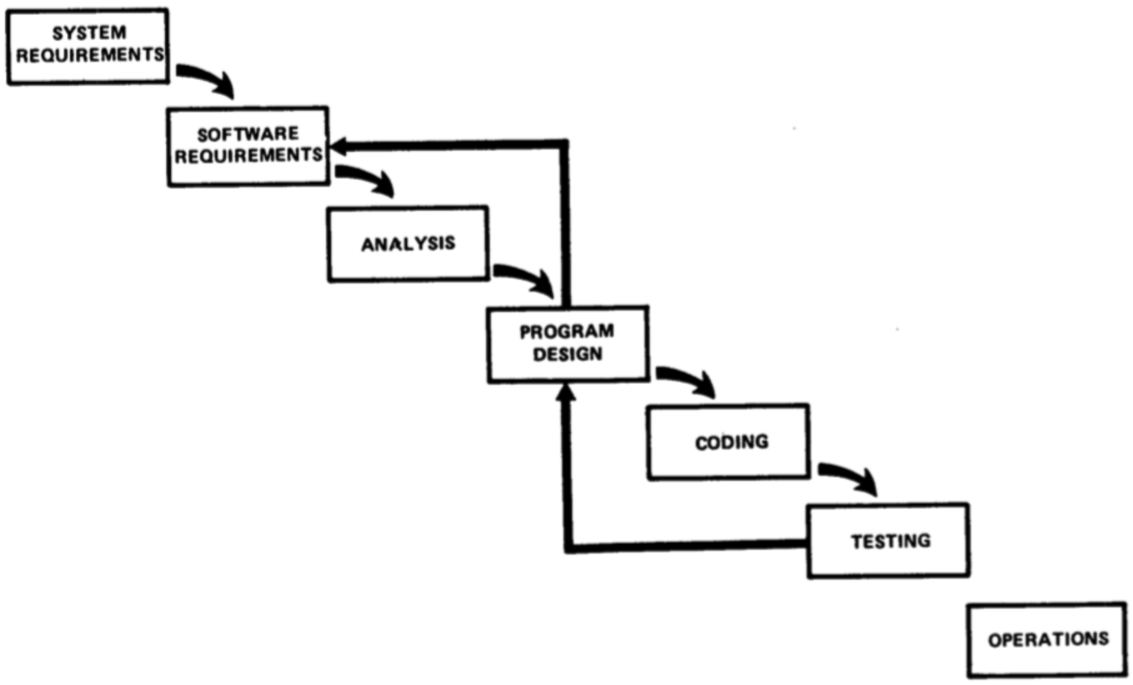


Figure 2.3: A problem with waterfall as described by Royce (1970) [4]

expects that requirements are set, stable and fully mature before the analysis. Phases are progressed linearly and previously visited phases are revisited only if the content created in that phase fails review or testing. [6] In this environment the waterfall model actually works well because it is very predictable as it follows closely the predetermined plan. This is also why it makes the architecture well-structured. [5]

In modern software development we know that change is unavoidable and it must be taken into account in the life cycle. It is not an erroneous state but a natural aspect of software development. It is easy to conceptualize with requirements: as our understanding about a concept grows, we begin to realize our initial misconceptions. It is incredibly hard if not impossible to understand something complex perfectly the first time around. If we fail to adapt our solutions to change, the cost to implement our solutions around the design flaws escalate exponentially. [6]

There have been attempts to improve the waterfall model of its shortcomings. An early model presented prototyping as a feedback mechanism so misconceptions in the design could be caught early. Other process models have attempted to mitigate risks by segmenting the process into smaller waterfalls that deliver the system incrementally. In the 1980s many replacements for the clearly flawed waterfall model started to emerge. [6]

2.2 Spiral model

The *spiral model* was introduced in 1988 by Barry Boehm in his article *A Spiral Model of Software Development and Enhancement*. It was presented as an alternative to ”traditional software process models”, especially including the waterfall model, which was already deemed as clearly ineffective and costly. The spiral model process is *risk-driven* rather than process-driven or code-driven as the waterfall model is. This risk-driven approach means that the quality of the software product is centerfold instead of merely the completion of the product. [1]

A visual description of the spiral model can be seen in Figure 2.4. The radial dimension represents the cumulative cost of accomplishing the phases in relation to time, whereas the angular dimension represents progress made. The model is traversed by following the cycles of the spiral and during each one, the following things are identified: goals of this portion of the product (performance, functionality, etc.), the alternative means of implementing this portion (plan A, plan B, buying it from outside sources, etc.) and the constraints that the alternatives impose on the application (cost, schedule, interface, etc.).

In the next phase the alternatives implementation methods are evaluated by comparing them to identified goals and constraints. This is the *risk analysis* phase. This phase will often identify areas of uncertainty in the project that are sources of possible project risk. To resolve these sources of risk, a cost-effective strategy should be formulated. This can be done by prototyping, simulation, user questionnaires, analytic modeling or with other risk-resolution techniques.

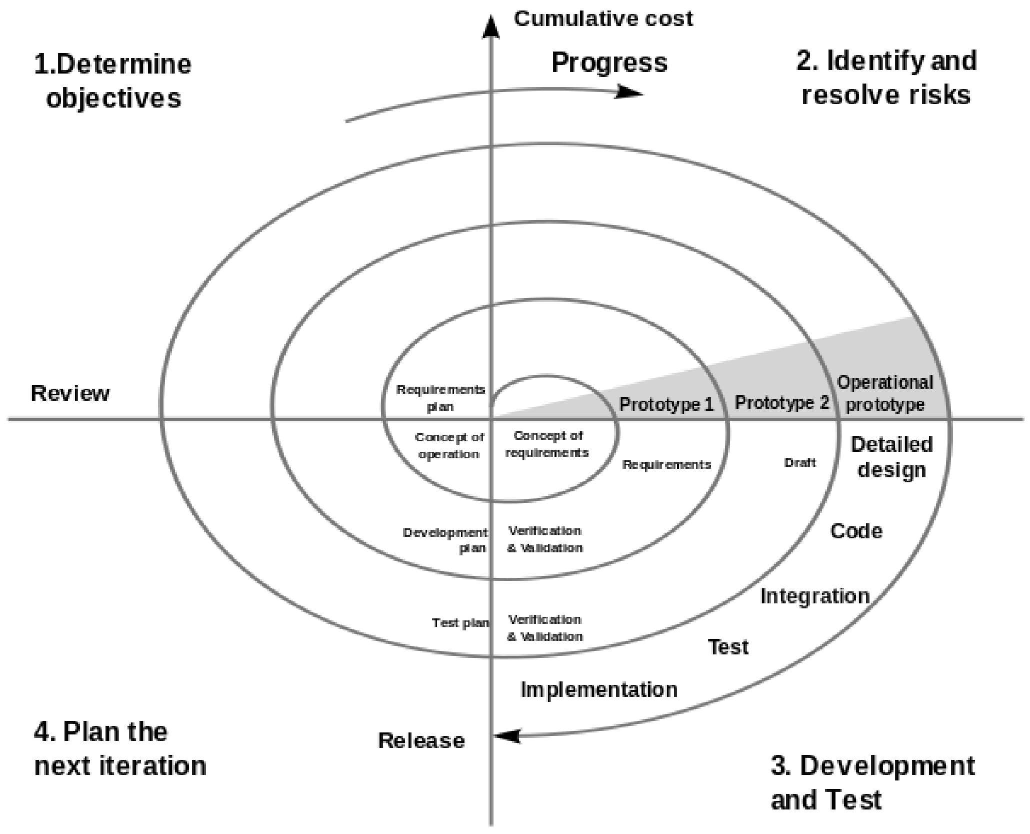


Figure 2.4: Spiral model as described by Boehm (1988) [1]

The content of the next phase depends on what kind of risks were identified. For example, if the development faces risks from the user interface, the follow-up would be to create a minimum-effort specification of the product to plan for the next level of prototyping. This prototype would then be implemented to resolve the biggest risks. If this prototype is functional and robust enough to be a version of the working product, the next risk-driven phases would be to create further prototypes to smooth out the remaining risks. In this instance, phases concerning with writing specifications would not be exercised. As we can see, this means that based on the risks found, only part of the phases in the spiral may be traversed. Once the risks have been resolved, the basic process follows the same phases as the previously addressed waterfall model, modified appropriately to accommodate incremental development.

The main concept of the spiral model is its risk-based nature. Risks must be analyzed after each spiral iteration and possible risks must be dealt with accordingly. Another important feature is that after each cycle a review of progress is held with all the primary stakeholders of the software product. At the end of the review, the next cycle is planned, so that everyone is mutually committed to the project.

Unlike the waterfall model, the spiral model does not describe the entire process of software product development, but rather the phases one may traverse during development and the order they should be completed in. The spiral model is iterated as development of the product continues. Figure 2.4 does not explicitly define the beginning and the end of the project. Thus the model is more of an implicit description rather than explicit set of phases.

In his article, Boehm acknowledges some risks that already have been identified in the spiral model upon its conception. Firstly, the model works well in internal software projects but public contract software projects can be troublesome because many of the software's needs may rise from risk analyses, which is hard to justify for customers. It is more difficult to define contracts to customers without specifying all of the features to be delivered in advance. Secondly, there is a heavy reliance on risk-assessment expertise which should not be taken as self-evident. Not everyone is capable of assessing risk accurately. Thirdly, as stated previously, the figure is very implicit which may require further elaboration to avoid false interpretation. [1]

As anyone familiar with agile practices can probably see, the spiral model is an obvious stepping stone towards agile software development, as it incorporates an iterative nature to software development that is so characteristic of agile. It also raises a greater emphasis to communication transparency between stakeholders with regular reviews of development progress.

2.3 Agile software development

Although agile methods became more popular in the beginning of the 2000s due to the release of Agile Manifesto, its roots go back decades. The heart of agile is *iterative and incremental development* (IID) which has many documented examples from the 1970s and the 1980s. Before the 1970s there were almost no official articles detailing IID and the earliest one promoting its benefits was from 1968 in Briant Randell's and F.W. Zucher's report at the IBM T.J. Watson Research Center. [7]

In 1970 as Winston Royce gave the non-working single-sequence waterfall model its first formal description, he actually proposed something of an iterative approach as the alternative: to do the waterfall twice. During the 1970s, there began to emerge more studies and cases of successful IID applications. It is suspected that the official description of waterfall Royce provided may have made it easier to conceptualize alternative approaches, including IID.

The 1980s continued to produce increasing numbers of articles about various IID models, despite waterfall still being the de facto software process model. A landmark IID publication was published in 1986 when Barry Boehm released "A Spiral Model of

Software Development and Enhancement" which we covered in the previous chapter. [7]

In the 1990s public awareness of IID in software development was increasing significantly with hundreds of books and papers promoting IID. One of the most important steps towards agile was a method Jeff Sutherland and Ken Schwaber had started developing in the early 1990s, which would later be described in 1999 as the Scrum method. This method was based on a Japanese IID approach to non-software products at Honda, Canon and Fujitsu in the 1980s. The method in question was described already in 1986 in an article called "The New New Product Development Game". As the word about IID methods were spreading during the 1990s, there were multiple examples of projects that were started with a waterfall approach, but because of an impending failure, had to adapt to an IID method and against previous odds actually succeeded. [7, 8]

The release of the Scrum development framework really brought IID to the forefront of software and eventually led to the release of Agile Manifesto in 2001.

2.3.1 Agile Manifesto

In February of 2001, a group of experts representing multiple now-agile methods (such as Scrum and XP) gathered and founded the Agile Alliance. From this came the "agile methods", all of which apply IID, and the Agile Manifesto that declares the core message of agile software development. The Agile Manifesto states the following:

- "Individuals and interactions over processes and tools."
- "Working software over comprehensive documentation."
- "Customer collaboration over contract negotiation."
- "Responding to change over following a plan."

It is easy to see how directly the manifesto addresses all of the faults that have generally been linked to the Waterfall method. Overly rigid processes that do not give leeway for changes or evolution. Comprehensive documentation that further solidify the requirements and design of the software which make changes hard to implement. Hard terms in contracts expect many of the features of the software to be determined ahead of time, which is nearly impossible to do as needs often change as the software is developed. Software is an organic entity so sticking to a plan when it does not make sense and when it hurts the product is a bad idea.

However, it is also easy to see why the waterfall model prevailed before agile. Software is a young field, and the document-driven approach of waterfall makes it easier to understand especially for non-technical people. The core idea is very simple to explain and remember: ”create the requirements, then design the software and implement the design”. The cyclic and iterative nature of IID methods make the whole process harder to conceptualize as it is much more organic. Waterfall gives off the illusion of order, accountability and measurability because the milestones are document-driven (e.g. ”the design is complete). From history, it can be seen that IID methods have always been preferred in software development by thought leaders of each time, but wide adoption has taken time. People are not quick to embrace new or complex ideas. This is why, even today, waterfall methods are preferred by certain organizations. [9, 7]

3 Modern software processes

Agile concepts have shaped very profoundly how software is crafted today. Iterative development became the de facto development method, as Scrum grew in popularity after the Agile Manifesto. Scrum is held as the cornerstone of agile development and almost all software companies use it or have used it at one point. As it is such a crucial part of agile, half of this chapter is devoted to learning the theory of Scrum in a compact form.

Lean software development is also a big player in modern software processes and it is adopted into the project life cycle more and more often. Lean software development is the application of lean manufacturing principles in the field of software. The idea of lean is to eliminate all waste in the development process so higher quality software can be delivered more rapidly. It also believes in self-governing teams and iterative development like agile, but the focus is more towards optimizing the whole process. I will also cover a couple lean concepts such as kanban and the Theory of Constraints.

In this chapter, we will cover the main concepts of both agile and lean and see how these theories work with customer communication. I have added a brief section at the end of each chapter to highlight what challenges may be in the customer interface and possible solutions. These guidelines are then gathered into lists at the end of both chapters. It is not only important to know the principles of both processes for their usefulness in software development, but also for communicating with the customer effectively. Especially in the beginning when initiating contact and beginning negotiations, the customer's level of understanding in these concepts should be gauged carefully.

I cover the theories in this chapter from the perspective of software consultancy companies where software team composition may vary from project to project. Software development is slightly different in so-called ”product companies” that develop a single software product, as the teams usually stay the same for much longer. Because the team composition varies, care must be given for assembling an effective team for the job every time.

3.1 Scrum

Scrum is a framework for creating and maintaining complex products, including software products. As it is a framework, various different processes and techniques can be used by following Scrum and Scrum does not explicitly dictate what kind of methods should be used. The Scrum is agile at its core so the processes it encourages to implement are iterative and incremental to maximize opportunities for feedback.

In this chapter I will use the Scrum Guides website as the primary source of Scrum Theory. [10] The content of the website is written by the official co-creators of Scrum, Ken Schwaber and Jeff Sutherland. Scrum can be considered mature at this point of its evolution, so a single veritable source should suffice.

There are three main concepts of Scrum Theory that are vital to understand: transparency, inspection and adaptation. *Transparency* means the product development process should be visible to all who work towards it. In other words the team developing a product must speak about concepts with commonly understood language and make sure no aspects of the product development are hidden from others in the team. *Inspection* means Scrum users must inspect Scrum Artifacts, the tools Scrum provides, and progress towards a commonly defined goal to detect any variances from the goal. This inspection should not get in the way of actual work. *Adaptation* means that the process or the resources must adapt to counteract possible changes that may affect the product detrimentally. This adaptation should be a part of the development process from the start to allow unhindered progression towards the goal.

3.1.1 Scrum Team

The team implementing Scrum is called a Scrum Team. Scrum Teams are *self-organizing* which means they independently decide how to complete their tasks in the best way, rather than being directed by external authority. Scrum Teams are also *cross-functional* which means they do not need to depend on anyone outside the team to complete their tasks because they have all the required skills. For ease of reading, I will use the pronoun ”she” when referring to the third pronoun. I will continue this trend throughout the thesis.

The Scrum Team is comprised of the Product Owner, the Development Team and the Scrum Master. The *Product Owner* is responsible for managing the Product Backlog, which lists all the known and understood requirements for a completed product. Her responsibility is to express the Backlog items in a clear manner, prioritize the items in the Backlog and ensure the Backlog is visible and understandable. She also needs to optimize the Development Team’s work by ensuring they sufficiently understand the items in the Backlog. The Product Owner is always just one person. The Product Owner cannot be successful if she does not have the support of the surrounding organization.

The *Development Team* is a group of professionals who work towards improving the software either by adding new functionality or maintaining and fixing the already implemented functionality. The Development Team is self-organizing which means no one tells it how to turn the Backlog into Increments of potentially releasable functionality, or Increments of ”Done”. The Development Team is cross-functional which means they have the skills necessary to create the product Increment. There are no titles or sub-teams within the Development Teams. Optimal size for Development Teams is from three to nine members.

The *Scrum Master* is responsible for helping the rest of the Scrum Team understand the concepts of Scrum and uphold the Scrum values. The Scrum Master is merely a facilitating leader for the Scrum Team, her job is not to manage daily work. Besides helping the Scrum Team internally, the Scrum Master also communicates to those outside the Scrum Team how to interact with the Scrum Team in a way that the Scrum Team's potential for creating value is maximized. The Scrum Master's responsibility is to facilitate Scrum Events whenever requested or otherwise necessary.

The Scrum Master works with the Product Owner by making sure the goal or goals, the scope and the product's domain are understood by everyone on the Scrum Team. She tries to find techniques for effective Backlog management and helps the Scrum Team craft clear and concise Backlog items. Her responsibility is also to ensure the Product Owner knows how to prioritize the Backlog to maximize the value produced.

The Scrum Master helps the Development Team by coaching them in self-organization and cross-functionality. Her job is to keep the value of the work in everyone's mind. She also does her best to remove obstacles from the Development Team's progress and keep the Scrum values clear even when surrounded by doubt from the organization.

The Scrum Master coaches the organization, its employees and stakeholders, to understand and adopt Scrum when necessary. Together with other Scrum Masters she does her best to positively affect the organization towards a wider adoption of Scrum principles.

Customer perspective: When developing a new software product, if it is decided that Scrum will be the chosen approach, a suitable Product Owner must be determined. The Product Owner can be from either the software company or from the customer organization. Preferably the Product Owner should be chosen from the customer organization, because then the Scrum Team will have first-hand knowledge of the product's domain. When the Product Owner is chosen from the software organization, there is always a chance of misinterpretation or miscommunication when the Product Owner communicates between the customer organization and the Scrum Team.

If the new Product Owner is from the customer's side, her skill level in Scrum should be carefully verified. If the new Product Owner is familiar with Scrum through experience there is likely to be little in the way of problems, as long as the rest of the Scrum Team also knows Scrum. However, if she is new to the concept or she has yet to really use her knowledge on the subject in practice, a careful approach should be used. The Product Owner should understand everybody's role in the Scrum Team, what her role is and what is expected of her. It is especially important that the Product Owner is a singular person and that she alone is responsible for communicating the wishes of the customer organization. Otherwise there exists a risk of miscommunication or a conflict of priorities.

The less experienced the Product Owner is, the more responsibility there is for the rest of the Scrum Team to keep the Product Owner up to speed. This may be cumbersome in the beginning of the project. The Product Owner may need to explain some of the Scrum concepts to the customer organization to make them understand what to expect. She will definitely need to explain the status of the product at certain intervals. This is why it is crucial that the rest of the Scrum Team understand Scrum, so they can teach the Product Owner when necessary.

If the Product Owner is from the software company, an experienced Scrum Master should be chosen especially when the customer is not very apt in Scrum or software development in general. In a situation like this there is a need for leadership as vagueness will undoubtedly lead to chaos and bad practices.

It is an ideal situation where the whole Scrum Team is well versed in Scrum but this is very rarely the case in public contract projects or with software consultancy companies in general. The team will be made up of members of various skill levels and possibly even from multiple companies. In internal projects this could be achieved more easily as they tend to last longer with the same team composition. So because the Scrum Team is very likely to be partially inexperienced in Scrum, the emphasis should be in finding at least a Scrum Master that is experienced in Scrum Theory and who can communicate it effectively to both the Product Owner and to the Development Team.

1. **A suitable Product Owner should be chosen, preferably from the customer organization**
2. **Product Owner should understand everybody's role in the Scrum Team and what is expected of her**
3. **Emphasis should be on electing an experienced Scrum Master**

3.1.2 Scrum Events

Scrum has various events that create regularity to the process and minimizes the need for non-Scrum-related meetings. At the heart of Scrum is the Sprint. The *Sprint* is a time-window of one month or less during which a Increment of ”Done” is created. Usually a Sprint is two weeks long. The duration of the Sprint cannot be changed once it has already started. Once the length is decided, it should also remain consistent throughout the development of the product. A new Sprint starts right after the previous Sprint ends. A Sprint contains the Sprint Planning, Daily Scrums, the development work, the Sprint Review and the Sprint Retrospective.

During the Sprint no changes are made that could endanger the Sprint Goal and the goals set for quality cannot decrease. However, if necessary, the project scope may be clarified or renegotiated between the Product Owner and Development Team. Sprints are limited to one calendar month because longer time periods may introduce more complexity and risk to the development effort.

The Product Owner may cancel the Sprint before the Sprint is over if she deems it necessary. A Sprint should only be cancelled if it makes no sense given any internal or external circumstances. Usually cancelling the Sprint makes no sense. Upon cancellation, any completed and ”Done” items in the Product Backlog are reviewed and incomplete items are returned to the Backlog. Sprint cancellations consume resources because a new Sprint Planning has to be organized to start another Sprint which is why they are usually the last resort.

Sprint Planning is an event at the start of every Sprint to plan the development work to be done and to define the Sprint Goal. The Scrum Master makes sure the event is held and that everyone understands its purpose. The Sprint Planning has three concerns, of which the first one is: What can be done during this Sprint? The Product Owner elaborates which Product Backlog items would achieve the Sprint Goal when completed. The Development Team works together to estimate the cost of these tasks and is solely responsible for choosing which tasks will be chosen for the upcoming Sprint from the prioritized Product Backlog.

The second concern in Sprint Planning is the plans for implementation. When the Sprint Goal has been defined and the items have been selected from the Product Backlog to the following Sprint, it is the responsibility of the Development Team to decide its methods of implementation. The tasks selected from the Product Backlog for the Sprint and the plan to implement them is called the Sprint Backlog. Enough work is planned by the Development Team during Sprint Planning that they know how to estimate how much it can do during the Sprint. If this estimation is difficult, The Product Owner can clarify items on the Product Backlog and make compromises when necessary. Also people outside the Scrum Team can be brought in for technical or domain advice. At the end of the Sprint Planning the Development Team should be able to explain to the Product Owner how they are going to work together to accomplish the Sprint Goal. This *Sprint Goal* is also the third concern of Sprint Planning. It is an important focal point for the Development Team's attention.

Daily Scrum is a 15-minute event for the Development Team that is held on every day of the Sprint. During the Daily Scrum the work for the next 24 hours is planned. During the Daily Scrum everybody answers three questions:

- ”What did I do yesterday towards the Sprint Goal?”
- ”What will I do today towards the Sprint Goal?”
- ”Have there been any obstacles?”

During the Daily Scrum only these questions are asked. The Development Team can meet right after the Daily Scrum to discuss issues in a more detailed manner, if it is necessary for progressing towards the Sprint Goal.

Sprint Review is held at the end of the Sprint to inspect the Increment of ”Done” completed during the Sprint and adapt the current approach if necessary. This is an informal meeting, and the main purpose of it is feedback and collaboration. It includes the Scrum Team and key stakeholders that the Product Owner deems necessary. Everything is tallied that has been ”Done” and not ”Done” during the Sprint. The Development Team goes through what went well during the Sprint and what obstacles were encountered and how they were solved. The current state of the Product Backlog is reviewed and next actions are estimated by the entire group so that the following Sprint Planning will go smoother. The result of the Sprint Review is a revised Product Backlog.

Sprint Retrospective is held after the Sprint Review and before the next Sprint Planning. Its purpose is to inspect how the last Sprint went from a bigger perspective, usually concerning relationships between Team members, the process and tools. During the Retrospective, the objective is to find the biggest things that went well and things that still need improving. The positive aspects should be acknowledged and if something needs improving, the implementation for these improvements is also planned during this meeting. By the end of the Sprint Retrospective the Scrum Team should have identified improvements for the next Sprint to make working together more efficient and enjoyable.

Customer perspective: The Product Owner should understand what her responsibility is during any Scrum Event she takes part in and what she can and should do. It is the Scrum Team’s responsibility to hold their Product Owner accountable for this. It is easy to stop having Scrum Reviews or Scrum Retrospectives if the Product Owner is too busy with other tasks. This can often happen if the customer's Product Owner has many other responsibilities besides the current project. In cases like this it is easy to slip into using an unorthodox approach to Scrum which may introduce more chaos and miscommunication into the process. Therefore, it is important to make sure that the Product Owner is available to do her job properly. Otherwise, another person should be chosen to be the Product Owner. [11]

Most software projects that claim to use Scrum do not actually implement the full spectrum of Scrum processes (such as all Events), and therefore do not use ”pure Scrum”. Instead the approach is combined and applied with other agile practices such as Extreme Programming and kanban. [11] Or even when a choice of Scrum has been made in the beginning of a project, it evolves (or in some cases, deteriorates) into a different approach altogether.

It is okay to use an applied approach, or a completely different approach altogether, as long as everybody on the Scrum Team agrees to it and understands how to execute within the custom approach. If an applied approach is chosen, it should be kept in mind that these processes have been developed for the sole purpose of maximizing the efficiency and value produced by teams. Choosing to execute an applied approach should be done with caution. A better idea would be to change completely to another type of software process model with another set of rules that may be more suitable to the project. However, due to the nature of software projects, iterative development that enables high feedback loops should not be compromised.

It is advisable that the Scrum Master takes a bigger role during the first few Sprints to supervise the performance of the Scrum Team and ease the team's journey to implement Scrum practices. When the team begins to understand the way Scrum works, the role of the Scrum Master becomes more of a facilitator rather than a leader.

4. Scrum Team should make sure the Product Owner can do her job properly

5. **If the Product Owner cannot do her job properly, a new Product Owner should be elected**

6. **Rules of the development process should be understood and agreed upon by all from the beginning of the project**

3.1.3 Scrum Artifacts

Product Backlog is a list where all the requirements of the product are collected and prioritized. The Product Owner is solely responsible for the Product Backlog. The Product Backlog is never complete: In the beginning it contains the initial requirements determined during first meetings, essentially an outline of the project, and it evolves as the product and its environment evolves. It contains all the features, functions, requirements, improvements and fixes that are to be included in future releases; all actions that will shape the product towards a desired outcome. It is essentially a to-do list for the product. Usually the higher priority a Product Backlog item has, the more detailed and well-understood it is. It is easier to make better estimates the more detailed the items are.

Sprint Backlog is the set of Product Backlog items that are selected for the Sprint together with the plan to implement them. The outcome of successfully finishing the Sprint Backlog is an Increment of "Done" and a realized Sprint Goal. The purpose of the Sprint Backlog is to make all the work the Development Team needs to complete transparent. If new work is required, the Development Team adds it to the Sprint Backlog. In this situation, the estimate of remaining work needs to be updated as well. The estimate of remaining work is also updated every time any work is performed or completed. Anything in the plan that is deemed unnecessary should be removed. Only the Development Team can change its Sprint Backlog during a Sprint.

When a Sprint is completed, the completed items are gathered and marked ”Done” while the incomplete items are re-estimated and re-prioritized and returned on the Product Backlog. An *Increment of "Done"* is these completed items summed with all the previous completed items. In other words it is the current progress of the project towards its completion. An Increment of ”Done” should be usable to be accepted and it should fit into the Scrum Team’s definition of ”Done”.

Customer perspective: All these Artifact definitions should be understood by the Product Owner, and she should also understand how they work together. The Product Backlog is her responsibility but the Sprint Backlog is not. She should understand that the detailing and prioritization of Product Backlog items is important so the Development Team can self-organize the work within the Sprint.

If the Product Backlog items are understood well, it will help the Development Team make informed choices in their implementation and reduce risk. It will also reduce the likelihood that the Sprint will be cancelled because of problems hidden in ambiguity. This is one important reason why it is a good idea to have the Product Owner come from the customer organization. The real-world problems the Product Backlog items solve can be defined well with the help of domain knowledge rooted in experience.

If the Product Owner comes from the software organization, this domain knowledge has to be acquired from the customer organization’s experts. In a situation like this, it should be clear how the Product Owner can acquire this information. Be it a single point of contact or multiple experts in the customer organization, there is a risk of information loss if there is ambiguity in this matter.

7. **Product Owner should have sufficient domain knowledge or at least resources to receive such knowledge**

3.1.4 Definition of ”Done”

Definition of “Done” is the commonly agreed state when a Backlog item is considered complete. This is when a Backlog item turns into an Increment of ”Done”. The purpose of this definition is measure progress towards the Sprint Goal and to guide the Development Team to estimate the work for the Sprint in relation to the Product Backlog.

The Scrum Team must define this definition for themselves by understanding the product and its environment. As the Scrum Team matures, it is expected that the definition of “Done” becomes more strict to make higher quality possible. As the definition of “Done” evolves, it may uncover work in previous Increments.

Customer perspective: The definition of ”Done” should be understood by the Product Owner. This also aids in defining the Product Backlog items as it will be clear for all when one task ends and another begins. When the Product Owner comes from the software organization, the definition should be agreed together with the customer. When an item is considered ”Done” it should not be visited again. Whenever this definition of ”Done” changes, it must be agreed upon by everybody in the Scrum Team.

8. The definition of ”Done” should be kept transparent from the beginning

Below is the complete list of customer communication guidelines with Scrum as defined in the previous chapters. This is by no means an exhaustive list but it collects some of the most important things when working with the customer interface in a Scrum project. This list of guidelines will be reintroduced and analyzed in Chapter 6 as material for an improved model for customer communication.

1. A suitable Product Owner should be chosen, preferably from the customer organization
2. Product Owner should understand everybody's role in the Scrum Team and what is expected of her
3. Emphasis should be on electing an experienced Scrum Master
4. Scrum Team should make sure the Product Owner can do her job properly
5. If the Product Owner cannot do her job properly, a new Product Owner should be elected
6. Rules of the development process should be understood and agreed upon by all from the beginning of the project
7. Product Owner should have sufficient domain knowledge or at least resources to receive such knowledge
8. The definition of ”Done” should be kept transparent from the beginning

3.2 Lean software development

Lean software development is the application of lean manufacturing principles to the field of software. The idea was first introduced in 2003 in the book *Lean Software Development: An Agile Toolkit* by Mary and Tom Poppendieck. [12] This book is also used as the primary source in this chapter for introducing concepts of lean. Similarly to the previous Scrum chapter, lean concepts can be considered mature enough to use a single veritable source of information.

Applying new principles such as lean to organizations is not simple; it requires a change in culture and organizational habits that may be difficult so some companies. First, a distinction must be made: *Principles* are guiding ideas and concepts about a discipline. *Practices* are the daily actions that are done to carry out these principles. While principles work universally, it can be hard to see how they apply to specific situations. Practices need to be adapted to the domain, which makes ”best practices” a bit misleading as they are highly contextual. Software development is a very broad discipline and practices for one domain within it may not apply easily to another. However, the principles work if they are appropriately translated to practices for each domain.

The theory of lean software development is comprised of seven principles. These seven principles are the same that were developed for lean manufacturing but adapted to the world of software development. I will go through the principles in detail in the following sections.

3.2.1 Eliminate waste

The first principle of lean is *eliminate waste*. Anything that does not add value to a product is waste. The first step to lean thinking is learning to see where the waste is. If something does not directly add value to the product, it is waste. The customer decides what is and is not valuable to the product. As we previously covered the waterfall model, Winston Royce stated in his paper that the only phases that directly contribute to the end product are analysis and coding. Based on this statement and the definition of waste in lean, it can be stated that in the waterfall model all other phases except analysis and coding are waste. Seven types of waste, that are presented below, have been identified to affect software development.

Partially done work has a habit of becoming obsolete and it slows done other necessary development. The main problem about partially done software is that it is not clear whether it will eventually work. Before it is in production, there is no way to know whether it solves the problem it is trying to. What if this piece of software never makes it to production? This can have huge financial risks, which is why minimizing partially done software is an essential way to minimize both risk and waste.

Extra processes, such as paperwork, is waste if no one cares to read it and therefore adds no value to the product. This is why any absolutely necessary paperwork should be kept short, high level and offline. The paperwork should be in a format that is easy for users and developers to quickly understand and validate. A good test to validate if paperwork is valuable or not is to see whether anyone is waiting for it. For example documenting the details of desired features should be delayed until they are implemented.

Extra features may seem harmless for the developer. Why not add a feature just in case if someone needs it or just to test out how it works? However, it is serious waste because not only does this extra feature have to be designed and implemented, it has to also be integrated, tested and even documented. There is also a great chance that this feature will be unused and eventually become obsolete. This is good rule to remember: if it was not requested, do not add it to the system.

Task switching is a great source of waste. This is why a developer should generally not be assigned to multiple projects at once. Every time a switch has to be made, there is a long time before the developer can perform efficiently in the new domain. It may be tempting to have developers working on multiple projects simultaneously, but the switching time creates a lot of waste that just slows things down.

Waiting causes unnecessary delays, which is obviously waste. Delays are common in software development but they are still damaging. Delay keeps the customer from realizing value as quickly as possible. It is especially dangerous if software is developed for an evolving domain. While it is necessary to delay decisions as late as possible, it is not wise to delay decisions if implementation cannot be executed swiftly.

Motion in lean is the concept of people and artifacts moving from one place to another. Motion is wasteful, because when a developer goes down a long hallway to get a question answered, it takes a long time to regain focus to the task at hand. Software development requires a lot of concentration, and the further a developer is pulled from the depths of

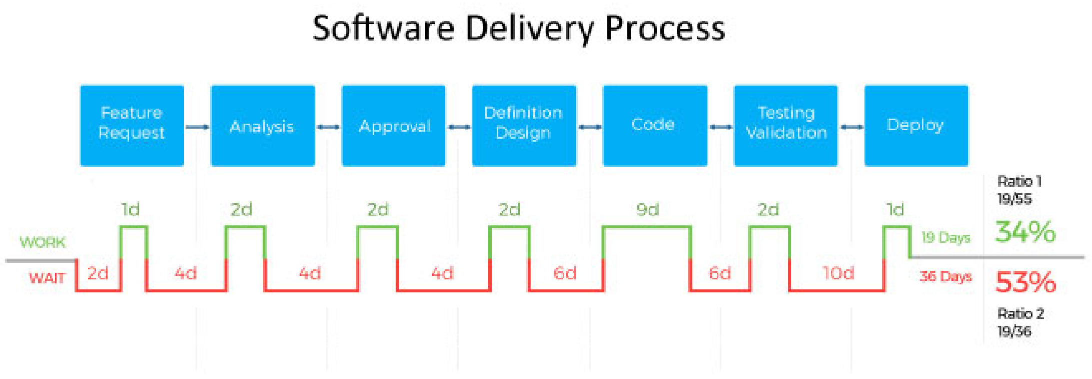


Figure 3.1: Example of a software feature's value stream [13]

their mind, the longer it takes to go back in there. This is why in agile practices it is usually recommended that teams work in a single room, where everybody is available for answering questions immediately. One example of artifacts in motion are the design documents designers make for programmers. The biggest waste in these documents is the fact that they cannot contain everything the recipient wants to know, which is why there is often a need to ask for details anyway.

Defects cause waste in relation to their scale and to the length of time they are in production. A major defect that is detected almost immediately does not cause much waste. A minor defect that goes unnoticed for days or weeks is much bigger waste. The best way to prevent these wasteful defects is to test early and test often and to release to production often.

While it is useful to know the signs of waste introduced above, it is even more important to identify them in practice. An easy way to discover where a software development process contains waste is to map its value stream. *Value stream* is a collection of steps between where a customer request is received to where it has been completed. For example the steps from when a bug is found by the customer to when a fix is finally deployed into the production application. At the bottom of the step sequence, there is a timeline that shows how long each step spends in valuable activities and how long it spends in waiting and other wasteful activities. From this map, it is easier to plan steps to reduce the waste. See Figure 3.1 for a value stream example.

Customer perspective: Starting from the first negotiations of a new software project, the signs of waste should be on everyone's mind. Does the customer understand what is wasteful? The whole product team benefits greatly if the customer knows how to identify the different faces of waste. If the customer does not understand waste, the most effective way to communicate it would be by example: map a value stream and visualize it for the customer. At first, an example value stream can be used if no actual examples are not currently available. This way it is simple to show to the customer where and how waste builds up in the development process. By identifying the waste this way, it can be dealt with swiftly. It will become easier to identify where the waste accumulates as the project goes on.

The development team should also be knowledgeable and wary of the sources of waste. Avoidance of waste is by no means a certainty even if the development team is knowledgeable in how it is created. Waste can accumulate whenever the team becomes complacent and stops paying attention to it. This is why it requires continuous mental effort to prevent waste. A system should be put in place to identify it regularly and prevent it from materializing. This system also helps with the cognitive overload of constantly having to identify sources of waste. One option for this is to regularly map the value stream of customer requests to see where the bottlenecks are formed.

1. **Sources of waste should be clear for everybody in the team**

2. **System to prevent waste should be put in place from the beginning**

3.2.2 Amplify learning

The second principle is *amplify learning*. When lean is applied to manufacturing, the main exercise is to reduce variation. However, when applied to software development, the main exercise is discovery and learning. Manufacturing is like cooking a dish while software development is like discovering the recipe. Chefs are not expected to execute a recipe perfectly on the first attempt; they iterate over several variations as a learning process to achieve a great tasting recipe that is easy to reproduce. Software development is similar, except that the team is bigger and the results are much more complex than a recipe. The best way to improve this process is to amplify learning.

In software, quality consists of perceived integrity and conceptual integrity. In this aspect software industry is more akin to service industry rather than manufacturing industry. In manufacturing there is an assumption of an unchanging set of customer expectations so the product can be replicated as accurately as possible. But in software development there is a changing set of customer expectations and the purpose of development is to produce the appropriate solutions.

The process of writing code is very similar to design; problems in software are solved at many levels and the solutions require deep understanding of the problem, experience-based pattern recognition and lots of experimentation and testing. This is why software development requires short and repeated investigation cycles and experimentation. There are two ways of developing software: The first way is to encourage developers to make sure the design and code is perfect first time around. The second way is to encourage developers to have small cycles to rapidly implement, test and fix which will eventually result in the complete piece of software. The first way does not yield good code, and the code usually needs multiple revisions before it can be considered a professional piece of work. The second way not only yields superior code, it also amplifies the developers' learning. It is also important to have the people who are knowledgeable about the specific problem to be readily available for questions, so development team can learn from them and build the product towards the desired outcome faster.

Traditional software process models often consider feedback loops dangerous because the feedback may alter the predetermined plan. Everything in these models is based on the plan: the scope, cost and schedule. This mode of thinking is still very deep-rooted in project management today in most other industries. This may be yet another reason why the waterfall model has not been abandoned. When a company has challenges in software development, they easily fall into thinking that imposing stricter control around it may improve the situation. This may introduce more documentation for requirements, agreements and other aspects around the development, lengthening the feedback loop. However, shortening the feedback loop usually has greater benefit: faster cycles to experiment by code, going closer to the actual users of the product or separating the goal into smaller tasks. The iteration based development that agile software development offers is an especially useful tool for this purpose.

Customer perspective: Problems in software systems are easy to determine, as the problems usually propagate directly to the user interface. While they are easy to spot, people who are not tech-savvy may also think they will be easy to fix. If the customer organization is inexperienced in software, it needs to be explained how a programmer works on a problem and why facilitating learning is vital. This is why knee-jerk reactions to impose stricter control should be managed. In these situations it may be crucial to remind how important iterative problem solving and high feedback loops are, so development does not become rigid and sequential.

Some domains, such as construction, work in a fundamentally different way than software. People in those domains may not understand why software development needs to be incremental and why there has to be a tight feedback loop. Customers like these may want to impose strict plans or use a sequential development model to ”decrease risk”. Especially to customers like these, the iterative nature of software development should be communicated early, and the software company may also need to take a stronger leadership position to lead the customer to an understanding.

3. **Strict processes around development should be avoided at all costs**

4. **Understanding customer's domain is relevant when teaching new concepts**

3.2.3 Decide as late as possible

The third lean principle is *decide as late as possible*. Decisions are easier to make when the future is closer and thus easier to predict. This is why it is valuable to delay decisions, so better ones can be made when there has been time to base these on facts instead of speculation. When the market keeps evolving, decisions on design are better kept open instead of committing early. In other words, there should be a capability for change within a system that follows lean principles.

Concurrent development means starting the development of the highest value features as soon as high-level conceptual design is complete for them but before the detailed requirements are done. This may sound counter-intuitive, but it can be thought of as an way to learn about various implementation options before committing to one that will constrain the development of smaller and less important features.

Sequential development (which waterfall is the embodiment of) is commonly thought to reduce risk by having the requirements and design completely ready before development begins. However, not everything can be known at these early stages, and designers easily commit to a potentially suboptimal path they cannot turn back from later.

Concurrent development can greatly reduce delivery time and overall cost if it is executed correctly. It requires some special skills. It requires the developers to have enough expertise in the domain to anticipate which way the design will evolve towards and to collaborate closely with the customer to understand how the business problem should be solved. While sequential development wants to get all the high-stakes decisions made as early as possible so the development can start, the case for concurrent development is the opposite. The high-stakes decisions should be delayed to as late as possible, because that will also reduce the amount of high-stakes constraints and reduce the need for change early in the development process. The point these decisions should be delayed to is called the *last responsible moment*. It signifies the point after which an important alternative option will be lost.

Lean software development delays committing to design decisions as late as possible also because these decisions can be easily changed when they are not yet made. Lean emphasizes a design that is very accepting of change and structures the system to be easily adaptable. Changes are usually introduced by the evolving business domain the system resides in. This is why the system should be made especially flexible in the areas where changes are expected to occur.

Predictive processes, such as what the waterfall model expects, may work in a highly predictable world. However, when customers are uncertain what they need, which they usually are, an adaptive process is much more suitable. Agile software development generates these possibilities for delaying decisions, but this does not mean agile approaches are unplanned. Plans in agile are build to maximize the flexibility to respond to change and they never speculate detailed actions in advance. Instead of speculation, experimentation and learning is used to fight uncertainty.

Customer perspective: The concept of delaying important decisions sounds counter-intuitive when heard for the first time, so great care must be used to make the customer understand the concept. It may require concrete examples to drive the point home which should be constructed if necessary. In these examples it is useful to hear the customer's perspective on the correct action and then compare it to how lean would handle the same situation.

This is also why extensive planning during the initiation of the project is not necessary and should be avoided. Once the wanted features are defined on a high level, the development can already start. If the customer wants to keep refining the details of the plan, this lean principle should be brought up.

5. **Concrete examples should be given when teaching complex concepts**

6. **Extensive planning during the initiation of a project should be avoided**

3.2.4 Deliver as fast as possible

The fourth principle is *deliver as fast as possible*. This seems very natural - customers like fast delivery. This is why immediate shipping and rapid delivery is so popular in online shopping, or any online service for that matter. Rapid development has many advantages over the careful and slow approach: Without speed there is no reliable feedback. The discovery cycle (design, implement, feedback, improve) especially characteristic of agile development is an important learning mechanism. The faster the cycle is, the more feedback is received and therefore the more can be learned. Customer requirements are never set in stone, and by having speed it can be ensured that the customer gets what they want now instead of what they wanted yesterday. It allows them to delay decision making until they really know what they want. Compressing this value stream is a fundamental strategy for eliminating waste in lean. While work in progress introduces risk, rapid delivery reduces it.

Deliver as fast as possible works together with decide as late as possible. The faster the delivery, the later decisions may be deferred. If a feature can be completed in three days, it is not critical to decide exactly what needs to be done until those three days before the feature is needed. Rapid delivery allows the team to keep their options open until uncertainty is removed and more informed decisions can be made.

Rapid delivery must be planned carefully, it does not happen by accident. Time must be optimized and all team members must know how to effectively contribute to the business. When members of the team do not know what to do, time may be lost and productivity suffers. To enable rapid delivery a *pull system* is used. Traditional work that is *pushed*

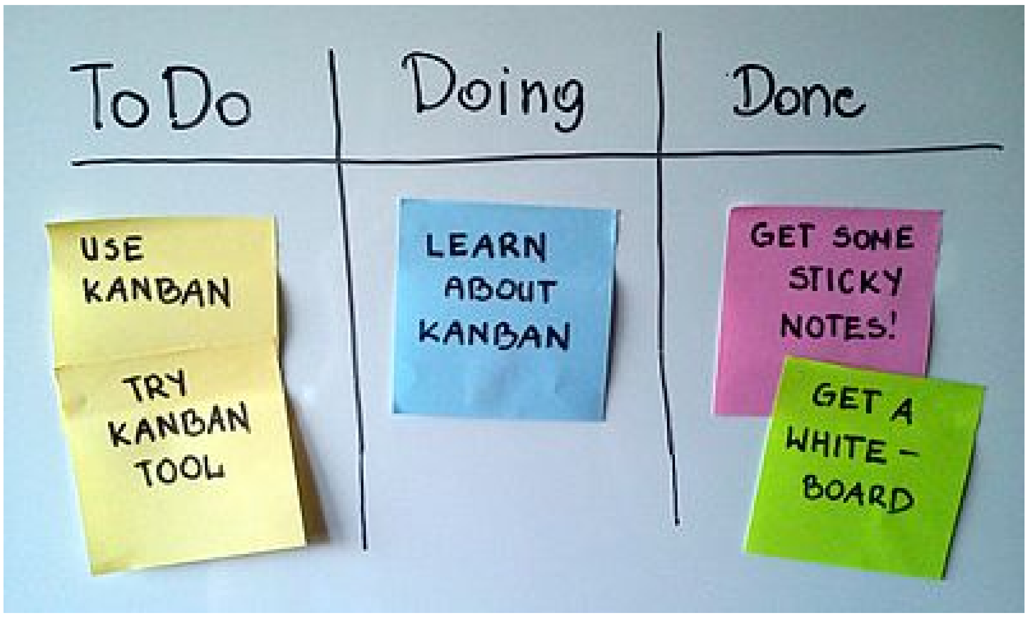


Figure 3.2: Simple kanban board [14]

from superiors to the workers does not work well in a software environment because it takes too long for information to travel up the chain of command and come back down as work. So the workers have to pull work for themselves autonomously.

The perfect tool for this is the *kanban* system. In all simplicity the idea of kanban is to write all the project tasks down on index cards. A meeting is held where developers estimate how long each task will take and the customers prioritize the cards. New cards are written in the planning meeting or when new work presents itself. The tasks on the cards should not be too big to avoid complications in implementation. This is why larger tasks should be split into smaller tasks to understand what they are made out of. These cards are then placed on a kanban board and grouped in various columns or “swimlanes”, of which there are usually at least three: “to do”, “in progress” and “done” (see Figure 3.2). There are usually more than that, describing different stages the work can be in, for example ”backlog”, ”blocked”, ”in review” and ”release queue”.

With kanban the work is self-directing, developers choose which cards they want to work on. Developers choose cards from the “to do” column one at a time and put it in the “in progress” column. It is good practice to only have one task in the ”in progress” column at a time. Once the work is done, it is moved to the “done” column and the developer picks a new card from the “to do” column. For the self-directing to work effectively, the status of the work should be clearly visible for all. This is why the kanban board should be in a visible place for everybody on the team to see, online or offline. This is called an *information radiator*.

The cards are not enough for the developers to know exactly what to do, which is why a regular brief meeting is recommended to help developers understand the big picture. This is where the Daily Scrum from agile principles is highly useful. This pull system should also be time boxed to be effective, for which Sprints from agile principles are ideal.

One important concept related to delivering fast, and one of the core concepts of lean, is the *Theory of Constraints*. The Theory of Constraints states that the best way to optimize an organization is to focus on the throughput of the organization. The way to increase throughput is to remove the bottlenecks that are slowing the completion of work. Optimizing areas outside the current bottleneck does not improve the throughput of the system. Also piling up work does not help if it cannot be completed immediately.

Customer perspective: The Theory of Constraints is one of the core discoveries of lean and it should be made sure that the customer understands it. When doing a large software project it is easy to get lost into the details of development. It may become tempting to optimize some part of the system if there are extra resources. However, this should be avoided unless the part is a bottleneck.

When the previous principle, decide as late as possible, is explained to the customer, this principle should also be brought into the same conversation as they work together. It should be emphasized that the decisions can be delayed further because the delivery will be fast. A great place to introduce these principles is at the beginning of the project when the initial plans are being drafted, to prevent them from becoming too restrictive.

7. Only the bottlenecks of the development process should be optimized

8. **”Deliver as fast as possible” and ”decide as late as possible” should be introduced together in the beginning of a project**

3.2.5 Empower the team

The fifth principle is *empower the team.* To create a high-quality product, it is important to get the details right. The people who hold the important details are the members of the development team. This is why the development team should be included in all technical decisions. Because decisions are made late and the execution is fast, an external authority figure cannot make decisions for them. So the authority lies with the developers themselves. They have to work as a self-organizing team.

Many notable IT professionals since the 1970s have written about the success of delegating power down. A lean organization respects software developers as professionals and expects them to design their own jobs with suitable training and assistance. It expects the developers to keep improving their skills and it will nourish this by giving them time and equipment to do their jobs well. When people are let to do their work freely, they motivate themselves to be much more productive than they would if they were ordered by external authority.

There is a high amount of evidence that people care deeply about their purpose and suffer greatly when they lack it. *Purpose* is intrinsic motivation that stems from the work we do, the pride we feel towards it and the feeling of usefulness. Purpose makes the work engaging. Intrinsic motivation is especially powerful if people work in a team to accomplish a purpose they all care about. For workers to keep a hold of this purpose, the purpose should be clear and compelling to the team, and also achievable to feel like the work is worth while. The closer the team works to their customer, the better feedback they get and reinforce their purpose to help the customer achieve what they want. The team should be able to choose themselves how to accomplish their work so they feel useful. The purpose of management in this scenario is to help the team accomplish their goal by getting them the resources they need and removing any obstacles in their path. Lastly, it is a good idea to keep all skeptics away from the team so they are not discouraged by negativity.

To foster motivation within the team, a sense of belonging should be an important goal. The team members must respect each other and be honest with each other. They should win as a group. If individuals within the team are given special credit, it may attract competition that creates winners and losers and kill the team spirit. The team members should also be allowed to make mistakes. If there is a mentality within the team that tolerates no mistakes, it makes people perfectionistic, wary and it kills initiative.

People need to believe they can succeed. Everybody wants to be on the winning team, and to accomplish this, the work environment should be disciplined to generate a sense of freedom. If the work environment is undisciplined, only the sense of doom is present. To create a disciplined environment, there should be best practices that are commonly agreed upon and the team members should have a mechanism to share ideas between themselves to help each other succeed.

However, no matter how well motivated the team is, if they do not see progress in their work they get discouraged. To entice a sense of progress, milestones should be planned, and at each milestone is is good practice to congratulate each other and celebrate. Medium-sized accomplishments should be celebrated with a brief escape from work and important accomplishments should be celebrated with public recognition, preferably immediately. Long days and late nights should be exercised with caution. While they may forge the team's purpose as they are sacrificing together, it may also amplify differences within the team, because not everyone may be able to make this sacrifice equally. Moderation with this kind of ”heroism” should be encouraged.

Usually in self-organizing teams, a *master developer* naturally emerges who becomes the leader of the group. This master developer can be a normal developer amongst others or a lead developer or an architect, the title is not really important. The master developer takes the leadership position because of her expertise and understanding of both the internal development goals and customer's business goals. Usually she also knows how to communicate with both parties in an understandable manner, so she becomes the focal point of the communication about the product. The self-organizing team needs this leadership to make good decisions and develop good software rapidly.

Customer perspective: In lean there is no Product Owner role like in Scrum but the master developer is the closest thing to this. While the master developer does not solely control what is on the kanban board like a Product Owner controls the Product Backlog, the master developer is the single point of communication between the development team and the customer in a similar way how a Product Owner is the link between the Scrum Team and the customer.

To build a successful project, these team empowerment guidelines should be included in the planning process from the very beginning. Especially the tight communication between the customer and the team should be facilitated in advance. These rules should preferably be integrated into the development process as an "etiquette". If everybody in the development team understand these etiquette rules it is easier to uphold them. Sometimes people have good days and sometimes they have bad days. If this etiquette is executed systematically, it can create a loop of positive reinforcement within the team that can sustain itself.

9. **Role of project manager is unnecessary in lean**

10. **Team empowerment should be practiced systematically with an etiquette**

3.2.6 Build integrity in

The sixth principle of lean software development is *build integrity in*. There are two types of integrity in a software product, or any product for that matter. *Perceived integrity* or *external integrity* is found when the user feels a product is balanced, intuitive, reliable and has everything the user needs. *Conceptual integrity* or *internal integrity* is found when the system's central mechanisms work together in a cohesive package like a well-oiled machine. Conceptual integrity is important for creating perceived integrity, it is the prerequisite. However, conceptual integrity alone is not enough. If the product does not meet the needs of the customer, the elegant architecture inside the product will be unnoticed by the users. Software needs additional integrity compared to other products as it has to maintain its usefulness over time. Software is usually expected to evolve as it adapts to the changing customer requirements. Because of this expected evolution, the software product's perceived integrity will change with the passage of time. Therefore, conceptual integrity must also change.

In *Product Development Performance* [15] Kim Clark and Takahiro Fujimoto suggested that perceived integrity reflects the information flow from customers and users to developers. On the other side, conceptual integrity reflects the integrity of the technical information flow within the development team. The way to build a system with high integrity is to have exceptional information flow through both. Research also shows that integrity is the product of wise leadership, relevant skills, working communication and good discipline. Processes and procedures cannot substitute for this.

The way to increase perceived integrity, is to keep customer values as a high priority for the development team. The master developer has very crucial role regarding this, as she has a good idea of the customer's values and she reminds the team of them regularly. When doing technical work for a long time, there is a good chance that developers get lost in the details and forget the business goals.

In sequential development models such as waterfall, the path of the perceived integrity goes through the whole process: First, the requirements are gathered from the customers and written down. Then these requirements are analyzed and based on this analysis the design is formulated. Analysis only attempts to understand what the requirements mean, but it does not give details for the implementation. The design step that follows analysis details the steps to implement the software. Only after this the design is passed down to the programmers. Often in this sequence of phases, each phase has a different group of people working on it. For example, the requirements are gathered by different people that do the analysis, and the analysis is done by another group that do the design. Perceived integrity is obviously lost here: users of a software application cannot describe what makes the app good any more than they can describe what they want in an ideal car. They just know it when they encounter it. It is simply too complex to predetermine comprehensively. Even if the requirements were perfect the first time around, the customer's perceived integrity is multiple steps away from the developers who actually create the application. So there are also multiple stages to lose information and add misinterpretations along the way. To avoid this, smaller systems should be developed with a single team that works closely to the user group of the system. When a team works closely to the customers and knows how to communicate with them, the perceived integrity of the system will be much better.

Conceptual integrity is achieved when the system has a good balance of flexibility, maintainability, efficiency and responsiveness. The system's architecture is the cornerstone of its conceptual integrity. This conceptual integrity is achieved with effective communication mechanisms within the development team. If the development team works together as a cohesive whole, often so will the system they create. It is also a good idea to use existing resources and standards. If a system is created using new and untested resources, there is sure to be difficulties with communication as well.

Another technique for achieving conceptual integrity is *integrated problem solving*. This means understanding and solving the problem should be happening at the same time. Preliminary information should be released early instead of being delayed. The idea is to transmit information in small batches, not in one large batch. This essentially means that documentation is a bad form of communication, and that most communication should happen orally.

No decisions should be made in isolation, as no one group understands the whole big picture. The development team should have experienced developers as well and these should be placed in all the critical areas of the system. In addition to experienced developers, a master developer is very useful for keeping the customer's business goals in the developers' minds.

Also, building a healthy architecture by including *testing* as an integral part of the development process and maintaining the architecture with critical *refactoring*, are great ways to build in integrity.

Customer perspective: As integrity is what creates valuable products, it should be advocated from the beginning. Whilst conceptual integrity comes from an experienced and well-communicating team, perceived integrity needs outside input. This is why the customer should be held accountable for keeping the line of communication open towards both the customer organization and the potential users of the application. Developing the software system without these is comparative to driving blindfolded.

To keep conceptual integrity an integral part of the development process, it may be beneficial to put a system in place that aids in its creation. Communication between the team members should be encouraged; this is where the team empowerment etiquette from the previous chapter could be really useful. This communication does not manifest itself arbitrarily but is rather an organic development of the team. Not all teams may naturally develop towards this, however, which is why it should be facilitated.

11. **Customer has the responsibility to keep communication lines open to themselves and the users**

12. **An environment should be created where product integrity can grow**

3.2.7 See the whole

Finally, the seventh principle of lean is *see the whole*. To have integrity in a complex system, there must be deep expertise within the development team from many different fields. Overall system performance may suffer if the experts of the team only focus on the area of their own expertise. If individuals are measured on the contributions to their specific area of expertise, there is a high possibility for a lack of optimization. This can be especially pronounced if there are multiple companies working on a product. People will naturally want to maximize the performance of their own company, which can affect the overall quality of the product detrimentally.

A system is not only a sum of its parts. It is also the product of their interactions. The system may have the best parts, but to make a good system these parts must also be working well together. This is called *systems thinking*, and it is a concept to look at organizations as systems and analyze how parts of this system interrelate and how the organization performs over time.

There are couple risks systems thinking is designed to remove. One of these risks is *limits to growth*. It means that a certain process may produce a desired result, but it will create a secondary effect that will eventually slow this progress down to a halt. The way to counteract this, is not to push towards growth, but to remove the limits to growth. When one limit is removed, another will take its place, so it will be an ongoing process. This is also the fundamental teaching of the Theory of Constraints.

Another common risk is *shifting the burden*. An underlying problem will cause symptoms that cannot be ignored. Because the underlying problem is difficult to confront, only the symptoms get addressed instead of the root cause. To counteract this problem, the root cause must be found and resolved.

The third major risk systems thinking tries to eliminate is called *suboptimization*. The more complex a system is, the more temptation there is to optimize it by splitting it into pieces and managing these subsystems locally. These subsystems often create system-wide effects that decrease overall performance. This performance decrease is often hidden, because it feels like optimizing every part of a system will also optimize the overall system. However, in lean thinking increasing the throughput of the system is more important than having a system of well-optimized subsystems. Just because all the developers in a team do their work exceptionally fast, does not mean the system will be created exceptionally fast as well. They also need to work together closely to make all the interrelated parts function. To optimize each one's performance would be foolish.

Customer perspective: Developers should not be encouraged to be busy all the time. As the developed system will have certain milestones, such as feature releases, the timing may be off if everybody's work is optimized. This is where systems thinking and Theory of Constraints come to play: you think of the development process as a system and you optimize the whole instead of the individual parts. You find the bottleneck of this system and remove it. Then you move on to the next bottleneck.

If there are multiple companies working on a single product, the whole endeavor should be viewed as a system. In other words, optimizing the output of company A does not simply make the product finish any quicker if company B is what is causing the bottleneck in the first place. In a situation like this, actions must be taken to remove these bottlenecks, be it helping company B to achieve their goals or replacing them altogether.

13. Developers should not be encouraged to be busy all the time

14. Each endeavour should be optimized as a whole

Below is the collection of guidelines for customer communication in lean software development. It complements the list of customer communication guidelines in Scrum that was introduced at the end of Chapter 3.1. Both lists are reintroduced in Chapter 6 and analyzed along with a list of guidelines from Chapter 5.

1. Sources of waste should be clear for everybody in the team
2. System to prevent waste should be put in place from the beginning
3. Strict processes around development should be avoided at all costs
4. Understanding customer's domain is relevant when teaching new concepts
5. Concrete examples should be given when teaching complex concepts
6. Extensive planning during the initiation of a project should be avoided
7. Only the bottlenecks of the development process should be optimized
8. ”Deliver as fast as possible” and ”decide as late as possible” should be introduced together in the beginning of a project
9. Role of project manager is unnecessary in lean
10. Team empowerment should be practiced systematically with an etiquette
11. Customer has the responsibility to keep communication lines open to themselves and the users
12. An environment should be created where product integrity can grow
13. Developers should not be encouraged to be busy all the time
14. Each endeavour should be optimized as a whole

4 Procurement of software

4.1 Process of buying

The process for buying software varies greatly, and certain phases may or may not appear in each buying procedure. There are as many variations in the buying process of software as there are people participating in them. For example, the more tech savvy a company is and the smaller the project is, the less of these phases there usually also are. When things are pretty clear from the get-go, there does not have to be extensive discussion over the details. Then again, if the buyer is inexperienced in buying software, some time may have to be allocated for discussing the details and even teaching the buyer how software is made and how the process moves forward. To make this easier and to skip unnecessary meetings, some companies have made software buyer's guides that explain the key concepts and processes. I will dig deeper into this in Chapter 5.

The concepts I present in the following sections concern procurements in the public sector. I present the ideal steps a public contract takes from conception to kick-off, and after these I will raise awareness to the challenges that are found in the private sector. The private sector includes the same concepts as public sector, but there is essentially no law governing how to go about making business. Therefore it is harder to prepare beforehand. On the other hand, it also has much less bureaucracy, making it much smoother when it works.

As a primary information source for these public sector procedures, I use a guide

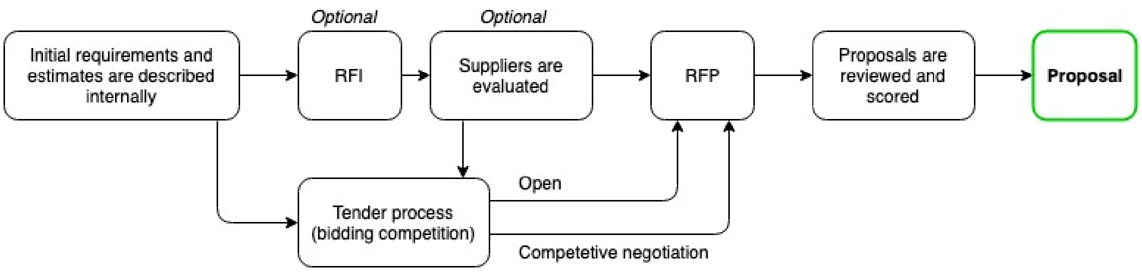

Optional
Optional
Initial requirements and estimates are described internally
RFI
Suppliers are evaluated
RFP
Proposals are reviewed and scored
Proposal
Tender process (bidding competition)
Open
Competetive negotiation

Figure 4.1: Simplification of the common software procurement process

made by a Finnish company called North Patrol [16]. North Patrol works as an counceling company for software procurements in Finland. The guide is based on the Finnish law of public procurements. While the law is worded to cover all manner of different projects, this guide is specifically tailored to software projects. It is an interpretation of this law in the domain of software. The procurement process presented by the guide is visualized in a roughly simplified form in Figure 4.1.

4.1.1 Request For Information

When procuring software development the first process the buyer organization can start with is the Request For Information (RFI). This is a process to collect information about a good or service from various suppliers to help how to move forward with the software project. The intent is usually to create a business product. If an RFI is initiated, it is usually the first phase of the buying process and there is almost always another phase that follows it. In the case of software projects this is usually the Request For Proposal (RFP).

Besides to gathering information, the secondary purpose of an RFI is to inform the suppliers of a potential need so they can start their planning and preparations for an RFP. A standard RFI will usually have the following items which should give the buyer an idea whether their idea has any merit or whether it is even possible to accomplish [17]:

- Description of the procurement target
- Delivery schedule

- Description of the expected results
- Evaluation criteria
- Terms of payment

Sales of supplier organizations will usually respond to these requests if they are interesting and align with the software organization's theme of buying.

4.1.2 Request For Proposal

Request For Proposal or an RFP (or Request For Offer, RFO) attempts to create the specifications for the software. It gives much more detail than RFI. Based on this the software organization can send out a proposal or book a meeting to discuss the specifications in detail.

A well-crafted RFP increases the odds that high-quality proposals are received and a project can be started. The Finnish procurement law restricts the negotiations between the buyer and the supplier, so it is very important that the RFP includes all the requirements and comparison criteria so a fitting supplier can be chosen. An RFP should describe how the proposals will be compared and what factors are important in the selection process. Usually these criteria are how the proposal corresponds with the requirements, the quality of the project plan and the total cost of the procurement. The price alone is rarely a good criteria, because the methods of approach between each supplier vary a lot.

An RFP should include the buyer's requirements for the project and requirements for the work tasks, skill set and methods of the supplier. The better the requirements are specified, the better the received proposals will likely be. The contract for extra work and enhancement should also be negotiated so the buyer can get their emerging needs met without the need to tender out the extra work. A good RFP also describes how the supplier's needs can be met or at the very least does not make it difficult.

Even unconditional requirements can be included in the RFP, which may concern the supplier's work references, the project team or suggested work experience and choices of technology. These unconditional requirements are meant to filter out suppliers or solutions which do not follow the criteria set by the buyer.

In Finnish procurement law, if the price of the procurement is more than 60 000 euros, it must follow the rules set by the procurement law. There are two suggested approaches to a software procurement tender process: The first is the *open procedure*, which is the most straightforward approach. It is ideal for procurements when the total price ranges between 60 000 euros and 500 000 euros. In an open procedure, the RFPs are not sent directly to chosen candidates, but the procurement must be published in the HILMA system at www.hankintailmoitukset.fi. HILMA lists all public sector procurements that follow the procurement law, i.e. exceed 60 000 euros in price. All willing parties can participate in the bidding competition. The HILMA system is essentially a filtering tool, the buyer organization can screen out the most suitable candidates for which an RFP can be sent. However, all participants must be included in the bidding competition. Open procedure aims to provide equal treatment for all suppliers so that no one-sided negotiations happen between the buyer and suppliers. The choice is made only based on the proposal documents and the criteria presented in the RFP.

The second suggested approach is the *competetive negotiation procedure*. This procedure is necessary when the procurement requires so complex and innovative solutions, that their definition beforehand is practically impossible. Commonly, the elements of software projects and their interrelations can be predicted well, so this procedure does not need to be used. However, when the software project is big, complex and the stakeholders are scattered, this procedure may be worth considering. It is important to note that the complete process may take up to six months to complete because of its many phases and arduous planning.

The procedure has two phases: *In the first phase* a sequence of negotiations are held to find a solution model that is commonly agreed upon. A procurement notice is published in the HILMA system that outlines the needs and requirements of the procurement. Suppliers are asked to apply for a bidding competition, and based on the received applications the buyer will choose which companies can participate in the negotiations. In the negotiation phase, the buyer goes through the solution plans with each chosen supplier, and builds a refined solution model based on them. *In the second phase* the suppliers are asked to send a proposal for this solution model.

4.1.3 Proposal

A proposal (or an offer) is the supplier organization's suggestion about how to create the procurement target. It contains what technology and resources would be used, preliminary costs and schedule for the project. The RFP sets the criteria for the suppliers how to craft their proposal to satisfy the needs of the buyer. When the RFP is made well, the review, scoring and comparison of proposals is simple. To avoid getting overwhelmed by the number of proposals, the RFP may include a set of minimum (unconditional) criteria that must be fulfilled in order to qualify for review.

In software projects, the proposals should never be only compared based on price. By focusing on the estimated price, quality becomes secondary. This may lead to higher costs later when the project must inevitably be maintained or enhanced with new features. A much better idea is to compare the proposals based on their quality, so the most suitable team can be found for the job. In other words, the comparison should be done based on the solution proposals, project models and maintenance services included in the proposal. These aspects are what matter most in the long-term life cycle of a software product. The best proposal is the one that can offer the most quality for the best price.

Although the emphasis on price comparison should have less than half the weight of quality comparison, it is still important nonetheless. When comparing the costs of proposals, all of the work done during the contract period should be included. In addition to development, this should also include maintenance and enhancement.

4.1.4 Answer

After a conclusion has been reached, an official answer will be crafted about the results of the tender. This document will be published for all of the tender participants to see. The procurement law states that the comparison process should be transparent, and thus objective metrics should be used. The answer should clearly state what the scoring differences between proposals were based on so that the software organizations understand the reasoning behind it. The document should also include instructions in case a loser of the bidding competition wants to complain about the conclusion.

The procurement decision that results from this answer does not yet mean that the winner of the tender is chosen to do the software project. A procurement contract will be signed separately based on the RFP's requirements. Therefore, there is time to re-evaluate the scoring in case a complaint is filed.

4.1.5 Challenges in private contracts

There can be a lot of variations to the above process. Sometimes the customer can just send out a request for information for many companies and tender out the winner, who will get to create the project. This is especially the case for tech savvy customers. Therefore, it is very useful to understand the phases of the process, in order to prepare for any sort of combination of them beforehand.

Hackathons are also commonly used to pick out a winner in a tender. The buyer organization organizes a hackathon for software companies to compete for who does the best implementation of the concept the buyer presents. Often this concept is an MVP version of the buyer's procurement target product. The winner of this hackathon can then claim the right to develop the buyer's project.

4.2 Contracts

What are the purposes of contracts? As the world gets more complex, the value of specialization increases. If the buyer wants the best kind of service for a specific domain, they can most likely get it from a supplier that is specialized in it. Additionally, it reduces costs and improves the likelihood of success if the job is left for the specialized professionals of that field.

The costs of outsourcing work is not only comprised of the money paid to the supplier for the project. Also the transaction costs should be accounted for. These include the cost of selecting potential suppliers, negotiating agreements, monitoring and enforcing the contract and billing operations. The third cost of outsourcing is the cost of ”lost opportunity” when there is poor communication between the buyer and the supplier. This is why contracts that promote collaboration are more likely to be successful and cost less. [12]

4.2.1 Fixed price

In a fixed-price contract the price of the contract is namely “fixed”, so it does not vary based on the resourced used or time spent developing. These types of contracts seem to keep their popularity despite many known failures, perhaps because of their simplicity in conceptualization and low risk for the buyer. [18]

A fixed-price contract is designed to protect the buyer, so that the price would not exceed the predetermined amount. If the supplier organizations compete only with price, it can encourage them to bid low but accumulate profit from the changes during the project. A fixed-price contract may also be an option when the buyer wants to transfer risk to the supplier. However, it is debatable whether there is much benefit from this as the buyer still suffers if the project does not succeed. Sometimes this leads to the supplier sticking to a detailed specification and charging extra for any changes, because it may be hard to get the buyer to acknowledge the completed work when the initial budget runs out.

Fixed-price contracts may involve great risk when the cost is estimated before any work has been done. A competent supplier will include this risk in the bid. A supplier that does not understand the problem, an overly optimistic supplier or simply a desperate supplier is likely to underbid others in hopes of securing the contract. Thus it is common that the buyer will choose a supplier that cannot deliver on the fixed-price contract. When this is discovered, it is usually too late.

Because the bulk of the risk is being transferred to the supplier, it leads to suppliers to protecting their interest at the expense of the buyer. This does not sound like a contract which builds trust between its parties. [12]

4.2.2 Time-and-material

Time-and-material contract is also known as *flexible-price contract* or *time-and-expenses contract*. In a time-and-materials contract the price of the contract depends on the development time spent by the supplier and any resources the supplier needs to complete their work. These types of contracts are usually used in projects where it is difficult to estimate the scale of work needed or when the predetermined requirements are likely to change along the way. Time-and-material contracts are priced in a flexible manner but they still often have an upper limit to what the project may cost. [19]

Time-and-material contracts are designed to handle uncertainty and complexity, but it shifts risk from the supplier to the buyer. They can work with agile software development if the contract makes concurrent development and collaboration easy to achieve. The iterative development that comes with agile, makes it easier for the buyer to retain the value created by the supplier. If after each iteration there is created value in the shape of working, integrated code, the buyer has the option to terminate the contract if they deem it necessary and not lose any purchased value. This is why it is also beneficial to deliver the highest priority features first. [12]

4.2.3 Multistage

There are two types of multistage contracts. First is the one that is intended to lead into a large fixed-price contract. These begin with a couple of smaller contracts to learn about the buyer's problem so a larger proposal with a fixed-price can be presented later. Generally only one supplier is involved in the contract, so if there is a requirement for a tender this may not be a suitable option. This type of contract is likely to have the same problems as the fixed-price contract, because it is likely that a hard specification is drafted and any deviation from this will cause pushback.

Second is the one that stays multistage for the whole length of the contract. This type of contract works well with agile. However, there are many opportunities for each party to abandon the relationship which may cause an imbalance in ”power”. One side may have a lot to lose if the other side leaves. This is why it is a good idea is to deliver value incrementally according to spent money. Another good idea is to deliver the highest priority features first and deliver working code during every iteration. Therefore the increased risk can be mitigated by maintaining trust between parties so that neither party has the need to leave the relationship. [12]

4.2.4 Target-cost

In a target-cost contract, the supplier and the buyer agree on the total target cost of tools, resources and changes. If the target is unreachable, parties negotiate who must pay for the extra cost. Because both parties may be paying more, incentive is created to work together so that the cost can be within the target. What differentiates this from a time-and-materials contract is that the supplier does not get benefits from prolonging the contract duration, but instead if they are under agreed costs. A valuable aspect of the target-cost contracts is that the cost expectations are communicated to everybody in the development team so it is easier to work together to plan the design to fit the target costs.

To work properly, a target-cost contract has to provide incentive for the buyer to keep their demands for features within target costs. The supplier must in turn be provided with incentive to complete the work within target costs. There are two common ways to incentivize this behaviour: In the first way, *cost plus fixed fee*, the supplier profit is not included in the target cost. However, this means there is a higher profit margin for the supplier the better the total cost fall below the target cost. Often a bonus is included if the work is completed below the target cost. If the contract goes above the target cost, the supplier usually agrees to work without profit for the remaining duration of the contract.

The second way is called *profit not to exceed*, and this time the profit is included in the target cost. The supplier agrees to reduce their rates after the target cost is reached and work without profit. However, since the profit is already in the target cost, this does not incentivize the supplier to complete work under the target cost. For this reason, a bonus should be awarded for completing work under the target cost. [12]

4.2.5 Target-schedule

If the team remains unchanged throughout the project and no resources are purchased or licensed, the target schedule and the target cost are equivalent. A target-cost contract can be turned into a target-schedule contract if the resources and schedule is fixed.

Software should also be deployed a good time before the deadline so any emerging bugs can be dealt with before the target schedule and cost are reached. Target-schedule contract makes the most sense, when the schedule is the only thing that is important, and the team is allowed to freely purchase resources to speed up the development.

4.2.6 Shared-benefit

Shared-benefit contracts want both parties to share the risks and rewards of the work. One way to do this is *profit-sharing* which means the profits made by one party is shared to the other party to incentivize collaboration. Another way is a *co-source contract* where the supplier's team is expected to teach the buyer's team to do the work for themselves, as the development progresses.

4.2.7 General agreement

A general agreement is a contract between one or many buyers and one or many suppliers. General agreements are especially suitable for procurements where the prices and the products develop quickly and when it is not beneficial to commit to fixed prices or requirements. These types of contracts are very common in the IT industry, and they are primarily used to produce savings.

The buyers must choose one or more suppliers from the parties of the general agreement. In other words, when a procurement process is started within the general agreement, only these parties are to be included in the tender process. When drafting a procurement notice, RFI or RFP, the amount of suppliers required for the procurement must be specified. The procurements created within a general agreement must include all original parties of the general agreement. In other words, once a procurement process has begun, it is not possible to add new parties to the general agreement. Additionally, the terms of the general agreement must not be significantly altered during its time of validity. If a buyer does not want to make use of the general agreement process, it can tender out the procurement normally according to the procurement law.

Finnish procurement law specifies a general agreement may be valid for a maximum of four years and it may only exceed that time in special circumstances. The procurement process within general agreements does not significantly differ from the one specified by the procurement law. [20]

4.3 Actors

It is hard or even impossible for the software organization to affect how the buyer organization and its stakeholders communicate internally. The aim is to do this with the proposal itself as its the first concrete thing that the customer representative will show the customer organization's management. Negotiations before this proposal help work towards a mutually beneficial outcome, but the proposal is the first step of turning the negotiations into something concrete.

4.3.1 Customer representative

The customer representative is the first point of contact between the buyer organization and the supplier organization. She communicates the wishes of the buyer organization to the supplier's development team and works together with the team to create the software product. In Scrum the customer representative is usually also the Product Owner, who manages the requirements list of the product.

The development team should work well together with the customer representative as she is the primary communicator of the buyer organization's wishes. By working well together, the customer representative will also positively influence the whole buyer organization to work better with the supplier. This co-operation can even breed new business, mutually beneficial business opportunities if continued long enough. A great strategy is to include the customer representative in the suggested team empowerment etiquette presented in Section 3.2.

4.3.2 Management

The management of the buyer organization wants to make a good product like everybody else. However, because they are usually far removed from the actual development of the product and they are restricted by budgets and organizational politics, their decisions may sometimes be disruptive. Counseling the management about the product is important, and this is the job of the customer representative. Having a great relationship and good communication with the customer representative, as presented earlier, is a great way to positively influence the buyer organization. This can help them understand the supplier's point of view better. The supplier company should also keep communication with the buyer company as clear and concise as possible, to facilitate mutual understanding. This especially includes the proposal that is presented to the buyer. The management is the one that makes the decision for starting or ending a project. They are the ones that the proposal needs to convince. If the proposal sounds unconvincing or unclear or it poses too much financial risk, they may not want to risk the future of the company. [21]

4.3.3 Sourcing department

Sourcing is an umbrella definition for other terms such as outsourcing, procurement or purchasing. In other words, it is the section of the company that handles all of its procurements. For ease of reading, I will continue to use the term "sourcing" not to confuse it with the act of software procurement. Sourcing is defined as "*integrating and coordinating common items, materials, processes, technologies, designs and suppliers across worldwide buying, design and operating locations*" by Trend and Monczka (2005). [22] As per the definition, sourcing department is only relevant for large and often multinational companies that need to coordinate their procurement efforts by a central authority. Small and medium-sized businesses can usually manage their procurements together with their sales and management

The act of sourcing does not only concern the buying from suppliers but also coordinating other operational and design functions. However, most goods and services a company buys (instead of makes) is the responsibility of the sourcing department.

Sourcing can be divided into centralized and decentralized sourcing. *Centralized* sourcing is sourcing on a global scale. When the company has many locations across the world, it can decide where to acquire goods and services based on where they can get the best quality and lowest prices. *Decentralized* sourcing is sourcing on a local scale. Sometimes procuring a solution from a single global supplier is not good enough, because it needs to be tailored for a certain locale. This is when relationships with local suppliers and service providers must be managed at a local level to achieve a suitable solution. [23]

This balance act between global and local sourcing is one factor that affects how the sourcing department works with its organization and software suppliers. The sourcing department must also follow organizational policies that affect procurements. For example, if the buyer company requires general agreements sourcing department must make sure the supplier companies have these before any contract can be made. These things are important to know, because the decisions of the sourcing department may sometimes feel like they do not align with those of the other actors. When a supplier company is beginning negotiations with a buyer, a good idea is to find out if the buyer has a sourcing department and what are their procurement strategies. This may be useful for the future; if the sourcing department's agenda is understood, its decisions in the future about specific procurement facets will not come as a surprise.

4.3.4 Development team

The development team consists of the software developers, designers, analytics and other necessary entities that create the company's products. Generally, the development team wants to make the best product they can and is not concerned about organizational politics or costs, even though these things do restrict the work.

If the customer organization has a development team of its own, there may be a situation where the supplier organization may need to work together with them. The customer organization may have a certain piece or pieces of technology, e.g. a database, that the development team needs to integrate into. Effective communication strategies should be used with these external development teams as well. Members of the customer's development team may not understand the complete context of the product, as they may not be actively invested in it, so the customer representative (and potential Product Owner) should be used as support in communication.

4.3.5 Multisourcing

Multisourcing is when a buyer company selects and combines IT and business services from multiple supplier companies. This is common for larger companies and government agencies that need a very tailored solution. Multisourcing also makes sure there is an air of competition between the supplier companies which keeps the service quality high and the price competitive. [24, 25]

In a multisourcing environment, the development team should work together just as cohesively if they were a part of the same supplier organization. The benefits are shared after all, so there is no need for competition within the development team. The competition has already happened when the most suitable experts have been selected for the job from each supplier company. There only needs to remain a secrecy regarding company secrets or strategies.

5 Buyer's guide for software development

The problem that exists between agile software development and rigid billing practices is caused by lack of understanding and a fear of the unknown. When the customer does not understand what creates value in software development, the natural instinct is to keep it under strict control. This imposed control is what restricts development and kills creativity. The solution to this is simple: to increase understanding. This can be done through communication.

A buyer's guide is an informative text that is written for the buyer of software development for a custom digital service or solution. Its purpose is to inform the buyer on how to make an informed purchase and to make the buying process much easier and less intimidating. A buyer's guide for software development usually explains the best practices of modern software development, the buying process and also the importance of soft skills.

Software buyer's guides have been created to educate buyers for the benefit of both the buyer and the supplier. Software projects have traditionally used fixed pricing, because it poses less risk for the buyer and software development has had a bad reputation of projects that have overspent their allocated resources. However, this fixed pricing does not work well with the agile nature of software development that needs breathing room to evolve towards a desirable outcome. This is why many other contract types have been introduced to bring more flexibility into the software development billing, such as the ones introduced in Section 4.2.

These different contract types alone are not enough to fix the problem. Each different contract type works in a different way, and there is no one-size-fits-all solution. The correct type of contract must be negotiated between the buyer and the supplier. To find a contract that is suitable for both the buyer and the supplier, there must be trust and understanding between both parties. This trust is not easy to come by when a buyer is still trying to find the best supplier for their product.

This is the best place to introduce a buyer's guide. The buyer's guide is valuable in two aspects. Firstly, it will teach the buyer valuable information about software development and the buying process. Secondly, this ”free value” that the guide gives the buyer can increase trust in the buyer that the supplier actually knows what they are talking about. This is not to say all software companies should have their own buyer's guide, as this may become repetitive for the buyer to read multiple guides that contain essentially the same thing. However, it is a good idea for the supplier to demonstrate in an early phase that they understand how to create a safe buying and development process for both parties.

To demonstrate this value the buyer's guide can bring into the buying process I will analyze a buyer's guide *Ohjelmistokehityksen ostajan pikaopas* [26] by Vincit Oy. Vincit is one of the biggest software consulting companies in Finland and they have written a 40-page buyer's guide to tackle the problem that is so common in modern software procurement. I will be analyzing it in terms of usefulness to the customer and especially how it highlights the importance of communication. I will gather these highlights into a list at the end of this chapter. This guide is the only source I will use in the following sections.

5.1 How to think of software development?

The first chapter has an introduction that brings forth the benefits that this guide offers. It states that the guide is aimed for people who want to become better at buying software development. To start this off, the guide explains that the success of a project is based on the success criteria. There is an example of the movie ”Titanic”, which doubled its planned budget to 200 million USD from the initial plan of 100 million. However, after its release Titanic was the most financially successful movie of all time for the next 10 years. Bringing perspective to the success criteria like this is a good idea, because most people know Titanic. This also conveys the message that even though a project may fail based on its initial success criteria, it may still be an overall success.

1. **Re-evaluate the project's success criteria throughout the project**

The guide also introduces the concept of a *project triangle* (see Figure 5.1). Project triangle communicates what kind of attributes a successful project is dependent on. At each of the triangle's edges, there is an attribute: cost, time and scope. Usually all three of these want to be locked in the beginning of a project. However, usually one or two of these attributes can be locked whereas the remaining is diminished. In other words, if the cost is locked, it diminishes the possible time and scope of the project. However, the problem identified with this model is that it does not recognize quality or the value it produces.

Therefore, another triangle has been proposed by Jim Highsmith in his book Agile Project Management. The triangle looks otherwise the same, but in this agile triangle the three edges are instead: value, quality and restrictions. The most important thing about this agile triangle is to acknowledge value and quality as goals for the project. Value is more important than simply the number of features in the scope. Quality is another attribute that is often forgotten. Especially the long-term effects of low quality.

Some of the concepts in this chapter are quite vaguely introduced. The fact that rigid

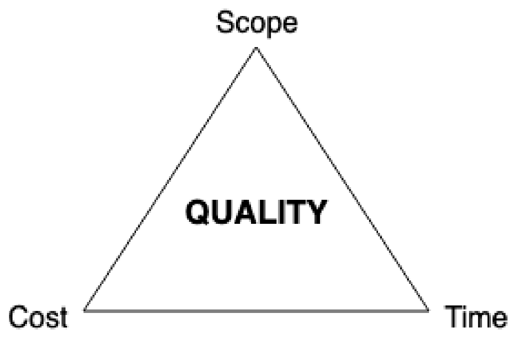


Figure 5.1: Project triangle

plans eventually lead to lower quality software is stated in a single sentence. This should be elaborated in a bit more detail, because it can be hard to justify shifting risk to the buyer if they do not understand the reason behind it. It is a good idea to include straightforward examples, when presenting complex concepts instead of staying on a theoretical level.

2. Present complex concepts with easy-to-understand examples

The rest of the chapter is material for how to help the software buyer to see the value of the product and ensure the quality of implementation. There are good points about keeping the product simple. Although the word ”simple” may be a bit misleading. The idea is to emphasize the core features of the product, the attributes that create the most value.

An example is given of Apple, how their products are simple but they still make great profit. In truth, their products are not simpler than those of their competitors but they have focused on their core value: making their products easy to use. In other words, it is more important to focus on the few features that create the most value, rather than all of the possible features you could have. This is a great guideline to remember in all software projects.

3. Core value of the product should be figured out and focused on

5.2 How to think of software in terms of value?

In the second chapter, the guide suggests that the problem should be defined in as few sentences as possible. Once the problem is defined well, the solution can be planned. To persuade a supplier of this solution, the guide suggest writing an elevator pitch. This pitch may have the benefit of engaging the supplier right from the start. At the very least, it will give the supplier a better idea whether they are the right fit for the project.

Also, the concept of *minimum viable product* (MVP) is introduced. MVP means a version of the product created by the least effort that is required to solve the business problem. An MVP is common practice in modern software development, as it validates the solution in practice with minimal risk. However, MVP does not mean the product needs to be released for use, but the MVP can be used as a base for the first release.

The guide smartly recommends limiting the development time of MVP artificially, so it does not consume too much resources. This requires careful analysis of the core values, and hence the core features, of the product. The guide also recommends that the MVP includes the features with the highest risk which are also the features with the highest value. By including the highest value features first, creating value will become a cornerstone of the project.

4. Highest value features should always be developed first

Role of end users in the development of the product is also emphasized. Instead of creating a product that needs an extensive manual, user experience should be at the forefront of the product values. User feedback will determine which features are deemed valuable and what needs improvement. The role of user experience designers and business designers is explained and spotlighted in this chapter.

What this chapter could have used is some more visualization. Although the textual description of MVP and user experience is fine, a good visualization would have brought much needed context to the concepts. After all, some people may be introduced to these concepts for the first time in this chapter. All in all, the guide is well illustrated with many cartoonish images that describe the subjects symbolically. This makes the guide much more pleasing to read. However, most of these are not informative and are there purely for aesthetics. More educational illustrations about the concepts themselves would make the text easier to understand.

5. **Visuals can be used to communicate difficult and important concepts**

5.3 How to choose the right supplier?

Chapter three is all about getting into the mindset to choose a supplier. Suppliers are chosen too often based purely on data: CVs, written answers and proposals. Perhaps this is to not let feelings affect business decisions. However, the guide suggests getting to know the people behind the supplier companies to get to know the best fit for the product. After all, it is hard to know what kind of a team is behind a list of CVs. Some important traits are how well a team works together, how they react to challenges and how diligent they are.

6. **Face-to-face meetings should be prioritized when meeting with the customer**

The guide also discourages only looking at hourly rates when choosing suppliers, and instead going directly to the supplier and asking about their previous products and services to find out how long they took and what was accomplished. This helps in uncovering how productive each supplier really is, which may not be evident in their hourly rate. When the buyer finds out how each supplier's team works together and what they can accomplish, it is much easier to make the choice. Hourly rates are the best suited billing option for software projects, at least when compared to fixed-pricing. In Section 4.2 we introduced multiple different types of contracts in the software development field, but the guide wisely only compares fixed pricing to hourly pricing to keep it simple.

More complex types of contracts are mentioned only briefly. The real conflict is usually between these two opposing approaches. Fixed-price shifts risk to the supplier, whereas hourly rates shift it to the buyer. Buyers are likely to shift more of the risk to the supplier, when no trust has evolved between them. The guide does give some options to bridge this gap of trust: satisfaction guarantees, MVP implementation or test sprints. Also in the beginning of the project, it is less intimidating for the customer to use a hybrid approach between fixed-pricing and hourly pricing. When trust is gained, it is easier to start using hourly pricing which allows for more flexibility.

7. **Flexibility should be applied in the beginning negotiation process, so a mutually beneficial agreement may be discovered**

The writers of the guide understand how important trust is in the software procurement and development process and they emphasize its importance. Sometimes processes and agreements are used to replace the trust between the buyer and the supplier. The guide strongly recommends against this. However, the buyer should still not trust blindly. The guide explains that the buyer should make sure they receive all of the project material and the intellectual property rights in the contract. This way, after the development ends with the first supplier, they can continue development with another team or an in-house team, because they own the rights to the software.

5.4 How is software developed?

In the fourth chapter, the guide gives a brief outline of waterfall and the basics of agile software development, and why it is important to know the difference. The principles of agile development and especially Scrum are well introduced in a short span of text but, again, visualizations would have been very beneficial in this section. The buyer is suggested to ask the supplier if they do not understand something, which is great advice.

It is also important to know when to stop development. The guide informs that if the backlog is groomed properly and the development team meets with the customer regularly, it will be known well in advance when the project is coming to an end. Because the development should be done following agile principles, creating big plans should be discouraged.

At the end of the chapter, a list called "checklist for the buyers of software development" is presented with tick boxes for important concepts introduced in previous chapter. This is a great way of bringing the concepts into the real world, and making them actionable for the buyer. Using this list, the buyer can proceed through the guide step-by-step.

5.5 How to develop software efficiently?

The fifth and final chapter attempts to dissuade the buyer from making extensive plans for the software development process. The supplier should be able to start development as soon as possible from the project kick-off. The guide suggests that critical thinking should be applied if the supplier needs to study before starting implementation, or if it takes unusually long to get tangible results. This may be a sign of a lack of vision or skill. In these cases, the buyer should question the supplier in their choices of technology, whether it is suitable for the project or not. Despite discouraging extensive plans, the guide encourages constant planning during the project in what to do and how much should be invested in each aspect of the project.

8. **Visible results should be made in the first two weeks of a project to increase trust in customer**

Features should be implemented end-to-end and one at a time, containing both back end and front end portions. Both of these concepts are also introduced in this chapter. The importance of testing and quality control is also highlighted.

Finally, the relation of man months to actual progress is questioned. Increasing the team size after a certain point will actually slow down the development. Usually this increase in team size happens because of an upcoming deadline and technical debt. A suggested solution to this is to reduce the amount of features so no technical debt may accrue.

9. **Increasing team size should be avoided**

At the very end of the chapter a list is presented with tick boxes that contain good software development practices introduced in the chapter. These list items also contain textual descriptions of the concepts, some of which, surprisingly, have more information than described in the main text itself. The introduced concepts are sound, and it is a great idea to gather them into an actionable list as well.

The guide ends with a epilogue chapter that brings awareness, once again, to trust. To increase trust between the buyer and the supplier, a good idea may be to choose a supplier that would also use the product that they are developing. This trust is nurtured the easiest by meeting face-to-face. Digital ways of communicating help with communicating through distance, but they are not useful to nurture trust. Public references of the project in the supplier's website also increase trust, and the guide encourages this.

10. **If possible, the development team should use the product they are developing**

5.6 Analysis

All in all, the guide gives a good overview of what it takes to develop modern software. It smartly emphasizes the importance of trust between the buyer and the supplier as it makes the product development much easier. Understandably, the guide cannot be too long or go in-depth into to the subject, because it could lose the interest of the reader.

However, the most important thing was mentioned: through close co-operation between the buyer and the supplier, almost all challenges can be overcome. Trust is very important, because the buyer must trust the supplier is capable and willing to make the best product they can. The supplier must also trust that the buyer is as invested in the development process as they are and that they are invested in creating value, in addition to profit, with the product.

Analyzing this guide, I found guidelines I have gathered in a list below. Two of these were potential improvements to the guide itself. I will go through these in more detail in Chapter 6 where I present an improved model for communication during software procurement and development.

1. Re-evaluate the project's success criteria throughout the project
2. Present complex concepts with easy-to-understand examples
3. Core value of the product should be figured out and focused on
4. Highest value features should always be developed first
5. Visuals can be used to communicate difficult and important concepts
6. Face-to-face meetings should be prioritized when meeting with the customer
7. Flexibility should be applied in the beginning negotiation process, so a mutually beneficial agreement may be discovered
8. Visible results should be made in the first two weeks of a project to increase trust in customer
9. Increasing team size should be avoided
10. If possible, the development team should use the product they are developing

6 Improved communication model

In this chapter I introduce improvements to the common software procurement process and highlight the areas, where the supplier should focus their communication. I do this by reintroducing the customer communication guidelines I gathered in Chapter 3 and Chapter 5. From each of these sets of guidelines, I will highlight the most important ones and go through them in detail. Most of the guidelines gathered from Chapter 3 are process-specific. If a software should be developed using Scrum, more care should be taken into reading the Scrum section and vice versa. Regardless, it is important to understand both software process models to make an informed choice between them. Understanding the context behind these guidelines is also necessary, so the previous chapters of this thesis are assumed as background information.

6.1 Scrum guidelines

Below are the guidelines for a Scrum process introduced in Section 3.1. The supplier should naturally understand how Scrum works, in order to understand the situations where to use these guidelines. Theory of Scrum should also be educated to the Product Owner (in case the Product Owner is from the customer organization, otherwise the customer representative) if necessary. Aside from understanding Scrum, these guidelines should be high in the list of priorities. They are some of the most important, but also most often neglected, things that impact the success of the project. Most of them are for the beginning of the project which is where the first critical mistakes are made.

1. **A suitable Product Owner should be chosen, preferably from the customer organization**
2. Product Owner should understand everybody's role in the Scrum Team and what is expected of her
3. **Emphasis should be on electing an experienced Scrum Master**
4. Scrum Team should make sure the Product Owner can do her job properly
5. If the Product Owner cannot do her job properly, a new Product Owner should be elected
6. **Rules of the development process should be understood and agreed upon by all from the beginning of the project**
7. Product Owner should have sufficient domain knowledge or at least resources to receive such knowledge
8. The definition of ”Done” should be kept transparent from the beginning

From this list I have highlighted three guidelines I think are the most important to achieve a successful software project. The first guideline I have highlighted is Guideline 1: ”A suitable Product Owner should be chosen, preferably from the customer organization”. If the Product Owner is well-qualified and skilled, the project is at a great advantage from the very beginning. If the Product Owner understands what are the core values of the customer's product and knows how to emphasize them, the Scrum Team can trust in her to make correct decisions for the product. An adept Product Owner will also convince the customer organization to make good decisions for the product. Also, it increases trust within the Scrum Team when everybody can work together towards a common goal without internal or external struggles.

However, in the real world complications may occur and the potential Product Owner may not be as skilled as the Scrum Team would optimally want. The emphasis should be to find the best-suited Product Owner for the product who can then learn on the way. The supplier organization may not be able to affect this outcome so other guidelines should be consulted in a case where the Product Owner is lacking in some areas.

If the customer organization does not have any potential Product Owners, the only option may be to choose one from within the supplier organization. This is usually not the ideal solution, because a Product Owner from the supplier organization will not have the same domain understanding as a Product Owner from the customer organization would have. In this situation Guideline 7 is very important.

The second guideline I have highlighted is Guideline 3: ”Emphasis should be on electing an experienced Scrum Master”. When the Scrum Master is experienced, the Scrum Team will be much better-equipped to tackle the inevitable obstacles during development. An experienced Scrum Master will make sure everybody is doing the right things at the right time and learning how to best work together. She will also fill any knowledge gaps the Scrum Team may have. In the negotiation stages of a software project, a Scrum Master can make sure that the project will have a good starting point for an iterative development environment to grow.

The third highlighted guideline is Guideline 6: ”Rules of the development process should be understood and agreed upon by all from the beginning of the project”. By choosing a certain development process, it is wise to understand the alternatives as well to be sure that the choice is correct. The software project will have potential for a good start when the the development process is determined in the very beginning of the project, in the negotiations phase, and understood by all stakeholders. When especially the Scrum Team and the customer organization’s stakeholders understand the development process, it is easier to make decisions based on it. If there is vagueness in the air, as there usually is when people talk about concepts they do not understand, there is a risk of miscommunication and error. This understanding cannot really be explicitly validated, but it should be carefully observed during communication. If it sounds like there is vagueness in a subject, a good idea is to focus on it by asking questions or using other means to deliberately expose and remove it.

The other guidelines I did not specifically highlight from the list are still very important, even though they were not selected. By focusing on the highlighted three in the beginning, the project has a great chance of being a success. Most of the other guidelines were specifically aimed for acquiring a suitable Product Owner for the project, when there are problems in this regard. This is why, if there is a great Product Owner from the beginning, these guidelines are less important. Then the focus should only be on close communication and teamwork.

6.2 Lean guidelines

Below are the guidelines for lean software development process introduced in Section 3.2. The team should be well versed in lean before reading these guidelines. These guidelines are similar to the Scrum guidelines in that the customer representative should be aware of the principles of lean so executing the lean practices will be easier. Most of these guidelines are lean-specific but some of them can be applied to any development process. Also, similarly to the previous chapter, I have highlighted the guidelines that I think are the most important to keep in mind when choosing the lean development process.

1. **Sources of waste should be clear for everybody in the team**
2. System to prevent waste should be put in place from the beginning
3. **Strict processes around development should be avoided at all costs**
4. Understanding customer's domain is relevant when teaching new concepts
5. Concrete examples should be given when teaching complex concepts
6. Extensive planning during the initiation of a project should be avoided
7. **Only the bottlenecks of the development process should be optimized**
8. ”Deliver as fast as possible” and ”decide as late as possible” should be introduced together in the beginning of a project
9. Role of project manager is unnecessary in lean
10. Team empowerment should be practiced systematically with an etiquette
11. **Customer has the responsibility to keep communication lines open to themselves and the users**
12. **An environment should be created where product integrity can grow**
13. Developers should not be encouraged to be busy all the time
14. Each endeavour should be optimized as a whole

I have chosen five of the lean development guidelines to highlight and explain further. Again, this does not mean the other guidelines are less important than these five. However, with these five highlighted guidelines, the project has a good chance of being successful.

The first guideline I have highlighted is Guideline 1: ”Sources of waste should be clear for everybody in the team”. Waste is one of the most important concepts of lean software development, which is why it should be clear to all in the team how to identify it. Continually eliminating waste should be a goal right from the start alongside product development. I chose Guideline 1 instead of Guideline 2 because it is more important to know how to identify sources of lean first. Once everybody knows how to identify the sources, a system can be constructed to prevent waste from accumulating.

The next highlighted guideline is Guideline 3: ”Strict processes around development should be avoided at all costs”. Strict processes hinder creativity and construct unnecessary filters in development. Strict processes are usually imposed because of fear. These processes will also hinder the all-important feedback loop that the development team needs from the customer. To prevent this, there needs to be sufficient trust between the buyer and the supplier. After all, why would one want to closely supervise something they already trust is matching their expectations. The buyer needs to be sure that the supplier’s development team can handle any challenges and hardships the development may face. This means the development team should have a close relationship with the customer representative, and the development team should be able to communicate any encountered problems swiftly so they can be solved together.

Lean software development is all about optimization. However, one of the important lessons learned from lean is the Theory of Constraints: a system is restricted by constraints or ”bottlenecks”. By identifying and eliminating these constraints, the system’s throughput will increase. Like the saying goes, a chain is as strong as its weakest link. This is why I have highlighted Guideline 7: ”Only the bottlenecks of the development process should be optimized”. When optimizing a system, such as a software development process, the optimizations should only be targeted towards the bottlenecks. Improving the throughput before or after the bottleneck does not increase the throughput of the system, because the constraint still lies in the bottleneck. The development team should be able to identify where the bottlenecks (or sources of waste) are and improve only these areas, starting from the most severe and moving on in a descending order.

The fourth highlighted guideline is Guideline 11: ”Customer has the responsibility to keep communication lines open to themselves and the users”. Like mentioned previously in this section, a high feedback loop between the development team and the customer representative is very important. It makes sure that any misunderstandings or problems are discovered at an early stage. The customer is the most knowledgeable in the product’s domain so any actual usage problems can be identified early, when the customer is actively reviewing what is happening in the development.

The users of the product should also be included in this feedback loop. Actual feedback from users is crucial, when the product is already in production. A good method to get feedback from users is to hold a user testing event, where new users test the app and give feedback on the user experience. Another good idea is to include a feedback section in the app and review the feedback regularly with the development team. There are also programmatical tools for gathering anonymous user feedback from web applications, in the case where the product is open to the public.

Sometimes the customer representative may be busy or otherwise unavailable to be very closely involved in the development. The importance of feedback should be emphasized from the development team’s side and its benefits explained. Once the customer understands how important feedback is to iterative development, there will hopefully be improvement in this regard.

The fifth guideline I highlighted is Guideline 12: ”An environment should be created where product integrity can grow”. We remember from Section 3.2 that product integrity is divided in external and internal integrity, or perceived and conceptual integrity. Perceived integrity is when the user feels a product is balanced, intuitive and reliable. Conceptual integrity is when the system works internally like a well-oiled machine. Both of these are required for a product to be successful. Perceived integrity is created by keeping customer values as a high priority. This is also why customer and user feedback (Guideline 11) is very important. Conceptual integrity is created when the system has good architecture and the team works well and communicates together. In other words, the communication within the development team and between the development team and external stakeholders needs to be working well, in order to facilitate the growth of integrity. Having effective communication should be one of the main goals of the development team. There needs to be a balance of just enough communication that it is meaningful, but it does not become noise. This is an ongoing process throughout the development, but it is very much achievable in time.

Again, the highlighted guidelines are only better in comparison, because they encapsulate most of the important aspects of the lean process. The other guidelines are almost equally as important but by focusing on these five, the project can be well on its way towards success.

6.3 Procurement and development guidelines

In Chapter 5, we saw how useful a buyer's guide is for new buyers of software development. A supplier that wants to stand out from others may want to present a text like this in their website or even give out a printed copy during first meetings with the buyer. However, it does not necessarily need to be a buyer's guide per se. The supplier needs to convey how they develop software and why their method works. This information can be conveyed in different ways. For example, the information could simply be spread out in the supplier's website. Alternatively, there could be a whole section in the website for educating buyers with articles or even videos. The primary objective should be to give free value to the customer and help them understand how software is developed in the supplier's organization.

The following guidelines have been gathered from the buyer's guide I analyzed in Chapter 5. The guidelines listed are generally for the software procurement and development process, and not as process-specific as the guidelines gathered from Chapter 3 that I introduced previously in this chapter. I have highlighted three of the most important guidelines from the list below.

1. Re-evaluate the project's success criteria throughout the project
2. Present complex concepts with easy-to-understand examples
3. **Core value of the product should be figured out and focused on**
4. Highest value features should always be developed first
5. Visuals can be used to communicate difficult and important concepts
6. **Face-to-face meetings should be prioritized when meeting with the customer**
7. **Flexibility should be applied in the beginning negotiation process, so a mutually beneficial agreement may be discovered**
8. Visible results should be made in the first two weeks of a project to increase trust in customer
9. Increasing team size should be avoided
10. If possible, the development team should use the product they are developing

The first highlighted guideline is Guideline 3: ”Core value of the product should be figured out and focused on”. To create a successful product, it should also have a strong brand. A brand encapsulates what the users expect from the product and other products from the same company. Therefore, it ties in closely with the core values of the product. The core values of the product are what build its brand, i.e. what users will come to expect from it. For example, Apple products have a brand of being well-designed, high-quality and easy-to-use. Therefore, those qualities are also what users expect from Apple products. It is certain that Apple has explicitly defined these qualities to be their core values that they abide by. Who would not want to create a brand like Apple?

The second highlighted guideline is Guideline 6: ”Face-to-face meetings should be prioritized when meeting with the customer”. Regular meetings with the customer should be held face-to-face to increase the quality of communication. The more there is face-to-face communication, the better the product quality will usually be. This is because usually the team also learns how to work together better this way and problems are solved faster. Purely digital communication always creates a slight disconnect between the participants and it increases the possibility of miscommunication.

Today, there are three main types of communication used in software development: First and most common is textual communication with email or instant messaging services, such as Slack or Microsoft Teams. Second is video conferencing services such as Google Hangouts and Skype. Third is face-to-face communication within a team space or a prearranged meeting. From of these, face-to-face communication should be preferred, but if this is not possible, a video conference is the next best thing. However, video conferences are usually useful when the communication is around a subject that is complex or requires input from many people. It is a bit too cumbersome to have a video conference over every little minute detail. This is where team chat services are useful. They should be used when the topic discussed is not too complex or the concept discussed is already clear for everybody and the communication is around the details of it. Another advantage chat services have is that they have a history that can be used to store information for later consumption.

The third and final highlighted guideline is Guideline 7: ”Flexibility should be applied in the beginning negotiation process, so a mutually beneficial agreement may be discovered”. During the time of procurement and negotiations, there should be flexibility from both the buyer’s and supplier’s side to move towards a satisfactory plan for both parties.

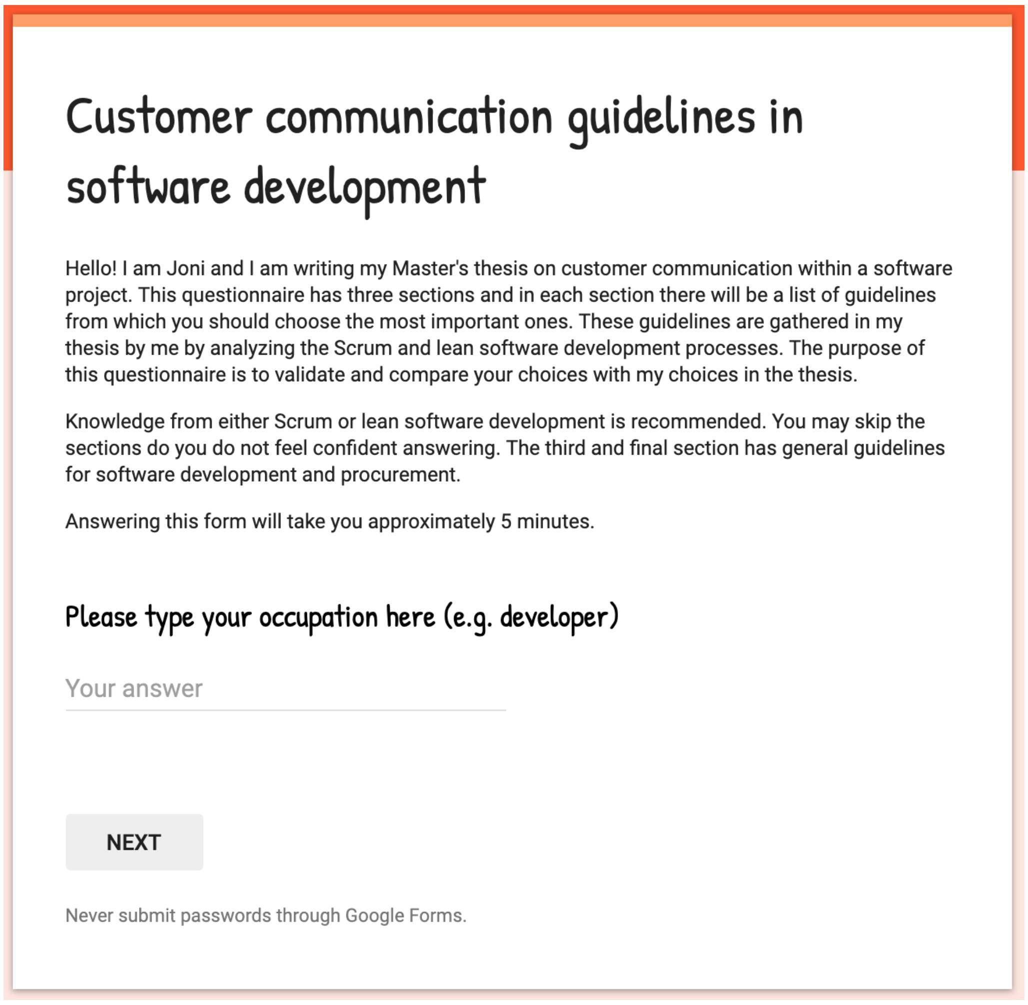
However, even though this thesis contains guidelines from the supplier's perspective, it should not invite the supplier to think that they know best in all situations. To achieve a successful project, there needs to be mutual respect between the buyer and the supplier, and they need to actively listen to one another. Without this respect, the project will surely have problems down the road as both parties will view the other less as people and more as tools to achieve their goals. It should be reminded that both parties are working together to achieve common benefits for both. Perhaps the product stakeholders could even have common events where they can meet together outside of work, to make everybody feel like they work in the same team towards a mutually beneficial goal.

6.4 Validation

In the previous sections I presented three lists of customer communication guidelines I think are the most important principles to keep in mind when communicating within the customer interface. In each of these lists, I highlighted the most important guidelines I thought deserved the most attention. In this section, I will validate these guidelines by presenting them to developers and other people within the IT industry. They will then select the most important guidelines in a similar fashion as I did in the previous sections of this chapter. I will then compare their selections with my own to gain insight on how my analysis stacks against the popular opinion.

I will present the three lists of guidelines to the employees of my software organization and to my LinkedIn network. I will have two separate forms to separate the answers and because the groups may be varied in their skill set. My software organization is primarily comprised of developers or other people closely tied to the development interface. My LinkedIn network is comprised of many different professional roles, although most of them are from the technology industry.

However, the way the Google Forms questionnaire is written may dissuade non-



Customer communication guidelines in software development

Hello! I am Joni and I am writing my Master's thesis on customer communication within a software project. This questionnaire has three sections and in each section there will be a list of guidelines from which you should choose the most important ones. These guidelines are gathered in my thesis by me by analyzing the Scrum and lean software development processes. The purpose of this questionnaire is to validate and compare your choices with my choices in the thesis.

Knowledge from either Scrum or lean software development is recommended. You may skip the sections do you do not feel confident answering. The third and final section has general guidelines for software development and procurement.

Answering this form will take you approximately 5 minutes.

Please type your occupation here (e.g. developer)

Your answer

NEXT

Never submit passwords through Google Forms.

Figure 6.1: Description of the questionnaire (LinkedIn)

technical people from answering. The Scrum and lean sections in the questionnaire are very process-specific and may as such discourage people from answering, if they are not familiar with the approaches. I tried to word the description of the questionnaire to encourage answers despite this, as the third section is generally about customer communication in software development.

6.4.1 Validation method

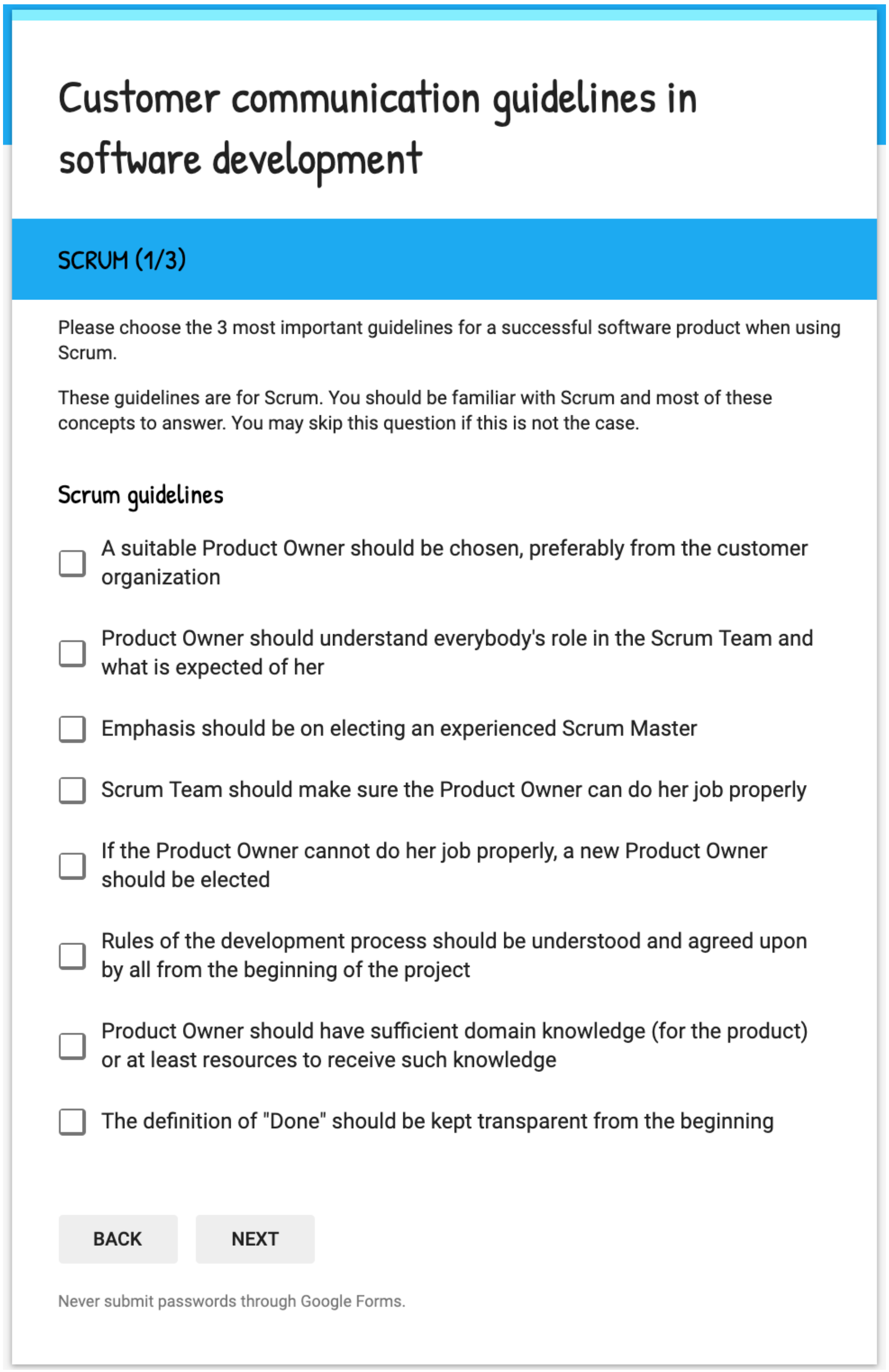
The method of validation is a questionnaire created with Google Forms. The questionnaire has five sections. The questionnaire begins with a description of its contents in the first section (Figure 6.1). I tried to keep it as simple as possible and maximize valid answers by suggesting skipping sections that may be unknown to the respondent. Additionally, the LinkedIn form includes a text field to enter the respondent's occupation. I chose this approach, because I could not be sure of the professional roles of everybody who participate in this sample. This text field is not present in the form I shared to my software organization, because almost all of the answers will definitely be from developers. The two forms were also color coded for easy cognitive separation, blue for the form I presented to my organization and red for the form I presented to my LinkedIn network.

The next three sections (Figures 6.2, 6.3 and 6.4) contain each a list of checkboxes, where the respondents must highlight an equal amount of important guidelines as I did earlier in this chapter: three for Scrum, five for lean and three general guidelines. The fourth and last section is a text field for open feedback (Figure 6.5).

When presenting this form to my software organization and my LinkedIn network, I made sure to mention the questionnaire is short and it contains skippable sections in order to maximize the amount of respondents. Aside from this, I gave a very brief description of the subject of the thesis and what the questionnaire contains.

6.5 Results of questionnaire

The questionnaire was open for two and a half days and it received 41 responses: 33 from my software organization and 8 from my LinkedIn network. In the beginning of the LinkedIn form there was a text field the respondents could write their occupation. The results can be seen in Figure 6.6. However, it seems most of the respondents were software developers despite my initial suspicions. When analyzing the results, I will use the term *blue form* to refer to the form I presented to my software organization and the term *red form* to refer to the form I presented to my LinkedIn network.


Customer communication guidelines in software development

SCRUM (1/3)

Please choose the 3 most important guidelines for a successful software product when using Scrum.

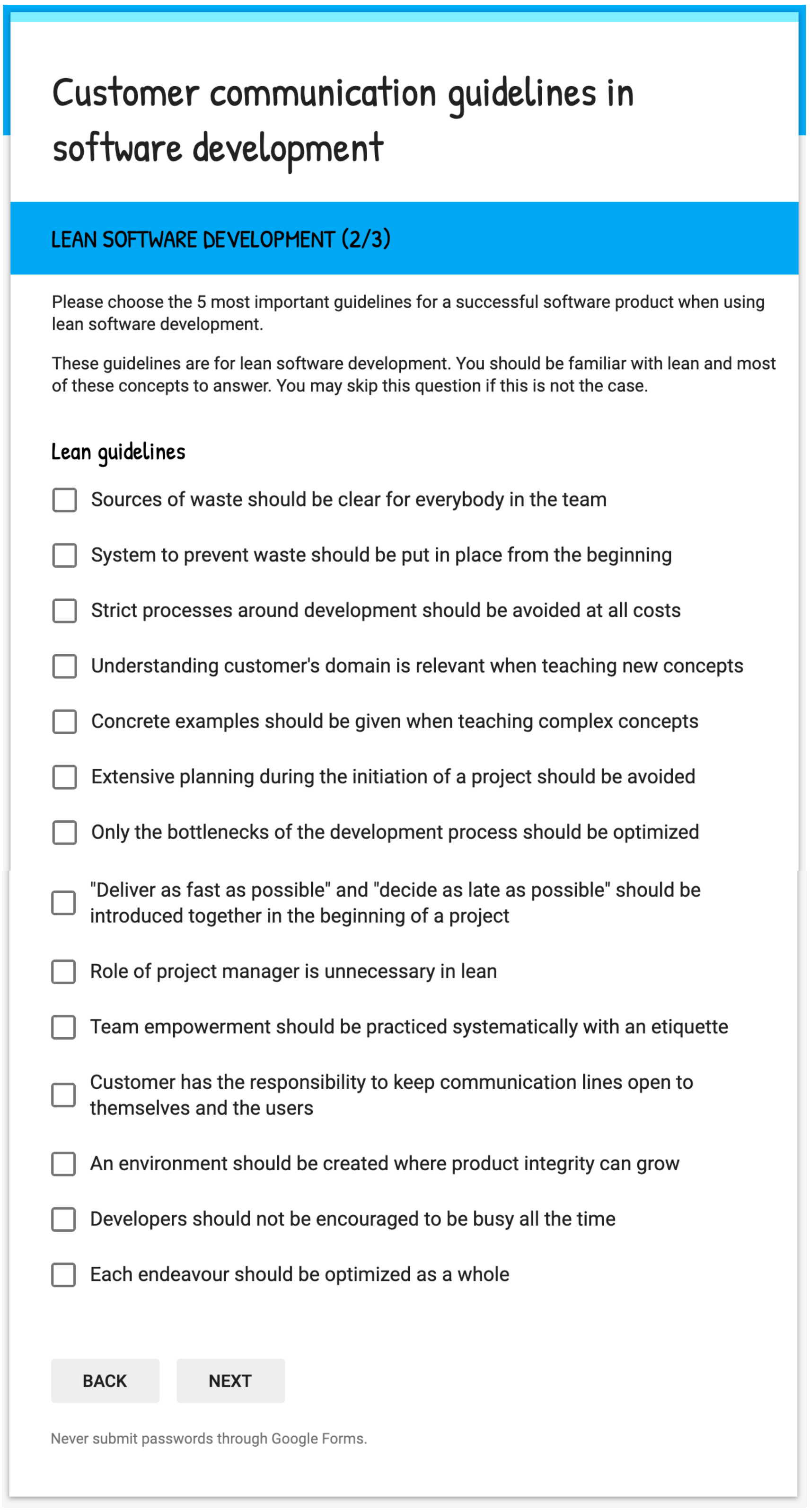
These guidelines are for Scrum. You should be familiar with Scrum and most of these concepts to answer. You may skip this question if this is not the case.

Scrum guidelines

- ☐ A suitable Product Owner should be chosen, preferably from the customer organization
- ☐ Product Owner should understand everybody's role in the Scrum Team and what is expected of her
- ☐ Emphasis should be on electing an experienced Scrum Master
- ☐ Scrum Team should make sure the Product Owner can do her job properly
- ☐ If the Product Owner cannot do her job properly, a new Product Owner should be elected
- ☐ Rules of the development process should be understood and agreed upon by all from the beginning of the project
- ☐ Product Owner should have sufficient domain knowledge (for the product) or at least resources to receive such knowledge
- ☐ The definition of "Done" should be kept transparent from the beginning

Never submit passwords through Google Forms.

Figure 6.2: Scrum guidelines in the questionnaire


Customer communication guidelines in software development

LEAN SOFTWARE DEVELOPMENT (2/3)

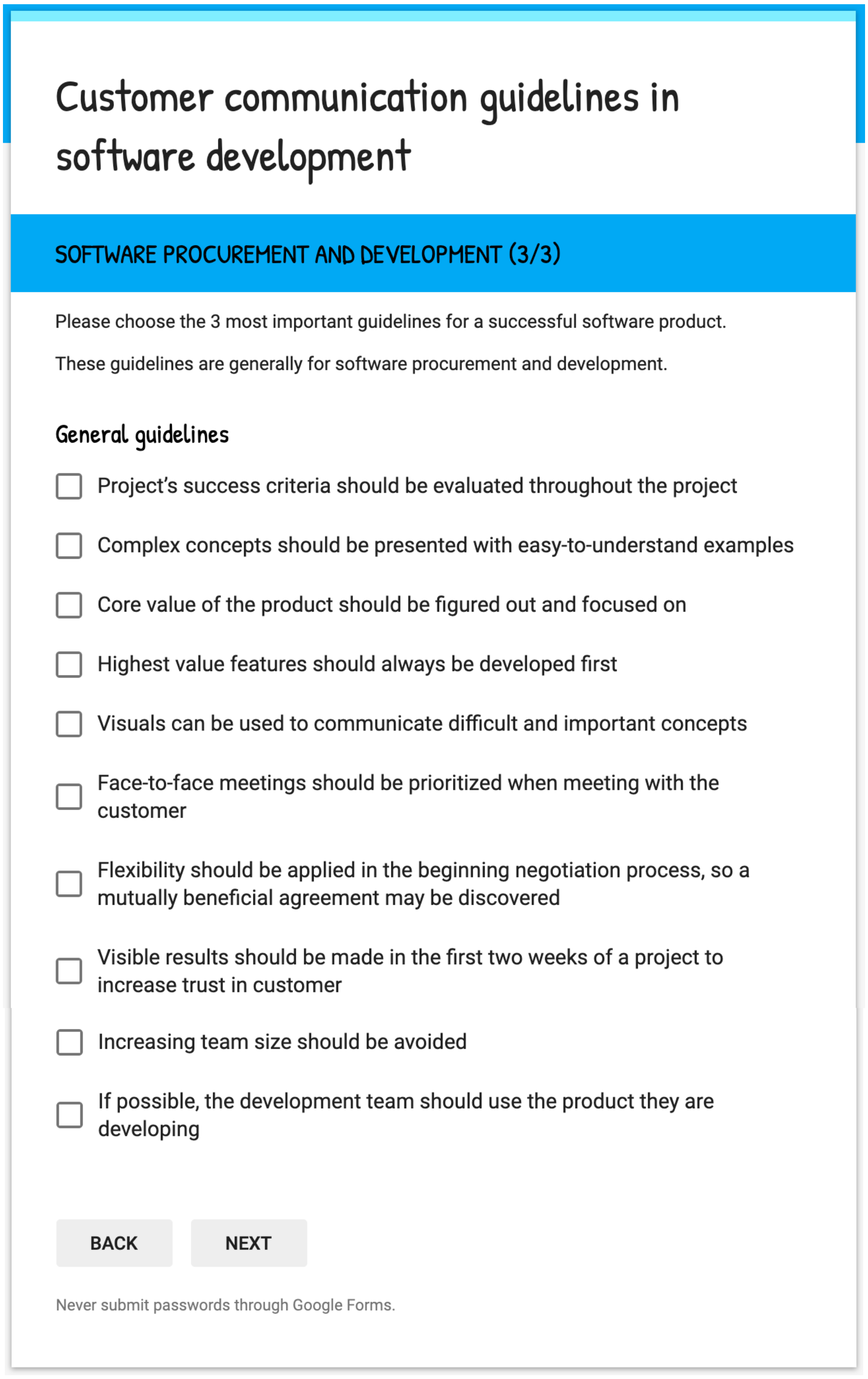
Please choose the 5 most important guidelines for a successful software product when using lean software development.

These guidelines are for lean software development. You should be familiar with lean and most of these concepts to answer. You may skip this question if this is not the case.

Lean guidelines

- ☐ Sources of waste should be clear for everybody in the team
- ☐ System to prevent waste should be put in place from the beginning
- ☐ Strict processes around development should be avoided at all costs
- ☐ Understanding customer's domain is relevant when teaching new concepts
- ☐ Concrete examples should be given when teaching complex concepts
- ☐ Extensive planning during the initiation of a project should be avoided
- ☐ Only the bottlenecks of the development process should be optimized
- ☐ "Deliver as fast as possible" and "decide as late as possible" should be introduced together in the beginning of a project
- ☐ Role of project manager is unnecessary in lean
- ☐ Team empowerment should be practiced systematically with an etiquette
- ☐ Customer has the responsibility to keep communication lines open to themselves and the users
- ☐ An environment should be created where product integrity can grow
- ☐ Developers should not be encouraged to be busy all the time
- ☐ Each endeavour should be optimized as a whole

Figure 6.3: Lean guidelines in the questionnaire



Customer communication guidelines in software development

SOFTWARE PROCUREMENT AND DEVELOPMENT (3/3)

Please choose the 3 most important guidelines for a successful software product.

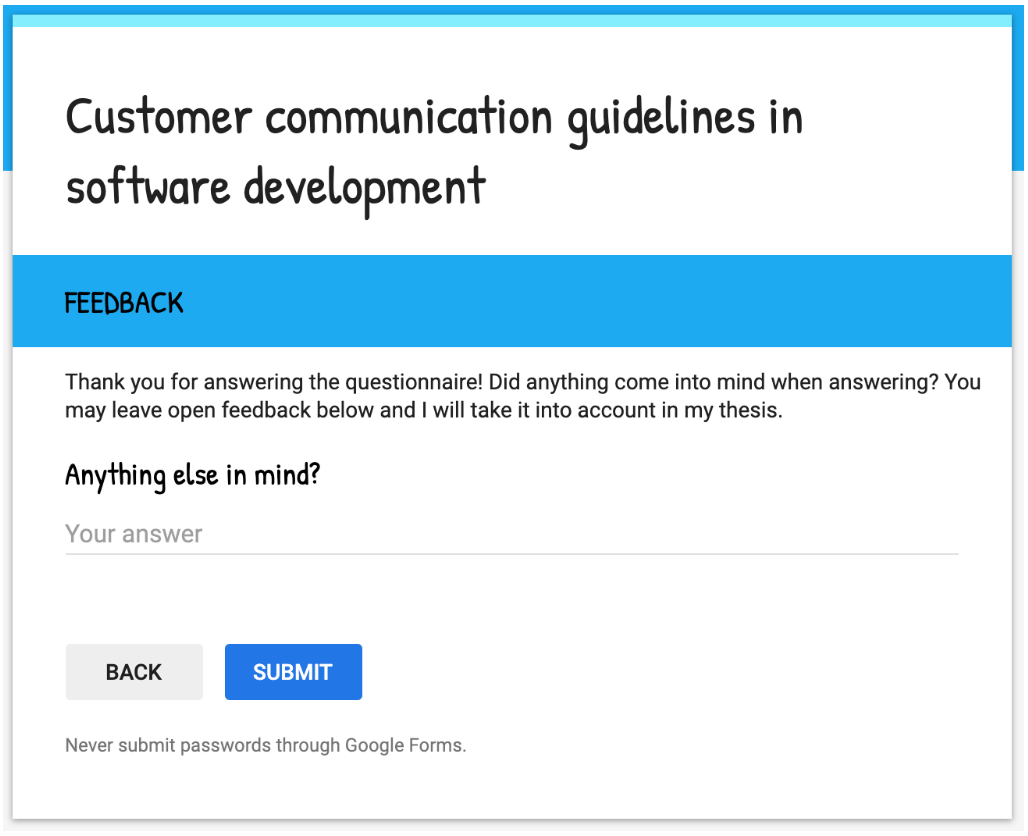
These guidelines are generally for software procurement and development.

General guidelines

- ☐ Project's success criteria should be evaluated throughout the project
- ☐ Complex concepts should be presented with easy-to-understand examples
- ☐ Core value of the product should be figured out and focused on
- ☐ Highest value features should always be developed first
- ☐ Visuals can be used to communicate difficult and important concepts
- ☐ Face-to-face meetings should be prioritized when meeting with the customer
- ☐ Flexibility should be applied in the beginning negotiation process, so a mutually beneficial agreement may be discovered
- ☐ Visible results should be made in the first two weeks of a project to increase trust in customer
- ☐ Increasing team size should be avoided
- ☐ If possible, the development team should use the product they are developing

Never submit passwords through Google Forms.

Figure 6.4: Procurement and development guidelines in the questionnaire



Customer communication guidelines in software development

FEEDBACK

Thank you for answering the questionnaire! Did anything come into mind when answering? You may leave open feedback below and I will take it into account in my thesis.

Anything else in mind?

Your answer

Never submit passwords through Google Forms.

Figure 6.5: Feedback section in the end of the questionnaire

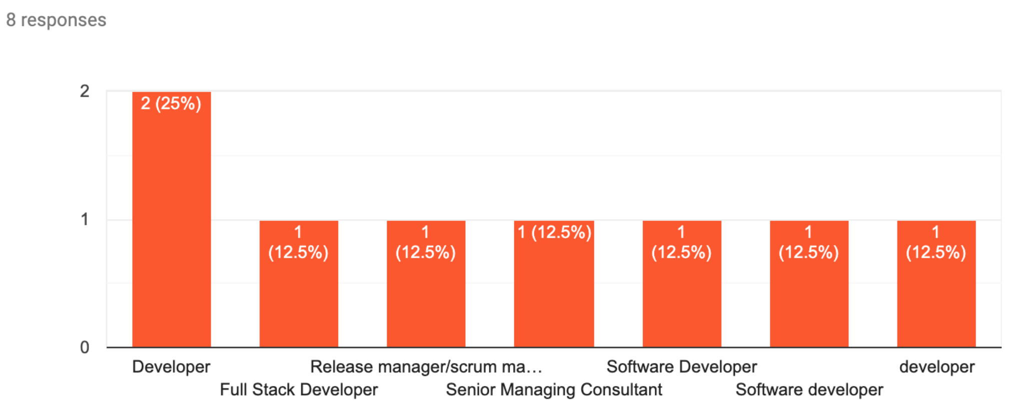


Figure 6.6: Occupations of LinkedIn respondents

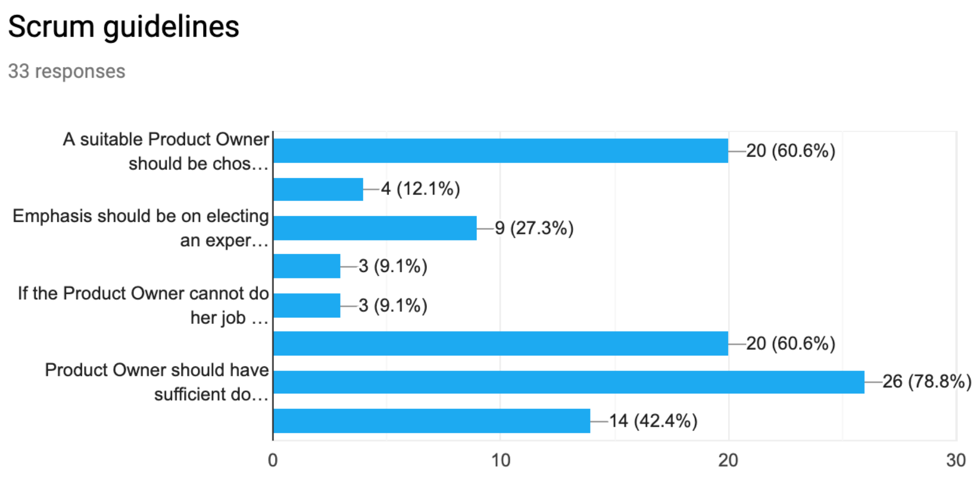


Figure 6.7: Results for Scrum guidelines

6.5.1 Scrum guidelines

The results for Scrum guidelines from the blue form can be seen in Figure 6.7 and from the red form in Figure 6.8. Unfortunately, Google Forms does not display the results data in a very readable manner. Despite this shortcoming, the data points are in order, so they can be examined based on the original list.

As a reminder, I highlighted Guideline 1: ”A suitable Product Owner should be chosen, preferably from the customer organization”, Guideline 3: ”Emphasis should be on electing an experienced Scrum Master” and Guideline 6: ”Rules of the development process should be understood and agreed upon by all from the beginning of the project”. Most of my choices have received some votes, especially in the blue form, where guidelines 1 and 6 both received 20 votes. Surprisingly, from the red form nobody selected Guideline 3. Is the role of the Scrum Master not important to some people? Perhaps it is not important, when a Scrum Team already knows how to work well within the rules of Scrum. Reaching this stage takes time in a software consultancy environment with changing projects and team compositions.

However, many more votes were given to Guideline 7: ”Product Owner should have sufficient domain knowledge or at least resources to receive such knowledge”, which re-

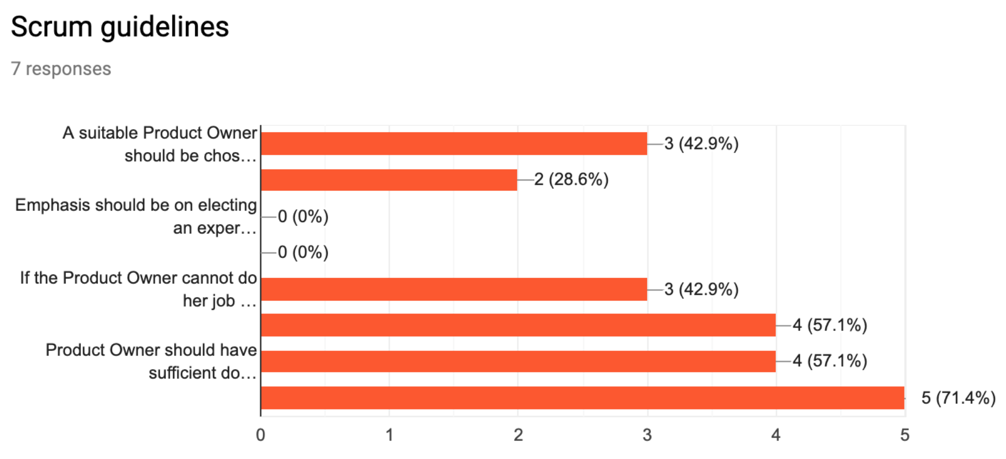


Figure 6.8: Results for Scrum guidelines (LinkedIn)

ceived 26 votes on the blue form and 4 votes on the red form. This may be a sign that in the software industry, or at least in our organization, there is a problem of Product Owners not having enough domain knowledge. It can also be simply a sign that the role of the Product Owner is very important to developers. The Scrum list contains multiple guidelines concerning the role of the Product Owner. it is interesting to see that the domain knowledge of the Product Owner is held as the most important one. Also, Guideline 8: ”The definition of ”Done” should be kept transparent from the beginning” received many votes, receiving 14 on the blue form and 5 on the red form. It was the most voted guideline on the red form, and it is not hard to see why as the definition of ”Done” truly is very important for Scrum. It is a valid argument to prioritize it higher than I did.

From the results, it is interesting to see how much less the Scrum Master is prioritized than the Product Owner. Also the role of the Scrum Team, seen in the votes for Guideline 4: ”Scrum Team should make sure the Product Owner can do her job properly” and Guideline 5: ”If the Product Owner cannot do her job properly, a new Product Owner should be elected”. Perhaps an argument can be made that it is not the role of the Scrum Team to affect the Product Owner’s performance or position. However, the reason I included it is because I feel it may sometimes be necessary if the Development Team is having trouble working with a troublesome Product Owner.

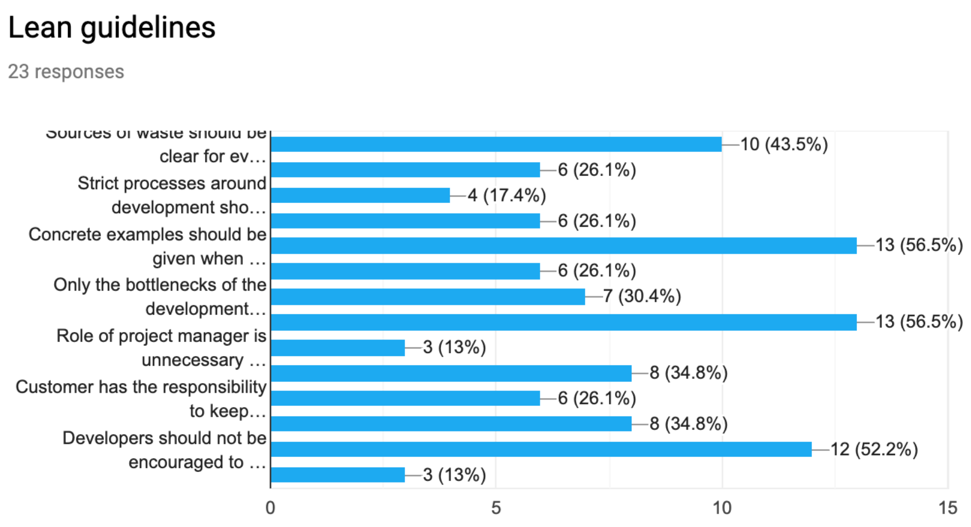


Figure 6.9: Results for lean guidelines

6.5.2 Lean guidelines

The results for lean guidelines from the blue form can be seen in Figure 6.9 and from the red form in Figure 6.10. As a reminder, I highlighted Guideline 1: ”Sources of waste should be clear for everybody in the team”, Guideline 3: ”Strict processes around development should be avoided at all costs”, Guideline 7: ”Only the bottlenecks of the development process should be optimized”, Guideline 11: ”Customer has the responsibility to keep communication lines open to themselves and the users” and Guideline 12: ”An environment should be created where product integrity can grow”.

Perhaps because of a higher number of guidelines and also a higher number of votes, the distribution of votes is much more even than in the Scrum section. It may also be because the guidelines are much more varied than the Scrum guidelines, which focused heavily on the Product Owner’s role. Comparing both blue and red forms to my votes, it seems none of them especially stand out as a self-evidently solid guideline; surprisingly, some of them even were disagreed with. In the blue form, Guideline 3 received only 4 votes and in the red form 0 votes. This may be an indication of the vagueness in the wording of the guideline. Perhaps the respondents thought by having ”no strict processes”

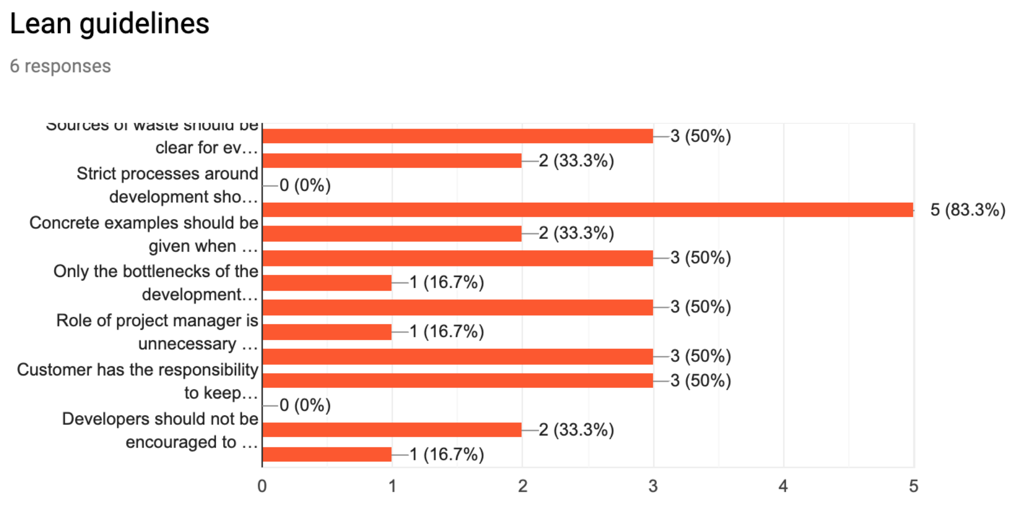


Figure 6.10: Results for lean guidelines (LinkedIn)

it would also mean development processes like Scrum or lean development? Or then strict processes are thought of as an inevitability at some point of a project's life cycle. This shows a slight weakness in my method of validation: the guidelines lack context.

Also, Guideline 12 received 0 votes from the red form. However, it did receive 8 votes from the blue form, being a middle road guideline. Perhaps the concept of ”product integrity” is not clear to either group of respondents. Or then the difference is simply because of a significant difference in sample size between the red and the blue form.

The clear highlights on the blue form were Guideline 1 about sources of waste, Guideline 5 about concrete examples, Guideline 8 about introducing ”deliver as fast as possible” and ”decide as late as possible” early and Guideline 13 about not encouraging developers to be busy all the time. All of these are very concrete guidelines related to the development of software and day-to-day work.

The red form on the other hand was much more even. There was a standout winner, with 5 out of 6 possible votes, Guideline 4 about understanding customer's domain when teaching new concepts. It is an easy pick as it is not directly related to lean development but it is important nonetheless. The rest are much more even, with guidelines 1, 6, 8, 10

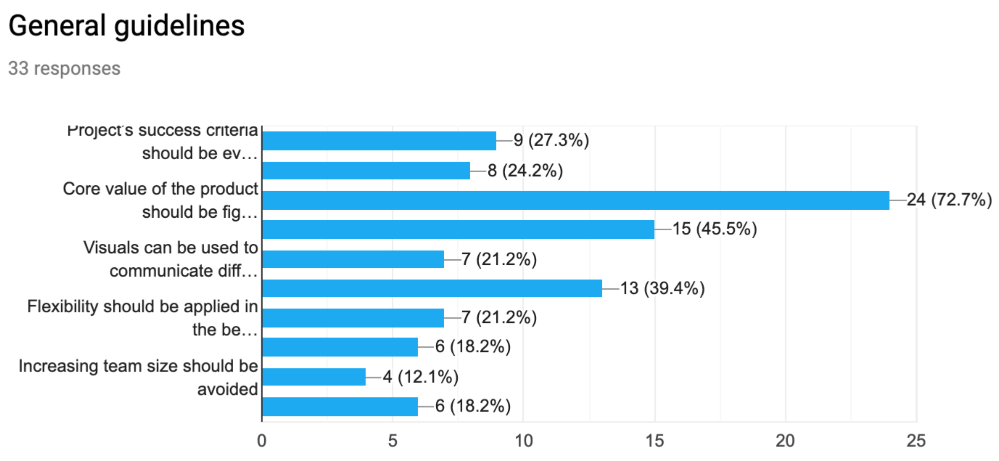


Figure 6.11: Results for procurement and development guidelines

and 11 having each 3 votes. Rest of the guidelines had less than 3 votes each.

Lean development is significantly less known compared to Scrum. Out of the 33 respondents to the blue form, everybody answered the Scrum section but only 23 answered the lean section. Similarly for the red form, out of 8 respondents 6 answered the lean section. This may also have contributed to the quality of answers. Perhaps the lean section was answered despite not knowing all of the guidelines.

6.5.3 Procurement and development guidelines

The results for guidelines generally related to procurement and development from the blue form can be seen in Figure 6.11 and from the red form in Figure 6.12. As a reminder, I highlighted Guideline 3: ”Core value of the product should be figured out and focused on”, Guideline 6: ”Face-to-face meetings should be prioritized when meeting with the customer” and Guideline 7: ”Flexibility should be applied in the beginning negotiation process, so a mutually beneficial agreement may be discovered”.

From all forms, a clear winner can be picked out as Guideline 3 is the most voted on all forms, sharing a number one spot with Guideline 4 on the red form. The core value of the product is obviously very important to the product and most developers seem to

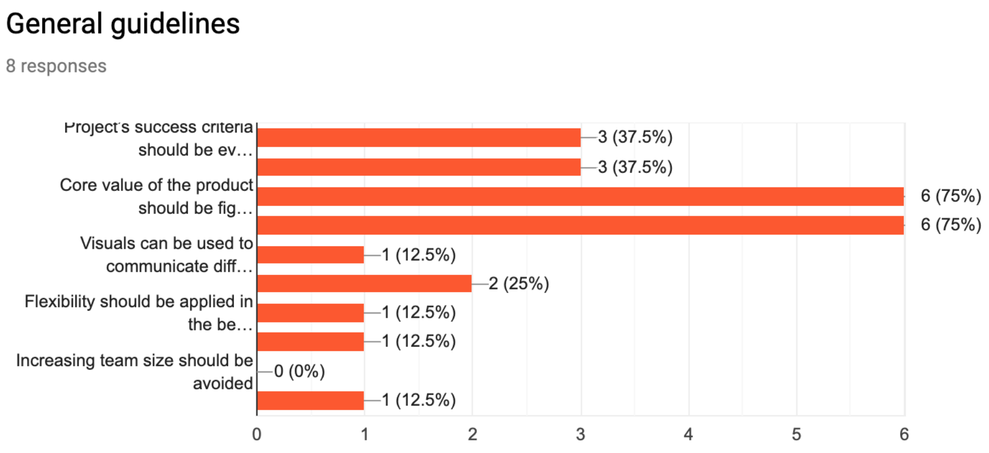


Figure 6.12: Results for procurement and development guidelines (LinkedIn)

understand this. Guideline 4 is relevant to this concept as it is about developing highest value features first. It received equal amount of votes on the red form and a number two spot on the blue form. It would have been an easy pick for my personal list as well, but I find the other two guidelines more important.

The next two highest voted guidelines were Guideline 1 about continuously re-evaluating the success criteria and Guideline 2 about presenting complex concepts with examples. These two guidelines shared the number two spot on the red form and had 9 and 8 votes on the blue form respectively. Both of these guidelines are easy choices.

Guideline 6 about prioritizing face-to-face meetings received the third most votes on the blue form with 13 votes, but only two votes on the red form. This may be because of our organization's advocacy of effective customer communication.

Surprisingly, the number of votes on Guideline 9 about avoiding increasing team sizes was the least of all. It had 4 votes on the blue form and 0 votes on the red form. I believe this may again be because of a lack of context. When the guideline is phrased ”increasing team size should be avoided”, it sounds like team sizes should be arbitrarily restricted. However, there are real situations when team sizes should and even must be increased. This may have been a mistake on my part, perhaps the guideline could have been better phrased as ”increasing team size in attempt to enhance team performance should be avoided”.

The rest of the guidelines were quite in the middle of the distribution and did not stand out as such. This is understandable, as most of them are not as important as the ones that peaked in both forms. However, I am a little surprised that my choice of Guideline 7 about flexibility in the beginning negotiations also lies in the middle of the distribution for others. An argument can be made that it is not as important than some of the others but I personally think it is still important for beginning a good relationship with the customer and a successful project. It is all about teamwork after all.

6.5.4 Feedback

The questionnaire received some open feedback from a couple respondents in the final section (Figure 6.5). One user gave feedback in the blue form that they would have liked to choose more than three guidelines in the general guidelines section. All of the guidelines are really useful, so a choice between them is difficult to make. I believe it was the right choice to have only three options in the final section, as it makes the top choices stand out more clearly. Perhaps the distribution would have been smoother, had there been more votes for this section.

Another user noted that the definition of ”Ready” should have also been mentioned alongside definition of ”Done”. This was an oversight on my part, the definition of ”Ready” should have been mentioned at the same time as definition of ”Done” in Section 3.1. Definition of ”Ready” of course means the commonly agreed definition within the Scrum Team, when a Backlog item is ready for implementation.

One of the employees in my organization sent a longer free-form feedback. This employee raised some questions about the vagueness of some of the guidelines and their wording. According to the feedback, the Product Owner’s does not need to understand the technical skills of the Scrum Team, but the role is still very important. Also, the importance of Scrum Master is questioned as the role is not significant alone. This feedback does agree with many of the customer communication guidelines raised in the general section, however, prioritizing face-to-face meetings, core values and flexibility in negotiations. These arguments of vagueness confirm my earlier suspicions found when analyzing the form results. This can be easily fixed by improving the wording of the guidelines and providing more context.

The red form received only one open feedback: *"The questionnaire didn't include any statements of demos for stakeholders or team retrospectives that I personally find the most valuable occasions to learn whether the project is on the right track or not. Furthermore, those events are great for identifying and thus preventing waste."* The guidelines could have also included development demos or team retrospectives as they are important for communication. This would a good area to improve this thesis in and conduct more research.

7 Conclusion

As the world gets ever more digitalized, there will be an ever-increasing demand for software. As the demand increases, the problems in development practices and especially communication will become amplified. Software is relatively speaking a young industry and iterative development an even younger methodology. It is only natural that the special development practices of software are not intuitive for newcomers in the industry. The openness and emphasis on communication in the software industry is not common in many other industries with stronger hierarchical structures. The flat organizational structure of software organizations require effective communication strategies because the information flows in many directions.

To communicate effectively about software procurement and development processes, it is important to know the basic information well. This is the reason why this thesis had a high emphasis on background information. In Chapter 2 we covered a brief history of software process models and how they have evolved over time. In Chapter 3 we were introduced to two modern software process models, and certain aspects that are important for customer communication were highlighted as guidelines. In Chapter 4 we covered the basics of a software procurement process and various possible contracts within this process, and we introduced the actors that commonly participate in the procurement.

In Chapter 5 we analyzed a buyer's guide made to solve the problem that exists between software procurement and software development: ignorance and the imbalance of risk. From this guide, more guidelines to customer communication were raised. In Chapter 6 the previously gathered guidelines were reintroduced and I highlighted the ones I felt were the most important. My choices were validated by presenting them to my professional network with a questionnaire. Differences in the answers were noted, and some of them were identified to be potentially because of flaws in the validation method. The validation results did, however, present interesting insight into what aspects developers think are valuable for a successful software project.

Some more research could be done to improve the results of this thesis. Instead of a questionnaire, rounds of interviews could be held to get deeper insight into what communication guidelines are important to different developers for a successful software project. Also, the role of the procurement process in customer communication could be explored further. This thesis was mostly from the perspective of development. To get answers for the procurement process, interviews could be held with sales personnel, who have a better perspective in that side of the equation. Interviews with customers would also be interesting, as the thesis subject was examined primarily from the supplier's side.

Emphasis in this thesis has been at focusing on the quantity of communication and its placement in the procurement and development processes. However, nothing has been mentioned about the quality of communication which is also very important. This is a different topic altogether and deserves an equally in-depth analysis. For simplicity's sake this thesis has made the assumption that all communication is of high-quality. In the real world this is not a reasonable assumption. Everybody is not equally skilled in communication and there can even be too much communication in which case it can become noise or even disruptive noise. Communication skills improve in continued use, but they should be deliberately practiced nonetheless.

References

[1] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, May 1988.

[2] Wikipedia: Software development process. `https://en.wikipedia.org/wiki/Software_development_process`. Accessed: 2018-10-03.

[3] Wikipedia: Waterfall model. `https://en.wikipedia.org/wiki/Waterfall_model`. Accessed: 2018-10-01.

[4] Winston W. Royce. Managing the development of large software systems. In *Proceedings, IEEE WESCON*, pages 1–9. The Institute Of Electrical and Electronics Engineers, Inc., TRW, August 1970.

[5] Kai Petersen, Claes Wohlin, and Dejan Baca. The waterfall model in large-scale development. In *PROFES*, page 386–400. Blekinge Institute of Technology, Springer, 2009.

[6] Phillip A. Laplante and Colin J. Neill. The demise of the waterfall model is imminent and other urban myths. *Queue*, 2004.

[7] Craig Larman and Victor R. Basili. Iterative and incremental development: A brief history. *Computer*, 36(6):47–56, June 2003.

[8] Hirotaka Takeuchi and Ikujiro Nonaka. The new new product development game. *Harvard Business Review*, 1986.

[9] Manifesto for agile software development. https://www.agilealliance.org/agile101/the-agile-manifesto/. Accessed: 2018-12-03.

[10] The official scrum guide. https://scrumguides.org/scrum-guide.html. Accessed: 2018-12-03.

[11] Philipp Diebold, Jan-Peter Ostberg, Stefan Wagner, and Ulrich Zendler. What do practitioners vary in using scrum? In *International Conference on Agile Software Development*, pages 40–51. Springer, 2015.

[12] Mary Poppendieck and Tom Poppendieck. *Lean Software Development: An Agile Toolkit*. Addison-Wesley, 1st edition, 2003.

[13] Jeff Keyes. Software delivery value stream management. https://www.plutora.com/wp-content/uploads/2018/08/software-delivery-process-value-stream-management-diagram.jpg, October 2018. Accessed: 2019-01-16.

[14] Jeff Lasovski. A scrum board suggesting to use kanban. https://commons.wikimedia.org/wiki/File:Simple-kanban-board-.jpg, December 2011. Accessed: 2019-01-16.

[15] Kim B Clark, Takahiro Fujimoto, and Andrew Cook. *Product development performance: Strategy, organization, and management in the world auto industry*. Harvard Business School Press Boston, MA, 1991.

[16] Uusi hankintalaki: Verkkopalvelun hankinta julkishallinnossa. https://northpatrol.fi/verkkopalvelun-hankinta-julkishallinnossa. Accessed: 2019-03-06.

[17] Wikipedia: Request for information. https://en.wikipedia.org/wiki/Request_for_information. Accessed: 2019-03-07.

[18] Wikipedia: Fixed-price contract. `https://en.wikipedia.org/wiki/Fixed-price_contract`. Accessed: 2019-03-24.

[19] Wikipedia: Time and materials. `https://en.wikipedia.org/wiki/Time_and_materials`. Accessed: 2019-03-24.

[20] Hankinnat: Puitejärjestelyt. `https://www.hankinnat.fi/eu-hankinta/menettelytekniikat/puitejarjestelyt`. Accessed: 2019-03-24.

[21] Why software fails. `https://spectrum.ieee.org/computing/software/why-software-fails/3`. Accessed: 2019-05-26.

[22] Robert M Monczka, Robert J Trent, and Robert B Handfield. *Purchasing and supply chain management*. South-Western, 2005.

[23] Kristopher Buczek. Aligning sourcing activities with corporate strategy. Master's thesis, Aalto University School of Business, 2014.

[24] Thomas Ph Herz, Florian Hamel, Falk Uebernickel, and Walter Brenner. It governance mechanisms in multisourcing–a business group perspective. In *2012 45th Hawaii International Conference on System Sciences*, pages 5033–5042. IEEE, 2012.

[25] Wikipedia: Multisourcing. `https://en.wikipedia.org/wiki/Multisourcing`. Accessed: 2019-05-26.

[26] Jarkko Järvenpää and Pasi Kovanen. *Ohjelmistokehityksen ostajan pikaopas 2.0*. Vincit, 2018.