Agility and Requirements Engineering

- Impact of Agile Processes in Requirements Engineering
- Current Agile Practices
- Variance
- Overview of Requirements Engineering Using Agile
- Managing Unstable Requirements
- Requirements Elicitation
- Agile Requirements Abstraction Model
- Requirements Management in Agile Environment
- Agile Requirements Prioritization
- Agile Requirements Modeling and Generation
- Concurrency in Agile Requirements Generation

Impact of Agile Process in Requirements Engineering

- Requirements engineering provides the mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a solution, specifying the solution, validating the specification, and managing the requirements.
- The goal of requirements engineering is to ensure that the system have been specified to meet the customer's needs and to satisfy the customer's expectations.
- The overall process for traditional requirements engineering includes four high-level sub processes.

Requirements Engineering

- These are concerned with evaluating the business usefulness of the system (feasibility study), discovering requirements (elicitation and analysis), converting requirements into some standard form (specification), and checking that the requirements define the system as customer wants (validation).
- In addition, requirements must be managed because they will change throughout the life of the system.

Agile requirements engineering practices

3.2.1 Interaction between the development team and the customer

To have a customer accessible or on-site is a key point in all agile approaches.

Paetsch et al. (2003) points also out that the customer is

involved under whole the development time but it is not guaranteed that every user or customer from necessary backgrounds are present.

In agile software development, it is highly recommended to have no communication layers between the development team and the customer.

When the development team communicates directly with the customer, the probability of misunderstandings between parties reduces significantly.

Agile requirements engineering practices

The aim of agile requirements engineering is to effectively translate ideas from the customer to the software development team, rather than create extensive documentation.

When using informal communication between the customer and the development team, the time-consuming documentation and approval processes are not needed anymore.

These things are perceived as unnecessary when the requirements are evolving.

Most organizations use simple techniques such as user stories to define requirements from customer.

When collecting the requirements from customer the whole development team should be involved and the common language of customer should be used.

Agile requirements engineering practices

This will reduce the probability of misunderstandings. In addition, if the requirements are too complex, the customer is asked to split them in the simpler ones.

On-site customer representation is difficult in most cases. If there is no possibility to an intensive interaction between customer and developers, this approach may result inadequately developed or totally wrong requirements.

The customer may have more than one group involved to the project and each of them having interest in different aspects of the system.

In this kind of case, it is hard to achieve a consensus between customer groups and then the project team must spend extra effort to integrate the requirements.

The person who represents all the stakeholders should be a domain expert and be able to make important decisions such as prioritize requirements.

3.2.2 Iterative requirements engineering

In the agile methods, functionalities are released in small and frequent cycles.

This allows the development team to get more and real-time feedback from customer.

At the beginning of the project the development team acquires a high-level understanding of the software's critical features.

Reasons for not to set too much effort on requirements engineering at the beginning include high requirements volatility, incomplete knowledge of technology used or customers who cannot clearly describe the requirements before they see them.

After the brief requirements identifying at the beginning, agile requirements engineering continues at each development cycle.

At the start of each cycle, the customer and development team meets and they discuss about the features that must be implemented.

3.2.2 Iterative requirements engineering

By that way the requirements are detailed gradually and iteratively as the development progress.

Gradual detailing ensures that requirements are actively worked with throughout the development process, which reduces the challenge of communication gaps within development as well as between business and development team.

An iterative approach for requirements engineering also makes it easier to keep system requirements specification up to date and creates more satisfactory relationship with the customer.

When the relationship with the customer is good, the customer will provide feedback for the development team.

3.2.2 Iterative requirements engineering

Iterative requirements engineering enables getting frequent feedback from customer. This helps reducing the waste in requirements such as unnecessary features.

According to the Cao & Ramesh (2008) the iterative requirements engineering can also be used in the stable environments where the changes in the requirements come from unforeseen technical issues.

Iterative requirements engineering faces at least two major challenges. Cost and schedule estimation for the project are difficult to do at the beginning because the project scope is subject to constant change.

3.2.2 Iterative requirements engineering

Obtaining management support for such projects can be challenging. (Cao & Ramesh 2008) Second, the lack of clear requirements picture at the beginning of the development will result in significant changes in requirements during the development. However, this approach will easier lead to a more feasible scope. (Bjarnason et al. 2011)

3.2.3 Requirements prioritization

In the agile development, the highest-priority features are implemented early so that the customers can realize the most business value.

The prioritization should be repeated frequently during the whole development process because the understanding of the project increases and new requirements are added during development.

Cao & Ramesh (2008) and Sillitti & Succi (2005) recommends that the requirements will be prioritized at the beginning of each development cycle.

The customer and the development team assign priorities for each feature to be secure that those requirements that are most important will be implemented first (Sillitti & Succi 2005).

Requirements Prioritization

Cao & Ramesh (2008) says also that the requirements prioritization is based only on one factor, the business value for customer.

Sillitti & Succi (2005) presents a four-step process for requirements prioritization:

1. The development team estimates the time required to implement the functionality.
2. The customer sets business priorities for each functionality.
3. The development team assigns a risk factor for each functionality according to the business priorities.
4. The features to be implemented in the iteration are identified by the development team and the customer.

Requirements Prioritization

When repeating the requirements elicitation and these four steps of prioritization at the beginning of each iteration, it is possible to identify the requirements that will not provide enough value for customer.

When customers are involved in development process, they can provide business reasons for each requirement at any development cycle.

By this way, the development team will achieve a clear understanding of the customer’s priorities and they can better meet the customer needs.

Prioritizing the requirements also reduces the possibility of including waste in requirements. This means such requirements that will not provide a real benefit for customer's business.

Requirements Prioritization

However, if business value is the only or primary criterion for requirements prioritization, it might result major problems as non-scalable architecture.

Also, it might result a system that cannot accommodate requirements that appear secondary to customer but that become critical for operational success.

Cao & Ramesh (2008) remind also that the continuous reprioritization may lead to instability if it is not practiced with care.

3.2.4 Non-functional requirements

Non-functional requirements can be understood as the constraints under which the entire system must operate.

In other words, the non-functional requirements do not tell what the software will do but how the software will do it. (Adams 2015) Non-functional requirements are, for example, scalability, maintainability, portability, safety, or performance (Cao & Ramesh 2008).

In the agile approaches, the non-functional requirements are often neglected. Customers often focus more on the core functionality.

The customers are telling what they want the system to do and, therefore, they do not normally think about non-functional requirements (Paetsch et al. 2003).

One common exception is, however, that customers focus on some requirements of the ease of use, interface, and safety (Paetsch et al. 2003, Cao & Ramesh 2008).

3.2.4 Non-functional requirements

Because of that low impact for non-functional requirements from the customer's side, the development team should guide the customer so that such hidden needs can be identified.

De Lucia & Qusef (2010) propose also that the customers and agile team leaders would arrange for meetings to discuss non-functional requirements even in the earliest stages.

Sillitti & Succi (2005) points also out that the need of specifying non-functional requirements is not so important in the context of agile software development than in the other context because of the continuous interaction with the customer.

3.2.4 Non-functional requirements

Customer can test software after each iteration and if he identifies some problems regarding non-functional requirements, the development team can adapt the system to meet such requirements in the following iteration.
(Sillitti & Succi 2005) It is important for the development team to know most of the non-functional requirements because they can affect the choice of the architecture (Paetsch et al. 2003).
Considering non-functional requirements by this way may have a big risk because there is a lack of specific techniques to manage them (Sillitti & Succi 2005).

3.3 Documentation in agile requirements engineering

In the agile practices the documentation is minimalistic and the requirements are not always documented.
However, some agile methods recommend to use requirements document but its extend depends on the development teams' decision.
Also, the team size should be considered when planning the documentation.
The agile team is considered to be more cost-effective and productive when over documenting is avoided.
The minimal documentation also increases the chances to keep the document up to date when the software is changed.

3.3 Documentation in agile requirements engineering

Customers often ask the team to produce documentation before the team is resolved but the scope of this documentation is very limited and concentrated on the core aspects of the system.

Although the modeling is used as a part of the agile requirements engineering, most of the models will not become part of the persistent documentation of the system.

3.3.1 User stories

A user story describes functionality that a user or a customer of the software or system keeps valuable. There are three aspects that the user stories are composed of:

- A written description of the story
- Conversations about the story
- Tests that convey and document details

While the written description may contain the text of the story, the detailed information is worked out in the conversations and is documented in the tests.

The written description can concern a small piece of functionality and be written on a piece of paper. These papers are then hanged on a pin board.

3.3.1 User stories

One user story does not contain very many details and, therefore, a basic question is that where the details are.
Cohn answers this question and says that many of the requirement details can be expressed as additional stories.
It is better to have more stories than have some stories that are too large.

3.3.2 Challenges of minimal documentation

In an overall look, agile methods tend to err on the side of producing not enough documentation.

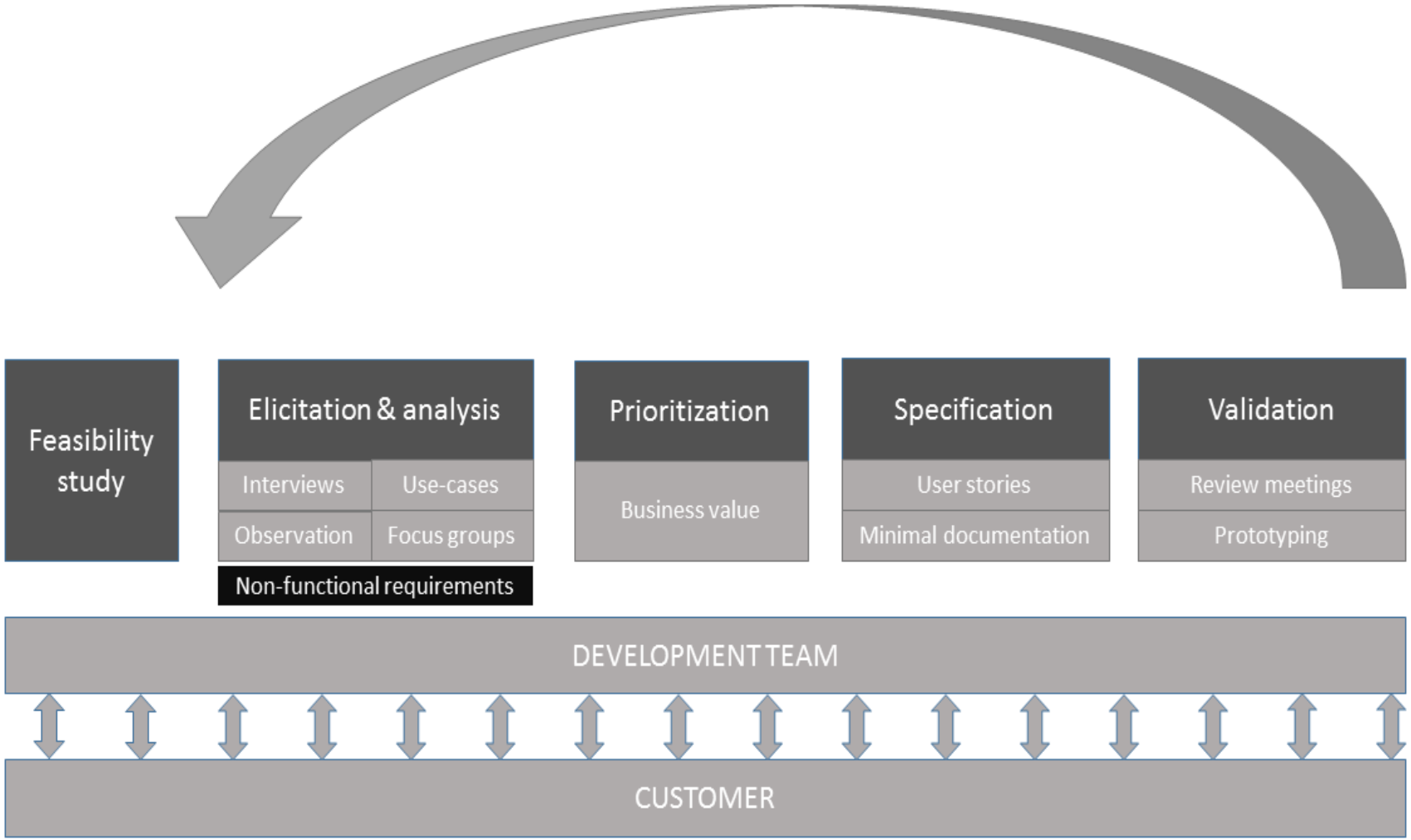
The lack of documentation might cause problems in agile teams because the documentation is used to share information between people.

A communication breakdown can occur owing to personnel turnover, rapid changes to requirements, unavailability of the right customer representative, or the application's growing complexity.

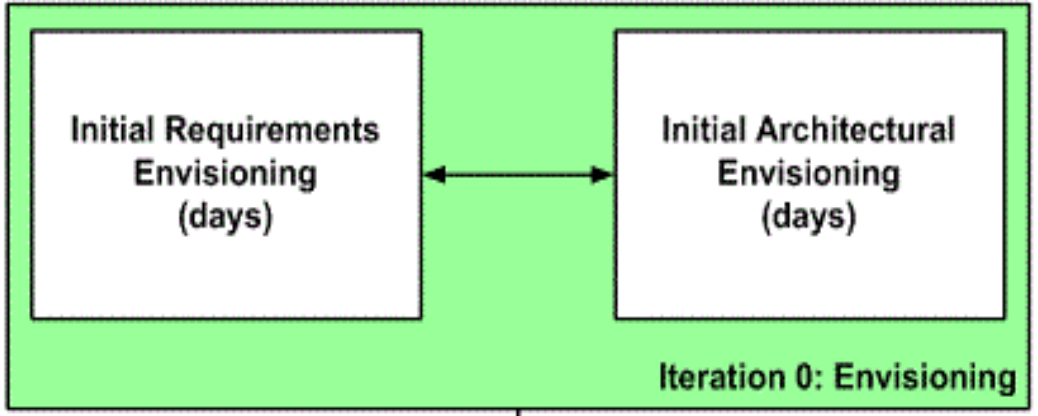
In the case of communication breakdown, there can be many problems. Problems are for example inability to scale the software, evolve software over the time or take new members into the development team.

A new team member will have many questions regarding the project and if he or she needs to ask other members whole the time, it will slow down the work.

A model for agile requirements engineering



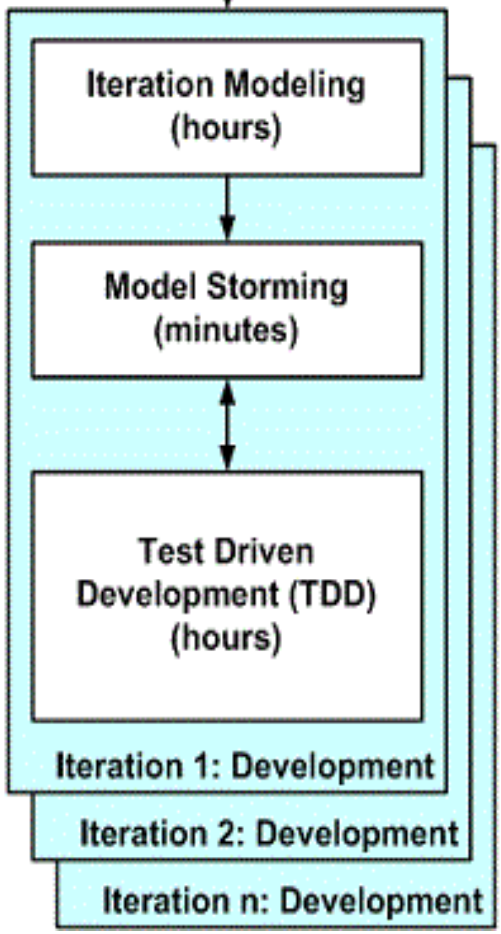
- Identify the high-level scope
- Identify initial "requirements stack"
- Identify an architectural vision



- Modeling is part of iteration planning effort
- Need to model enough to give good estimates
- Need to plan the work for the iteration

- Work through specific issues on a JIT manner
- Stakeholders actively participate
- Requirements evolve throughout project
- Model just enough for now, you can always come back later

- Develop working software via a test-first approach
- Details captured in the form of executable specifications



Reviews (optional)

All Iterations (hours)

1. Agile Requirements Modeling in a Nutshell

Figure 1 depicts the Agile Model Driven Development (AMDD) lifecycle, which depicts how Agile Modeling (AM) is applied by agile software development teams.

The critical aspects which we're concerned about right now are initial requirements modeling, iteration modeling, model storming, and acceptance test-driven development (ATDD).

The fundamental idea is that you do just barely enough modeling at the beginning of the project to understand the requirements for your system at a high level, then you gather the details as you need to on a just-in-time (JIT) basis.

1.1 Initial Requirements Modeling

At the beginning of a project you need to take several days to envision the high-level requirements and to understand the scope of the release (what you think the system should do). Your goal is to get a gut feel for what the project is all about, not to document in detail what you think the system should do: the documentation can come later, if you actually need it. For your initial requirements model is that you need some form of:

1. **Usage model**. As the name implies usage models enable you to explore how users will work with your system. This may be a collection of essential use cases on a Unified Process (UP) project, a collection of features for a Feature Driven Development (FDD) project, or a collection of user stories for an Extreme Programming (XP) project, or any of the above on a disciplined agile team.
2. **Initial domain model**. A domain model identifies fundamental business entity types and the relationships between then. Domain models may be depicted as a collection of Class Responsibility Collaborator (CRC) cards, a slim UML class diagram, or even a slim data model. This domain model will contain just enough information: the main domain entities, their major attributes, and the relationships between these entities. Your model doesn't need to be complete, it just needs to cover enough information to make you comfortable with the primary domain concepts.

3.User interface model. For user interface intensive projects you should consider developing some screen sketches or even a user interface prototype.

What level of detail do you actually need? My experience is that you need requirements artifacts which are just barely good enough to give you this understanding and no more. For example, Figure 2 depicts a simple point-form use case. This use case could very well have been written on an index card, a piece of flip chart paper, or on a whiteboard. It contains just enough information for you to understand what the use case does, and in fact it may contain far too much information for this point in the lifecycle because just the name of the use case might be enough for your stakeholders to understand the fundamentals of what you mean. Figure 3, on the other hand, depicts a fully documented formal use case. This is a wonderful example of a well documented use case, but it goes into far more detail than you possibly need right now. If you actually need this level of detail, and in practice you rarely do, you can capture it when you actually need to by model storming it at the time. The longer your project team goes without the concrete feedback of working software, the greater the danger that you're modeling things that don't reflect what your stakeholders truly need.

Figure 2. A point-form use case.

Name: Enroll in Seminar
Basic Course of Action:

- Student inputs her name and student number
- System verifies the student is eligible to enroll in seminars. If not eligible then the student is informed and use case ends.
- System displays list of available seminars.
- Student chooses a seminar or decides not to enroll at all.
- System validates the student is eligible to enroll in the chosen seminar. If not eligible, the student is asked to choose another.
- System validates the seminar fits into the student's schedule.
- System calculates and displays fees
- Student verifies the cost and either indicates she wants to enroll or not.
- System enrolls the student in the seminar and bills them for it.
- The system prints enrollment receipt.

Figure 3. A detailed use case.

Name: Enroll in Seminar
Identifier: UC 17
Description:
Enroll an existing student in a seminar for which she is eligible.
Preconditions:
The Student is registered at the University.
Postconditions:
The Student will be enrolled in the course she wants if she is eligible and room is available.
Basic Course of Action:

1. The use case begins when a student wants to enroll in a seminar.
2. The student inputs her name and student number into the system via *UI23 Security Login Screen*.
3. The system verifies the student is eligible to enroll in seminars at the university according to business rule *BR129 Determine Eligibility to Enroll*. [Alt Course A]
4. The system displays *UI32 Seminar Selection Screen*, which indicates the list of available seminars.
5. The student indicates the seminar in which she wants to enroll. [Alt Course B: The Student Decides Not to Enroll]
6. The system validates the student is eligible to enroll in the seminar according to the business rule *BR130 Determine Student Eligibility to Enroll in a Seminar*. [Alt Course C]

7. The system validates the seminar fits into the existing schedule of the student according to the business rule *BR143 Validate Student Seminar Schedule*.
8. The system calculates the fees for the seminar based on the fee published in the course catalog, applicable student fees, and applicable taxes. Apply business rules *BR 180 Calculate Student Fees* and *BR45 Calculate Taxes for Seminar*.
9. The system displays the fees via *UI33 Display Seminar Fees Screen*.
10. The system asks the student if she still wants to enroll in the seminar.
11. The student indicates she wants to enroll in the seminar.
12. The system enrolls the student in the seminar.
13. The system informs the student the enrollment was successful via *UI88 Seminar Enrollment Summary Screen*.
14. The system bills the student for the seminar, according to business rule *BR100 Bill Student for Seminar*.
15. The system asks the student if she wants a printed statement of the enrollment.
16. The student indicates she wants a printed statement.
17. The system prints the enrollment statement *UI89 Enrollment Summary Report*.
18. The use case ends when the student takes the printed statement

Alternate Course A: The Student is Not Eligible to Enroll in Seminars.

A.3. The registrar determines the student is not eligible to enroll in seminars.

A.4. The registrar informs the student he is not eligible to enroll.

A.5. The use case ends.

Alternate Course B: The Student Decides Not to Enroll in an Available Seminar

B.5. The student views the list of seminars and does not see one in which he wants to enroll.

B.6. The use case ends.

Alternate Course C: The Student Does Not Have the Prerequisites

C.6. The registrar determines the student is not eligible to enroll in the seminar he chose.

C.7. The registrar informs the student he does not have the prerequisites.

C.8. The registrar informs the student of the prerequisites he needs.

C.9. The use case continues at Step 4 in the basic course of action.

1.2 Iteration Modeling

Iteration modeling is part of overall iteration planning efforts performed at the beginning of an iteration. You often have to explore requirements to a slightly more detailed level than you did initially, modeling just enough so as to plan the work required to fulfill a given requirement.

1.3 Model Storming

Detailed requirements are elicited, or perhaps a better way to think of it is that the high-level requirements are analyzed, on a just in time basis.

When a developer has a new requirement to implement, perhaps the "Enroll in Seminar" use case of Figure 2, they ask themselves if they understand what is being asked for. In this case, it's not so clear exactly what the stakeholders want, for example we don't have any indication as to what the screens should look like. They also ask themselves if the requirement is small enough to implement in less than a day or two, and if not then the reorganize it into a collection of smaller parts which they tackle one at a time. Smaller things are easier to implement than larger things.

To discover the details behind the requirement, the developer (or developers on project teams which take a pair programming approach), ask their stakeholder(s) to explain what they mean.
This is often done by sketching on paper or a whiteboard with the stakeholders.
These "model storming sessions" are typically impromptu events and typically last for five to ten minutes (it's rare to model storm for more than thirty minutes because the requirements chunks are so small). The people get together, gather around a shared modeling tool (e.g. the whiteboard), explore the issue until their satisfied that they understand it, and then they continue on (often coding). Extreme programmers (XPers) would call requirements modeling storming sessions "customer Q&A sessions".
In the example of identifying what a screen would look like, together with your stakeholder(s) developer sketch what the want the screen to look like, drawing several examples until you come to a common understanding of what needs to be built. Sketches such as this are inclusive models because you're using simple tools and modeling techniques, this enabling the Agile Modeling (AM) practice of Agile Stakeholder Participation. The best people to model requirements are stakeholders because they're the ones who are the domain experts, not you.

1.4 Acceptance Test Driven Development (ATDD)

Test-driven development (TDD) (Beck 2003; Astels 2003), is an evolutionary approach to development which that requires significant discipline and skill (and good tooling).

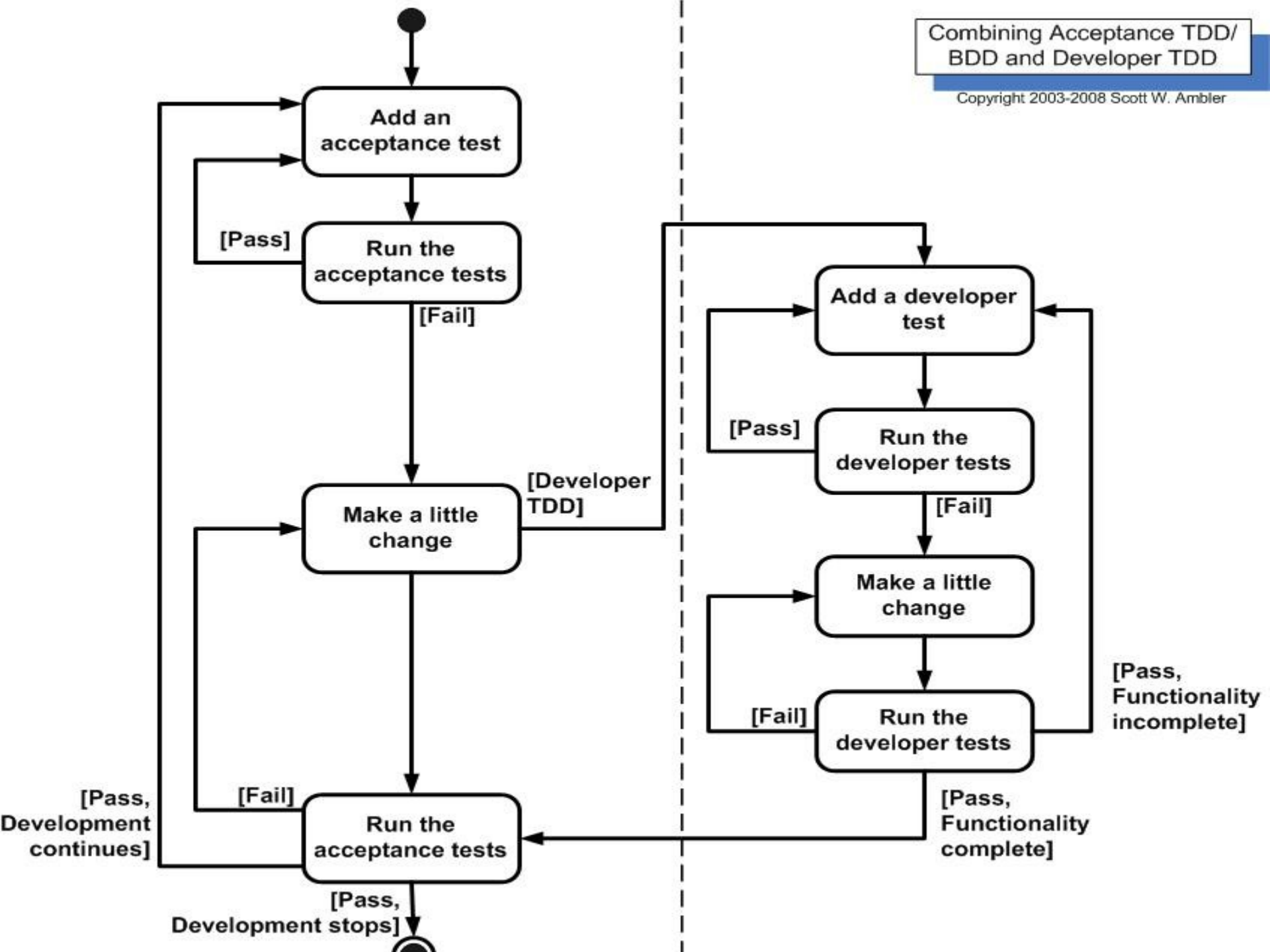
The first step is to quickly add a test, basically just enough code to fail. Next you run your tests, often the complete test suite although for sake of speed you may decide to run only a subset, to ensure that the new test does in fact fail. You then update your functional code to make it pass the new tests. The fourth step is to run your tests again. If they fail you need to update your functional code and retest. Once the tests pass the next step is to start over (you may first need to refactor any duplication out of your design as needed). As Figure 4 depicts, there are two levels of TDD:

Acceptance TDD (ATDD). With ATDD you write a single acceptance test, or behavioral specification depending on your preferred terminology, and then just enough production functionality/code to fulfill that test. The goal of ATDD is to specify detailed, executable requirements for your solution on a just in time (JIT) basis. ATDD is also called Behavior Driven Development (BDD).

Developer TDD. With developer TDD you write a single developer test, sometimes inaccurately referred to as an unit test, and then just enough production code to fulfill that test. The goal of developer TDD is to specify a detailed, executable design for your solution on a JIT basis. Developer TDD is often simply called TDD.

Figure 4. How ATDD and developer TDD fit together.

Combining Acceptance TDD/ BDD and Developer TDD



With ATDD you are not required to also take a developer TDD approach to implementing the production code although the vast majority of teams doing ATDD also do developer TDD.

As you see in Figure 4, when you combine ATDD and developer TDD the creation of a single acceptance test in turn requires you to iterate several times through the write a test, write production code, get it working cycle at the developer TDD level. Clearly to make TDD work you need to have one or more testing frameworks available to you. For acceptance TDD people will use tools such as Fitnesse or RSpec and for developer TDD agile software developers often use the xUnit family of open source tools, such as JUnit or VBUnit. Commercial testing tools are also viable options. Without such tools TDD is virtually impossible.

There are several important benefits of ATDD.

First, the tests not only validate your work at a confirmatory level they are also in effect executable specifications that are written in a just-in-time (JIT) manner.

By changing the order in which you work your tests in effect do double duty.

Second, traceability from detailed requirements to test is automatic because the acceptance tests are your detailed requirements (thereby reducing your traceability maintenance efforts if you need to do such a thing).

Third, this works for all types of requirements -- although much of the discussion about ATDD in the agile community focuses on writing tests for user stories, the fact is that this works for use cases, usage scenarios, business rules, and many other types of requirements modeling artifacts.

2. Where Do Requirements Come From?

Your project stakeholders - direct or indirect users, managers, senior managers, operations staff members, support (help desk) staff members, testers, developers working on other systems that integrate or interact with your, and maintenance professionals - are the only OFFICIAL source of requirements (yes, developers can SUGGEST requirements, but stakeholders need to adopt the suggestions).

In fact it is the responsibility of project stakeholders to provide, clarify, specify, and prioritize requirements. Furthermore, it is the right of project stakeholders that developers invest the time to identify and understand those requirements.

This is concept is critical to your success as an agile modeler - it is the role of project stakeholders to provide requirements, it is the role of developers to understand and implement them.

To identify **potential requirements** you may also, often with the aid of **your project stakeholders, work through existing documents such as corporate policy manuals, existing legacy systems, or publicly available resources such as information on the web, books, magazine articles, or the products and services of your competitors.** Once again, it is your project stakeholders that are the ultimate source of requirements, it is their decision and not yours. I cannot be more emphatic about this.

There are several "best practices" which should help you to become more agile with your requirements modeling efforts:

- Stakeholders actively participate
- Adopt inclusive models
- Take a breadth-first approach
- Model storm details just in time (JIT)
- Prefer executable specifications over static documentation
- Your goal is to implement requirements, not document them
- Create platform independent requirements to a point
- Smaller is better
- Question traceability
- Explain the techniques
- Adopt stakeholder terminology
- Keep it fun
- Obtain management support
- Turn stakeholders into developers
- Treat requirements like a prioritized stack

4. Types of Requirements

I'm a firm believer in separating requirements into two categories: Behavioral. A behavioral requirement describes how a user will interact with a system (user interface issues), how someone will use a system (usage), or how a system fulfills a business function (business rules). These are often referred to as functional requirements. Non-behavioral. A non-behavioral requirement describes a technical feature of a system, features typically pertaining to **availability, security, performance, interoperability, dependability, and reliability.** Non-behavioral requirements are often referred to as "non-functional" requirements due to a bad naming decision made by the IEEE (as far as I'm concerned non-functional implies that it doesn't work).

It's important to understand that the distinction between behavioral and non-behavioral requirements is fuzzy - a performance requirement describing the expected speed of data access is clearly technical in nature but will also be reflected in the response time of the user interface which affects usability and potential usage.

Access control issues, such as who is allowed to access particular information, is clearly a behavioral requirement although they are generally considered to be a security issue which falls into the non-behavioral category.

Loosen up a bit and don't allow yourself to get hung up on issues such as this. The critical thing is to identify and understand a given requirement, if you mis-categorize the requirement who really cares?

5. Potential Requirements Artifacts

Because there are several different types of requirements, some or all of which may be applicable to your project, and because each modeling artifact has it's strengths and weaknesses, you will want to have several requirements modeling artifacts in your intellectual toolkit to be effective.

Table 1 summarizes common artifacts for modeling requirements, artifacts that are described in greater detail in the article Artifacts for Agile Modeling.

The type(s) of requirement that the artifact is typically used to model is indicated as well as a potential "simple" tool that you can use to create the artifact (the importance of using simple tools was discussed earlier in the section Some Philosophy).

Table 1. Candidate artifacts for modeling requirements.

Artifact	Type	Simple Tool	Description
Acceptance Test	Either	FITNesse	Describes an observable feature of a system which is of interest to one or more project stakeholders
Business Rules	Behavioral	Index cards	A business rule is an operating principle or policy that your software must satisfy.
CRC Card	Either, Usually Behavioral	Index cards	A Class Responsibility Collaborator (CRC) model is a collection of standard index cards, each of which have been divided into three sections, indicating the name of the class, the responsibilities of the class, and the collaborators of the class. A class represents a collection of similar objects, a responsibility is something that a class knows or does, and a collaborator is another class that a class interacts with to fulfill its responsibilities.

Table 1. Candidate artifacts for modeling requirements

Change Case	Either	Index card	Change cases are used to describe new potential requirements for a system or modifications to existing requirements.
Constraints	Either	Index card	A constraint is a restriction on the degree of freedom you have in providing a solution. Constraints are effectively global requirements for your project.
Data Flow Diagram	Behavioral	Whiteboard drawing	A data-flow diagram (DFD) shows the movement of data within a system between processes, entities, and data stores. When modeling requirements a DFD can be used to model the context of your system, indicating the major external entities that your system interacts with.
UI Prototype	Either	Post It notes and flip chart paper	An essential user interface (UI) prototype is a low-fidelity model, or prototype, of the UI for your system - it represents the general ideas behind the UI but not the exact

Table 1. Candidate artifacts for modeling requirements

Essential Use Case	Behavioral	Paper	A use case is a sequence of actions that provide a measurable value to an actor. An essential use-case is a simplified, abstract, generalized use case that captures the intentions of a user in a technology and implementation independent manner.
Feature	Either, Usually Behavioral	Index card	A feature is a "small, useful result in the eyes of the client". A feature is a tiny building block for planning, reporting, and tracking. It's understandable, measurable, and do-able (along with several other features) within a two-week increment .
Technical Requirements	Non-Behavioral	Index card	A technical requirement pertains to a non-functional aspect of your system, such as a performance-related issue, a reliability issue, or technical environment issue.

Usage Scenario	Behavioral	Index card	A usage scenario describes a single path of logic through one or more use cases or user stories. A use-case scenario could represent the basic course of action, the happy path, through a single use case, a combination of portions of the happy path replaced by the steps of one or more alternate paths through a single use case, or a path spanning several use cases or user stories.
Use Case Diagram	Behavioral	Whiteboard sketch	The use-case diagram depicts a collection of use cases, actors, their associations, and optionally a system boundary box. When modeling requirements a use case diagram can be used to model the context of your system, indicating the major external entities that your system interacts with.
User Story	Either	Index card	A user story is a reminder to have a conversation with your project stakeholders. User stories capture high-level requirements, including behavioral requirements, business rules, constraints, and technical requirements

Requirement Elicitation Techniques

Technique	Description	Strength(s)	Weakness(es)	Staying Agile
Active Stack Holder Participation	Extends On-Site Customer to also have stakeholders (customers) actively involved with the modeling of their requirements.	• Highly collaborative technique • The people with the domain knowledge define the requirements • Information is provided to the team in a timely manner • Decisions are made in a timely manner	• Many stakeholders need to learn modeling skills • Stakeholders often aren't available 100% of the time • Airs your dirty laundry to stakeholders	• It doesn't get more agile than this
Electronic Interviews	You interview a person over the phone, through video conferencing, or via email.	• Supports environments with dispersed stakeholders • Provides a permanent record of the conversation	• Restricted interaction technique • Limited information can be conveyed electronically • Risky when it is your only means of communication	• Ideally used to support other techniques, not as your primary means of elicitation • Face-to-face interviews should be preferred over electronic ones

Requirement Elicitation Techniques

Technique	Description	Strength(s)	Weakness(es)	Staying Agile
Face to Face Interviews	You meet with someone to discuss their requirements. Although interviews are sometimes impromptu events, it is more common to schedule a specific time and place to meet and to provide at least an informal agenda to the interviewee. It is also common to provide a copy of your interview notes to the interviewee, along with some follow up questions, for their review afterward. One danger of interviews is that you'll be told how the person ideally wants to work, not how they actually work. You should temper interviews with actual observation	• Collaborative technique • You can elicit a lot of information quickly from a single person • People will tell you things privately that they wouldn't publicly	• Interviews must be scheduled in advance • Interviewing skills are difficult to learn	• Be prepared to follow-up • Hold the interview at a whiteboard, so that you can sketch as you talk, turning the interview into a model storming session • Actively listen to what they're saying

Requirements Elicitation Techniques

Technique	Description	Strength(s)	Weakness(es)	Staying Agile
Focus Groups	You invite a group of actual and/or potential end users to review the current system, if one exists, and to brain storm requirements for the new one.	• Collaborative technique • Significant amounts of information can be gathered quickly • Works well with dispersed stakeholders • Works well when actual users do not yet exist	• Must be planned in advance • Lots of unimportant information will be conveyed • It's difficult to identify the right people • Focus groups can be diverted by a single strong-willed individual	• Hold it in a room with whiteboards or flip chart paper so people can drawn as they talk

Technique	Description	Strength(s)	Weakness(es)	Staying Agile
Joint Application Design (JAD)	A JAD is a facilitated and highly structured meeting that has specific roles of facilitator, participant, scribe, and observer. JADs have defined rules of behavior including when to speak, and typically use a U-shaped table. It is common practice to distribute a well-defined agenda and an information package which everyone is expected to read before a JAD. Official meeting minutes are written and distributed after a JAD, including a list of action items assigned during the JAD that the facilitator is responsible for ensuring are	• Facilitator can keep the group focused • Significant amounts of information can be gathered quickly • Works well with dispersed stakeholders	• Restricted interaction technique • Facilitation requires great skill • JADs must be planned in advance	• Loosen the rules about when people can talk • Hold it in a room with whiteboards or flip chart paper so people can drawn as they talk

Technique	Description	Strength(s)	Weakness(es)	Staying Agile
Legacy Code Analysis	You work through the code, and sometimes data sources, of an existing application to determine what it does.	• Identifies what has been actually implemented	• Restricted interaction technique • The actual requirements usually differ from what you currently have • It can be very difficult to extract requirements from legacy code, even with good tools	• Must be tempered with more interactive techniques such as interviews and active stakeholder participation .
Observation	You sit and watch end users do their daily work to see what actually happens in practice, instead of the often idealistic view which they tell you in interviews or JADs. You should take notes and then ask questions after an observation session to discover why the end users were doing what they were doing at the time.	• Helps to identify what people actually do • Provides significant insight to developers regarding their stakeholder environments	• Restricted interaction technique • It is hard to merely observe, you also want to interact • Seems like a waste of time because you're "just sitting there" • Can be difficult to get permission	• Observation is best done passively

Technique	Description	Strength(s)	Weakness(es)	Staying Agile
On-Site Customer	In XP the customer role is filled by one or more people who are readily available to provide domain-related information to the team and to make requirements-related decisions in a timely manner.	• Collaborative technique • Information is provided to the team in a timely manner • Decisions are made in a timely manner	• Airs your dirty laundry to stakeholders • Stakeholders need to be educated in their role	• Get your stakeholders involved with development by evolving towards an active stakeholder participation approach
Reading	There is often a wealth of written information available to you from which you can discern potential requirements or even just to understand your stakeholders better. Internally you may have existing (albeit out of date) system documentation and vision documents written by your project management office (PMO) to justify your project. Externally there may be web sites describing similar systems, perhaps the sites of your competitors, or even text books describing the domain in which you're currently working.	• Opportunity to learn the fundamentals of the domain before interacting with stakeholders	• Restricted interaction technique • Practice usually differs from what is written down • There are limits to how much you can read, and comprehend, and a single sitting	• Read details in a just-in-time (JIT) manner

7. Common Requirements Modeling Challenges

- Limited access to project stakeholders
- Geographically dispersed project stakeholders
- Project stakeholders do not know what they want
- Project stakeholders change their minds
- Conflicting priorities
- Too many project stakeholders want to participate
- Project stakeholders prescribe technology solutions
- Project stakeholders are unable to see beyond the current situation
- Project stakeholders are afraid to be pinned down
- Project stakeholders don't understand modeling artifacts
- Developers don't understand the problem domain
- Project stakeholders are overly focused on one type of requirement
- Project stakeholders require significant formality regarding requirements
- Developers don't understand the requirements

8. Agile Requirements Change Management

Agile software development teams embrace change, accepting the idea that requirements will evolve throughout a project. Agilists understand that because requirements evolve over time that any early investment in detailed documentation will only be wasted. Instead agilists will do just enough initial modeling to identify their project scope and develop a high-level schedule and estimate; that's all you really need early in a project, so that's all you should do. During development they will model storm in a just-in-time manner to explore each requirement in the necessary detail.

Because requirements change frequently you need a streamlined, flexible approach to requirements change management. Agilists want to develop software which is both high-quality and high-value, and the easiest way to develop high-value software is to implement the highest priority requirements first. Agilists strive to truly manage change, not to prevent it, enabling them to maximize stakeholder ROI. Your software development team has a stack of prioritized requirements which needs to be implemented - XPers will literally have a stack of user stories written on index cards. The team takes the highest priority requirements from the top of the stack which they believe they can implement within the current iteration. Scrum suggests that you freeze the requirements for the current iteration to provide a level of stability for the developers. If you do this then any change to a requirement you're currently implementing should be treated as just another new requirement.

Prioritization of Requirements

Large software systems have a few hundred to thousands of requirements. Neither are all requirements equal nor do the implementation teams have resources to implement all the documented requirements. There are several constraints such as limited resources, budgetary constraints, time crunch, feasibility, etc., which brings in the need to prioritize requirements.

Most customers on their part have a reasonable idea of what they need and what they want. But during requirements elicitation the customer provides the Business Analyst (BA) with all the requirements that he feels will make his work easier. The customer is not wrong on his part; the BA needs to understand the needs of the business to prioritize the requirements.

Prioritization means "*Order of importance*". BABOK 3.0 suggests 8 factors that influence the prioritization of requirements.

Factors

Benefit - It is the advantage that the business accrues as a result of the requirement implementation. The benefit derived can refer to functionality, quality or strategic / business goals.

Penalty - It is the consequence of not implementing a requirement. It can refer to the loss in regulatory penalties, poor customer satisfaction or usability of the product.

Cost - It is the effort and resources that are required to implement a requirement. A resource can refer to finance, man-power or even technology.

Risk - It is the probability that the requirement might not deliver the expected value. This can be due to various reasons ranging from difficulty in understanding the requirement to implementing the requirement.

Factors

Dependencies - It is the relationship between requirements. As such, a requirement will require the completion of another requirement for its implementation.

Time Sensitivity - Everything comes with an expiry date. There has to be mention of what time the requirement will expire or also if the requirement is seasonal.

Stability - It refers to the likelihood of the requirement remaining static.

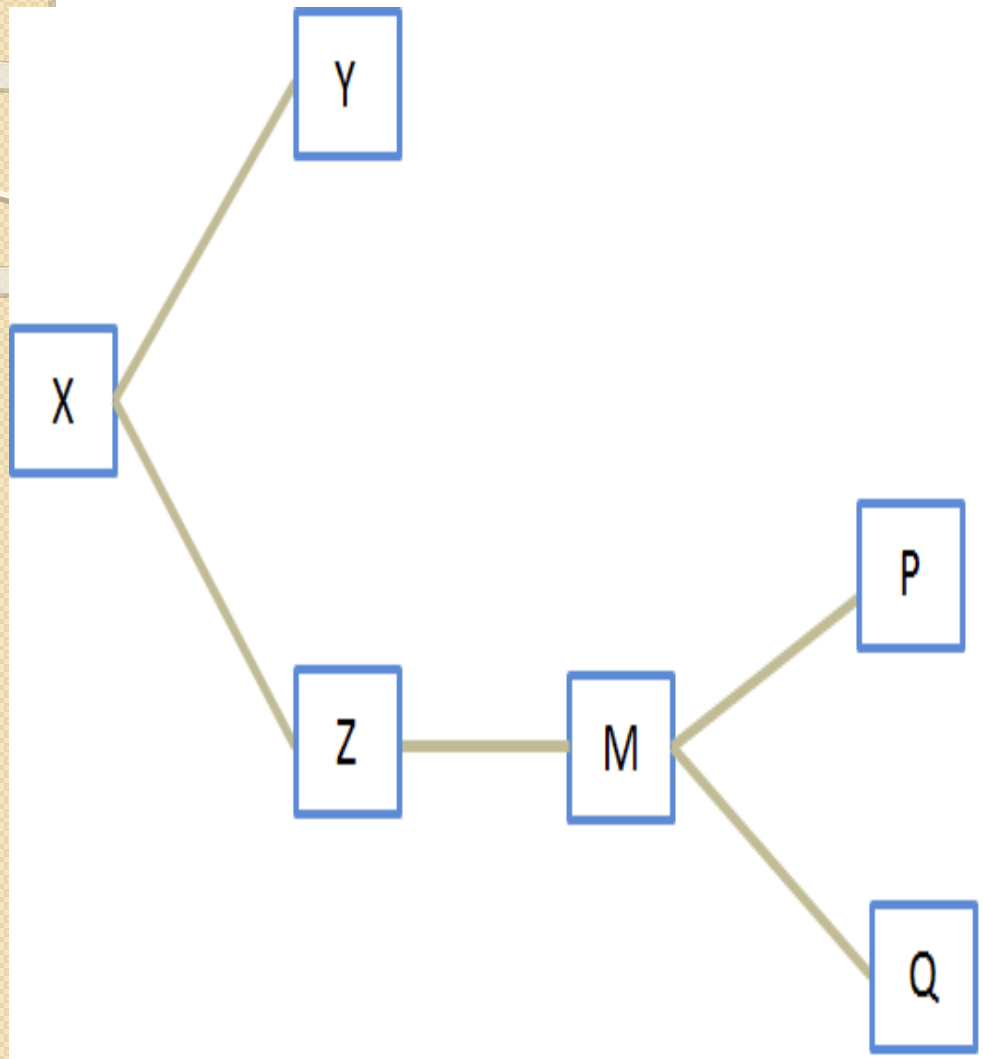
Regulatory/Policy Compliance – Those requirements that must be implemented to meet the regulatory requirements.

The requirements dependency map

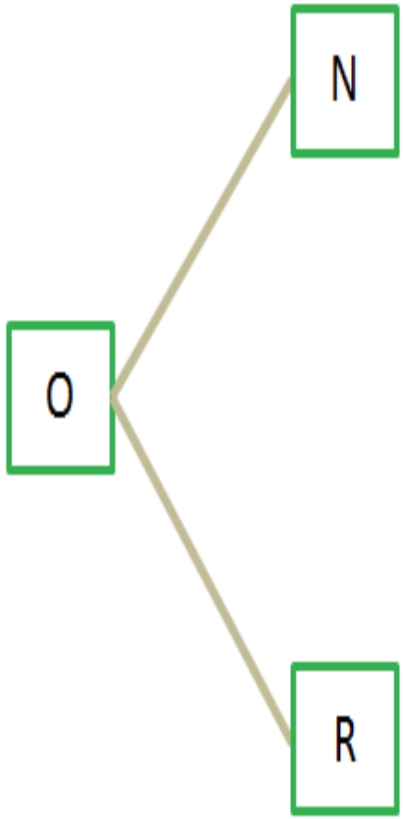
Most requirements are interdependent and you will hardly find any requirement that exists independently. To understand why we need a dependency map – let us take a scenario where you have 9 requirements X,Y,Z,P,Q,R,M,O and N with priorities, on a 5- level scale where 1 is most critical and 5 least critical, as 1,2,1,4,5,1,2,2,3. So, with these priorities it would be logical to begin with requirements X, Z and R.

Now, consider the below dependency chart for the above requirements. The chart clearly brings out the idea that we need to complete X before commencing with Y, although X and Y have the same priority levels. Similarly, we need to complete O before commencing R, although when R has higher priority than O.

The requirements dependency map



Dependency Map 1



Dependency Map 2

The Requirements Prioritization

Understanding the requirements dependency is as just as important as prioritization. Without understanding the requirements dependency, it is highly unlikely that you will arrive at the right order of requirements implementation. So, it is a good idea to have the requirements dependency map in place before prioritizing the requirements.

Numerous methods on how to prioritize requirements have been developed. While some work best on a small number of requirements, others are better suited to very complex projects with many decision-makers and variables. This list of requirements prioritization techniques provides an overview of common techniques that can be used in prioritizing requirements.

Kano Model

The Kano model was developed in the 1980s by Professor Noriaki Kano. Under the Kano Model, features are categorized according to needs and expectations of customers. There are a variety of versions of the Kano model. The original, however, classifies items using five thresholds: Must-be, Attractive, One-Dimensional, Indifferent, and Reverse.

Must-Be — These are expected by your customers. They are features that will not WOW them. They *must be* included in your product, and are often taken for granted.

Attractive — These make users happy when they're there, but don't disappoint them when they're not.

One-Dimensional — These are features that make users happy when they're there, unhappy when they're not.

Kano Model

Indifferent — These have no impact on customer satisfaction levels. For example, refactoring parts of your code so that it is easier to read and understand. There is no direct value to the customer, but it will make it easier for you to maintain in the future.

Reverse — These make users unhappy when they're there, happy when they're not. For example, you might implement high-security features requiring an extra step to login. However, if customers do not value enhanced security, they will become dissatisfied with the extra step.

Why Use the Kano Model?

The Kano model is tremendously useful in organizations that tend to only do Must-Be features. But, to succeed in the marketplace, you also need to deliver attractive and one-dimensional features. So, when time comes to prioritize what goes into a release, it's a good practice to pick one from each of the important categories above.

MoSCoW Model

The MoSCoW model, credited to pioneering data scientist Dai Clegg, has roots in Agile software development. The MoSCoW model has nothing to do with Russia's capital. Rather, it takes its name from the letters that make up its criteria: **M**ust Have, **S**hould Have, **C**ould Have and **W**on't Have.

Must Have — If you would have to cancel your release if you couldn't include it, then it's a Must Have. Must-Have user stories are those that you guarantee to deliver because you can't deliver without, or it would be illegal or unsafe without. If there's any way to deliver without it — a workaround, for example — then it should be downgraded to **Should Have** or **Could Have.** That doesn't mean it won't be delivered. It means that delivery is not guaranteed.

MoSCoW Model

Should Have — Should Have features are important, but *not absolutely vital* to the success of your release. They can be painful to leave out, and may have an impact on your product, but they don't affect minimum viability of your product.

Could Have — Could Have items are those that are wanted or desirable but are less important than a Should Have item. Leaving them out will cause less pain than a Should Have item.

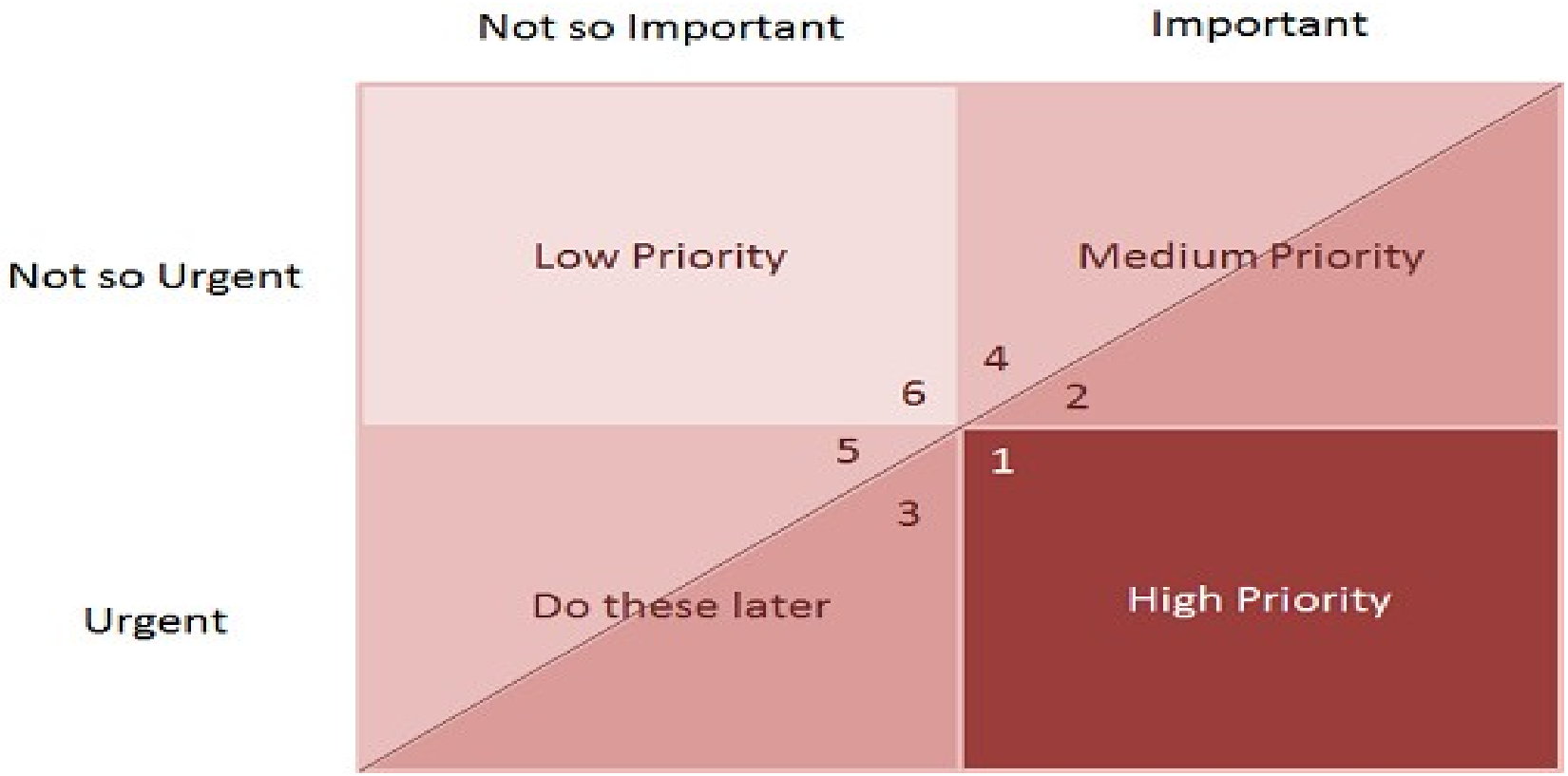
Won't Have— Won't Have user stories are those in which everyone has agreed not to deliver *this time around.* You may keep it in the backlog for later, if or when it becomes necessary to do so.

Combining these two methods, therefore, can be very valuable for Agile teams. By delivering a combination of “must-be”, “attractive”, and “one-dimensional” features, along with “must-haves” according to MoSCoW, you ensure your product has a good mix of customer-friendly, market-ready features.

3. Three-level Scale – When a BA categorizes the requirements in any of the ordering or ranking scale, it is subject to the analyst's understanding of the business. Many analysts suggest that this method has some drawbacks and advocate methods that have more than one scale.

Covey, Rebecca and Merrill would have never in their wildest dreams have thought that their "The four-quadrant **'Eisenhower Decision Matrix'** for importance and urgency", from their self-help book First things First, would become one of the most widely used prioritization techniques in the IT space.

Three level Scale



Three Level Scale

With the numbering on the different sections of the diagram, the priority of the sections is implicit. Important items have the highest preference, while urgent items have lower preference.

High Priority – These requirements are urgent and important. These are requirements that are generally with respect to compliance or contract that cannot be left out. These requirements need to be implemented in the current release and not implementing the same will have some adverse effect on the business.

Medium Priority – These requirements are important but not as urgent. Implement these after you implement the high priority items. If you see closely there is a line that splits this quadrant into 2 parts. Implement the items that are on the right side of the line first as they are relatively of higher medium priority.

Three Level Scale

Do these later – These items are urgent but do not have a lot of effect on the business. Hence do it after completing the more important medium priority items. Similar to the medium priority items, this quadrant has also been split into two; the items on the right side have a higher priority relatively to the items on the left.

Low Priority – These items are neither important nor are they urgent. Complete the items at your leisure after completing the items in sections 4 and 5 respectively.

The items on the right hand side of the diagonal have higher priority. Start with the bottom-right corner of the high-priority quadrant and work your way up and left.

4. Timeboxing/Budgeting – Timeboxing or fixed budgeting process is used when there are fixed timelines/ budgets to achieve the project milestones. Timeboxing is used in projects that are constrained by deadlines where the delivery of the project is as important as the project being delivered on time or being developed within the budget.

This technique is based on the premise that it is more important to have at least the basic product features and release the product on time rather than having all features and launching the product at a later date. Miranda, Program Director, Ericsson Research Canada, in her paper proposes 2 points of estimate – the normal completion effort and the safe completion effort. The normal completion effort is the happy case scenario of requirement being developed while safe completion effort is the estimation based on the worst case scenario.

In Timeboxing, which is the refined version of MoSCoW analysis, the requirements are grouped into small subsets that can be called wholly; these are given relative importance and the time schedule required for implementation. The idea is to move the more important features or ***M****ust,* ***S****hould* and ***C****ould* to earlier phases of implementation to ensure completion of more important features at the time of product launch.

This method is highly used in agile projects as it helps in identifying the more important features and it works very efficiently for iterative approach.

According to BABOK V2, there are 3 approaches that may be used for determining the requirements to include in iteration:

All In – Include all the requirements necessary for the solution to be developed and then remove/postpone the requirements whose implementation will cause the project to miss the deadline.

All Out – Exclude all the requirements for a start and then include the ones that can be implemented within the constrained time schedule.

Selective – Identify the high priority requirements and then add or remove requirements to meet the deadline.

5. Voting – This is the simplest way to prioritize the requirements. When there are too many requirements that need to be categorized into different categories with inputs from different stakeholders, voting is one of the best ways to sort out the prioritization of requirements. I will suggest the voting technique that worked for me in one of the projects I was managing.

In this method we gathered all the stakeholders in a room (others not physically provided were looped in through a call) and each of the stakeholder was given points(say 200, 100, 50 based on the stakeholder's importance) to vote on each of the requirement that had to be incorporated. Then we ran through over 400 requirements on which the stakeholders voted. The requirements with the highest points were chosen for implementation in the first iteration.

Care has to be taken while putting out the requirements to vote such that the requirements are grouped in a logical manner and that the stakeholders are voting on a requirement that can be delivered in entirety. The entire process took a day but it worked out really well.

Concluding remarks

Prioritization is one of the most important stages in the requirements analysis stage. Involve the stakeholders for better categorization and prioritization of requirements.

Analyze the requirements and understand the effect of different factors listed above on the requirements.

Choose the requirements prioritization technique that best suits your need and start prioritizing your requirements.

Challenge the importance of requirements. Lower the priority the better it is.

Business analysis is all about taking the most complex of requirements and breaking it down to simple problems that can be implemented by anyone.