Project Management Tips

Software Engineering II
WS 2018/19

Christoph Matthies (christoph.matthies@hpi.de)

Enterprise Platform and Integration Concepts
1. Value-based Software Engineering
   - Requirements Prioritization
   - Minimum Viable Product
2. Organizing your Project
3. Git Tricks
Value-based Software Engineering

- "Requirements are often analyzed in a value-neutral environment" [1]
  - "Earned value" systems track project cost and schedule, not stakeholder value
  - "separation of concerns": developers only turn requirements into verified code
- 80% of the value is expressed in 20% of the requirements (Pareto principle) [2]
- A value-oriented approach is more appropriate

How to do that?
- Identify the system's success-critical stakeholders
- Obtain their value propositions with respect to the system
- Estimate / find out value of a requirement to the stakeholders
- Estimate effort to implement a requirement

MoSCoW Prioritization

Reach **common understanding** with stakeholders on the importance of delivering each requirement

**MoSCoW: Must have, Should have, Could have, and Won't have**
- Description instead of *high, med* and *low*
- Get customers to better understand the impact of setting a priority
- Try to deliver all the *Must haves, Should haves* and *Could haves*
- *Should haves* and *Could haves* will be removed first if plan for delivery is threatened

MoSCoW Prioritization

Must have
- Critical for success of delivery
- If only a single Must have is missing, project delivery is considered a failure

Could have
- Desirable, but not necessary. Included if time and resources permit
- Could improve customer satisfaction for little development cost

Should have
- Important, but not necessary for delivery in the next iteration
- Can be as critical as Must haves, maybe not as time-sensitive or workaround exist

Won't have (this time)
- Lowest-payback items or not appropriate at this time
- Not planned into the schedule for the next delivery. Outside of current scope.
- Either dropped or reconsidered for inclusion in a later timebox

MoSCoW Prioritization

Criticism

- Lack of reason
  - Why is a requirement *Must have* and not *Should have*?
  - Why is this requirement *Won't have*?
- Lack of time information
  - Are *Won't have* requirements not in this delivery or never?
- Dealing with technical debt
  - What priority does refactoring have?
  - What about bux fixes?

Value-based Requirements Prioritization

Idea
- Plot requirements on the dimensions of value and effort
- Implement: Above 2x
- Skip: Below 1/2x
- In-between: Review

Challenges
- Whole truth?
- Beware of dependencies!
- Keep in sync
Value-based Requirements Prioritization

Lean Startup 2x2 Matrix
- Do first: Quick Wins
- Do second: Big Bets
- Think about Maybes
- Try to avoid Time Sinks

Minimum Viable Product (MVP)

Product with just enough features to satisfy early customers, and to provide feedback for further development.
MVP (Dis-)Advantages

Advantages

■ **Early user feedback**
  □ Test initial understanding of user needs, test product hypothesis
  □ Limited resources spent on MVP

■ Move into production early
  □ Software is developed for a reason, solve a problem!
  □ Generate revenue
  □ Entering a market first can be a competitive advantage

Challenges

■ Definition of *minimally viable* (*why?*)
  □ *Smallest possible* way to meet the market need with a *useful output*
  □ Requires smart requirements management

■ Requires early focus on usability, deployment, support, marketing
"Minimum Viable Product" is used in many contexts. Some possible variants:

- Marketing MVP
  - Product to test the market that is being targeted
  - Check demand assumptions

- Technical Demonstration MVP
  - Prototype or proof-of-concept
  - Explore software designs
  - Prove that it will work using the technology

- "Must-Haves" MVP
  - Product with only "the most important features"
  - Might not be truly minimal in terms of effort
  - Smaller version of full software? Is the main goal feedback collection?
Agenda

1. Value-based Software Engineering

2. Organizing your Project
   - Scrum Burn-Down Chart
   - Communication
   - Dealing with Dependencies
   - Estimating Large Backlogs
   - Beyond Scrum

3. Git Tricks

4. Outlook
Organizing your Project

Questions:
■ Which stories are part of Sprint#1?
■ Who is working on which tasks?
■ Which version is a good one that can be shown to the customer?

Tools that might help:
■ Put your user stories & tasks into Github's issue tracker
  □ Assign issues to developers
  □ Use milestones to assign user stories to sprints
  □ Use issue tags, e.g. to denote responsible teams or status
  □ Use "project management" tools that give an overview of GH issues, e.g. https://waffle.io/ or https://www.zenhub.io/
■ Tag versions that can be presented
  
  $ git tag --a v0.1 --m 'version after Sprint#1 without US #2'

Side note:
When assigning tickets to devs it's helpful if usernames are identifiable (or there is some info on the profile).
"Who is ,gronkh12‘ again?"

Tips & Tricks — Software Engineering II
Scrum Burn-Down Chart

- Graphical representation of work left to do versus time
- X-Axis: sprint timeline, e.g. 10 days
- Y-Axis: work that needs to be completed in sprint (time or story points)
- "Ideal" work remaining line: straight line from start to end
- Actual work remaining line
  - above ideal → behind schedule, below ideal → ahead schedule
Scrum Boards – Virtual vs. Real-Life
Definition of Done

How do I know when to stop?
- Acceptance criteria fulfilled
- All tests are green
- Code looks good
- Objective quality goals
- Second opinion
- Internationalization
- Security
- Documentation

The Definition of Done is the team’s consensus of what it takes to complete a feature.
Definition of Ready

- Similar to Definition of Done, but for user stories
- Answer the question: **When is a user story ready for implementation?**

**Examples**
- Estimated
- Acceptance criteria
- Mockups for UI stories
Communication

Questions:

■ How do we communicate in and between teams?
■ How do I find out about architecture changes?
■ How do I know how to use other people's code?

Tools that might help:

■ Github wiki to (briefly!) document how to use components
■ Code comments explaining the larger context, common pitfalls
■ One(!) common communication channel for announcing changes, e.g. E-Mail list, IRC, IM, Slack, Google Hangouts, Facebook group
Dealing with Dependencies

Ambassadors

- Mutual Exchange of team members
  - Improves efficiency of communications
  - Allows deeper understanding of problems
  - Prevents coordination problems early in the process
- Ambassadors should be fully integrated team members
- Especially useful for API development, design, etc.
Dealing with Uncertainty

Spikes

What can we do if no team members lack knowledge in a particular domain?

- Hard to estimate with little knowledge
- Take time out of the sprint to research and learn
- Spike
- For example, evaluate new technologies
Bucket Estimation (Jukka Lindström) [Scrumcenter, 2009]

- Create physical buckets based on examples (2-3 per bucket)
- Assign items to buckets one by one through
  - Comparing & discussing
  - Planning Poker
Affinity Estimation (Lowell Lindstrom) [Scrumcenter, 2009]

- Read each story to the entire team
- Arrange stories horizontally based on size (no talking!)
- Place Fibonacci numbers above the list
- Move each story to the preferred number
Beyond Scrum

Scrum critique:

- Scrum and agile are by no means universally accepted as "the way" to do software engineering ("Agile Hangover")
- Michael O. Church - Why “Agile” and especially Scrum are terrible (2015)
  https://michaelochurch.wordpress.com/2015/06/06/why-agile-and-especially-scrum-are-terrible/
  - Business-driven engineering — Scrum increases the feedback frequency while giving engineers no real power
  - Terminal juniority — Architecture and R&D and product development aren’t part of the programmer’s job
  - It’s stupidly, dangerously short-term — engineers are rewarded or punished solely based on the completion, or not, of the current two-week “sprint”
Agenda

1. Value-based Software Engineering
2. Organizing your Project
3. Git Tricks
Git Tricks — amend, interactive staging

Change commit message of previous commit
(Careful, don't do this if you already pushed the commit)

```
$ git commit --amend -m "new message"
```

Forgot to commit files?

```
$ git add [missing files]
$ git commit --amend #uses the previous commit's message
```

Undo the amending

```
$ git reset --soft HEAD@{1}
$ git commit -C HEAD@{1}
```

Interactive staging (also allows committing only parts of files)

```
$ git add -i
$ git add --patch [file]
```

Opinion:
Interactive staging (git add -i) is probably the most powerful git feature you're not using yet.
Git Tricks — reflog, diff, stash

Log of all recent actions

$ git reflog

What did I work on recently?
Show differences that are not staged yet

$ git diff

Shows differences between staging and the last file version

$ git diff --staged

Temporarily store/retrieve all modified tracked files

$ git stash
$ git stash pop

List all stashed changesets

$ git stash list

Tip:
$ git stash is often helpful if you don't want to directly commit your changes, but need to checkout another branch/commit.
Git Tricks — log, blame, rebase

Shorter version of the git log

$ git log --abbrev-commit --pretty=oneline

Show pretty graph of git history

$ git log --graph --decorate --pretty=oneline --abbrev-commit

Show changesets in the log

$ git log -p

Show what revision and author last modified each line

$ git blame --date=short [file]

History is becoming cluttered with merge commits

$ git rebase <branch>

Warning:
Do not rebase commits that others have worked with!
"people will hate you, and you’ll be scorned by friends and family."
Git Rebase — setup

- Created "experiment" branch to try something out

  
  $ git checkout –b "experiment"
  $ git commit –a –m "C3"

- Easiest way to integrate the branches is merge
  - Will create merge commits

  $ git checkout master
  $ git merge experiment

**Git Rebase — execution**

- **git rebase**
  - Take all the changes that were committed on one branch and replay them on another one
  - Only do this with local commits

  ```
  $ git checkout experiment
  $ git rebase master
  ```

- Afterwards: fast-forward the master branch
  - No merge commits

  ```
  $ git checkout master
  $ git merge experiment
  ```

[Diagram showing Git rebase execution process]

---

Git cherry-pick

- **Problem**: Quickly get changes from other commits without having to merge entire branches

- `git cherry-pick`
  - apply the changes introduced by existing commits

```bash
$ git checkout master
$ git log --abbrev-commit --pretty=oneline
d7ef34a  C3: Implement feature
0be778a  C4: critical change introduced

$ git checkout experiment
$ git cherry-pick 0be778a
```

Branching

Ideas

- Never merge in master or release branches
- Never break build in shared branches

http://nvie.com/posts/a-successful-git-branching-model/
Git Self-help Resources

- How to undo (almost) anything with git – guide by Github

- Git cheat sheet – by Github

- Git FAQ – answers to common questions
  - [http://gitfaq.org/](http://gitfaq.org/)
  - [https://git.wiki.kernel.org/index.php/Git_FAQ](https://git.wiki.kernel.org/index.php/Git_FAQ)

- Git pretty – troubleshooting flowchart
  - [http://justinhileman.info/article/git-pretty/](http://justinhileman.info/article/git-pretty/)
Tooling suggestions

  - Make some complex git interactions much simpler
  - Draw pretty commit graphs, overviews of branches and merges
  - *GitX, TortoiseGit, SourceTree, Tower, SmartGit, gitg, git-cola*

- Github Integration
  - Github also provides git tools
    - [https://mac.github.com/](https://mac.github.com/), [https://windows.github.com/](https://windows.github.com/)

- Git extras ([https://github.com/tj/git-extras](https://github.com/tj/git-extras))
  - Common git commands bundled