# Agile Requirements Elicitation

Sometimes it is very difficult or not economical to produce a complete, verifiable set of requirements. In this chapter, we explain:

- The circumstances when traditional and/or use case requirements specifications may not be appropriate
- The practices of producing low-ceremony, high-level requirements that are predicted to change throughout development.

*The hardest part of the software task is arriving at a complete and consistent specification, and much of the essence of building a program is in fact the debugging of the specification.* – Fred Brooks, 1987 (Brooks, 1987)

There are many benefits to devising and verifying a software requirements specification. The requirements guide the rest of the software development – so it can be exceedingly costly and/or devastating to the project if the requirements are incorrect or incomplete. Additionally, the requirements can act as a contract between the customer and the software development.

**Traditional Versus Agile Approaches**

As you've probably realized, though, it takes a lot of hard work to develop a verifiable set of requirements—sometimes it can even seem impossible. As you can see from the above quote, presumably the world's most famous software engineer, Fred Brooks, agrees! He goes even further to say that once you are done with this hardest task of arriving at a complete and consistent specification, you end up spending a great deal of time debugging the creation you worked so hard on. These difficulties have led to a "don't even bother" philosophy when working on a project that is expected to have a lot of requirements changes. With the agile software development model, on the other hand, a high-level, low-ceremony version of the requirements specification is produced; the agile team is ready to respond to the inevitable changes in these requirements.

**Context Determines the Approach**

Both the agile approach and the verifiable approaches to requirements engineering are appropriate in their own context. Projects with a lot of change that need to get out to the market quickly might be best done with high-level, low-ceremony requirements practices. Stable projects with safety-critical implications could best be done with a plan-driven, well-documented specification.

## 1 Essential Aspects of Agile Requirements

*Agile practices are based on the belief that neither the customer nor the developers have full knowledge in the beginning and that the important consideration is having practices that will allow both [the customer and the*

> *developer] to learn and evolve as that knowledge is gained—without ongoing recrimination.* (Highsmith, 2002)

Those software developers who use agile methods believe that the statement of requirements will evolve through the whole project. As a result, a verifiable, documented form of the requirements (that would exhibit the characteristics of being understandable, non-prescriptive, correct, complete, consistent, concise, unambiguous, testable, traceable, and feasible) is never produced. Instead, agile requirements are expressed as high-level, brief written statements of the best information fairly easily available.

## 1.1 Frequent, Personal Interactions

Because the requirements are just short statements of the best information easily available, an *essential aspect* of agile requirements is to have *frequent, personal interactions with customers and/or stakeholders*. These interactions are necessary for the developers to understand the details of what the customers really want. Best case, a customer is available on-site and resides with the development team. This ability for a developer to easily have personal interaction with a customer makes the difference between (1) the software developer finding out what the customer really wants and (2) the software developer making assumptions about what the customer probably wants. Acting on the latter of these two options is dangerous – so the readily-available customer is critical for the success of the project. In the case of "shrink-wrapped" software (such as Microsoft Office) that is distributed to hundreds, thousands, or even millions or customers, it can be difficult to get a real customer to sit with the team. Instead, "proxy users" are inserted between the developers and the "real" customers. These proxy users are business analysts and/or systems analyst who work directly with real users and can hopefully communicate the wishes of the majority of customers.

## 1.2 Frequent Delivery of Software

Another essential aspect of agile requirements is to have *frequent delivery of software to customers*. Most agile teams provide a new version of the working system to customers every two weeks. Only when the customer can actually use the evolving project can he or she provide vital feedback on what has been done and a new, refreshed view of follow-on requirements based on the progress so far. Additionally, frequent delivery helps to keep the momentum of customers working with developers.

> *There is nothing that focuses requirements better than seeing the nascent system come to life. Therefore, capturing the specific details about the requirement long before it is implemented is likely to result in wasted effort and premature focusing. Therefore, if requirements are developed on an as-you-go basis, in an agile approach, the development can be more efficient in the long run than if requirements were elicited up front. (Martin, 2003)*

## 1.3 Expressing Requirements as Features

In varying form, agile methods use high-level, low-ceremony requirements practices. For example, the Feature-Driven Development (Coad, LeFebvre et al., 1999) and Scrum methodologies (Schwaber and Beedle, 2002) express their requirements as features. A *feature* is *a small, client-valued function expressed in the form <action.><result><object> (e.g. calculate the total of a sale)* (Palmer and Felsing, 2002). An example of a Monopoly feature might be:

- Purchase available property.

With such a short description of the requirement, it is essential that software developers and stakeholders talk often to clarify exactly what is needed for each feature.

## 2 The Basics of User Stories

Of the agile methodologies, the Extreme Programming (Beck, 2000) (XP) methodology specifies its requirements practices with the most detail and rigor. Therefore, in this chapter we will discuss the XP form of requirements in great deal with an extensive example. XP specifies its requirements in the form of user stories. A *user story represents a feature customers want in the software* (Beck and Fowler, 2001). User stories are written by a customer, maybe with the assistance of the developers. Formal requirements and use cases are often documented in an archived document, available to the development team via a hardcopy document or online. Sometimes user stories are entered into an online system, but most often the stories are written on index cards.

### 2.1 Using Index Cards

The index cards are passed to the team member(s) who will work on the requirements and/or they are pinned to a "big visible board" in the area where the team works. As with features, the few sentences on the card are not enough for a developer to really understand what the customer wants. Instead these cards are a "mnemonic token of an ongoing conversation" (Martin, 2003) and the customer and the developer must converse often to gain a thorough understanding of what is desired.

### 2.2 Estimating Ideal Development Time

After the customer writes the stories, the developers estimate how long the stories might take to implement. Consider a team that has two-week iterations. Then, each story will get an estimate of days of ideal development time. *Ideal development time* describes how long it would take to implement the story in code if there were no distractions, no other assignments, and you knew exactly what to do (Beck, 2000). If a story would take longer than two weeks, the story must be broken down into multiple smaller stories. If a story takes less than one day, the story is too detailed a level and should be combined with other small stories.

## 3 Gathering User Stories

The initial requirements should be gathered during a small, preferably offsite meeting. The meeting is attended by software developers and a small group of customers who bring domain expertise and are representative of the user population. There are two ways of gathering user stories, the goal oriented approach and the scattergun approach.

With the *goal-oriented approach* (Astels, Miller et al., 2002), the meeting starts with a goal of the system, such as "Players can move around the board." The group then considers what steps the user takes when trying to achieve this goal. These steps are written as user stories.

Conversely, with the *scattergun approach* (Astels, Miller et al., 2002), user stories are generated as expectations arise in the conversation. No structure is imposed on the way the meeting progresses.

Story writing is iterative and interactive. Customers propose a story. The developers consider if the story can be tested and/or estimated. If not, the team converts the story into one that can be tested and estimated. Then the developers estimate the story, asking the customer many questions of clarification to do this. If the story is too big or too small, the story is split or combined.

## 4 How to Write an Agile Requirement User Story

Here is an example of a user story for the Monopoly game. From it we can extract some criteria to guide us in our writing of a user story. The user story appears on an index card like this:

| | | |
|---|---|---|
| Title: Draw Lose Money Card | | |
| Acceptance Test: communityChest2 | Priority: 3 | Story Points: 1 |
| When a player lands on a Community Chest or Chance cell, the player draws a card from the Community Chest or Chance. If the card is a lose money card, the player pays the money to the bank. If he does not have enough money, he is out of the game, and the cells he owns become available without any houses. | | |

### 4.1 Index Card Elements

Now let us dissect this user story:

- *Title.* Write a two or three word title for this user story. The title should begin with a present-tense verb phrase in active voice (similar to the name of a use case). Write this title in the middle of the top line of an index card.
- *Acceptance Test.* List the unique identifiers of the acceptance tests for the user story. The unique identifier can be a word (such as LoginTest in the example above) or a alphanumeric string.
- *Priority*. The customer must decide how important each of the stories is so that the most important stories can be done first. We are using a 1-2-3 priority scheme where a 1 is given to the most important stories.
- *Story Points*. The number of days of ideal development time.
- *Description:* Write 1–2 sentences in the main space of the index card. These describe a single step toward achieving the goal.

**4.2 Criteria for User Stories**

Here are some important points to remember when writing user stories (Beck and Fowler, 2001):

- *Stories must be understandable to the customer*. The stories should be written in the natural language of the customer, not in any kind of technical language or form.
- *Each story must provide value to the customer*. Therefore, there are no stories for things such as designing databases or developing an infrastructure. Databases and infrastructure do not provide value to the customer. Instead, the first story that needs the database would need to include the resources for designing and developing the database needed to complete the story.
- *Developers do not write stories.* As developers, we might think we know about requirements the customer hasn't thought of yet. However, the customer needs to be involved in every story. So, if the developer thinks he or she has thought of something the customer hasn't yet thought of, this must be discussed with the customer before a story can be written.
- *Stories need to be of a size that several of them can be completed in each iteration.* At the end of each iteration (generally every two weeks), new functionality must be demonstrated (and perhaps delivered) to the customer. This functionality needs to be of value to the customer.
- *Stories should be as independent of each other as possible.* By this we mean that it should not be necessary for one story to be completed before the development of another can start. It is not possible to completely remove dependencies between stories, but we should strive to minimize dependencies as much as possible. By minimizing the dependencies, we can have the freedom to schedule the work in any order, giving the customer maximum flexibility for scheduling their priorities.
- *Each story must be testable.* We need to definitely know whether or not we are done with a story. This prevents ambiguous stories, such as "The betting process must be easy to understand." The phrase "easy to understand" is ambiguous because whether or not this story is implemented depends on the interpretation of the tester. We'll discuss this more in the next section.

## 5 Acceptance Tests

In the agile approach, in addition to gathering requirements, writing user stories, another important step is the development of acceptance test cases to verify the implementation of the user story.

An *acceptance test* is a test case written by the customer (in partnership with the developers). When the customer runs the test case and it runs, he or she can feel confident that the team has, indeed, implemented the desired functionality. One or more automated acceptance tests must be created to verify the user story has been correctly implemented.

The details about the user stories are captured in the form of acceptance tests specified by the customer. Often details of the requirements are worked out as the test case is written. Together, the user stories plus the corresponding acceptance test case(s) are used to verify that the system is behaving as the customers have specified. (Martin, 2003)

Ideally, the acceptance test cases are automated by the development team. Then, the suite of automated acceptance test cases can grow over the course of development. The test suite can be run again and again (at least daily) to ensure that no new functionality/code has broken the previously working user stories.

There should be traceability between the user story and the acceptance test(s) used to verify that user story. Then, if an acceptance test works, this is "proof" that the story is working under the specific conditions of the test; if an acceptance test does not work, the story has not been implemented to properly work under the conditions of the test. Often, agile developers will state that there should be at least one acceptance test case per use story. Sufficient testing would necessitate using many more than one test case per requirement. Usually there is one acceptance test—a basic, "everything goes smoothly" success test—just scratching the surface.

## 6 Documenting Non-Functional Requirements and Constraints

Use cases and user stories are functional requirements. As you know, functional requirements are only part of the story. In the process of eliciting user stories, you must all pay special attention to understanding and documenting the non-functional requirements, security and privacy requirements, and constraints. Because so much of the conversation will be focused on the user stories, you must make a dedicated effort to understand the customers' usability, reliability, availability, and performance needs. Additionally, you must understand the security and privacy concerns of the project and, as much as possible, translate these concerns into user stories. For example, the players of the Monopoly game will not like to wait for very long for the dice to "roll" or for their game piece to move.

Most often the user story cards are augmented with a simple listing of the non-functional requirements (including security and privacy) and the constraints. Although non-functional requirements and constraints can't really be written as use stories, they impact many stories. For example, if a non-functional requirement was that all transactions must be performed in under 1 second, every transactional user story must understand this criteria. Or, if the system must be 96% reliable, the whole team is impacted in everything they do by the need for high reliability and the need to meet this criteria for customer satisfaction with the system. Therefore, the user story cards should be augmented with a listing of non-functional requirements and constraints.

## 7 Examples of User Stories—Online Monopoly Game

We will now show you a selection of user story cards with corresponding acceptance test cases followed by some example non-functional requirements and constraints for the online auctions management system.

**Table 1: User Story Summary**

| User Stories | Priority | Points | Acceptance Tests |
|---|---|---|---|
| Move Player | 1 | 1 | playerMovement1 |
| Move Player in Turns | 1 | 1 | playerMovement2 |
| Pass Go | 2 | 2 | passGo |
| Free Parking | 2 | 1 | freeParking |
| Go to Jail | 2 | 2 | jail1 |
| Get Out of Jail | 2 | 2 | jail2 |
| Purchase Property | 1 | 2 | purchasingProperty1 |
| Pay Rent | 3 | 1 | payRent1 |
| Pay Rent and Bankruptcy | 3 | 2 | payRent2 |
| Trade Properties | 3 | 3 | tradeAccept, tradeDecline |
| Buy Railroad | 3 | 1 | railroad1 |
| Pay Rent to Railroad | 3 | 2 | railroad2 |
| Buy Utility | 3 | 1 | util1 |
| Pay Rent to Utility | 3 | 2 | util2 |
| Buy House | 1 | 2 | buyHouse1 |
| Draw Jail Card | 2 | 2 | communityChest1 |
| Draw Lose Money Card | 3 | 1 | communityChest2 |
| Draw Gain Money Card | 2 | 1 | communityChest3 |
| Dra31w Move Player Card | 3 | 2 | communityChest4 |
| **Total Story Points** | | 31 | |

### 7.1 Functional Requirements

The functional requirements of the Monopoly game are now listed as user stories.

| Title: Move Player | | |
|---|---|---|
| Acceptance Test: playerMovement1 | Priority: 1 | Story Points: 1 |
| A player moves based on the dice roll (two dice, each with six faces). When the user reaches the end of the board, he cycles around. | | |

| Title: Move Players in Turns | | |
|---|---|---|
| Acceptance Test: playerMovement2 | Priority: 1 | Story Points: 1 |
| The players should play in turns. | | |

| | | |
|---|---|---|
| Title: Pass Go | | |
| Acceptance Test: passGo | Priority: 1 | Story Points: 2 |
| When a player passes or lands on the GO cell, the bank gives the player $200. | | |

| | | |
|---|---|---|
| Title: Free Parking | | |
| Acceptance Test: freeParking | Priority: 2 | Story Points: 1 |
| When a player lands on Free Parking, nothing happens. | | |

| | | |
|---|---|---|
| Title: Go To Jail | | |
| Acceptance Test: jail1 | Priority: 2 | Story Points: 2 |
| When a user lands on the "Go to Jail" cell, the player goes directly to jail, does not pass go, and does not collect $200. | | |

| | | |
|---|---|---|
| Title: Get Out of Jail | | |
| Acceptance Test: jail2 | Priority: 2 | Story Points: 2 |
| When a player is in Jail, he must pay 50 dollars to get out of jail in the next turn. If he does not have enough money, he is out of the game, and the cells he owns become available without any houses. | | |

| | | |
|---|---|---|
| Title: Purchase Property | | |
| Acceptance Test: purchasingProperty1 | Priority: 1 | Story Points: 2 |
| When a player lands on a property cell, and it is available, the player may purchase it. The price is the land value of that property. | | |

| | | |
|---|---|---|
| Title: Pay Rent | | |
| Acceptance Test: payRent1 | Priority: 3 | Story Points: 1 |
| When a player (A) lands on a property owned by another player (B), A must pay rent to B. The level of rent paid is a base level of rent, unless the owner has a monopoly or houses/hotel. | | |

| | | |
|---|---|---|
| Title: Pay Rent and Bankruptcy | | |
| Acceptance Test: payRent2 | Priority: 3 | Story Points: 2 |
| If player B owes player A more money than player B currently has, player B is bankrupt, and must give all of their property to player A . | | |

| | | |
|---|---|---|
| Title: Trade Properties | | |
| Acceptance Test: tradeAccept, tradeDecline | Priority: 3 | Story Points: 3 |
| If player 2 wishes to purchase a property from player 1, player 2 will name an amount of money to pay player 1 for the property they wish to own. Player 1 can decide to accept or decline the offer. | | |

| | | |
|---|---|---|
| Title: Buy Railroad | | |
| Acceptance Test: railroad1 | Priority: 3 | Story Points: 1 |
| The land value of the railroads is the same. | | |

| | | |
|---|---|---|
| Title: Pay Rent to Railroad | | |
| Acceptance Test: railroad2 | Priority: 3 | Story Points: 2 |
| When player A lands on player B's railroad, A pays rent to B based on the number of railroads B owns. If the base rent of a railroad is R, and the number of the railroads B owns is N, the amount of rent A needs to pay B is R * $2^{N-1}$. | | |

| | | |
|---|---|---|
| Title: Buy Utility | | |
| Acceptance Test: util1 | Priority: 3 | Story Points: 1 |
| The land value of the utilities is the same. | | |

| | | |
|---|---|---|
| Title: Pay Rent to Utility | | |
| Acceptance Test: util2 | Priority: 3 | Story Points: 2 |
| When player A lands on player B's utility, A pays rent to B based a dice roll. If player B owns 1 utility, A pays 4 times the dice roll. If player B owns 2 utilities, A pays 10 times the dice roll. There can only be two utilities on a game board. | | |

| | | |
|---|---|---|
| Title: Buy House | | |
| Acceptance Test: buyHouse1 | Priority: 1 | Story Points: 2 |
| A player has monopoly when he purchases all the properties of a color group. When a player has a monopoly of a color group, he can buy houses for those properties at the beginning of his turn. Player cannot purchase more than 5 houses on any given monopoly. | | |

| | | |
|---|---|---|
| Title: Draw Jail Card | | |
| Acceptance Test: communityChest1 | Priority: 2 | Story Points: 2 |
| When a player lands on a Community Chest or Chance cell, the player draws a card from the Community Chest or Chance. If the card is a Jail card, the player goes to Jail without getting paid when passing the Go cell. | | |

| | | |
|---|---|---|
| Title: Draw Lose Money Card | | |
| Acceptance Test: communityChest2 | Priority: 3 | Story Points: 1 |
| When a player lands on a Community Chest or Chance cell, the player draws a card from the Community Chest or Chance. If the card is a lose money card, the player pays the money to the bank. If he does not have enough money, he is out of the game, and the cells he owns become available without any houses. | | |

| | | |
|---|---|---|
| Title: Draw Gain Money Card | | |
| Acceptance Test: communityChest3 | Priority: 2 | Story Points: 1 |
| When a player lands on a Community Chest or Chance cell, the player draws a card from the Community Chest or Chance. If the card is a gain money card, the player gets the money from the bank. | | |

| | | |
|---|---|---|
| Title: Draw Move Player Card | | |
| Acceptance Test: communityChest4 | Priority: 2 | Story Points: 2 |
| When a player lands on a Community Chest or Chance cell, the player draws a card from the Community Chest or Chance. If the card is a move player card, the player goes to the specified cell. If the player passes go, he or she is paid $200 from the bank. | | |

**7.2 Non-functional requirements**

1. A user shall respond to any user input within 0.01 seconds.
2. The system shall update user data within 0.01 seconds.
3. The system shall catch improper input from all text fields

**7.3 Constraints**

1. The system shall be developed using Java.
2. The system shall be tested using the JUnit and FIT frameworks.

## 7.4 Sample Acceptance Tests

For illustrative purposes, two sample test cases for two of the user stores are given below in Table 2.

**Table 2: Sample acceptance test cases**

| Test ID | Description | Expected Results | Actual Results |
|---|---|---|---|
| util1* | Precondition: Game is in test mode.<br>Number of players: 2<br>• Enter 2 in # of player's dialog<br>• Enter player 1's name as 1<br>• Enter player 2's name as 2<br>• Press 1's Roll Dice button<br>• Enter dice roll of 12<br>• Press 1's Purchase Property button<br>• Press 1's End Turn button | • 1 has $1350<br>• 2 has $1500<br>• 1 is located at Electric Company<br>• 2 is located at Go<br>• 1 owns Electric Company<br>• 2 does not own any property | |
| purchasingProperty1* | Precondition: util1 has passed<br>• Press 2's Roll Dice button<br>• Enter dice roll of 3<br>• Press 2's Purchase Property button<br>• Press 2's End Turn button | • 1 has $1350<br>• 2 has $1440<br>• 1 is located at Electric Company<br>• 2 is located at Baltic Avenue<br>• 1 owns Electric Company<br>• 2 owns Baltic Avenue | |

## 6 Summary

Several practical tips were provided for documenting requirements in a low-ceremony, high level manner – and when it was appropriate to use this type of requirement documentation rather than traditional or use case requirements. These tips are summarized in Table 3.

| | |
|---|---|
| 🔑 | Agile requirements are appropriate for projects with short cycle times/iterations and volatile requirements. |
| 🔑 | Because agile requirements are not stated completely, developers must have regular access to customers so the specifics of the requirements can be clarified as necessary. |
| 🔑 | Each user story should have at least one acceptance tests. The acceptance test helps to determine the specifics of what the customer wants. |
| 🔑 | Acceptance test cases are written by the customer in partnership with the developer. The customer uses the acceptance test cases to determine if a user story has been completed. |

**Table 3: Key Ideas for Agile Requirements**

**Glossary of Chapter Terms**

| **Term** | **Definition** | **Source** |
|---|---|---|
| Feature | a small, client-valued function expressed in the form <action.><result><object> (e.g. calculate the total of a sale) | (Palmer and Felsing, 2002). |
| User story | a feature customers want in the software | (Beck and Fowler, 2001) |

## References

Astels, D., G. Miller, et al. (2002). Prentice Hall. Upper Saddle River, NJ, Prentice Hall.

Beck, K. (2000). Extreme Programming Explained: Embrace Change. Reading, Massachusetts, Addison-Wesley.

Beck, K. and M. Fowler (2001). Planning Extreme Programming. Reading, Massachusetts, Addison Wesley.

Brooks, F. P. (1987). "No Silver Bullet." IEEE Computer **20**(4): 10-19.

Coad, P., E. LeFebvre, et al. (1999). Java Modeling in Color with UML, Prentice Hall.

Highsmith, J. (2002). Agile Software Development Ecosystems. Boston, MA, Addison-Wesley.

Martin, R. C. (2003). Agile Software Development: Principles, Patterns, and Practices. Upper Saddle River, Prentice Hall.

Palmer, S. R. and J. M. Felsing (2002). A Practical Guide to Feature-Driven Development, Prentice Hall PTR.

Schwaber, K. and M. Beedle (2002). Agile Software Development with SCRUM, Prentice-Hall.

**Chapter Questions:**

1. AgileGo is a development team that practices agile methodologies. After several successful projects, they found out that the team's velocity is 20 story points per week (which means they can finish 20 story points every week). AgileGo has just got a new project recently. After working with the customer, they had an initial version of user stories. These stories weighted 230 points totally.
   A. Based on experience, how long will AgileGo need to finish the project?
   B. The client needs the software working in two months. How many story points can AgileGo finish in two months?
   C. What can AgileGo do to deliver the software in two months?
2. It is important to have an on-site customer if we practice Extreme Programming. However, we cannot find the real customers for shrink-wrapped software like Microsoft Office. How shall we develop the requirements for such software?
3. Agile methods are said to fit into software projects of turbulent requirements. List the features of agile requirements elicitation, and discuss, from the perspective of requirement elicitation, why agile methods are suitable for e-commerce projects.
4. When writing user stories, the development team uses velocity to estimate performance. Velocity is the number of story points the team can finish in a period of time, e.g. 20 points/week. If you participated in an agile software project, how do you estimate the performance of yourself, and your team? (Hint: Agile developers always use the easiest way to solve problems.)
5. The students need to do a term project in the Operating Systems course. This is a teamed project, and each team consists of 4 students. In the project, the students are asked to write a memory management system. The students can choose their own algorithm or memory management scheme. The final score of the project is given based on the effectiveness of the program. Given a fixed amount of memory, if a team's program can load more data, the team will get a higher score.

   If you were a student in this course, will you use user stories to manage the requirements of this project? Justify your answer. If this is not a school project, but a commercial project, will you change your answer? Why?
6. In this chapter, we've learned about agile requirements elicitation. We know that agile practices are best used with projects which have volatile requirements. Suppose we were developing a software product using an agile approach. One day, the customer wants to change a feature, which is described in a user story card and has been developed and tested. Worse still, this feature is related to several other features. Discuss the things we need to do to make this change.
7. There is an interesting comparison of traditional and agile processes. Traditional plan-driven processes are like "ready, ready, ready, …, aim, aim, aim, …, fire!" and hope the bullet hit the target. Agile processes are like "ready, aim, fire, ready, aim, fire, …, fire," and the bullet will hit the target at last. In this chapter, we know that agile processes are used to address volatile requirements, or "moving targets." Actually, in real life, we can see some sports, like golf, which have a fixed target and still require

the "ready, aim, fire, ready, aim, fire, …, fire" technique. Is there any software project that does not have volatile requirements and still is suitable for agile processes? Discuss the considerations, other than requirements volatility, that the developers take to employ agile processes.

---

You are going to write user stories for the systems below. There are two ways to do so:

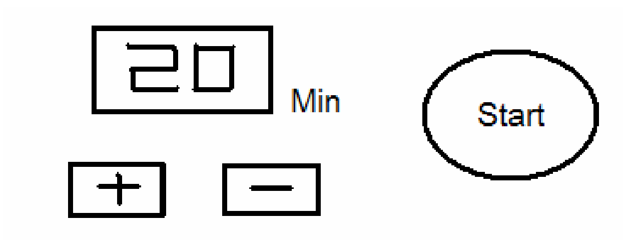
1. As teamed exercise. Each team has 3 or 4 people. One of them is the customer, and the others are developers. The customer needs to understand the problem domain. The developers will work collaboratively with the customer for the user stories. This simulates how we do things in an agile way.
2. As individual exercise. You will work on the exercise alone, following the steps:
   a. List the specification of the system. You will act as the role of customer, so the specification should be written using natural language.
   b. Write the user stories based on the specification from step a. You will act as the role of developer. If you have a problem with a user story, read the specification and recall what you have thought about it in step a.
   c. During step b, if you think the specification is not correct, correct it and make sure the user story and the specification is consistent.
   d. Evaluate the size of each user story.
   e. Prioritize each user story.

This is not how agile developers do the requirements, however. In real agile projects, developers and the customer work together, and the user stories are developed in an iterative fashion.

Be sure to evaluate and prioritize each user story. Also, you need to consider the nonfunctional requirements.

---

8.



Above is a sketch of the control panel of a drier machine. The panel has three buttons and a display. The display can show any two-digit number. The purpose of the + and – buttons is to set up the timer. When the + or – button is pressed, the drying time is increased and decreased, respectively, by five minutes. This drier allows the user to select the drying time from 5 minutes to 60 minutes. The selected drying time is shown in the display.

When the user pushes the start button, the machine starts running. The display then shows the remaining time. The display only shows the minute. For example, if there are 19 minutes and 20 seconds left, the display shows 20. If there are 19 minutes and 0 second left, the display shows 19. The machine stops when the selected time is up.

The drying machine has a safety feature. If the door is opened when the machine is running, the machine will stop immediately. The timer remains unchanged, though. Therefore, the user can close the door again and push the start button, and the machine will run until the timer counts to zero.

9. A university uses a campus-wide card – it is called CampusPass. With this card, the students, faculties, and staff of the university can deposit money in their accounts, and use it in the grocery stores and food courts in the university. Recently, to deposit money, the card holders need to go to the cashier's office.

   To make the deposit process easier and more accessible, several kiosks are going to be deployed at several spots. They are connected to the central database. The card holders can deposit money at the kiosks. A software team is assigned a task of developing the software for the deposit stations. Make necessary assumptions, and develop the user stories for the software.

   Also note that the money deposited in the kiosk is locked in a safe. The software does not need to worry about how the money is drawn.

10. Suppose you are going to start a web-based business. You have got a new idea: an online book shelf. A user can pay little amount of money and enjoy ubiquitous access to a limited Internet hard disk space. (Of course, the user can always pay more money to buy more space.) The user can upload files as long as he or she has Internet connection. Additionally, the user can have these files printed and shipped to his/her home or office, with additional charge. Write the user stories for this web site.