CS 169 Software Engineering
SaaS Architecture

Outline

- § 9.5 Identifying What's Wrong: Smells, Metrics, SOFA
- § 7.1 Intro to BDD & User Stories

Administrivia

- § 7.3 Points, Velocity, and Pivotal Tracker
- § 7.2 SMART User Stories
- § 7.7 Lo-Fi User Interface Sketches and Storyboards
- § 7.4 Agile Cost Estimation
- § 7.10 Plan-and-Document Perspective

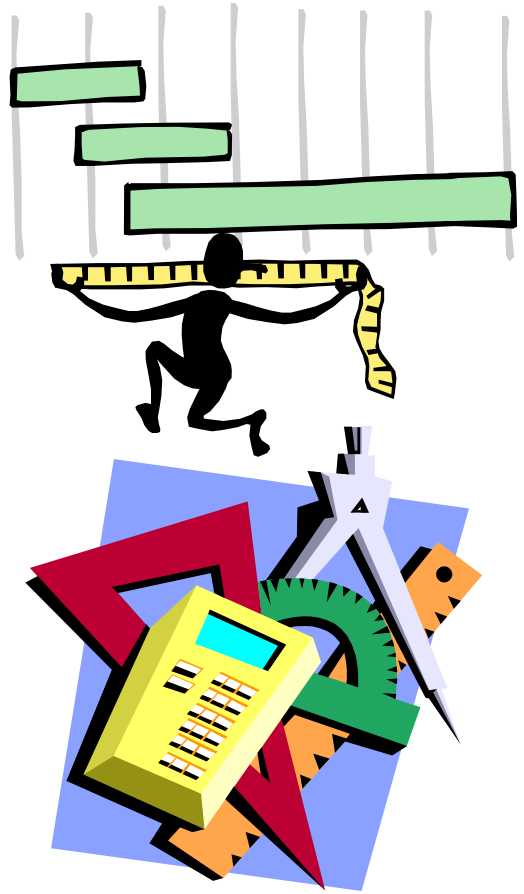
Identifying What's Wrong: Smells, Metrics, SOFA

(ESaaS §9.5)

http://pastebin.com/gtQ7QcHu

Beyond Correctness

- Can we give feedback on software *beauty*?
  - Guidelines on what is beautiful?
  - Qualitative evaluations?
  - Quantitative evaluations?
  - If so, how well do they work?
  - And does Rails have tools to support them?



Qualitative: Code Smells

SOFA captures symptoms that often indicate code smells:

- Is it Short?
- Does it do One thing?
- Does it have Few arguments?
- Is it a consistent level of Abstraction?
- Rails tool `reek` finds code smells

Single Level of Abstraction

- Complex tasks need divide & conquer
- Yellow flag for “encapsulate this task in a method”
- Like a good news story, classes & methods should read “top down”!
  - Good: start with a high level summary of key points, then go into each point in detail
  - Good: Each paragraph deals with 1 topic
  - Bad: ramble on, jumping between “levels of abstraction” rather than progressively refining

Why Lots of Arguments is Bad

- Hard to get good testing coverage
- Hard to mock/stub while testing
- Boolean arguments should be a yellow flag
  - If function behaves differently based on Boolean argument value, maybe should be 2 functions
- If arguments "travel in a pack", maybe you need to *extract a new class*
  - Same set of arguments for a lot of methods

Program X & Smells

```
class TimeSetter
  def self.convert(d)
    y = 1980
    while (d > 365) do
      if (y % 400 == 0 ||
          (y % 4 == 0 &&
       y % 100 != 0))
        if (d > 366)
          d -= 366
          y += 1
        end
      else
        d -= 365
        y += 1
      end
    end
    return y
  end
end
```

time_setterTimeSetter#self.convert calls (y + 1) twice (Duplication)

.rb -- 5 warnings:

1. TimeSetter#self.convert calls (y + 1) twice (Duplication)
2. TimeSetter#self.convert has approx 6 statements (LongMethod)
3. TimeSetter#self.convert has the parameter name 'd' (UncommunicativeName)
4. TimeSetter#self.convert has the variable name 'd' (UncommunicativeName)
5. TimeSetter#self.convert has the variable name 'y' (UncommunicativeName)

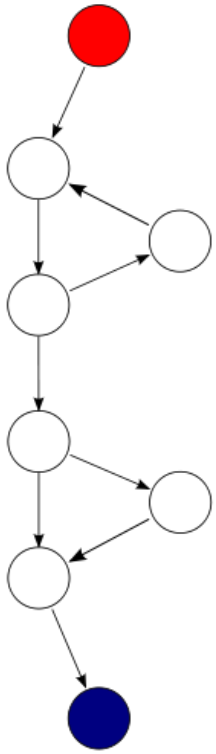
Quantitative: ABC Complexity

- Counts Assignments, Branches, Conditions
- Score = Square Root($A^2 + B^2 + C^2$)
- NIST (Natl. Inst. Stds. & Tech.): ≤20 /method
- Rails tool `flog` checks ABC complexity

Quantitative: Cyclomatic complexity

- # of linearly-independent paths thru code = E– N+2P (edges, nodes, connected components)

```
def mymeth
  while(...)
    ....
  end
  if (...)
    do_something
  end
end
```



Rails tool `saikuro` calculates cyclomatic complexity

- Here, E=9, N=8, P=1, so CC=3
- NIST (Natl. Inst. Stds. & Tech.): ≤10 /module

Quantitative: Metrics

Metric	Tool	Target score
Code-to-test ratio	rake stats	≤ 1:2
C0 (statement) coverage	SimpleCov	90%+
Assignment-Branch-Condition score	flog	< 20 per method
Cyclomatic complexity	saikuro	< 10 per method (NIST)

- "Hotspots": places where *multiple metrics* raise red flags
  - add `require 'metric_fu'` to **`Rakefile`**
  - **`rake metrics:all`**
- Take metrics with a grain of salt
  - Like coverage, better for *identifying where improvement is needed* than for *signing off*

Leap Year & Quantitative

```
class TimeSetter
  def self.convert(d)
    y = 1980
    while (d > 365) do
      if (y % 400 == 0 ||
          (y % 4 == 0 && y % 100 != 0))
        if (d > 366)
          d -= 366
          y += 1
        end
      else
        d -= 365
        y += 1
      end
    end
    return y
  end
end
```

- ABC score of 23
  (>20 so a problem))
- Gets code complexity score of 4
  (≤ 10 so not a problem)

Revised Leap Year & Metrics

```
class TimeSetter
  def self.convert(day)
    year = 1980
    while (day > 365) do
      if leap_year?(year)
        if (day >= 366)
          day -= 366
        end
      else
        day -= 365
      end
      year += 1
    end
    return year
  end
```

```
private
  def self.leap_year?(year)
    year % 400 == 0 ||
      (year % 4 == 0 && year % 100 != 0)
  end
end
```

Reek: No Warnings

Flog (ABC):

TimeSetter.convert = 11

TimeSetter.leap_year? = 9

Saikuro (Code Complexity) = 5

Which SOFA guideline is most important for unit-level testing?

1. Short
2. Do One thing
3. Have Few arguments
4. Stick to one level of Abstraction

Comments Should Describe Things That Aren't Obvious From The Code: Why, not What

(§9.4 *ESaaS*)

(from John Ousterhout)

Bad Comments

```
// Add one to i.
i++;
```

```
// Lock to protect against concurrent access.
SpinLock mutex;
```

```
// This function swaps the panels.
void swap_panels(Panel* p1, Panel* p2) {...}
```

Comments, cont'd

Comments should be at a higher abstract level than code:

```
# Scan the array to see if the symbol exists
```

not

```
# Loop through every array index, get the
# third value of the list in the content to
# determine if it has the symbol we are looking
# for. Set the result to the symbol if we
# find it.
```

Introduction to Behavior-Driven Design and User Stories

(Engineering Software as a Service §7.1)

David Patterson

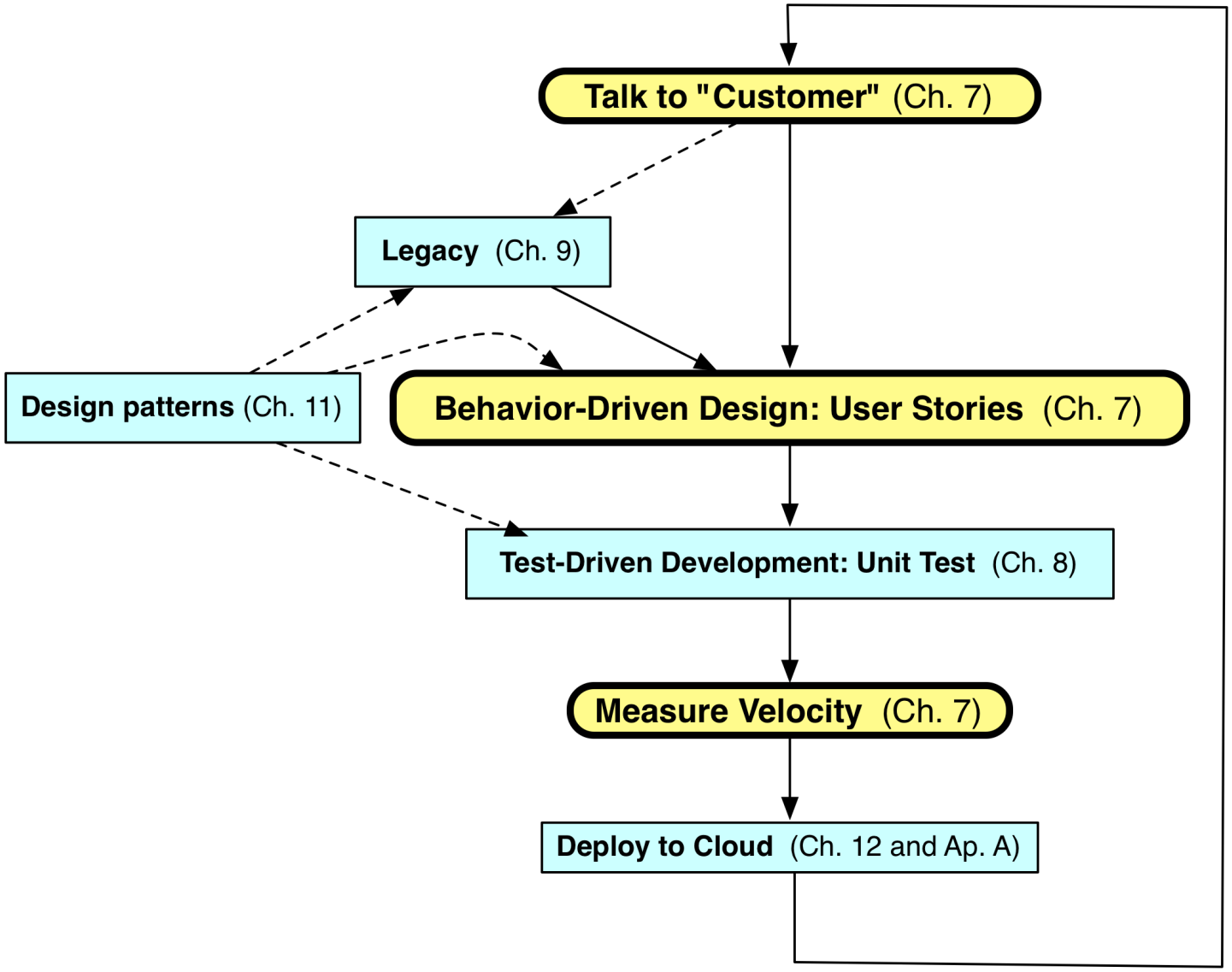
Why do SW Projects Fail?

- Don't do what customers want
- Or projects are late
- Or over budget
- Or hard to maintain and evolve
- Or all of the above
- How does Agile try to avoid failure?

Agile Lifecycle Review

- 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

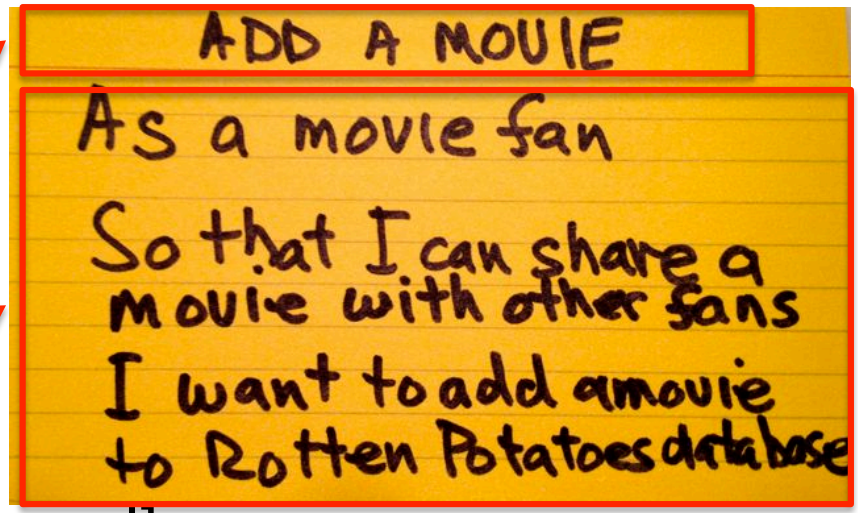


Behavior-Driven Design (BDD)

- BDD asks questions about behavior of app *before and during development* to reduce miscommunication
  - Validation vs. Verification
- 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 (future segments) 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



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 Pivotal Tracker)
- Prioritize so most valuable items highest
- Organize so they match SW releases over time

Related Issue

- Spike
  - Short investigation into technique or problem
    - E.g. spike on recommendation algorithms
  - After spike done, code *must* be thrown away
    - Now that know approach you want, write it right

Which expression statement regarding BDD and user stories is FALSE?

1. BDD is designed to help with validation (build the right thing) in addition to verification
2. BDD should test app implementation
3. User stories in BDD play same role as design requirements in Plan-and-Document
4. This is a valid User Story: "Search TMDb
   I want to search TMDb
   As a movie fan
   So that I can more easily find info"

Administrivia - Projects

- We think all but ~2 out of ~240 students in teams
- Every team has a project in spreadsheet
  - Please send the private Piazza post – follow the directions
- Arrange first meetings with customer by Sunday
  - Send to your TA your customer meeting time
  - Can start with User stories, iterations, Lo-Fi UI (next), Pivotal Tracker (next)
- Sorry about prioritized signup announcement
  - Didn't know other classes had matching algorithms
  - If unhappy with project, talk to Omer

Points, Velocity, and Pivotal Tracker

(Engineering Software as a Service §7.3)

David Patterson

Productivity and Tools

- Don’t we want to avoid major planning effort in Agile? If so, how estimate time without a plan?
- Can User Stories be used to measure progress on project?
- What should a tool do to help measured progress for Agile?

Measuring Productivity

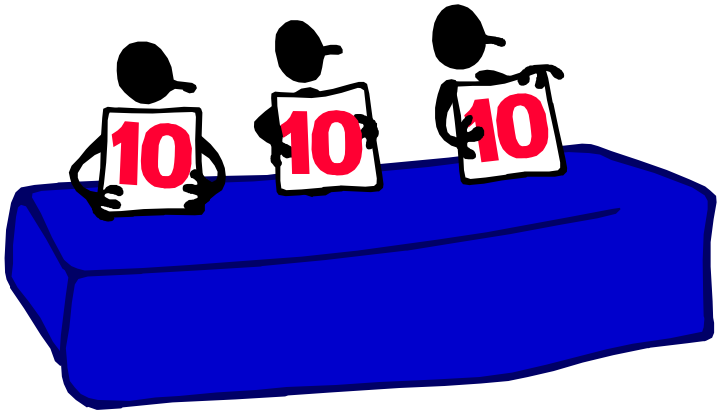
- A measure of team productivity: calculate avg no. stories / week?
  - But some stories much harder than others
- Rate each user story in advance on a simple integer scale
  - 1 for straightforward stories, 2 for medium stories, 3 for very complex stories
- Velocity: avg number of *points* / week

More on Points

- Once get experience, Fibonnaci scale is commonly used: 1, 2, 3, 5, 8
  - (Each new number is sum of previous 2)
  - At Pivotal Labs, 8 is extremely rare
- Teams assign value: vote by holding up fingers simultaneously, take average
  - If a big disagreement (2 and 5), discuss more

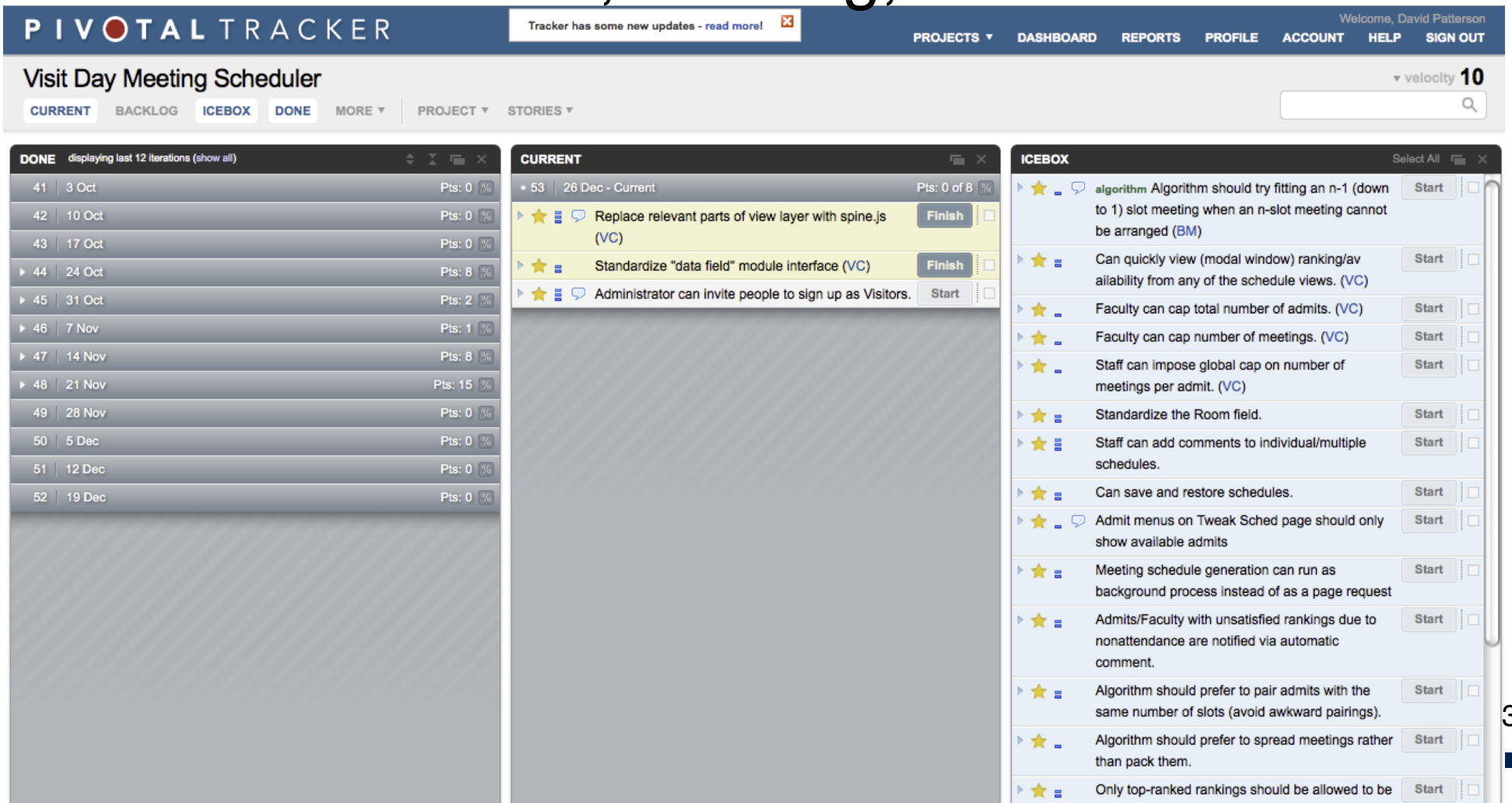
More on Points

- ≥5 => divide user story into simpler stories
  - backlog not too demanding
- Doesn't matter if velocity is 5 or 10 points per iteration
  - As long as team consistent
- Idea is to improve self-evaluation and suggest number of iterations for feature set



Pivotal Tracker

- Calculates velocity for team, manages user stories: Current, Backlog, Icebox



Pivotal Tracker

- Prioritize user stories by where place them in Current, Backlog, Icebox panels
- When completed, move to Done panel
- Can add logical Release points, so can figure out when a Release will really happen
  - Remaining points/Velocity
- *Epic* (with own panel)
  - Combine related user stories together
  - Ordered independent of user story in Backlog

Tracker Roles

- Developers don't decide when user stories completed
  - Pushes Finish button, which sends to "Product Owner" (as in Scrum team organization)
- Product Owner tries out the user story and then either hits
  - Accept, which marks user story as done, or
  - Reject, which marks story as needing to be Restarted by developer

Pivotal Tracker: Features vs. Chores

- Features
  - User stories that provide verifiable business value to customer
    - "Add agree box to checkout page"
  - Worth points & therefore must be estimated
- Chores
  - User Stories that are necessary, but provide no direct, obvious value to customer
    - "Find out why test suite is so slow"
  - No points

Team Cyberspace Whiteboard

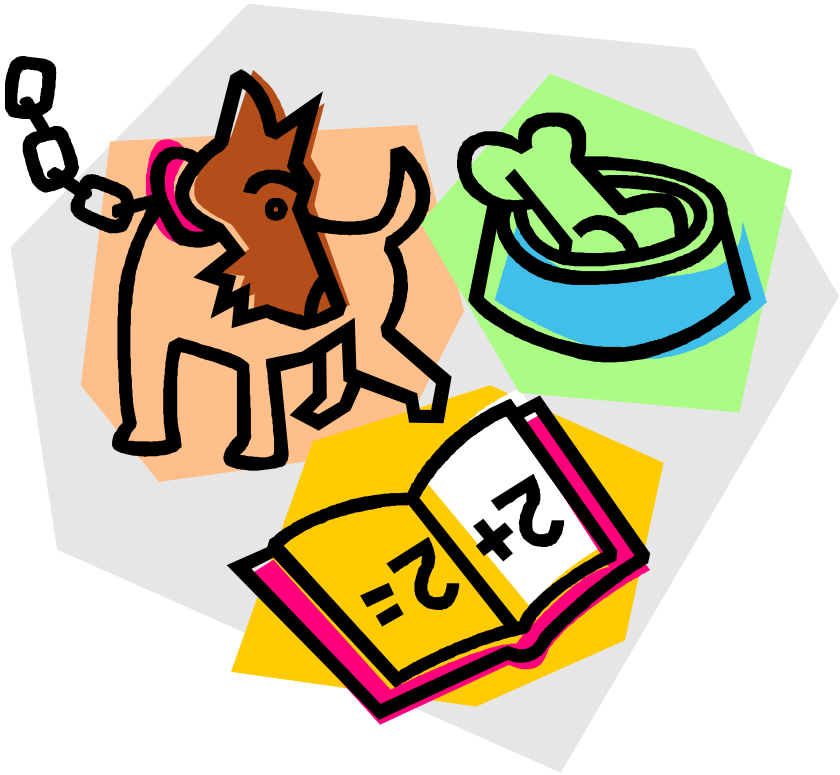
- Tracker allows attaching documents to User stories (e.g., LoFi UI)
- Wiki with Github repository
- Google Documents: joint creation and viewing of drawings, presentations, spreadsheets, and text documents
- Campfire: web-based service for password-protected online chat rooms

Which expression statement regarding Points, Velocity, and Tracker is TRUE?

1. When comparing two teams, the one with the higher velocity is more productive
2. When you don't know how to approach a given user story, just give it 3 points
3. With Tracker, developers pick the user stories and mark as Accepted when done
4. Tracker helps prioritize and keep track of user stories and their status, calculates velocity, and predicts software development time

SMART User Stories

(Engineering Software as a Service §7.2)



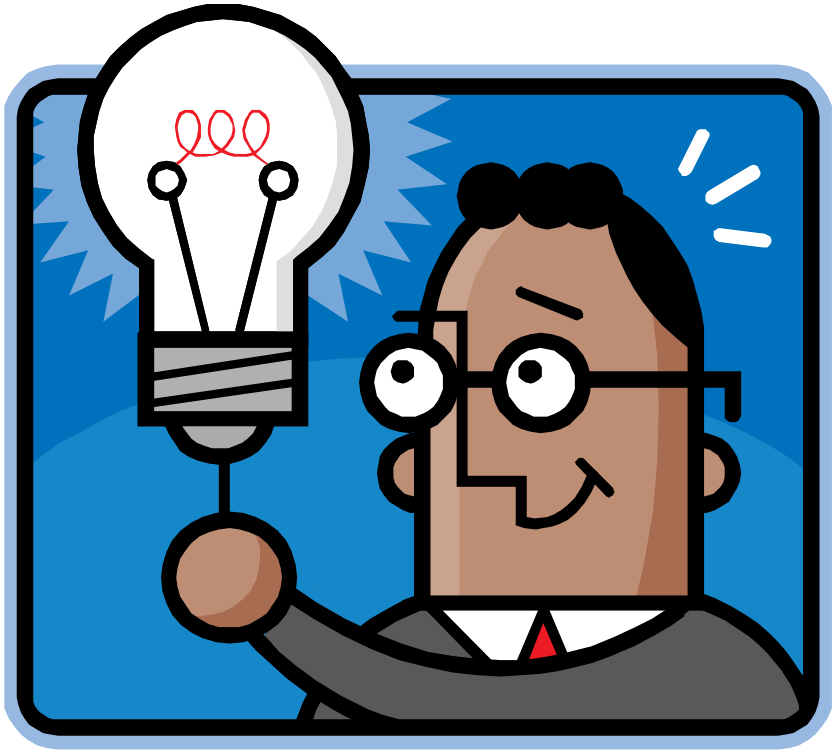
David Patterson

Creating User Stories

- How do you know if you have a good user story vs. bad user story?
  - Right size?
  - Not too hard?
  - Is worthwhile?

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.

1. *Given* some specific starting condition(s),
2. *When* I do X,
3. *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
- If <1 story per iteration, need to improve point estimation per story

Relevant: “business value”

- 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

http://wiki.github.com/aslakhellesoy/cucumber has a good example

5 Whys to Find Relevance

- *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

  1. *Why?*
  2. *Why?*
  3. *Why?*
  4. *Why?*
  5. *Why?*

Timeboxed

- Stop story when exceed time budget
  - Give up or divide into smaller stories or reschedule what is left undone
- To avoid underestimating length of project
- Pivotal Tracker tracks velocity, which helps avoid underestimate

Which feature below is LEAST SMART?

1. User can search for a movie by title

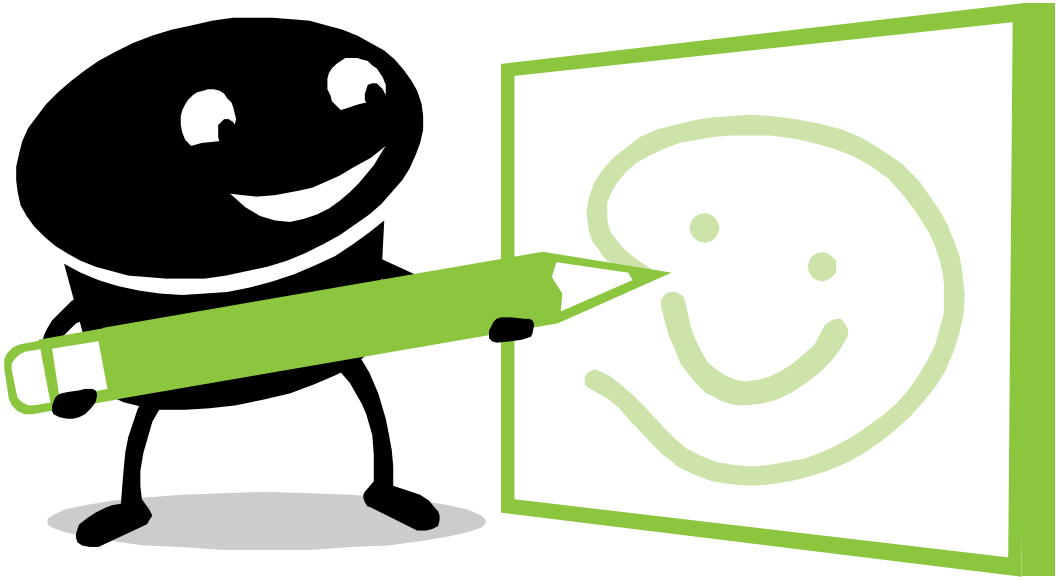
2. Rotten Potatoes should have good response time

3. When adding a movie, 99% of Add Movie pages should appear within 3 seconds

4. As a customer, I want to see the top 10 movies sold, listed by price, so that I can buy the cheapest ones first

Lo-Fi UI Sketches and Storyboards

(Engineering Software as a Service §7.7)



David Patterson

Building Successful UI

- SaaS apps often faces users

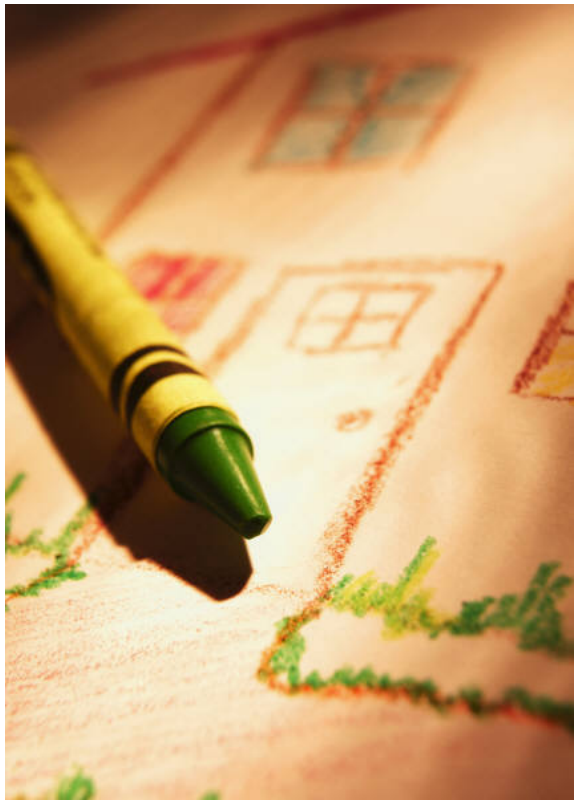
⇒User stories need User Interface (UI)

- How get customer to participate in UI design so is happy when complete?
  - Avoid WISBNWIW* UI?
  - UI version of 3x5 cards?
- How show interactivity without building prototype?

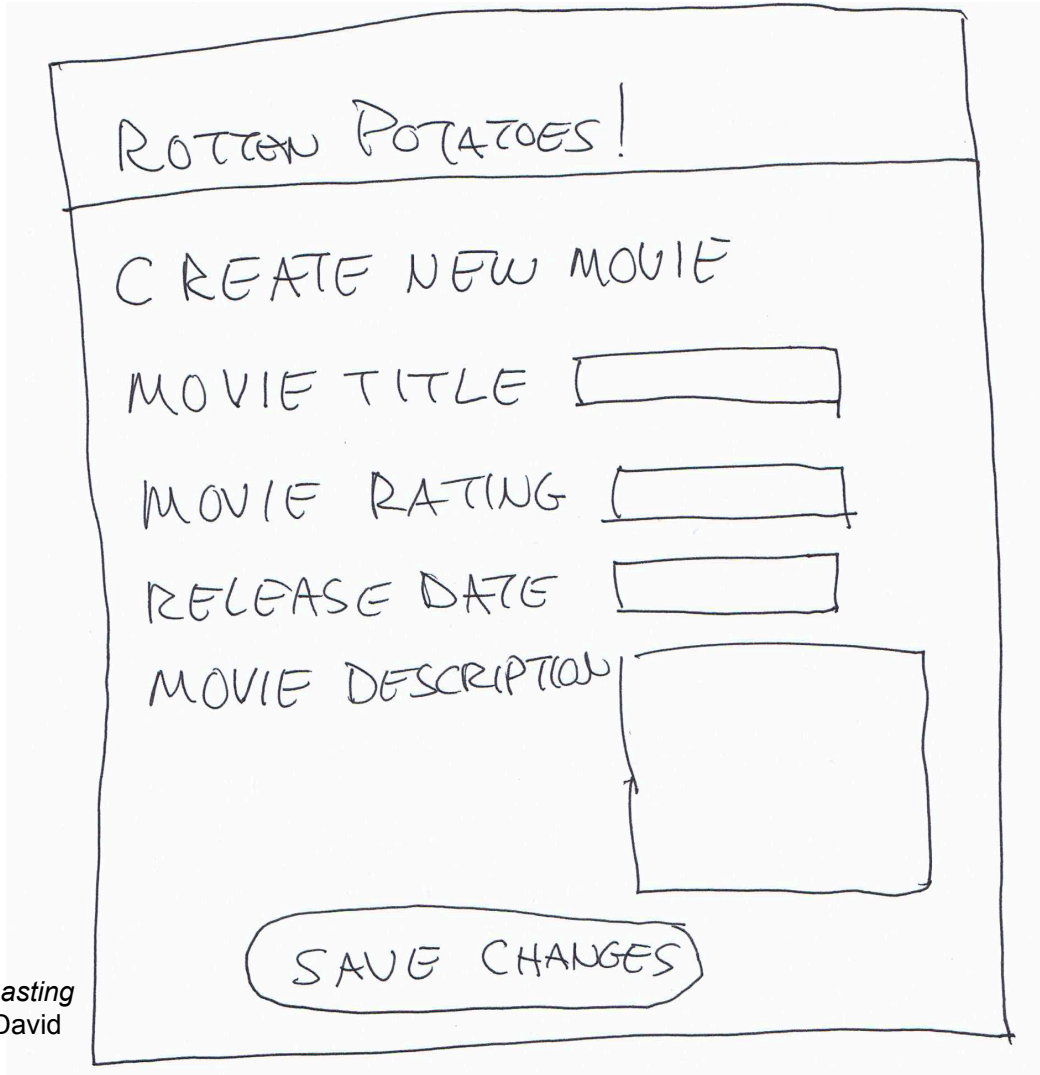
* What-I-Said-But-Not-What-I-Want

SaaS User Interface Design

- UI Sketches: pen and paper drawings or "Lo-Fi UI"



Lo-Fi UI Example

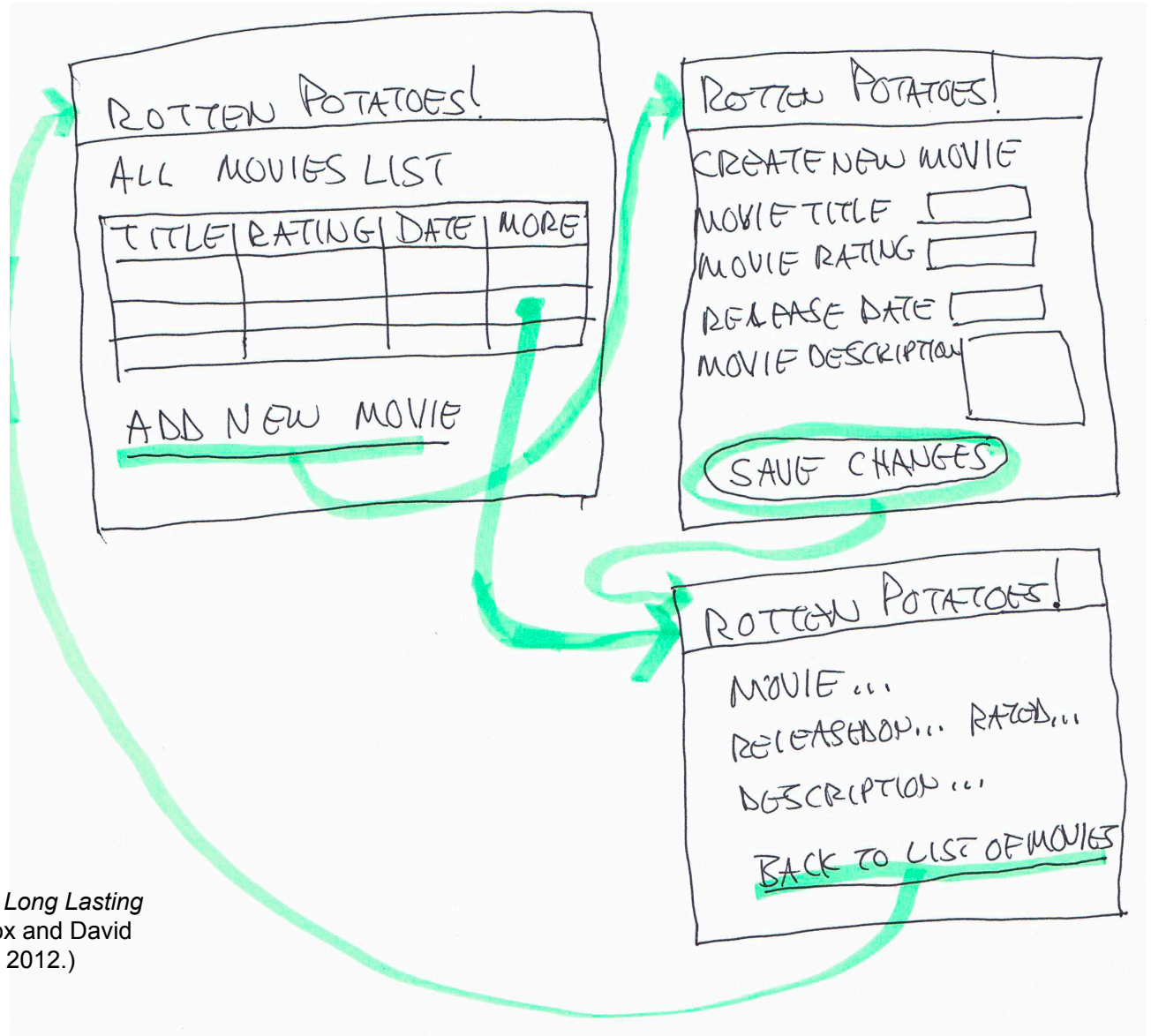


(Figure 4.3, *Engineering Long Lasting Software* by Armando Fox and David Patterson, Alpha edition, 2012.)

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: CSS (Cascading Style Sheets) and Haml (later)
  - Make it pretty *after* it works

Which is FALSE about Lo-Fi UI?

- ☐ Like 3x5 cards, sketches and storyboards are more likely to involve all stakeholders vs. code
- ☐ The purpose of the Lo-Fi UI approach is to debug the UI before you program it
- ☐ SaaS apps usually have a user interfaces associated with the user stories
- ☐ While it takes more time than building a prototype UI in CSS and Haml, the Lo-Fi approach is more likely to lead to a UI that customers like

Agile Cost Estimation

(Engineering Software as a Service §7.4)

David Patterson

Agile Cost Estimation

- Real world needs to estimate cost before customer can agree to project
- If no careful planning and schedule in Agile, how can you estimate the cost of a project?

Pivotal Labs Model

- Pivotal Labs teaches customer Agile
- Using Agile, Pivotal never commits to delivering features X, Y, and Z by date D
- Instead, commits resources to work in the most efficient way possible up to date D
  - Customer works with team to define priorities continuously up to date D
- Still need estimate for project

Pivotal Labs Agile Estimate

1. 1 hour phone call to explain method
   - Joint effort, customer time commitment, …
2. Customer visits for 1.5 hour "scoping"
   - Customer brings designer, developer, designs
     - Anything to clarify what want done
   - Pivotal brings 2 engineers who ask questions
     - Trying to identify what adds uncertainty to estimate
3. Engineers take ½ hour to estimate weeks
   - Little vs. large uncertainty: 20-22 vs. 18-26 wks
4. Bid cost as time and materials to customer

Which expression statement regarding cost estimation is TRUE?
(PL = Pivotal Labs)

1. As practitioners of Agile Development, PL does not use contracts
2. As practitioners of pair programming, PL estimates cost for 1 pair, which it assigns to complete project
3. The cost bid is for PL time and materials that covers number of weeks in the estimate
4. As studies show 84%-90% projects are on-time and on-budget, plan-and-document managers promise customers a set of features for an agreed upon cost by an agreed upon date

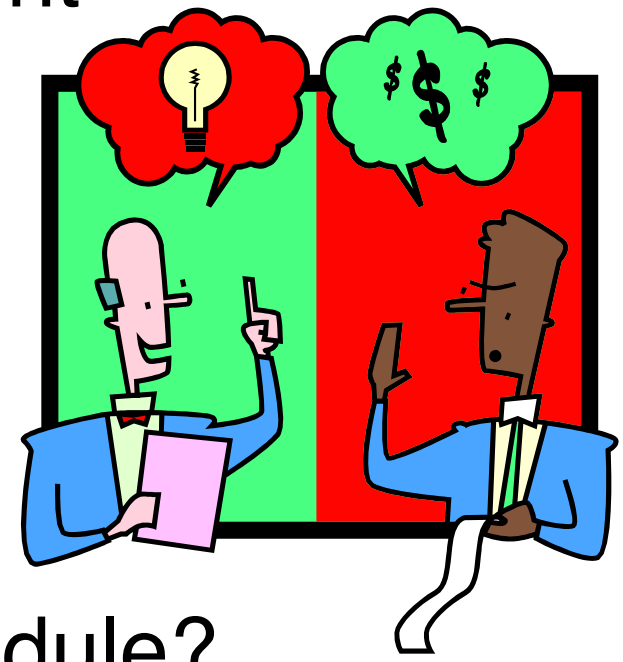
Plan-And-Document Perspective

(Engineering Software as a Service §7.10)

David Patterson

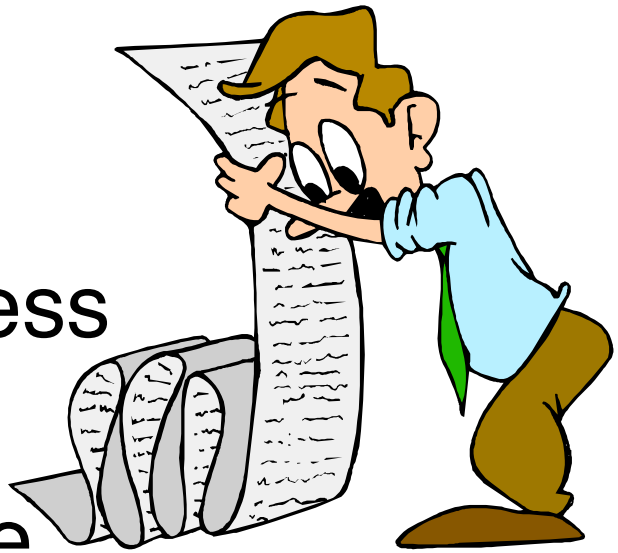
Introduction

- What does Plan-and-Document do instead of
  - User stories?
  - Points?
  - Velocity?
- How does a project manager estimate costs? Make a Schedule?



P&D Equivalents+

1. Requirements Elicitation
2. Requirements Documentation
3. Cost Estimation
4. Scheduling & Monitoring Progress
5. Change Management for Requirements, Cost, &Schedule
6. Ensuring Implementation Matches Requirement Features
7. Risk Analysis & Management



P&D Requirements Elicitation

- Elicit *functional*&*non-functional* requirements:

1. *Interviewing* - see how currently *really* done
   - Stakeholders answer predefined questions
   - Or just have informal discussions
2. Cooperatively create *Scenarios*
   - Initial state, show flow for happy & sad paths, what is concurrent, final state
3. Create *Use Cases*
   - List of steps to achieve goal between user and system; has language (UML) to describe

P&D Requirements Documentation

- Document requirements via *Software Requirements Specification* (*SRS*)
  - 100s of pages; IEEE standard for SRS!
- Have stakeholders read SRS, or build basic prototype, or generate test cases to check:
  - *Validity*: are all requirements necessary?
  - *Consistency*: do requirements conflict?
  - *Completeness*: are all requirements and constraints included?
  - *Feasibility*: can requirements be implemented?

P&D Cost Estimation

- Manager decomposes SRS into tasks
- Estimates weeks per tasks
  - 1 week ≤ Tasks ≤ 8 weeks
- Covert person-weeks into $ via salaries and overhead
- Estimate *before* & *after* contract
  - Add safety margin: 1.3 to 1.5X
  - Make 3 estimates (best case, expected, worst) then make best guess

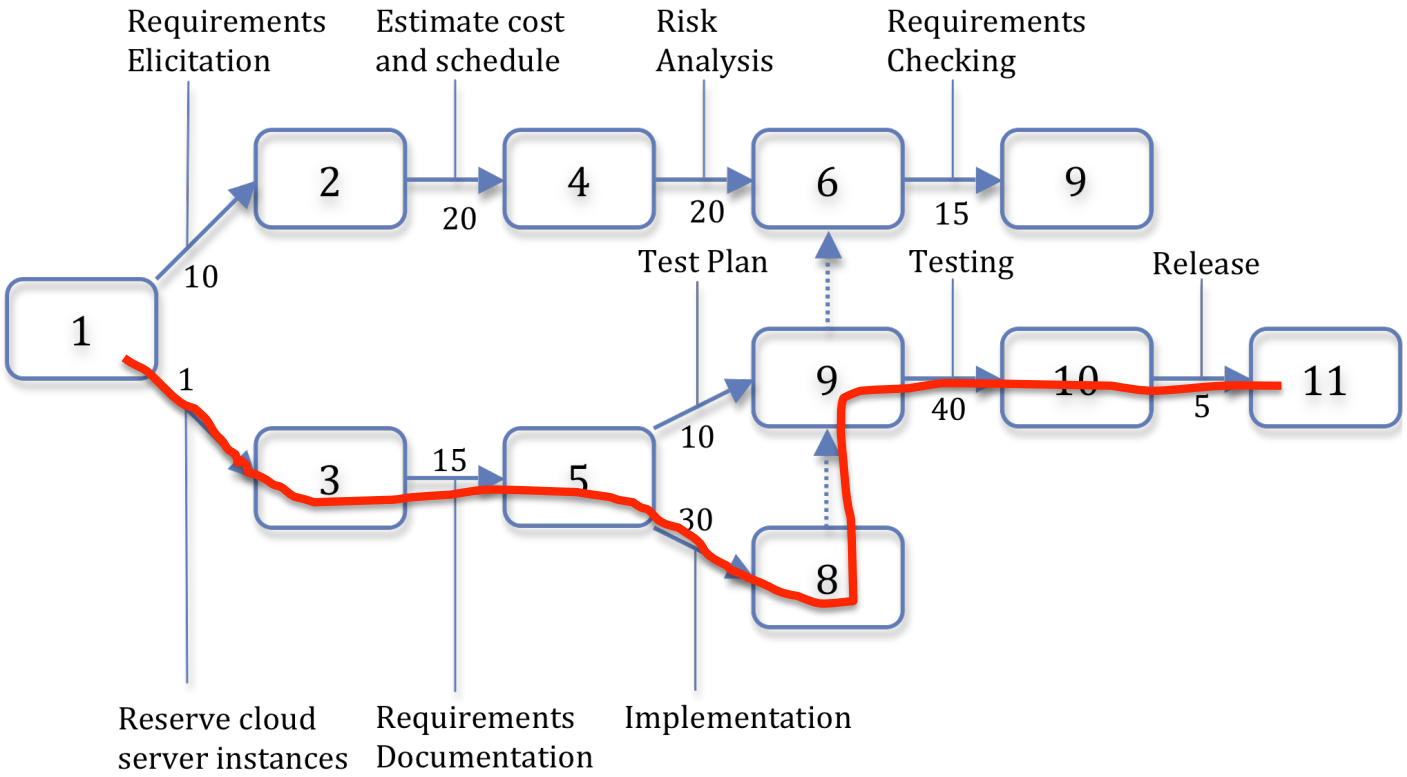
P&D Cost Estimation

1. Experiential Estimate
   - Rely on manager's experience to be accurate
2. Quantitative Estimate
   - Estimate tasks in lines of code (LOC), divide by LOC/person-month
   - COCOMO (Constructive Cost Model): Effort = OrgFactor*CodeSize^{Penalty} * ProdFactor
     - Organization Factor = 2.94, 1.10≤SizePenalty≤1.24, 0.90 ≤ Product Factor ≤1.40

- 92% use experiential vs. formula

P&D Scheduling

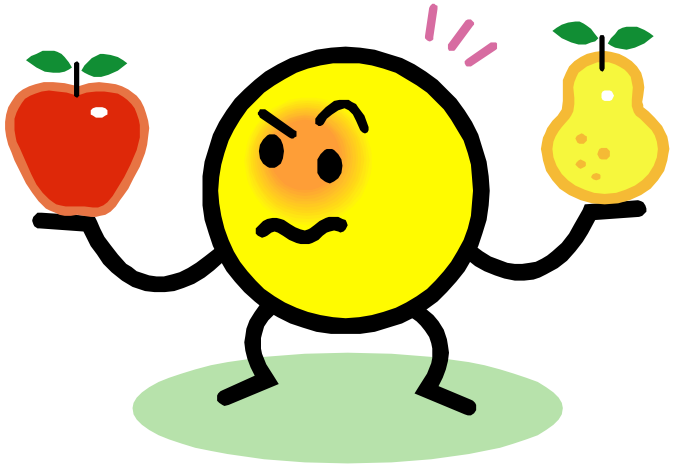
- Use PERT chart to show task parallelism and critical path to make schedule



  - Nodes are milestones, link names are tasks, link numbers are effort, arrows are dependencies

P&D Monitoring Progress

- Compare predicted to actual
  - Time for tasks
  - Expenditures
- Intermediate milestone help all stakeholders see if project is on schedule & on budget



P&D Requirements Checking

- Manager uses tools for *requirements traceability*
- Tool has cross references between
  - Portion of SRS with requirement
  - Portion of code that implements requirement
  - Tests that validate requirement

P&D Risk Analysis & Management

- To improve accuracy of budget/schedule
- Identify risks early to
  - Do extra work to reduce risk
  - Change plan to avoid risk
- Technical: RDB can't scale
- Organizational: J2EE???
- Business: too late for market
- Create Risk Table: % chance x impact
  - Address top 20%, hoping 80% of risk

P&D vs. Agile Requirements and Schedule/Cost Estimation

Tasks	*In Plan and Document*	*In Agile*
Requirements Documentation	Software Requirements Specification such as IEEE Standard 830-1998	User stories, Cucumber, Points, Velocity
Requirements Elicitation	Interviews, Scenarios, Use Cases	
Change Management for Requirements, Schedule, and Budget	Version Control for Documentation and Code	
Ensuring Requirements Features	Traceability to link features to tests, reviews, and code	
Scheduling and Monitoring	Early in project, contracted delivery date based on cost estimation, using PERT charts. Milestones to monitor progress	
Cost Estimation	Early in project, contracted cost based on manager experience or estimates of task size combined with productivity metrics	Evaluate to pick range of effort for time and materials contract
Risk Management	Early in project, identify risks to budget and schedule, and take actions to overcome or avoid them	

Figure 7.14: The relationship between the requirements related tasks of Plan-and-Document versus Agile methodologies.

larger than the original software project! Risk reduction is a major reason for iteration in both

Which expression statement regarding P&D requirements and cost estimation is FALSE?

1. The closest to the P&D schedule and monitoring tasks are Agile points and velocity
2. The closest to the P&D Software Requirements Specification (SRS) document is Agile User Stories
3. Agile has no equivalent to ensuring requirements, such as traceability
4. Actually, 1, 2, and 3 are all true; none are false

And in Conclusion: §§9.4-9.5, 7.1-7.4, 7.7

- SOFA methods: Short, do One thing, Few arguments, consistent Abstraction
  - Metrics point to problem pieces of program
- BDD: User stories to elicit requirements
  - SMART: Specific, Measureable, Achievable, Relevant, Timeboxed
- Points/Velocity to calculate progress (Tracker)
- Lo-Fi UI/storyboard: sketch to elicit UI design
- Lifecycles: Plan&Document (careful planning & documentation) v. Agile (embrace change)