Recap: Software development process models

Traditional models
- Waterfall model
- Iterative and incremental
- Prototyping
- Spiral model

Agile models
- XP (Extreme Programming)

Same goals
- Manage risks and produce high quality software.

Same activities and steps
- E.g., specification, design, implementation, and testing.

Recap: How to choose an appropriate model?

Consider
- The project and task at hand.
- Risk management and quality/cost control.
- Customer involvement and feedback.
- Well-definedness of requirements.
- Experience of management and team members.

Project management triangle (pick any two)

Today

Another agile model
- Scrum

Introduction to version control systems (VCS)
- Centralized VCS
- Distributed VCS
- Git tutorial
Scrum: Overview

A time-boxed model
- Each sprint (time box): max 30 days.
- Fixed number of tasks for each sprint.
- Each sprint results in a sprint review.
- Small number of team members: 7 (+/- 2).
- Daily scrum meeting: 15 min max.
- Sprint review (product demo): 0.5-3 hours.
- Sprint retrospective (post-mortem): 1-4 hours.

Roles
- Product owner (customer)
- Scrum master (project manager)
- Scrum team (development team)

Prioritization:
- Must have vs. Should have vs. Could have vs. Won't have
Scrum: Activities and planning

**Daily scrum meeting (15min):**
- What did I do since the last meeting?
- Any obstacles or blocking issues?
- What will I do until the next meeting?

Scrum: Sprint planning

**Sprint review**

**Who and why?**
- Product owner, scrum master, scrum team, and potentially other developer teams and managers etc.
- Review sprint goals, achievements, and potentially adapt.

Scrum: Sprint retrospective

**Who and why?**
- Product owner, scrum master, and scrum team.
- Reflect, change, improve.

What went well? What needs to be improved? Action items
- Continue doing
- Stop doing
- Start doing

Introduction to version control
What is version control?

- Version control records changes to a set of files over time, making it easy to review or revert to specific versions later.
- Simple Example
  - Alice writes a research paper, using version control: v1.0
  - Alice corrects grammatical mistakes: v1.1
  - Alice discovers new findings and rewrites her paper: v1.2
  - Alice realizes her findings are wrong: revert back to v1.1

Why use version control?

- There exists one "central" copy of a project where all developers commit their changes to.
- Each developer has a working copy. As soon as she/he commits, the repository gets updated.
- Examples: Subversion, CVS.
Distributed version control

- Multiple clones of a repository exist.
- Each developer has access to a local (private) repository.
- All committed changes remain local unless pushed to another repository.
- No external changes are visible unless pulled from another repository.
- Examples: Mercurial (Hg), Git.

Conflicts

- Conflicts occur when two users make a simultaneous change to the same line of a file.
- When conflicts arise, the last committer needs to choose which line(s) to keep.

Conflicts: example

**Original (committed) program:**
```
#!/usr/bin/env python
print "Hello world!"
```

**Alice changes the program to:**
```
#!/usr/bin/env python
print "Hello, world!"
```

**Bob changes the program to:**
```
#!/usr/bin/env python
print "Hello world."
```

Bob commits last and gets the following:
```
#!/usr/bin/env python
<<<<<<<<<<< .mine
print "Hello world."
=======
print "Hello, world." 
>>>>>>> .r4
```

```
Conflicts: example

Bob commits last and gets the following:
```python
#/usr/bin/env python
<<<<<<< .mine
print "Hello world."
=======
print "Hello, world!"
>>>>>>> .r4
```

What the ...?
- `<<<<<<<<< and >>>>>>>>>` indicate a conflict.
- `=======` separates the versions.
- The top version is Bob’s local copy.
- The bottom version is the version that Alice committed (revision 4).

Options to resolve the conflict
- Keep mine, keep theirs, manually merge

Distributed version control with Git

Creating a local (empty) repository:
- `git init`

Cloning an existing repository:
- `git clone`

Making (local) commits:
- `git add <file or directory>`
- `git commit`

Communicating with the world:
- `git push <remote> <branch>`
- `git pull <remote> <branch>`
- `git fetch <remote> <branch>`

Relationship between local repository and remotes

Branches
- The master branch is the main development branch.
- To add a new feature, it’s useful to create a new branch -- an independent line of development.

Creating a branch:
- `git branch`

Checking out a branch:
- `git checkout <branch>`

Merging branches:
- `git merge <branch>`
- `git pull <remote> <branch>`
**Version control history**

Looking through the history:
- `git log [<identifier>]`
- Commits are identified by a unique identifier (commit hash).

Checking out an old version:
- `git checkout <identifier>`

Note that usually the first few characters (4-6) are sufficient to uniquely identify a commit.

**Git cheat sheet**

https://www.git-tower.com/blog/git-cheat-sheet/

**Step by step tutorial**

Set up:
1. Create a folder `cs320`.

In the cs320 folder:
1. Create a local (bare) repository: `git init --bare main_repo`
   (Alternatively, create an empty repository on GitHub/Bitbucket).
2. Clone the repository: `git clone main_repo clone1`
3. Clone the repository: `git clone main_repo clone2`

In the cs320/clone1 folder:
1. Create a new file with one line: `<editor> group.txt`
2. Check the status: `git status`
3. Add the new file: `git add group.txt`
4. Check the status: `git status`
5. Commit the new file: `git commit -m "Added group.txt"`
6. Push the changes to the main repo: `git push`

In the cs320/clone2 folder:
1. Pull all changes from the main repo: `git pull`
2. View the current history: `git log`
3. Add two new lines to the end of group.txt: `<editor> group.txt`
4. Check the status: `git status`
5. Commit the changes: `git commit group.txt -m "Added two new names"`
6. Push the changes to the main repo: `git push`

In cs320/clone1:
1. Add one new line at the beginning of group.txt: `<editor> group.txt`
2. Commit the changes: `git commit group.txt -m "Added a new name"`
3. Push the changes to the main repo (oops): `git push`
4. Pull (i.e., fetch + merge) all changes from the main repo: `git pull`
5. View the history after the automated merge: `git log --graph`
6. Push the changes to the main repo: `git push`
Step by step tutorial

Next steps
1. Create a conflict by editing the same line in clone1 and clone2, and resolve it.
2. Experiment with `git checkout` and `git branch`.

Resources

http://blogs.atlassian.com/2012/02/version-control-centralized-dvcs/
http://stackoverflow.com/questions/1057564/pretty-git-branch-graphs
https://www.git-tower.com/blog/git-cheat-sheet/
http://gitref.org/