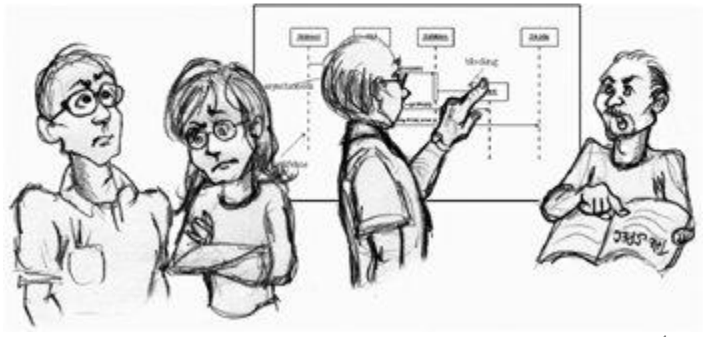
המכללה האקדמית להנדסה ירושלים

הנדסת תוכנה

9. מבוא לעבודה עם קוד קיים (עקרונות תיכון מונחה עצמים I (OODP I

“Simplicity is prerequisite for reliability”
- E. W. Dijkstra

"Writing code a computer can understand is science.
Writing code other programmers can understand is an art.“, Jason Gorman



ses\se\
es.zip

מקורות

- Pressman 8.13.11
- Robert Martin,
  Design Principles and Design Patterns
  http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf (pointers article)
  - Books: "Agile Software Development, Principles, Patterns, and Practices", "Agile Principles, Patterns, and Practices in C#"
- Metz, Practical Object Oriented Design in Ruby (motivation for design 1:40min)
- Smith, http://www.pluralsight-training.net/microsoft/Courses#software-practices
  Head First OOA&D
- Motivation slides
  www.lostechies.com/blogs/derickbailey/archive/2009/02/11/solid-development-principles-in-motivational-pictures.aspx

קישורים

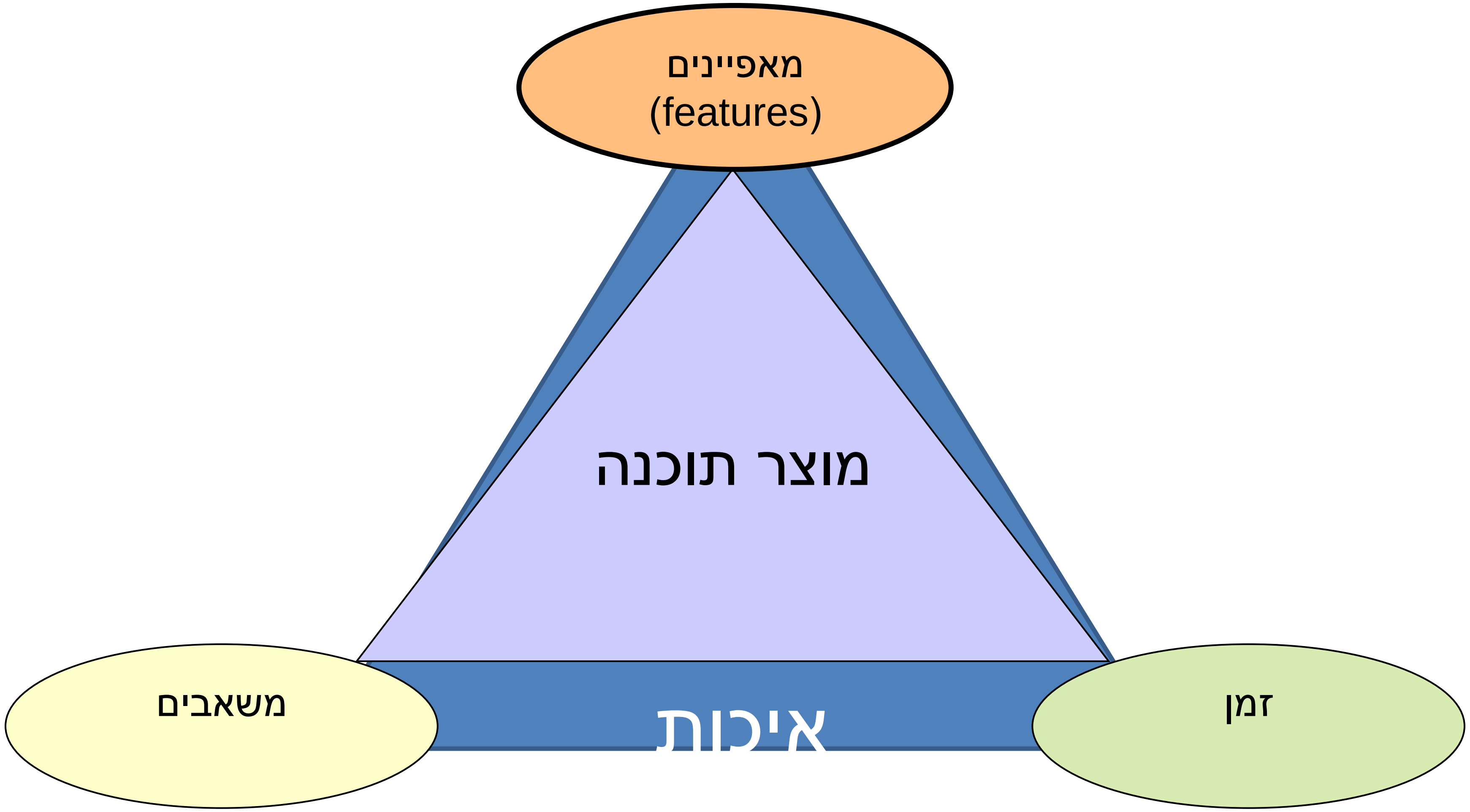
- OOP+SOLID (short series) ThingsYouShouldKnow
- Ottinger&Langr, Agile Design Series, Cohesive, Coupling, Abstraction, Volatility, PragBub Mag. , 2010-11
- SOLID presentation infoq http://www.infoq.com/presentations/SOLID-Software-and-Design-Patterns-for-Mere-Mortals
- "Software Design Patterns For Everyone” http://amazedsaint.net/patterns.pdf
- http://sourcemaking.com/, stuff on design patterns, uml, refactoring
- DI&IOC, blog 2010
- Pablo’s SOLID Software Development ebook

- Likness: "Solid & Dry“ posts Part 1, Part 2
- Robert C. Martin (Uncle Bob) materials:
- Main article:Design Principles and Design Patterns
- Links to detailed articles
- Interviews: hanselminutes, Pragmatic podcast: What's on Uncle's Bob Mind
- Clean code video: Øredev 2008 - Agile - Clean Code III: Functions Favorite
- http://www.artima.com/weblogs/viewpost.jsp?thread=250296
- Video: Neil Ford, Emergent Design
- Presentation: Clean Code, 2010

מה היום?

- ראינו: בדיקות ברמות שונות כדרך למוצר איכותי
- עוד על איכות תוכנה: עקרונות תיכון מונחה עצמים Object Oriented Design Principles - חלק א'
- הדגמת העקרונות (כולל בדיקות)
- בפעם הבאה (אחרי הרצאת אורח) – חלק ב': המשך העקרונות ותבניות תיכון
- הרצאה 3\תרגיל:
  - סקר איכות התיכון – בעדיפות מקום שכבר גיליתם קושי לעבוד איתו
  - בפרויקט – תיעוד של הבעיה והצעה לפתרון כמה משפטים בויקי + הצגה בסקר

תזכורת: פרויקט תוכנה:



איכות תוכנה

- מרכיבים חיצוניים:
  - נראים ללקוח\למשתמש
  - דוגמאות:
    - בעלי ערך ללקוח!
    - שמישות
    - נכונות
    - עמידות
    - הרחבתיות?

- מרכיבים פנימיים
  - נראים למפתחים
  - דוגמאות:
    - הסתרת מידע
    - עקיבות
    - פשטות \ קריאות
    - סגנון הקוד
    - יכולת להשתנות

[Begel & Simon 08], עובדים חדשים במיקרוסופט מבלים את רוב השנה הראשונה בקריאת קוד

איכות תוכנה

- איך משיגים תוכנה איכותית?
- ראינו יעדים כלליים: צימוד (coupling) נמוך, לכידות (cohesion) גבוהה
  כיצד משיגים אותם?
- עקרונות + תבניות + הרגלים = תוכנה איכותית
- ~~פיתוח תוכנה מונחה עצמים שולט, לכן נדון מכיוון זה~~
- ~~מצד שני: Panel: Objects On Trial, Data Oriented Design~~

מתוך קורס הנדסת תוכנה בברקלי

- קורס בן כמה שנים מקביל לשלנו (וידאו)
  - Fox & Patterson
  - peer instruction
- לאחרונה כקורס מקוון בשילוב תוכנה כשירות (SaaS) > 50K סטודנטים
- קורס בחירה דומה במכללה (ענן וה"ת II)
  - נלמד הפעם כמה עקרונות כלליים יותר
  - לפי הזמן גם מונחה עצמים
  - גיוון והשוואה

Legacy Code & Refactoring

Armando Fox, David Patterson, and Koushik Sen

Spring 2012

Outline

- What is Legacy & How Can Agile Help?
- High-Level Architecture Exploration
- Code Base Exploration
- Establishing Ground Truth By Adding Tests
- Intro to Code Smells and Design Smells
- Good Methods are SOFA
- Method-Level Refactoring
- A Good Class Architecture is SOLID
- Class-Level Refactoring
- Improving Internal Documentation

What Makes Code “Legacy” and How Can Agile Help?

Armando Fox

Legacy Code Matters

- Maintenance typically consumes 40 to 80% (average: 60%) of software costs. Therefore, *it is probably the most important life cycle phase of software . . .*

"Old hardware becomes obsolete; old software goes into production every night."

Robert Glass, *Facts & Fallacies of Software Engineering (fact #41)*

Maintenance != bug fixes

- Enhancements: 60% of maintenance costs
- Bug fixes: 17% of maintenance costs

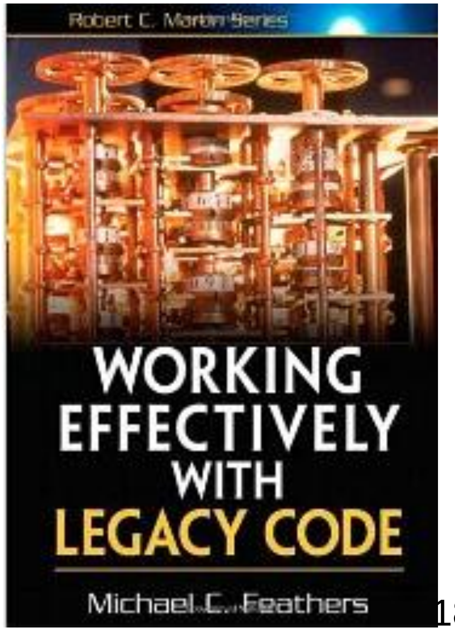
Hence the “60/60 rule”:

- 60% of software cost is maintenance
- 60% of maintenance cost is enhancements.

Glass, R. *Software Conflict.* Englewood Cliffs, NJ: Yourdon Press, 1991

What makes code "legacy"?

- Still meets customer need, **AND:**
- You didn't write it, and it's poorly documented
- You did write it, but a long time ago (and it's poorly documented)
- *It lacks good tests (regardless of who wrote it)*—Feathers 2004



2 ways to think about modifying legacy code

- Edit & Pray
  - "I kind of think I probably didn't break anything"

- Cover & Modify
  - Let *test coverage* be your safety blanket

How Agile Can Help

1. **Exploration:** determine where you need to make changes (*change points*)
2. **Refactoring**: is the code around change points (a) tested? (b) testable?
   - (a) is true: good to go
   - !(a) && (b): apply BDD+TDD cycles to improve test coverage
   - !(a) && !(b): **refactor**

How Agile Can Help, cont.

3. Add tests to **improve coverage** as needed
4. **Make changes**, using tests as *ground truth*
5. **Refactor** further, to leave codebase better than you found it

- This is "embracing change" on long time scales

"Try to leave this world a little better than you found it."

Lord Robert Baden-Powell, founder of the Boy Scouts

If you've been assigned to modify legacy code, which statement would make you happiest if true?

- ☐ "It was originally developed using Agile techniques"
- ☐ "It is well covered by tests"
- ☐ "It's nicely structured and easy to read"
- ☐ "Many of the original design documents are available"

Approaching & Exploring Legacy Code

Armando Fox

Goals of Exploration

- Understand how the app works, from its various stakeholders’ point of view
- Understand design based on
  - available design documents
  - “informal” design documents
  - creating and drawing architectural diagrams
- Understand “big picture” of the code
  - enough to start zooming in on where you’ll be making changes

Interlude/Armando's Computer History Minute

- Always mount a scratch monkey

- More folklore: http://catb.org/jargon

Kicking the tires

- Run the app, and/or watch customer use it
- Get customer to do demo and talk you through what they're doing
- Check out a *scratch branch* that won't be checked back in, and get it to run
  - In a production-like setting or development-like setting
  - Ideally with something resembling a copy of production database
  - Some systems may be too large to clone

Look at design documents

- [This slide intentionally left blank]

Look at “informal” design docs

- Lo-fi UI mockups and user stories, especially if they are executable integration tests like Cucumber scenarios
- Photos of whiteboard sketches about the application architecture, class relationships, etc.
- Unit, functional and integration tests, especially if they are written with ease-of-reading in mind, like RSpec specs
- Comments in the code (sometimes)
- READMEs and technical documentation in the source tree

Informal design docs, cont.

- Documentation embedded in the code (e.g. RDoc)
- Archived email, newsgroup, internal wiki pages or blog posts, etc. about the project
- Transcripts, notes, or video recordings of code reviews and design reviews (like Campfire or Basecamp)
- Commit logs in the version control system (**`git log`**)

Code base exploration

- “Read all the code in one hour”
  (Nierstrasz et al., *Object-Oriented Reengineering Patterns,* 2009)
  - *Non-goal:* understand all the code
- Goals: high-level “gestalt” understanding
  - What is the subjective code quality? (Elegant? overly terse? tangled?)
  - How much code is there, and how much test code? (**rake stats**)
  - If quality varies, how bad are frequently-changing parts?
  - What are the major subdivisions? (e.g.if MVC framework, what are models/views/controllers?)
  - What’s database schema? (**rake db:schema:dump)**

Example: *railroady*

- Create a model interaction diagram automatically (**gem install railroady**)
- Create diagram manually by inspection
- Look for highly-connected classes
- What are the main *classes*, their *responsibilities*, and their *collaborators?*

Class-Responsibility-Collaboration (CRC) cards

- Proposed by Kent Beck & Ward Cunningham,1989
  - *A Laboratory for Teaching Object-Oriented Thinking,* OOPSLA'89 & SIGPLAN Notices 24(10)
- Goal: think about app in terms of objects (vs. procedurally) from the very beginning
- Process: for each of several *scenarios* (stories)*:*
  - identify the *classes* (or actors in scenario)
  - identify *responsibilities* of each—things it knows or does
  - identify *collaborators*—perform actions for it or manage information it needs
  - modify/refine/replace cards as go thru more scenarios

CRC card examples

Showing	
Responsibilities	*Collaborators*
Knows name of movie	Movie
Knows date & time	
Computes ticket availability	Ticket

Ticket	
Responsibilities	*Collaborators*
Knows its price	
Knows which showing it's for	Showing
Computes ticket availability	
Knows its owner	Patron

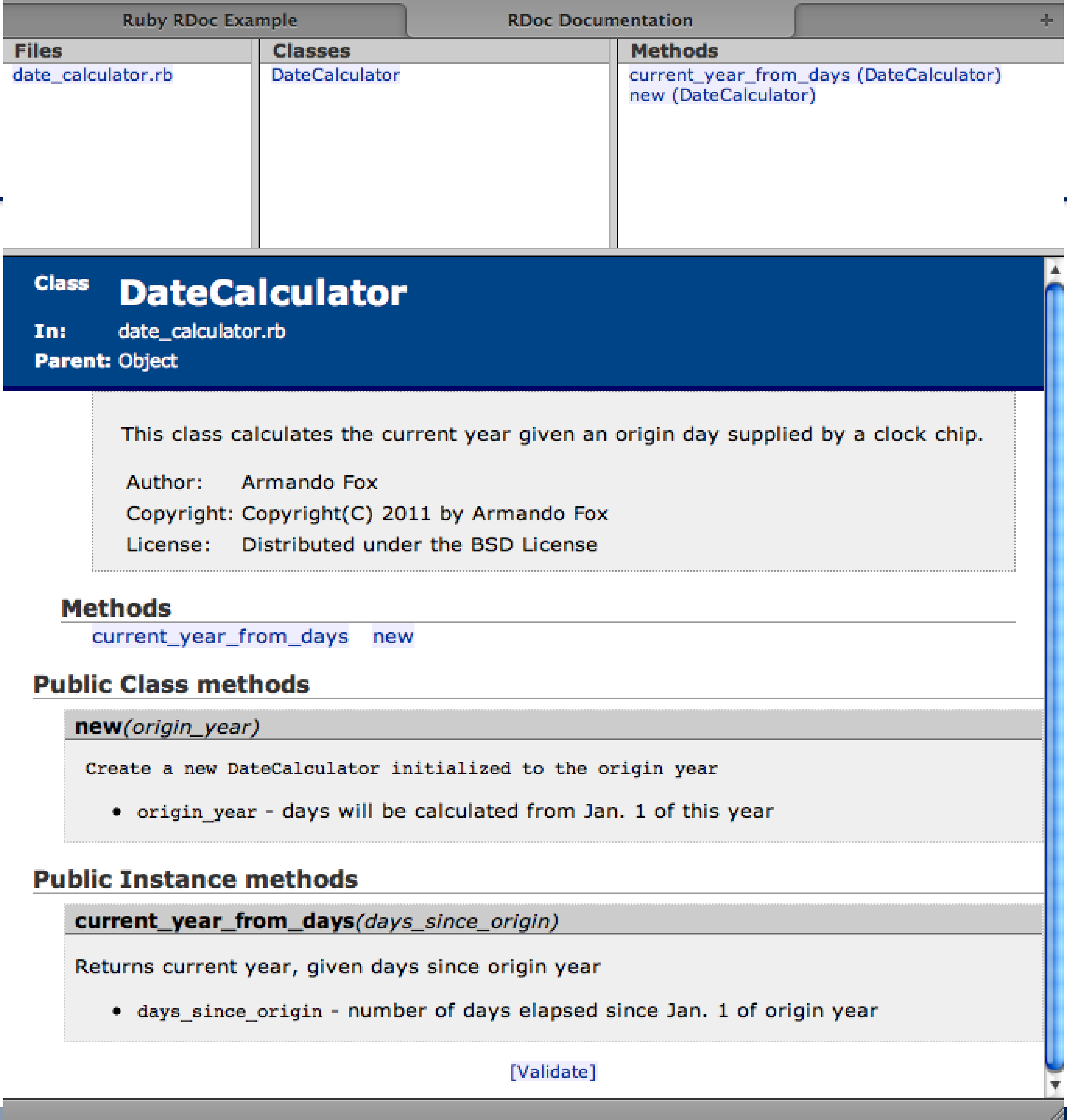
Order	
Responsibilities	*Collaborators*
Knows how many tickets it has	Ticket
Computes its price	
Knows its owner	Patron
Knows its owner	Patron

Insert inline documentation

- RDoc gem (like Javadoc) extracts documentation from code

http://pastebin.com/QARUzTnh

- **rake doc** generates HTML docs for entire Rails project



Files
date_calculator.rb

Classes
DateCalculator

Methods
current_year_from_days (DateCalculator)
new (DateCalculator)

Class **DateCalculator**
In: date_calculator.rb
Parent: Object

This class calculates the current year given an origin day supplied by a clock chip.

Author:	Armando Fox
Copyright:	Copyright(C) 2011 by Armando Fox
License:	Distributed under the BSD License

Methods
current_year_from_days new

Public Class methods

new(*origin_year*)

Create a new DateCalculator initialized to the origin year

- origin_year - days will be calculated from Jan. 1 of this year

Public Instance methods

current_year_from_days(*days_since_origin*)

Returns current year, given days since origin year

- days_since_origin - number of days elapsed since Jan. 1 of origin year

[Validate]

Summary: Exploration

- "Size up" the overall code base
- Identify key classes and relationships
- Identify most important data structures
- Ideally, identify place(s) where change(s) will be needed
- Keep design docs as you go
  - diagrams
  - GitHub wiki
  - comments you insert using RDoc

Which statement can you reasonably expect to be TRUE while doing exploration?

- ☐ Exploration is a reasonable time to fix "minor" aesthetic problems in the code
- ☐ Once I discover important collaborations between classes, I'll be able to stub them
- ☐ If it's working in production, it should be easy to get it to run in development
- ☐ It's worth capturing the way it works now , since it's serving a customer need

Establishing Ground Truth With Tests

Armando Fox

Tests Are Your Friend

- When modifying your own code without tests, you're cocky
- When modifying someone else's code without tests, you should be terrified
- *Tests establish <u>ground truth</u> against which future changes can be compared.*

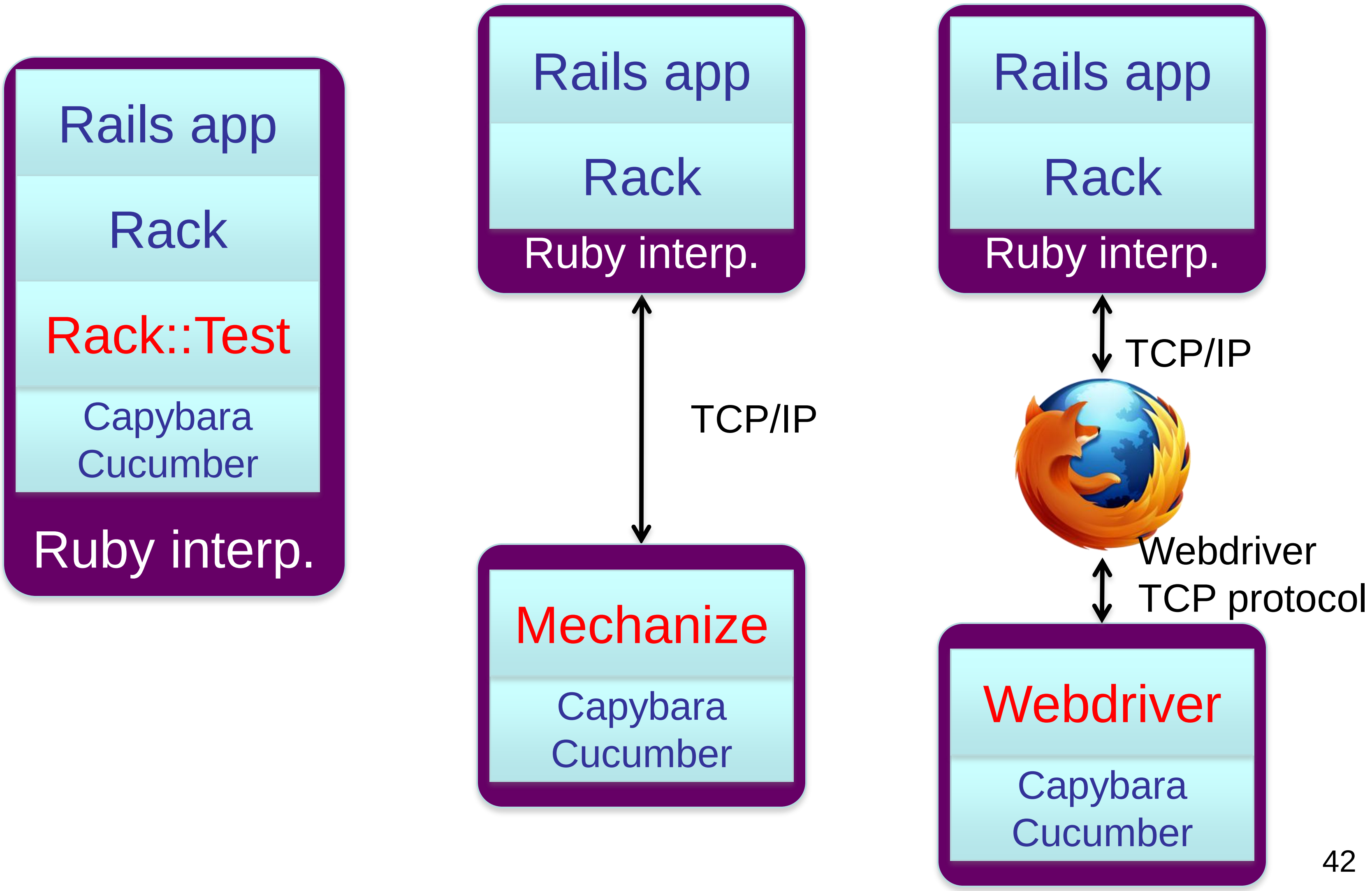
Characterization Tests

- Captures *how the app works today*
  - *Even if that behavior is "buggy"*
- Makes known behaviors **R**epeatable
- Goal: increase confidence that you're not breaking anything
- **Pitfall: don't try to make improvements at this stage!**
  - Current goal is to *establish confidence* that you won't break stuff when making changes
  - "This change is easy" is almost *never* right

Integration-Level Characterization Tests

- Natural first step, since black-box/integration level
  - don't rely on your understanding app structure
- Use the Cuke, Luke
  - Additional Capybara back ends like Mechanize make almost everything scriptable
  - Do imperative scenarios now
  - Convert to declarative or improve `Given` steps later when you understand app internals

In-process vs. out-of-process



Unit- and Functional-Level Characterization Tests

- Cheat: write tests to learn about code as you go

```
it "should calculate sales tax" do
  order = mock('order')
  order.compute_tax.should == -99.99
end
# object 'order' received unexpected message 'get_total'
it "should calculate sales tax" do
  order = mock('order', :get_total => 100.00)
  order.compute_tax.should == -99.99
end
#   expected compute_tax to be -99.99, was 8.45
it "should calculate sales tax" do
  order = mock('order', :get_total => 100.00)
  order.compute_tax.should == 8.45
end
```

Identifying What's Wrong: Smells & Metrics

Armando Fox

Alpha Version Warning…

What's wrong with this method?

http://pastebin.com/gtQ7QcHu

- Variable names not descriptive
- Structure too complex: think about testing the nested conditional
- Too long
  - Ancient wisdom: function <= 1 screenful, so can quickly grasp *main purpose* of method
  - But today monitors display 10x chars as 1980s!
- Lacks documentation
  - != comments, though that's also true here

What are code smells?

- Like a real smell, alerts you that something *may* not be right
  - sometimes a false alarm!
- Method level: method code is inelegant, non-DRY, hard to read, …
- Class level: division of labor and code among classes results in inelegance, non-DRYness, hard to read, …
  - sometimes called *design smells* in this context
- Sometimes self-evident, other times becomes evident when you try to make changes

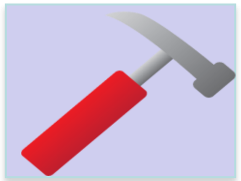
Nailing it down a bit better

- Quantitative: code complexity metrics
  - cyclomatic complexity
  - ABC score (assignment, branch, condition)
  - LCOM (lack of cohesion of methods)
- Qualitative: *code smells & design smells*
  - long method
  - long class
  - data clumps
  - shotgun surgery
  - inappropriate intimacy …

saikuro

flog

reek

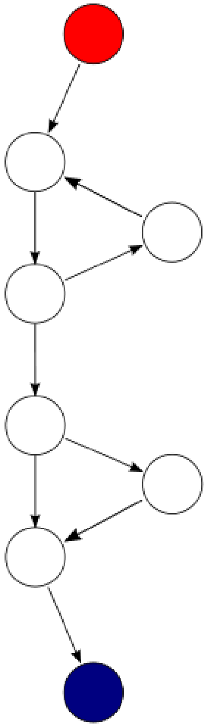


...metric_fu

Cyclomatic complexity (McCabe, 1976)

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

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



- Here, E=9, N=8, P=1, so CC=3

Metric	Tool	Target score
Code-to-test ratio	rake stats	≤ 1:2
C0 coverage	SimpleCov	90%+
ABC 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*

Which is generally FALSE about code smells?

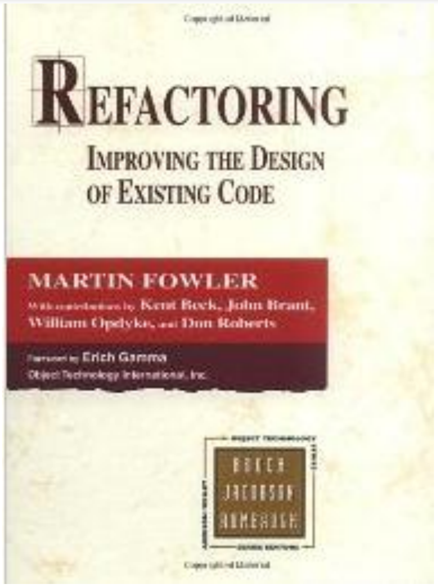
- ☐ They can occur both within a class and in interactions among classes
- ☐ They may indicate correctness problems
- ☐ They do not necessarily require repair
- ☐ More code is bad; less code is good

Intro to Method-Level Refactoring

Armando Fox

History & Context

- Fowler et al. developed mostly definitive catalog of refactorings
  - – Adapted to various languages
  - – Method- and class-level refactorings
- Each refactoring consists of:
  - – Name
  - – Summary of what it does/when to use
  - – Motivation (what problem it solves)
  - – Mechanics: step-by-step recipe
  - – Example(s)



REFACTORING

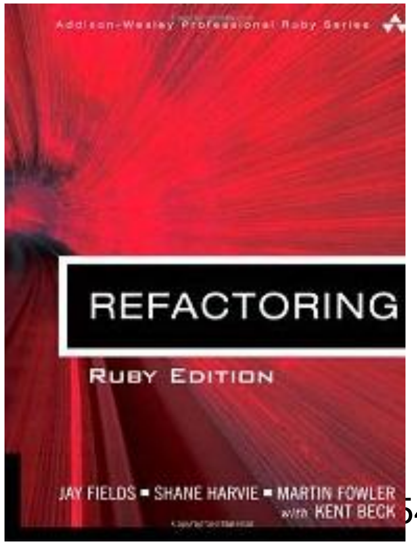
IMPROVING THE DESIGN OF EXISTING CODE

MARTIN FOWLER

With contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts

Foreword by Erich Gamma

Object Technology International, Inc.



Addison-Wesley Professional Ruby Series

REFACTORING

RUBY EDITION

JAY FIELDS ▪ SHANE HARVIE ▪ MARTIN FOWLER with KENT BECK

Refactoring: Idea

- Start with code that has 1 or more problems/smells
- Through a series of *small steps*, transform to code from which those smells are absent
- Each change step is protected by tests to the extent possible
- *Minimize time during which tests are red*

Fixing TimeSetter

- Fix stupid names http://pastebin.com/pYCfMQJp
- Extract method http://pastebin.com/sXVDW9C6
- Extract method, encapsulate class http://pastebin.com/zWM2ZqaW
- Test extracted methods http://pastebin.com/DRpNPzpT
- Some thoughts on unit testing
  - Glass-box testing can be useful while refactoring
  - Common approach: test *critical values* and *representative noncritical values*

What did we do?

- Made date calculator easier to read and understand using simple *refactorings*
- Found a bug

http://pastebin.com/0Bu6sMYi

- Observation: if we had developed method using TDD, might have gone easier!
- Did we improve our **flog** & **reek** scores?

Other Examples & Remedies

Smell	Refactoring that may resolve it
Large class	Extract class, subclass or module
Long method	Decompose conditional Replace loop with collection method Extract method Extract enclosing method with `yield()` Replace temp variable with query Replace method with object
Long parameter list/data clump	Replace parameter with method call Extract class
Shotgun surgery; Inappropriate intimacy	Move method/move field to collect related items into one DRY place
Too many comments	Extract method introduce assertion replace with internal documentation
Inconsistent level of abstraction	Extract methods & classes

Which is NOT a goal of method-level refactoring?

- ☐ Reduce code complexity
- ☐ Eliminate code smells
- ☐ Eliminate bugs
- ☐ Improve testability

Good Methods are SOFA

Armando Fox

What makes a good method?

- What makes a news article easy to read?
- Good: start with a high level summary of key points, then go into each point in detail
- Bad: ramble on, jumping between “levels of abstraction” rather than progressively refining

Methods should be SOFA

- Be short
- Do one thing
- Have few arguments
- Consistent level of abstraction

- You can use these as a checklist
- Having trouble coming up with a unit test strategy? Try the checklist

It should be short

- If you have to scroll to read it, it’s too long
- Why? Because *what it does* should be quick to grasp
- If it’s a compound task, it should probably be split up across >1 function
- *Most of the other desiderata for functions can be derived from this one*

It Should Have Few Arguments

- Lots of arguments == testing badness
  - Code coverage is hard: combinatorial explosion
  - Isolation is hard: may have to mock or stub a lot of stuff to isolate effects of varying 1 argument
  - In general, excessive coupling of tests
- Boolean arguments should be a yellow flag
  - If function behaves pretty differently based on Boolean argument value, maybe should be 2 functions

Arguments, cont.

- What if your functions *need* to pass a lot of arguments to communicate with each other?
- If they share that much context, maybe you need a new class.

  (the Extract Class refactoring)

Example: AvailableSeat

- Real Example: *AvailableSeat*
  - Shows have seat inventory for sale, at different prices and for different sections (premium vs. regular, eg)
  - Some seats only available to “VIP” customers
  - Some seat types only sold for certain # of days prior to showdate, or have limited inventory
- Result: Same “seat” has different availability restrictions depending on customer, show, time, ...
  - Theater Manager can override all restrictions
- *Scenario:* customer comes to website and wants to buy a ticket. Which class “owns” computing the available seats for *this customer?*

Single Level of Abstraction

- Complex tasks need divide & conquer
- Yellow flag for "encapsulate this task in a method":
  - line N of function says *what to do*
  - but line N+1 says *how to do* something
- Example: encourage customers to opt in

http://pastebin.com/AFQAKxbR

SOFA & Unit Testing

- Few arguments => can test all important combinations
- Lots of short functions => can selectively mock out as needed
- Do one thing => each test can focus on corner cases for one particular functionality

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

- ☐ Short
- ☐ Do one thing
- ☐ Have few arguments
- ☐ Stick to one level of abstraction

SOLID Class Architecture & Class-Level Refactoring

Armando Fox

Alpha Content Warning

α

SOLID OOP principles

- Elucidated by Robert C. Martin and other co-authors of *Agile Manifesto*
- *Concrete*, implementable suggestions for keeping your code modular

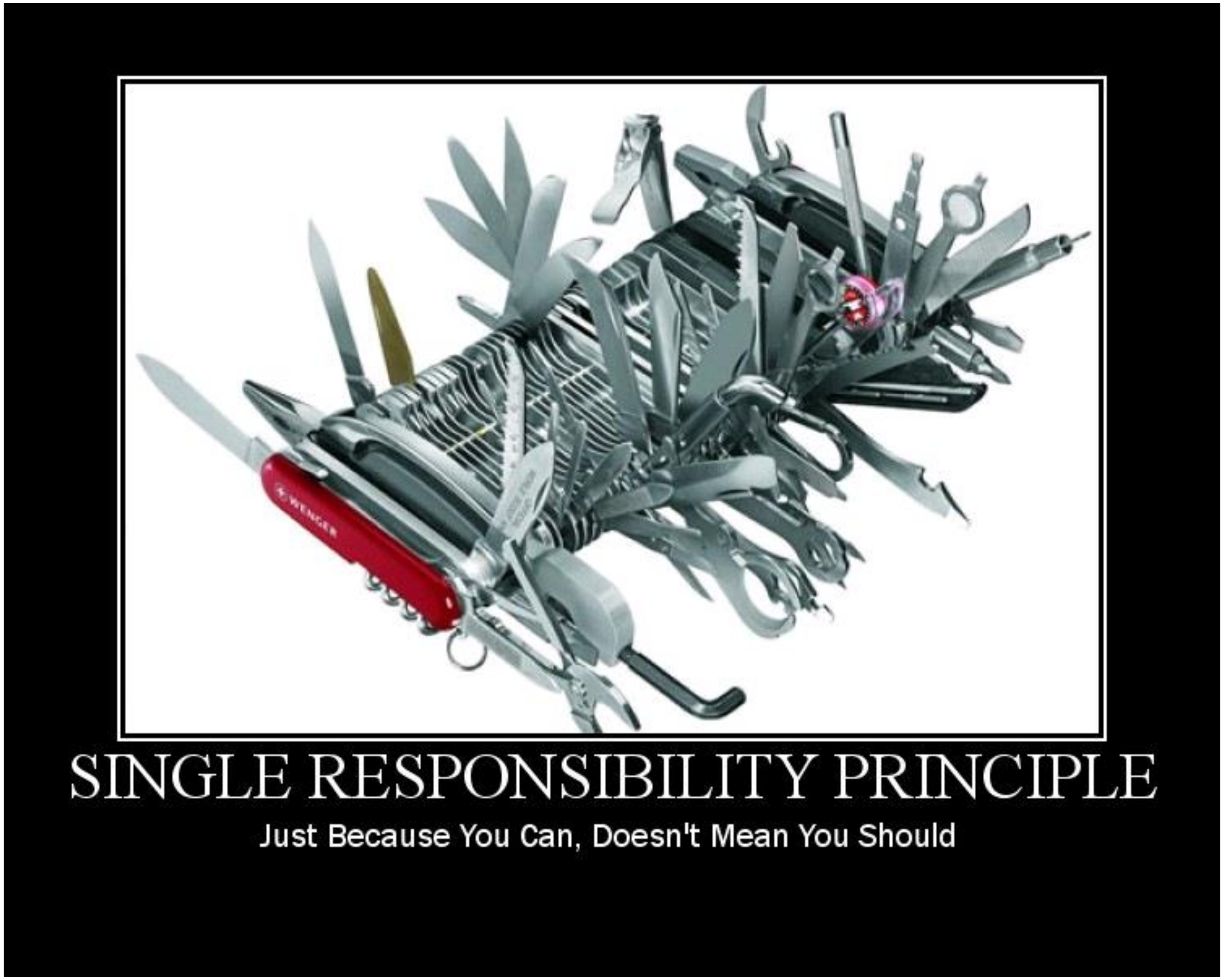
SOLID OOP principles

- Single Responsibility principle
- Open/Closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle
- Demeter principle
- *Common motivation: minimize cost of change*

JCE STOP

עקרונות תומכים במודולאריות, Martin - SOLID

- The Single-Responsibility Principle - **SRP** - A class should have only one reason to change.
- The Open-Closed Principle - **OCP** - A class should be extensible without requiring modification
- The Liskov Substitution Principle - **LSP** - Derived classes should be substitutable for their base classes
- The Dependency Inversion Principle - **DIP** - Depend upon abstractions. Do not depend upon concretions
- The Interface Segregation Principle - **ISP** - Many client specific interfaces are better than one general purpose interface.


WENGER
SINGLE RESPONSIBILITY PRINCIPLE
Just Because You Can, Doesn't Mean You Should

Single-Responsibility Principle (SRP)

A class should have only one reason to change (Martin).

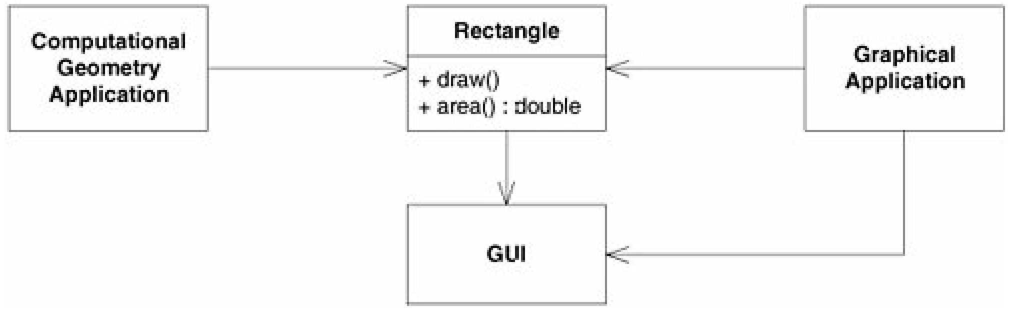
- אחריות היא סיבה לשינוי
- התפקידים שיש למחלקה הם צירי שינוי. אם יש לה שני תפקידים הם צמודים ביחד ומשתנים ביחד
- עיקרון פשוט אך לא תמיד קל להגיע אליו
- הדרך: האצלה (delegation)

מה הקשר ללכידות?

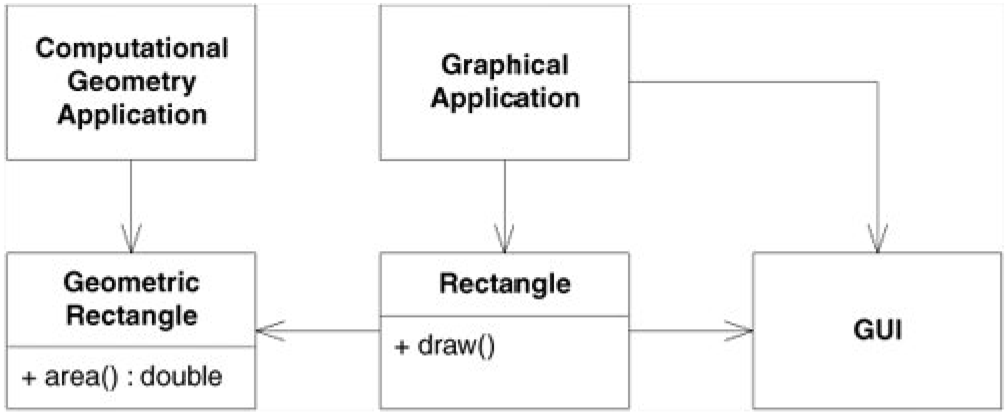
על אילו סימנים התגברנו?

SRP

- מה הבעיה כאן?



- פתרון:



```
public class PrintServer
{
    public string CreateJob(PrintJob data) { //...
    }
    public int GetStatus(string jobId) { //...
    }
    public void Print(string jobId, int startPage, int endPage) { //...
    }


    public List<Printer> GetPrinterList() { //...
    }
    public bool AddPrinter(Printer printer) { //...
    }

    public event EventHandler<JobEvent> PrintPreviewPageComputed;

    public event EventHandler PrintPreviewReady;

    // ...
}
```

```
public class PrintServer {


    public string CreateJob(PrintJob data) { //...
    }

    public int GetStatus(string jobId) { //...
    }

    public void Print(string jobId, int startPage, int endPage) { //...
    }

}


public class PrinterList {

    public List<Printer> GetPrinterList() { //...
    }

    public bool AddPrinter(Printer printer) { //...
    }
}
```

בפעם הבאה

- אורח – (לכל קבוצה סביבת עבודה עם בדיקות יחידה)
- בהמשך לפי הזמן:
- המשך עקרונות תיכון מונחה עצמים
  - סקרי קוד בהתאם
- מימוש מקובל של עקרונות:
  תבניות עיצוב (תיכון) Design Patterns
- עוד על Refactoring
- ~~קריאה~~

לסיכום

- מתי לתקן או לשפר? כשמגלים בעיה? כיצד להתגבר על קשיי ההתחלה?
- לפעמים, מותר גם לתכנן מראש...
- כתיבת בדיקות מראש לגילוי הצרכים

  יש טוענים ש-TDD עם mocks ושות' גורם לעמידה ב-SOLID אפילו כשהמפתחים לא מכירים את העקרונות
- מתפתחים כלים ברמות שונות בנושא איכות הקוד
- עקרונות נוספים במאמרים ובספרים של Martin ואחרים, למשל The Boy Scout Rule, The Law Of Demeter
  ויש מתנגדים, למשל:
- SOLID Fight: http://www.artima.com/weblogs/viewpost.jsp?thread=250296