Development Infrastructure I

Stepan Bolotnikov
Who’s this guy

- Stepan Bolotnikov
- MSc Software Engineering from UT (2018)
- User Interface Engineer at Guardtime
- stepan.bolotnikov@guardtime.com
The next 2 lectures

- Development is more than just sitting down and writing code.
- Essential tools and technologies that empower developers to do their job.
- Increasing productivity, decreasing errors
- We cover just a small part
Today - “Development tools”

- **IDE (Integrated Development Environment)**
  - Software that helps you read, write, and interact with your code better
  - IntelliJ IDEA

- **VCS (Version Control Systems)**
  - Software that helps you collaborate, structure teamwork, and keep history
  - Git
Integrated Development Environment

- Integrates different parts of software development process into one software system.

- Typically has
  - Code editor
  - Debugger
  - Interpreter/compiler
  - Build automation tools
  - Version control system
Maximising coder productivity

- Tight-knit tools enable useful features
  - Intelligent code completion
  - Live debugging
  - Live error reporting
  - Class and method lookup
- Consistent user interface
Popular IDEs for Java

- NetBeans
- Android Studio
- Eclipse
- IntelliJ IDEA
IntelliJ IDEA Community Edition

- Open-source
- Free
- For JVM and Android development
- Paid version - € 149.00/year
  - Free for students and teachers
- JetBrains has developed IDEs for other languages and tech stacks
  - WebStorm
  - PyCharm
  - PhpStorm
  - RubyMine
Demo time

Let’s look at some often overlooked features that showcase the power of having tightly integrated tools.
Demo time

1. Reading
   a. Ctrl+Click to navigate
   b. Find class/symbol
   c. Finding IDE functions by name

2. Writing
   a. Smart autocomplete
   b. Auto importing types
   c. Refactoring

3. Debugging
   a. Console
   b. Breakpoints
      i. Step into
      ii. Smart breakpoints
   c. Force return
In conclusion

- IDEs have many useful features outside of just being code editors
- Whatever IDE you end up with, learn to be productive with it
Version Control System

- Over time, code in a project changes
- Need to deliver/share particular version of project
This doesn’t scale

● Progressively harder to manage
● What if you have collaborators?
● What if you want to go back to a certain state?
● How do you track changes?
VCS software

- Automates the heavy lifting behind version management
- Enables you to:
  - Track changes in data
  - View history of data
  - Revert to a previous state of data
- Not just for code
Terminology

- “Working copy” - current state of the data that you are working on
- “Repository” - a database containing the history
- “Revision” / ”commit (noun)” - the state of data at a given time
- “Check out” - to retrieve a revision from the repository
- “Check in” / “commit (verb)” - to send a revision to the repository
Revisions are connected to each other

- Every revision has: revision ID/number; message; author; time
- Every revision (except the first) has at least one parent revision that it builds upon.
- The evolution of the data can be observed by moving along those connections.

Revision 1 -> Revision 2 -> Revision 3 -> Revision 4
History needn’t be linear

- “Branch” - where the history diverges, 2+ revisions have a common parent
- “Merge” - where one branch connects with others, one revision has 2+ parents
History needn’t be linear

- “Branch” - where the history diverges, 2+ revisions have a common parent
- “Merge” - where one branch connects with others, one revision has 2+ parents

Main branch - “master”; “trunk”
History needn’t be linear

- “Branch” - where the history diverges, 2+ revisions have a common parent
- “Merge” - where one branch connects with others, one revision has 2+ parents
History needn’t be linear

- “Branch” - where the history diverges, 2+ revisions have a common parent
- “Merge” - where one branch connects with others, one revision has 2+ parents
A short history of VCS

- Earliest VCS (SCCS, 1972)
- “Local VCS”
- Only one file per repository
- Repository kept on the same computer
- Only one user can access the file at a time
- Revisions stored as deltas*
- Considered obsolete

* “Delta” - shows what has been changed compared to last revision

Source: “Pro Git” by Scott Chacon and Ben Straub
Moving on

- “Centralized VCS”
- Repository is held on one computer ("server")
- Working copies can be checked out to multiple computers ("clients")
- Handles multiple files, folders, later renaming and moving of files
- Became very popular worldwide (CVS, 1990; Subversion, 2000)
- Popularised open-source development (Sourceforge)
- Still developed and in use

Source: “Pro Git” by Scott Chacon and Ben Straub
Problems with centralised VCS

- Most actions require communication with the server
  - Slow
  - Needs internet/network
- Repository is held in one place
  - If it’s lost, so is the whole history
State of the art

- “Distributed VCS”
- Repository - including the whole history - is held on each developer’s own machine
- New revisions are checked in to local repository
- Local repository can be synchronised with other remote repositories to send your changes to other people

Source: “Pro Git” by Scott Chacon and Ben Straub
Upgrade over central VCS

- Most operations done on the local repository
  - Much faster
  - Possible offline

- Each developer has the full history of the project
  - If anyone loses theirs - they just need to clone someone else’s copy
  - Changes can easily be sent to different remote repositories
    - E.g. always send to “development” repository, only send to “production” repository when code is tested
BitKeeper (2000)

- Distributed VCS
- Originally proprietary software (later released as open-source)
- Free version used to develop Linux kernel 2002-2005
- 2005: support for free version ended, spurring creation of Git and Mercurial
Git and/ vs Mercurial

- April 2005
- Distributed VCS
- Cross-platform, free, and open-source
- Well-known, used, and loved in the industry
- Several repository hosting options
- Mercurial
  - Abstracts/hides some stuff for easier usage
  - Easier to extend/integrate
- Git
  - Built for speed and scale
  - Follows unix software principles
  - More popular
Everything that can be done with Git, can be done with Mercurial, and vice versa
Where to put Git repositories

- Share directly between users
- Install Git on your own server
- Github
  - The most popular open source repository hosting service
  - Able to make private repositories
  - Project management tools
- Bitbucket
  - Public and private repositories
  - Project management tools
- Gitlab
  - Can be installed on own server
  - Project management tools
Demo time

1. Creating a repository on Bitbucket
2. Cloning it into your computer
3. Adding a file
4. Making commits
5. Seeing history
6. Going back to a certain state
7. Pushing and pulling changes
Back to Branches

- Branches enable you to have multiple parallel versions of data
- Separate development of unrelated features
- Generally the main branch ("master") should have the latest tested & stable version of the software
Git is built for branches

- In early VCS, branching meant moving files around and so was slow
- Git stores branches as just links to certain commits
- “Branch early; branch often”
- Branches can always be deleted or just abandoned, you can always go to previous state
Demo time

1. Creating branch
2. Switching branches
3. Commit on branch
4. Merging branches
Merge conflicts

- Git merges most changes automatically
- As long as two branches change different files or different lines in the same files, it’s OK
- “Merge conflict” - when Git is not sure how to combine changes in a file
Successful merge

An old silent pond...
A frog jumps into the pond, splash! Silence again.

An old silent lake...
A frog jumps into the pond, splash! Silence again.

An old silent pond...
A frog jumps into the pond, bang! Silence again.

An old silent lake...
A frog jumps into the pond, bang! Silence again.
An old silent pond...
A frog jumps into the pond, splash! Silence again.

An old silent pond...
A rock jumps into the pond, splash! Silence again.

An old silent pond...
A rock falls into the pond, splash! Silence again.

An old silent pond...
A frog falls into the pond, splash! Silence again.
$ git merge newbranch

Auto-merging haiku.txt
CONFLICT (content): Merge conflict in haiku.txt

Automatic merge failed; fix conflicts and then commit the result.
What to do with a merge conflict

1. Take a deep breath
2. Take a step back
3. Assess the situation
4. Roll up the sleeves
5. Open the file

An old silent pond...

A rock jumps into the pond,

A frog falls into the pond,

splash! Silence again.
6. Find the conflict

An old silent pond...

A rock jumps into the pond,

A frog falls into the pond,

splash! Silence again.
7. Inspect the markers

An old silent pond...

A rock jumps into the pond,

A frog falls into the pond,

splash! Silence again.

"Conflict resolution markers"
8. Find “ours”

An old silent pond...

```plaintext
<<<<<< HEAD
A rock jumps into the pond,

A frog falls into the pond,
```

>>> newbranch

splash! Silence again.

“HEAD” is a pointer to the latest checked out state. From here until separator it’s “ours”, “current”, e.g. changes from the current branch.
9. Find “theirs”

An old silent pond...

<<<<<<<<< HEAD

A rock jumps into the pond,

=========

A frog falls into the pond,

>>>>>> newbranch

splash! Silence again.

The name of the branch that we are trying to merge, also called “theirs”, “incoming”. From the separator until here it’s the changes that we are trying to merge in.
10. Decide what needs to stay

An old silent pond...

A rock jumps into the pond,
A frog falls into the pond,
splash! Silence again.
11. Write the solution, delete the markers

An old silent pond...
A rock falls into the pond,
splash! Silence again.

12. Add and commit your conflict resolution

13. Pat self on back
Preventing conflicts

- **Branch early, branch often**
  - Isolating your work from that of others helps avoid accidental conflicts
- **Keep updated**
  - Pull from remote often to know what’s going on
- **Make small commits often (“atomic”)**
- **Commit only what’s necessary**
  - `git status`, `git diff`, and `git add` are your best friends; `git add .` is your biggest enemy
- **Commits can be merged, changed, and removed even after they’re made**
- **It is okay to change history when working with Git**
Git has much more stuff

- Aliases
  - Shortcuts for advanced Git commands
- Modifying history
  - Unlike Mercurial, it is mostly OK to rewrite commits in Git
- Searching through history
- Running scripts when certain events occur
  - Before, during, and after commit/push/merge/etc
- Branching workflows, good practices
- etc
In conclusion

- Everybody uses VCS
- You’re probably going to use either Git or Mercurial
- Learn both for good measure
- Educate yourself about good VCS practices
  - Branch all the time
  - Make atomic commits
  - Leave helpful commit messages
Cool hang. Next time:

- Dependency management and build task automation
- Issue tracking
- Intra-team communication & collaboration
Questions?