Continuous Integration

CSCI 5828: Foundations of Software Engineering
Lecture 27 — 04/24/2012

Goals

- Review the material in Chapter 15 of the Agile textbook
    - Continuous Integration
        - Source Control
        - Automated Builds
        - A culture of incremental changes
- Apply these techniques to a simple example

Continuous Integration

- Continuous Integration can refer to a number of things
  - A practice of always building a working version of your system each day
  - A practice of always running your test cases each day
  - A practice of committing changes to your version control system each day
- An automated system that monitors changes to your version control system such that when it detects the commit
  - it checks out that version of your system
  - builds it, runs all tests, and verifies that it passes
  - if it does, it creates a new official release
  - if it doesn't, it notifies the appropriate developers

Why Do You Need It?

- The book starts this chapter with a scenario in which
    - your boss calls to say that he's bringing investors to the office in an hour to check out the most recent version of your system
        - you have less than 60 minutes to create a stable build, deploy it onto a server, and prepare for a demo
- The book offers two ways this scenario plays out
    - A big production that ends in failure
        - The state of the project is unclear, the merge is ugly, the demo crashes
    - A nonevent that ends in success
        - The state of the project is known; a deployment is straightforward

Culture of Production Readiness

- In the beginning stages of development, it is easy to think of deployment as an event that will happen in the distant future
    - Such thinking can get you into trouble, especially if you do not establish a culture of always building a working system
- Continuous Integration fosters a sense of always being production ready
    - meaning, you always have a running system
- Now, at first, your system may run but not do anything
    - This is better than a system that can’t be compiled or invoked
- With that as a base, you can “grow the system”
    - adding features incrementally, always ensuring that the system can be run

The Practice of Continuous Integration

- As a practice, continuous integration is taking
    - all the changes developers make to their software and
    - integrating them all together continuously throughout the day
- Building a working system multiple times per day is commonplace and expected behavior
    - Such practices let you know when you have “broken the build” by “failing fast”
        - You can then focus on getting things fixed and ensuring that by the end of the day, you’re back to having a running system
- If you can achieve this, a demo becomes a non-event. You’re already building working systems every day; the demo is just one more build and deploy

Making It Work

- To make continuous integration possible, you need
  - a source code repository
  - a check-in process
  - an automated build process
  - a willingness to work incrementally
- The first two are enabled by version control systems (which have a long history in software engineering research)
  - The most popular version control systems (git, mercurial) are known as distributed version control systems
    - this basically means that all developers have a “master copy” of the repository; you have everything you need to do development on your local machine

Check-In Process

- By a check-in process, the book means that each of your developers has a similar way in which they develop code
  - Grab the latest version of the system (however that's accomplished)
  - Write tests and make changes until you're ready to check in your work
  - Run all tests and ensure that all of them pass
  - Check for any updates that occurred while you were working
  - Run all tests again and ensure that all of them pass
  - Check in your changes
- At the end of this process, if the team is using an automated system, it will kick in, check out your changes, test them, and then update the official release (assuming everything passed)

Automated Builds

- Most programming languages are associated with automated build frameworks
  - C and Make
  - Java and Ant / Maven
  - Ruby on Rails and rake
- All of these depend on a specification of some sort and a command you invoke to run the build
- Most integrated development environments have automated builds as one of their core capabilities
  - You create a project, add/modify files, click “Run”, and the project is built for you automatically

Work Incrementally

- The key is to integrate new code into the system in small chunks
  - You want to be integrating new code or changes into your system and building your system several times per day
  - This ensures that you are finding incompatible changes quickly
- If you fail to integrate new/changed code multiple times per day, you will set yourself up for pain in the near future
  - perhaps right at the end of an iteration, where everything is supposed to come together, but instead fails, leading to a missed demo
    - and extra work trying to integrate large sets of changes all at once

That's It!

- Those are the basics
  - Now, we'll spend some time looking at a few of these concepts in more detail
    - Version control systems: why do we need them?
      - Learn a few concepts related to Git
    - Automated build system
      - Use Ant as an example
- Finally, we'll finish with an example in which we step through a variant of the energy source example from Lecture 12
  - Based on Chapter 5 of the Concurrency Textbook

Without a Net (I)

- Doing software development without configuration management is “working without a net”
  - Configuration management refers to both a process and a technology
    - The process encourages developers to work in such a way that changes to code are tracked
      - changes become “first class objects” that can be named, tracked, discussed and manipulated
    - The technology is any system that provides features to enable this process

Without a Net (II)

- If you don't use configuration management then
  - you are not keeping track of changes
  - you won't know when features were added
  - you won't know when bugs were introduced or fixed
  - you won't be able to go back to old versions of your software
- You would be "living in the now" with the code
  - There is only one version of the system: this one
- You would have no safety net

Without a Net (III)

Developer 1

Developer 2

Two developers need to modify the same file for the task they are working on

Demo Machine

A

Without a Net (IV)

Developer 1

A

working copy

Developer 2

A

They both download the file from the demo machine, creating two working copies.

Demo Machine

A

Without a Net (V)

They both edit their copies and test the new functionality.

Developer 1

A1

Developer 2

A2

Demo Machine

A

Without a Net (VI)

Developer 1

A1

Developer 2

A2

Developer 1 finishes first and uploads his copy to the demo machine.

Demo Machine

A1

Without a Net (VII)

Developer 1

AI

Developer 2

A2

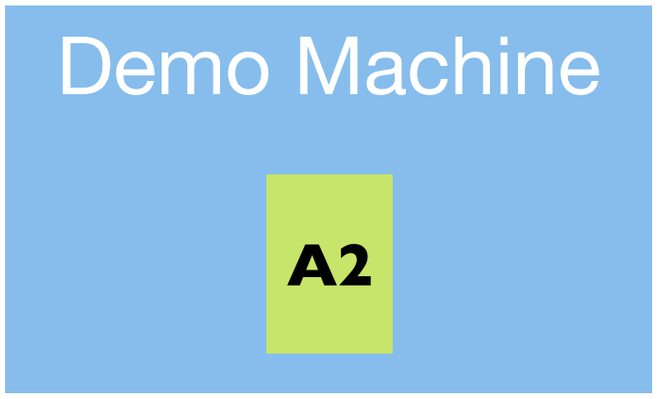
Developer 2 finishes second and uploads his copy to the demo machine.

Demo Machine

A2

Without a Net (VIII)

This is known as “last check in wins”

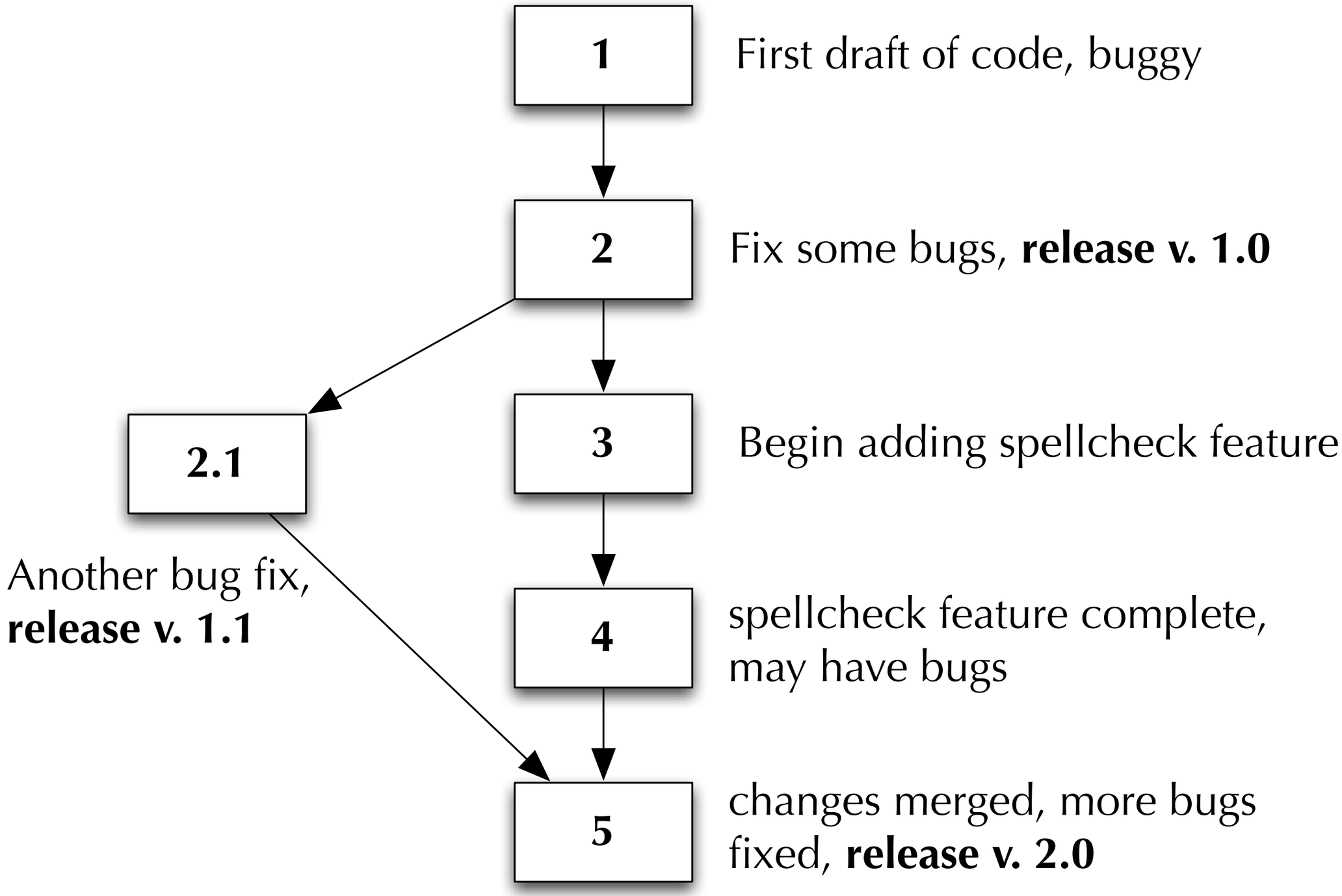


At best, developer 1’s work is simply “gone” when the demo is run; At worst, developer 1 checked in other changes, that cause developer 2’s work to crash when the demo is run.

Not Acceptable

- This type of uncertainty and instability is simply not acceptable in production software environments
  - That's where configuration management comes in
- You might sometimes encounter the term "version control"
  - But in the literature, "version control" is "versioning" applied to a single file while "configuration management" is "versioning" applied to collections of files

Versioning

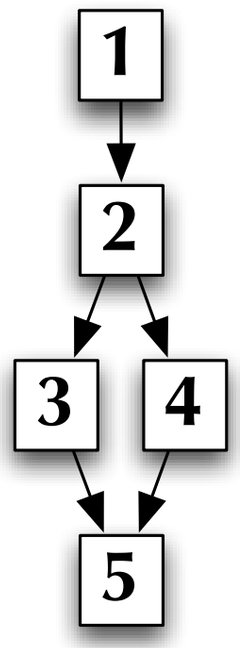

1
First draft of code, buggy
2
Fix some bugs, release v. 1.0
2.1
Another bug fix, release v. 1.1
3
Begin adding spellcheck feature
4
spellcheck feature complete, may have bugs
5
changes merged, more bugs fixed, release v. 2.0

Configuration Management

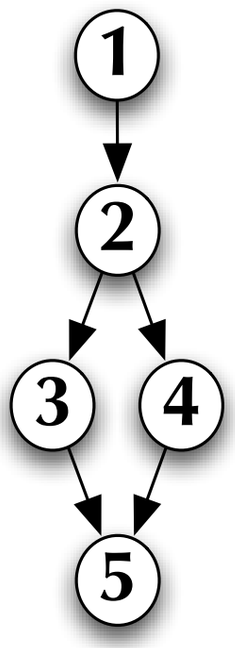
Particular versions of files are included in...

... different versions of a configuration

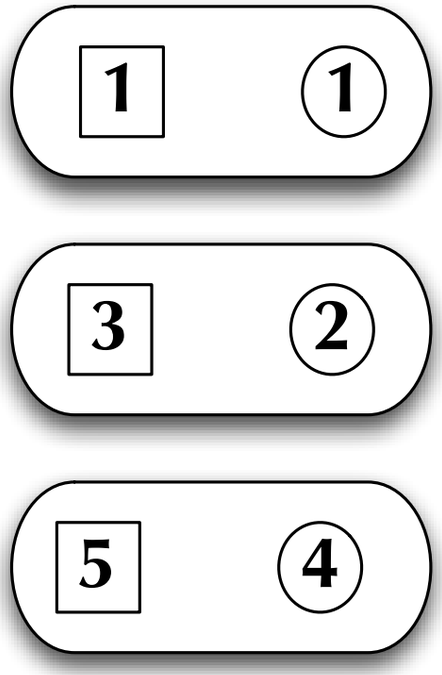
File A



File B



Configuration Z



With a Net (I)

Developer 1

Repository

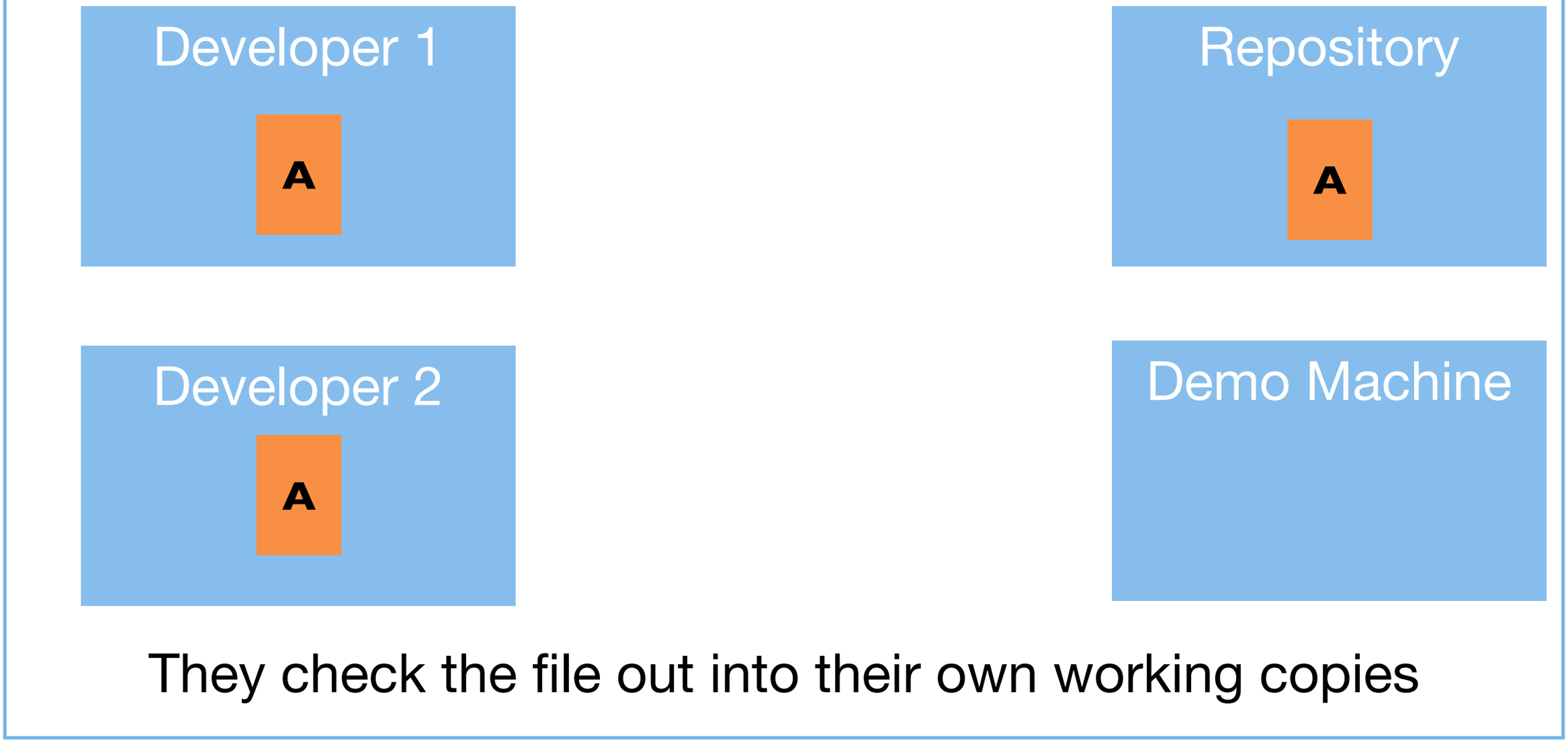
A

Developer 2

Demo Machine

Two developers need to modify the same file for separate tasks

With a Net (II)



They check the file out into their own working copies

With a Net (III)

Developer 1

A1

Repository

A

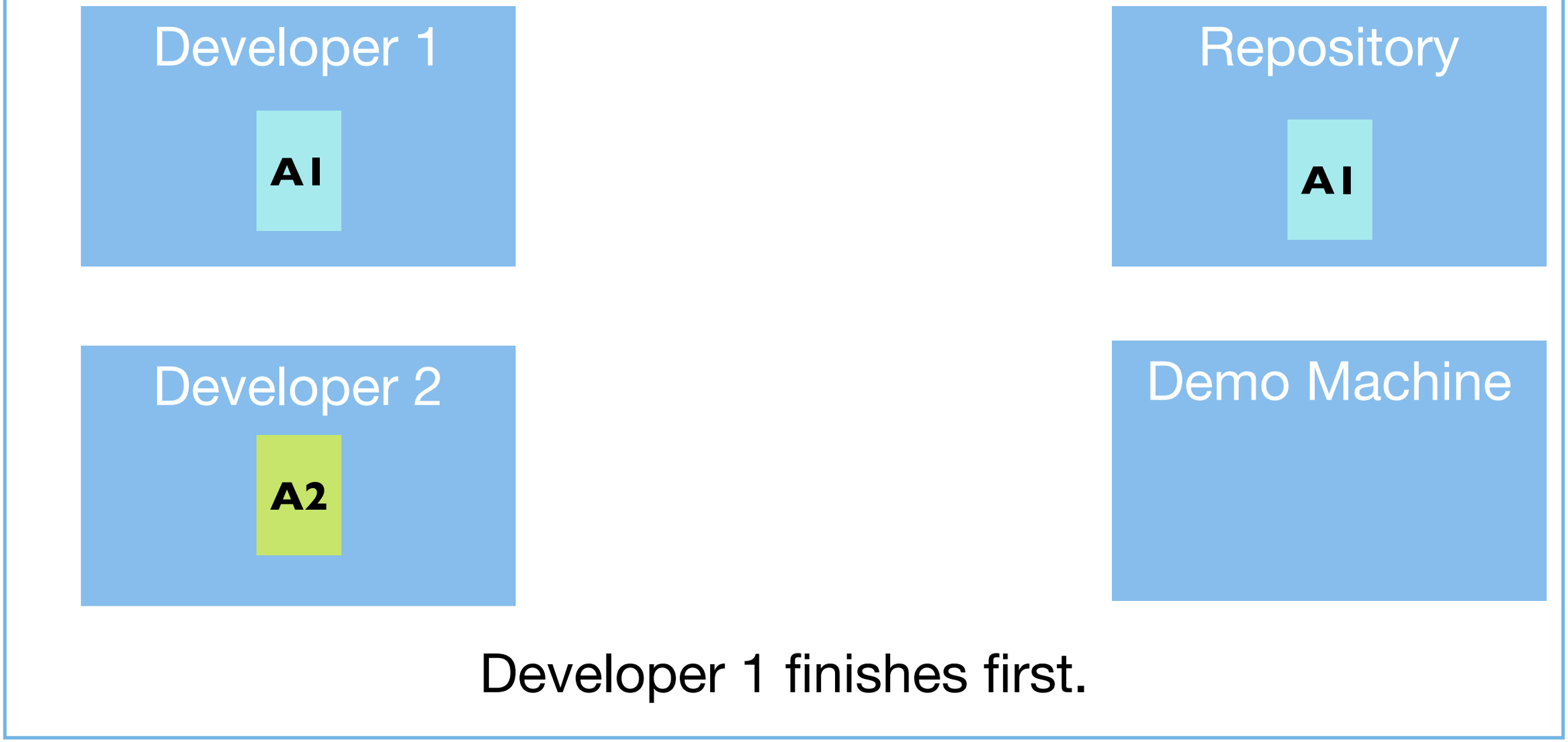
Developer 2

A2

Demo Machine

They modify their copies.

With a Net (IV)



Developer 1 finishes first.

With a Net (V)

Developer 1

A1

Repository

A2

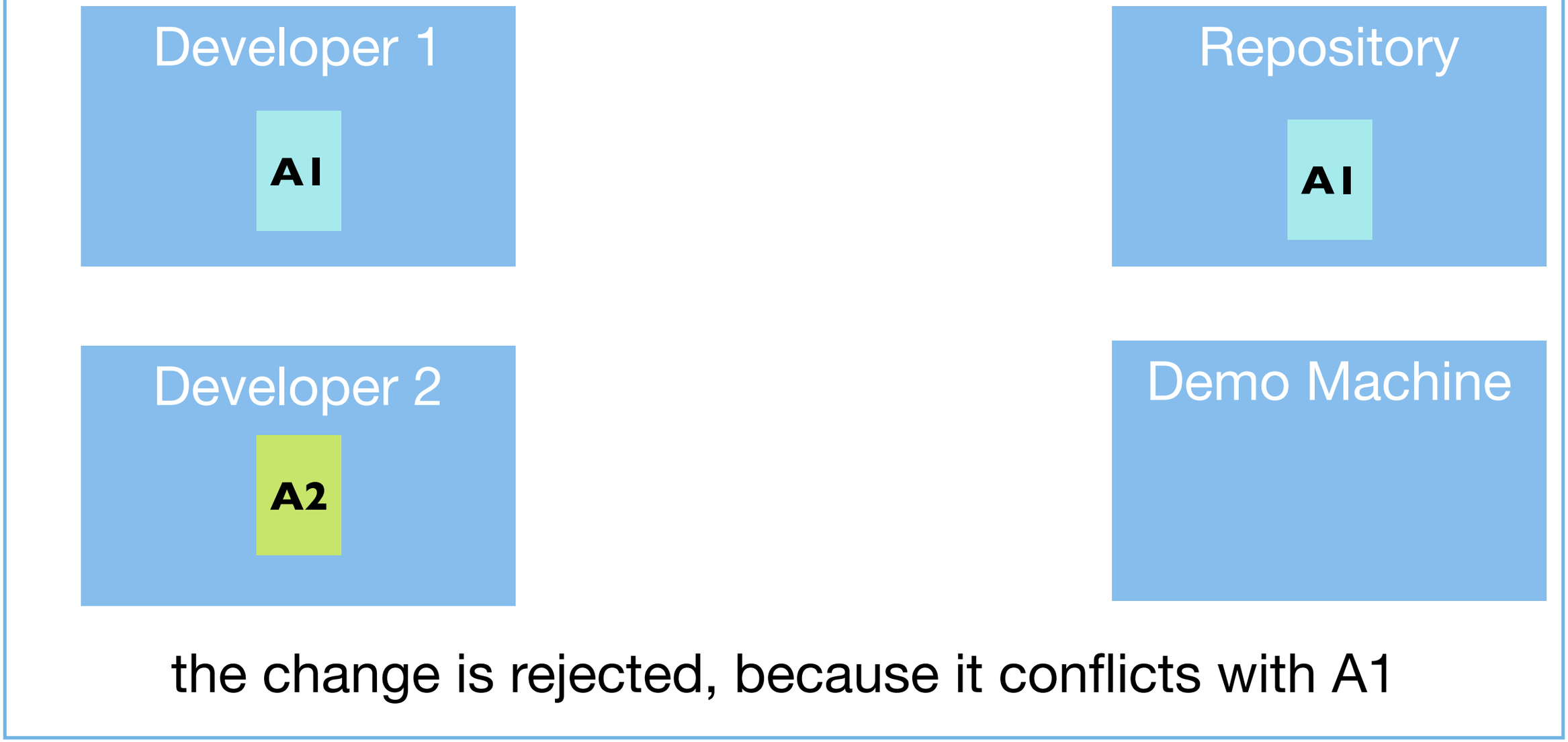
Developer 2

A2

Demo Machine

Developer 2 finishes and tries to check in, but...

With a Net (VI)



the change is rejected, because it conflicts with A1

With a Net (VI)

Developer 1

A1

This is known as “first check-in wins”!

Repository

A1

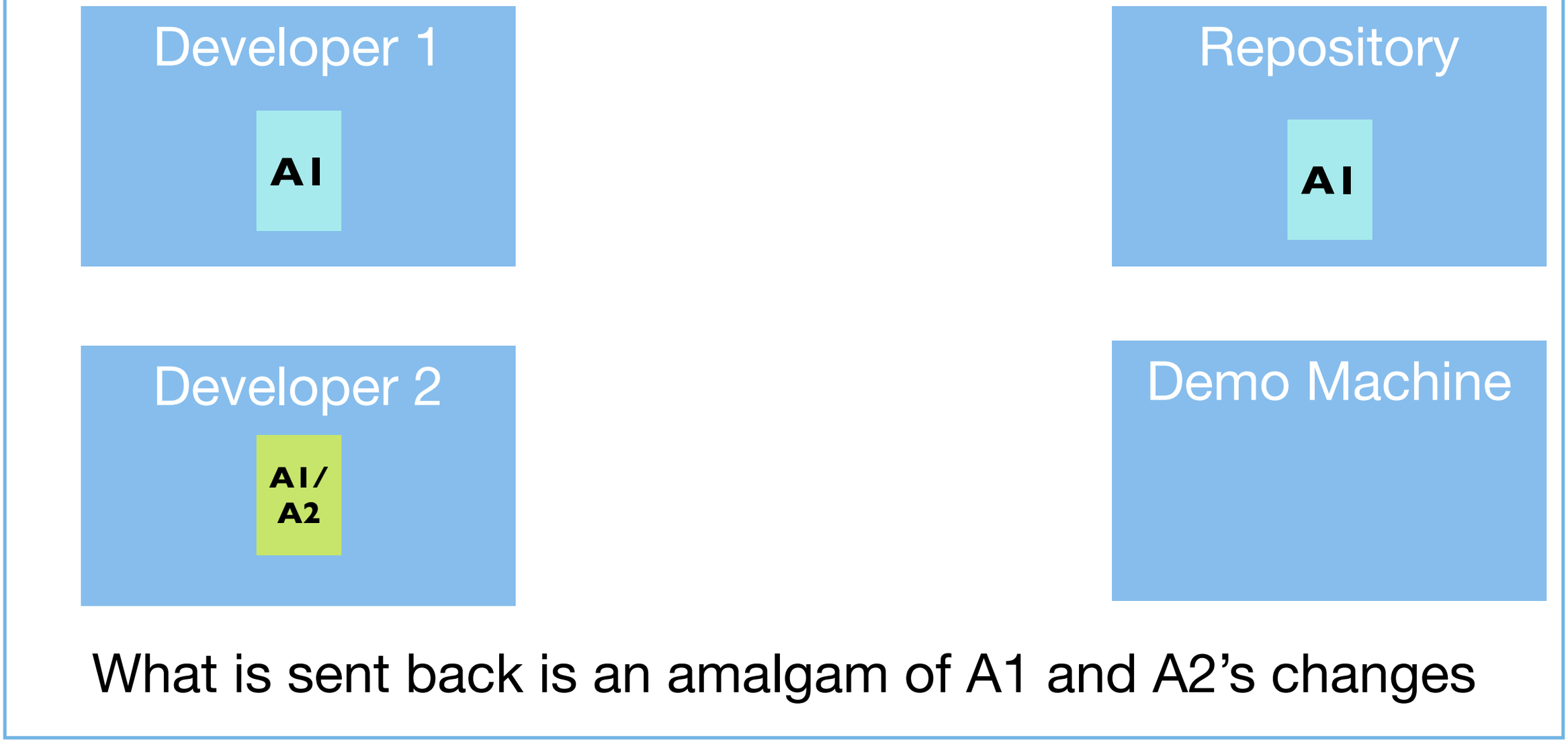
Developer 2

A2

Demo Machine

the change is rejected, because it conflicts with A1

With a Net (VII)



What is sent back is an amalgam of A1 and A2's changes

With a Net (VII)

Developer 1

A1

Developer 2

A1/
A2

The file will not be syntactically correct and will not compile!

Repository

A1

Demo Machine

What is sent back is an amalgam of A1 and A2's changes

With a Net (VII)

Developer 1

A1

Repository

A1

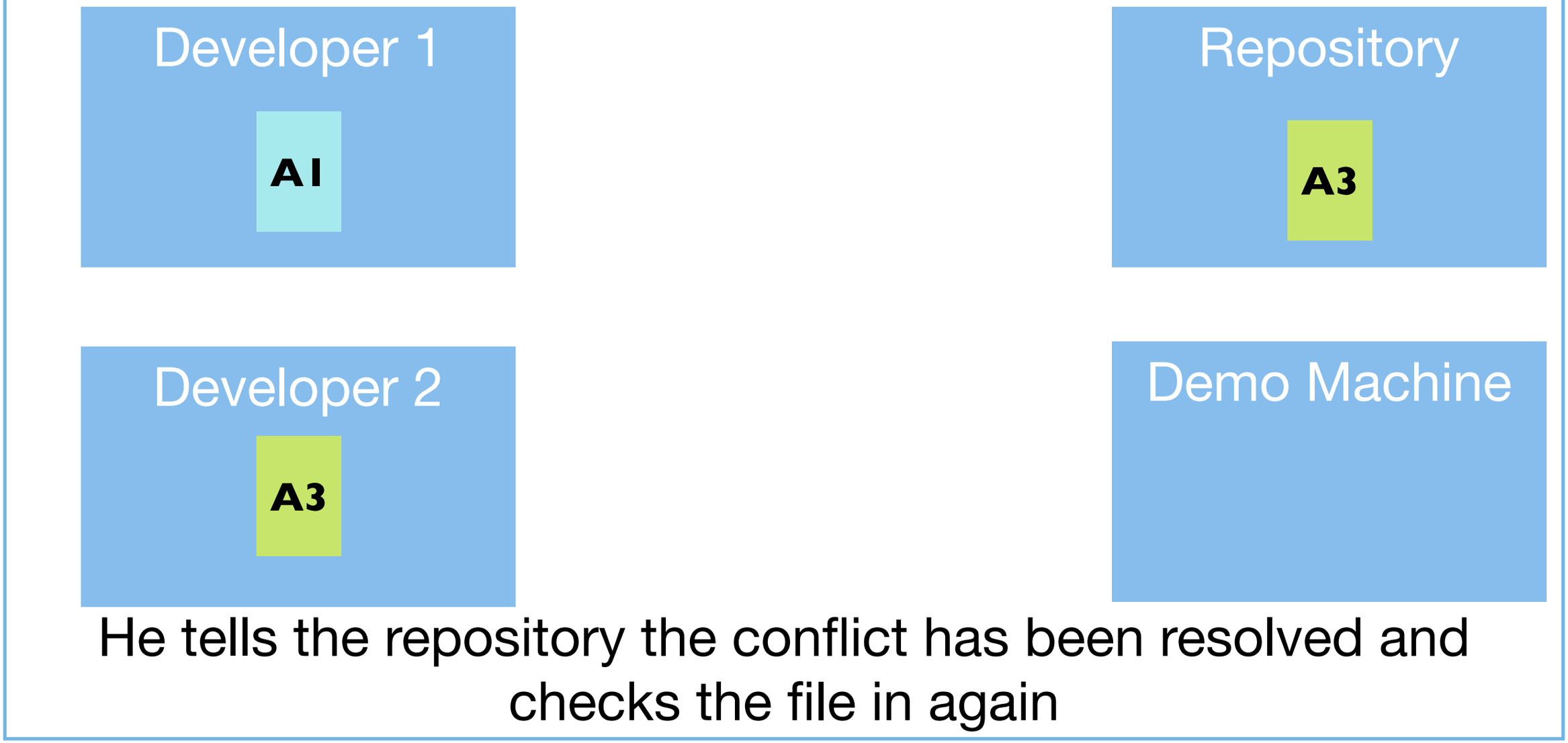
Developer 2

A3

Demo Machine

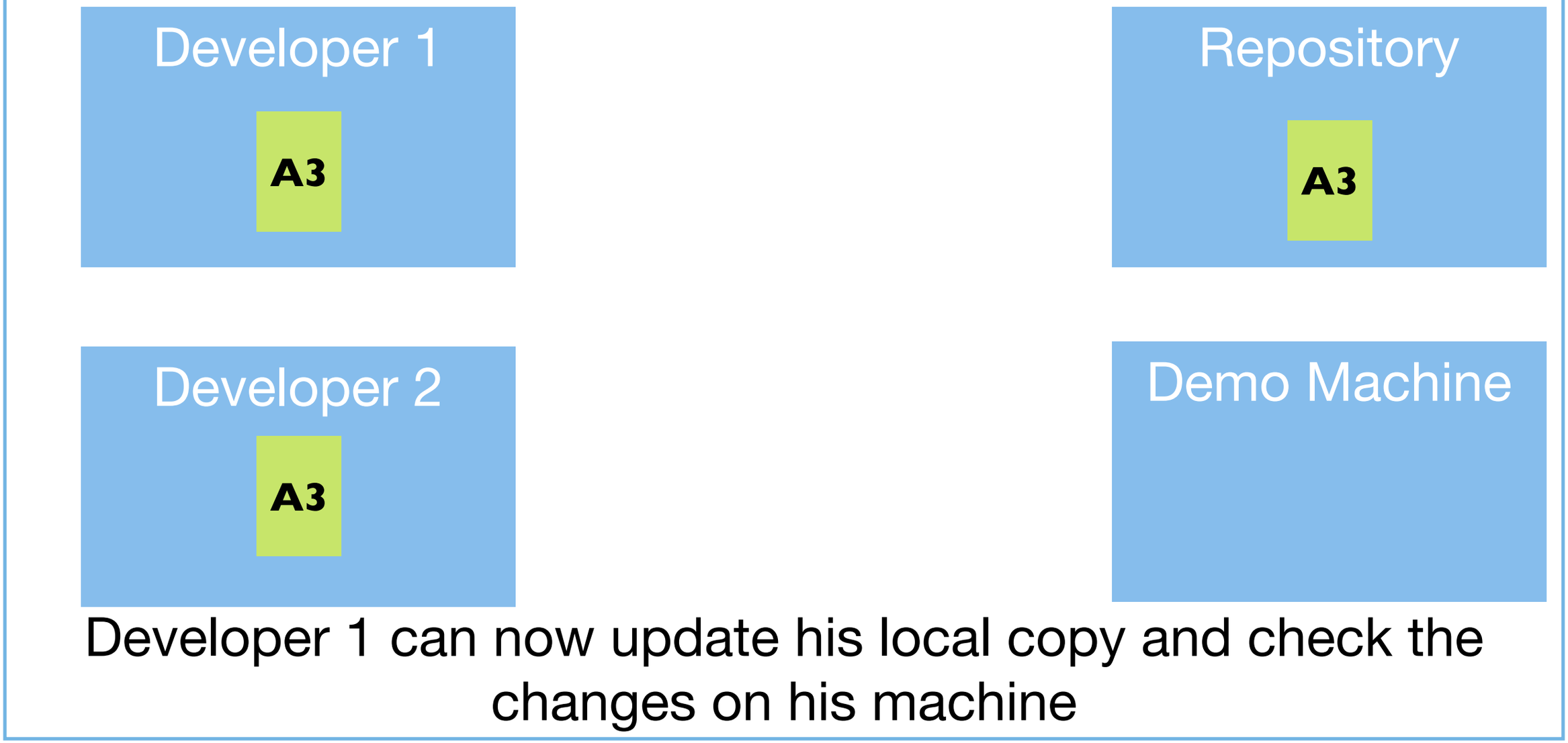
It is up to Developer 2 to merge the changes correctly!

With a Net (VII)



He tells the repository the conflict has been resolved and checks the file in again

With a Net (VII)



Developer 1 can now update his local copy and check the changes on his machine

With a Net (VII)

Developer 1

A3

Repository

A3

Developer 2

A3

Demo Machine

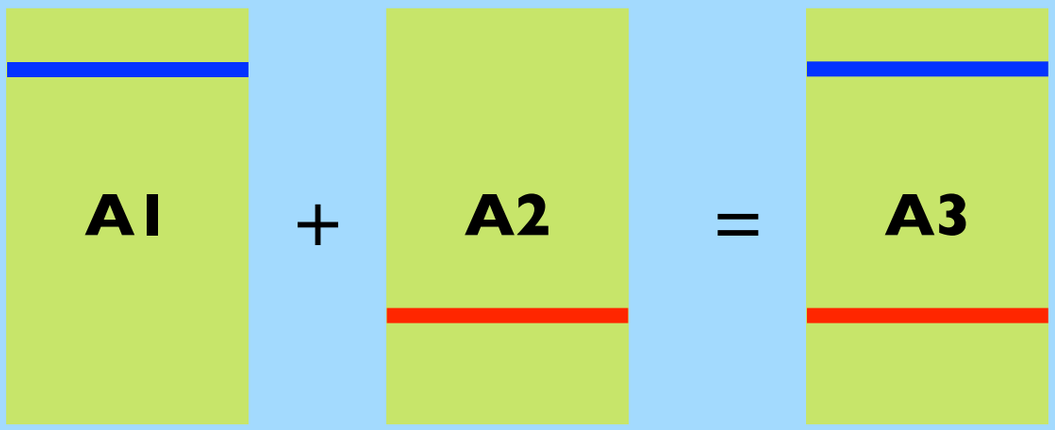
A3

When they are both satisfied, the system can be deployed to the demo machine and a successful demo occurs!

Why Multiple Copies?

- Old versioning systems (RCS) did not allow multiple developers to edit a single file at a same time
  - Only one dev. could “lock” the file at a time
- What changed?
  - The assumption that conflicts occur a lot
  - data showed they don’t happen very often!

When two developers edit the same file at the same time, they often make changes to different parts of the file; such changes can easily be merged



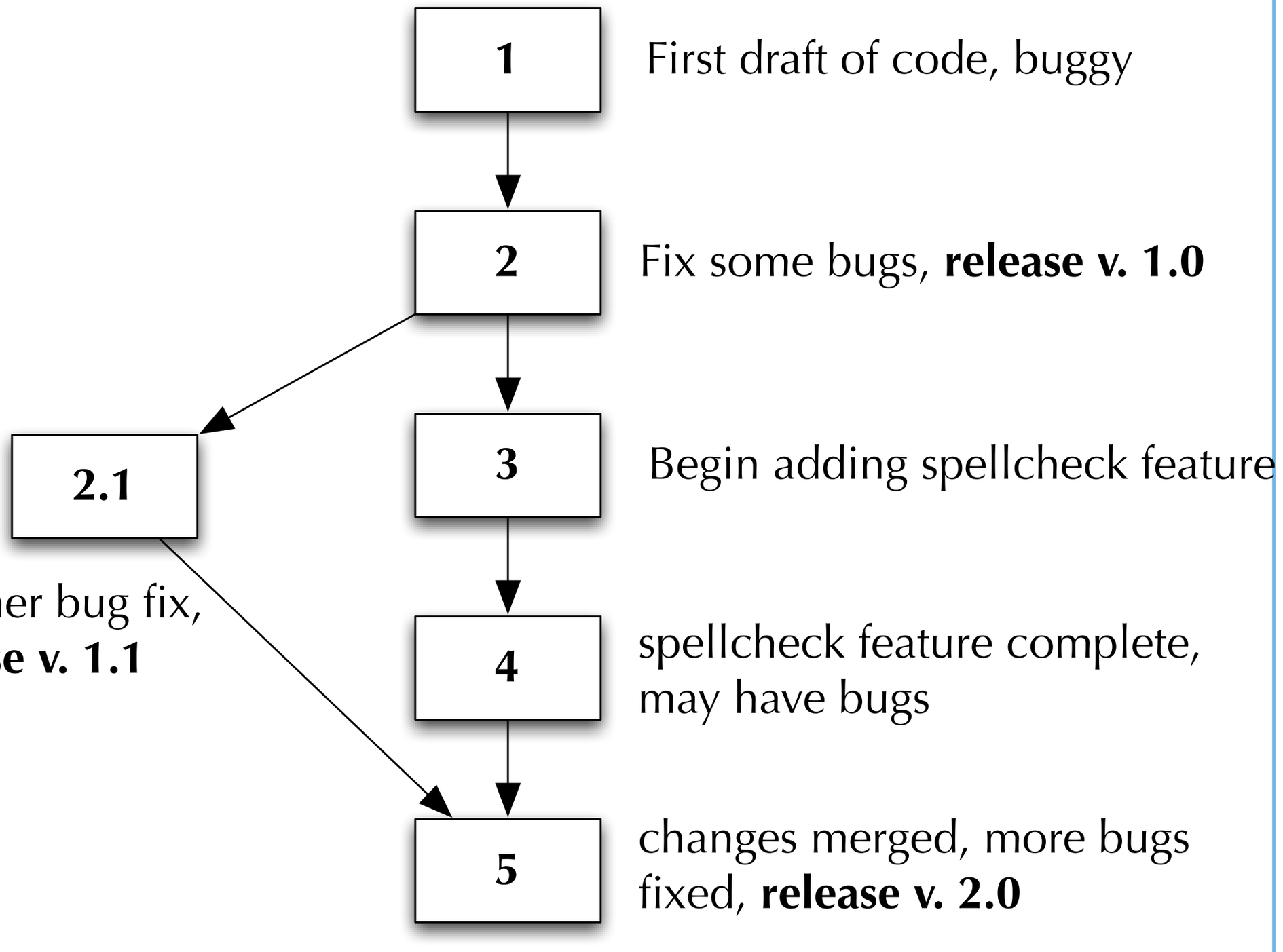
Tags, Branches, and Trunks, Oh My!

- Configuration management systems can handle the basics of checking out the latest version of a system, making changes, and checking the changes back in
  - These changes are committed to what is typically called “the trunk” or main line of development
    - git calls it the “master” branch
- But configuration management systems can do much more than handle changes to the version of a system that is under active development
  - and that’s where tags and branches come in

Scenario (I)

- Imagine that a development team has released version 1.0 of a system and has moved on to work on version 2.0
  - they make quite a bit of progress when their customer reports a significant bug with version 1.0
- None of the developers have version 1.0 available on their machines and none of them can remember what version of the repository corresponded to “release 1.0”
  - This highlights the need for good “commit messages”
    - when you are checking in changes be very explicit about what it is you have done; you may need that information later

Remember this diagram? The numbers in boxes are repository versions; the text in bold represent tags



Scenario (II)

- To fix the bug found in version 1.0 of their system, the developers
  - look at the log to locate the version that represented “release 1.0”
  - associate a symbolic name with that version number to “tag it”
    - In this case the tag might be “release_1.0”
  - create a branch that starts at the “release 1.0” tag
  - and fix the bug and commit the changes to the branch
    - They don’t commit to the trunk, since the associated files in the trunk may have changed so much that the patch doesn’t apply
      - once the patch is known, a developer can apply it to the trunk manually at a later point; or use a “merge/fix conflicts” approach

Branches are Cheap

- In any complicated software system, many branches will be created to support
  - bug-fixes
    - e.g. one branch for each official release
  - exploration
    - possibly one branch per developer or one per “risky” feature
      - e.g. switching to a new persistence framework
- Because of this, modern configuration management systems make it easy to create branches

Distributed Configuration Management (I)

- With subversion and cvs (and many others), configuration management depends on an “official” repository
  - There is a notion that somewhere there is a “master copy” and that all working copies are subservient to that copy
- This can be a limiting constraint in large projects with lots of developers; (single point of failure; off-line work is hard)
  - so much so that the large project may be tempted to write its own configuration management system just to make progress
    - this is what happened with the Linux project; they produced git because no other configuration management system met their needs!

Distributed Configuration Management (II)

- In distributed configuration management systems, like git, the notion of a centralized repository goes away
  - each and every developer has their own “official” repository
    - with a master branch and any other branches needed by the local developer
  - then other developers can “pull” branches from publicly available git repositories and “push” their changes back to the original repository
- You can learn more about git at the git tutorial
  - <http://www.kernel.org/pub/software/scm/git/docs/gittutorial.html>

More on Git

- There are three main areas of a Git project
  - The git directory (.git in the root dir of the project)
  - A working directory and a staging area
- Files and dirs in the git directory have been committed (with a detailed history maintained of all changes)
- Files and dirs in the working directory represents the latest version of the project, along with any local modifications
- Files and dirs in the staging directory have been modified and are scheduled to be included in the next commit

Automated Build Systems

- Automated build systems are ubiquitous
    - Each time you click “Run” in XCode, Visual Studio, Eclipse, etc., an automated build system is
        - checking to see if any file changed
        - if so, it gets compiled
        - then any files that depend on it are also recompiled (and so on)
        - then the entire system is linked (made ready to run)
        - and then launched automatically

Make

- An early automated build system (still in use today)
    - Dependency information between files in a project are placed in a file called a Makefile
        - Typing “make” at the command, processes those dependencies, and causes the files of the project to be compiled in the correct order
        - A developer can then make changes to the project files, save them, and then, just type “make” to recompile and rebuild the project
- The Makefile is a specification file that can contain many rules; each rule specifies how a particular file (or category of files) depends on other files
    - Almost all build systems rely on dependency information between files to work; in some cases, dependency information is automatically inferred

Ant

- Ant is a build system used to build large Java programs
  - It's specification is stored in a file called build.xml
  - build.xml is (surprise!) an XML file that contains rules to build the system-under-development
- Instead of typing "make", developers instead type "ant"
  - ant then looks for a build.xml file and uses it to compile (and possibly run) a system
- Ant has largely been superseded by Maven
  - Maven will automatically download dependencies from remote repositories
    - It is much more powerful than Ant

Example

- Let's see an example of using some of these tools together
  - The example is incomplete as I will not be integrating tests into my automated builds
    - I will also not be using an automated continuous integration system
      - See the software engineering presentations for details on at least two of these systems (Bamboo and Jenkins)
- Example taken from Lecture 12
  - The energy source example that starts out in a very bad state
  - We'll mimic some of the changes made to this system in Lecture 12 and this time check changes into Git and automate the build with Ant

Demo

- Step 1: Create directory with initial source files
- Step 2: Create a build.xml file to build and run system
- Step 3: Check initial files into Git
- Step 4: Start to apply changes from Lecture 12
  - After each change use Ant to build system and Git to check in changes

Summary

- Reviewed the material in Chapter 15 of the Agile textbook
    - Continuous Integration
        - Source Control
        - Automated Builds
        - A culture of incremental changes
- Apply these techniques to a simple example
    - Git and Ant to manage example from Lecture 12
        - Based on example from Chapter 5 of the Concurrency textbook

Coming Up Next

- Lecture 28: Software Engineering Presentations
- Lecture 29: Grand Central Dispatch
- Lecture 30: TBD