User Stories and Tasks

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 5828 — Lecture 9 — 02/10/2009

© University of Colorado, 2009
Goals

- Review material from Chapter 4 of Pilone & Miles
  - Tasks
  - Big Board & Burn Down Rate
  - Standup Meetings
- Supplementary Material
  - Agile Methods: Philosophy, Background, Techniques, & Extreme Programming
Once you and your customer have
- defined Milestone 1.0 (via user stories)
- and agreed on a deadline
And once you have
- developed an iteration plan that keeps in mind the number of people on your team and team velocity
You are ready to work!
- This chapter discusses what can happen during the first couple of iterations and what practices you should be following
First Task? Create Tasks

- User stories are written from the customer point of view
  - This is great for developing a shared understanding with your customer but not so great for guiding design and development
- To make progress, each user story needs to be split into tasks
  - Each task then needs an estimate associated with it
  - The entire team should participate in breaking a user story into tasks; planning poker should be used to assign estimates
Note: the example application in this chapter concerns a “date planner” as in “Jack and Jane go on a date” not “Jack uses this application to plan his week”

- The example assumes a U.S. perspective with respect to the norms associated with dating
  - And so may not be culturally sensitive (or even make sense) from the perspective of our international students

- Note: this is not an apology, simply an acknowledgment

- As such, I will try to keep the discussion focused on software engineering issues as much as possible
User Story: Create a Date in the System
  Estimate: 11 days

Tasks
  Create a date class that contains events: 3 days
  Create user interface to create, view and edit a date: 5 days
  Create the schema for storing dates in a database: 3 days
  Create SQL scripts for adding/finding/updating dates: 2 days

Total Task Time: 13 days!
Problem: Task ≠ Story

- Our task estimate did not equal our story estimate
  - The tasks are much more specific than the stories and may reveal additional work and/or assumptions in planning poker than the more abstract user story
- Now they tell us!
  - As a result, the book recommends that we
    - perform task decomposition during requirements gathering
    - always play planning poker with respect to tasks, not stories
- This will lead to more accurate estimates and iter. plans
Fortunately, the burn-down chart gives us a specific action item whenever an estimate changes or work gets done

- Update the burn-down chart

In the case of an estimate changing, calculate its impact on the work remaining and plot your status

- In the book, the original estimate for the iteration was 43 days of productive work; a 2 day increase in the first story pushes the amount of work left to 45 days
  - and they spent a day working on task decomposition

The following chart contains this info. plus more
The Big Board is a major feature of your team’s workspace.

- It is updated at least once per day during the stand up meeting (discussed next).
- But could be useful to update it more often than that.
- It is a one-stop shop for getting a “big picture” view of the current iteration.
<table>
<thead>
<tr>
<th>User Stories</th>
<th>In Progress</th>
<th>Complete</th>
<th>Burn Down</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Next</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Start with this...</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Title: Show Current Deals</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong> The website will show current deals to Orion’s Orbits users.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Title: Book package</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong> An Orion’s Orbits user will be able to book a special package with extras online.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Title: Pay online</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong> An Orion’s Orbits user will be able to pay for their bookings online.</td>
</tr>
</tbody>
</table>

Completed
<table>
<thead>
<tr>
<th>User Stories</th>
<th>In Progress</th>
<th>Complete</th>
<th>Burn Down</th>
</tr>
</thead>
</table>
| **Title:** Book package  
**Description:** An Orion’s Orbits user will be able to book a special package with extras online. |
| | | | ![Burn Down Chart] |
| **Title:** Pay online  
**Description:** An Orion’s Orbits user will be able to pay for their bookings online. |
| | | | |
| **Title:** Show Current Deals  
**Description:** The website will show current deals to Orion’s Orbits users. |
<p>| | | | |
| | | | |
| And add user stories |
| one per swim lane |
| Next |
| Completed |</p>
<table>
<thead>
<tr>
<th>User Stories</th>
<th>In Progress</th>
<th>Complete</th>
<th>Burn Down</th>
</tr>
</thead>
</table>
| Title: Book package  
Description: An Orion's Orbits user will be able to book a special package with extras online. |
| Title: Pay online  
Description: An Orion's Orbits user will be able to pay for their bookings online. |
| Title: Show Current Deals  
Description: The website will show current deals to Orion's Orbits users. |

Now add tasks each with a description and estimate
User Stories

<table>
<thead>
<tr>
<th>Title</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Book package</td>
<td>An Orion’s Orbits user will be able to book a special package with extras online.</td>
</tr>
<tr>
<td>Pay online</td>
<td>An Orion’s Orbits user will be able to pay for bookings online.</td>
</tr>
<tr>
<td>Show Current Deals</td>
<td>The website will show current deals to Orion’s Orbits users.</td>
</tr>
</tbody>
</table>

Tasks that a developer is working on move to “In Progress”
### User Stories

<table>
<thead>
<tr>
<th>Title</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Book package</td>
<td>An Orion's Orbits user will be able to book a special package with extras online.</td>
</tr>
<tr>
<td>Pay online</td>
<td>An Orion's Orbits user will be able to pay for their bookings online.</td>
</tr>
<tr>
<td>Show Current Deals</td>
<td>The website will show current deals to Orion's Orbits users.</td>
</tr>
</tbody>
</table>

If you stop working on a task, it goes back to its user story.
<table>
<thead>
<tr>
<th>User Stories</th>
<th>In Progress</th>
<th>Complete</th>
<th>Burn Down</th>
</tr>
</thead>
</table>
| **Title:** Book package  
**Description:** An Orion’s Orbits user will be able to book a special package with extras online. | ![In Progress](progress_icon) | ![Complete](complete_icon) | ![Burn Down Chart](burn_down_chart) |
| **Title:** Pay online  
**Description:** An Orion’s Orbits user will be able to pay for bookings online. | ![In Progress](progress_icon) | ![Complete](complete_icon) | |
| **Title:** Show Current Deals  
**Description:** The website will show current deals to Orion’s Orbits users. | ![In Progress](progress_icon) | ![Complete](complete_icon) | |

Tasks that get finished move to the “Complete” section of the swim lane and more tasks get started.
Be sure to update the Burn Down chart as you make (or don’t make) progress
User Stories

<table>
<thead>
<tr>
<th>In Progress</th>
<th>Complete</th>
<th>Burn Down</th>
</tr>
</thead>
</table>

Title: Book package
Description: An Orion’s Orbits user will be able to book a special package with extras online.

Title: Pay online
Description: An Orion’s Orbits user will be able to pay for their bookings online.

Title: Show Current Deals
Description: The website will show current deals to Orion’s Orbits users.

Next

Completed

It’s okay to be working on more than one task for a single user story.
Eventually all tasks for a user story are complete; the whole story moves to the Completed section.
### User Stories

#### Title: Pay online
**Description:** An Orion’s Orbits user will be able to pay for their bookings online.

#### Title: Show Current Deals
**Description:** The website will show current deals to Orion’s Orbits users.

### In Progress

### Complete

### Burn Down

### Next

### Completed

If a user story gets bumped (for whatever reason); move it to the Next section.
<table>
<thead>
<tr>
<th>User Stories</th>
<th>In Progress</th>
<th>Complete</th>
<th>Burn Down</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Keep working until all stories are complete or have been pushed to the next iteration**
### User Stories

**Title:** Book package  
**Description:** An Orion’s Orbits user will be able to book a special package with extras online.

**Title:** Pay online  
**Description:** An Orion’s Orbits user will be able to pay for their bookings online.

**Title:** Show Current Deals  
**Description:** The website will show current deals to Orion’s Orbits users.

---

**Don’t forget to update the Burn Down chart with your final status**
Standup Meeting

- A daily meeting used to
  - keep the team motivated and aware of progress (or not)
  - keep your board up-to-date
  - highlight problems early

- It should
  - Track progress, update burn-down rate, update tasks, discuss what happened yesterday and plan today’s activities, bring up issues, and last between 5 and 15 minutes

- “It’s so short, no one has time to sit down”
Design Issues

- In the example, one of the issues raised at a standup meeting involved the design of the system.
  - In particular, one developer was having problems with an unwieldy design that needed to be updated in lots of different places when a change request came in.

- We’ll look at this design problem in more detail in lecture 11.
  - In the meantime, take a look at the Appendix 1 for a refresher on the notation used to present/discuss this problem.
The example in the book also showed an unexpected request come in from the client

- The CEO of iSwoon wants the developers to demo the system to the CEO of Starbuzz who is interested in integrating his beverage-related services into iSwoon

What to do? Add the unplanned task as a new task to this iteration and update the burn-down rate

- Unplanned tasks become new user stories that have to be integrated into the current iteration, if at all possible
Velocity may help

- Velocity builds a little flexibility into the schedule
  - 3 developers working 20 days can theoretically get 60 days worth of work done
    - That’s not realistic, so we add in velocity: $3 \times 20 \times 0.7 = 42$
    - However, if we are more productive than our velocity accounted for, then we have “float” or “slack” in the schedule
      - In this case, we have up to 18 days of float time (60 - 42)
      - So, one or two small unplanned tasks may not upset the iteration
    - But, remember, you’ll be burning through float naturally, so this is not a panacea
Successful software development is about knowing where you are

- All of these practices, add certainty to the development process
  - You may be behind, but at least you KNOW you’re behind
- Armed with this information, you can make better decisions about what to do next
- This, in turn, gives you increased confidence which increases your odds at success
Our textbook is teaching an agile approach to software development

- Lets look at the philosophy behind Agile and examine an Agile life cycle known as Extreme Programming
  - The material for this supplement is based on content from “Agile Software Development: Principles, Patterns, and Practices” by Robert C. Martin
    - As such, some of this material is copyright © Prentice Hall, 2003

- Note: some of this material is review
  - We’ll skim quickly over duplicated material
(Very) Briefly introduce the concepts of Agile Design and Extreme Programming

Agile Design is a design framework

Extreme Programming is one way to “implement” agile design

Other agile life cycles include SCRUM, Crystal, feature-driven development, and adaptive software development

See http://www.agilealliance.org/ for pointers
Agile development is a response to the problems of traditional “heavyweight” software development processes:
- too many artifacts
- too much documentation
- inflexible plans
- late, over budget, and buggy software
A manifesto (from the Agile Alliance)

“We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value

- individuals and interactions over processes and tools
- working software over comprehensive documentation
- customer collaboration over contract negotiation
- responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more”
From this statement of values, agile development has identified twelve principles that distinguish agile practices from traditional software life cycles.

Let's look at five of them:

- Deliver Early and Often to Satisfy Customer
- Welcome Changing Requirements
- Face to Face Communication is Best
- Measure Progress against Working Software
- Simplicity is Essential

- Strong correlation between quality of software system and the early delivery of a partially functioning system:
  - The less functional the initial delivery, the higher the quality of the final delivery!

- Strong correlation between final quality of software system and frequent deliveries of increasing functionality:
  - The more frequent the deliveries, the higher the final quality!

- Customers may choose to put these systems into production use or simply review and provide feedback.
Welcome Changing Requirements

- Welcome change, even late in the project!
- Statement of Attitude
  - Developers in agile projects are not afraid of change; changes are good since it means our understanding of the target domain has increased
  - Plus, agile development practices (such as refactoring) produce systems that are flexible and thus easy to change
In an agile project, people talk to each other!

- The primary mode of communication is conversation
  - there is no attempt to capture all project information in writing
- artifacts are still created but only if there is an immediate and significant need that they satisfy
  - they may be discarded, after the need has passed
Agile projects measure progress by the amount of software that is currently meeting customer needs.

- They are 30% done when 30% of required functionality is working AND deployed.

- Progress is not measured in terms of phases or creating documents.
This refers to the art of maximizing the amount of work NOT done

- Agile projects always take the simplest path consistent with their current goals
- They do not try to anticipate tomorrow’s problems; they only solve today’s problems
- High-quality work today should provide a simple and flexible system that will be easy to change tomorrow if the need arises
The other seven principles are

- Deliver working software frequently
- Stakeholders and developers work together daily
- Build projects around motivated individuals
- Agile processes promote sustainable development
- Continuous attention to technical excellence and good design enhances agility
- Agile team members work on all aspects of the project
- At regular intervals, the team reflects on how to become more effective
Extreme Programming (XP) takes commonsense software engineering principles and practices to extreme levels.

For instance:

- “Testing is good?”
- then
- “We will test every day” and “We will write test cases before we code”

As Kent Beck says extreme programming takes certain practices and “sets them at 11 (on a scale of 1 to 10)”
The best way to describe XP is by looking at some of its practices.

There are fourteen standard practices:

- Customer Team Member
- User Stories
- Short Cycles
- Acceptance Tests
- Pair Programming
- Test-Driven Development
- Collective Ownership

- Continuous Integration
- Sustainable Pace
- Open Workspace
- The Planning Game
- Simple Design
- Refactoring
- Metaphor
Customer Team Member

- The “customer” is made a member of the development team
  - The customer is the person or group who defines and prioritizes features
  - A customer representative should be “in the same room” or at most 100 feet away from the developers
  - “Release early; Release Often” delivers a working system to the client organization; in between, the customer representative provides continuous feedback to the developers
User Stories (I)

- We need to have requirements
- XP requirements come in the form of “user stories” or scenarios
  - We need just enough detail to estimate how long it might take to support this story
    - avoid too much detail, since the requirement will most likely change; start at a high level, deliver working functionality and iterate based on explicit feedback
User stories are not documented in detail

- we work out the scenario with the customer “face-to-face”; we give this scenario a name
- the name is written on an index card
  - developers then write an estimate on the card based on the detail they got during their conversation with the customer

The index card becomes a “token” which is then used to drive the implementation of a requirement based on its priority and estimated cost
Short Cycles (I)

- An XP project delivers working software every two weeks that addresses some of the needs of the customer.
- At the end of each iteration, the system is demonstrated to the customer in order to get feedback.
Short Cycles (II)

- **Iteration Plan**
  - The collection of user stores that will be implemented during this iteration
  - determined by a “budget” of points
  - the budget is determined by the progress made on the previous iteration

- **Release Plan**
  - A plan that maps out the next six iterations or so (3 months)
  - A release is a version of the system that can be put into production use
Acceptance Tests

- Details of a user story are captured in the form of acceptance tests specified by the customer
  - The tests are written before a user story is implemented
  - They are written in a scripting language or testing framework that allows them to be run automatically and repeatedly
  - Once a test passes, it is never allowed to fail again (at least for very long)
  - These tests are run several times a day each time the system is built
Pair Programming (I)

- All production code is written by pairs of programmers working together at the same workstation
  - One member drives the keyboard and writes code and test cases; the second watches the code, looking for errors and possible improvements
  - The roles will switch between the two frequently
- Pair membership changes once per day; so that each programmer works in two pairs each day
  - this facilitates distribution of knowledge about the state of the code throughout the entire team
Studies indicate that pair programming does not impact efficiency of the team, yet it significantly reduces the defect rate!

- [Laurie Williams, 2000] [Alistair Cockburn, 2001] [J. Nosek, 1998]
Test-Driven Development

- All production code is written in order to make failing test cases pass
  - First, we write a test case that fails since the required functionality has not yet been implemented
  - Then, we write the code that makes that test case pass
  - Iteration between writing tests and writing code is very short; on the order of minutes
- As a result, a very complete set of test cases is written for the system; not developed after the fact
Collective Ownership

- A pair has the right to check out/improve ANY module
  - Developers are never individually responsible for a particular module or technology
- Contrast this with Fred Brook’s conceptual integrity and the need for a small set of “minds” controlling a system’s design
  - Apparent contradiction is resolved when you note that XP is designed for use by small programming teams; I haven’t seen work that tries to scale XP to situations that require 100s or 1000s of developers
Continuous Integration

- Developers check in code and integrate it into the larger system several times a day.
- Simple Rule: first one to check-in “wins”; everyone else merges.
- Entire system is built every day; if the final result of a system is a CD, a CD is burned every day; if the final result is a web site, they deploy the web site on a test server, etc.
  - This avoids the problem of cutting integration testing to “save time and money”
A software project is not a sprint; it’s a marathon

- A team that leaps off the starting line and races as fast as it can will burn out long before the finish line
- The team must instead “run” at a sustainable pace

An XP rule is that a team is not allowed to work overtime

- This is also stated as “40 hour work week”
Open Workspace (I)

- The team works together in an open room
  - There are tables with workstations
  - There are whiteboards on the walls for the team members to use for status charts, task tracking, UML diagrams, etc.
- Each pair of programmers are within earshot of each other; information is communicated among the team quickly
  - “War room” environments can double productivity
Joel on Software disagrees

The Planning Game (I)

- Customer decides how important a feature is
- Developers decide how much that feature costs
- At the beginning of each release and/or iteration, developers give customers a budget based on productivity of previous iteration
The Planning Game (II)

- Customers choose user stories whose costs total up to but do not exceed the budget.
  - The claim is that it won’t take long for customer and developers to get used to the system.
  - And then the pace can be used to estimate cost and schedule.
Simple Design

- An XP team makes their designs as simple and expressive as they can be
  - They narrow focus to current set of stories and build the simplest system that can handle those stories

- Mantras
  - Consider the Simplest Thing That Could Possibly Work
  - You Aren’t Going to Need It
  - Once and Only Once (aka Don’t Repeat Yourself)
XP teams fight “code rot” by employing refactoring techniques constantly.

- They have the confidence to do this because they also use test-driven design.
- By “constantly” we mean every few hours versus “at the end of the project”, “at the end of the release”, or “at the end of the iteration”.
Metaphor (I)

- The big picture that ties the whole system together
  - Vocabulary that crystallizes the design in a team member’s head
Example

A system that transmits text to a screen at 60 chars per second; programs write to buffer, when buffer full, programs are suspended, when buffer empty, programs are activated

Metaphor: Dump Trucks Hauling Garbage

Screen = “Garbage Dump”, Buffer = “Dump Truck”, Programs = “Garbage Producer”
Example

- network traffic analyzer, every 30 minutes, system polled dozens of network adapters and acquired monitoring data; Each adaptor provides block of data composed of several variables
  - Metaphor: A toaster toasting bread
  - Data Block = “Slices”
  - Variables = “Crumbs”
  - Network analyzer = “The Toaster”
  - Slices are raw data “cooked” by the toaster
Benefits of XP

- Customer Focus
- Emphasis on teamwork and communication
- Programmer estimates before implementation
- Emphasis on responsibility for quality
- Continuous measurement
- Incremental development

- Simple design
- Frequent redesign via refactoring
- Frequent testing
- Continuous reviews via pair programming
Criticisms of XP

- Code centered vs. Design centered
  - Hurts when developing large systems
- Lack of design documentation
  - Limits XP to small systems
- Producing readable code is hard
- Code is not good documentation
- Lack of structured inspection process (can miss defects)

- Limited to narrow segment of software application domains
- Methods are only briefly described
- Difficult to obtain management support
- Lack of transition support (how do you switch from waterfall or other process?)
Coming Up

- Lecture 10: Shared Objects & Mutual Exclusion
  - Chapter 4 of Magee and Kramer
- Lecture 11: Good Enough Design
  - Chapter 5 of Pilone & Miles