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)
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
● Scrum master
● Scrum team
Scrum: Overview

Product Backlog → Sprint Backlog → Sprint → Working increment of the software

24 h
30 days
Scrum: Overview

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 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.

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

What went well?
What needs to be improved?
Action items

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

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

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

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

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

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

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

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

Who and why?
- Product owner, scrum master, and scrum team.
- Reflect, change, improve.
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?
Why use version control?
Centralized 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:**
```python
#/usr/bin/env python
print "Hello world!"
```

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

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

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:
```
#!/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

- **Local**
  - working directory
  - staging area
  - local repo

- **Remote**
  - remote repo

Actions:
- `git add`
- `git commit`
- `git push`
- `git fetch`
- `git checkout`
- `git merge`
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

- **CREATE**
  - Clone an existing repository:
    ```
    git clone ssh://user@domain.com/repo.git
    ```
  - Create a new local repository:
    ```
    git init
    ```

- **LOCAL CHANGES**
  - Changed files in your working directory:
    ```
    git status
    ```
  - Changes to tracked files:
    ```
    git diff
    ```
  - Add all current changes to the next commit:
    ```
    git add .
    ```
  - Add some changes in `<file>` to the next commit:
    ```
    git add -p <file>
    ```
  - Commit all local changes in tracked files:
    ```
    git commit -a
    ```
  - Commit previously staged changes:
    ```
    git commit
    ```
  - Change the last commit:
    ```
    Don't amend published commits!
    git commit --amend
    ```

- **COMMIT HISTORY**
  - Show all commits, starting with newest:
    ```
    git log
    ```
  - Show changes over time for a specific file:
    ```
    git log -p <file>
    ```
  - Who changed what and when in `<file>`:
    ```
    git blame <file>
    ```

- **BRANCHES & TAGS**
  - List all existing branches:
    ```
    git branch -av
    ```
  - Create a new branch on your current HEAD:
    ```
    git branch <new-branch>
    ```
  - Create a new tracking branch based on a remote branch:
    ```
    git checkout --track <remote/branch>
    ```
  - Delete a branch:
    ```
    git branch -d <branch>
    ```
  - Mark the current commit with a tag:
    ```
    git tag <tag-name>
    ```

- **UPDATE & PUBLISH**
  - List all currently configured remotes:
    ```
    git remote -v
    ```
  - Show information about a remote:
    ```
    git remote show <remote>
    ```
  - Add new remote repository, named `<remote>`:
    ```
    git remote add <shortname> <url>
    ```
  - Download all changes from `<remote>` but don't integrate into HEAD:
    ```
    git fetch <remote>
    ```
  - Download changes and directly merge/integrate into HEAD:
    ```
    git pull <remote> <branch>
    ```
  - Publish local changes on a remote:
    ```
    git push <remote> <branch>
    ```
  - Delete a branch on the remote:
    ```
    git branch -dr <remote/branch>
    ```
  - Publish your tags:
    ```
    git push --tags
    ```

- **MERGE & REBASE**
  - Merge `<branch>` into your current HEAD:
    ```
    git merge <branch>
    ```
  - Rebase your current HEAD onto `<branch>`:
    ```
    git rebase <branch>
    ```
  - Don't rebase published commit:
    ```
    git rebase <branch>
    ```
  - Abort a rebase:
    ```
    git rebase --abort
    ```
  - Continue a rebase after resolving conflicts:
    ```
    git rebase --continue
    ```
  - Use your configured merge tool to solve conflicts:
    ```
    git mergetool
    ```
  - Use your editor to manually solve conflicts and (after resolving) mark file as resolved:
    ```
    git add <resolved-file>
    git rm <resolved-file>
    ```

- **UNDO**
  - Discard all local changes in your working directory:
    ```
    git reset --hard HEAD
    ```
  - Discard local changes in a specific file:
    ```
    git checkout HEAD <file>
    ```
  - Revert a commit (by producing a new commit with contrary changes):
    ```
    git revert <commit>
    ```
  - Reset your HEAD pointer to a previous commit:
    ```
    ...and discard all changes since then
    git reset --hard <commit>
    ```
  - ...and preserve all changes as unstaged changes:
    ```
    git reset <commit>
    ```
  - ...and preserve uncommitted local changes:
    ```
    git reset --keep <commit>
    ```

---

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`
Step by step tutorial

In cs320/clone2:
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/