Good-Enough Design & Version Control

Kenneth M. Anderson
University of Colorado, Boulder

CSCI 5828 — Lecture 11 — 02/16/2010

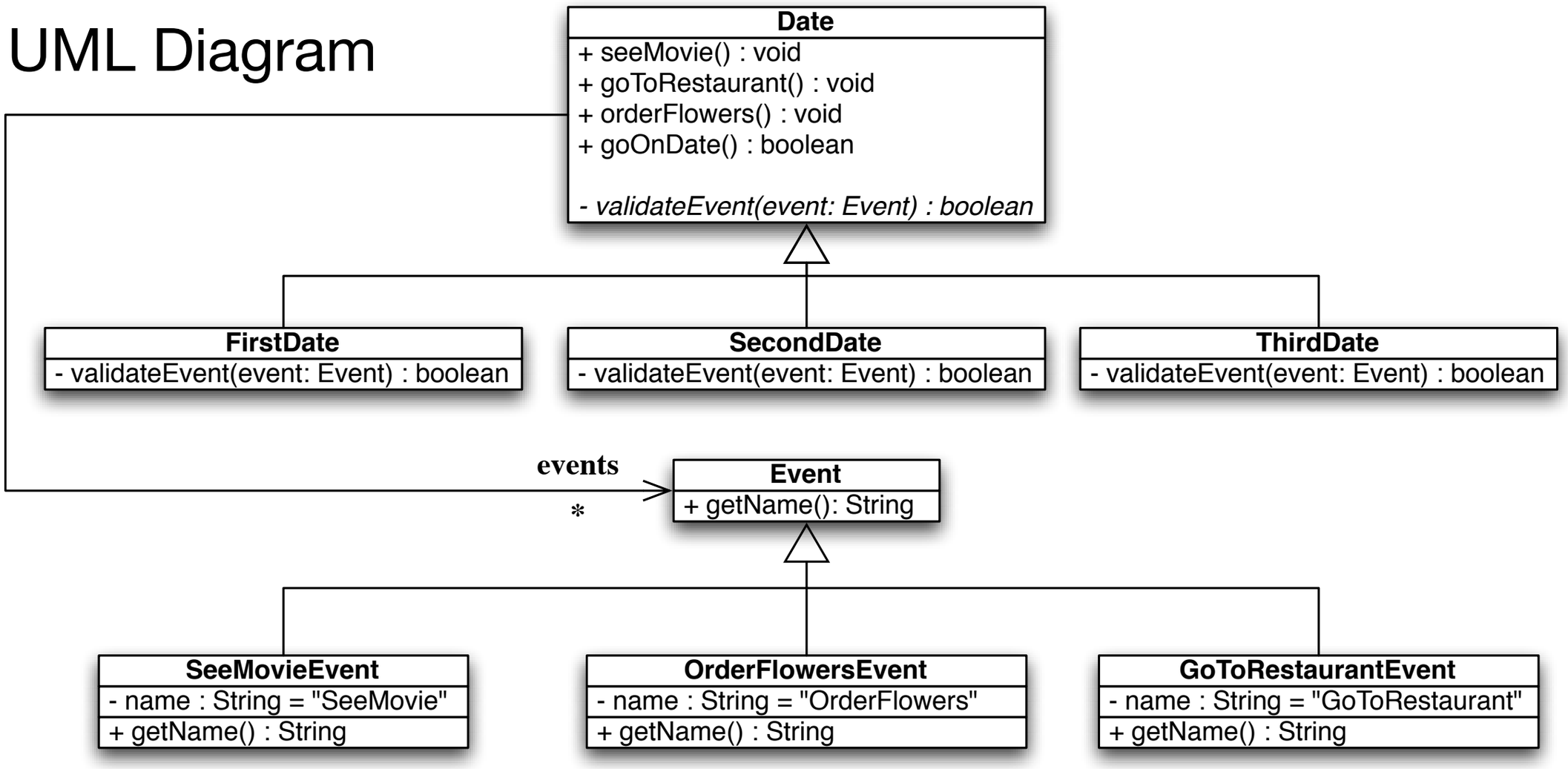
Goals

- Review material from Chapter 5 of Pilone & Miles
  - Software Design: Need for Good OO A&D principles
    - SRP: Single Responsibility Principle
    - DRY: Don’t Repeat Yourself Principle
- Review material from Chapter 6 of Pilone & Miles
  - Version Control & Configuration Management
    - Working “Without a Net”
    - Repository Management
      - Init, Add, Branch, Merge

iSwoon in Trouble

- The previous chapter presents a design for associating dates and events that was causing problems
  - Date objects maintain a list of its planned events
    - An Event object is a “dumb data holder” storing only a name
      - It has no logic of its own
  - Date objects provide methods that internally add events to a planned date; The Date object contains information about what events are allowed on a particular date
- The UML diagram is shown on the next slide

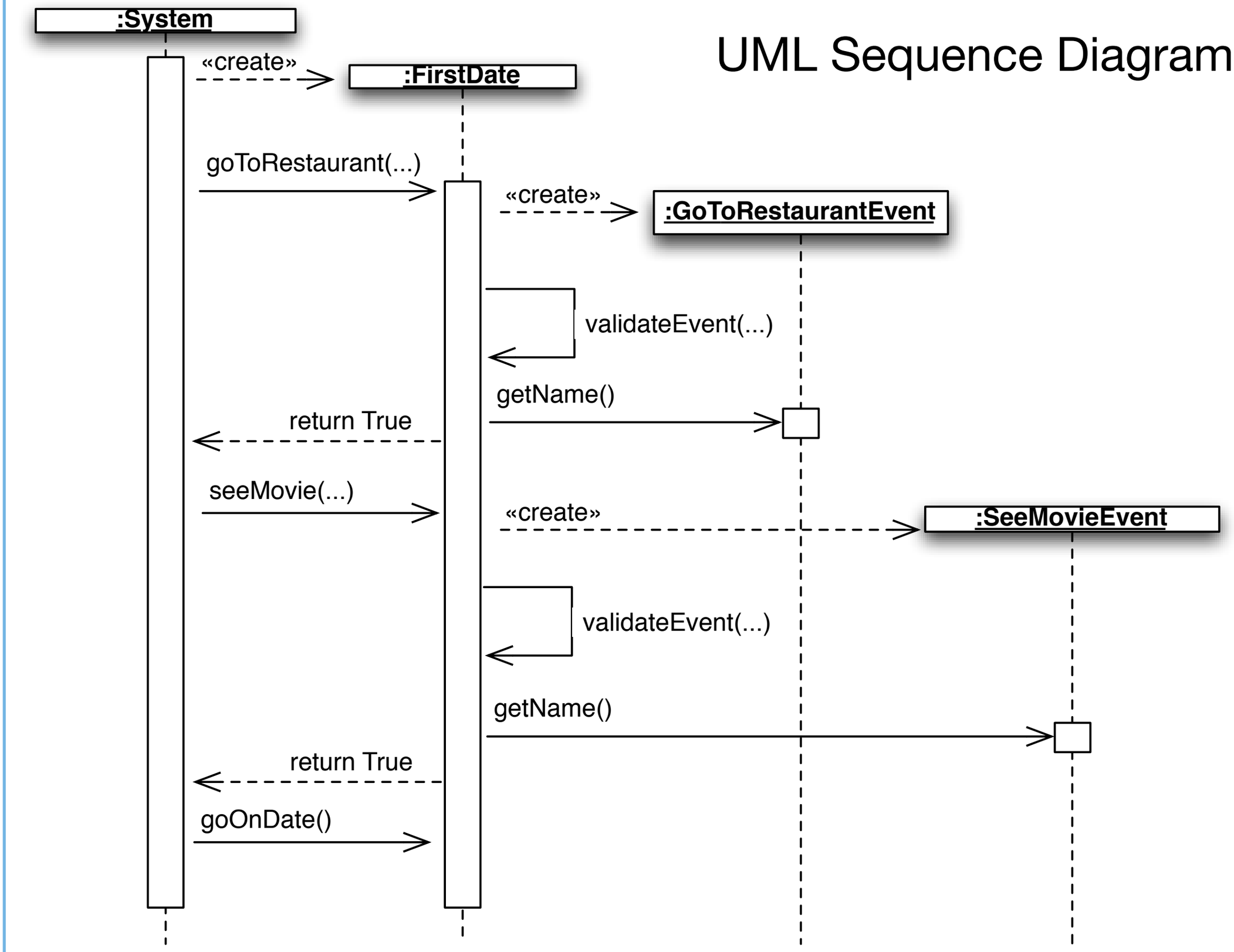
UML Diagram



UML Primer:

Each rectangle represents a class that can have attributes and methods. A “+” symbols refers to “public” visibility; “-” indicates private visibility. The “*” means zero or more. The “large triangle” indicates inheritance. The arrow head indicates “one way navigation”; in the diagram above Dates know about Events while Events are blissfully unaware of Dates

UML Sequence Diagram



Bad Design (I)

- This design has a lot of problems
  - The Event class is completely useless
    - Why not have Date store an array of strings?
  - Date's API is pretty bad
    - Event creation methods are specified for all possible events; that means that some dates have event creation methods for events that are not valid for them!
    - The Date class has a list of allowable events but doesn't show it on the diagram (or it doesn't show the list of planned events; either way it has two lists but only shows one)

Bad Design (II)

- But those are relatively minor issues
  - The main reason why this design is bad is that its **inflexible** with respect to the types of changes that occur regularly for this application domain
    - It can't easily handle the addition of a new type of Event
    - It can't easily handle changing the name of an existing Event
    - It can't easily handle the changing of what events are valid for what dates

Good Design

- A primary goal in OO A&D is producing a design that makes
  - **likely changes, straightforward**
    - typically by adding a new subclass of an existing class
    - or by adding an object that implements a known interface
  - no need to recompile the system or even it bring it down
- You can't anticipate **arbitrary changes** and there is no reason to invest time/$$ into planning for **unlikely** changes
  - So use good OO A&D principles to handle likely changes

Single Responsibility Principle (SRP) (I)

- The Date class has multiple responsibilities
  - tracking the events planned for a date
  - tracking the events allowed for a date
- It has multiple reasons to change
- The single responsibility principle says
  - Every object in your system should have a single responsibility and all the object's services should be focused on carrying out that single responsibility
    - This is also known as "having high cohesion"

SRP (II)

- Granularity?
  - When we say “responsibility” we are not talking about low level concerns, such as
    - “insert element *e* into array *a* at position *i*”
  - but design level concerns, such as
    - “classify documents by keyword”
    - “store client details”
    - “manage itinerary of Jack and Jill’s second date”

SRP (III)

- The existing iSwoon design is bad because each time we add a new event
  - We need to add a new Event subclass
  - Add a new method to Date
  - Update each of Date's subclasses (cringe!)
- We need to migrate to a design, in which the addition of a new type of event results in the addition of a new Event subclass and nothing more

Textual Analysis (I)

- One way of identifying high cohesion in a system is to do the following
  - For each class C
    - For each method M
      - Write “The C Ms itself”
- Examples
  - The Automobile drives itself
  - The Automobile washes itself
  - The Automobile starts itself

Textual Analysis (II)

- Sometimes you need to include parameters in the sentence
  - The CarWash washes the Automobile itself
- If any of these sentences doesn't make sense then investigate further
  - You may have discovered a service **that belongs to a different responsibility of the system** and should be **moved to a different class**
  - This may require first **creating a new class** before performing the move

Textual Analysis (III)

- Textual analysis is a good heuristic
  - While its useful for spot checking a design, its not perfect
- But the underlying principle is sound
  - Each class in your design should “pull its weight”
    - have a single responsibility with a nice balance of both data AND behavior for handling that responsibility

Other Problems

- The iSwoon design also has problems with duplication of information (indeed duplication can often lead to classes with “low cohesion” that violate SRP
- The duplication in iSwoon is related to Event Types
  - The names of event types appear in
    - Event subclass names
    - The name attribute inside of each event subclass
    - The method names in Date
- In addition, duplication occurs with validateEvent() in each of the Date subclasses

Don’t Repeat Yourself (I)

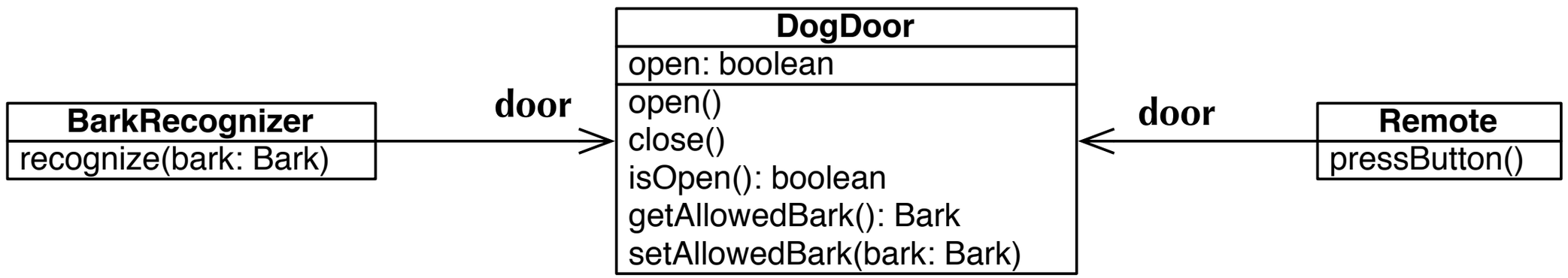
- The DRY principle
  - Avoid duplicate code by abstracting out things that are common and placing those things in a single location
- Basic Idea
  - Duplication is Bad!
  - At all levels of software engineering: Analysis, Design, Code, and Test

DRY (II)

- We want to avoid duplication in our requirements, use cases, feature lists, etc.
- We want to avoid duplication of responsibilities in our code
- We want to avoid duplication of test coverage in our tests
- Why?
  - Incremental errors can creep into a system when one copy is changed but the others are not
  - Isolation of Change Requests: We want to go to ONE place when responding to a change request

Example (I)

- Duplication of Responsibility



- "The dog door should automatically close 30 seconds after it has opened"
- Where should this responsibility live?
  - It would be easy to put this responsibility in the clients
  - But it really should live in DogDoor (which method?)

Example (II)

- DRY is really about ONE requirement in ONE place
  - We want each responsibility of the system to live in a single, sensible place
- This applies at all levels of the project, including requirements
  - Imagine a set of requirements for the dog door…

Example (III)

- The dog door should alert the owner when something inside the house gets too close to the dog door
- The dog door will open only during certain hours of the day
- The dog door will be integrated into the house's alarm system to make sure it doesn't activate when the dog door is open
- The dog door should make a noise if the door cannot open because of a blockage outside
- The dog door will track how many times the dog uses the door
- When the door closes, the house alarm will re-arm if it was active before the door opened

Beware of Duplicates!!!

Example (IV)

- The dog door should alert the owner when something inside the house gets too close to the dog door
- The dog door will open only during certain hours of the day
- The dog door will be integrated into the house's alarm system to make sure it doesn't activate when the dog door is open
- The dog door should make a noise if the door cannot open because of a blockage outside
- The dog door will track how many times the dog uses the door
- When the door closes, the house alarm will re-arm if it was active before the door opened

Example (V)

- The dog door should alert the owner when something is too close to the dog door
- The dog door will open only during certain hours of the day
- The dog door will be integrated into the house's alarm system
- The dog door will track how many times the dog uses the door

- Duplicates removed!

Example (VI)

- Ruby on Rails makes use of DRY as a core part of its design
  - focused configuration files; no duplication of information
  - for each request, often single controller, single model update, single view
- But prior to Ruby on Rails 1.2 there was duplication hiding in the URLs used by Rails applications
  - POST /people/create # create a new person
  - GET /people/show/1 # show person with id 1
  - POST /people/update/1 # edit person with id 1
  - POST /people/destroy/1 # delete person with id 1

Example (VII)

- The duplication exists between the HTTP method name and the operation name in the URL
  - POST /people/create
- Recently, there has been a movement to make use of the four major “verbs” of HTTP
  - PUT/POST == create information (create)
  - GET == retrieve information (read)
  - POST == update information (update)
  - DELETE == destroy information (destroy)
- These verbs mirror the CRUD operations found in databases
  - Thus, saying “create” in the URL above is a duplication

Example (VIII)

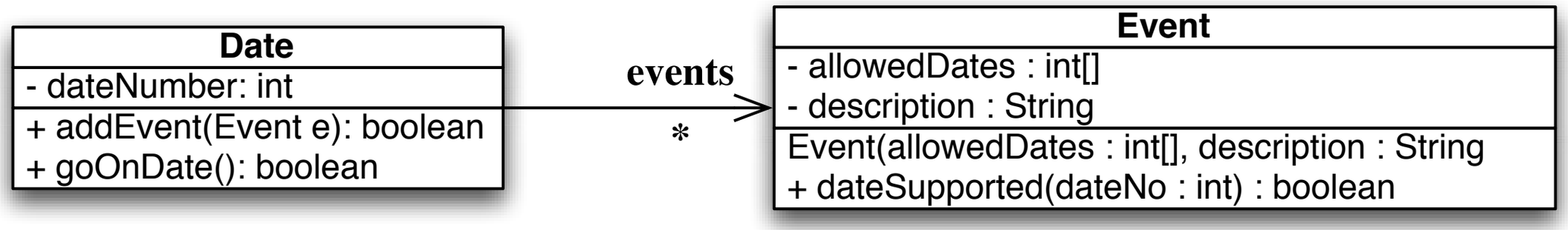
- In version 1.2, Rails eliminates this duplication for something called “resources”
- Now URLs look like this:
  - POST /people
  - GET /people/1
  - PUT /people/1
  - DELETE /people/1
- And the duplication is **logically** eliminated
  - Disclaimer: … but not **actually** eliminated… Web servers do not universally support PUT and DELETE “out of the box”. As a result, Rails uses POST
    - POST /people/1 ; Post-Semantics: DELETE

Other OO Principles

- **Classes are about behavior**
  - Emphasize the behavior of classes over the data
- **Encapsulate what varies**
  - Use classes to achieve information hiding in a design
- **One reason to change**
  - Promotes high cohesion in a design
- **Code to an Interface**
  - Promotes flexible AND extensible code
- **Open-Closed Principle**
  - Classes should be open for extension and closed for modification

Take CSCI 5448 for more details!

New iSwoon Design



Subclasses eliminated; Events now keep track of what Dates they are allowed on; When you add an event to a Date, Date calls Event.dateSupported() to validate it

You can easily add a new type of Event; just create a new instance of Event with a different description; nothing else changes! To add a new date, just increase the number

Impact on Tasks

With the right design, multiple tasks estimated to take days may take only one (or less than one)

Task: Create Send Flowers Event

Estimate: 2 days

Task: Create a Book Restaurant Event

Estimate: 3 days

Task: Add Order Cab Event

Estimate: 2 days

A great design helps you be more productive!

Discussion

- The underlying message of Chapter 5 is that everyone on your team needs to understand good OO A&D principles
- On a daily basis, you look for ways in which the design can be improved
  - Small changes can occur via refactoring
  - Large changes need to become tasks and tracked like all others
- You welcome such changes since they'll make life easier and more productive down the line

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

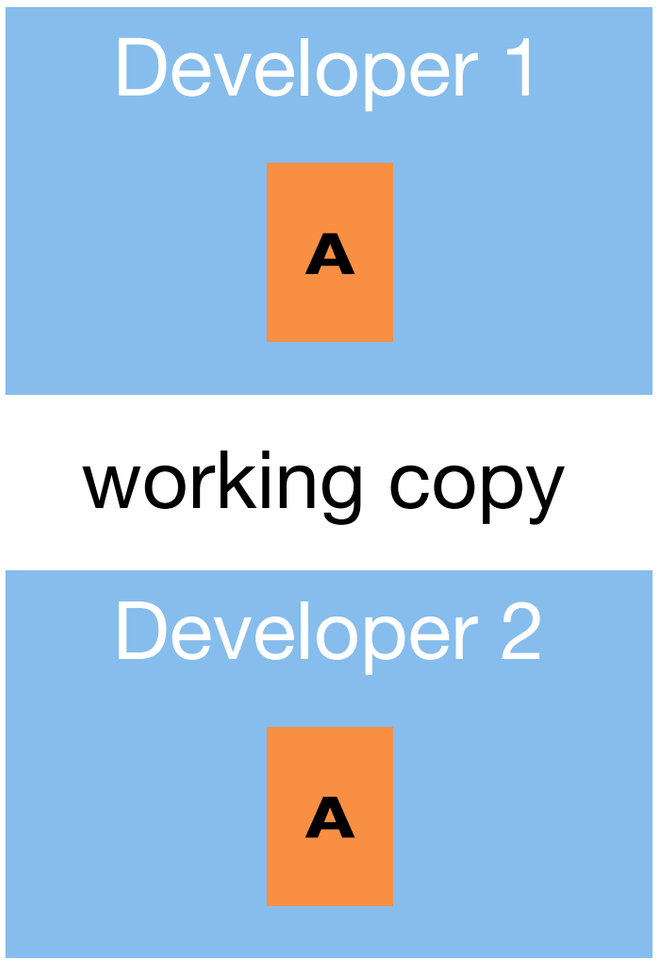
Developer 2

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

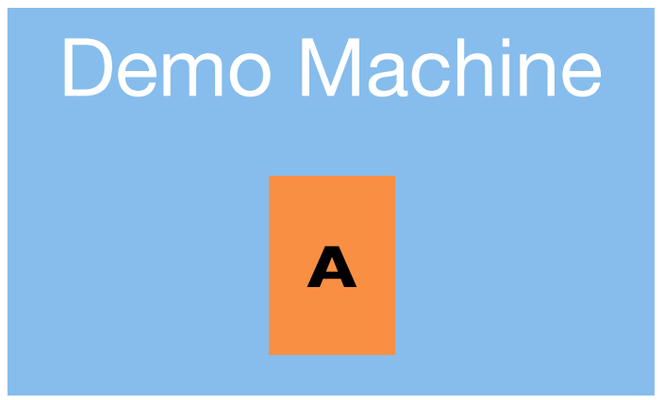
Demo Machine

A

Without a Net (IV)

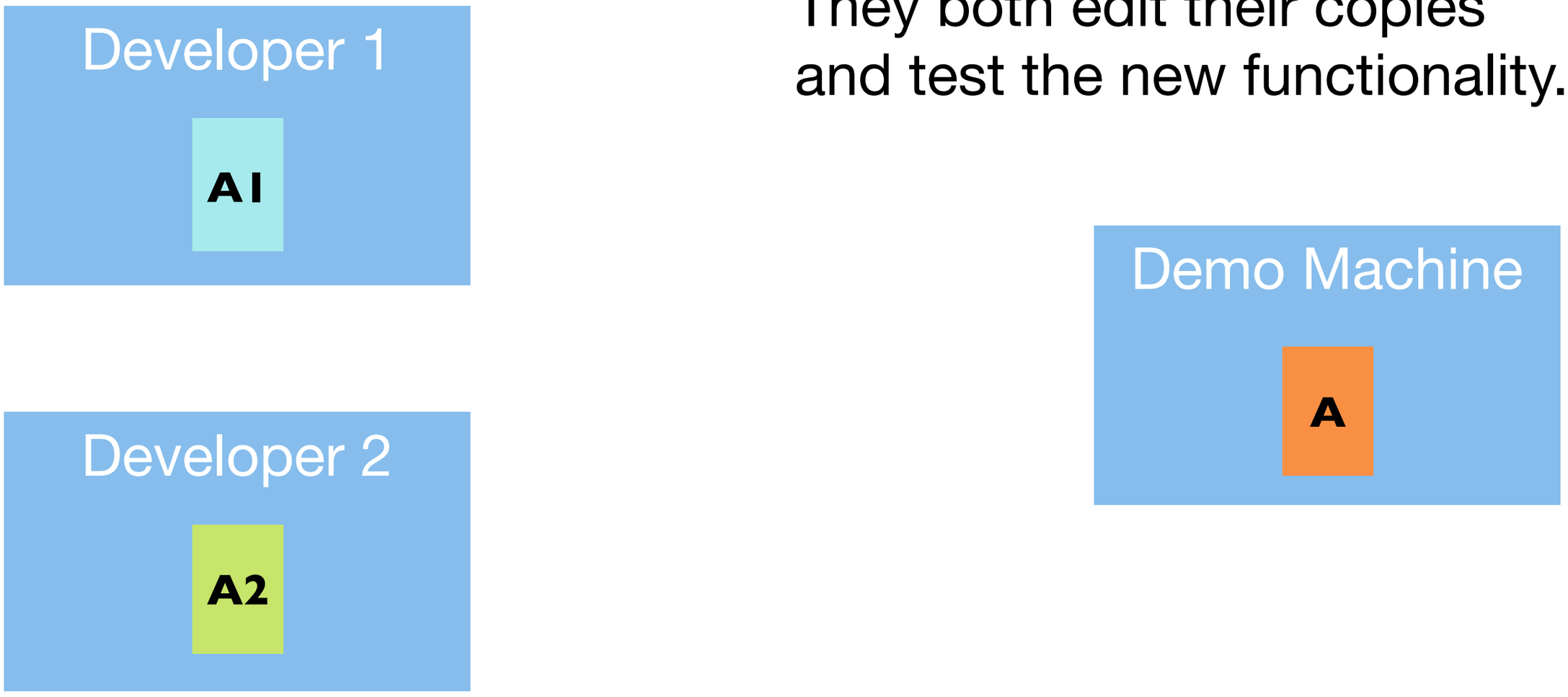


They both download the file from the demo machine, creating two working copies.



Without a Net (V)

They both edit their copies and test the new functionality.



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

A1

Developer 2

A2

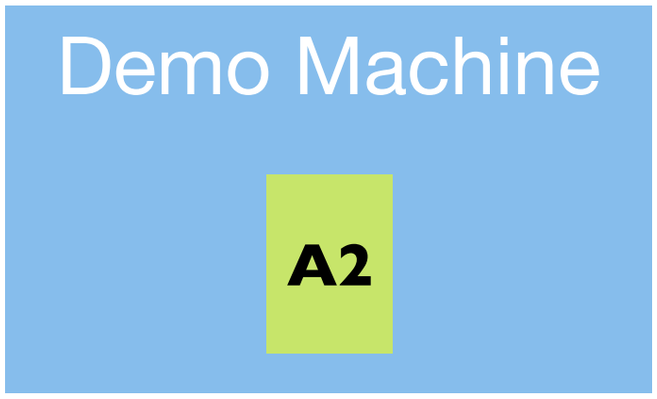
Developer 2 finishes second and uploads his copy to the demo machine.

Demo Machine

A2

Without a Net (VIII)

This is known as “last check in wins”

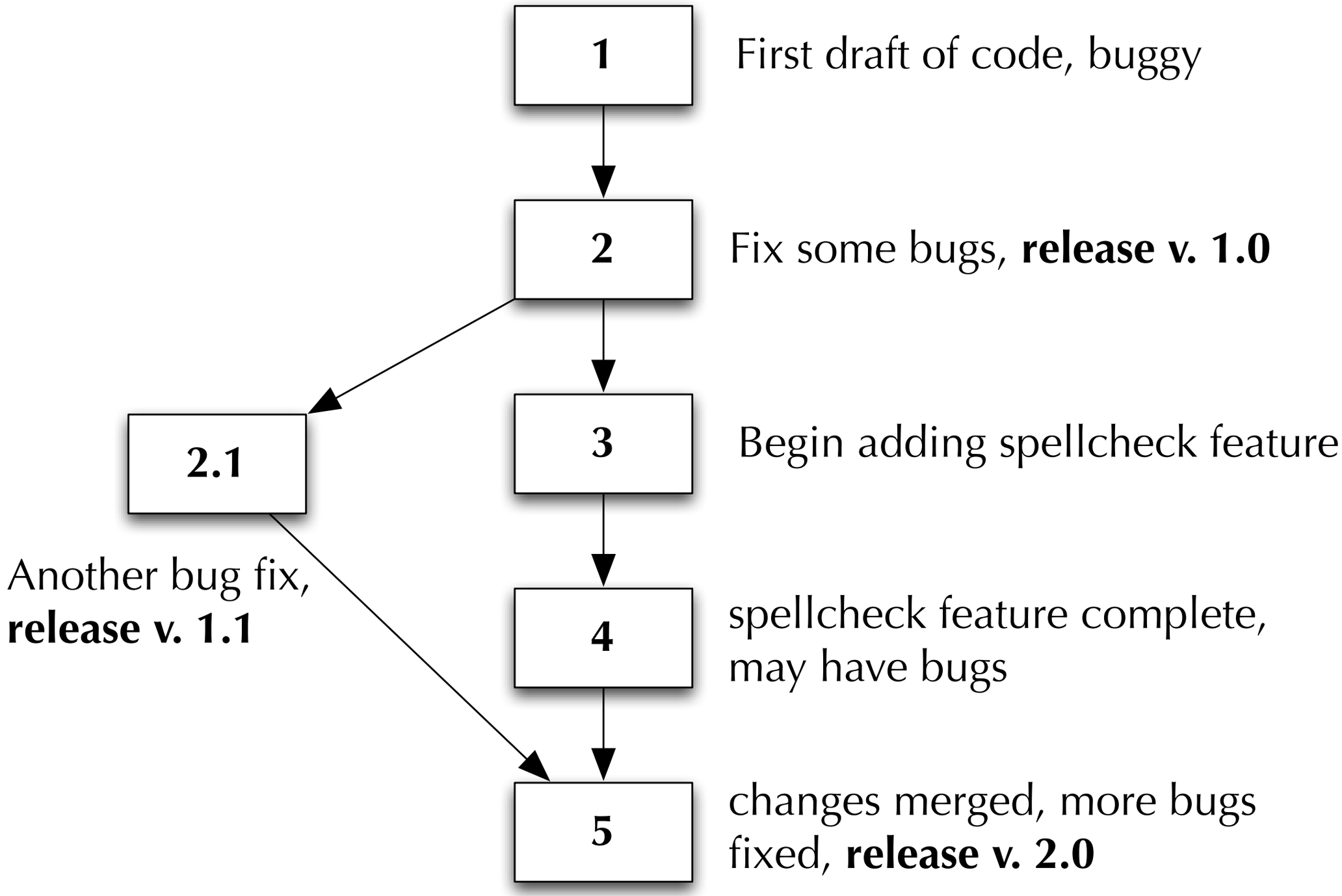


At best, developer 1’s work is simply “gone” when the demo is run; At worst, developer 1 checked in other changes, that cause developer 2’s work to crash when the demo is run.

Not Acceptable

- This type of uncertainty and instability is **simply not acceptable** in production software environments
  - That's where configuration management comes in
- The book uses 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

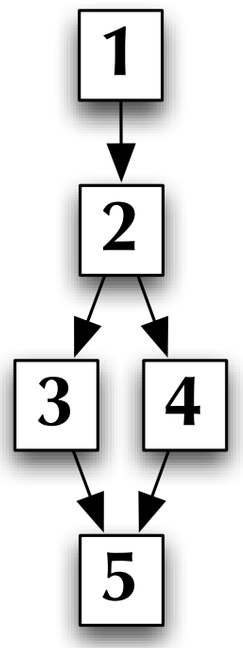

1
First draft of code, buggy
2
Fix some bugs, release v. 1.0
2.1
Another bug fix, release v. 1.1
3
Begin adding spellcheck feature
4
spellcheck feature complete, may have bugs
5
changes merged, more bugs fixed, release v. 2.0

Configuration Management

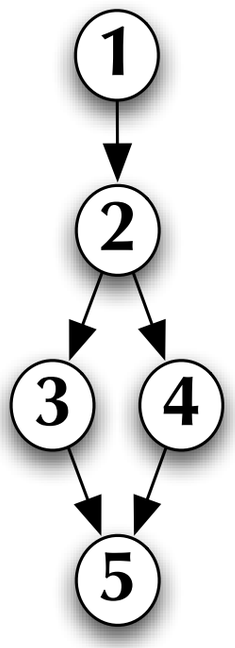
Particular versions of files are included in...

... different versions of a configuration

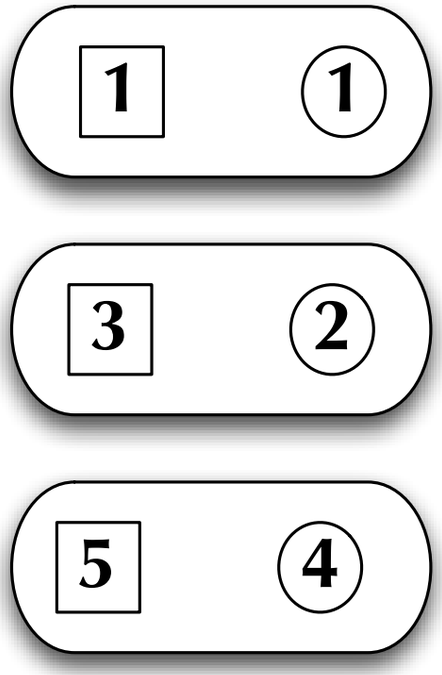
File A



File B



Configuration Z



With a Net (I)

Developer 1

Repository

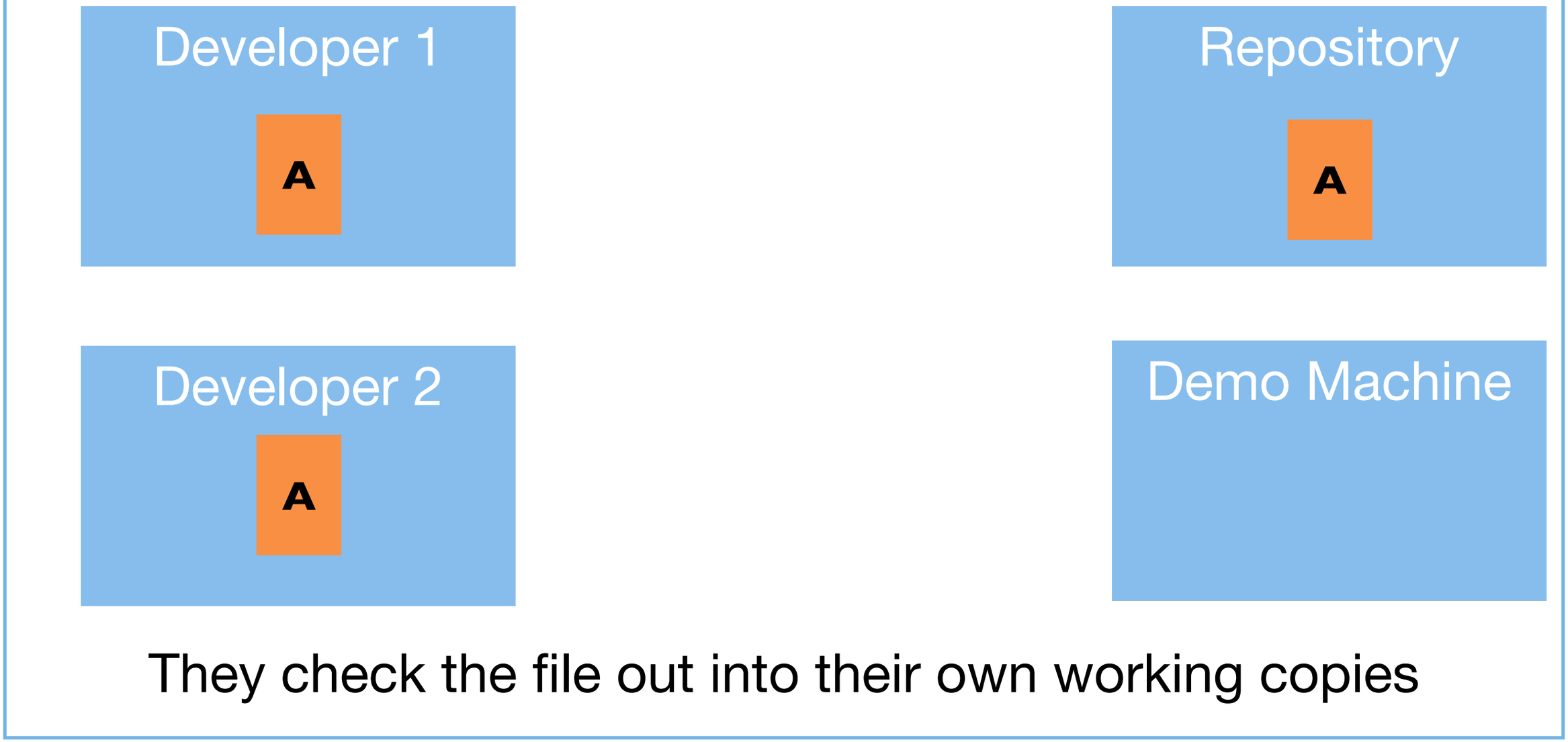
A

Developer 2

Demo Machine

Two developers need to modify the same file for separate tasks

With a Net (II)

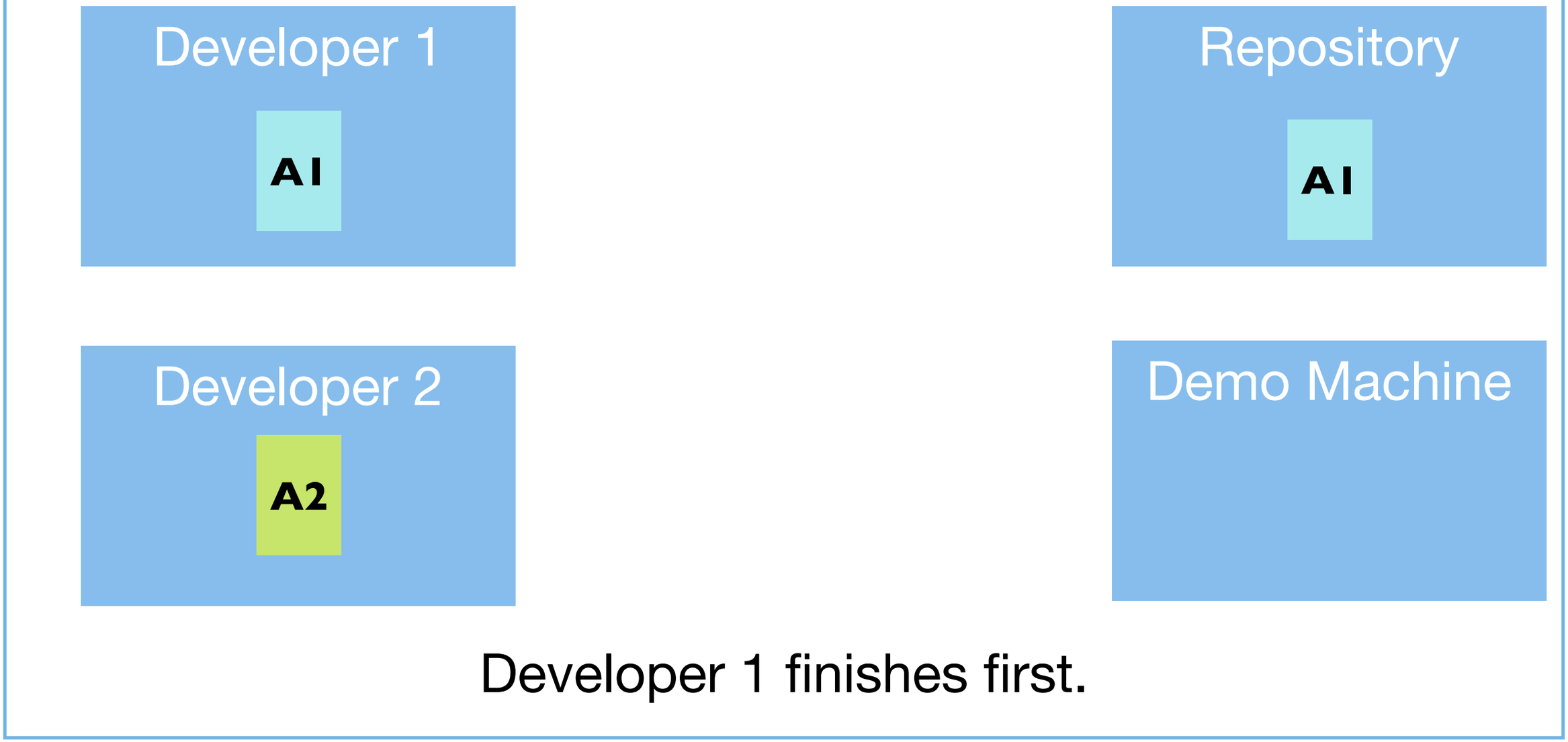


They check the file out into their own working copies

With a Net (III)

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)

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)

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

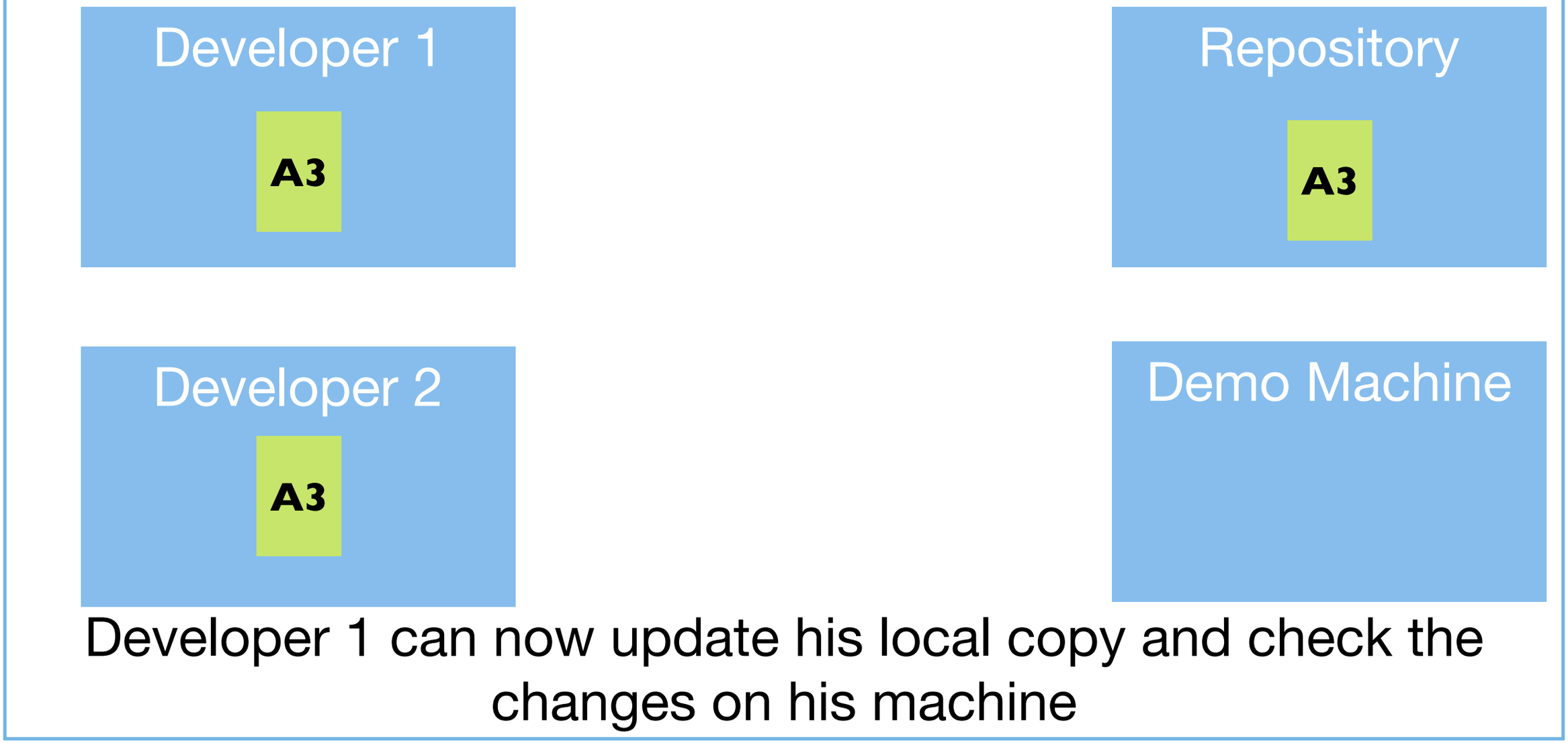
Demo Machine

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

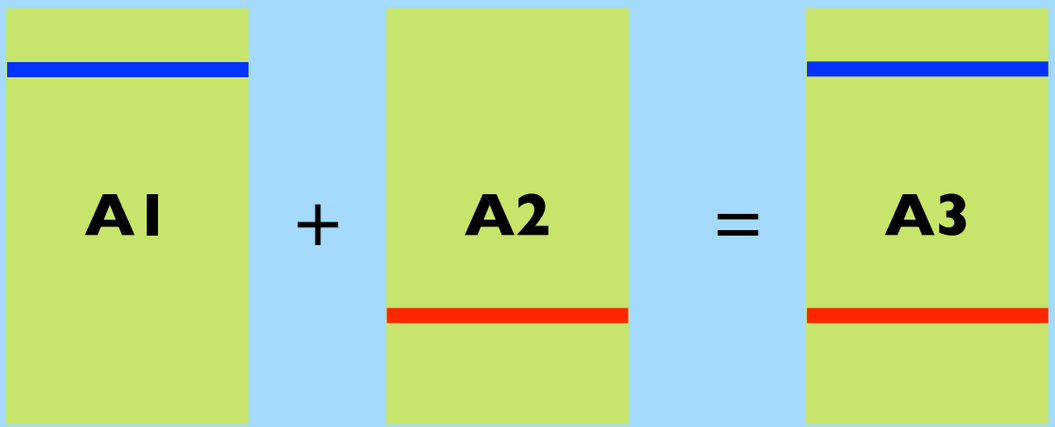
A3

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



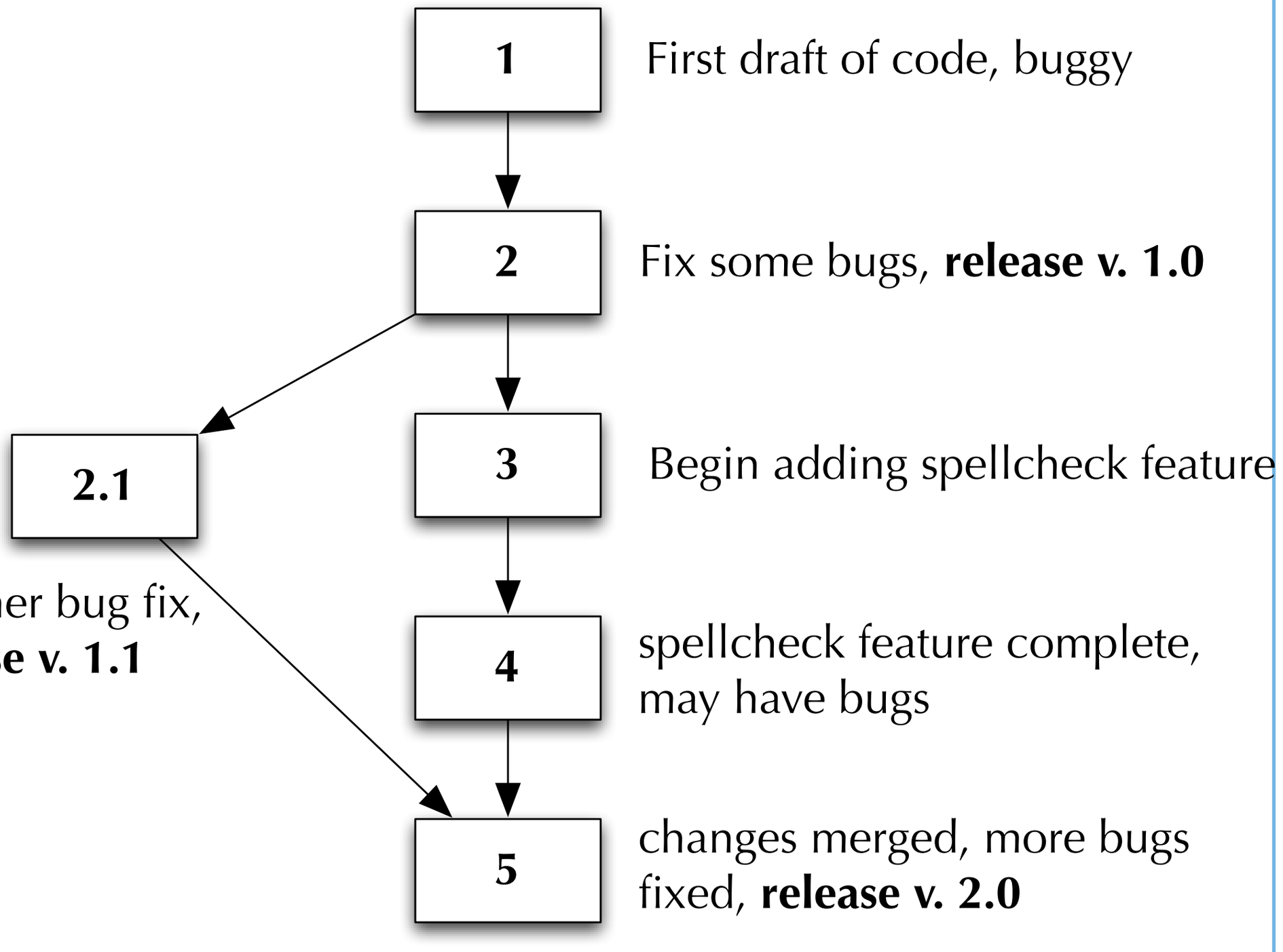
Tags, Branches, and Trunks, Oh My!

- 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)

- In the book, 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

Subversion Branches

- In subversion, tags and branches are made in the same way
  - by creating a copy of the trunk (or any specified revision)
  - the project can be huge, containing thousands of files, and it doesn't matter, branch/tag creation is completed in constant time and without the size of the repository changing
    - all that subversion does on a copy is note what the copy represents by pointing at the "source" version number

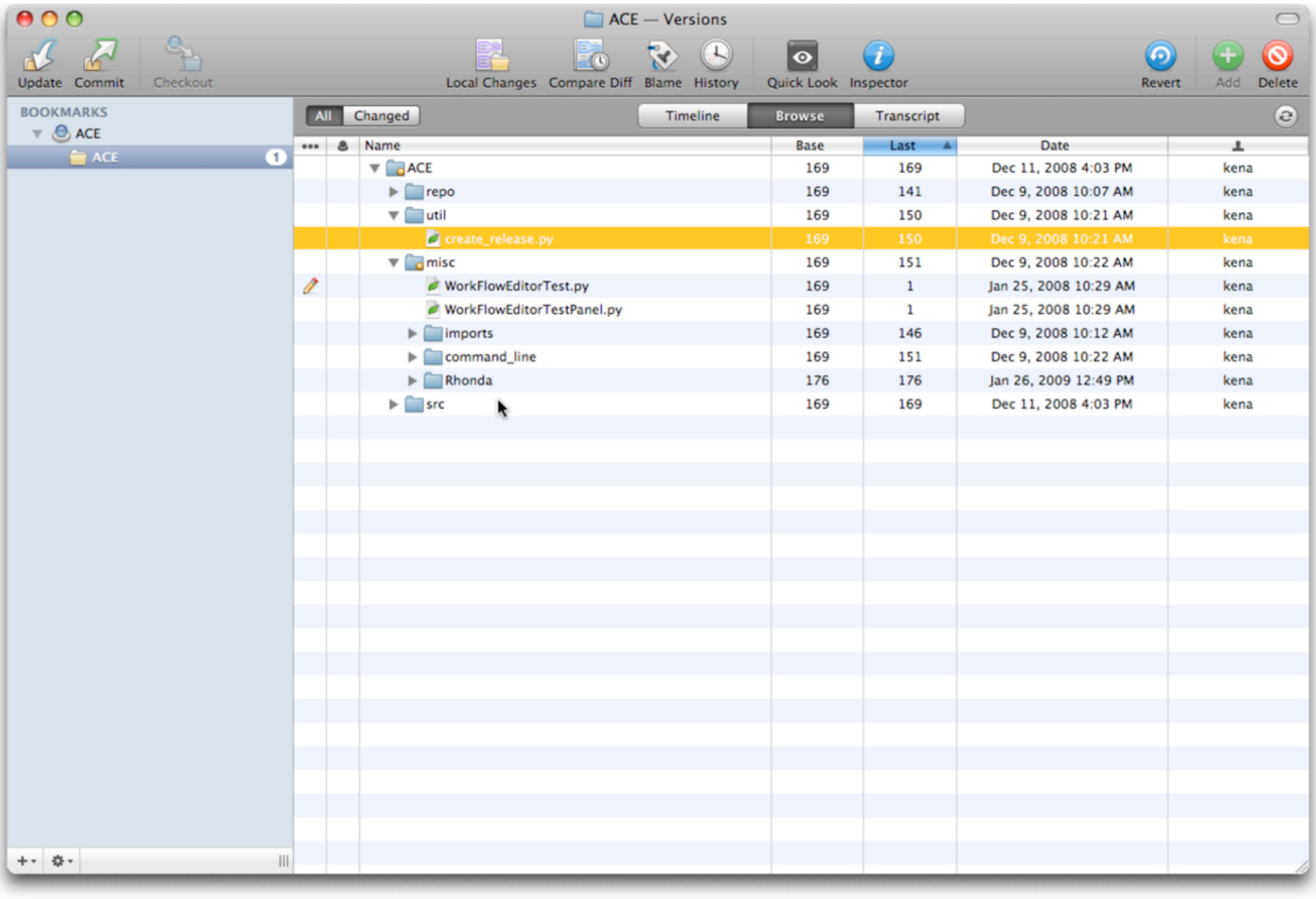
subversion cheat sheet

- Create a new repository
  - svnadmin create <repo>
- Check in new project
  - svn import <dir> <repo>/<project>/trunk
- Check out working copy
  - svn checkout <repo>/<project>/trunk <project>
- Check for updates
  - svn update
- Check in changes
  - svn commit
- Creating a tag
  - svn copy -r <version> <repo>/<project>/trunk <repo>/<project>/tags/<tag>
- Creating a branch
  - svn copy -r <version> <repo>/<project>/trunk <repo>/<project>/branches/<branch>
- tag/branch creation identical!

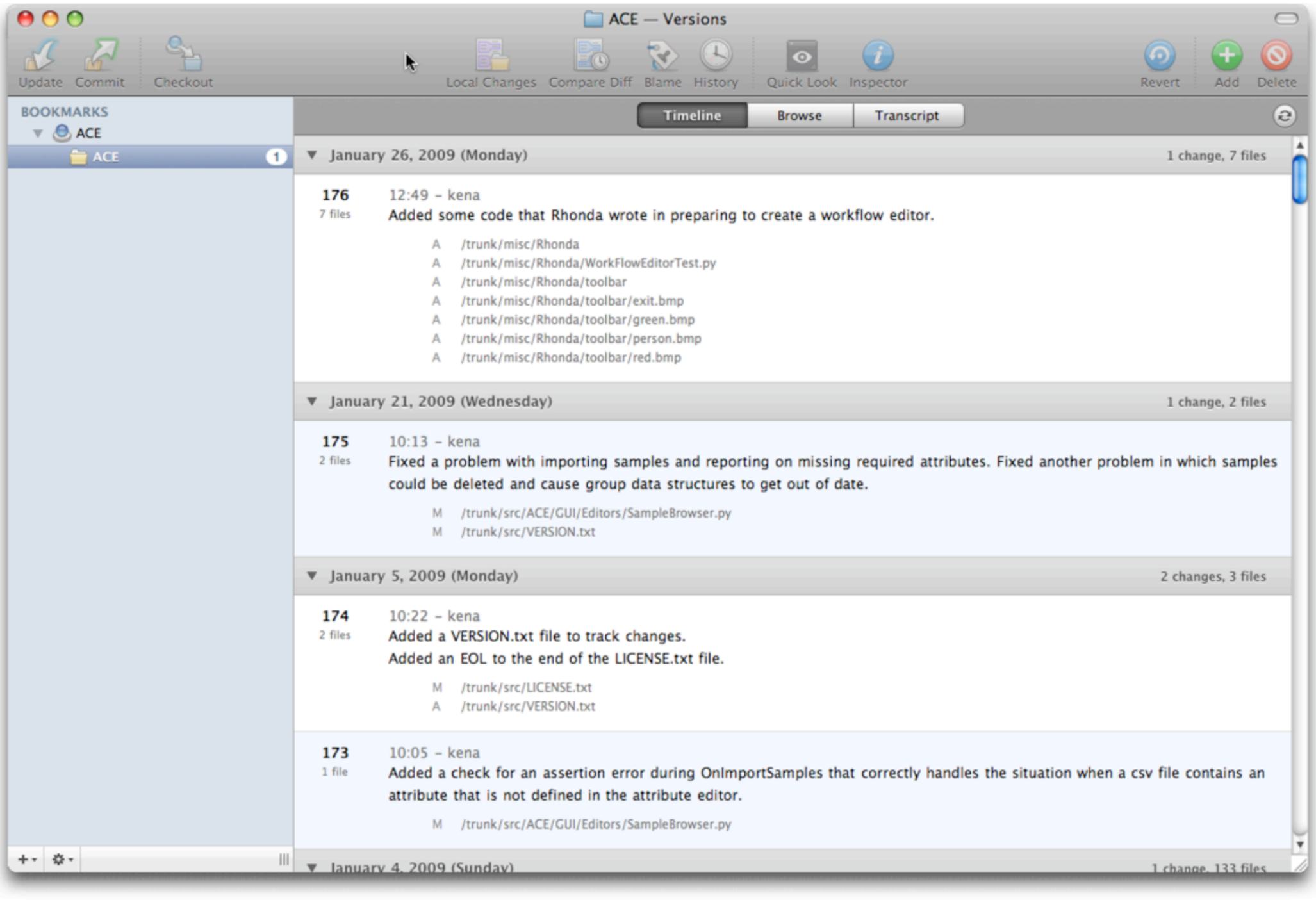
Many Graphical Tools

- Standalone Applications
  - Versions <http://versionsapp.com/>
- Integration into Development Environments
  - TextMate <http://macromates.com/>
- These are just examples, both for MacOS X, because that's my primary platform
  - but there are examples of these tools for multiple platforms

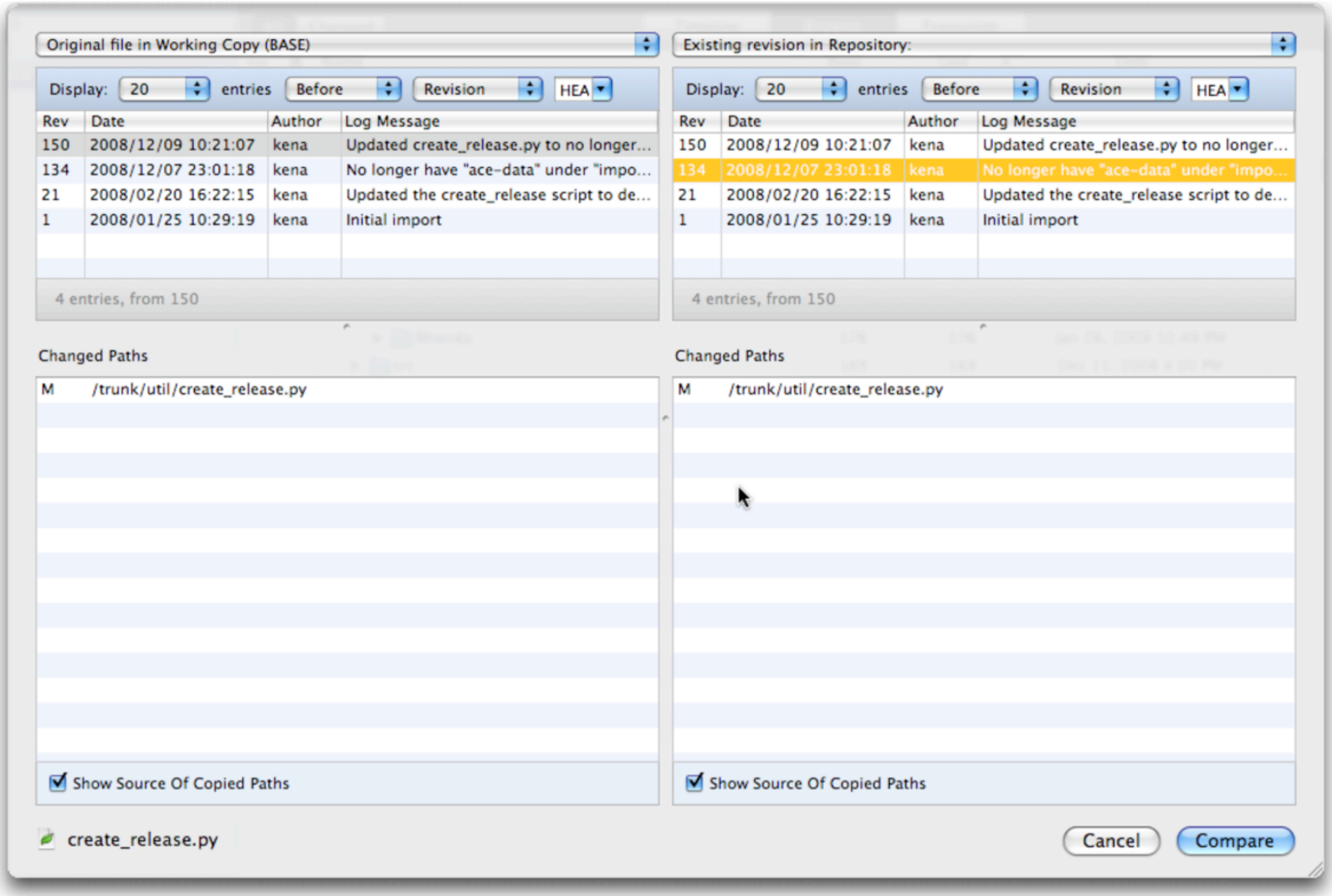
Versions: Browsing Project Files



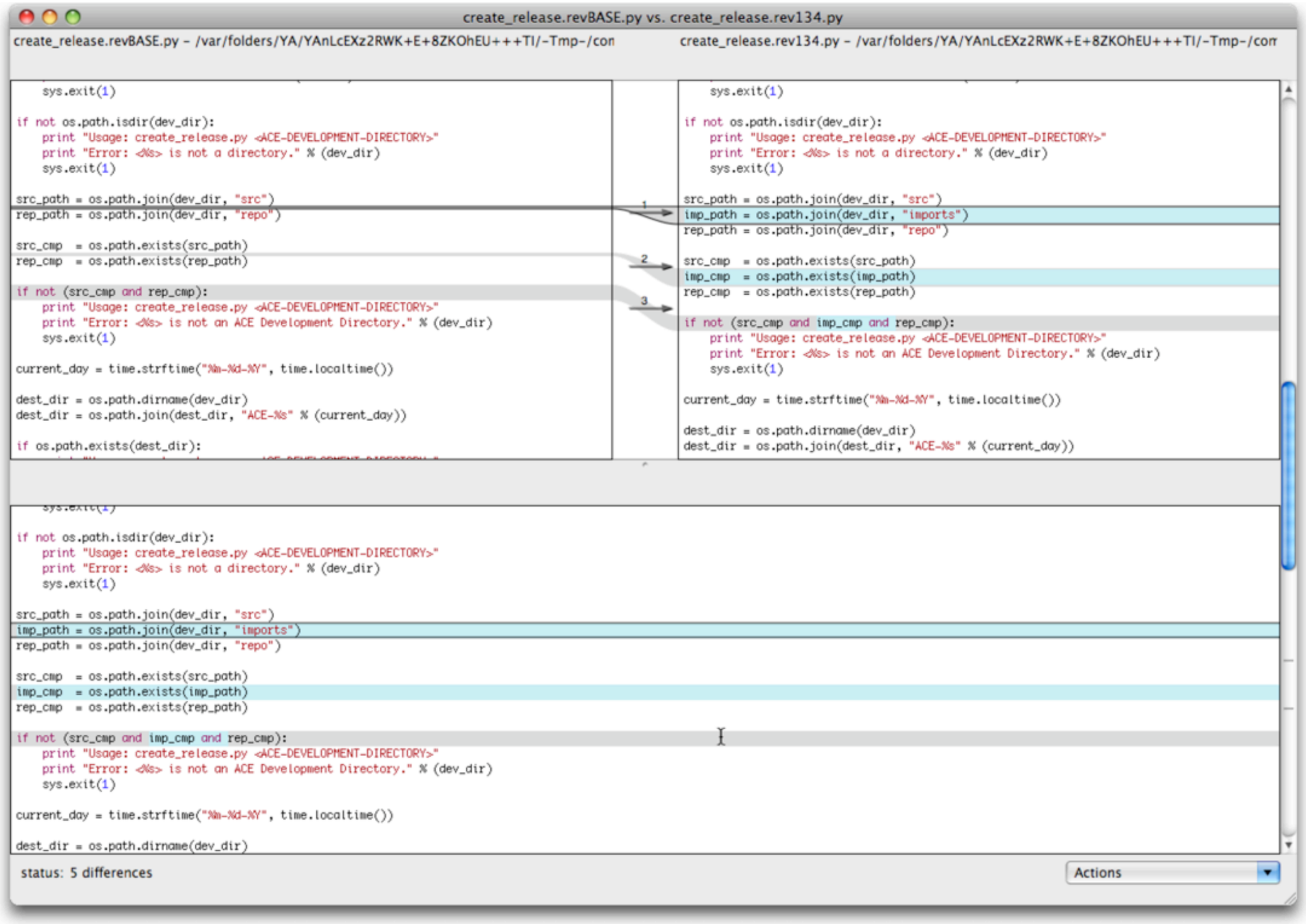
Versions: Viewing Log Messages



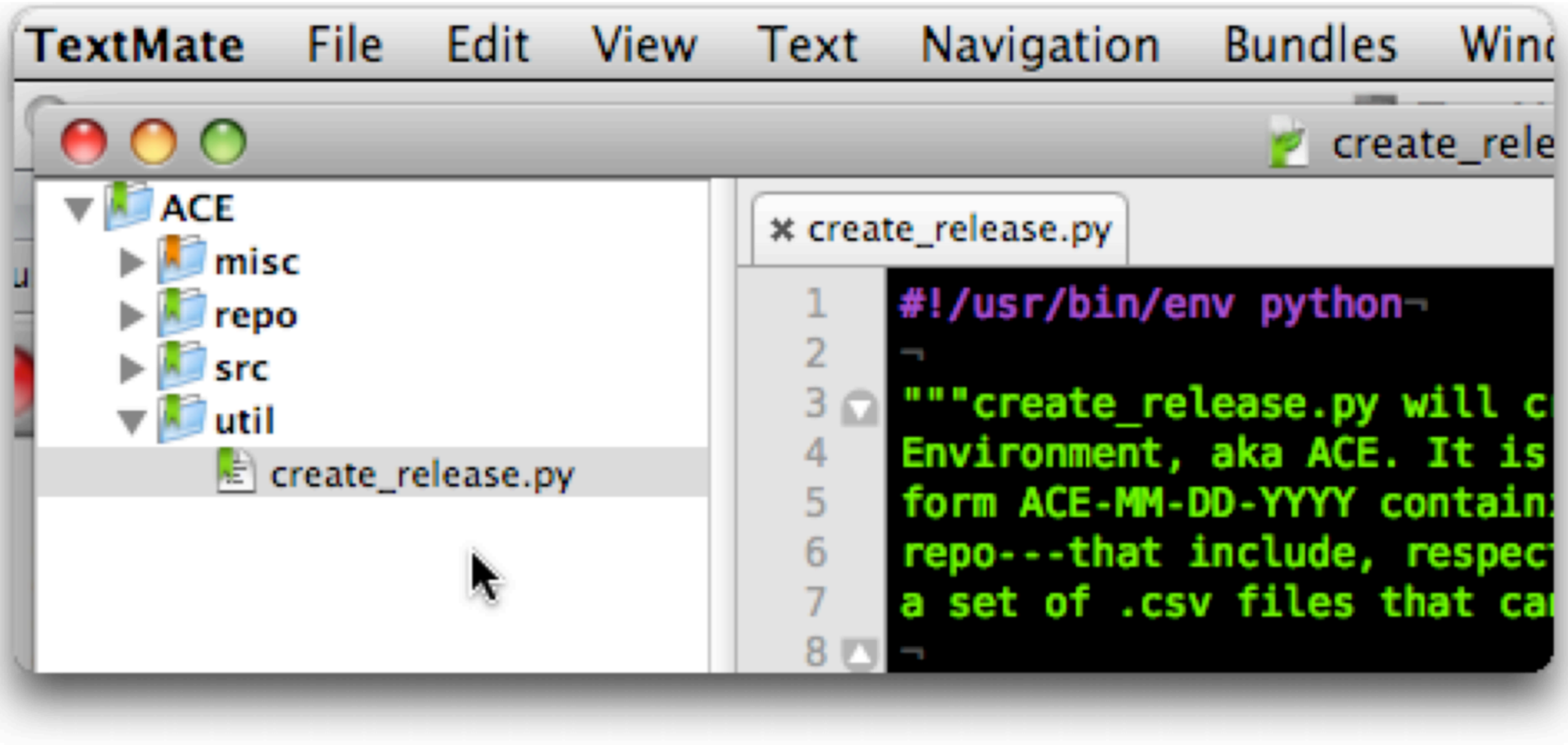
Versions: Selecting different versions of a file for comparison



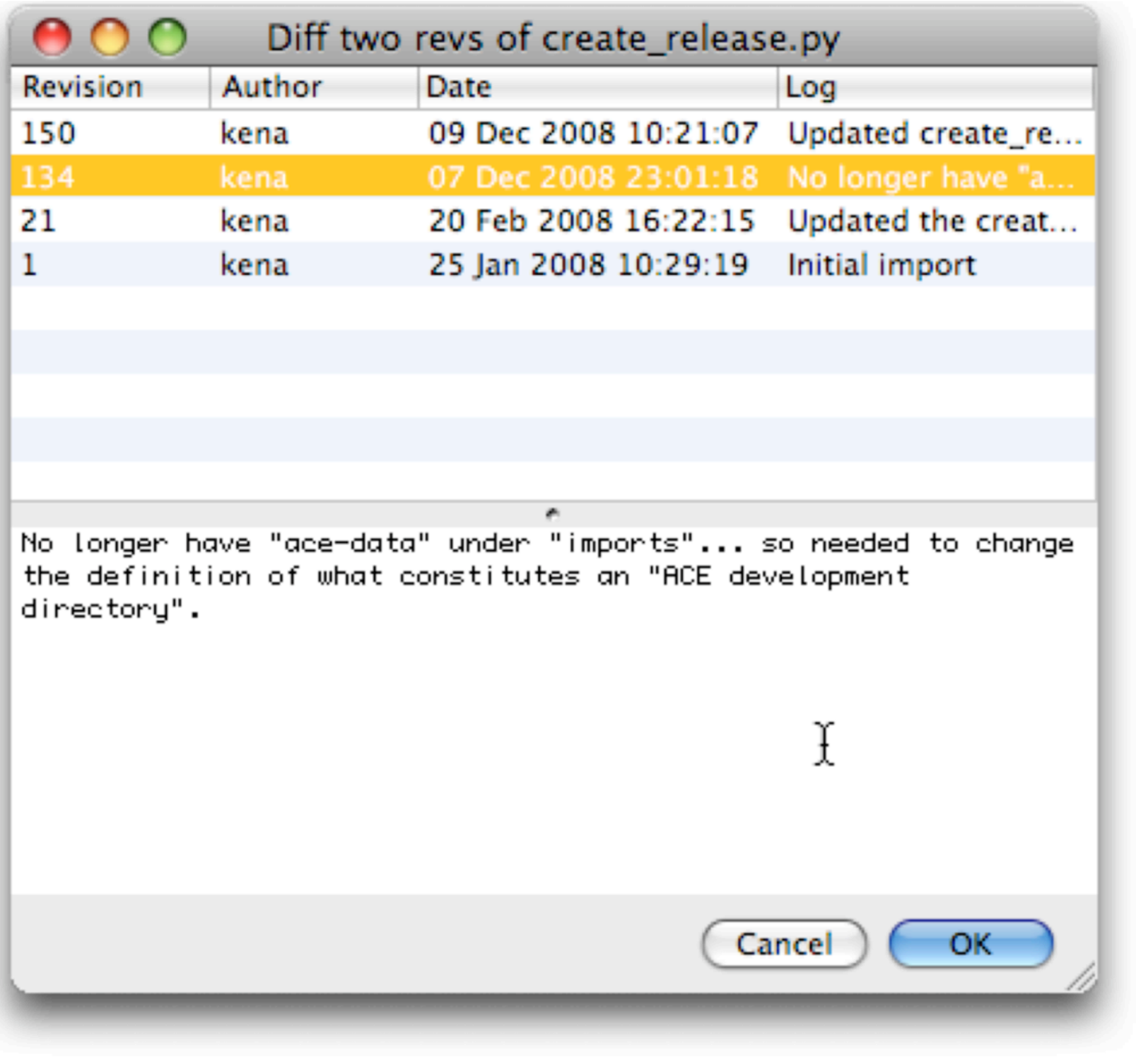
Versions: Using Apple's FileMerge to see differences



TextMate: Showing subversion information on files



TextMate: Selecting versions of a file for comparison



TextMate: Viewing the differences as a “patch” file

svndiff create_release.py.20535.0

```
PYTHONPATH: Undefined variable.
PYTHONPATH: Undefined variable.
PYTHONPATH: Undefined variable.
PYTHONPATH: Undefined variable.
PYTHONPATH: Undefined variable.
Index: /Users/kena/Desktop/ACE/util/create_release.py
===================================================================
--- /Users/kena/Desktop/ACE/util/create_release.py  (revision 134)
+++ /Users/kena/Desktop/ACE/util/create_release.py  (revision 150)
@@ -41,14 +41,12 @@
     sys.exit(1)

 src_path = os.path.join(dev_dir, "src")
-imp_path = os.path.join(dev_dir, "imports")
 rep_path = os.path.join(dev_dir, "repo")

 src_cmp  = os.path.exists(src_path)
-imp_cmp  = os.path.exists(imp_path)
 rep_cmp  = os.path.exists(rep_path)

-if not (src_cmp and imp_cmp and rep_cmp):
+if not (src_cmp and rep_cmp):
     print "Usage: create_release.py <ACE-DEVELOPMENT-DIRECTORY>"
     print "Error: <%s> is not an ACE Development Directory." % (dev_dir)
     sys.exit(1)
@@ -64,13 +62,11 @@
     sys.exit(1)

 dest_src_path = os.path.join(dest_dir, "src")
-dest_imp_path = os.path.join(dest_dir, "imports")
 dest_rep_path = os.path.join(dest_dir, "repo")

 os.mkdir(dest_dir)

 copytree(src_path, dest_src_path)
-copytree(imp_path, dest_imp_path)
 copytree(rep_path, dest_rep_path)

 # remove .DS_Store files and .pyc files
```

Line: 40 Column: 1 Plain Text Soft Tabs: 4

I think I like FileMerge a bit better! :-)

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; why?
  - 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>

git cheat sheet

- Create a new repository
  - git init
- Check in new project
  - git add . ; get commit
- Check out working copy
  - N/A
- Check for updates
  - N/A
- Check in changes
  - git add <file>; git commit
- Creating a tag
  - git tag <tag> <version>
- Creating a branch
  - git branch <branch>
- Collaboration
  - git clone <remote> <local>
  - Update
    - git pull <remote> <branch>
  - Commit
    - git push <remote>

Accidental Difficulties?

- svn
  - adds .svn dir to each directory in your repository
    - if you ever have supporting files stored in a directory of your repository that your application reads, it needs to be aware of the .svn dirs and ignore them
  - single repository version number even in the presence of multiple projects
    - <repo>/<project1>/trunk
    - <repo>/<project2>/trunk
      - Make a change in project 2 and the version number for project 1 is incremented!

Accidental Difficulties?

- git
  - The git FAQ seems to indicate that this tool has its own set of accidental difficulties (you can't avoid them!)
    - <http://git.or.cz/gitwiki/GitFaq>
  - I just don't have enough personal experience with git to detail them here.

Wrapping Up

- Software Design
  - Everyone needs to understand good design principles
  - SRP: Single Responsibility Principle
  - DRY: Don’t Repeat Yourself Principle
- Version Control & Configuration Management
  - Inject safety and confidence into software development
  - Lots of tools available
    - cvs, svn, git, Mercurial, Visual Source Safe

Coming Up

- Lecture 12: Model-Based Approach to Designing Concurrent Systems, Part 1
- Lecture 13: Model-Based Approach to Designing Concurrent Systems, Part 2
- Lecture 14 will be a review for the Midterm
  - Chapters 1-6 of Pilone & Miles
  - Chapters 1-4 of Breshears
  - Lecture 12 and 13