WHAT THE USER WANTS…? MOVING FROM IDEAS TO ACTION

INTRODUCTION TO BEHAVIOR-DRIVEN DESIGN AND USER STORIES

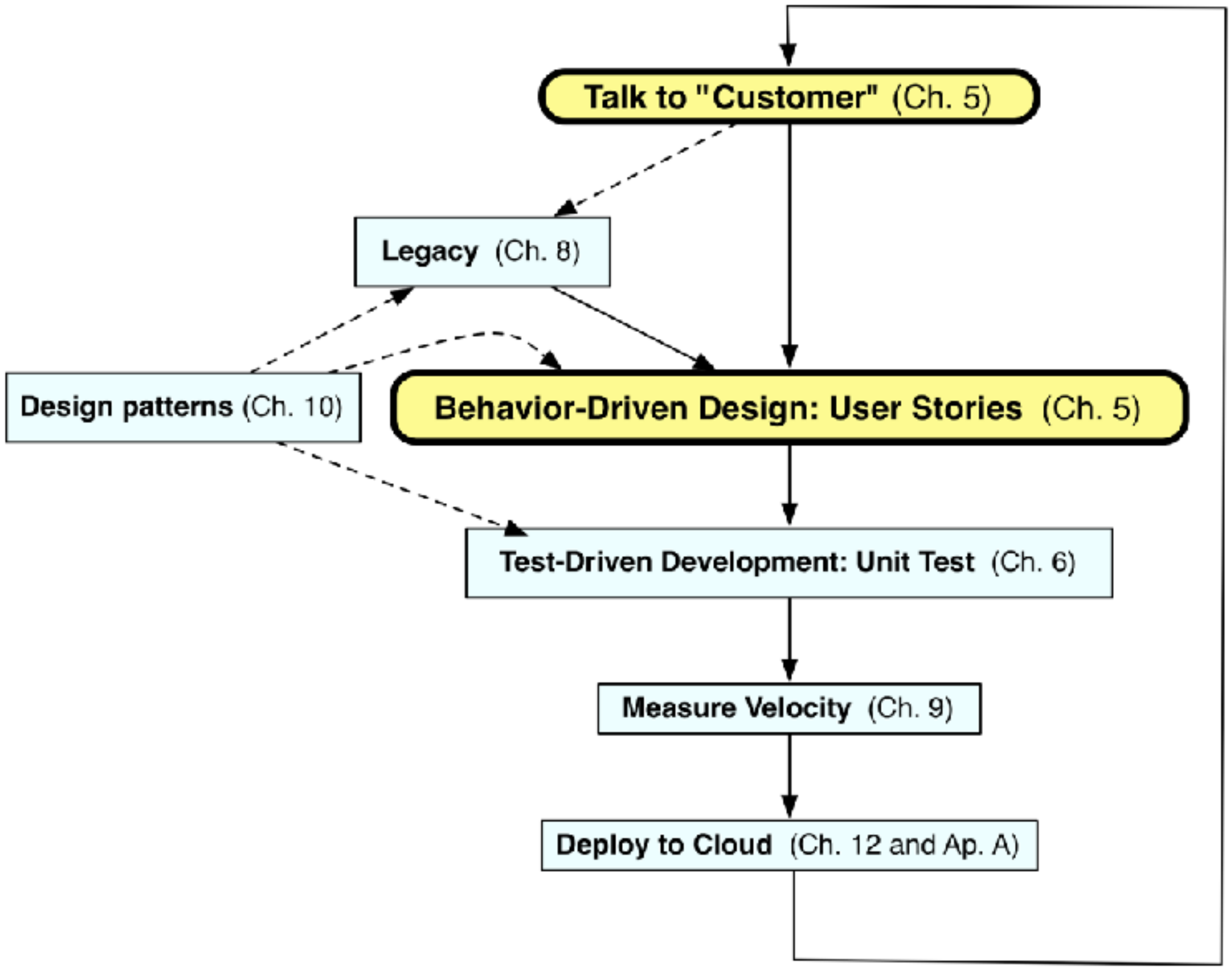
WHY DO SW PROJECTS FAIL?

- Don't do what customers want
- Or projects are late
- Or over budget
- Or hard to maintain and evolve
- Or all of the above
- Inspired Agile Lifecycle

AGILE LIFECYCLE

- Work closely, continuously with stakeholders to develop requirements, tests
  - Users, customers, developers, maintenance programmers, operators, project managers, …
- Maintain working prototype while deploying new features every **iteration**
  - Typically every 1 or 2 weeks
  - Instead of 5 major phases, each months long
- Check with stakeholders on what's next,
  to validate building right thing (vs. verify)

AGILE ITERATION

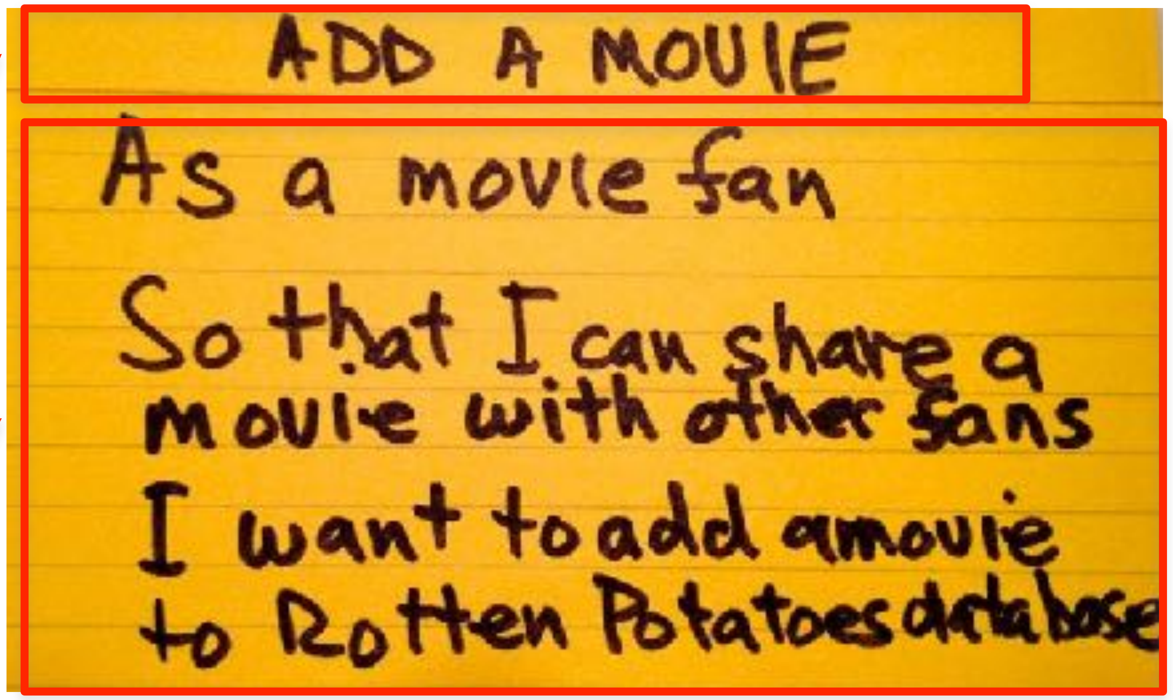

Talk to "Customer" (Ch. 5)
Legacy (Ch. 8)
Design patterns (Ch. 10)
Behavior-Driven Design: User Stories (Ch. 5)
Test-Driven Development: Unit Test (Ch. 6)
Measure Velocity (Ch. 9)
Deploy to Cloud (Ch. 12 and Ap. A)

BEHAVIOR-DRIVEN DESIGN (BDD)

- BDD asks questions about behavior of app *before and during development* to reduce miscommunication
- Requirements written down as *user stories*
  - Lightweight descriptions of how app used
- BDD concentrates on *behavior* of app vs. *implementation* of app
  - Test Driven Design or TDD (next chapter) tests implementation

USER STORIES

- 1-3 sentences in everyday language
  - Fits on 3" x 5" index card
  - Written by/with customer
- "Connextra" format:
  - Feature name
  - As a [kind of stakeholder],
    So that [I can achieve some goal],
    I want to [do some task]
  - 3 phrases must be there, can be in any order
- Idea: user story can be formulated as *acceptance test before* code is written



ADD A MOVIE

As a movie fan
So that I can share a movie with other fans
I want to add a movie to Rotten Potatoes database

WHY 3X5 CARDS?

- (from User Interface community)
- Nonthreatening => all stakeholders participate in brainstorming
- Easy to rearrange => all stakeholders participate in prioritization
- Since stories must be short, easy to change during development
    - As often get new insights during development

DIFFERENT STAKEHOLDERS MAY DESCRIBE BEHAVIOR DIFFERENTLY

- *See which of my friends are going to a show*
  - As a theatergoer
  - So that I can enjoy the show with my friends
  - I want to see which of my Facebook friends are attending a given show

---

- *Show patron's Facebook friends*
  - As a box office manager
  - So that I can induce a patron to buy a ticket
  - I want to show her which of her Facebook friends are going to a given show

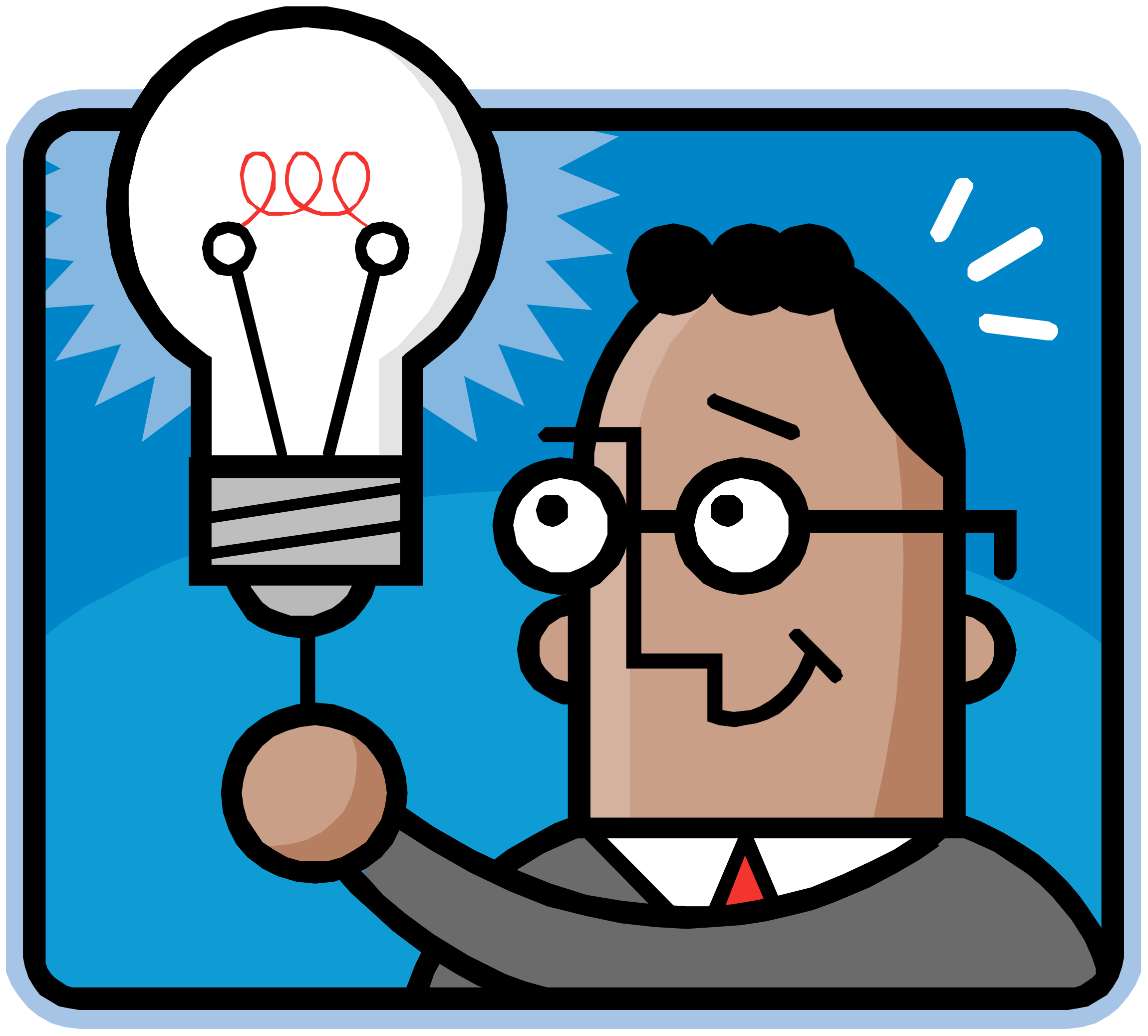
PRODUCT BACKLOG

- Real systems have 100s of user stories
- *Backlog*: User Stories not yet completed
  - (We'll see Backlog again with Zenhub)
- Prioritize so most valuable items highest
- Organize so they match SW releases over time

SMART USER STORIES

SMART STORIES

- **S**pecific
- **M**easurable
- **A**chievable (ideally, implement in 1 iteration)
- **R**elevant (“the 5 why’s”)
- **T**imeboxed (know when to give up)



SPECIFIC & MEASURABLE

- Each scenario testable
  - Implies known good input and expected results exist
- Anti-example: "UI should be user-friendly"
- Example: Given/When/Then.
  - *Given* some specific starting condition(s),
  - *When* I do X,
  - *Then* one or more specific thing(s) should happen

ACHIEVABLE

- Complete in 1 iteration
- If can’t deliver feature in 1iteration, deliver subset of stories
  - Always aim for working code @ end of iteration

TIMEBOXED

- Estimate what's achievable using *velocity*
  - Each story assigned *points* (1-3) based on progress amount
  - Velocity = Points completed / iteration
  - Use measured velocity to plan future i
    points per story
- Pivotal Tracker (later) tracks velocity

RELEVANT: “BUSINESS VALUE”

- Ask “Why?” recursively until discover business value, or kill the story:
  - Protect revenue
  - Increase revenue
  - Manage cost
  - Increase brand value
  - Making the product remarkable
  - Providing more value to your customers
-

STORIES ARE SMART— BUT FEATURES SHOULD BE RELEVANT

- Specific & Measurable: can I test it?
- Achievable? / Timeboxed?
- Relevant? use the "5 whys"
- *Show patron's Facebook friends*

As a box office manager

So that I can induce a patron to buy a ticket

I want to show her which Facebook friends are going to a given show

LO-FI UI SKETCHES AND STORYBOARDS

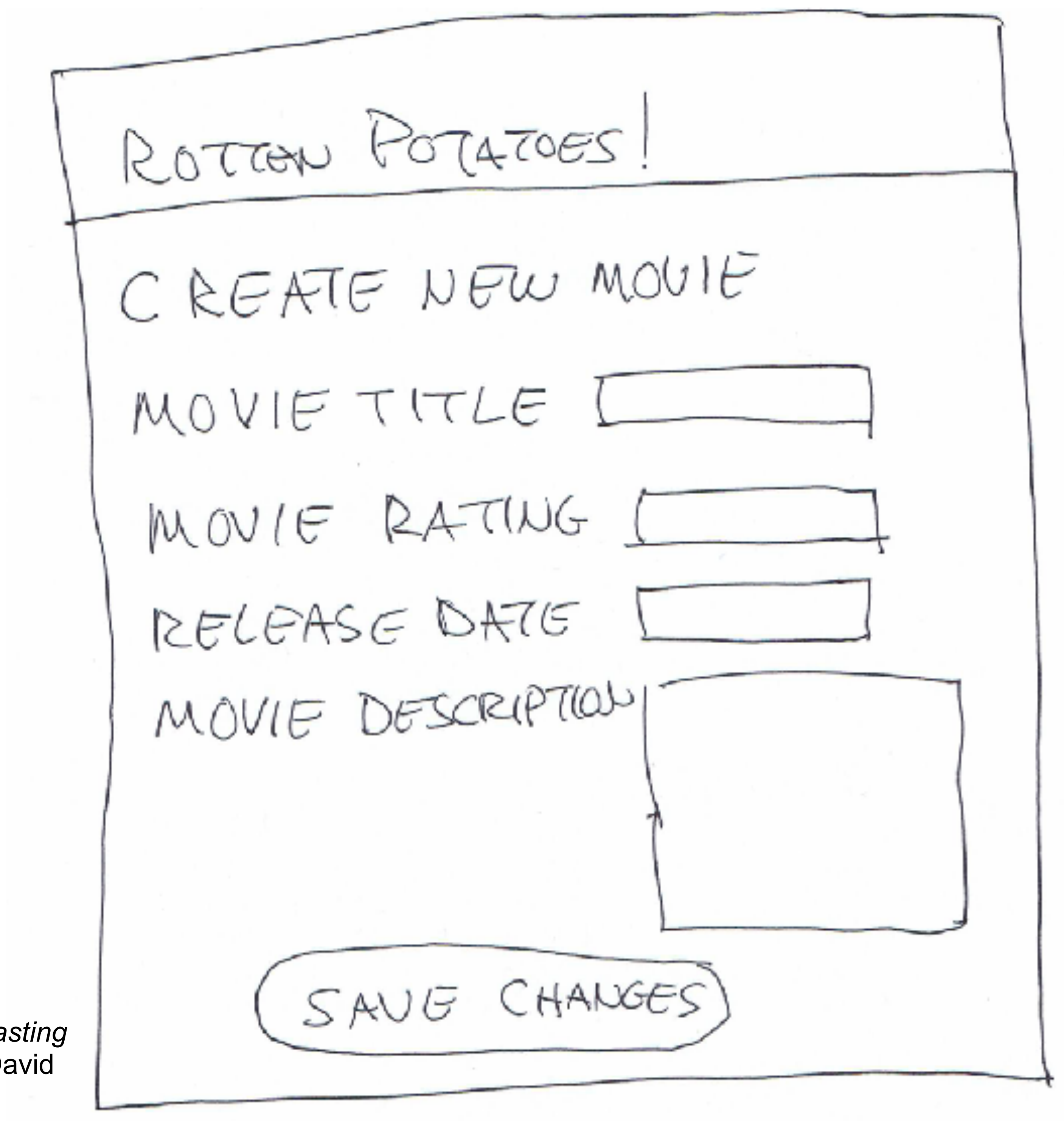
SAAS USER INTERFACE DESIGN

- SaaS apps often faces users

⇒ User stories need User Interface (UI)

- Want *all* stakeholders involved in UI design
  - Don't want UI rejected!
- Need UI equivalent of 3x5 cards
- Sketches: pen and paper drawings or "Lo-Fi UI"

LO-FI UI EXAMPLE

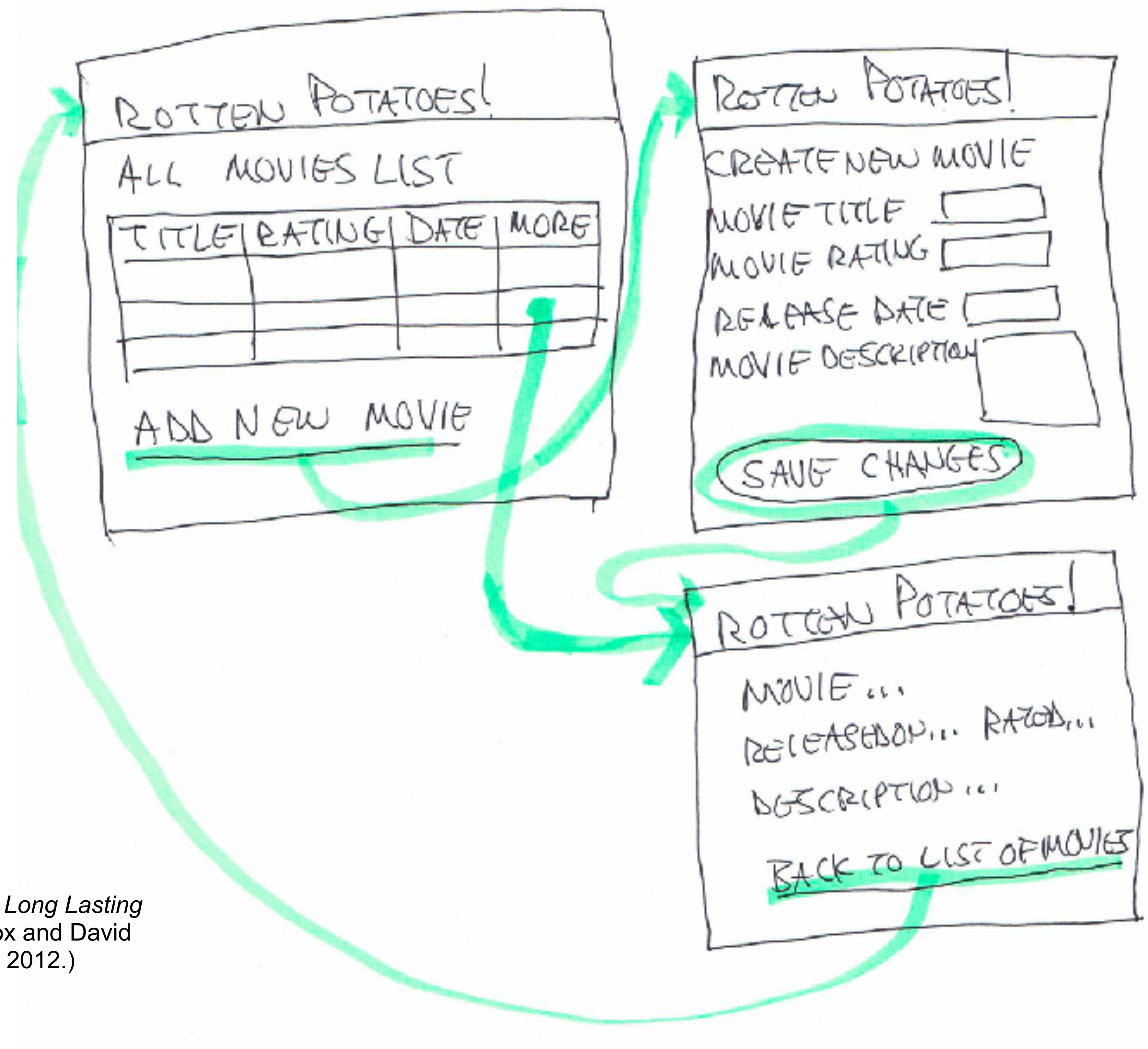


(Figure 4.3, *Engineering Long Lasting Software* by Armando Fox and David Patterson, Alpha edition, 2012.)

STORYBOARDS

- Need to show how UI changes based on user actions
- HCI => "storyboards"
- Like scenes in a movie
- But not linear

EXAMPLE STORYBOARD



(Figure 4.4, *Engineering Long Lasting Software* by Armando Fox and David Patterson, Alpha edition, 2012.)

LO-FI TO HTML

- Tedious to do sketches and storyboards,
  but easier than producing HTML!
  - Also less intimidating to nontechnical stakeholders =>
    More likely to suggest
    changes to UI if not code behind it
  - More likely to be happy with ultimate UI
- Next steps: More on CSS (Cascading Style Sheets) and Haml
  - Make it pretty *after* it works

40 Years of Version Control

SCCS & RCS (1970s)

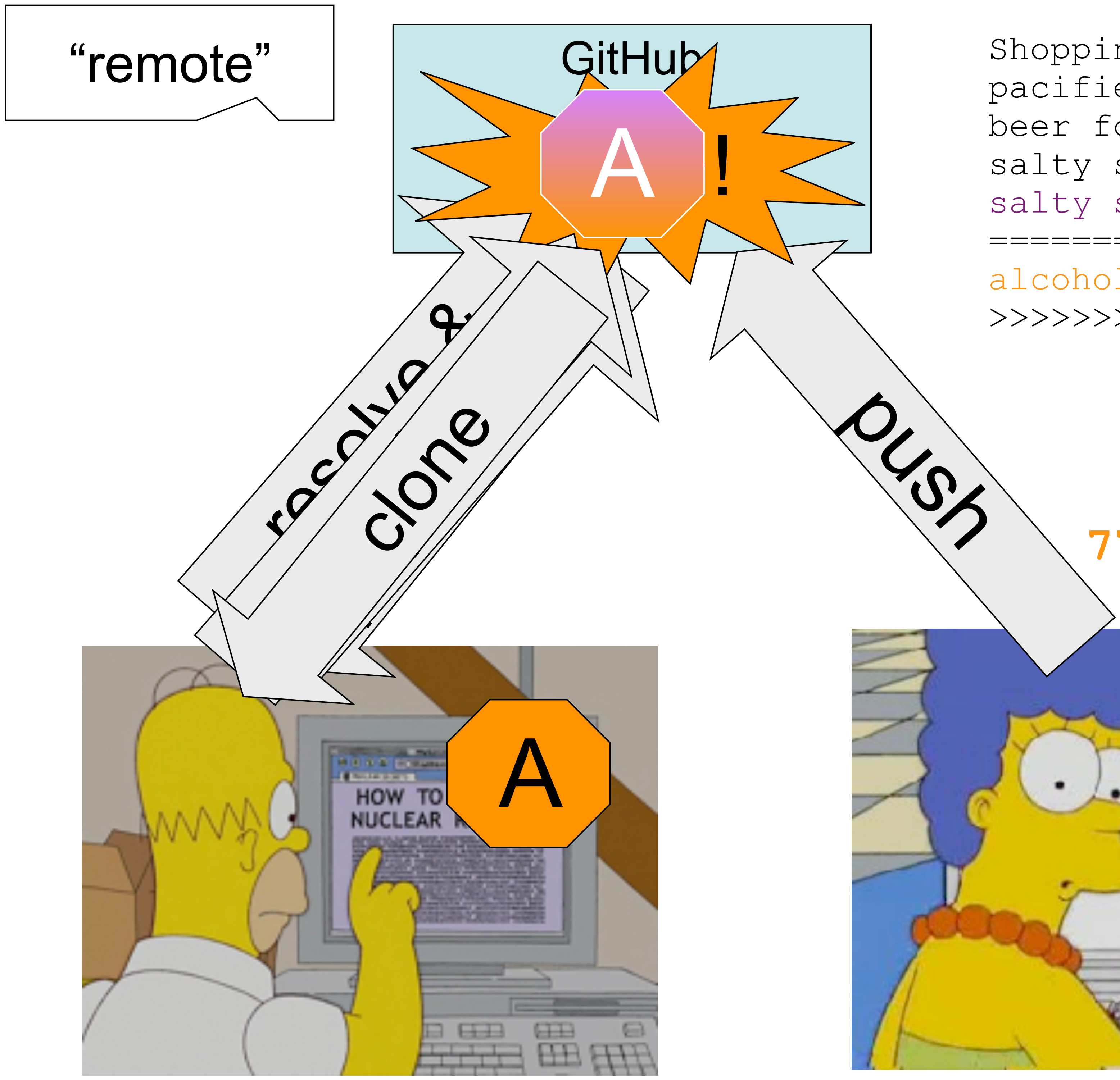
CVS (1986)

Subversion (2001)

Git (2005)
Image © TheSun.au

*

Merge Conflict



```
Shopping list:
pacifiers for Maggie
beer for Homer
salty snacks...mmm, snacks
salty snacks...mmm, snacks
=========
alcohol-free beer for Homer
>>>>>>>> 770dfb346ace24:shop_list.txt
```

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…*

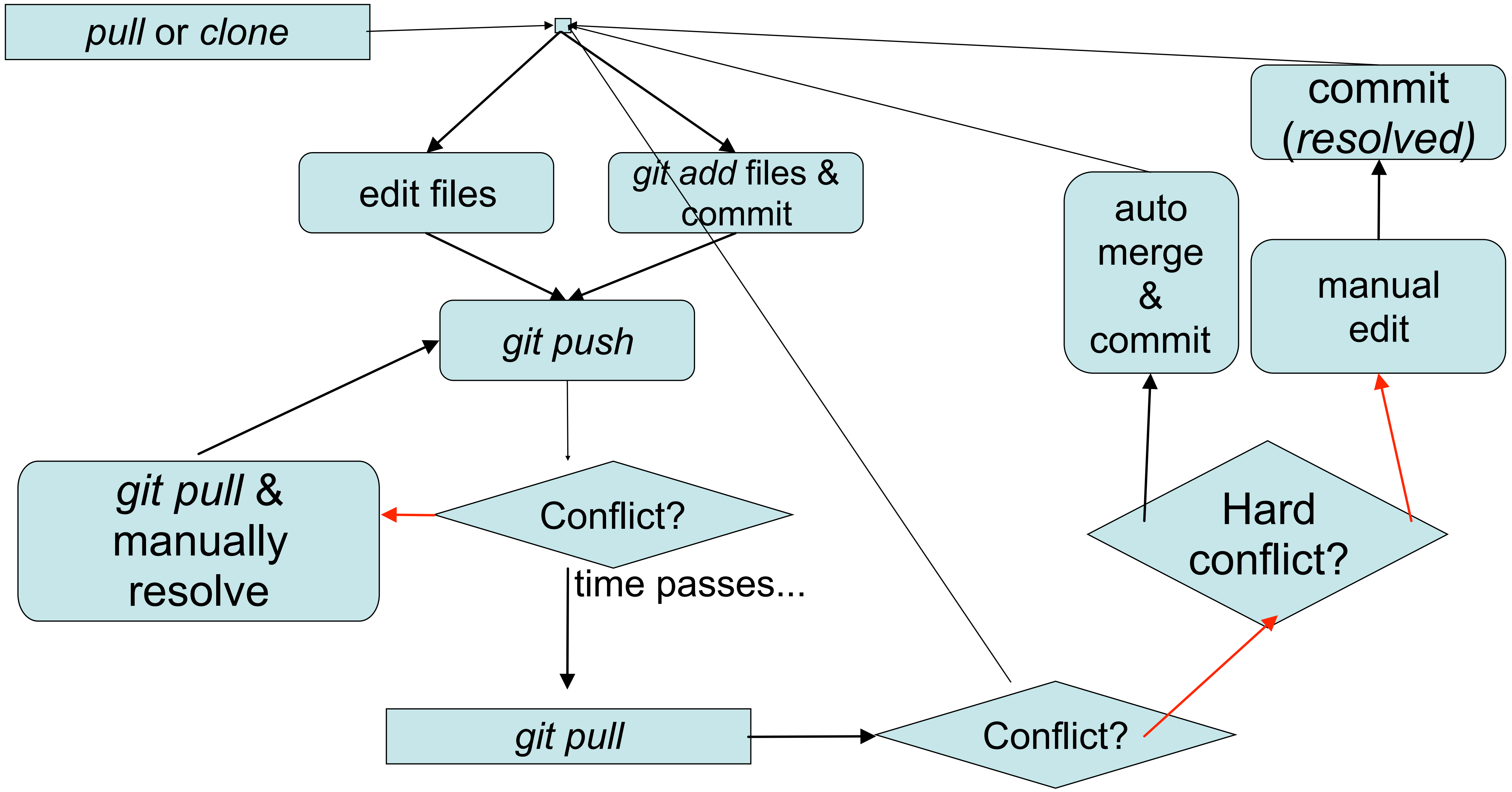
git diff "master@{01-Sep-12}" -- *files*

git blame *files*

git log *files*

*

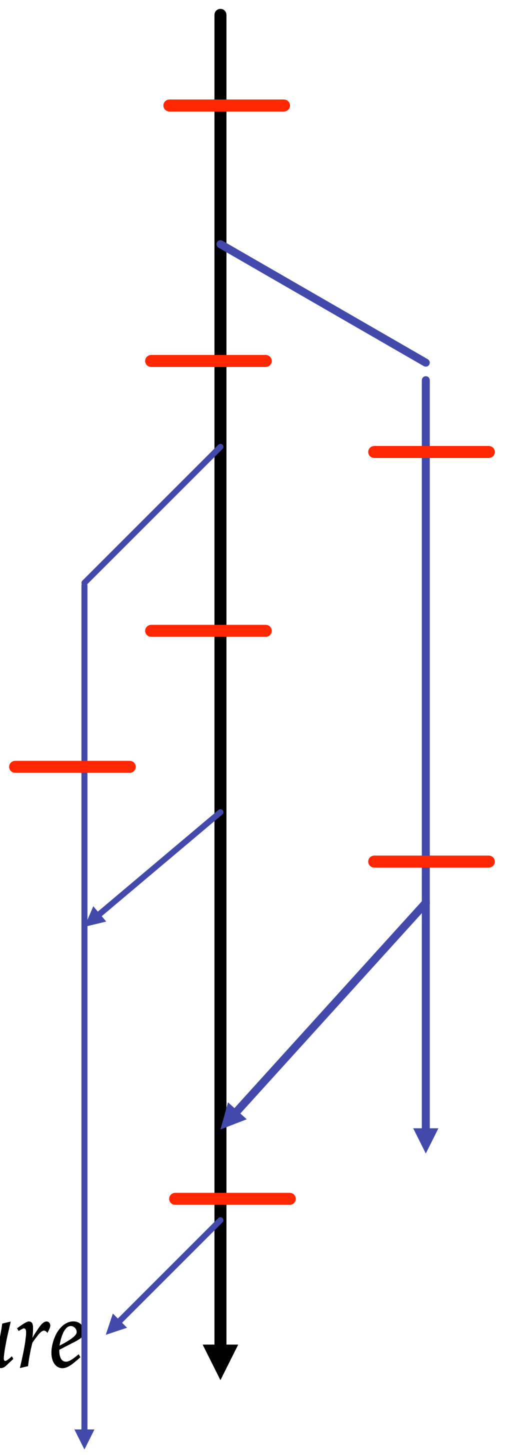
Version control with conflicts



Effective Branching

Branches

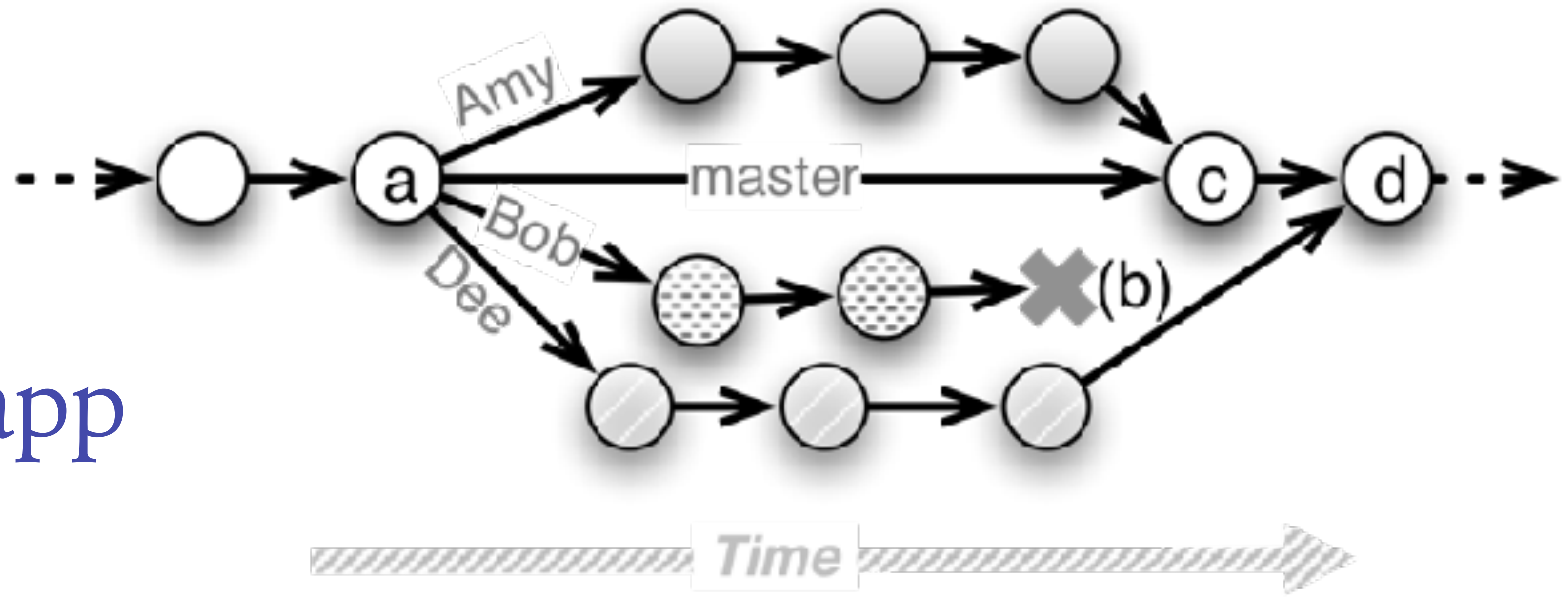
- Development **trunk** vs. **branches**
  - trunk is called “master branch” in Git
  - Creating branch is *cheap!*
  - switch among branches: *checkout*
- Separate commit histories per *branch*
- *Merge* branch back into trunk
  - ...or with *pushing* branch changes
  - Most branches eventually die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

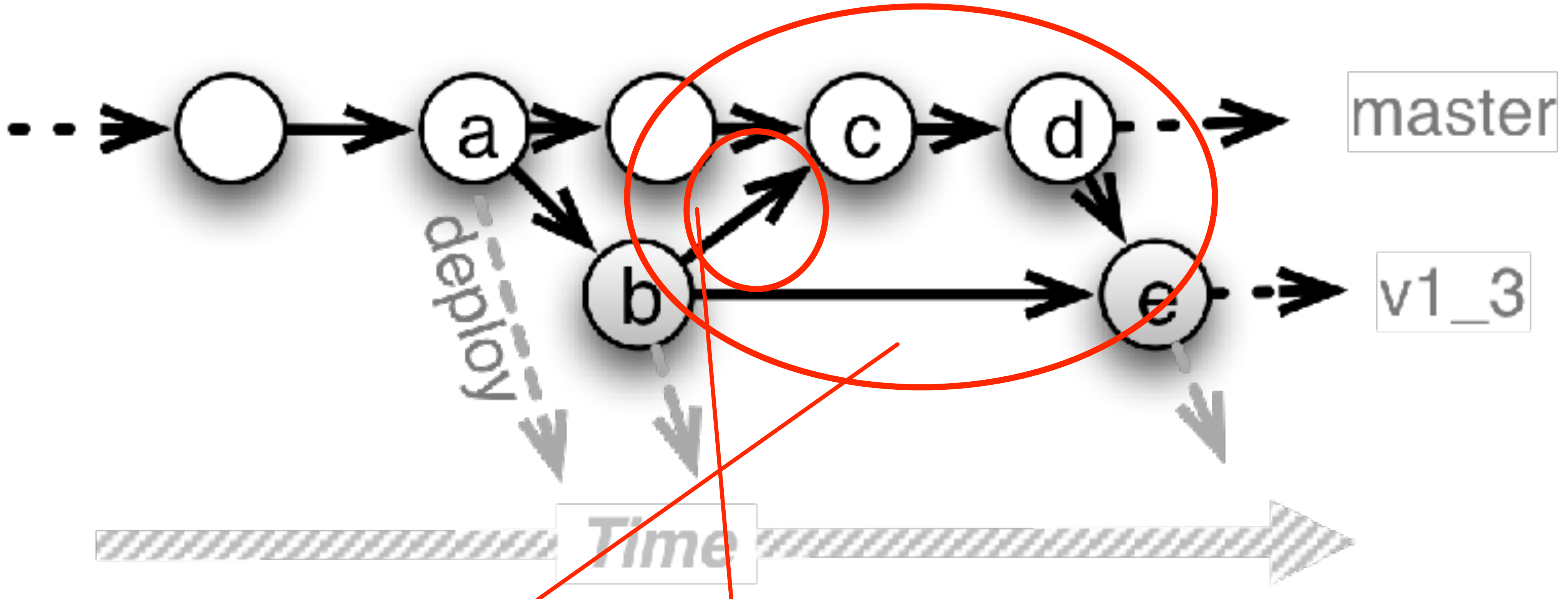
`git merge CoolNewFeature` ← *warning!!*

*

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



criss-cross merge

git cherry-pick *commit-id*

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