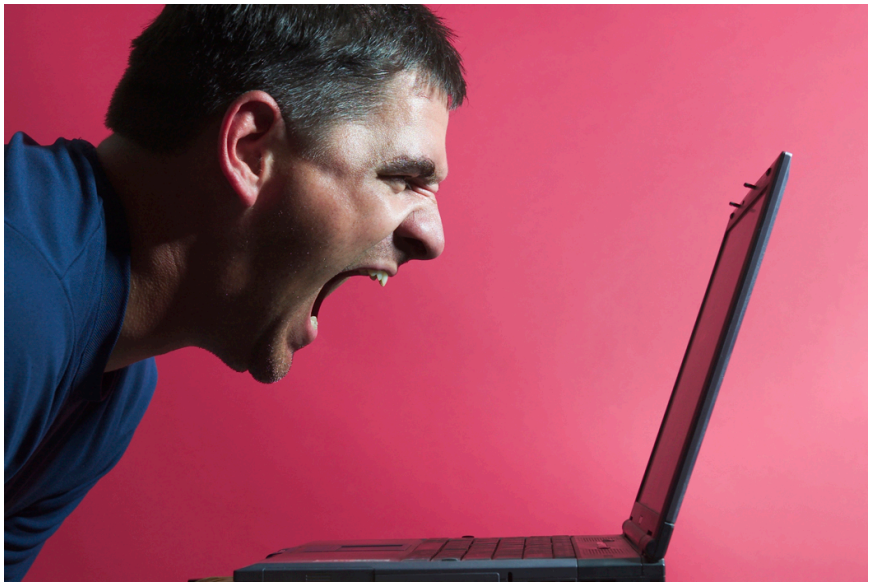
# Course Goals

- Learn Software Engineering Principles by understanding new challenges, opportunities, and open problems of SaaS
- Take a SaaS project from conception to public deployment
  - Solve Non-Technical Customer problem
  - Server side: Ruby on Rails
  - Client side: HTML, CSS, AJAX, JavaScript
  - Deploy using cloud computing

# Engineering Software is Different from Engineering Hardware



(*Engineering Long Lasting Software* §1.1-§1.2)

David Patterson

# Engineering Software is Different from Hardware

- Q: Why so many SW disasters and no HW disasters?
  - Ariane 5 rocket explosion
  - Therac-25 lethal radiation overdose
  - Mars Climate Orbiter disintegration
  - FBI Virtual Case File project abandonment
- A: Nature of the 2 media and subsequent cultures that developed

# Independent Products vs. Continuous Improvement

- Cost of field upgrade
- HW_____
  - ➔HW designs must be finished before manufactured and shipped
  - ➔Bugs: Return HW (lose if many returns)
- SW_____
  - ➔Expect SW gets better over time
  - ➔Bugs: Wait for upgrade
- HW decays, SW long lasting

# Legacy SW vs. Beautiful SW

- **Legacy code**: old SW that continues to meet customers' needs, but difficult to evolve due to design inelegance or antiquated technology
  - 60% SW maintenance costs adding new functionality to legacy SW
  - 17% for fixing bugs
- Contrasts with **beautiful code**: meets customers' needs and easy to evolve

# Legacy Code: Vital but Ignored

- Missing from traditional SWE courses and textbooks
- Number 1 request from industry experts we asked: What should be in new SWE course?
- Will have legacy lectures and programming assignments later in course

Question: Which type of SW is considered an epic failure?

- ☐ Beautiful code
- ☐ Legacy code
- ☐ Unexpectedly short-lived code
- ☐ Both legacy code and unexpectedly short lived code

# Development processes: Waterfall vs. Spiral vs. Agile

(*Engineering Long Lasting Software* §1.3)

David Patterson

# Development Processes: Waterfall vs.Sprial vs. Agile

- Waterfall "lifecycle" or development process
  - A.K.A. "Big Design Up Front" or BDUF

1. Requirements analysis and specification
2. Architectural design
3. Implementation and Integration
4. Verification
5. Operation and Maintenance

- Complete one phase before start next one
  - Why? Earlier catch bug, cheaper it is
  - Extensive documentation/phase for new people

# How well does Waterfall work?

- Works well for SW with specs that won't change: NASA spacecraft, aircraft control, …
- Often when customer sees it, wants changes
- Often after built first one, developers learn right way they should have built it

# How well does Waterfall work?

- *"Plan to throw one [implementation] away; you will, anyhow."*

- Fred Brooks, Jr.

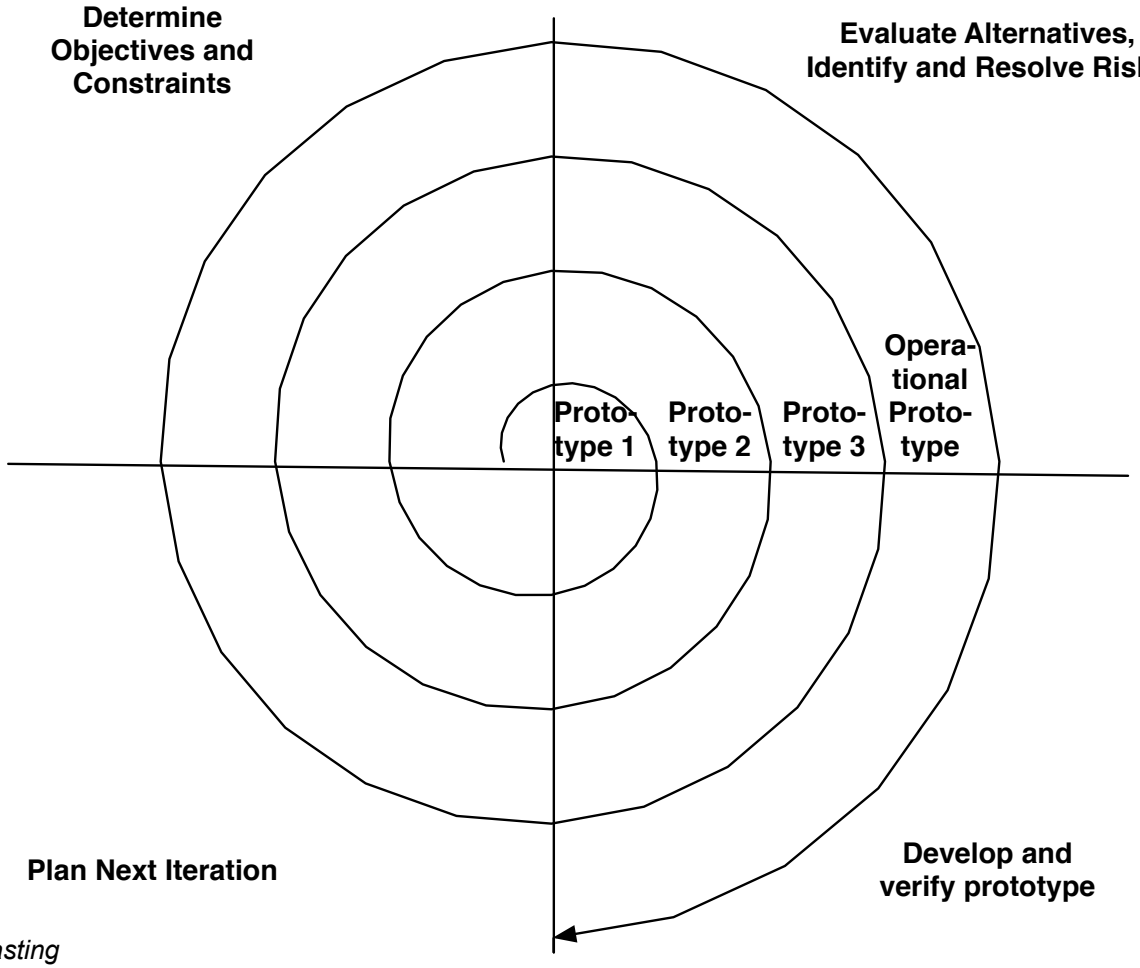
(received 1999 Turing Award for contributions to computer architecture, operating systems, and software engineering)

(Photo by Carola Lauber of SD&M www.sdm.de. Used by permission under CC-BY-SA-3.0.)

# Spiral Lifecycle

- Combine Big Design Up Front with prototypes
- Rather than document all requirements 1st, develop requirement documents across the iterations of prototype as they are needed and evolve with the project

# Spiral Lifecycle




(Figure 1.1, *Engineering Long Lasting Software* by Armando Fox and David Patterson, Beta edition, 2012.)

# Spiral Problems

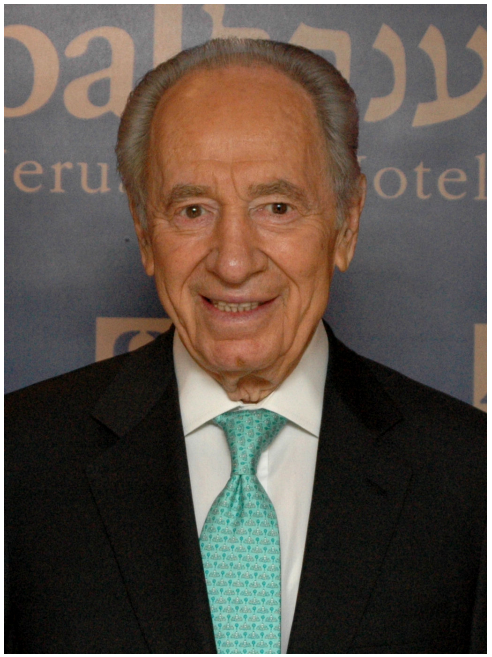
- Iterations involve the customer before the product is completed, which reduces chances of misunderstandings
- However, iterations 6 to 24 months long, so there is plenty of time for customers to change their minds!
- *And the users exclaimed with a laugh and a taunt: "It's just what we asked for, but not what we want."*

# Peres's Law

"If a problem has no solution,
it may not be a problem,
but a fact, not to be solved,
but to be coped with over time."

— Shimon Peres

(winner of 1994
Nobel Peace Prize
for Oslo accords)



(Photo Source: Michael Thaidigsmann, put in public domain,
See http://en.wikipedia.org/wiki/File:Shimon_peres_wjc_90126.jpg)

# Agile Manifesto, 2001

"We are uncovering better ways of developing SW by doing it and helping others do it. Through this work we have come to value

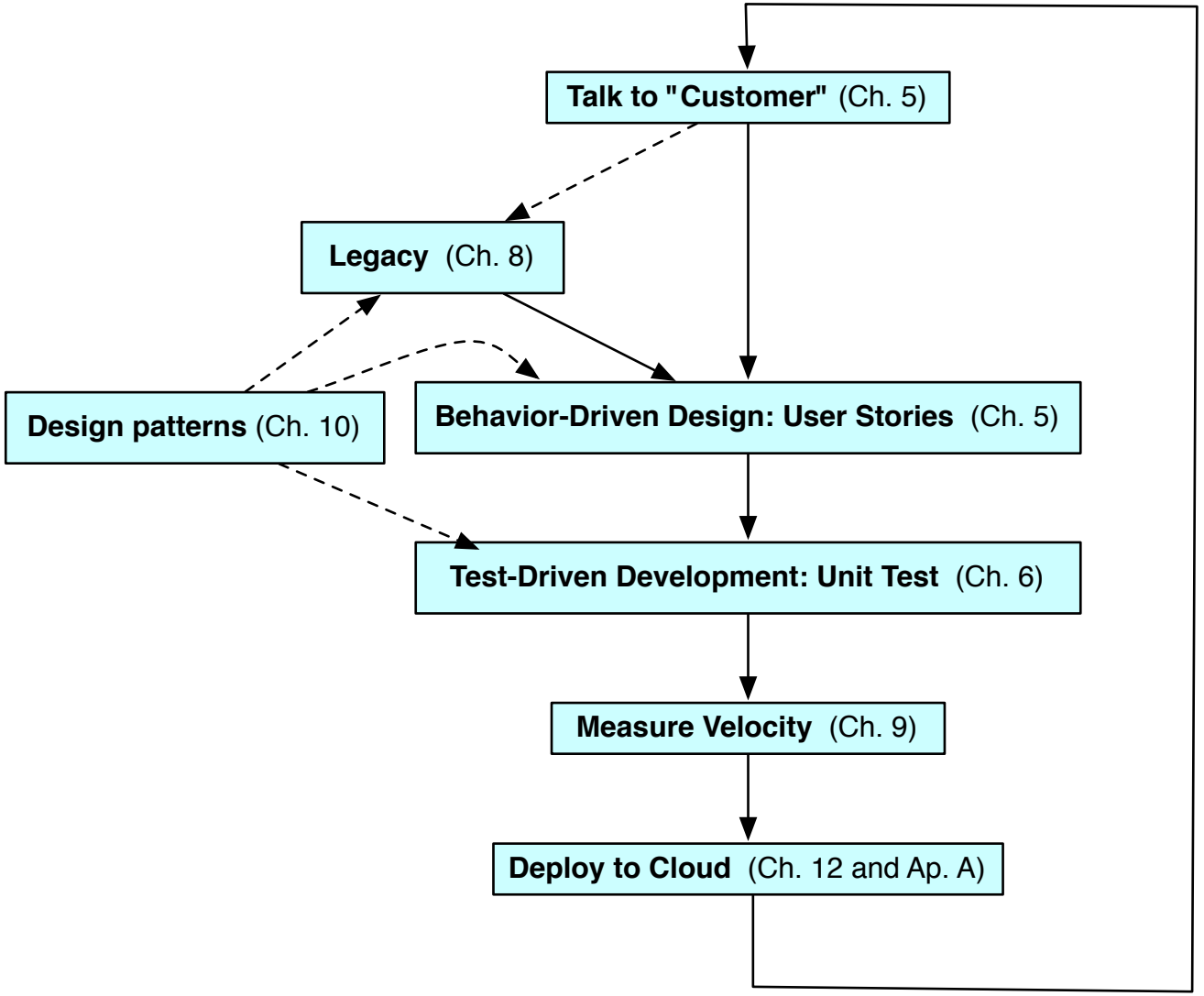
- Individuals and interactions over processes & tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more."

# Agile lifecycle

- Embraces change as a fact of life: continuous improvement vs. phases
- Developers continuously refine working but incomplete prototype until customers happy, with customer feedback on each Iteration (every ~2 weeks)
- Agile emphasizes Test-Driven Development (TDD) to reduce mistakes, written down User Stories to validate customer requirements, Velocity to measure progress

# Agile Iteration/ Book Organization




(Figure 1.5, *Engineering Long Lasting Software* by Armando Fox and David Patterson, Beta edition, 2012.)

# Question: What is NOT a key difference between Waterfall and Spiral and Agile lifecycles?

- ☐ Waterfall uses long phases, Agile uses quick iterations, Spiral long iterations
- ☐ Waterfall has no working code until end, Sprial & Agile working code each iteration
- ☐ Waterfall & Spiral use written requirements, but Agile does not use anything written
- ☐ Waterfall & Spiral have architectural design phases, but you cannot incorporate SW architecture into the Agile lifecycle

# Assurance: Testing and Formal Methods

(*Engineering Long Lasting Software* §1.4)

David Patterson

# Assurance

- Verification: Did you build the thing right?
  - Did you meet the specification?
- Validation: Did you build the right thing?
  - Is this what the customer wants?
  - Is the specification correct?
- Hardware focus generally Verification
- Software  focus generally Validation
- 2 options: Testing and Formal Methods

# Testing

- Exhaustive testing infeasible
- Divide and conquer: perform different tests at different phases of SW development
  - Upper level doesn't redo tests of lower level

_____test: integrated program meets its specifications

_____test: interfaces between units have consistent assumptions, communicate correctly

_____test: across individual units

___test: single method does what was expected

# Testing

- Exhaustive testing infeasible
- Divide and conquer: perform different tests at different phases of SW development
  - Upper level doesn't redo tests of lower level

System or acceptance test: integrated program meets its specifications

Integration test: interfaces between units have consistent assumptions, communicate correctly

Module or functional test: across individual units

Unit test: single method does what was expected

# More Testing

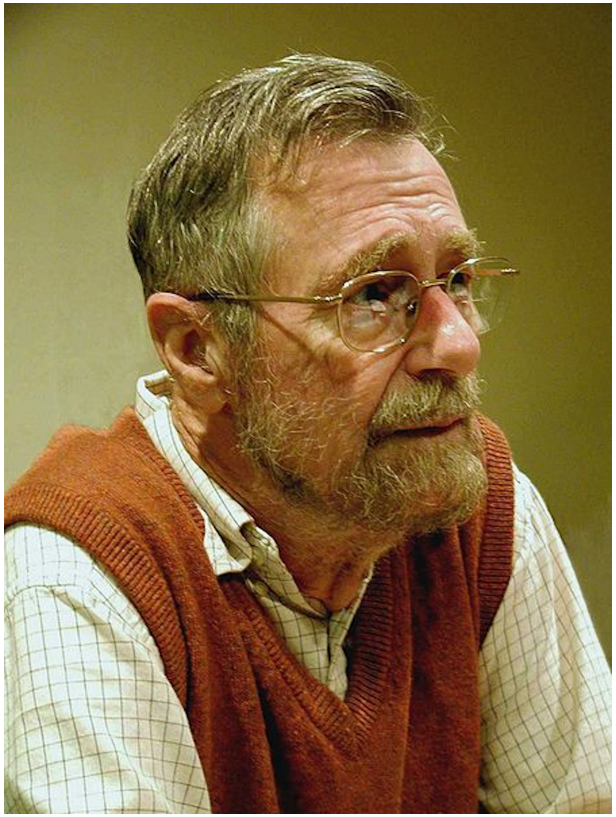
- Coverage: % of code paths tested
- Regression Testing: automatically rerun old tests so changes don't break what used to work
- Continuous Integration Testing: continuous regression testing vs. later phases
- Agile => Test Driven Design (TDD)
  write tests *before* you write the code you wish you had (tests drive coding)

# Limits of Testing

- Program testing can be used to show the _____ of bugs, but never to show their _____!

  – Edsger W. Dijkstra

(received the 1972 Turing Award for fundamental contributions to developing programming languages)

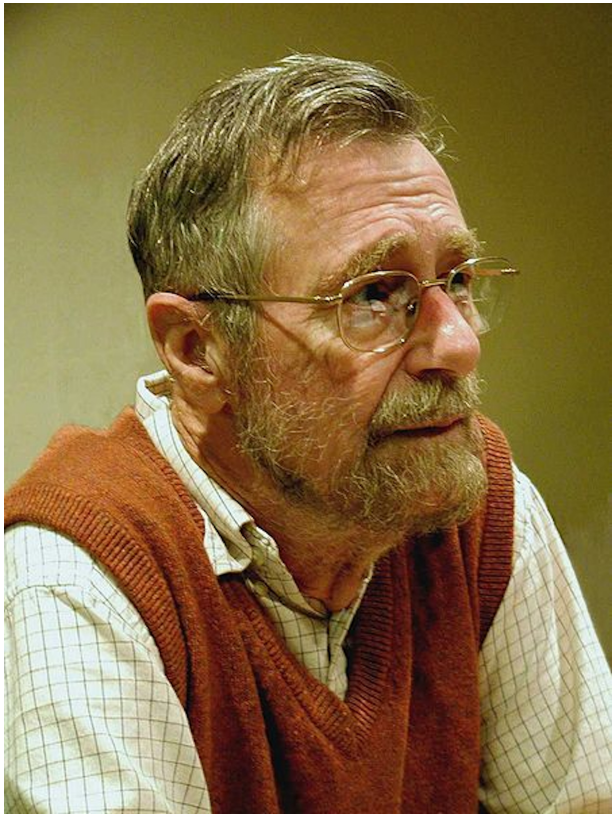


(Photo by Hamilton Richards. Used by permission under CC-BY-SA-3.0.)

# Limits of Testing

- Program testing can be used to show the presence of bugs, but never to show their absence!
  - Edsger W. Dijkstra

(received the 1972 Turing Award for fundamental contributions to developing programming languages)



(Photo by Hamilton Richards. Used by permission under CC-BY-SA-3.0.)

# Formal Methods

- Start with formal specification & prove program behavior follows spec. Options:

1. Human does proof
2. Computer via automatic theorem proving
   - Uses inference + logical axioms to produce proofs from scratch
3. Computer via model checking
   - Verifies selected properties by exhaustive search of all possible states that a system could enter during execution

# Formal Methods

- Computationally expensive, so use only if
  - Small, fixed function
  - Expensive to repair, very hard to test
  - E.g., Network protocols, safety critical SW
- Biggest: OS kernel 10K LOC @ $500/LOC
  - NASA SW $80/LOC
- This course: rapidly changing SW, easy to repair, easy to test => no formal methods

# Question: Which statement is NOT true about testing?

- ☐ With better test coverage, you are more likely to catch faults
- ☐ While difficult to achieve, 100% test coverage insures design reliability
- ☐ Each higher level test delegates more detailed testing to lower levels
- ☐ Unit testing works within a single class and module testing works across classes

# Productivity: Conciseness, Synthesis, Reuse, and Tools


(*Engineering Long Lasting Software* §1.5)

David Patterson

# Productivity

- Moore's Law => 2X resources/1.5 years
  - ⇒ HW designs get bigger
  - ⇒ Faster processors and bigger memories
  - ⇒ SW designs get bigger
  - ⇒ Had to improve HW & SW productivity
- 4 techniques
  1. Clarity via conciseness
  2. Synthesis
  3. Reuse
  4. Automation and Tools

# Clarity via conciseness

1. Syntax: shorter and easier to read
   assert_greater_than_or_equal_to(a,7)
   vs. _____
2. Raise the level of abstraction:
   - HLL programming languages vs. assembly lang
   - Automatic memory management (Java vs.C)
   - Scripting languages: reflection, metaprogramming

# Clarity via conciseness

1. Syntax: shorter and easier to read
   assert_greater_than_or_equal_to(a,7)
   vs. a.should be ≥ 7
2. Raise the level of abstraction:
   - HLL programming languages vs. assembly lang
   - Automatic memory management (Java vs.C)
   - Scripting languages: reflection, metaprogramming

# Synthesis

- Software synthesis
  - BitBlt: generate code to fit situation & remove conditional test
- Research Stage: Programming by example

# Reuse

- Reuse old code vs. write new code
- Techniques in historical order:

1. Procedures and functions
2. Standardized libraries (reuse single task)
3. Object oriented programming: reuse and manage *collections* of tasks
4. Design patterns: reuse a general strategy even if implementation varies

# Automation and Tools

- Replace tedious manual tasks with automation to save time, improve accuracy
  - New tool can make lives better (e.g., make)
- Concerns with new tools: Dependability, UI quality, picking which one from several
- We think good software developer must repeatedly learn how to use new tools
  - Lots of chances in this course: Cucumber, RSpec, Pivotal Tracker, …

# Question: Which statement is TRUE about productivity?

- ☐ Copy and pasting code is another good way to get reuse
- ☐ Metaprogramming helps productivity via program synthesis
- ☐ Of the 4 productivity reasons, the primary one for HLL is reuse
- ☐ A concise syntax is more likely to have fewer bugs and be easier to maintain

# DRY

- “Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.”
  - Andy Hunt and Dave Thomas, 1999
- Don't Repeat Yourself (DRY)
  - Don’t want to find many places have to apply same repair
- Refactor code so that can have a single place to do things

# Software as a Service (SaaS)


(*Engineering Long Lasting Software* §1.6)

David Patterson

# Software as a Service: SaaS

- Traditional SW: binary code installed and runs wholly on client device
- SaaS delivers SW & data as service over Internet via thin program (e.g., browser) running on client device
  - Search, social networking, video
- Now also SaaS version of traditional SW
  - E.g., Microsoft Office 365, TurboTax Online
- Instructors think SaaS is revolutionary

# 6 Reasons for SaaS

1. No install worries about HW capability, OS
2. No worries about data loss (at remote site)
3. Easy for groups to interact with same data
4. If data is large or changed frequently, simpler to keep 1 copy at central site
5. 1 copy of SW, controlled HW environment => no compatibility hassles for developers
6. 1 copy => simplifies upgrades for developers *and* no user upgrade requests

# SaaS Loves Agile & Rails

- Frequent upgrades matches Agile lifecycle
- Many frameworks for Agile/SaaS
- We use Ruby on Rails ("Rails")
- Ruby, a modern scripting language: object oriented, functional, automatic memory management, dynamic types, reuse via mix-ins, synthesis via metaprogramming
- Rails popular – e.g., Twitter

# Why take time for Ruby/Rails?

- 10 weeks to learn:
  - SaaS, Agile, Pair Programming, Behavior Driven Design, LoFi UI, Storyboards, User Stories, Test Driven Development, Enhance Legacy Code, Scrum, Velocity, JavaScript, Design Patterns, UML, Deployment, Security
  - Part time (taking 3 other classes or full time job)
- Only hope is highly productive language, tools, framework: We believe Rails is best
  - See "Crossing the Software Education Chasm," A. Fox, D. Patterson, *Comm. ACM*, May 2012

# Which is WEAKEST argument for a Google app's popularity as SaaS?

- ☐ Don't lose data: Gmail
- ☐ Cooperating group: Documents
- ☐ Large/Changing Dataset: YouTube
- ☐ No field upgrade when improve app: Search

# And in Conclusion

- CS169 teaches software engineering principles that you demonstrate via an app developed by a team of 4 or 5 for non-technical customer deployed in Cloud
- Agile lifecycle – working prototype iterations
- Test driven development: unit to acceptance
- Productivity: Conciseness, Synthesis, Reuse, Tools/Automation
- SaaS less hassle for developers and users