Agile Tutorial for the Senior Project Class
School of Computing and Information Sciences
Florida International University

What is Agile?

In simple terms, Agile is a collection of ideas to guide both the management of software projects and the development of software products.

Agile incorporates ideas proposed decades ago and practiced by software engineers over the years such as refactoring and the use of coding standards. Agile's inclusion of these ideas is significant nonetheless because it gives them a renewed relevance in the context of new ideas such as the development of software in short, horizontal iterations that always produce working software.

Although quite a lot has been said and written about Agile, it is important to keep in mind that its ideas are still evolving, with teams and companies adopting their own custom versions of the Agile principles and practices to better suit the culture of their organizations, the needs of their clients, and the nature of their software projects.

Agile Principles and Practices

The following Agile principles and practices were selected for their relevance in the context of the Senior Project class, and adapted to satisfy its requirements and time constraints. This is not an exhaustive compendium on the subject by any measure, but it does contain many of the key ideas and current practices. Learning and following the ideas presented here should provide you with a solid footing into the Agile world, in much the same way that the Senior Project class will give you the main building blocks required to complete a software project, and a taste of the industry practices in an academic setting.

Customer at the Center

In many traditional software engineering approaches such as the Waterfall Model, the client's involvement was mainly limited to the very begining of the project during requirements elicitation, and to the very end during the acceptance test once the vast majority of the development was completed.

In Agile, the interaction with the client is a key part of the entire development process. The client may not have a detailed list of all the functionality required from the software product from the beginning, but the ones that are well rounded may be implemented by the team and presented to the client while the others are refined along the way.

Agile considers not only the client but all stakeholders such as end users, executives, and lawyers. Frequent interactions with the stakeholders and acceptance of incremental pieces of functionality avoid one of the main threats software projects face: building a software system that does not meet the stakeholders' needs.

This incremental and somewhat open-ended approach to building a software product does not

eliminate the need for an initial set of requirements that captures the bulk of the development effort. This is especially important when developing commercial software where a cost estimate or even a contract is required before work can begin.

Self-organizing Team

The main job of the team in Agile is deciding which tasks to tackle next. This model directly opposes the traditional role of a manager dictating every aspect of the team's day-to-day operations. It is precisely in this contrast where the essence of the self-organization notion lies. However, giving the team full autonomy in planning their daily tasks does not imply the absence of a guiding force or strategy to keep the project on the right track. In practice, many Agile teams have a manager that operates at a higher level than traditional managers, or they may completely do away with this role and rely on peer pressure to keep the team accountable and focused.

User Stories

A user story is a detailed description of a piece of functionality of the system from the point of view of a specific user. For instance, “As a professor, I want to be able to modify the attendance of a student to be able to fix possible data entry mistakes that can occur while taking attendance.” As opposed to the concept of a use case you learned in your Software Engineering class, which describes an entire interaction scenario with the system, a user story is much more specific.

In order to measure progress and make realistic plans on the amount of work that can be accomplished in each iteration, user stories are assigned story points. As an Agile artifact, a story point is just an integer that estimates the relative difficulty of a user story compared to the other user stories. In this sense, story points are not indicators of time values but of relative effort. To translate the story points to time in order to assess the progress of an iteration, another Agile artifact comes into play: velocity.

In Agile, velocity denotes the number of story points completed in the current iteration. That is, velocity is a measure of the amount of work accomplished in the current iteration. To find out how much time it takes to the team to complete one story point you can simply divide the number of completed story points by the number of working hours in the iteration. However, a more common and important question is whether or not the project is on track at any given point in time. This question is not answered by the previous calculation but by looking at a burndown chart, another key Agile artifact discussed later in this tutorial.

User stories are a great tool for capturing what the customer wants and play an essential role during acceptance testing. However, relying only on user stories to define the functionality of the system risks losing the overall picture and developing redundant or inconsistent subsystems.

You should collect the user stories and then refine them by grouping, splitting, merging, and generalizing them until you have a collection of higher-level requirements that will serve as the input to the design phase.

Sound Initial Design

Agile advocates the complete elimination of an *upfront* design, arguing that repeatedly refactoring an initially minimal code base and system architecture as they grow can yield great software. In practice, however, given that refactoring only changes nonfunctional attributes of the software and not its functionality, refactoring a flawed design will still yield poor software. Even if repeated refactoring could eventually converge to a good design, it would likely be prohibitively time consuming and expensive.

A good compromise is the use of a combination of an upfront design with refactoring. You should start with a sound design that includes the essential features of all the major components of the system. This will help protect you from the pitfalls of not having an upfront design, and at the same time will allow you to start the development phase as soon as possible, which is a key Agile idea.

Develop Only the Agreed-upon Functionality

The number one priority in Agile is the timely delivery of a software product that works as expected and behaves as the client wanted. In this context, software features traditionally deemed desirable in software engineering such as reusability and extendibility are of less or no importance. This idea is motivated by the fact that creating software with such features is time consuming and has almost zero visibility to the client. Also, those features frequently end up being a waste when the software is never extended or reused.

Applying this approach gets tricky when you consider the other side of that same coin. The client may request a new feature or a change to an existing feature that is very simple from his/her point of view, but may entail a complete redesign of the system if such a request was not anticipated.

In practice, striving for those features is still very much a good idea as long as it does not risk the main goal of delivering the requested product on time.

Agile also advocates for implementing only the functionality requested. Functionality that is not needed or used only by a few users takes away resources from other more important features and may delay releases. Taken to the extreme, some authors advise that if a given functionality is not requested as part of a user story, it should not be implemented even if the team knows that it will be eventually needed.

In the context of this class, however, proposing extra features for a system that is too small in order to make it significant is encouraged as you will be evaluated in part by this metric.

Coding Standards and Egoless Programming

The use of coding standards is one of the most used examples of a practice that was not introduced by Agile, but in which Agile plays an important role by advocating for it. The team should agree on a set of style rules tailored for their chosen software stack and follow them strictly and consistenly.

The idea of egoless programming may be rather broad. Its inclusion here is intended as a

reminder that all code written by the team should be seen as a coherent piece and that individual team members should constantly strive for making their style consistent with that of the whole team. Consistency and clarity are more important than individual personalities or styles as they will significantly reduce the effort required by current and future team members to understand the code written by other members.

Short Working Iterations

Agile development is iterative. Each iteration should last from two to four weeks and at the end should produce a working system with part of the agreed-upon functionality. The fact that each iteration produces a partial but functioning system is referred to as horizontal iterations, and differs from traditional notions of iterative development in which iterations could produce successive subsystems such as the data, the business logic, and the user interface layers. Agile horizontal iterations provide and end-to-end user experience that albeit limited, allows the customer to provide early feedback.

In this class, we will adopt a two-week iteration cycle. The Scrum section of this tutorial contains more details on the schedule of an iteration.

Iterations in Agile are time-boxed, that is, their duration is fixed in advance. If some planned functionality cannot be finished during the iteration, it gets moved to a future iteration or discarded, but the iteration must end on its scheduled date.

The number of unfinished tasks spilling over from one iteration to the next should be minimized. This is accomplished by assessing the effort required for the tasks to be completed as accurately as possible, which is in turn a result of years of experience and team rapport.

Frequent Integration

Software integration consists of putting together the existing software components, building them, and running the regression test suite. Agile advocates frequent integration cycles. The definition of frequent varies widely in the literature, with some advocates suggesting integrating every couple of hours. The appropriate frequency for integration depends on the time and effort it takes to integrate the various pieces that make up the software system, but in general, integrating every one or two days is a good middle ground to avoid wasting too much time and also avoid ending up with incompatible components that are too hard to integrate.

There are many tools to aid development teams manage their development efforts, but at the bare minimum, every team should use a version control system and publish their work as soon as a functioning piece of it is complete.

Accompany New Code with New Tests

The importance of tests in the creation of reliable and robust software is not new and cannot be overstated. As with coding standards, Agile's contribution has been to emphasize even more the key role of tests in software development. So much so that some Agilists suggest what is known as Test-driven Development in which tests are written before the code, and the completion and correctness of a function is based on whether it passes all tests, and no new development can begin until the current function does.

In practice, it is sufficient to accompany all new code with corresponding tests; whether the test is written before or after the code is not as signifficant. It is also important not to start new work until the current functionality is complete, has passed all tests, and has been integrated.

What is Scrum?

Scrum is an Agile method that focuses on the organizational aspects of the software engineering process. It is currently one of the most popular Agile methods.

Key Scrum Practices

Scrum's popularity is due significantly to its specific definition of iterative development through its iteration model: the sprint. The following are the key Scrum practices followed during each iteration. These practices are becoming industry standards and you will become familiar with them during this class as the semester's schedule has been designed around them.

Sprint Planning

At the beginning of each sprint there should be a meeting to plan the iteration. This meeting, which may last several hours, is reserved for the team and the product owner only. Refer to the Scrum Roles sections for a description of each role.

The purpose of this meeting is to specify the goal of the sprint, the sprint backlog, and the list of acceptance critera.

The sprint goal succintly describes what the team plans to accomplish by the end of the sprint. All the tasks carried out during the spring should contribute to materializing this goal.

The collection of all refined user stories and pending defects is referred to as the product backlog. During the sprint planning, the product owner selects a subset of the user stories from the product backlog based on their business value, and then the team decomposes them into tasks and estimates the cost of implementing each task. The collection of these tasks is known as the sprint backlog. The team is also responsible for assigning the tasks to specific members for their implementation. This assignment takes place outside of the meeting.

The list of acceptance criteria establishes a quantifiable method for determining if the produced piece of functionality meets the client's expectations.

Closed-window Rule

One of Scrum's main contributions is the so called Closed-window Rule, which specifies that no one, regardless of rank, can change requirements while an iteration is in progress. This rule is a practical approach to the Agile principle of welcoming change. Although changes are allowed during the software development cycle, the progress of the current iteration cannot be disrupted.

This rule is sustainable because the iterations are short, making the rejections only temporary. It also gives people the opportunity to reconsider the pertinence and feasibility of their request for new or modified functionality.

Daily Scrum

A daily meeting at the beginning of every working day is a core Agile practice. Known as the "daily scrum" in the Scrum terminology, the goals of this meeting are to keep track of the progress and identify impediments to be addressed later. This meeting is reserved for the team and the Scrum Master. Refer to the Scrum Roles sections for a description of each role.

The focus of the meeting, which is supposed to be as short as possible, is to have each team member concisely answer three questions: What did you do on the previous working day? What will you do today? What impediments are you facing? Far from being a status update where the boss puts on spot those members behind schedule, the daily scrum represents an opportunity for team members to make commitments to each other.

For this class, where team members typically do not share a physical working space during the entire duration of the project, a face-to-face daily meeting may not be feasible. Consequently, every team will use an online tool to hold the meetings and record the answers to three aforementioned questions. Details on this will be discussed in a separate tutorial.

Definition of "Done"

To accurately measure progress the team must agree on the precise meaning of "done" and apply it consistently to all the ongoing tasks. A good example of a definition of "done" is "unit- and integration-tested; ready for acceptance test; deployed on demo server."

Task Board and Burndown Chart

The task board and the burndown chart complement each other to show the progress of the project.

A task board, which can be a virtual or a material board, contains all the tasks planned in the current iteration represented as cards and grouped in columns reflecting their status. As the work progresses, the team members move the cards through the columns indicating changes in their status. The number of states a task can be in may differ depending on the granularity the team wants to capture, but, in general, most teams use four states: To do, In progress, Under test, and Done.

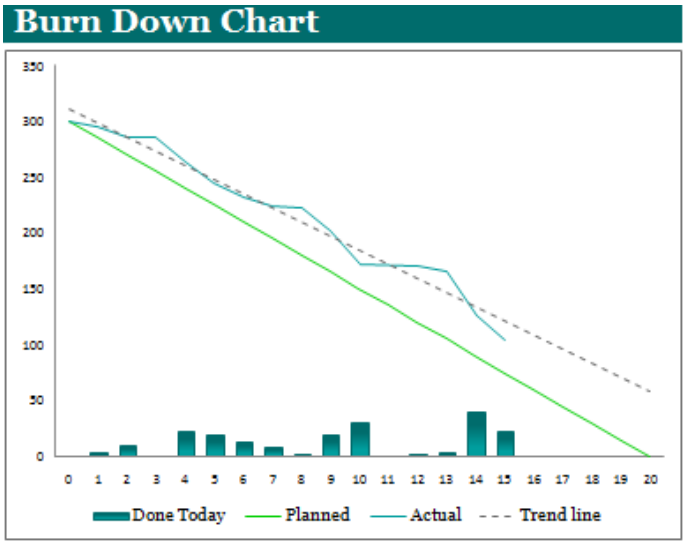
The burndown chart illustrates how fast the project progresses, and more importantly, it provides an estimate of the overall completion time based on the team's velocity. The x-axis represents the sprint (or project) timeline and the y-axis the amount of work that needs to be completed in story points. See below for an example of a burndown chart[1] in which 300 story points are to be completed in a 20-day sprint.

Anyone interested in knowing whether the project is on schedule needs only to look at the

1 Chart taken from http://www.expertprogrammanagement.com/2011/08/burn-down-chart-template/.

burndown chart and compare the “ideal work remaining line” (green line in the chart below) with the “actual work remaining line” (blue line in the chart below). If the blue line is above the green line, then there is more work left than initially predicted and the project is behind schedule; otherwise, the project is on or ahead of schedule.

In order to know the overall estimated completion time, one needs to look at the trend line (dashed line in the chart below), which averages out the actual work line data.



Example of burndown chart

Sprint Review

At the end of the sprint, the team meets with the product owner to present what was achieved and what was not achieved in the iteration considering the initial goals, cost estimates, and acceptance criteria. This meeting, which is focused on results and not on process, is a good opportunity to review the product backlog to make any pertinent changes.

Sprint Retrospective

After the sprint review, the team and the Scrum Master hold a separate meeting to discuss what went well and not so well during the sprint to make any necessary improvements for the next iteration. This meeting is known as the the sprint retrospective and the product owner may participate.

Scrum Roles

Agile redefines the roles of managers, clients, and the development team, and Scrum goes even further by not including the manager role at all. In Scrum, there are only three roles: the Product Owner, the Scrum Master, and the Team.

Product Owner

The product owner represents the client and has enough domain knowledge to answer the questions the development team may have about the software product. The main responsibilities of the product owner are to create and maintain the product backlog as well as to decide which functionalities are to be implemented first based on their business value.

The product owner is involved at the start and at the end of each sprint. At the start, to select the user stories to be implemented, and at the end, to evaluate the result of the sprint. The responsibilities of the product owner role overlap somewhat with those of a traditional manager in the sense that it prioritizes tasks, but the responsibilities notably do not include the assignment of tasks to specific members and other details of the team's day-to-day operations.

Scrum Master

In essence, the Scrum Master is a facilitator. This role guides the team in the application of the Scrum methodology and removes impediments identified during the daily scrum. It also takes care of protecting the team from distractions and interference from the rest of the organization so as to allow the development to proceed without unnecessary hurdles.

The Scrum Master may or may not be a member of the development team.

Team

The team in Scrum, and in Agile in general, takes over many of the traditional manager reposibilities including the assignment of tasks and their implementation details. It is a self-organizing entity that relies on peer pressure to maintain focus and accountability.

Source

The contents of this tutorial are largely based on the book *Agile!: The Good, the Hype and the Ugly* by Bertrand Stabley (ISBN 978-3-319-05154-3). This tutorial is also based on its authors' professional experience and on discussions carried out by graduate students during the Advanced Software Engineering class tought by Dr. Masoud Sadjadi in Fall 2014.