Version Control for the 2-Pizza Team: Merge Conflicts

(ELLS §9.5)

Armando Fox

Source & configuration management (SCM)

- What is it?
- *Version* (snapshot) code, docs, config files, etc. at key points in time
- Complete copy of every versioned file per snapshot
- Implementation: deltas? complete file copy? symlink?
- Why do it?
- Roll back if introduce bugs
- Separate deployed from development version of code
- Keep separate *branches* of development
- Documented history of who did what when
- Track what changed between revisions of a project

40 Years of Version Control

SCCS & RCS (1970s)

CVS (1986)

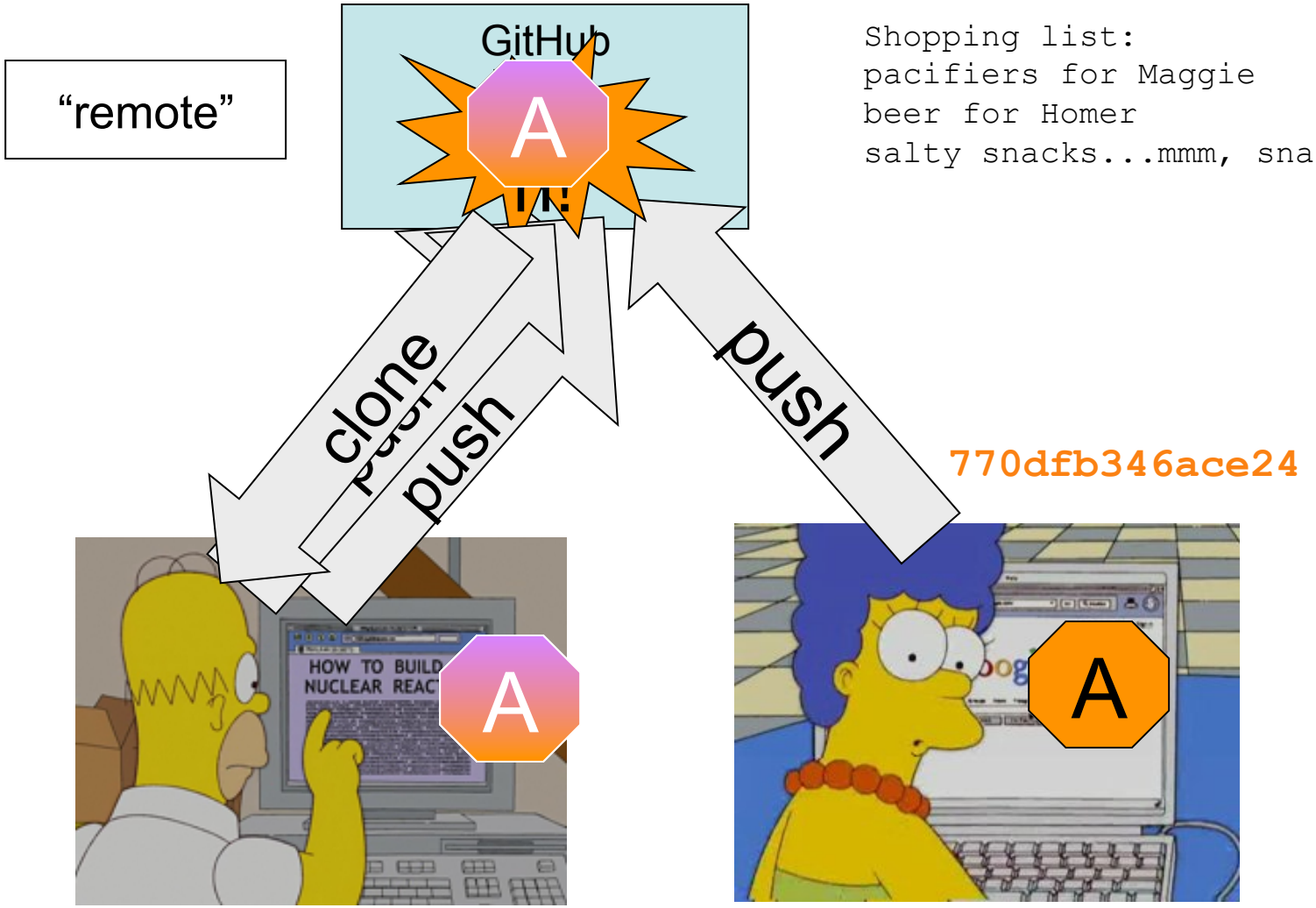
Subversion (2001)

Git (2005)

Image © TheSun.au

*

Merge Conflict




```
Shopping list:
pacifiers for Maggie
beer for Homer
salty snacks...mmm, snacks
```

Pull = Fetch + Merge

- Merge two repos = try to apply commits in either one to both
- Conflict if different changes to same file “too close” together
- `git pull = git pull origin master`
- Successful merge implies commit!
- Always commit before merging/pulling
- Commit early & often—small commits OK!

Commit: a *tree snapshot* identified by a commit-ID

- 40-digit hex hash (SHA-1), unique in the universe…but a pain
- use unique (in this repo) prefix, eg `770dfb`

`HEAD`: most recently committed version on current branch

`ORIG_HEAD`: right after a merge, points to pre-merged version

`HEAD~`***n***: n'th previous commit

`770dfb~2`: 2 commits before `770dfb`

`"master@{01-Sep-2012}"`: last commit prior to that date

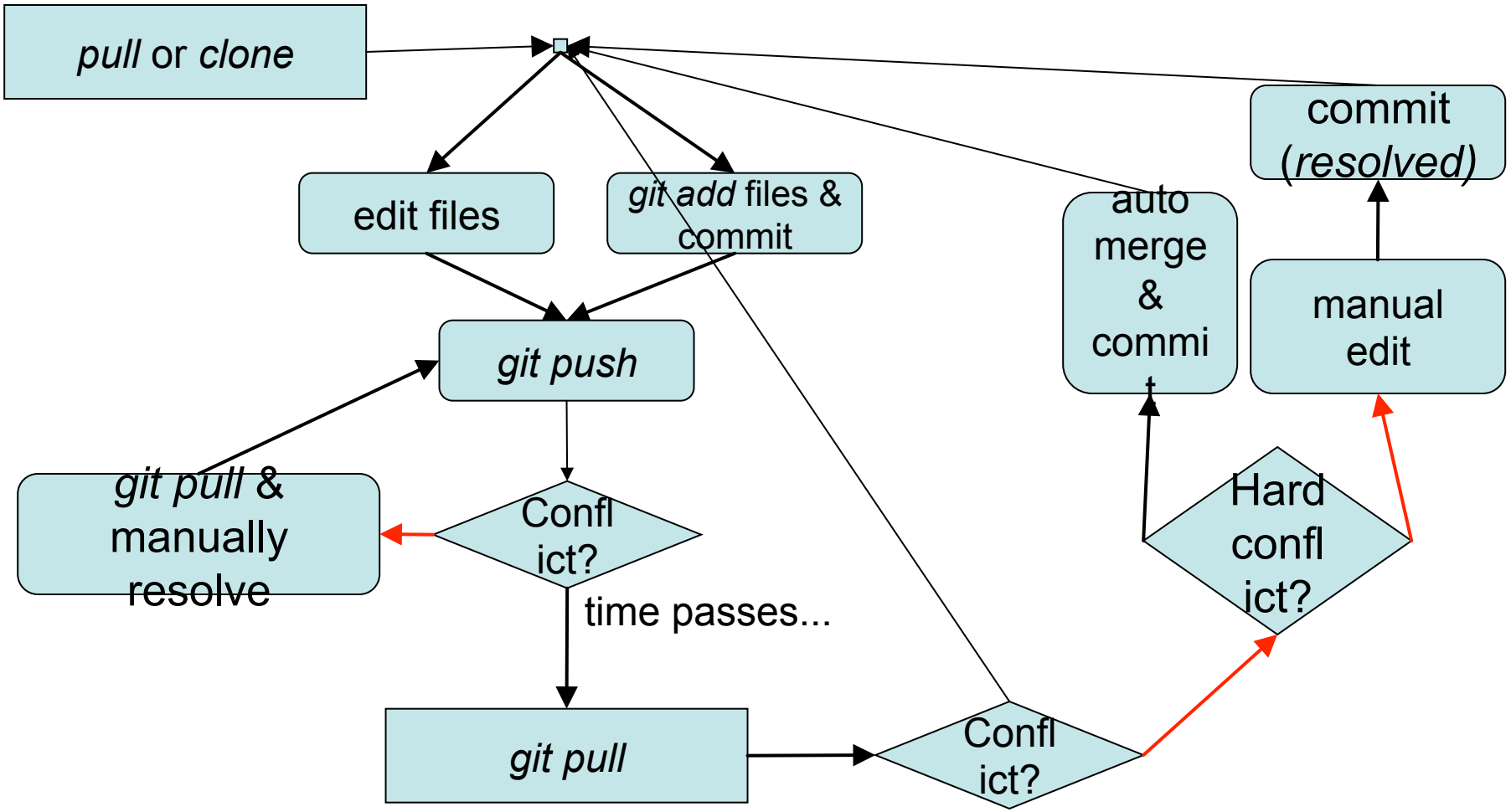
Undo!

```
git reset --hard ORIG_HEAD
git reset --hard HEAD
git checkout commit-id -- files…
```

- Comparing/sleuthing:

```
git diff commit-id -- files…
git diff "master@{01-Sep-12}" -- files
git blame files
git log files
```

Version control with conflicts



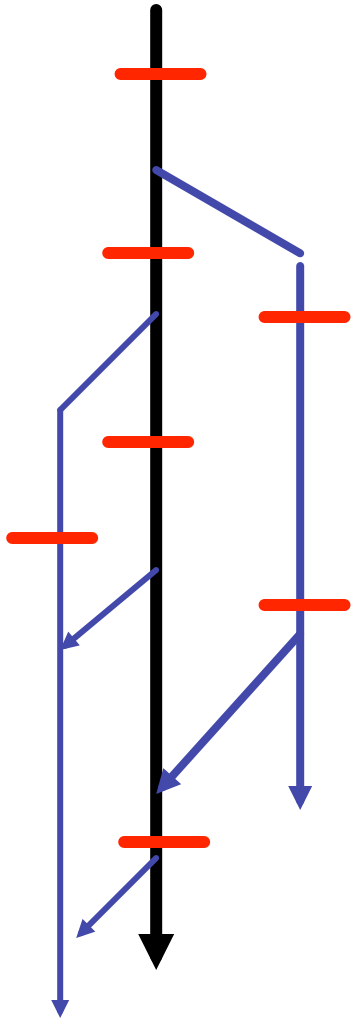
If you try to push to a remote and get a "non-fast-forward (error): failed to push some refs", which statement is FALSE?

- Some commits present at remote are not present on your local repo
- You need to do a merge/pull before you can complete the push
- You need to manually fix merge conflicts in one or more files
- Your local repo is out-of-date with respect to the remote

*

Branches

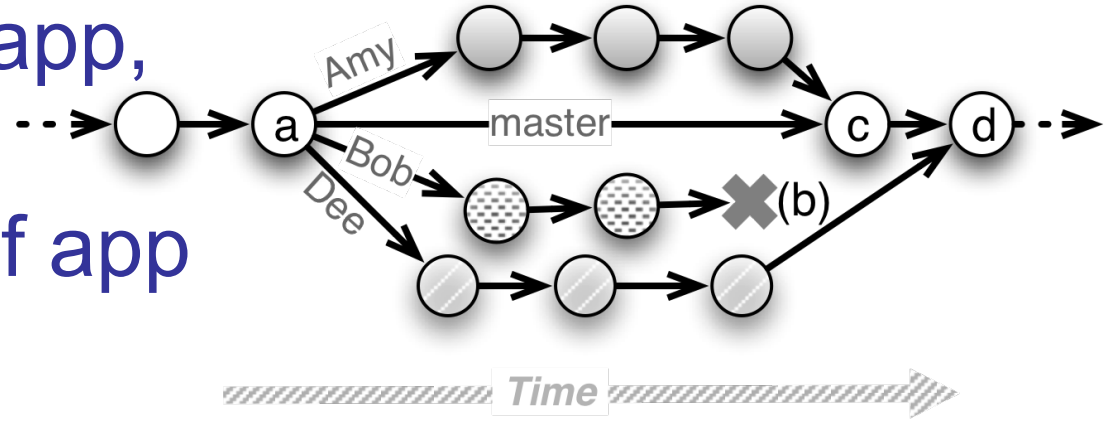
- Development **trunk** vs. **branches**
- trunk is called “master branch” in Git
- Creating branch is *cheap!*
- switch among branches: *checkout*
- Separate commit histories per *branch*
- *Merge* branch back into trunk
- ...or with *pushing* branch changes
- Most branches eventually die
- Killer use case for agile SaaS: *branch per feature*



Creating new features without disrupting working code

1. To work on a new feature, create new branch *just for that feature*
   - many features can be in progress at same time
2. Use branch *only* for changes needed for *this feature*, then merge into trunk
3. Back out this feature ⇔ undo this merge

In well-factored app,
1 feature shouldn't
touch many parts of app



Mechanics

- Create new branch & switch to it

```
git branch CoolNewFeature
git checkout CoolNewFeature
```
←*current branch*

- Edit, add, make commits, etc. on branch
- Push branch to origin repo (optional):

```
git push origin CoolNewFeature
```

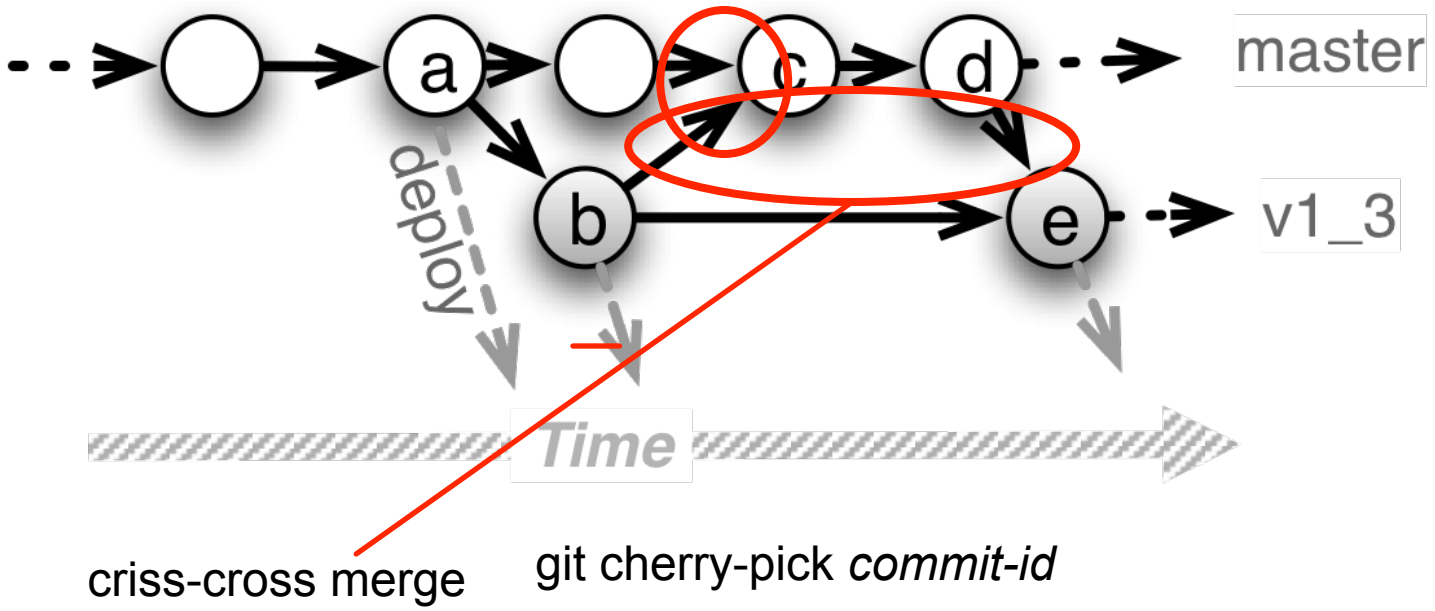
- creates *tracking branch* on remote repo
- Switch back to master, and merge:

```
git checkout master
git merge CoolNewFeature
```
← *warning!!*

Branches & Deployment

- Feature branches should be short-lived
- otherwise, drift out of sync with master, and hard to reconcile
- git rebase can be used to “incrementally” merge
- git cherry-pick can be used to merge only specific commits
- “Deploy from master” is most common

Release/bugfix branches and cherry-picking commits



Rationale: release branch is a stable place to do incremental bug fixes

Branch vs. Fork

- Git supports *fork & pull* collaboration model
- branch: create temporary branch in *this repo*
- merge: fold branch changes into master (or into another branch)
- fork: clone *entire repo*
- pull request: I ask you to pull specific commits from my forked repo

Gitfalls ☺

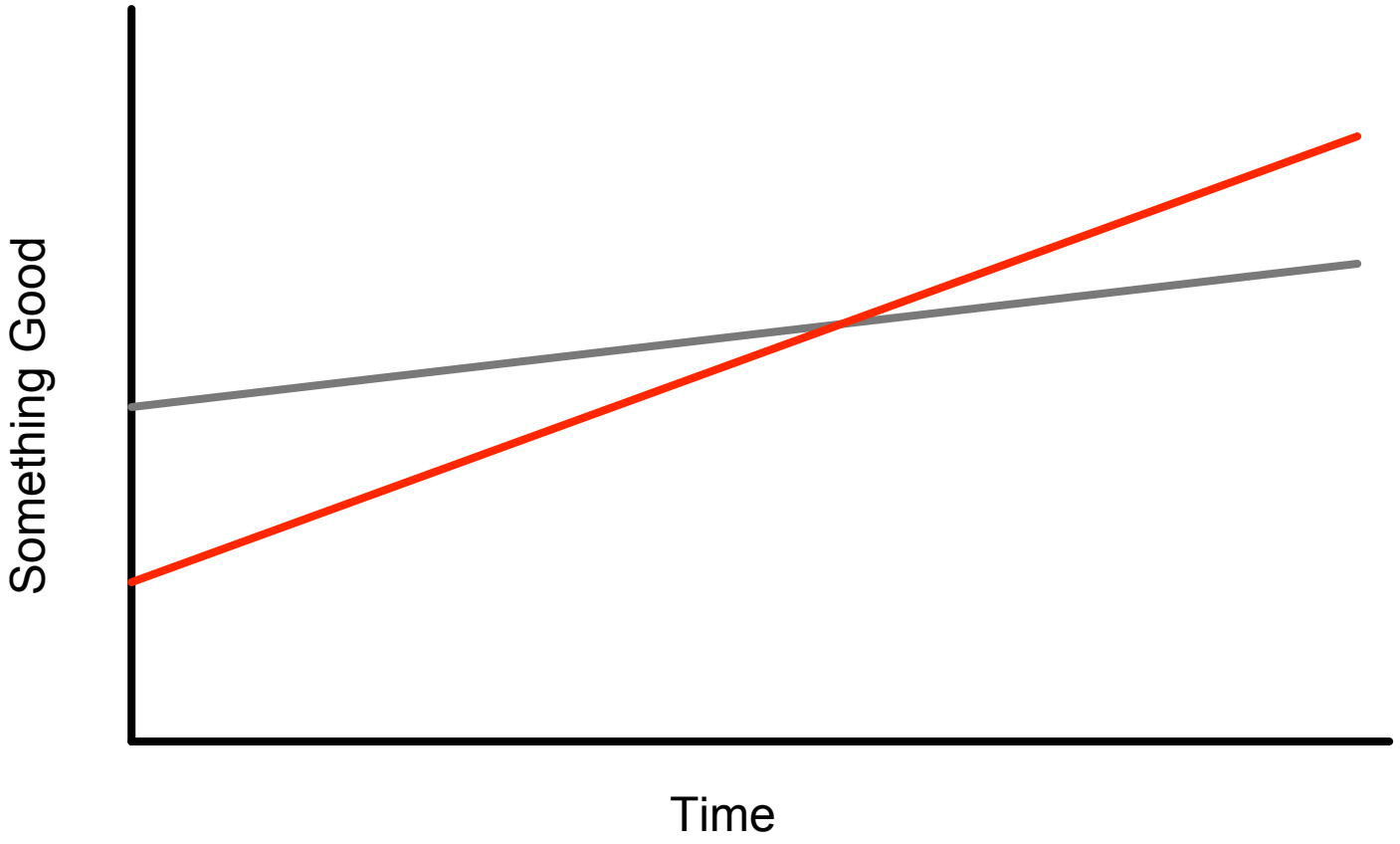
Stomping on changes after doing a merge or switching branches

Making “simple” changes directly on master branch

If separate sub-teams are assigned to work on *release bug fixes* and *new features,* you will need to use:

- Branch per release
- Branch per feature
- Branch per release + Branch per feature
- Any of these will work

Day Thoughts: A Little Bit of Slope Makes Up For a Lot of Y-intercept



*

Fixing Bugs: The Five R's*(ELLS §9.7)*

Armando Fox

No Bug Fix Without a Test!

- **R**eport
- **R**eproduce and/or **R**eclassify
- **R**egression test
- **R**epair
- **R**elease the fix (commit and/or deploy)

- Even in non-agile organizations
- But, existing agile processes can be *adapted* to bug fixes

*

Report

- Pivotal Tracker
- bug = 0-points story (but not zero effort!!)
- automation: GitHub *service hooks* can be configured to mark Tracker story "delivered" when properly-annotated commit is pushed
- GitHub "issues" feature
- Full-featured bug tracking, eg Bugzilla
- *Use the simplest tool that works for your team & project scope*

Reclassify? or Reproduce + Repair?

- Reclassify as “not a bug” or “won’t be fixed”
- Reproduce with *simplest possible* test, and add it to regression
- minimize preconditions (e.g. before blocks in RSpec, Given or Background steps in Cuke)
- Release: may mean either “pushed” or “deployed”

Which type of bug could you *not* create a regression test for?

- Bug that depends on behavior of external service
- Bug that depends on time of day, day of year, etc.
- Bug that is statistical in nature ("Player must win jackpot at least X% of the time")
- You can reproduce any of these dependencies in regression tests