Lean & Agile Software Development

Hands-on Practices, Principles, Agile Modeling, TDD, and more

Craig Larman

v.39

www.craiglarman.com
Copyright © 2013 Craig Larman, All rights reserved
May not be reproduced without written consent of the author.

- Please...
- Do not copy or share this material, or re-use for other education.
  - Exceptions require prior written consent of the author.

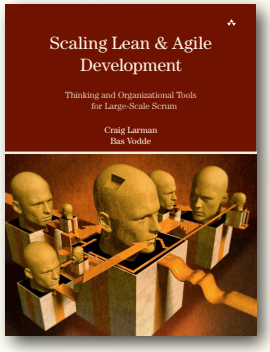
Copyright (c) 2013, Craig Larman. All rights reserved.

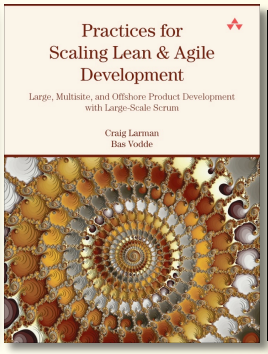
The Big Picture

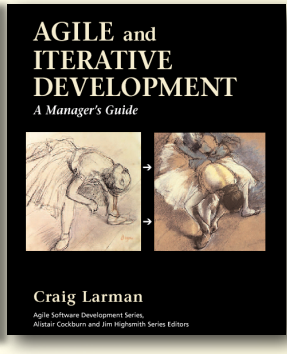
org agility

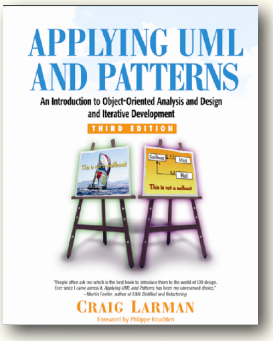
business agility

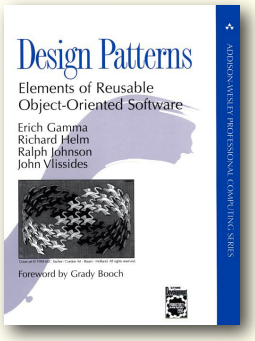
technical / design agility

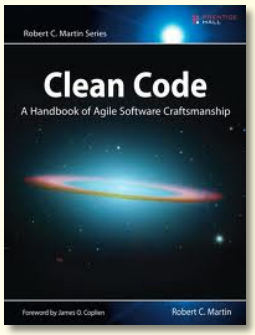












"Oh my god, i hope i don't have to touch that code!"

low technical agility ->
low business agility (and low velocity)

Overview

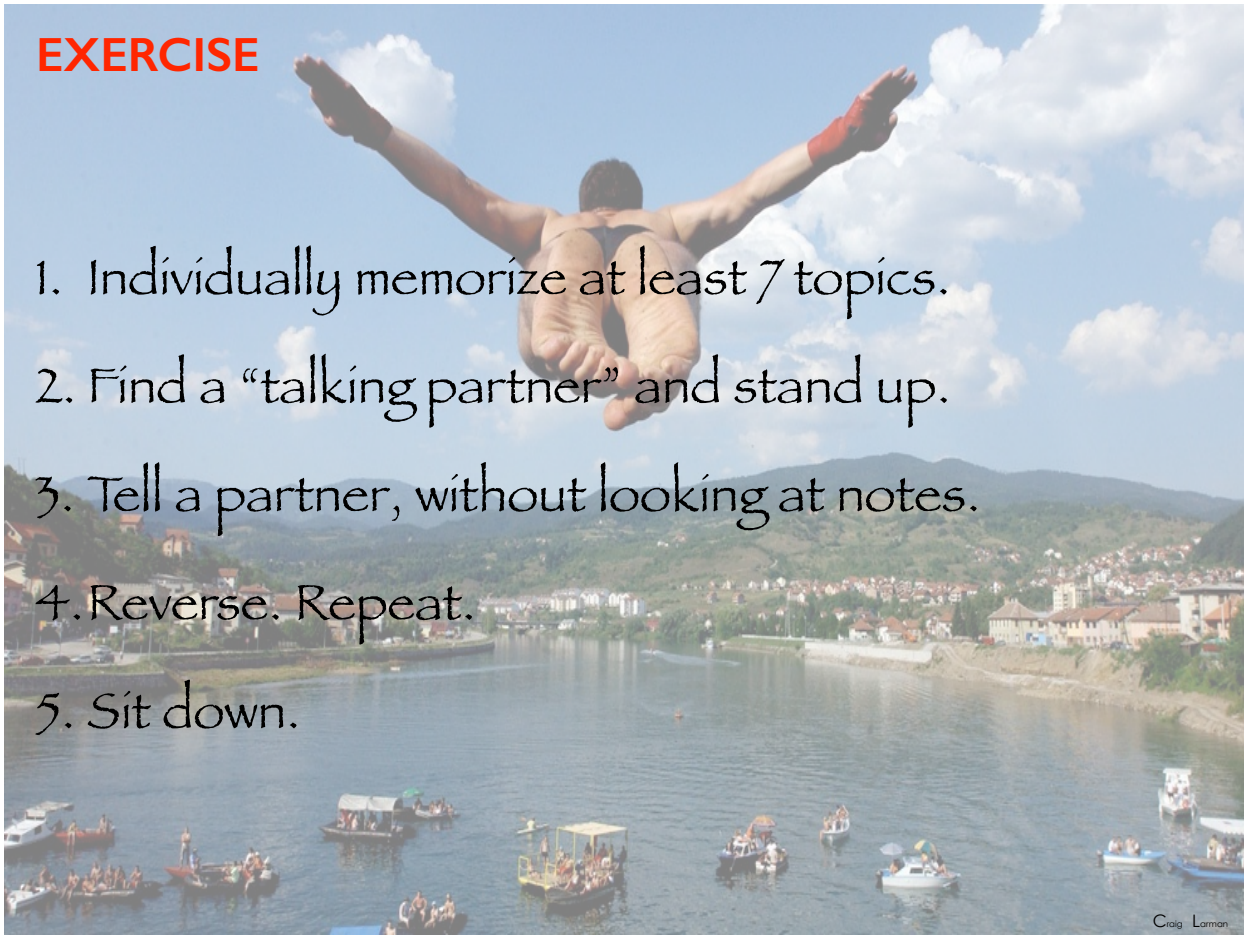
related...

- Hands-on TDD with Legacy Code
- Agile Modeling & Advanced Object Design with Patterns
- Agile Architecture Documentation Workshop
- ....

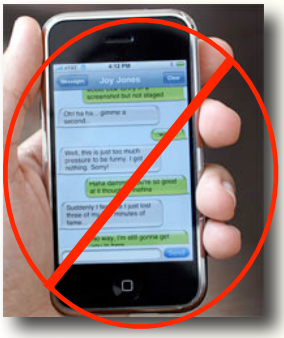
Topics

- Lean & Agile Modeling
- Lean & Agile Requirements
- User-Centered Design
- Story Mapping
- Telling Stories
- Splitting Features
- Learn with System-Sequence Diagrams
- Learn with a Domain Model
- Learn with Specification by Example
- Acceptance TDD
- BAD vs GOOD Automated Tests
- Designing with Layers
- Create Algorithms with Activity Diagrams
- GRASPing Object Design Principles
- Create Object Design with Communication Diagrams
- Create Object Design with Design Class Diagrams
- Continuous Integration
- Create Code Driven by Unit Tests (Unit TDD)
- Lean Thinking
- Lean Software Development
- Agile Values & Principles
- SOLID Principles
- Generalization
- Polymorphism & Class Hierarchy Design
- Feature Toggle Patterns
- Simple Patterns (Simple Factory, Null Object, …)

EXERCISE



1. Individually memorize at least 7 topics.
2. Find a "talking partner" and stand up.
3. Tell a partner, without looking at notes.
4. Reverse. Repeat.
5. Sit down.



Craig Larman

Lean & Agile Principles

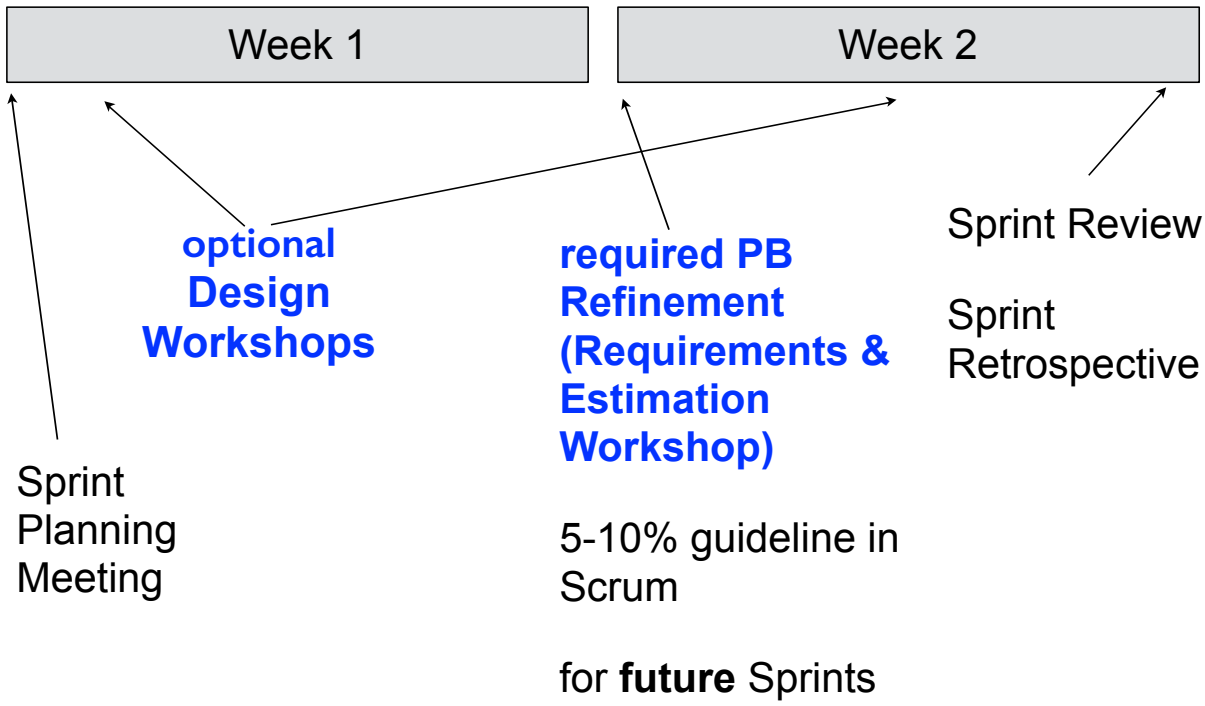
we will explore these conceptual "knowing" topics later…

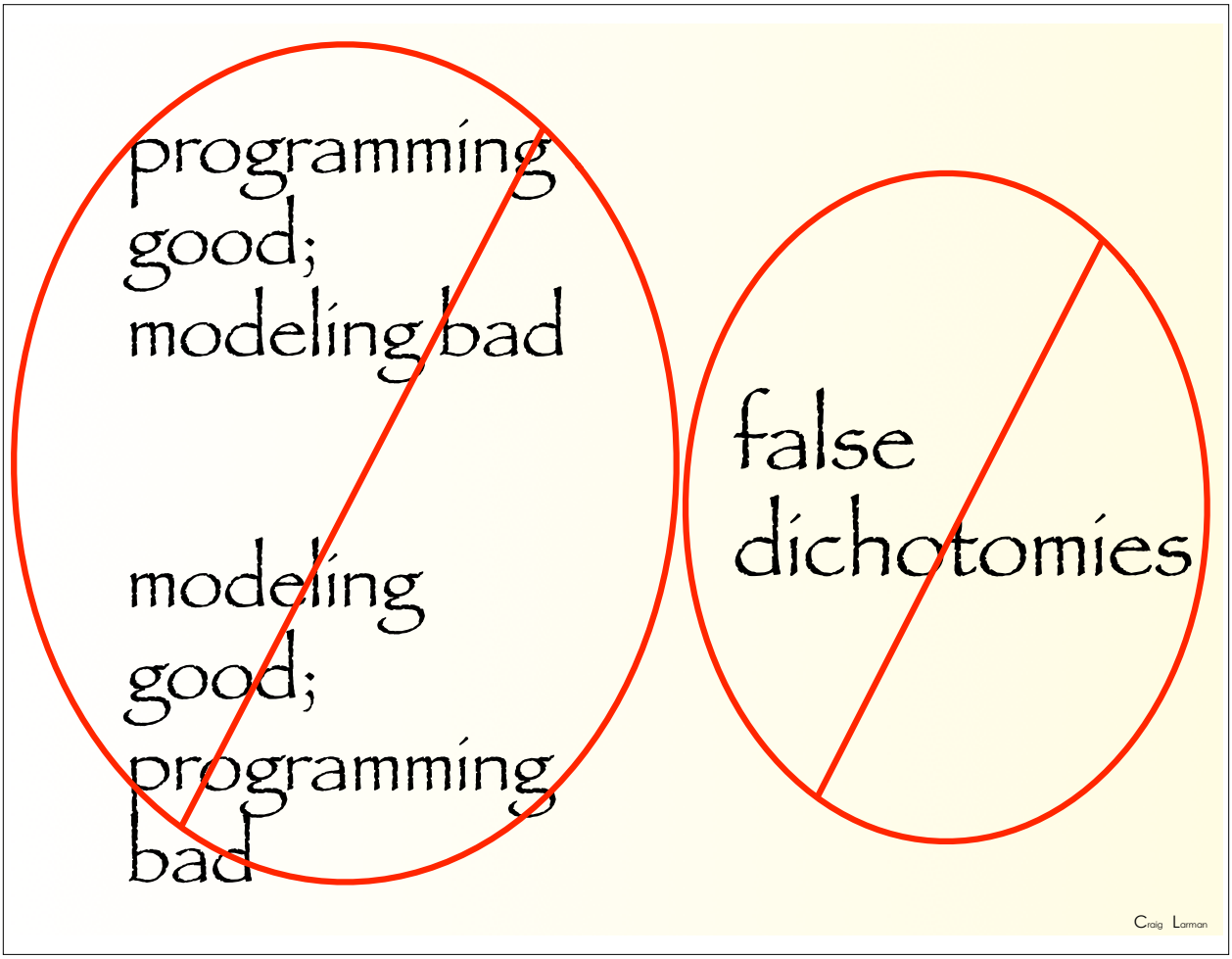
after first some days of "doing"

Lean & Agile Modeling

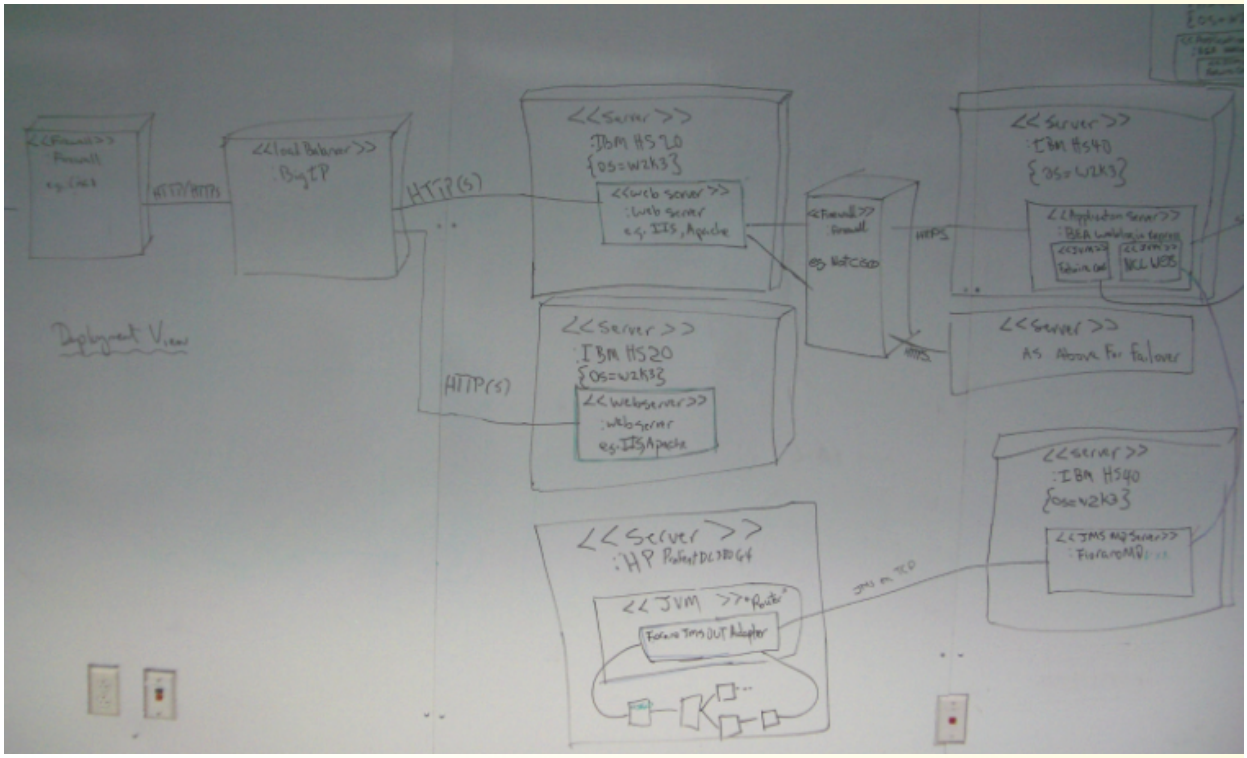
requirements workshops (e.g., for Scrum PBR)

design workshops

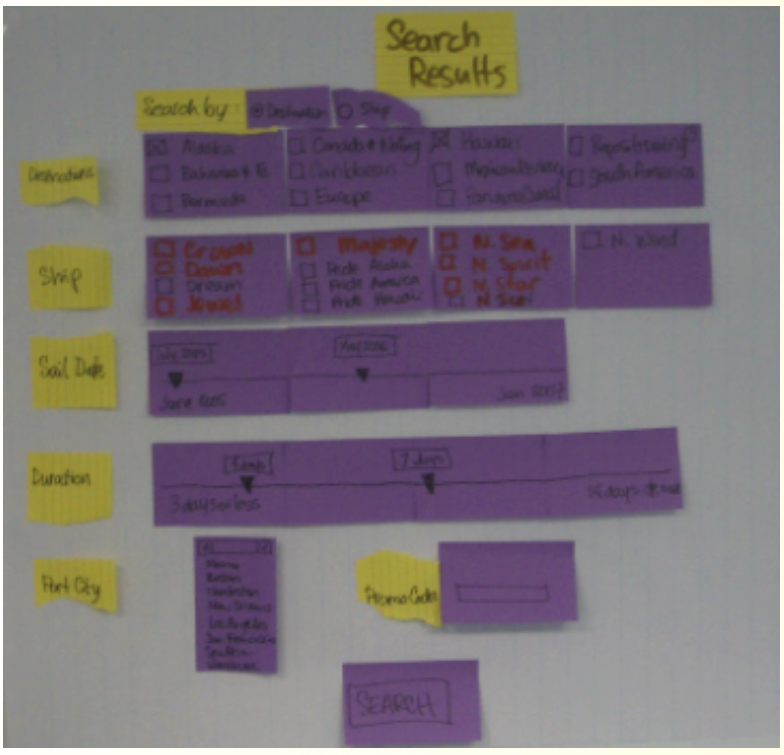

Week 1
Week 2
optional
Design
Workshops
required PB
Refinement
(Requirements &
Estimation
Workshop)
Sprint Review
Sprint
Retrospective
Sprint
Planning
Meeting
5-10% guideline in
Scrum
for future Sprints


programming
good;
modeling bad
modeling
good;
programming
bad
false
dichotomies
Craig Larman

simple tools for communication and collaboration



for low-fidelity UI modeling, too

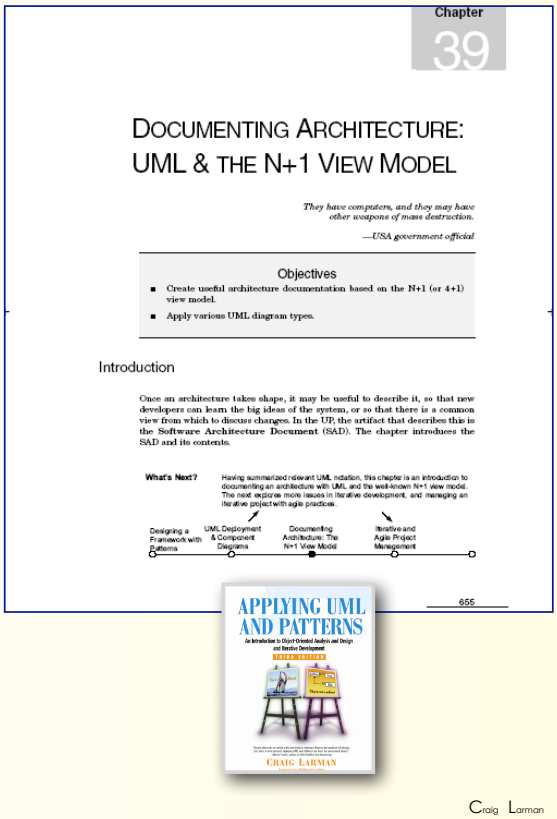


agile modeling guidelines

- Model in groups, not alone
- Multiple modeling teams, or team representatives, in a "Joint Design Workshop" if scaling
- Multiple models in parallel
  - e.g., class and interaction diagrams (static and dynamic views) on different walls, created in parallel

architectural documentation

- Usually, most useful AFTER the code is written and tested!
- Create in an agile documentation workshop with many participants
- Store in wiki?


Chapter
39

DOCUMENTING ARCHITECTURE: UML & THE N+1 VIEW MODEL

They have computers, and they may have other weapons of mass destruction.

—USA government official

Objectives

- Create useful architecture documentation based on the N+1 (or 4+1) view model.
- Apply various UML diagram types.

Introduction

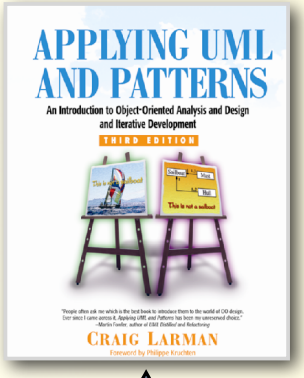
Once an architecture takes shape, it may be useful to describe it, so that new developers can learn the big ideas of the system, or so that there is a common view from which to discuss changes. In the UP, the artifact that describes this is the Software Architecture Document (SAD). The chapter introduces the SAD and its contents.

What's Next? Having summarized relevant UML notation, this chapter is an introduction to documenting an architecture with UML and the well-known N+1 view model. The next explores more issues in iterative development, and managing an iterative project with agile practices.

Designing a Framework with Patterns | UML Deployment & Component Diagrams | Documenting Architecture: The N+1 View Model | Iterative and Agile Project Management

655

agile documentation workshop:
N+1 View Model + Technical Memos

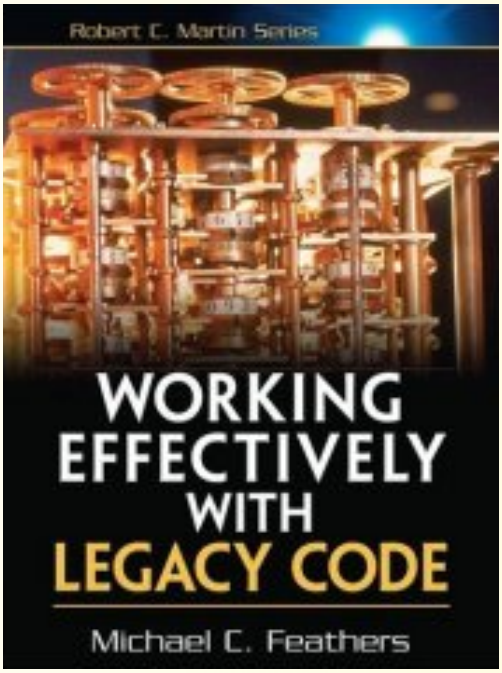


Chapter 39:
Documenting
Architecture

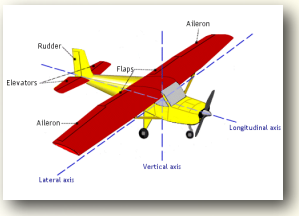
improving legacy code

= improving legacy design

= more agility



Models are not documentation!



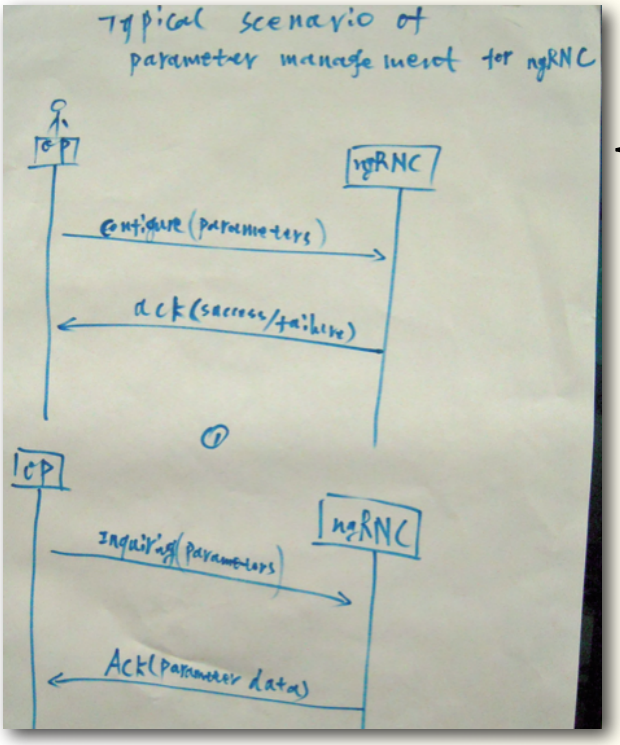
- We model to have a conversation – to group problem solve – and to develop shared understanding

Larman's Law of Modeling

we model to have a conversation

Every model is "wrong" – and that's OK

we will be sketching in the UML...



UML:

Just a diagramming notation standard (common language is useful)

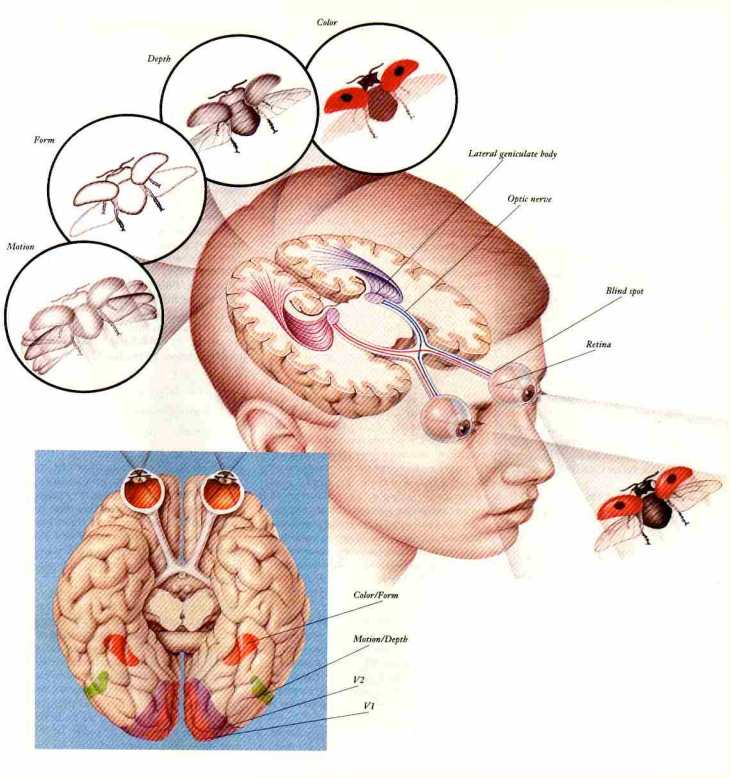
Trivial and relatively unimportant.

Not a method, process, or design guide.

what's important? diagramming notation?

- The point of the UML is often obscured in notational details and CASE tool sales pitches

- It's really very simple…



EXERCISE

1. standing with 1 "talking partner": without referring to notes, what are some implications of lean & agile modeling?

EXERCISE

1. set up the walls for agile modeling

Craig Larman

Lean & Agile Requirements?

EXERCISE

1. table discussion: what do you think "lean & agile requirements" implies?

Craig Larman

"we are doing traditional requirements, and need consulting help to do agile..."

"with Agile Consultant advice,
now we are doing agile! ..."

the analysis group writes use cases and sends them to the programming team,

"they are not agile"

the analysis group writes stories and sends them to the programming team,

"now, they are agile!"

writing "epics" & "stories" has NOTHING to do with being agile

changing your terminology or tool has

NOTHING

to do with being agile

agility ➠

lowering the cost of change, to compete on the ability to change quickly and cheaply

lean thinking ➠

(many, including...)

increase learning, "out-learn the competition", build quality in, eliminate waste

lean & agile requirements is a behavior, not a format or change of names

The 12 Agile Principles — & Agile Analysis

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most effective method of conveying information to and within a development team is face-to-face conversation.

The 12 Agile Principles — & Agile Analysis

7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self-organizing (self-managing) teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

doing lean & agile requirements implies working in ways that support agility, increases learning, and reduce the lean wastes

lean & agile requirements

≠

no requirements or "sloppy" requirements

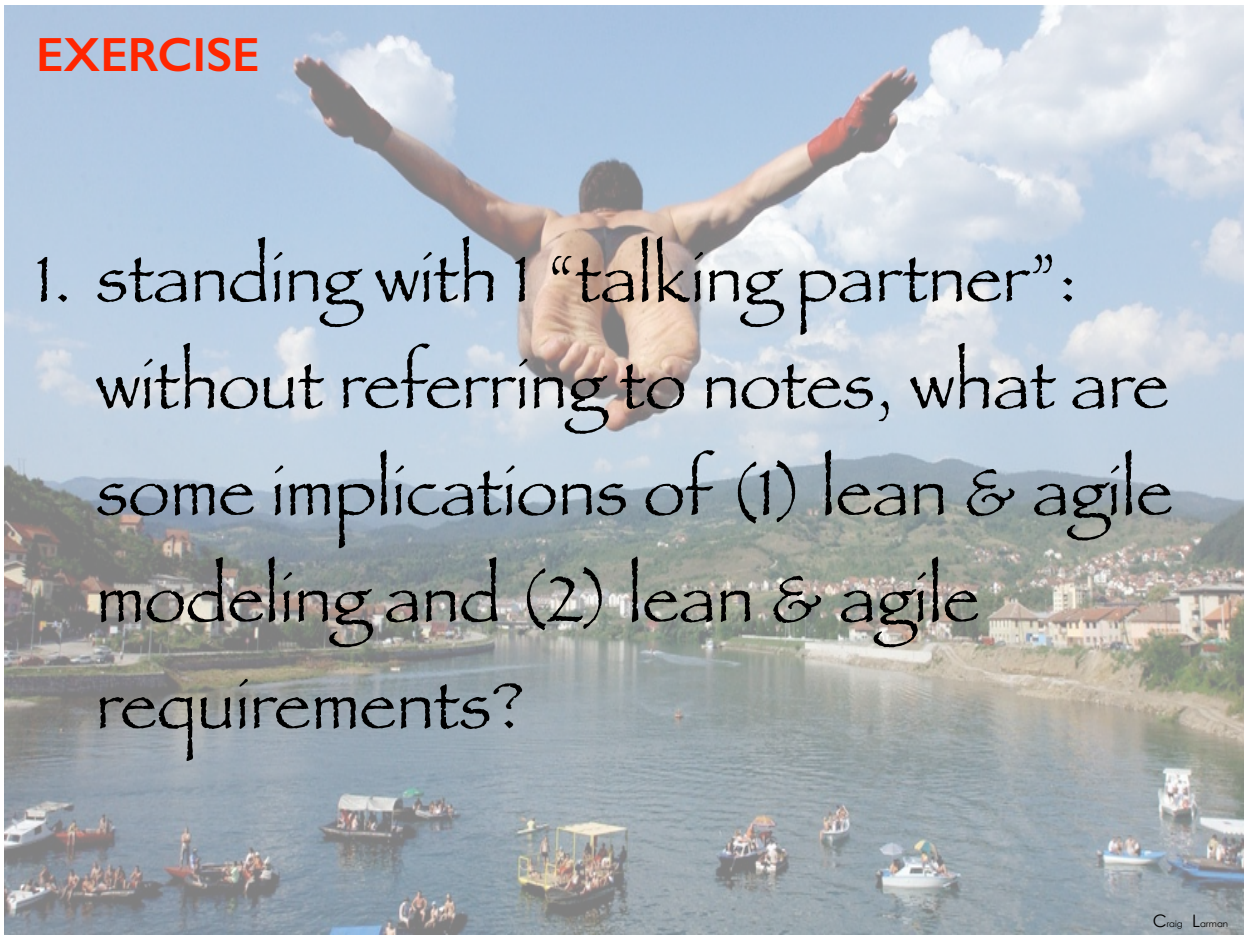
lean & agile requirements

≠

user stories

doing lean & agile requirements

- no separate analysis group (BAs, product mgrs) writing requirements
- no separate UX or UI design group
- no hand-off (no one outside the Team writes requirements)
- increased face-2-face (e.g., via video)
- multi-skilling (a person does analysis, programming, testing, ...)
- low WIP, especially for lower priority requirements
- simple, fast tools and representations
- less information and knowledge scatter/loss
- increase learning
- less "over-processing" and information loss by less transformations (e.g., english requirements -> english test cases -> automated test cases)

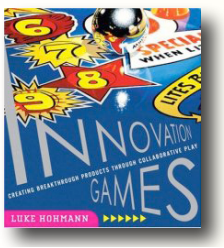


Forming a Vision

the details and practice are beyond the scope of this course, but there are learning resources for "lean & agile visioning..."

"agile" techniques for visioning

- Product Box
- Buy a Feature
- Speed Boat
- Spider Web
- Show & Tell
- Start Your Day
- Me & My Shadow
- Prune Product Tree
- Give Them a Hot Tub
- 20/20 Vision
- The Apprentice
- Remember the Future

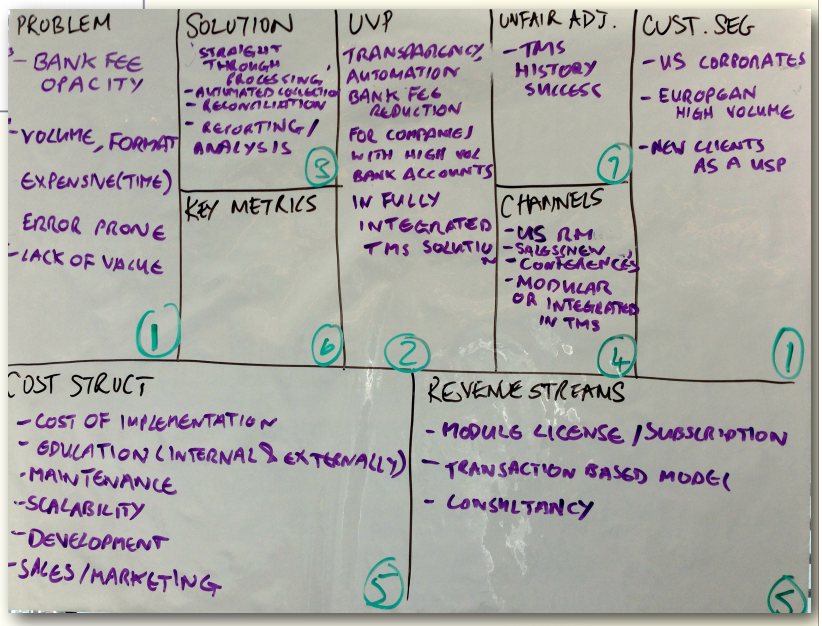


innovationgames.com

Lean Canvas

Problem	Solution	Unique Value Proposition	Unfair Advantage	Customer Segments
Top 3 problems 1	Top 3 features 3	Single, clear, compelling message that states why you are different and worth buying 2	Can't be easily copied or bought 7	Target customers 1
	Key Metrics Key activities you measure 6		Channels Path to customers 4	

Cost Structure	Revenue Streams
Customer Acquisition Costs Distribution Costs Hosting People, etc. 5	Revenue Model Life Time Value Revenue Gross Margin



User-Centered Design

(after this section, which frames a moderate body of ideas & practices, we will get into the "real work")

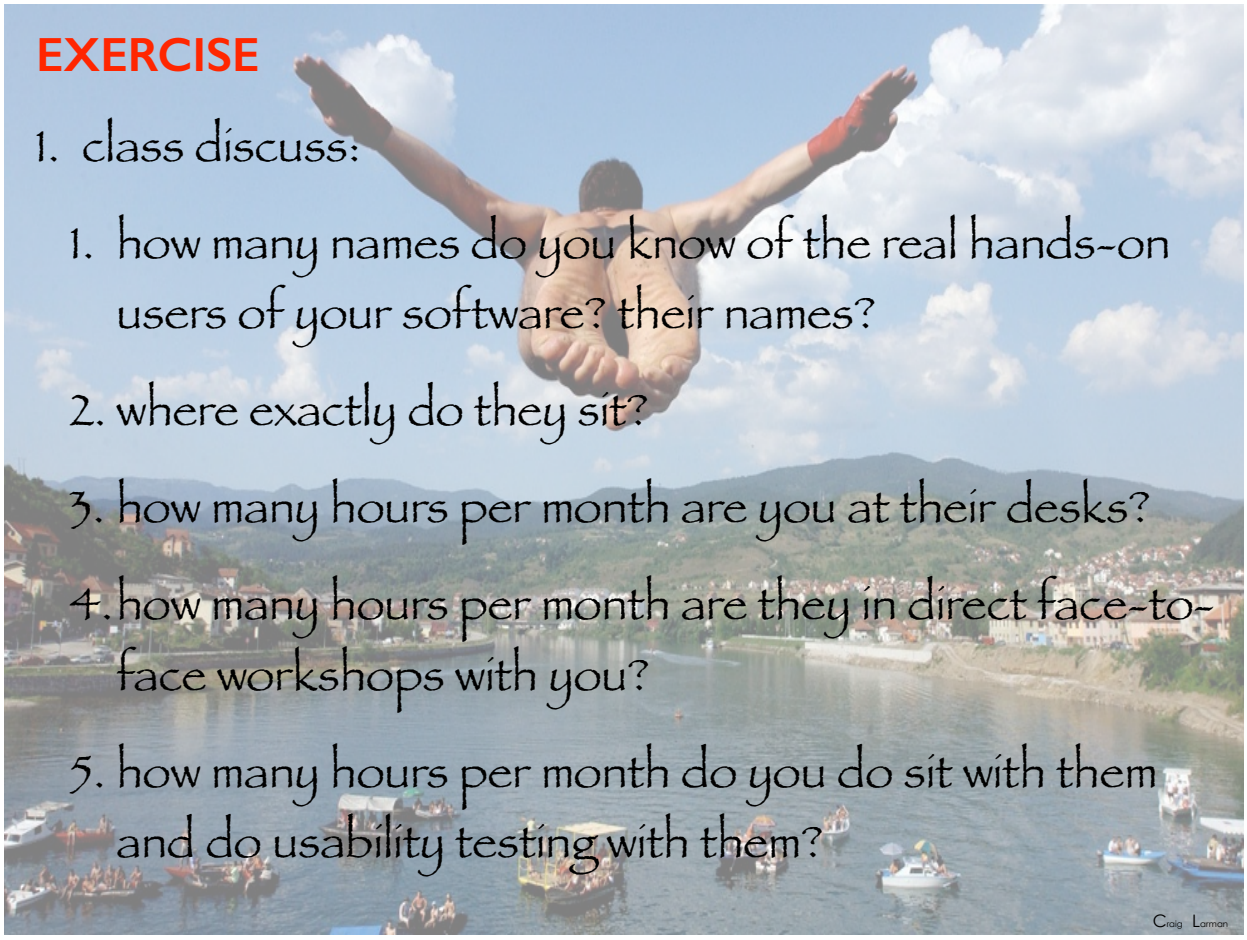
- user-centered design is a key culture & practice, for useful & usable products
- but… is a medium-sized body of concepts & techniques, largely beyond our scope
- therefore… we will only highlight & do a few key elements

beware doing user-centered design (UCD) and user experience (UX) design focusing on secondary minor techniques (personas, mockups, …)

… rather than primary UCD elements

if you don't first focus on primary UCD elements, doing secondary techniques is largely a waste, and can give the false impression that there is UCD

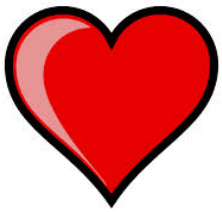
Fake UCD



EXERCISE

1. class discuss:
   1. how many names do you know of the real hands-on users of your software? their names?
   2. where exactly do they sit?
   3. how many hours per month are you at their desks?
   4. how many hours per month are they in direct face-to-face workshops with you?
   5. how many hours per month do you do sit with them and do usability testing with them?

the heart of it…



Developers regularly face to face with REAL USERS

is more important than

UX and UI modeling techniques

empathy

Fundamental Attribution Error

the inclination to explain people's behavior by the way they are rather than the situation they are in

Developers need to really know the user situation, so Developers do…

field studies (real users)

Developers exploring requirements & UIs with hands-on users

(not proxies, not business analysts, …)

Developers discovering, in face-2-face conversations with hands-on users, the *real* problems they are trying to solve

a bias to action rather than analysis — implement and deliver software in short cycles, to get feedback from real users

feedback to Developers about the running software with hands-on users

Developers seek early & honest criticism from hands-on users & others about the user design

(be open to criticism & don't delay it)

usability testing with hands-on users

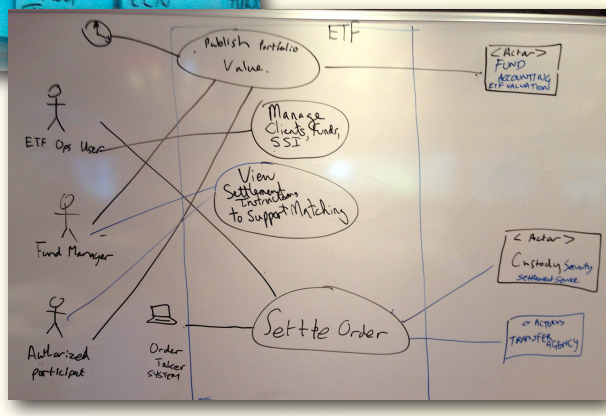
"3"

perhaps one could identify other "major" UCD ideas & practices, but if the prior basic key elements are not in place, most else is "rearranging the deck chairs on the Titanic"

2

secondarily? ...

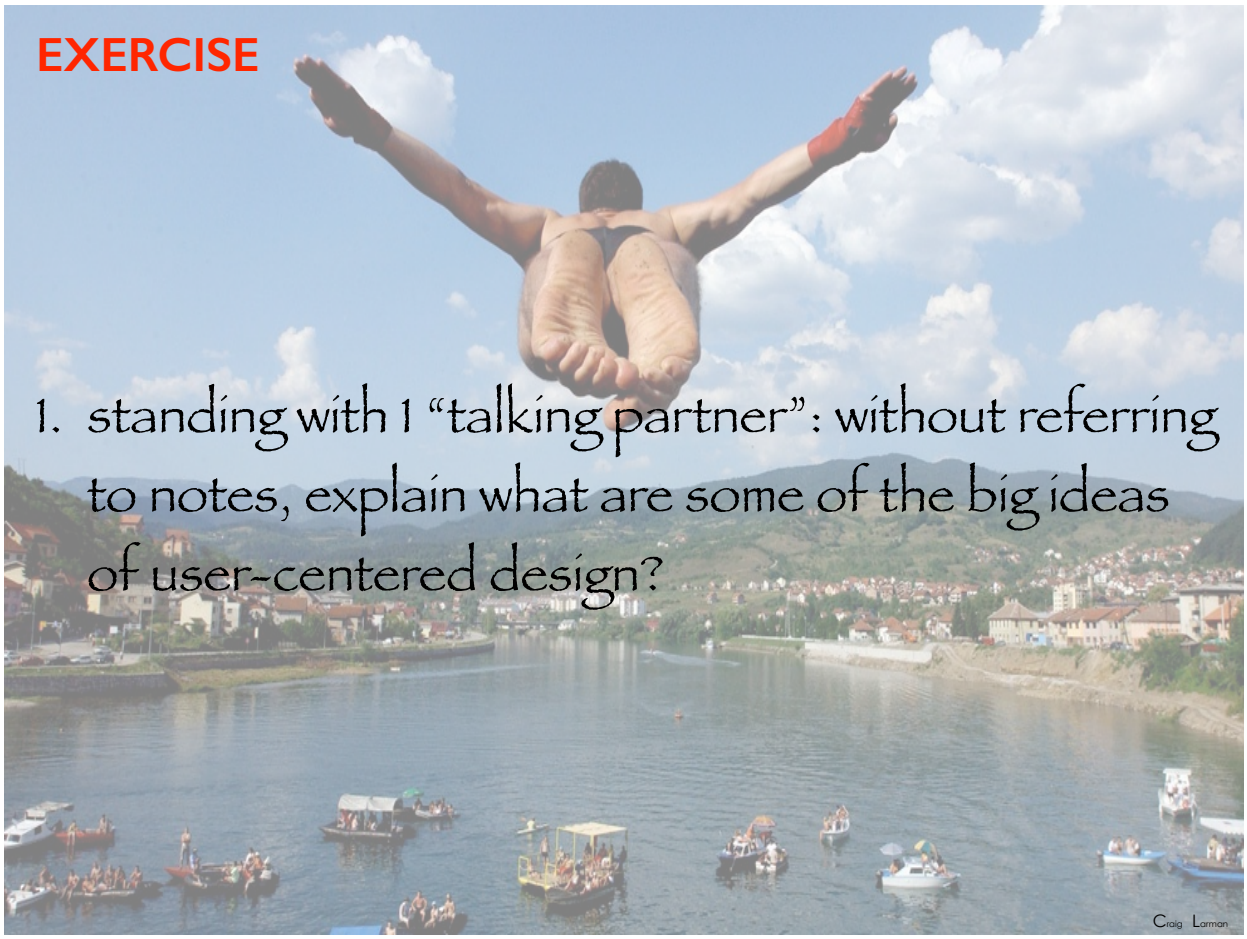
user task analysis



current and/or future state

how might it change when there is software?

(and more, later)

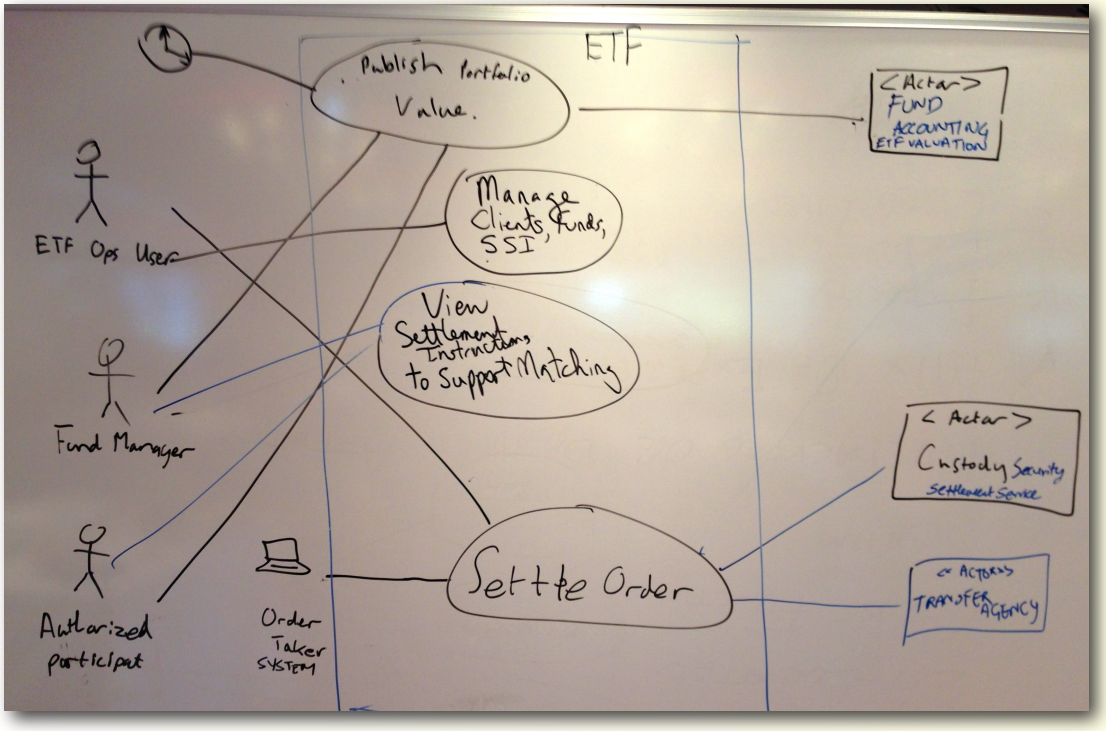


Identifying Users, Goals, & Tasks with a Use Case Diagram

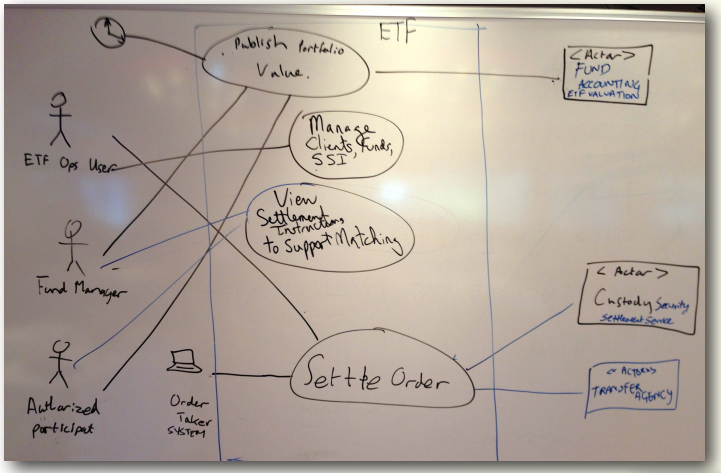
(notice that this model is very user-centric)

- a technique in user-centered design for user-task analysis
- what actors (especially, *human* users) use the system, and what are their goals & tasks?

EXAMPLE: use case diagram



agile modeling — of course, at a wall with USERs



DEFINITION: actors

- the system is used by primary actors (users)
  - e.g., Reconciliation Clerk
  - External agents (human or computer)
- Use supporting actors
  - CreditAuthorizationSystem

GUIDELINE: testing for good use cases

- elementary business process?
- relatively big, many steps?
  - a use case is NOT (usually) one step; it is a complete set of end-to-end steps

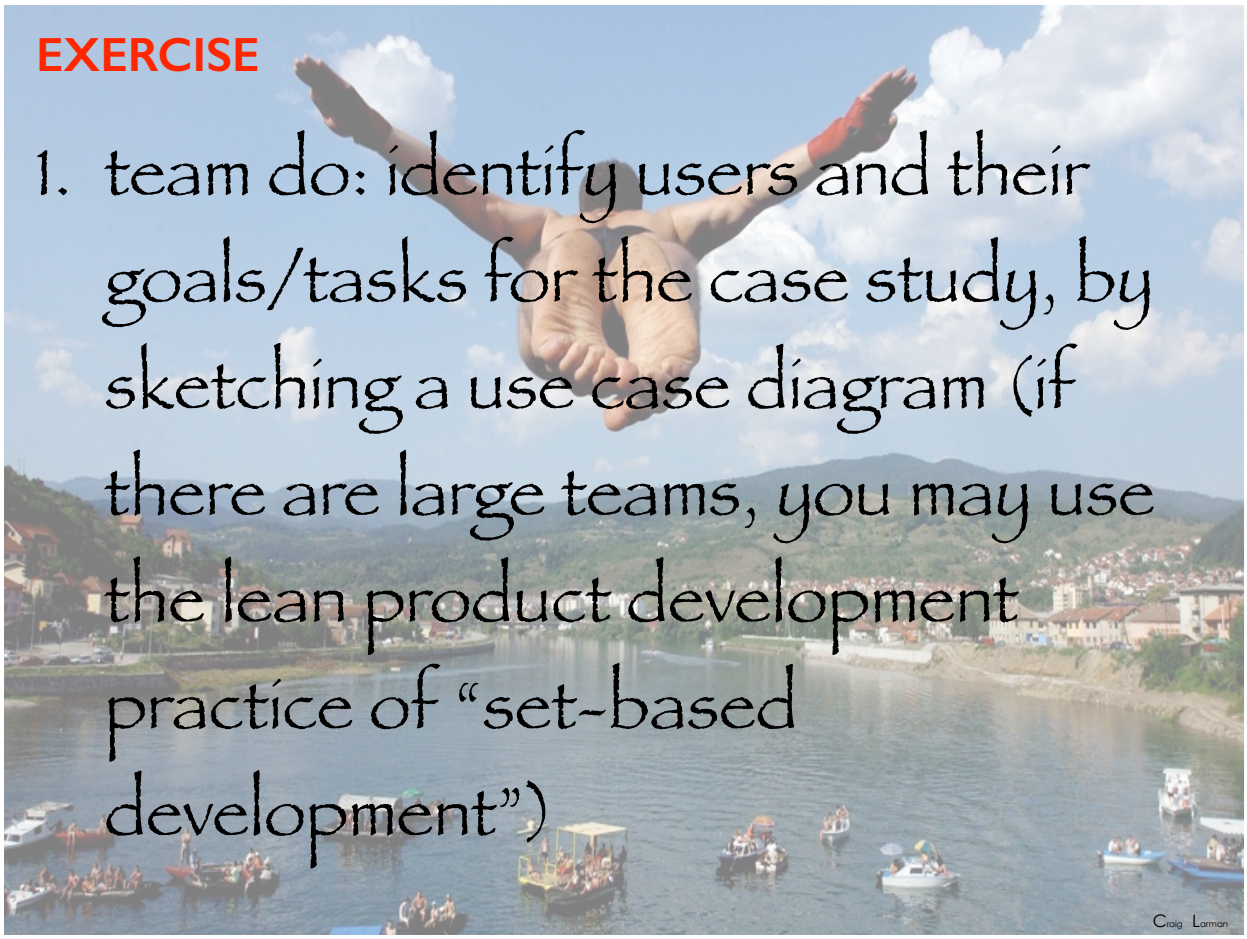
GUIDELINE: identifying & naming use cases

- many systems have Start Up, Shut Down, Recover, and Administer System use cases
- If small CRUD operations, group into one use case... Manage Network Addresses
- If large CRUD operations, separate into different use cases ... Create Network Addresses, ...
- starts with verb... Print Job
- "keep the UI out" of the name
  - Delete Job versus "Press Delete Key to Delete Print Job" ... Consider the 100 Year Guideline

GUIDELINE: names should reflect goals

- good use case names usually reflect user goals...
  - Manage Network Addresses
  - Predict Subsurface Well Flow
  - Print PDF document with variable data

EXERCISE

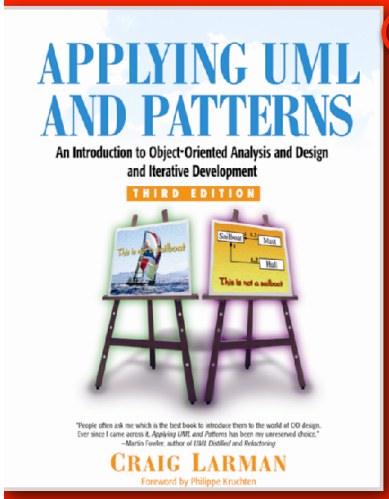


1. team do: identify users and their goals/tasks for the case study, by sketching a use case diagram (if there are large teams, you may use the lean product development practice of "set-based development")

Craig Larman

a use-case diagram is a model!

Craig Larman



1 Object-Oriented Analysis and Design 3
2 Iterative, Evolutionary, and Agile 17
3 Case Studies 41
PART II INCEPTION
4 Inception is Not the Requirements Phase 47
5 Evolutionary Requirements 53
6 Use Cases 61
7 Other Requirements 101
PART III ELABORATION ITERATION 1 — BASICS
8 Iteration 1—Basics 123
9 Domain Models 131
10 System Sequence Diagrams 173
11 Operation Contracts 181
12 Requirements to Design—Iteratively 195
13 Logical Architecture and UML Package Diagrams 197
14 On to Object Design 213
15 UML Interaction Diagrams 221
16 UML Class Diagrams 249
17 GRASP: Designing Objects with Responsibilities 271
18 Object Design Examples with GRASP 321
19 Designing for Visibility 363
20 Mapping Designs to Code 369
21 Test-Driven Development and Refactoring 385
22 UML Tools and UML as Blueprint 395
PART IV ELABORATION ITERATION 2 — MORE PATTERNS
23 Iteration 2—More Patterns 401
24 Quick Analysis Update 407
25 GRASP: More Objects with Responsibilities 413

Identifying Users, Goals, & Tasks with a Story Mapping Backbone

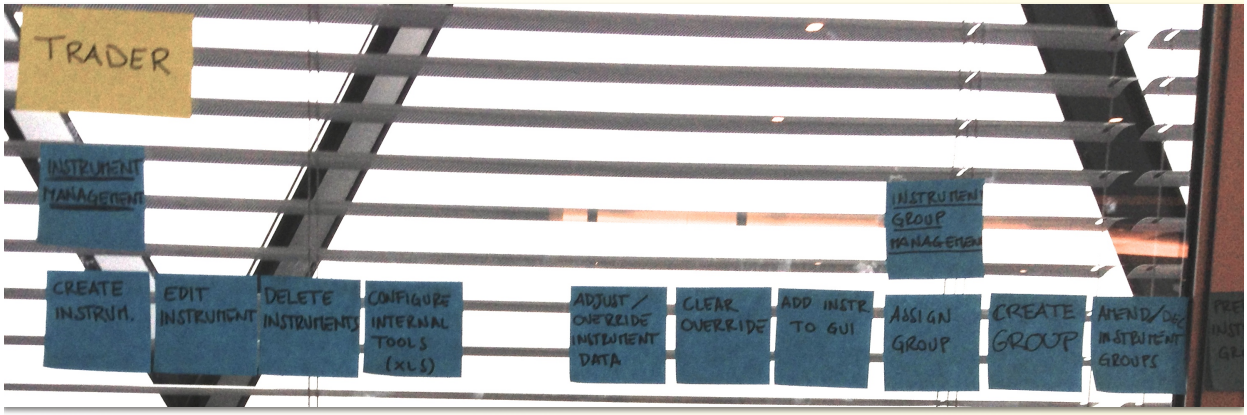
(notice that this model is very user-centric)

story mapping backbone

- a technique in user-centered design for user-task analysis
- what actors (especially, *human* users) use the system, and what are their goals & tasks?

story map backbone

story map backbone

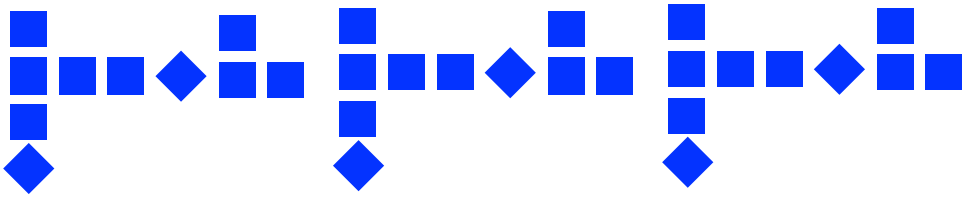


some or all of the major user tasks should be identifiable from a prior use-case diagram

step 1: user task analysis

backbone (user task analysis)

backbone by...
(1) actors (users), (2) composite task groupings, (3) tasks



left to right: typical time sequence

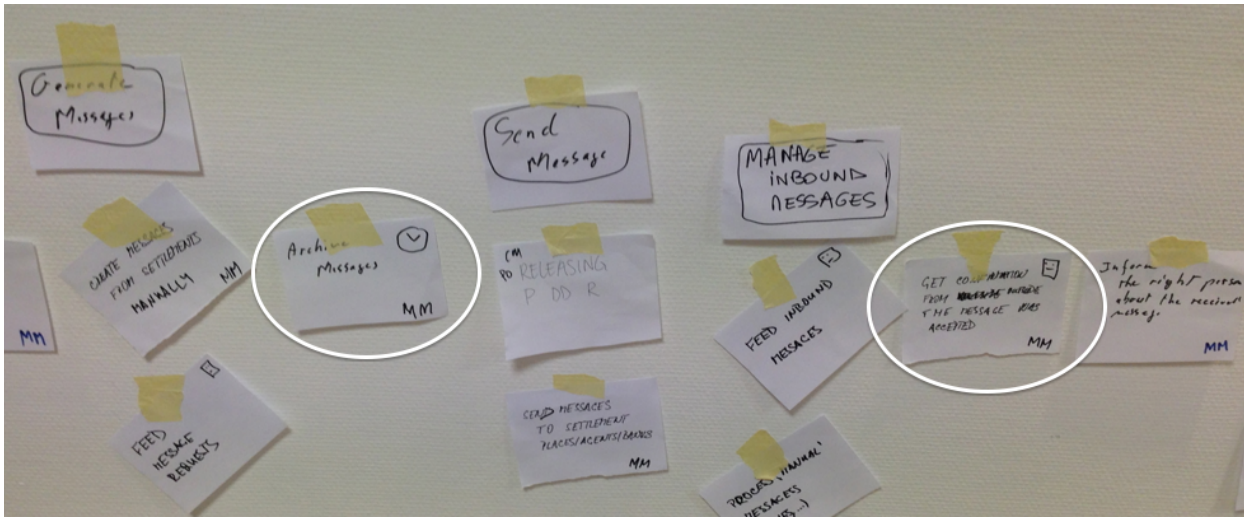
vertical -> variations

skeleton (stories)

vertical -> optionality

in some domains, noteworthy “tasks” are not initiated by humans, but by external “robots” (e.g., an external trading system) or time events (scheduled report generation)

in addition to human-task analysis, consider adding robot & time-driven “tasks” to the backbone (since many features may be mapped to these)



do story mapping with users and Developers

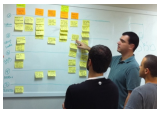
EXERCISE

- team do: at a wall, create a story map backbone

a story map is a model!

Story Map Concepts

Use simple maps to visualize the stories you tell about your software



Users

A map tells a story about a type of person doing something to reach a goal. Make sure to include them in your map along with a little information about them.

Try using lightweight persona sketches to describe your users.

User Tasks

User's tasks are short verb phrases that are the basic building block of a map. If I ask you what you did earlier today when using email, you'll likely respond with tasks like:

- Read an email message
- Respond to a message
- Mark a message as spam

Goal-Level

The actions that users take in order to reach their larger goals have a goal level themselves that's tied to user behavior.

Summary: lists of tasks done in support of a bigger goal.

Functional: I'd expect to complete this task before taking a break.

Sub-Functional: smaller things done in support of a bigger tasks.

As you read across tasks in the backbone, check to make sure that tasks are of a similar goal level.

Activities

Activities organize tasks done by similar people at similar times to reach a goal. For your email software activities might include

- Going through my inbox
- Configuring my email client
- Organizing messages into folders

Backbone

Activities and tasks at a higher goal level give the story map it's structure. The backbone is arranged in a narrative flow. Smaller sub-tasks, details and variations hang down to form the ribs connected to the backbone.

Narrative Flow

The left to right axis in a story map is organized in the order you'd tell the story about your user to someone else.

Of course any specific user might choose to do different things in a different order. Use conversation to explain the details and variations.

If you're looking for the precision of a workflow model, flow chart, or UML model, then a story map isn't your best choice.

A story map will take lots of conversation to use effectively. But then that's the purpose of stories.

Release Slice

Use a tape line to identify slices of tasks that users might use your software for to reach their goals. The smallest number of tasks that allow your specific target users to reach their goal compose a viable product release.

Use release slices to identify small experiments, minimal viable product releases, or a "walking skeleton" version of your product.

Identify the target outcomes of your slice in a sticky note or card to the left of the slice.

User Tasks make great story titles!

Write short verb phrases on cards or stickers. Use them next to your story titles. If you use the story template to write descriptions, the task fits nicely right after "I want to..." The activity fits right after "so that..."

Details, Details...

Break down high goal level tasks into:

- Sub-tasks
- Alternative tasks
- Exceptions
- Details

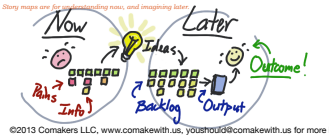
Down in the details of the map, it's OK to include details about what UI might look like or what the system might do in the background.

Map Now & Later

Use a map to describe the world as it is today. Inject pains, joys and rewards, details and observations, and solution ideas.

Evolve the map using design and discovery to describe the behavior you expect users to have in the future.

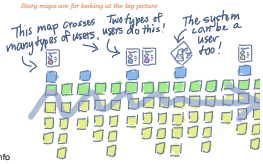
Story maps are for understanding now, and imagining later.



Map the Whole System

Map a whole process as it crosses through a number of types of users

Story maps are for looking at the big picture



©2013 Comakers LLC, www.comakewith.us, youshould@comakewith.us for more info

Story Map Process

The story map evolves with your understanding of your users and your product solution

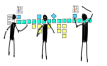
1 Frame

Before mapping, create a short product or feature brief to frame and constrain what you map. Think of this as the big story.

What names the product, feature to add to a product, or problem you'd like to solve.

Who names the different types of users who will use the product, and the "chosen" or customers who will buy it. For each user and chooser state the benefit they get from using the product.

Why describes the benefit your organization gets by building the product or feature. Say what users do and how their use leads to increased revenue or reduced costs.



2 Map the Big Picture

Focus on getting the whole story. Think "mile-wide, inch-deep." The activities and high-level user tasks that tell the whole story form the backbone of your story map.

Start with the user type most critical to your product's success. Imagine a typical day in your user's life with your new product. Map the steps they take as user tasks left to right.

Identify user activities – groups of tasks that work together to support a common goal. Activities often emerge after you see more of the story.

Add in additional users. As you follow the typical use of your product, you may find other types of users enter your story. Continue modeling their story left to right.

3 Explore

Fill the body of your story map by breaking down larger user tasks into smaller subtasks and user interface details. During this phase you'll add cards, split one card into two, rewrite cards, and reorganize them.

Use this phase to think "blue sky" about all the great possibilities. Use this time to think of everything that could go wrong. Don't worry if your ideas are "in or out of scope." You'll deliberately move things out of scope later.

- **Play "wouldn't it be cool if..."** to help think of great product ideas.
- **Look for variations:** What else might users of the system have done?
- **Look for exceptions:** What could go wrong, and what would the user have to do to recover?
- **Consider other users:** What might other types of users do to reach their goals?
- **Add in other product details** like: Description of proposed UI; Business rules; Data elements

Involve others. Tell your product's story to others that understand users and use. They'll find holes in your story and help build it up. Tell your product's story to software developers. They'll point out risky or expensive areas, and add in great technology solutions.

4 Slice Out Viable Releases

Slice your map into holistic product releases that span the users and use of the product. These slices form an incremental product release roadmap where each release is a minimal viable product release.

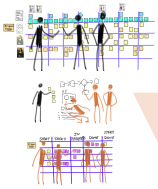
For each release name the target outcomes and impact. Outcomes and impact says how this release contributes to the overall goal in the "big why" that motivates building the product, and how users will behave in a way that helps us reach the goal.

For each release, identify product success metrics. Answer the question: "what would we measure to determine if this product was successful?" Ideally you'll find specific changes in user's behavior as they use the product the way your story map imagines.

5 Slice Out a Development Strategy

Slice the first release of your map into three or more delivery phases that allow you and your team to learn fast and avoid risk. Think of the opening, mid, and end-game phases of a chess game.

This development strategy will help you release the best product possible in the time constraints you have.



- **Opening Game builds a "functional walking skeleton"** – the simplest possible functional version of the product. As you finish "Opening game" vet the product with users and other stakeholders. Begin validating performance and scalability.
- **Mid Game complete all major functionality** and makes existing functionality richer and more complete. Continue user testing and leverage feedback to adjust the product. Continue testing performance and scalability.
- **End Game refines the product** in preparation for release. Continuously assess release readiness based on your release level product goals. Count on unforeseen work to emerge during this last stretch of development.

Plan the work necessary to refine stories.

Workshop stories with developers and testers to work through details and agree on acceptance criteria.

Plan development and testing.

Build and verify parts of working software.

Feature-Based Evolutionary Sprints

Feature != subsystem or component or algorithm step

~

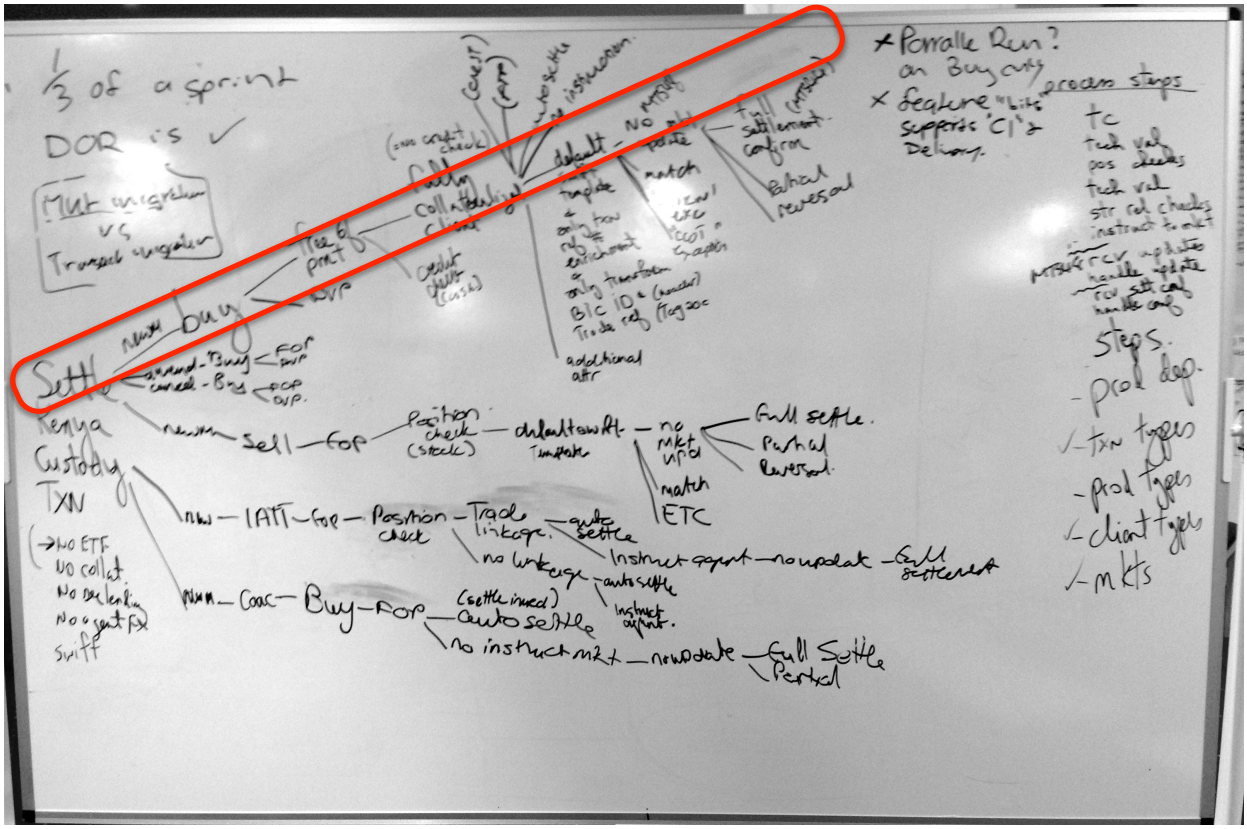
Feature = customer-centric "vertical" end-2-end functionality

Do features in Sprints...

Includes all cross-component and cross-functional work: systems engineering, analysis, architecture, interaction design, engineering, testing, documenting...

vertical slices, not horizontal components/ frameworks/algorithm steps/…

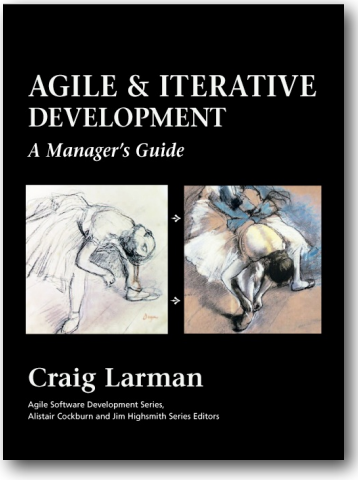
AKA: use-case driven, feature-driven



vertical slices, not "parts"

iterative & evolutionary, not "build components"

example from: Jeff Patton



Identifying & Mapping Feature Solutions with a Story Mapping Skeleton

(notice that this model is very user-centric)

(we are taking our first step at proposing feature solutions)

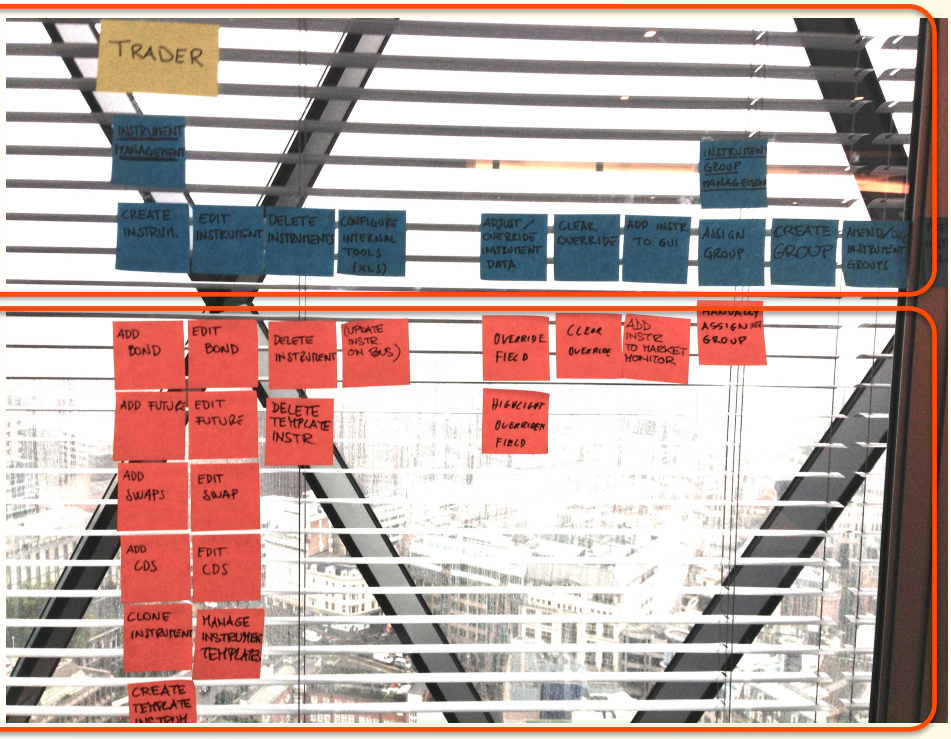
a complete story map includes a "skeleton" below the "backbone" that maps software features to user tasks

BACKBONE

users & their tasks

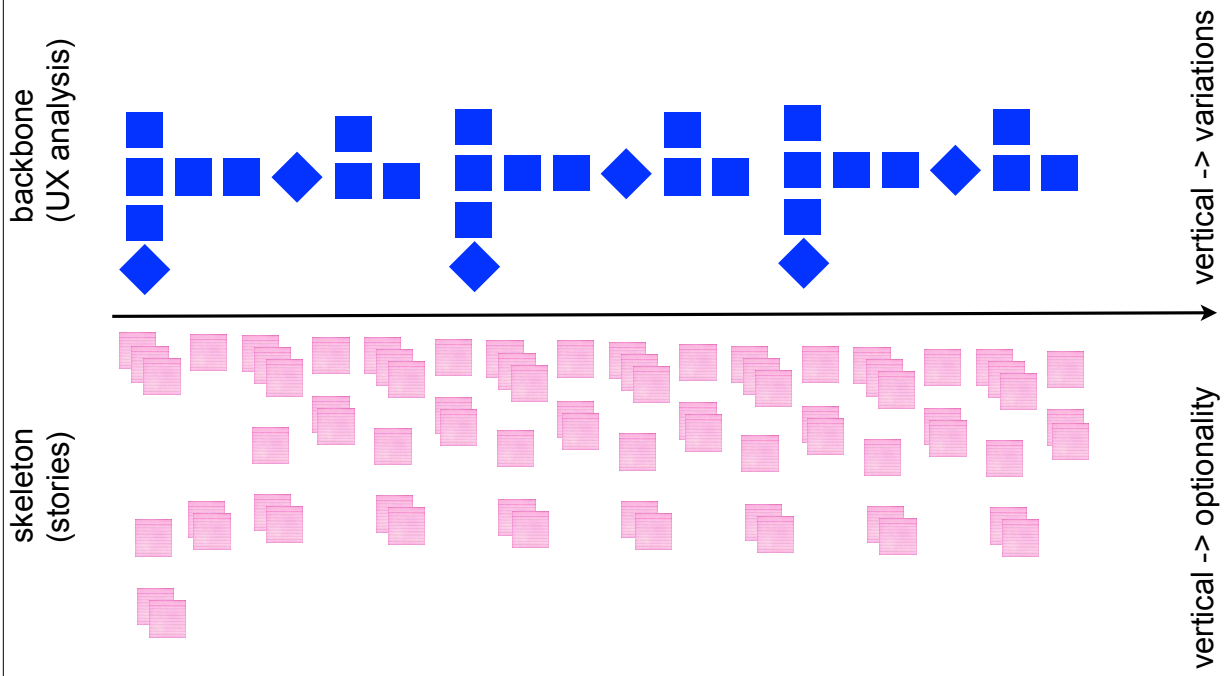
SKELETON

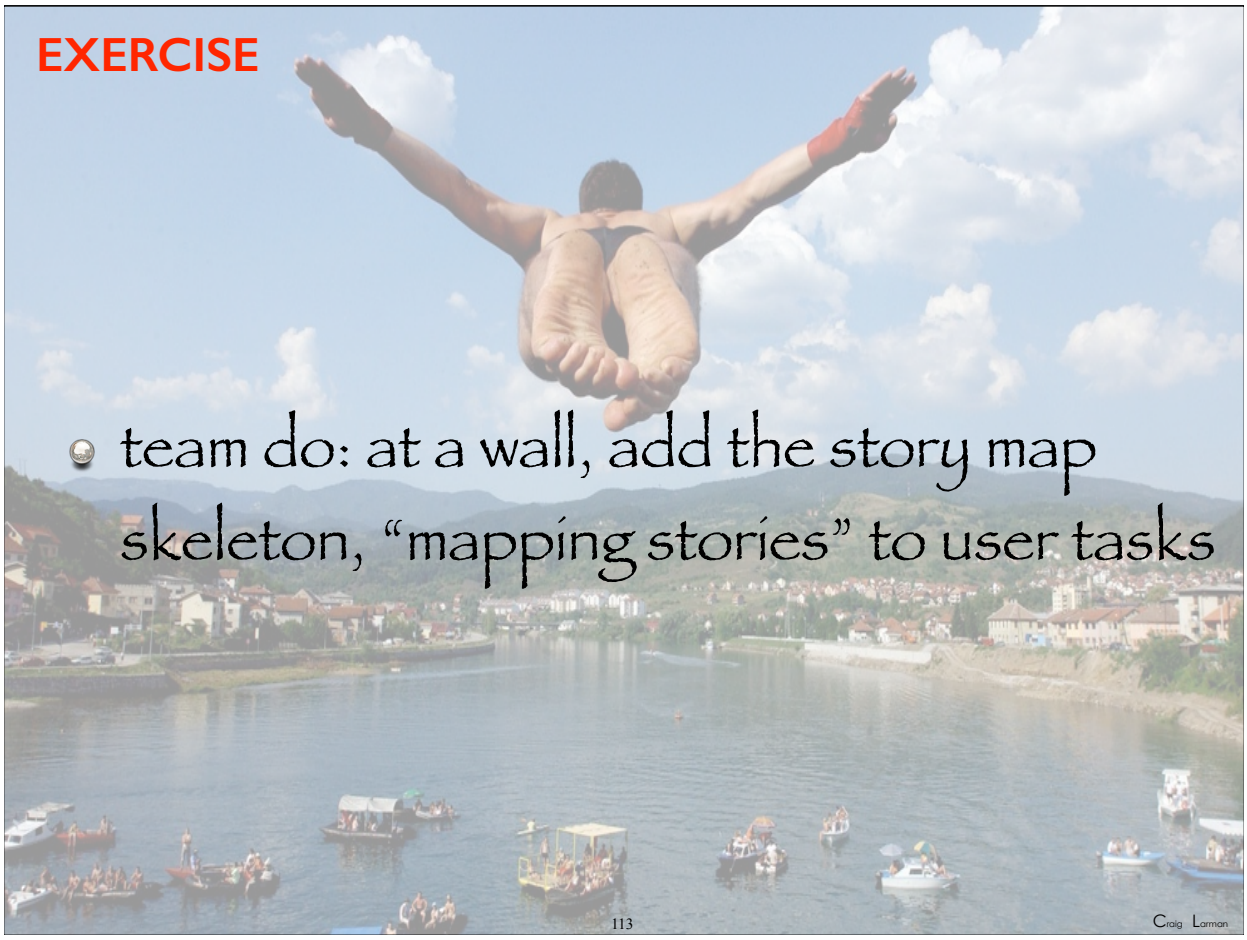
feature "stories" mapped to user tasks



Craig Larman

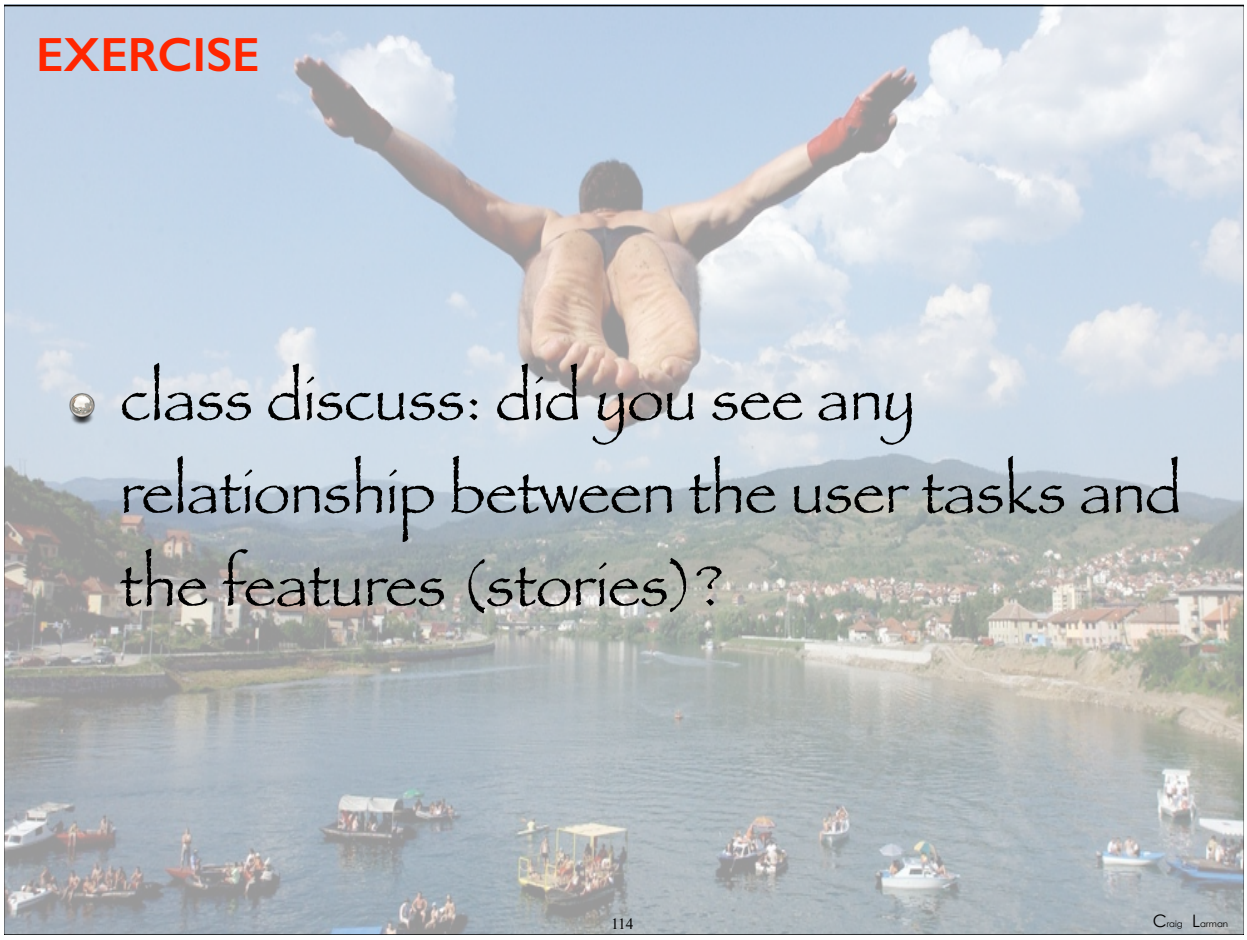
step 2: design of *hypothesized* solutions





EXERCISE

- team do: at a wall, add the story map skeleton, "mapping stories" to user tasks

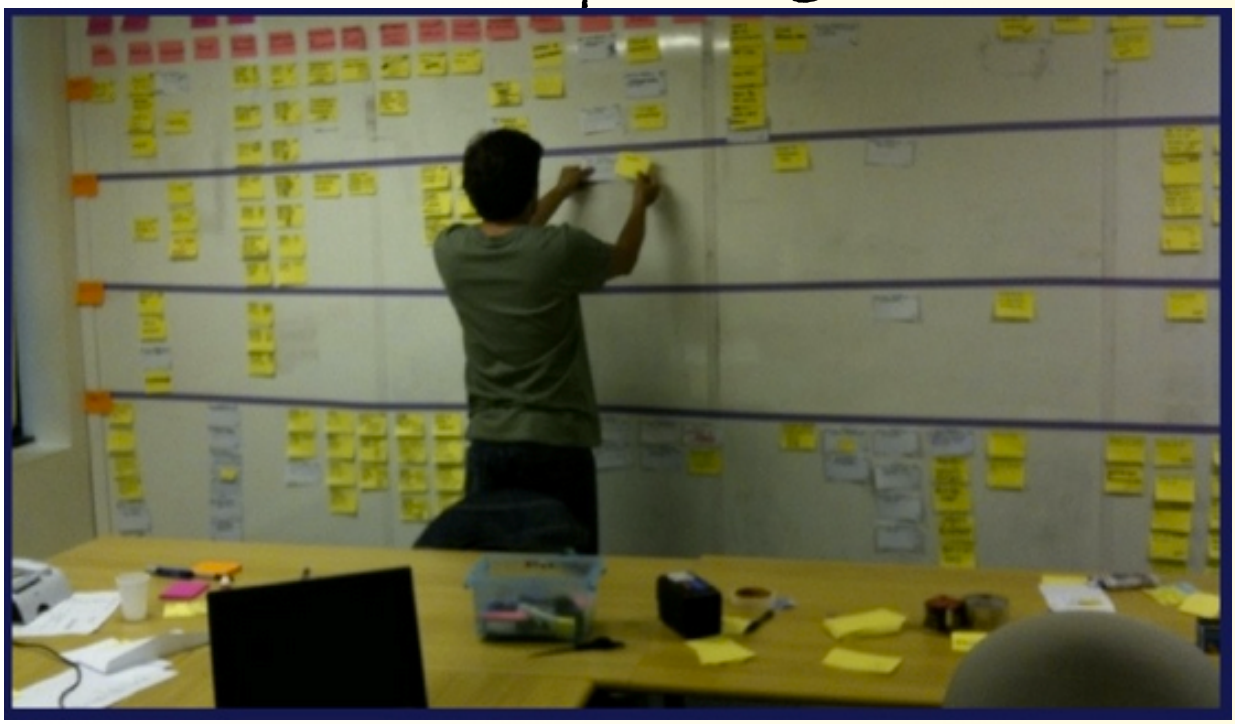


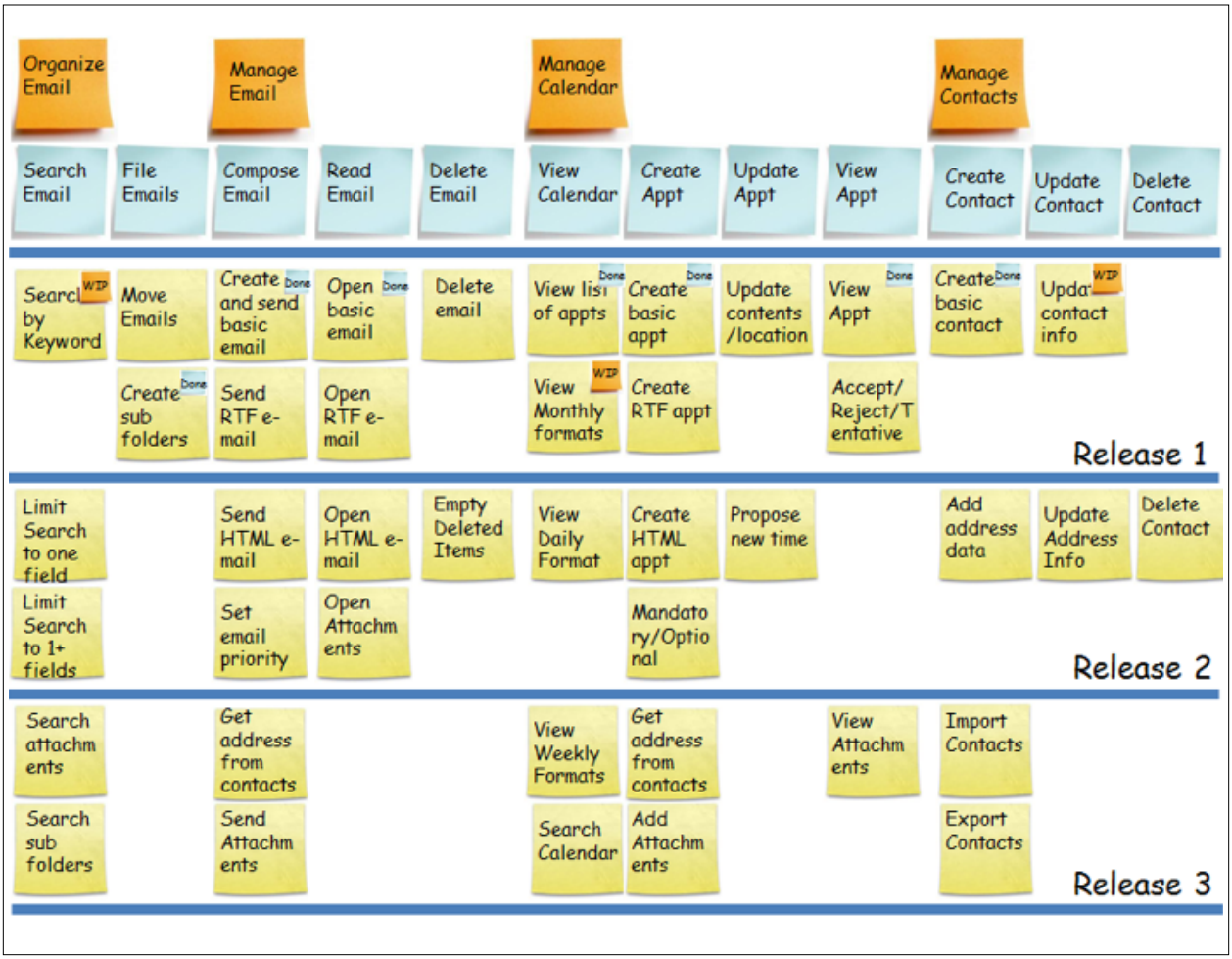
EXERCISE

- class discuss: did you see any relationship between the user tasks and the features (stories)?

Release Planning with a Story Map

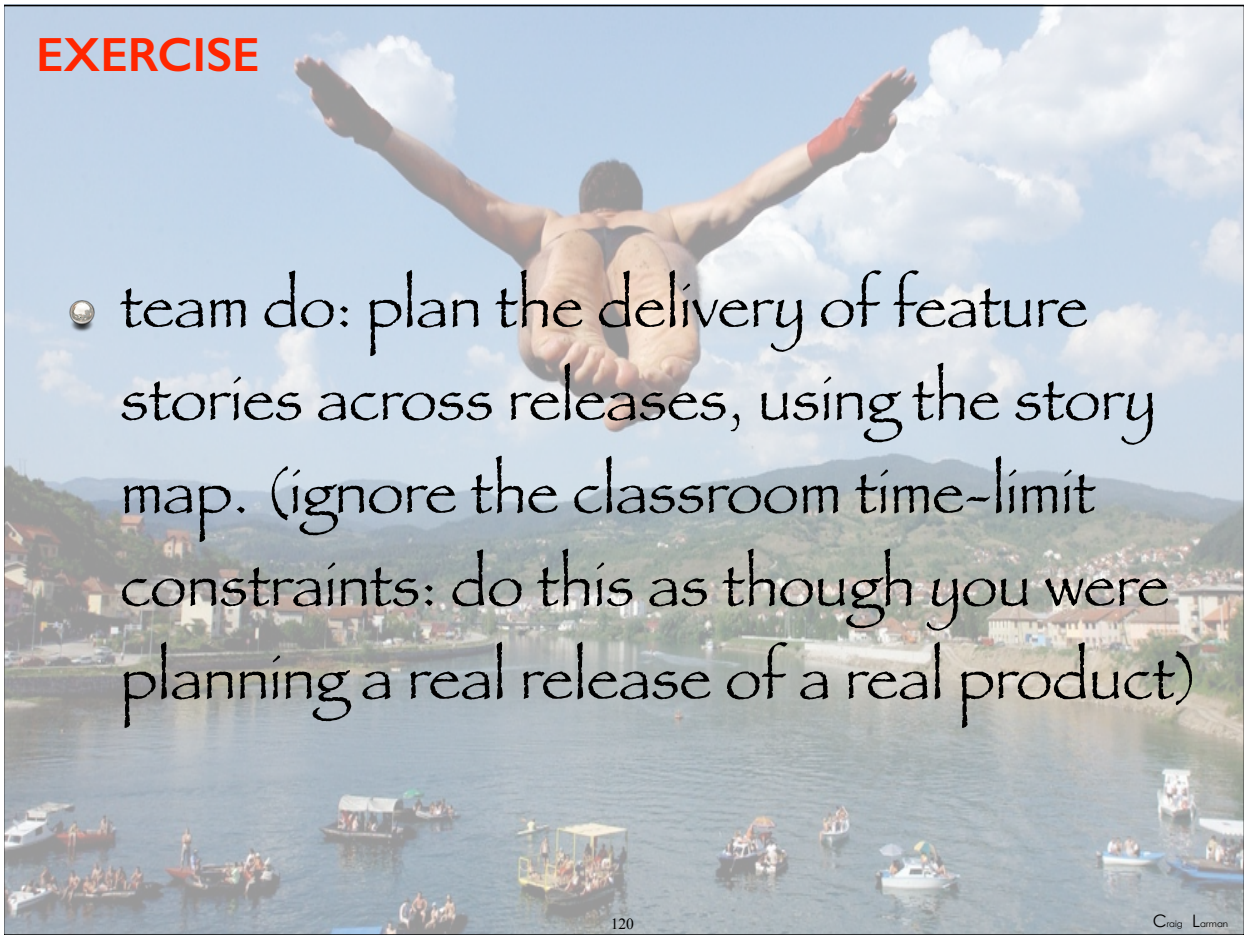
we can and should use the story map as a release planning tool





(notice that this way of planning is very user-centric, because it focuses on the question: What cohesive set of features will we deliver, that are minimally usable in an end-2-end solution to help users cohesively in their tasks?)

minimal usable feature set



EXERCISE

- team do: plan the delivery of feature stories across releases, using the story map. (ignore the classroom time-limit constraints: do this as though you were planning a real release of a real product)

EXERCISE

- class discuss: in Scrum, who makes these planning decisions?

Telling Stories

- a technique for lean & agile analysis
- a behavior to refine requirements

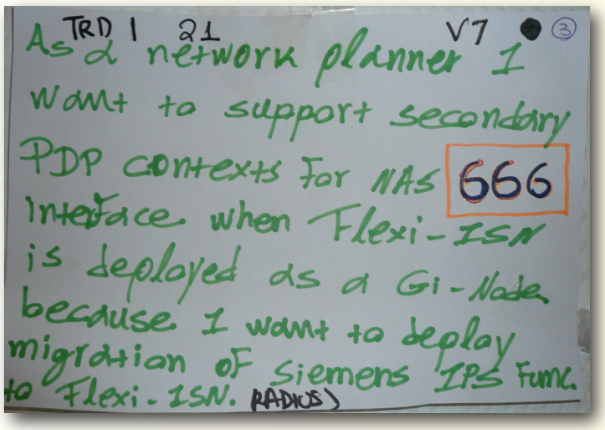
fact: (their creator) Kent Beck named these only "stories" (not "user stories")

why did Kent call them "stories"?

telling stories

EXAMPLE: story

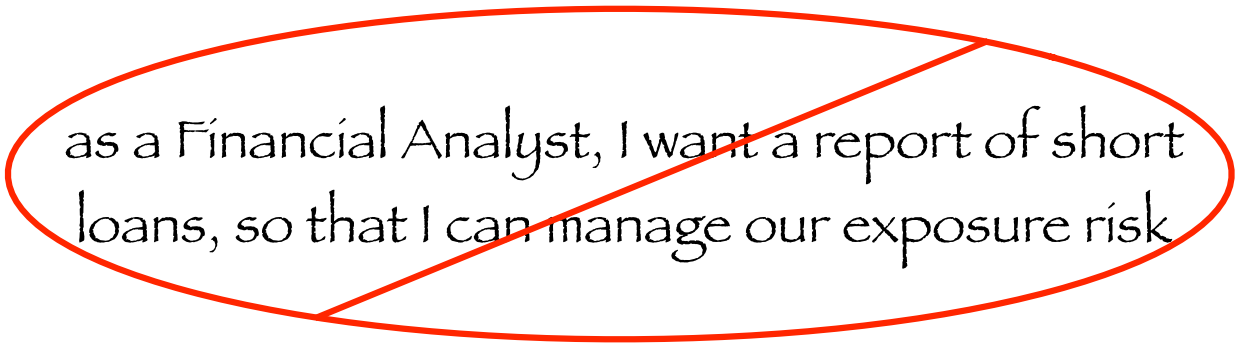
- story: "configure a media gateway"
- or...



there is a widespread misunderstanding about stories

what is it? ...

contrary to common misunderstanding "stories" do NOT mean to write requirements in this template style:



"doing stories" is a BEHAVIOR, summarized by the ...

3 C's

Card

Conversation

Confirmation

story =

Card +

Conversation +

Confirmation

(notice that we have already been "doing stories" since there is card and conversation via story mapping)

the key idea of stories is NOT a special format of writing requirements

the key idea is REQUIREMENTS BY COLLABORATION & CONVERSATION between Product Owner, SMEs, and Team, rather than handing off documents from PO or "analysis group"

a separate analysis group or Product Owner that "writes stories" and hands them off to the Team for implementation has COMPLETELY misunderstood the point of stories

Card...

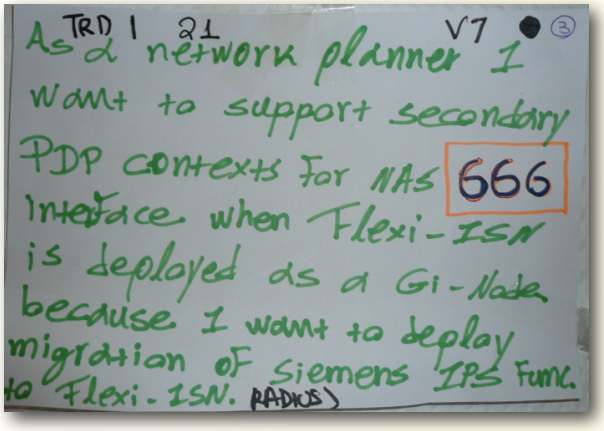
story =

Card +

Conversation +

Confirmation

- literally, we start with a card
- why?



Card...

"Connextra template"...

as a <customer role>
i want <feature/function> so that <benefit>

or...reordered (better!)

to <achieve benefit>
as a <customer role> i want <feature/function>

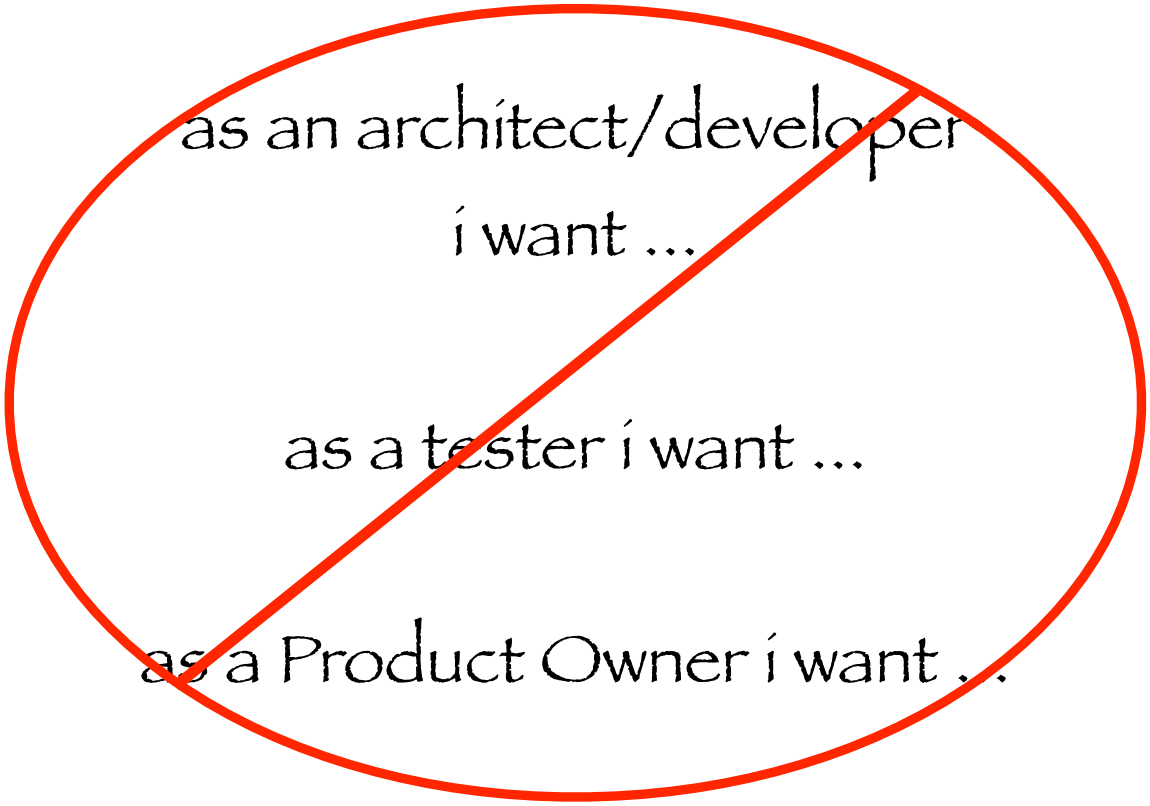
or...leave the solution flexible and creative for the team; sometimes this is best!

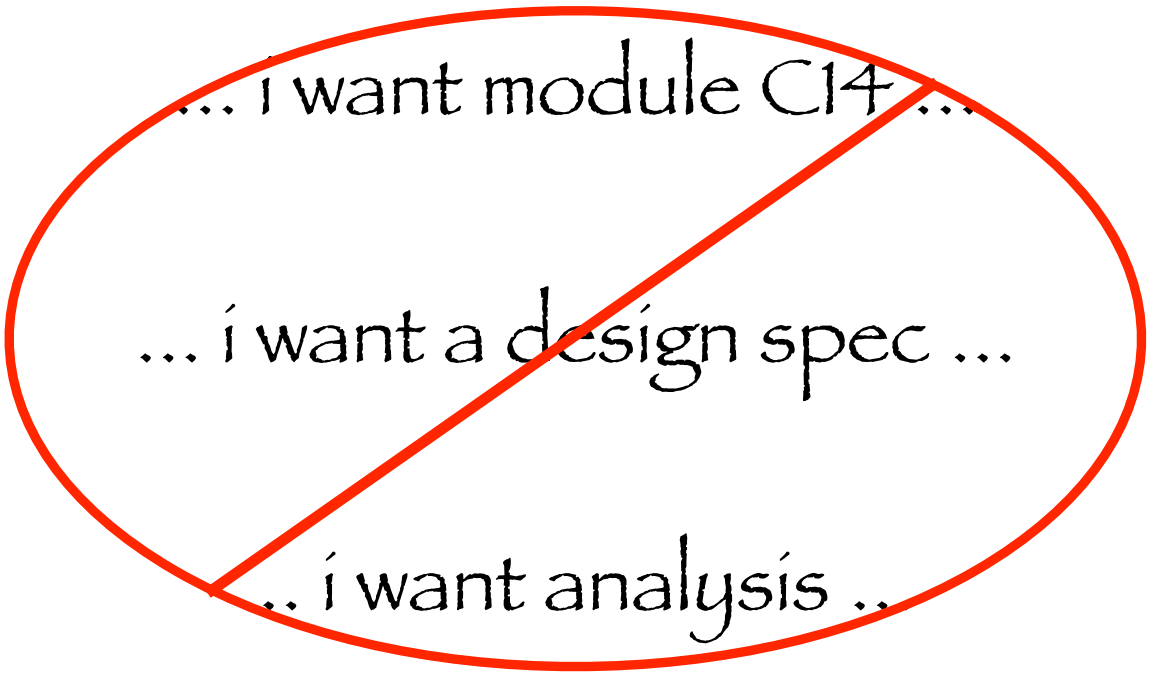
to <achieve benefit>
as a <customer role> i want some solution

example...

to manage our exposure risk, as a Financial Analyst, i want some solution

to increase bandwidth, as a Network Planner, I want to configure a Viper card for system-level vectoring




... i want module C14 ...
... i want a design spec ...
... i want analysis ..

"technical story"

a story is

NOT

a task

Splitting Features

We want to split features small enough to do well- within the scope of one team and one Sprint

e.g., 1/4 or 1/3 of a Sprint

(reduction in variability, deliver something "done" with low WIP, ...)

story =

Card +

Conversation +

Confirmation

we want "smaller" cards and smaller stories — though they should still be end-2-end *features*

1 Introduction 1
2 Large-Scale Scrum 9
Action Tools
3 Test 23
4 Product Management 99
5 Planning 155
6 Coordination 189
7 Requirements & PBIs 215
8 Design & Architecture 281
9 Legacy Code 333
10 Continuous Integration 351
11 Inspect & Adapt 373
12 Multisite 413
13 Offshore 445
14 Contracts 499

splitting

- features can be split; for example, to separate use case scenarios

- use case Trade a Simple Option:
  - main success scenario,
  - scenario 1,
  - scenario 2, ...

GUIDELINE: split features by...

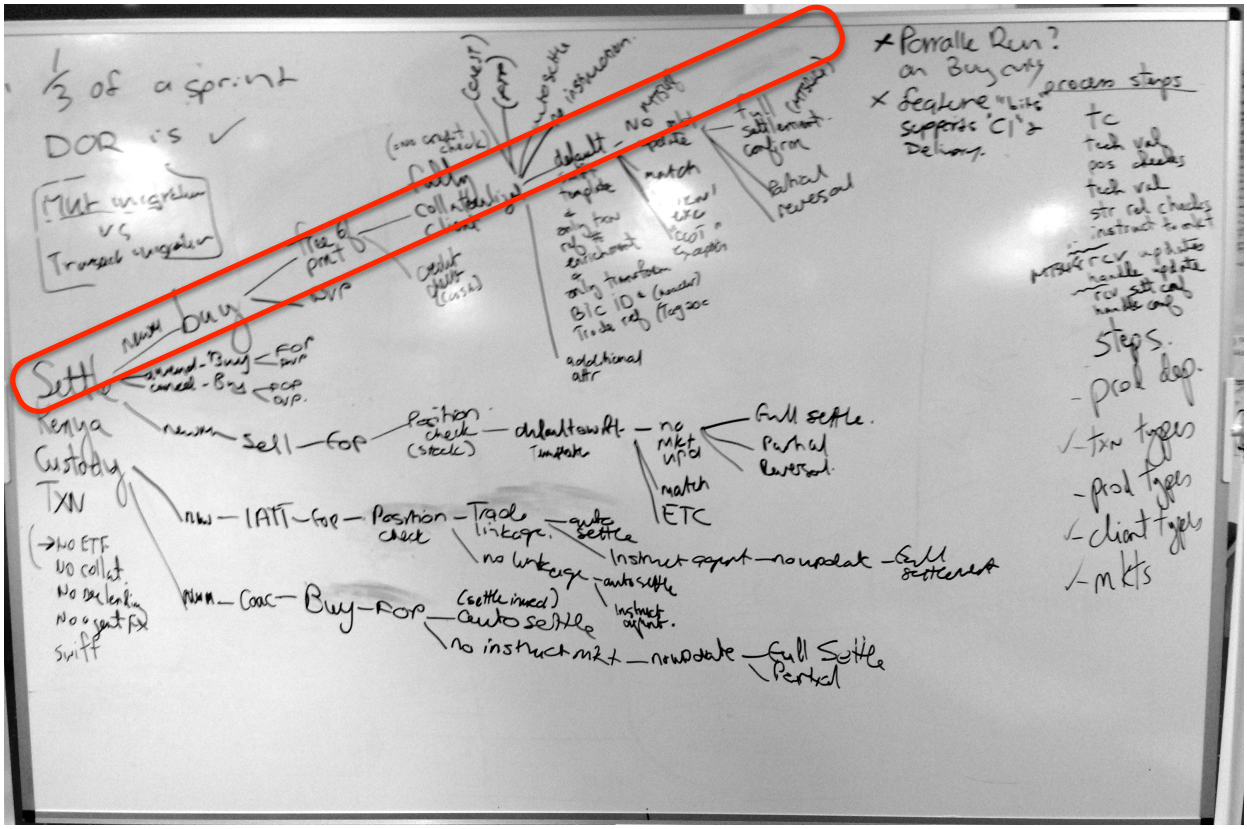
- use cases
  - CRUD use cases
- scenarios of a use case
- data parts
- "type" specialization (e.g., types of trades, workflows, data, jobs, assets, ...)
- scenario steps (not internal algorithm steps)
- integration with external elements
- configurations
- I/O channels (e.g., via GUI, non-GUI, CLI, API,...)
- data formats
- customer role or persona
- "non-functional" requirements
- operation (e.g., HTTP GET, PUT)
- with stubs

trade derivatives on exchanges ...

... trade futures on Globex...

... trade futures on other exchanges...

... trade other derivatives on exchanges...



dhcp. DHCP server support

dhcp.request. request an IP address so that I can easily get a viable IP address and then use the network

dhcp.renew. renew an IP address so that I can easily keep using my current IP address when its lease expires

dhcp.full-support. ...full DHCP support

dhcp.request. request an IP address so that I can easily get a viable IP address and then use the network

dhcp.request.success.main. ...request an IP address and free ones are available...

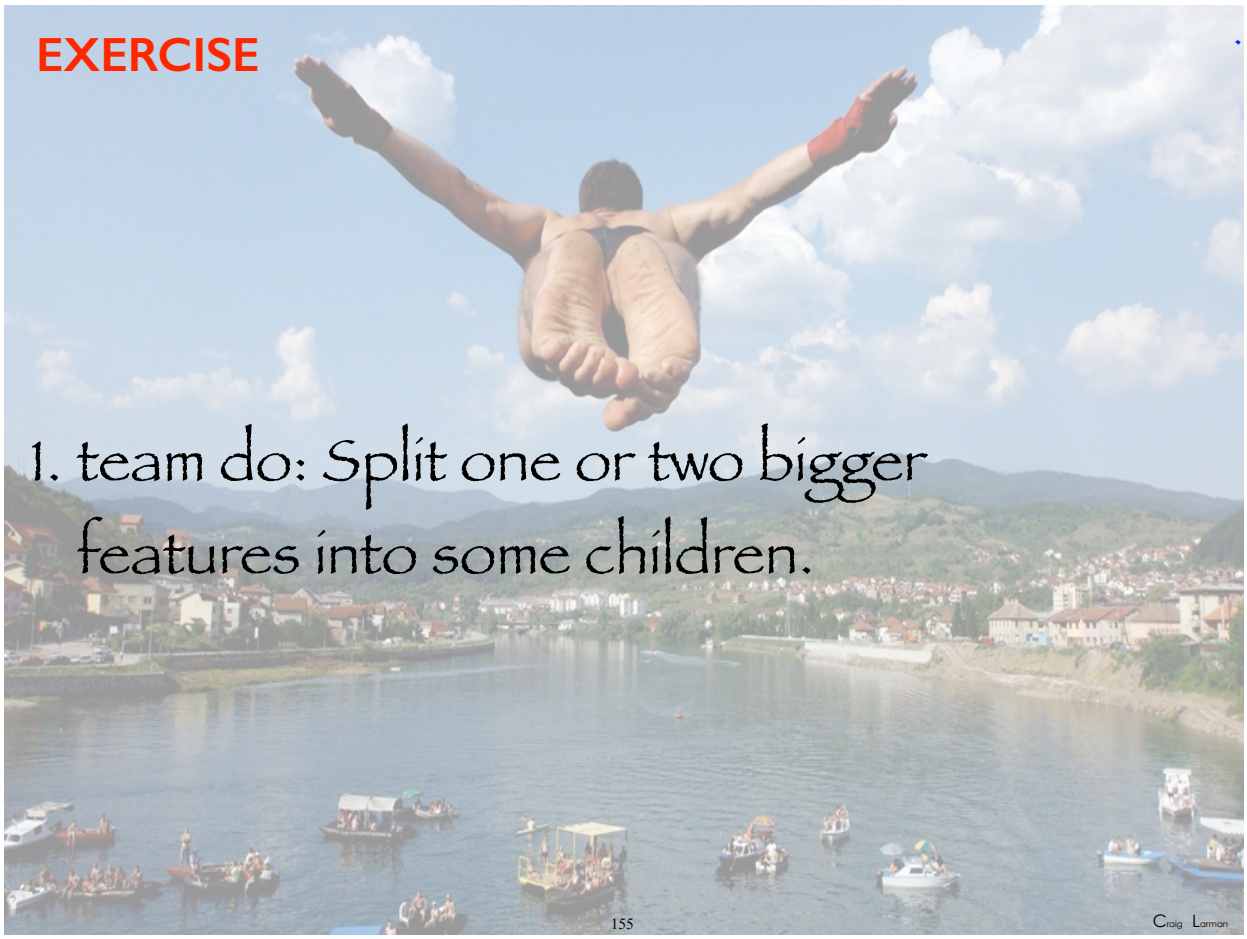
...request any IP address and free ones are available...

...request specific IP address, X, and X is available

dhcp.request.fail.none free. ... error message when requesting an IP address and none are free...

dhcp.request.other. Everything else.

EXERCISE



1. team do: Split one or two bigger features into some children.

Craig Larman

Discussion

What to do with parent features that have been split into children? Should the "parent" remain in the Product Backlog?...

what to do with ancestors?

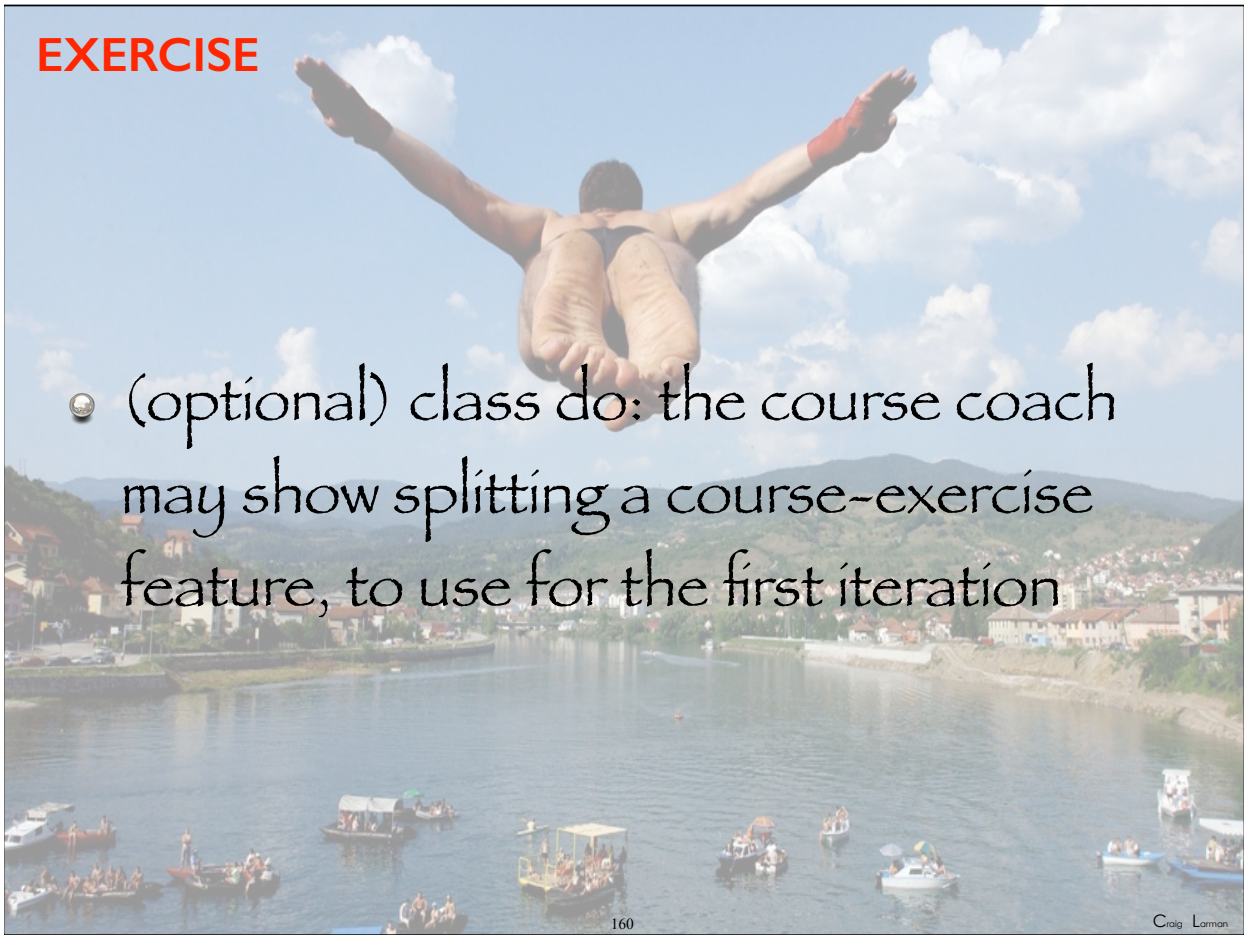
example parent = "decryption services for network traffic"

Item
detect encrypted traffic
decrypt BitTorrent-encrypted traffic
decrypt Blowfish-encrypted traffic

Item	(any meaningful) Ancestor
detect encrypted traffic	
decrypt BitTorrent…	decryption services for network traffic
decrypt Blowfish…	decryption services for network traffic

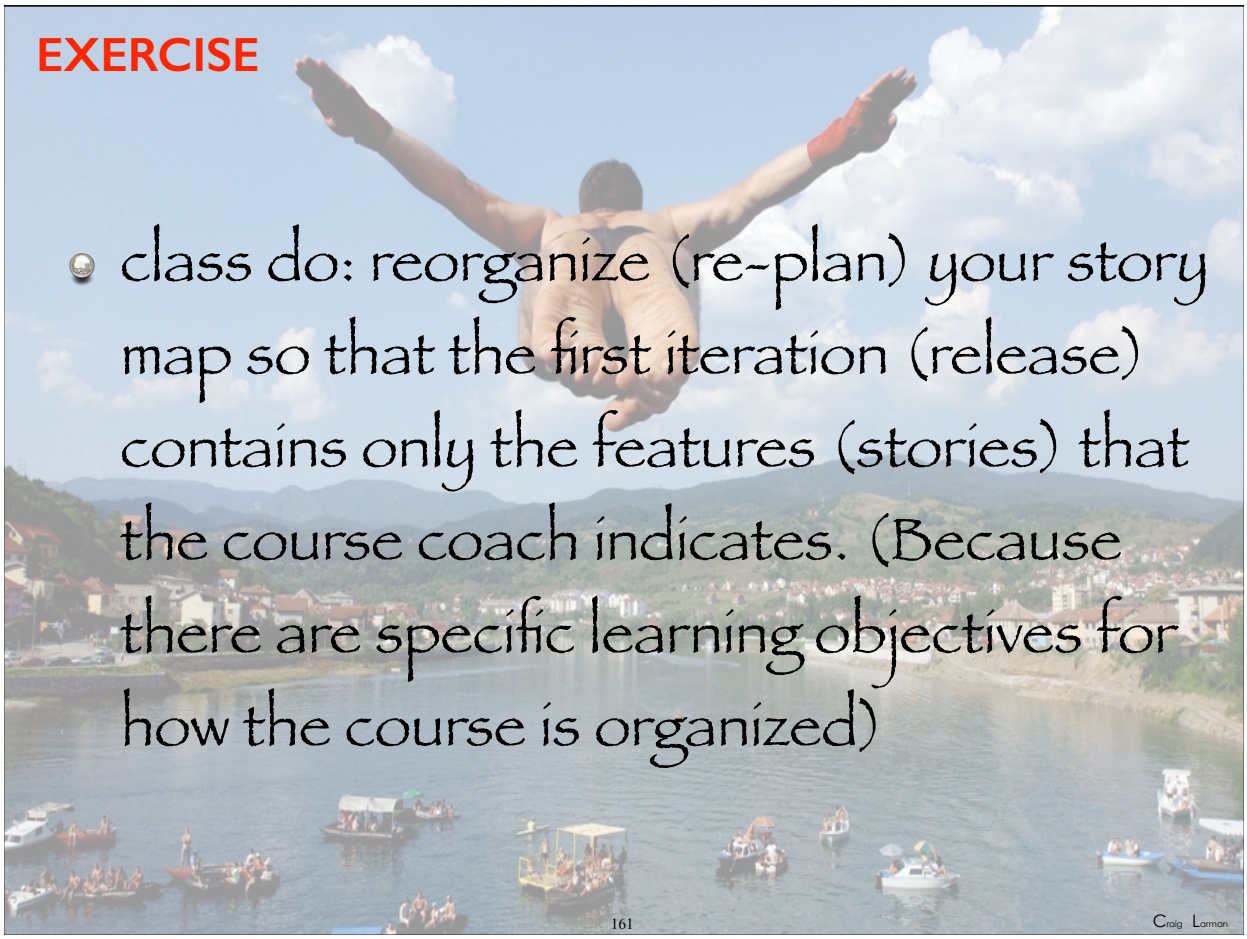
Preparing for Iteration 1

(some miscellaneous small but useful topics & activities we need, before the first iteration)



EXERCISE

- (optional) class do: the course coach may show splitting a course-exercise feature, to use for the first iteration



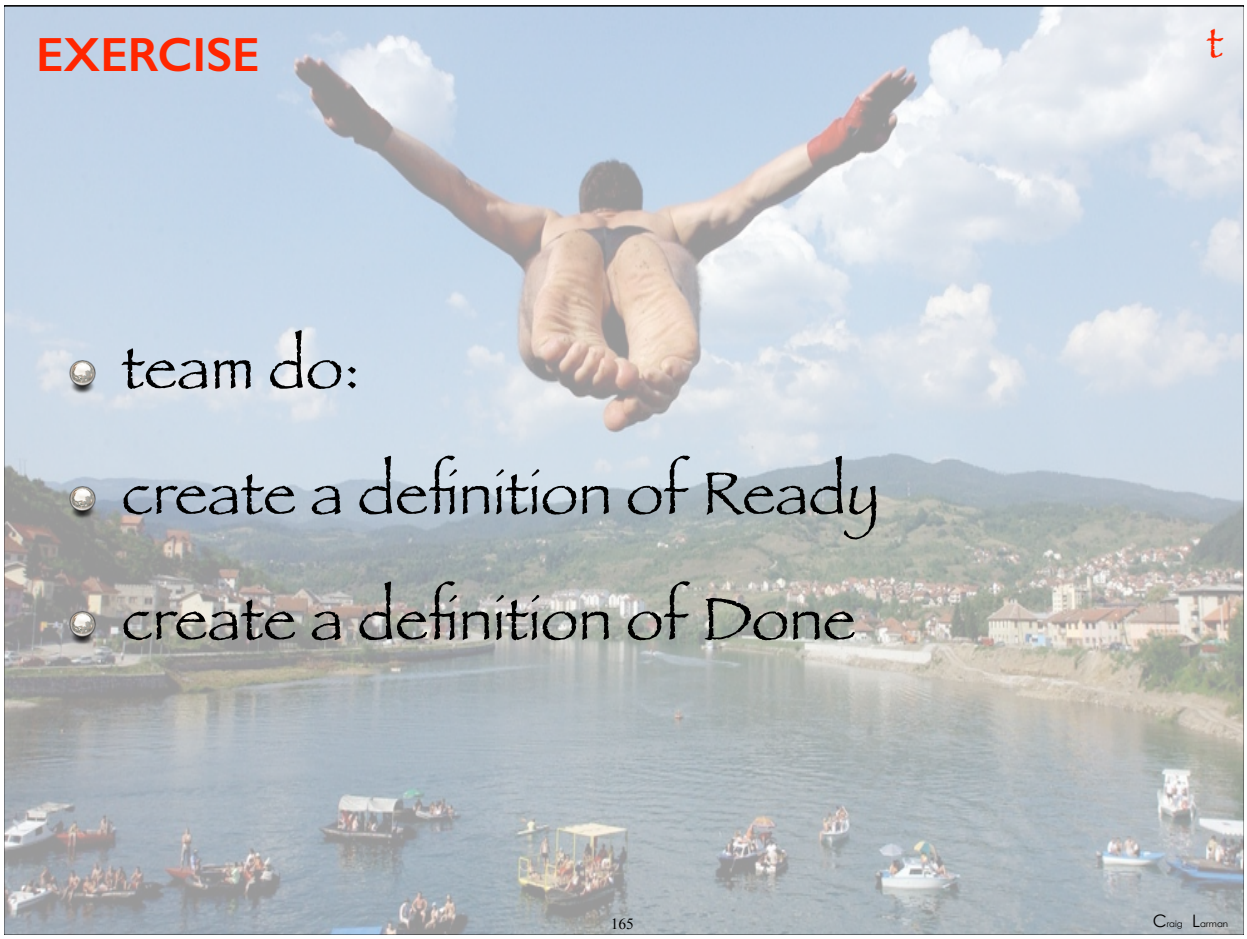
EXERCISE

- class do: reorganize (re-plan) your story map so that the first iteration (release) contains only the features (stories) that the course coach indicates. (Because there are specific learning objectives for how the course is organized)

"Ready" & "Done" ...

"Ready" or "definition of ready"

DoR → / STORIES ↓	Narrative format	Understood business terms	Acceptance tests defined	Spec by example	Controls + exceptions defined	UI wireframes agreement	External data inputs/outputs defined	Story size is 1/3 Sprint Size	Stakeholders identified and available	Persons are human	Non-functional requirements defined	Priority assigned by P.O.	User story info stored in some place	User stories must be solution agnostic
#1 CREDIT RISK	✓				✓	N/A							✓	✓
#2 MTM PRICING	✓				✓	N/A			✓				✓	✓
#3 GENERAL RECONCILIATION	✓					N/A			✓				✓	✓
#4 LIQUIDITY RISK	✓				✓	N/A			✓				✓	✓
#5 P+L RECONCILIATION (MS)	✓					N/A							✓	✓
#6 REG REPORTING	✓	✓	✓		✓	N/A	✓	✓	✓	✓			✓	✓
#7 CORPORATE GL FEED	✓				✓	N/A			✓				✓	✓
#8 CORPORATE GL VS DOM FIXED INCOME	✓					N/A			✓				✓	✓



EXERCISE

- team do:
- create a definition of Ready
- create a definition of Done

Craig Larman

FURPS+ ...

a feature is not really "Ready" for implementation until both functional and non-functional requirements are "confirmed."

Remember FURPS+ to help you…

remember FURPS+

WHAT ARE THE TYPES AND CATEGORIES OF REQUIREMENTS?

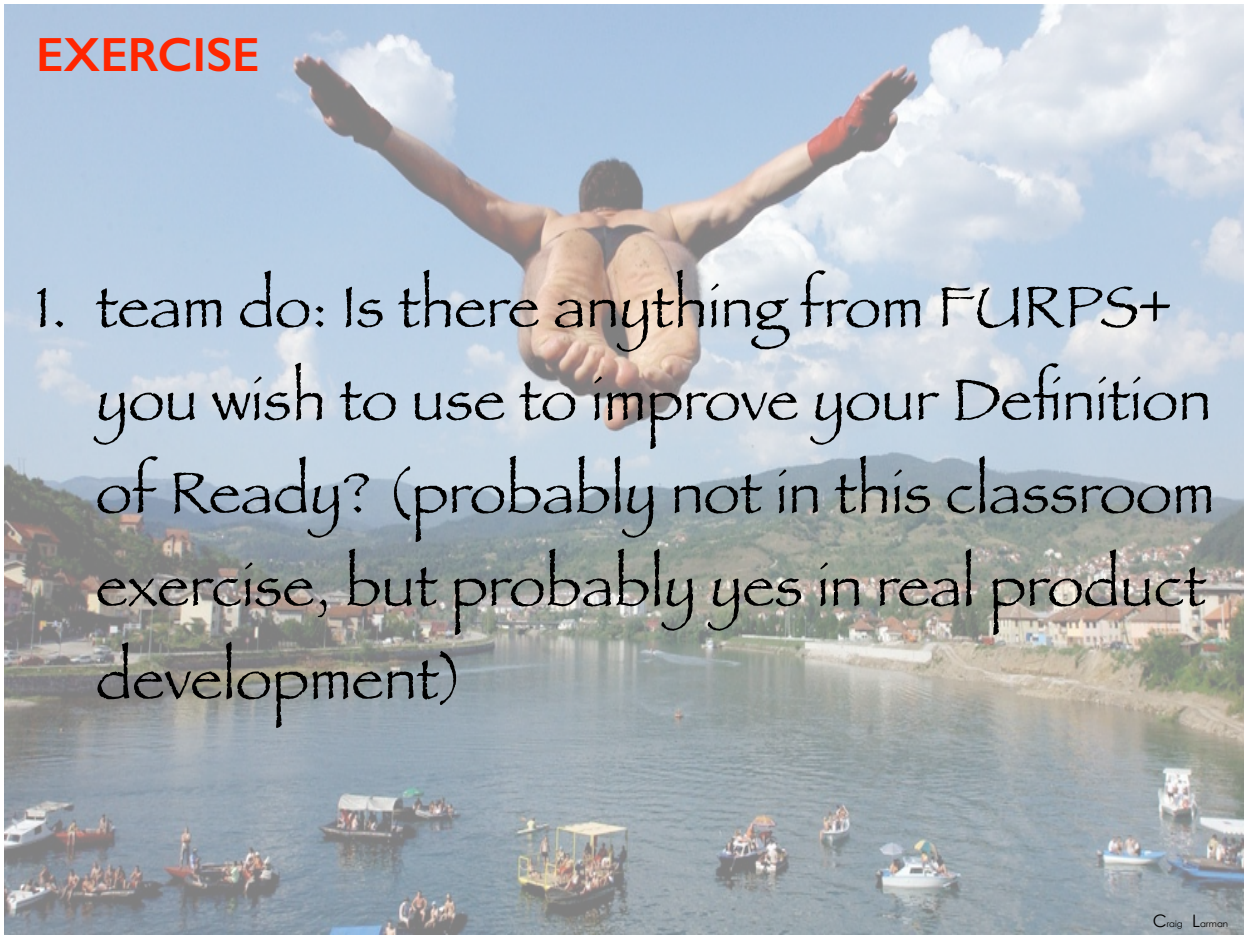
[Grady92], a useful mnemonic with the following meaning:[1]

- **Functional**—features, capabilities, security.
- **Usability**—human factors, help, documentation.
- **Reliability**—frequency of failure, recoverability, predictability.
- **Performance**—response times, throughput, accuracy, availability, resource usage.
- **Supportability**—adaptability, maintainability, internationalization, configurability.

The "+" in FURPS+ indicates ancillary and sub-factors, such as:

- **Implementation**—resource limitations, languages and tools, hardware, ...
- **Interface**—constraints imposed by interfacing with external systems.
- **Operations**—system management in its operational setting.
- **Packaging**—for example, a physical box.
- **Legal**—licensing and so forth.

EXERCISE
1. individual do: review the FURPS+ list
2. standing with 1 "talking partner": without referring to notes, define FURPS+.
Craig Larman


EXERCISE
1. team do: Is there anything from FURPS+ you wish to use to improve your Definition of Ready? (probably not in this classroom exercise, but probably yes in real product development)
Craig Larman

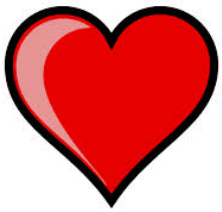
User-Centered Design

beware doing user-centered design (UCD) and user experience (UX) design focusing on secondary minor techniques (personas, mockups, …)

… rather than primary UCD elements

1

the heart of it…



empathy

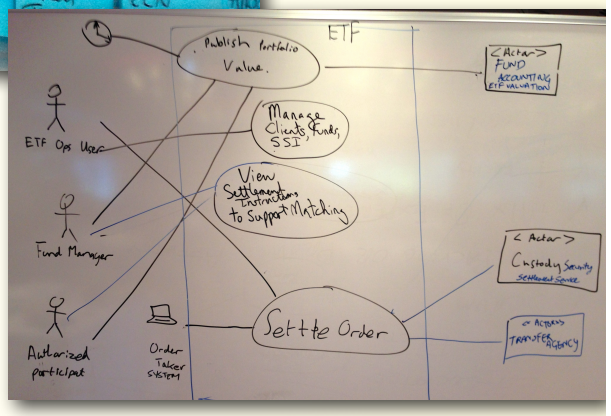
and… Developers regularly face to face with REAL USERS in a system of feedback loops

is more important than

UX and UI modeling techniques

secondarily? …

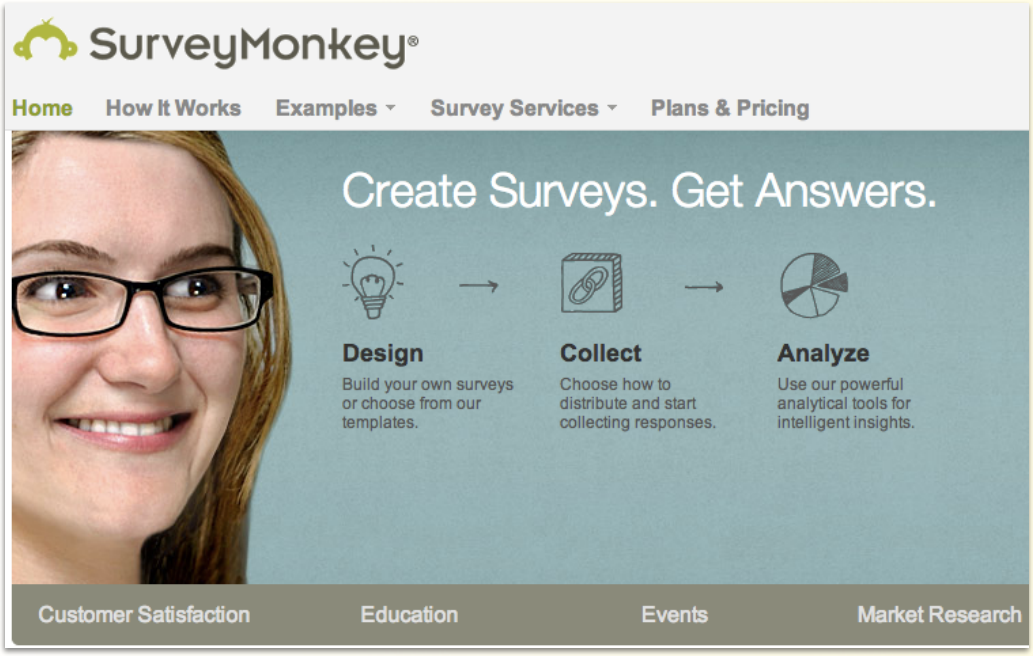
user task analysis



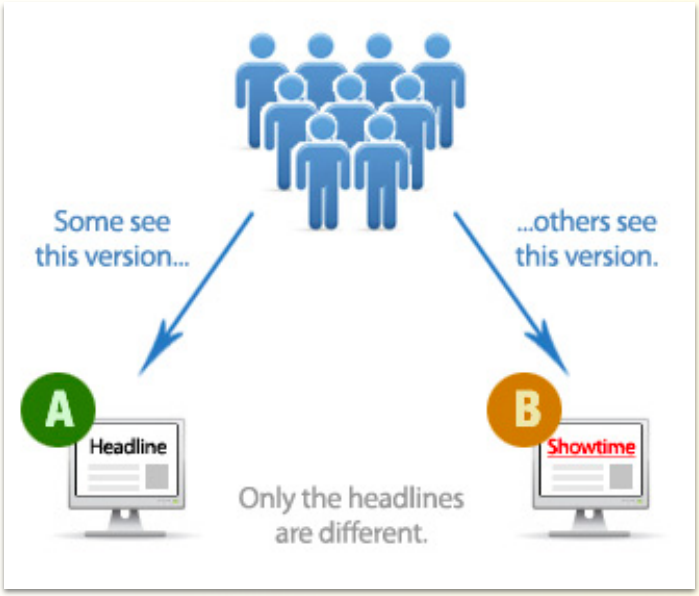
current and/or future state

how might it change when there is software?

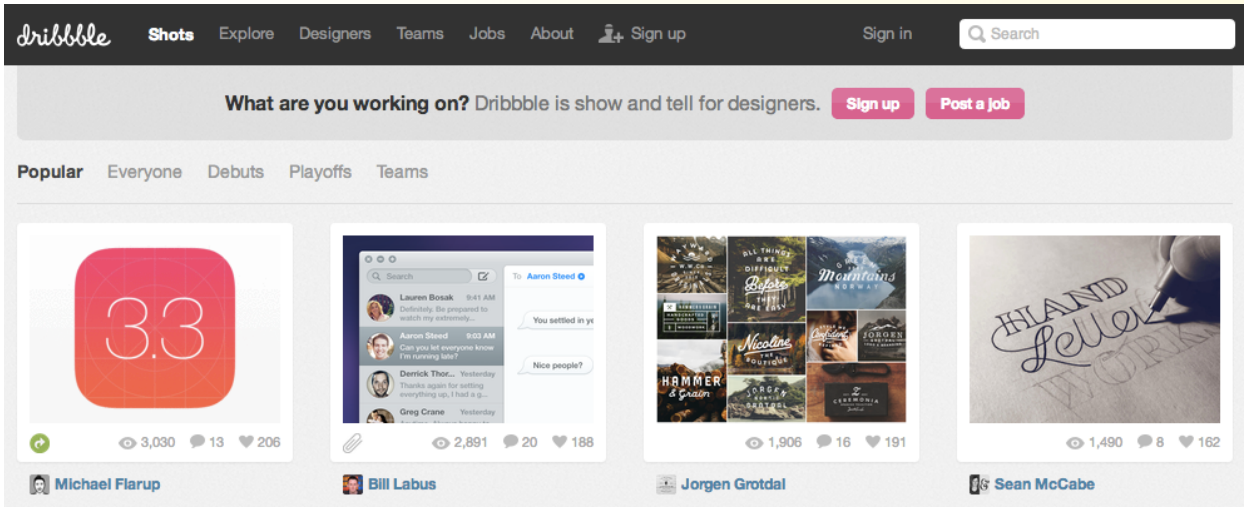
surveys



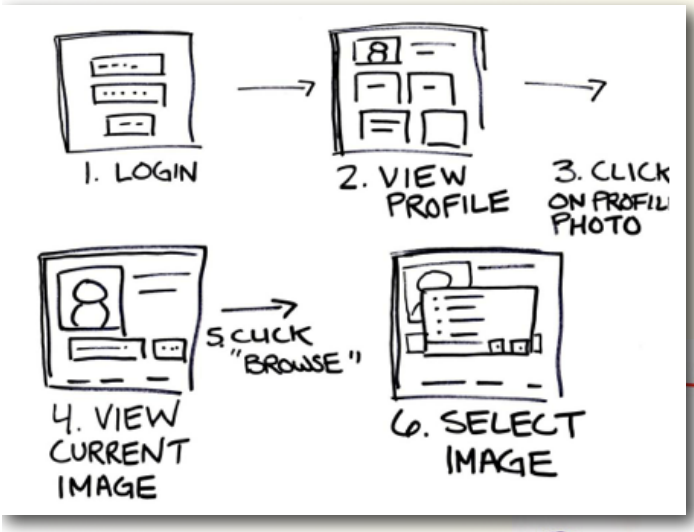
A/B testing

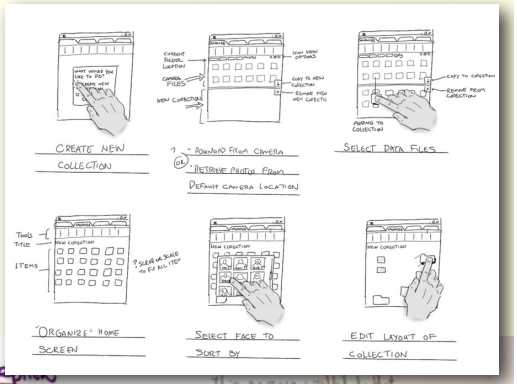


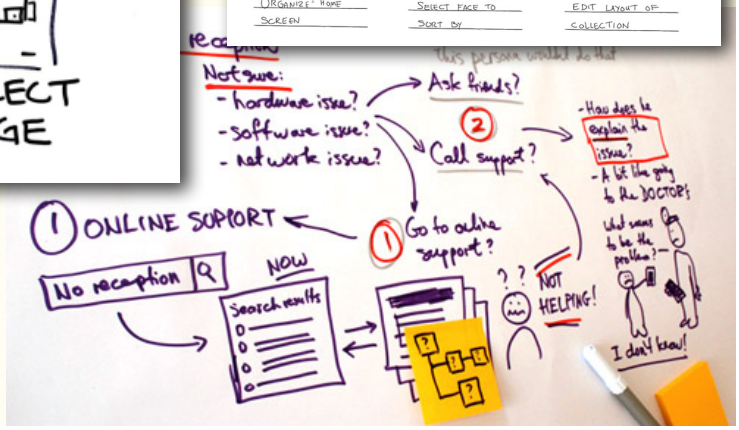
steal UI ideas



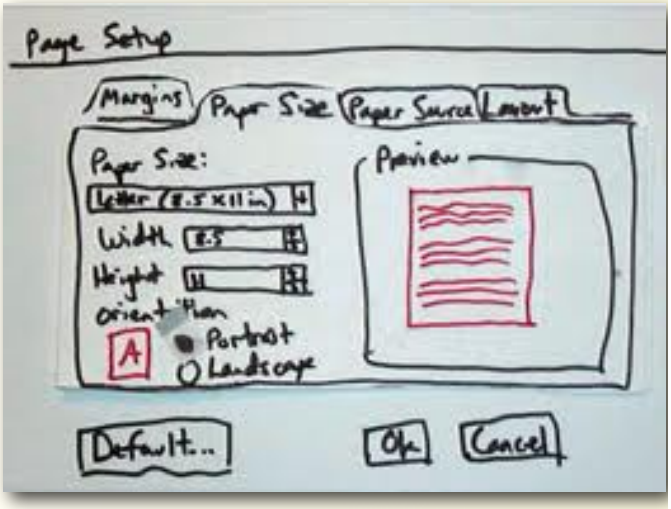
storyboarding



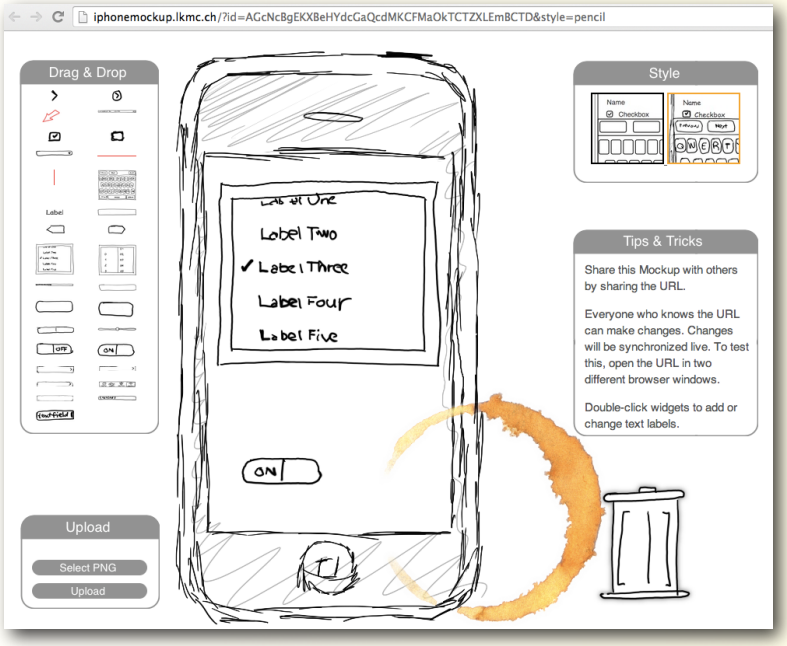




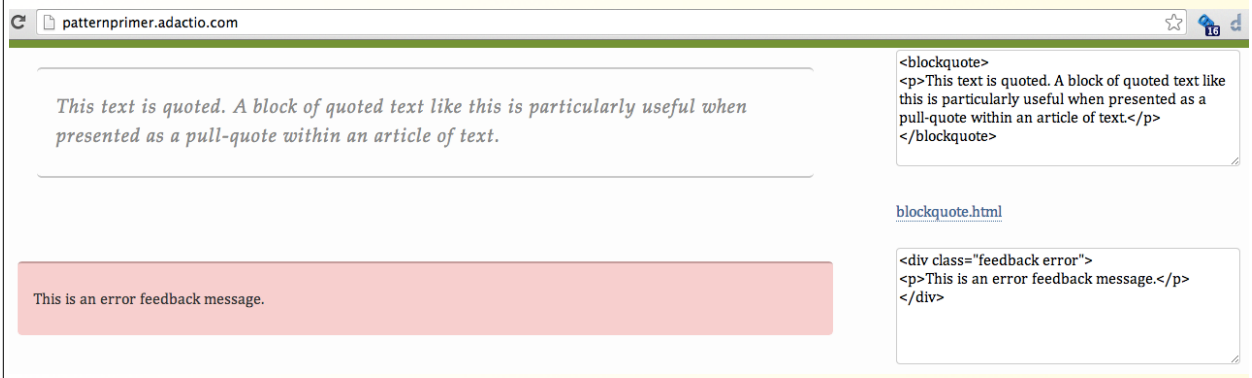
low-fidelity paper/whiteboard mockup for information content & layout



higher-fidelity software prototypes with styles that are clearly "rough" ("pencil style"), so users feel they can freely criticize



LIVE UI style guide

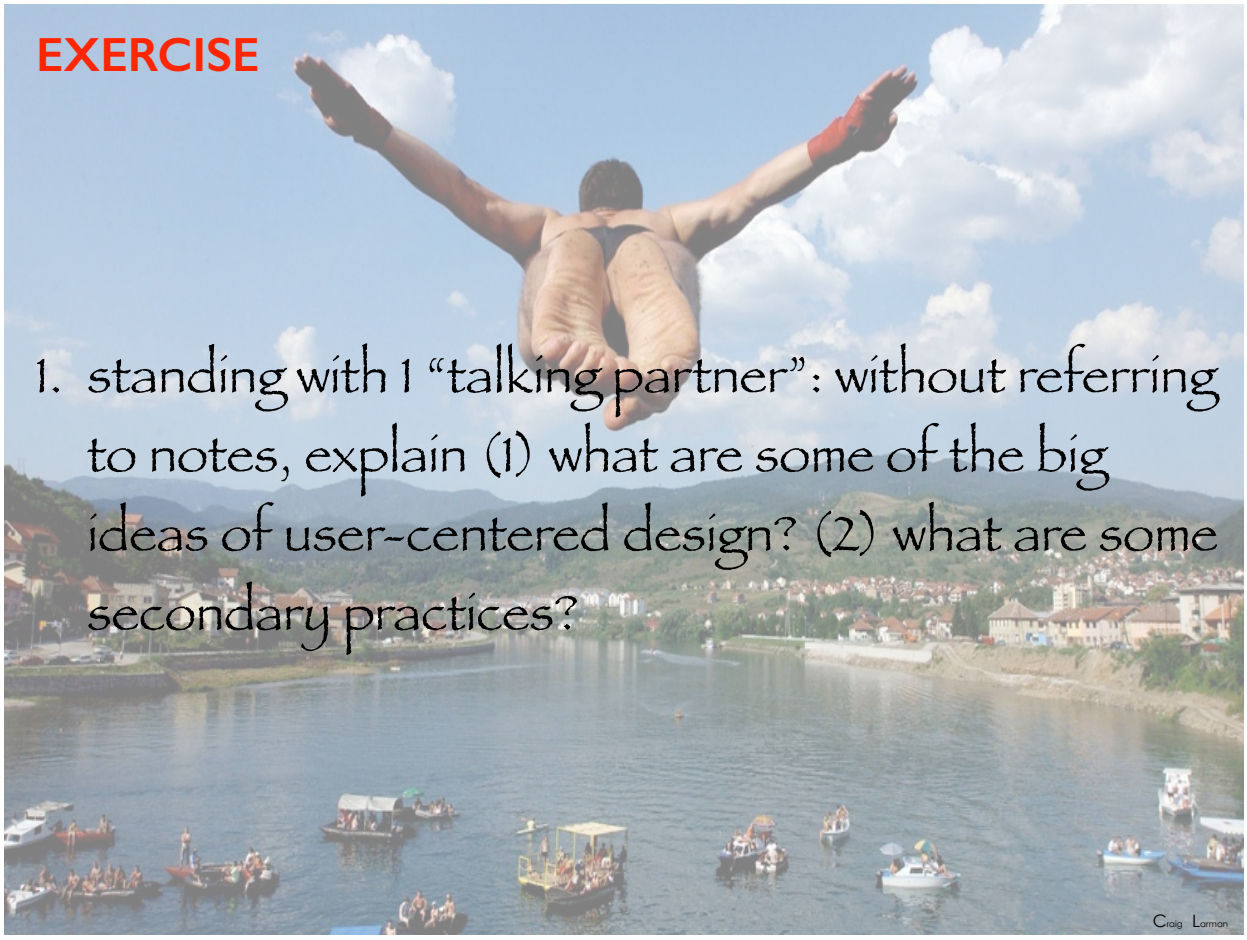


Know UI Design Principles

- proximity
- visibility, visual feedback, visual prominence
- hierarchy
- mental models & metaphors
- progressive disclosure
- consistency
- affordance & constraints
- confirmation
- hick's law
- fitt's law
- …

The details and application are outside the scope of this course, but these are basic & important for every Developer creating a UI

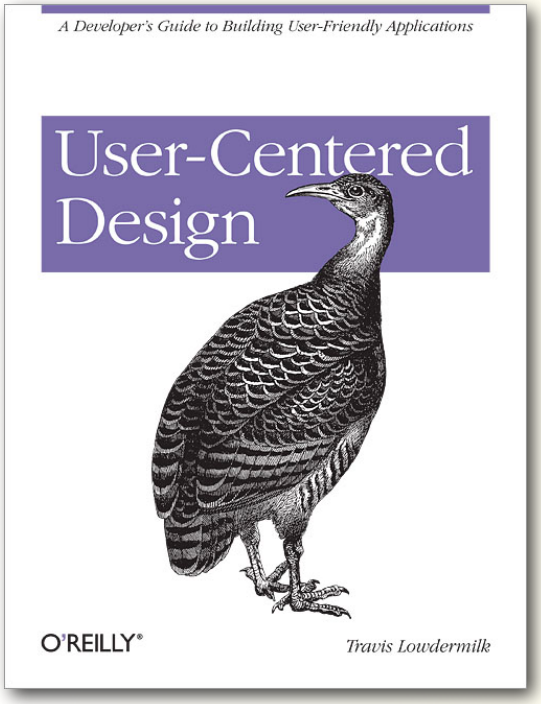
EXERCISE

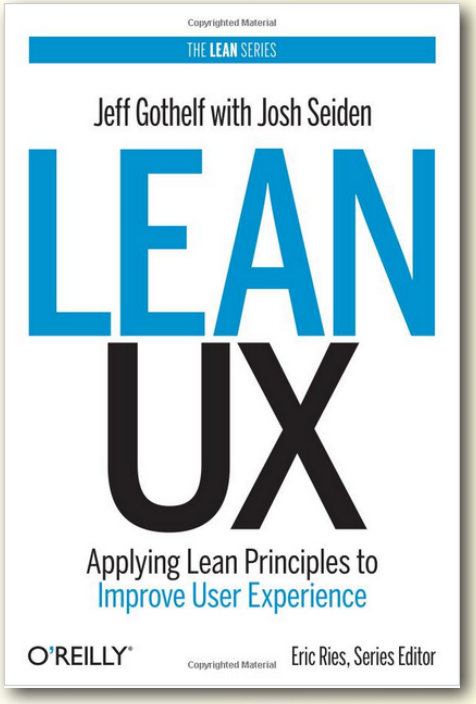


1. standing with 1 "talking partner": without referring to notes, explain (1) what are some of the big ideas of user-centered design? (2) what are some secondary practices?

(the course won't usually point to learning resources, but Developers often are not aware of UCD and UX resources, so…)

Learning Resources



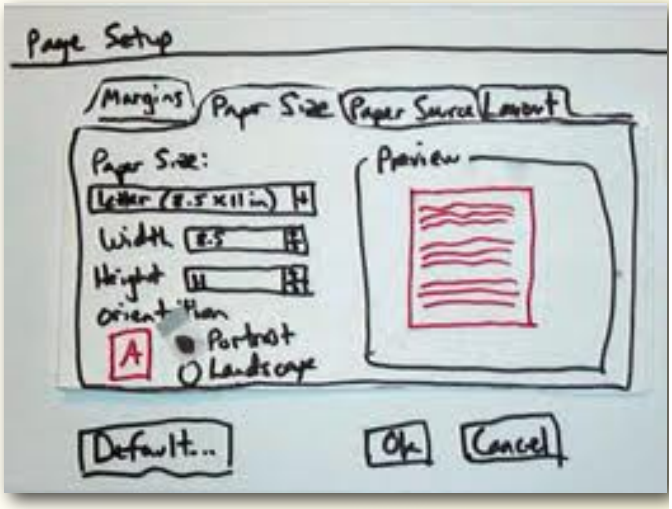


Learning Resources

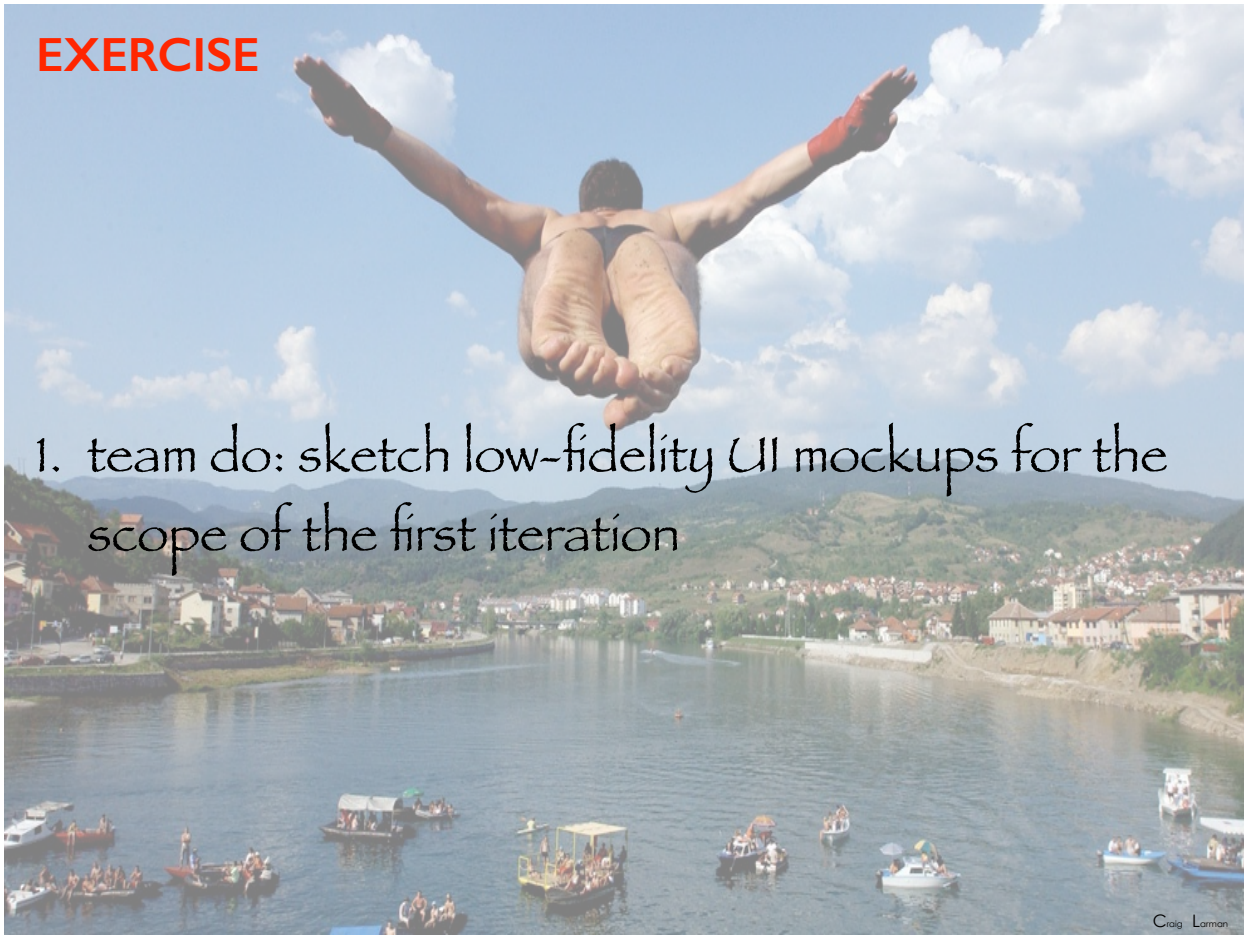
- boxesandarrows.com
- netmagazine.com
- smashingmagazine.com
- uxmatters.com

Learn with a Low-Fidelity UI Mockup

low-fidelity paper/whiteboard mockup for information content & layout



EXERCISE



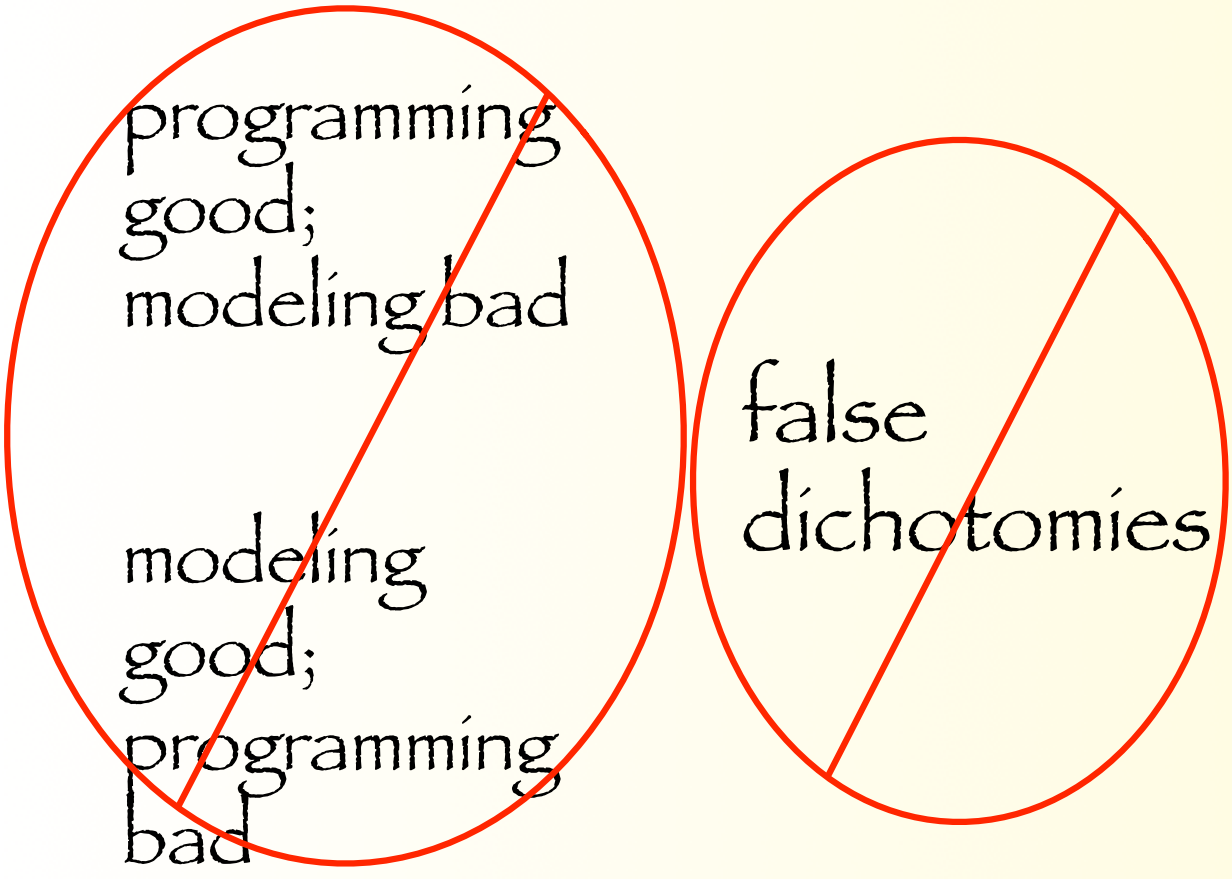
1. team do: sketch low-fidelity UI mockups for the scope of the first iteration

a low-fi UI mockup is a model!

Models to Have Conversations and Learn? ...

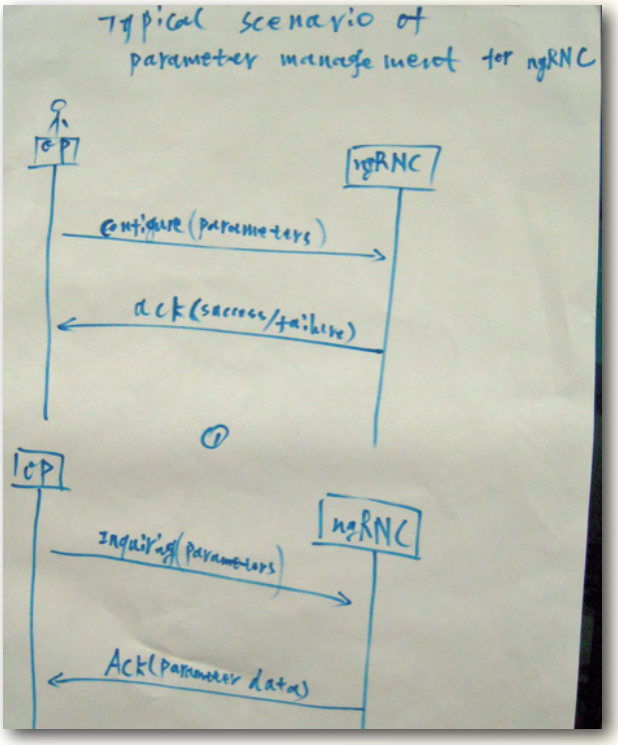
categories & examples of models

	analysis	design
static	story map use case dgm domain object model	design class dgm
dynamic	story map use case dgm stories low-fidelity UI mockups system sequence diagrams activity dgm of process specification by example state machine dgm of domain object	activity dgm of algorithm communication dgm

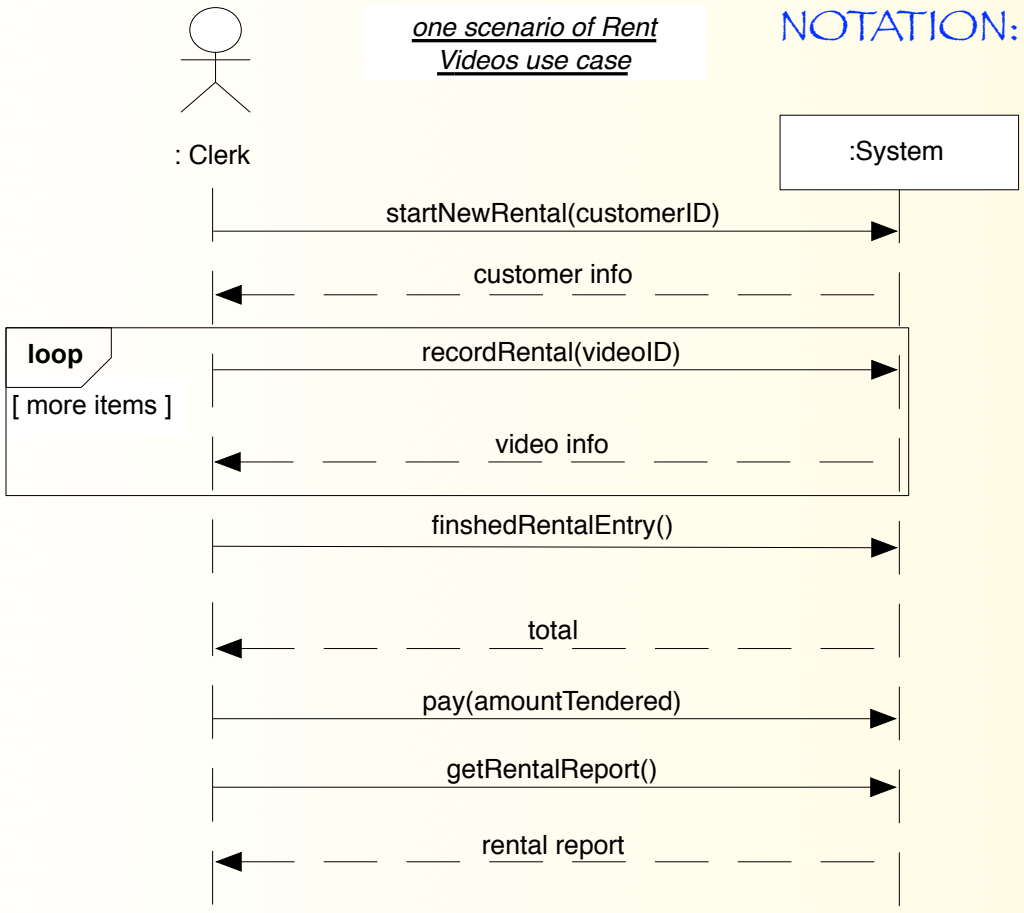


Learn with System Sequence Diagrams

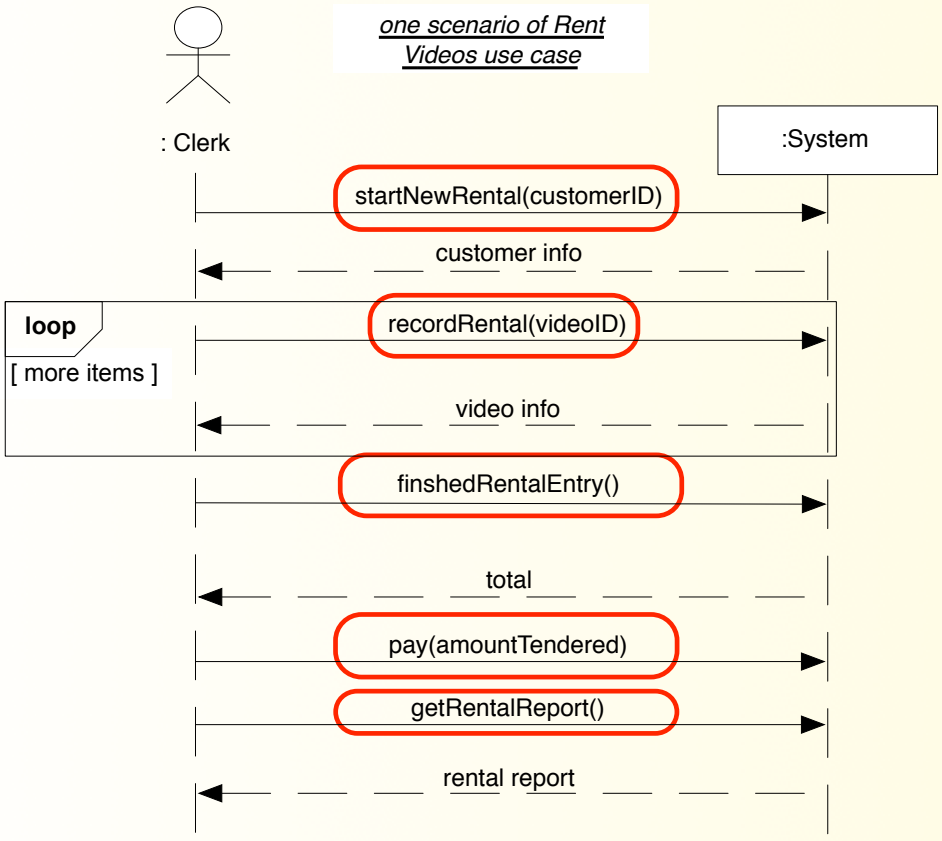
SSDs show system I/O for a scenario



NOTATION: SSD

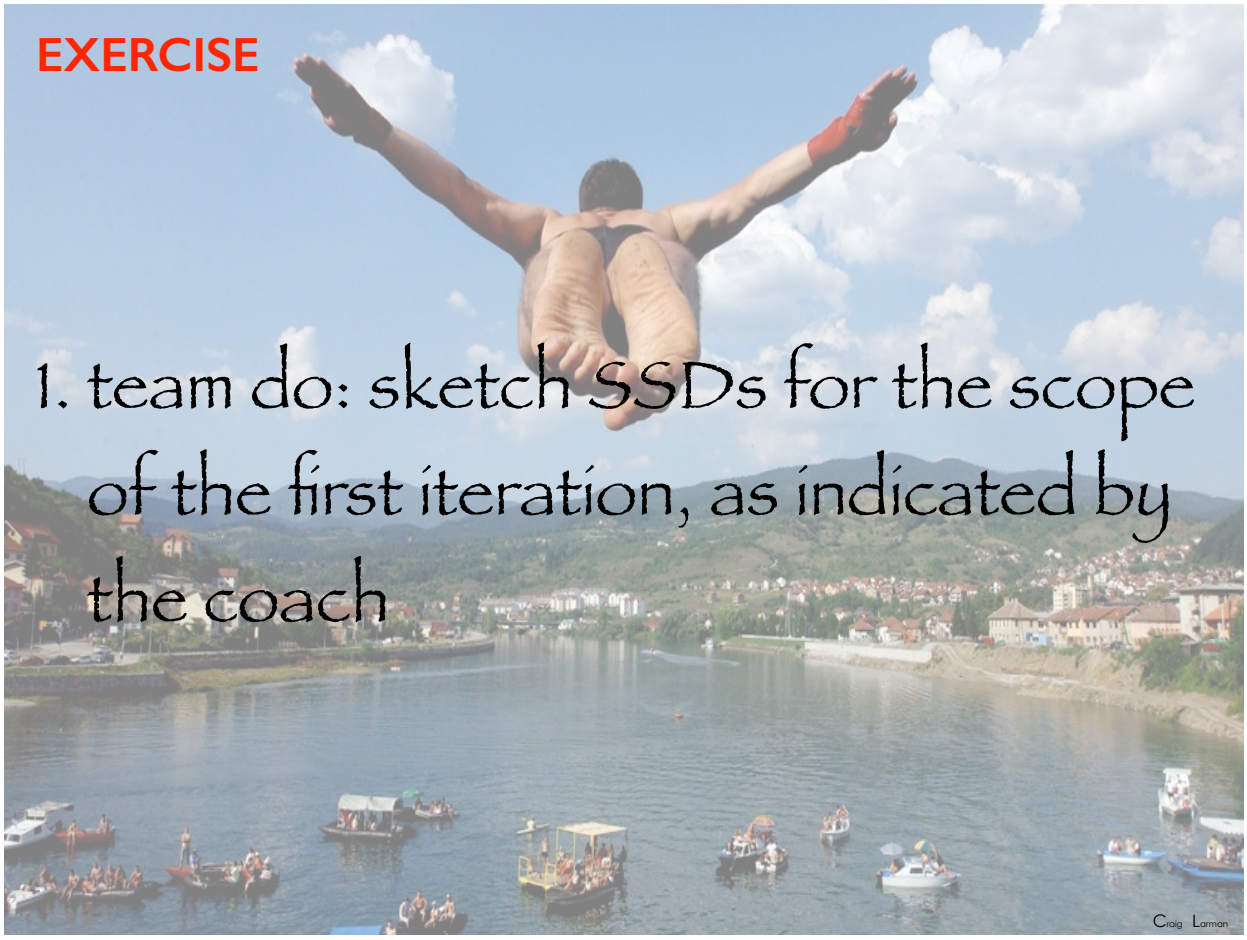


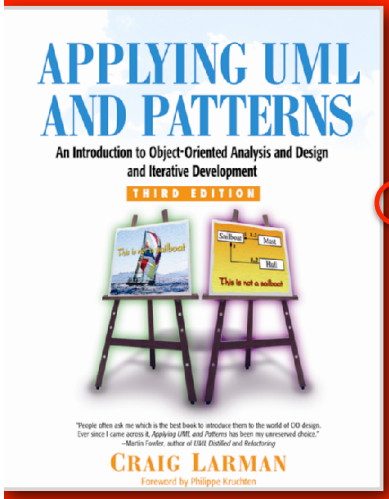
DEFINITION: System Operation



GUIDELINES: SSD

- at least for the Main Success Scenario of one or more use cases
  - showing EXAMPLES
- name system operations starting with verb
  - payFines(...) authorize(...)
- name system operations at the level of intention. "keep the UI out."
  - Worse: scan(videoID)
  - Better: recordRental(videoID)





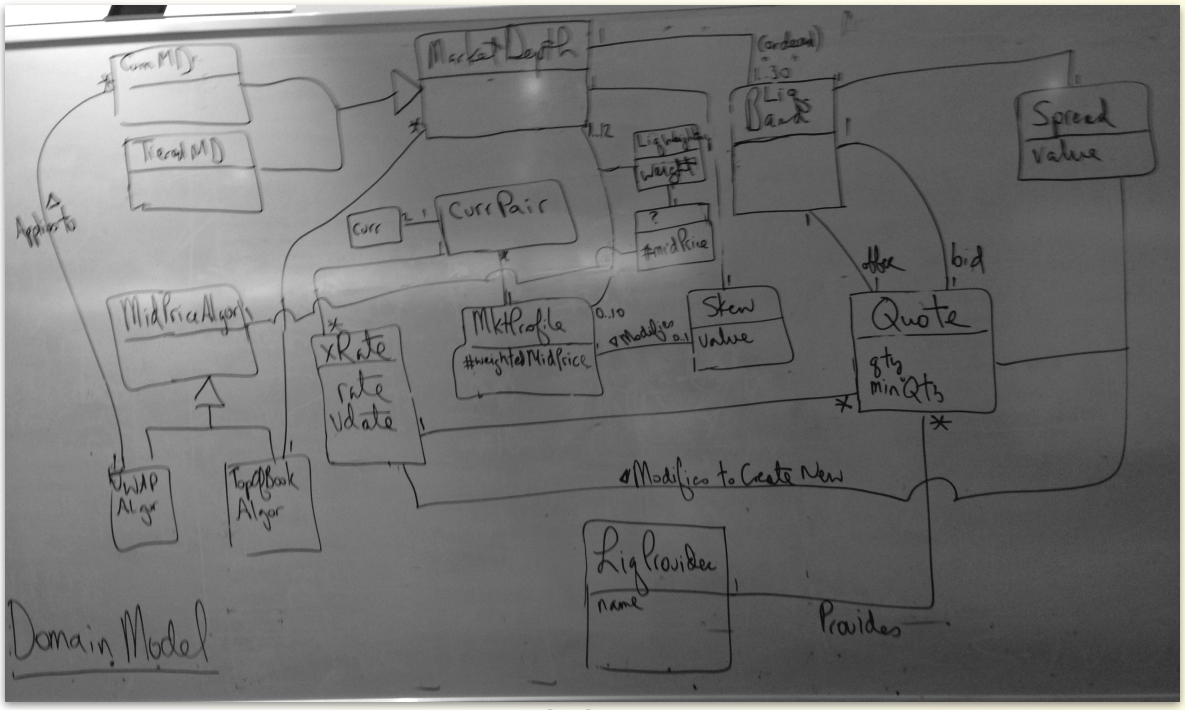
1 Object-Oriented Analysis and Design 3
2 Iterative, Evolutionary, and Agile 17
3 Case Studies 41
PART II INCEPTION
4 Inception is Not the Requirements Phase 47
5 Evolutionary Requirements 53
6 Use Cases 61
7 Other Requirements 101
PART III ELABORATION ITERATION 1 — BASICS
8 Iteration 1—Basics 123
9 Domain Models 131
10 System Sequence Diagrams 173
11 Operation Contracts 181
12 Requirements to Design—Iteratively 195
13 Logical Architecture and UML Package Diagrams 197
14 On to Object Design 213
15 UML Interaction Diagrams 221
16 UML Class Diagrams 249
17 GRASP: Designing Objects with Responsibilities 271
18 Object Design Examples with GRASP 321
19 Designing for Visibility 363
20 Mapping Designs to Code 369
21 Test-Driven Development and Refactoring 385
22 UML Tools and UML as Blueprint 395
PART IV ELABORATION ITERATION 2 — MORE PATTERNS
23 Iteration 2—More Patterns 401
24 Quick Analysis Update 407
25 GRASP: More Objects with Responsibilities 413

Learn with a Domain Model

- sometimes it is useful to...

- learn the noteworthy concepts in a domain
- establish a common vocabulary
- understand the relationships between concepts

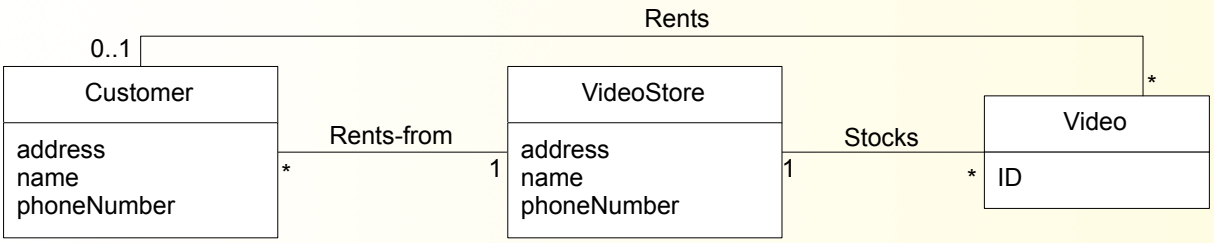
EXAMPLE: Domain Model



a visual dictionary

a context for a conversation to learn noteworthy concepts and to create shared vocabulary

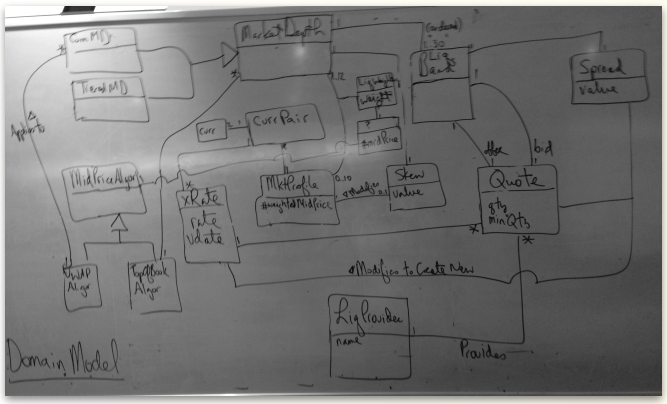
NOTATION: Domain Model



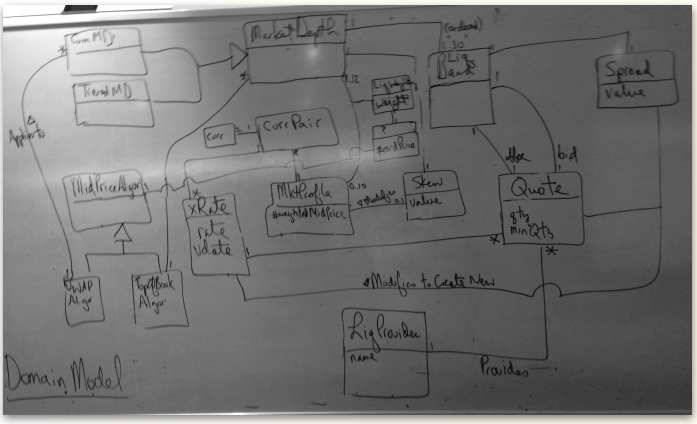
a visual dictionary

a context for a conversation to learn noteworthy concepts and to create shared vocabulary

GUIDELINE: diagram at wall to sketch domain model



warning! NOT a software model



this is a "philosophical" conceptual model

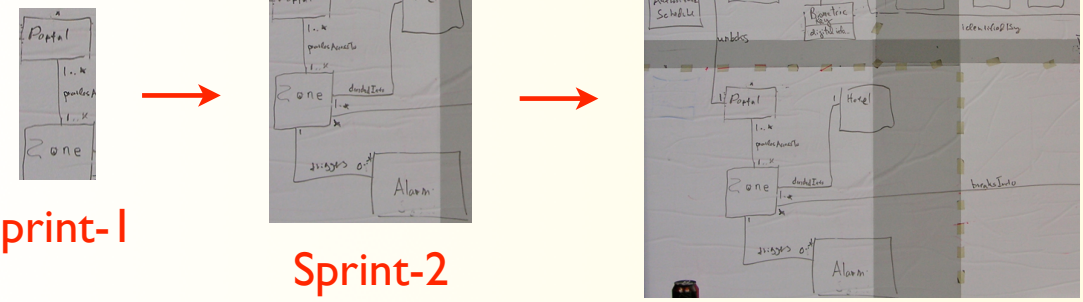
it is a picture of people's *mental model* of a domain

it is NOT a picture of software design, Java, C++, or data bases, or logical data models

Craig Larman

GUIDELINE: evolutionary domain modeling

- do NOT create a big domain model for the complete scope of the final system
- in general, modeling beyond the current Sprint is OK, but that doesn't mean modeling "everything"
- each Sprint, evolve the model (if useful), as new scope is added



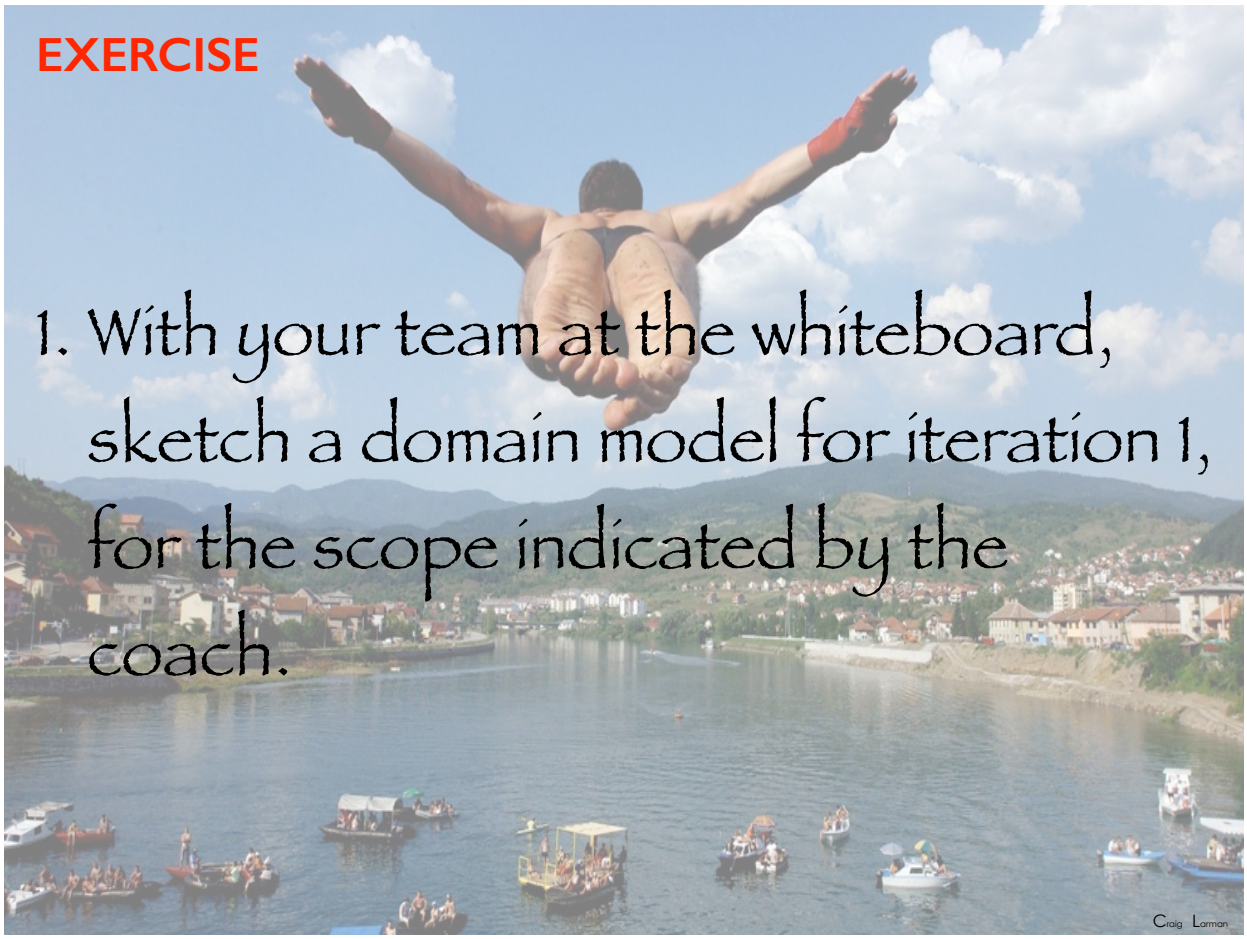
Sprint-1

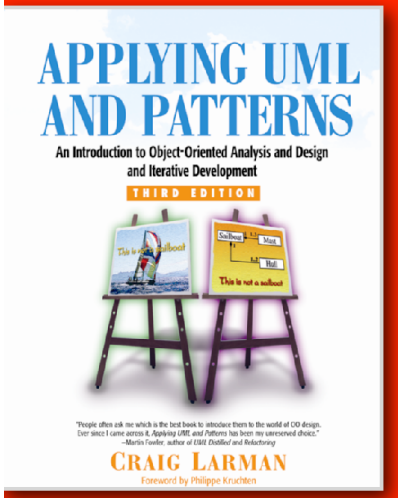
Sprint-2

Sprint-3

Craig Larman

EXERCISE





PART III ELABORATION ITERATION 1 — BASICS

8 Iteration 1—Basics 123
9 Domain Models 131
10 System Sequence Diagrams 173
11 Operation Contracts 181
12 Requirements to Design—Iteratively 195
13 Logical Architecture and UML Package Diagrams 197
14 On to Object Design 213
15 UML Interaction Diagrams 221
16 UML Class Diagrams 249
17 GRASP: Designing Objects with Responsibilities 271
18 Object Design Examples with GRASP 321
19 Designing for Visibility 363
20 Mapping Designs to Code 369
21 Test-Driven Development and Refactoring 385
22 UML Tools and UML as Blueprint 395

PART IV ELABORATION ITERATION 2 — MORE PATTERNS

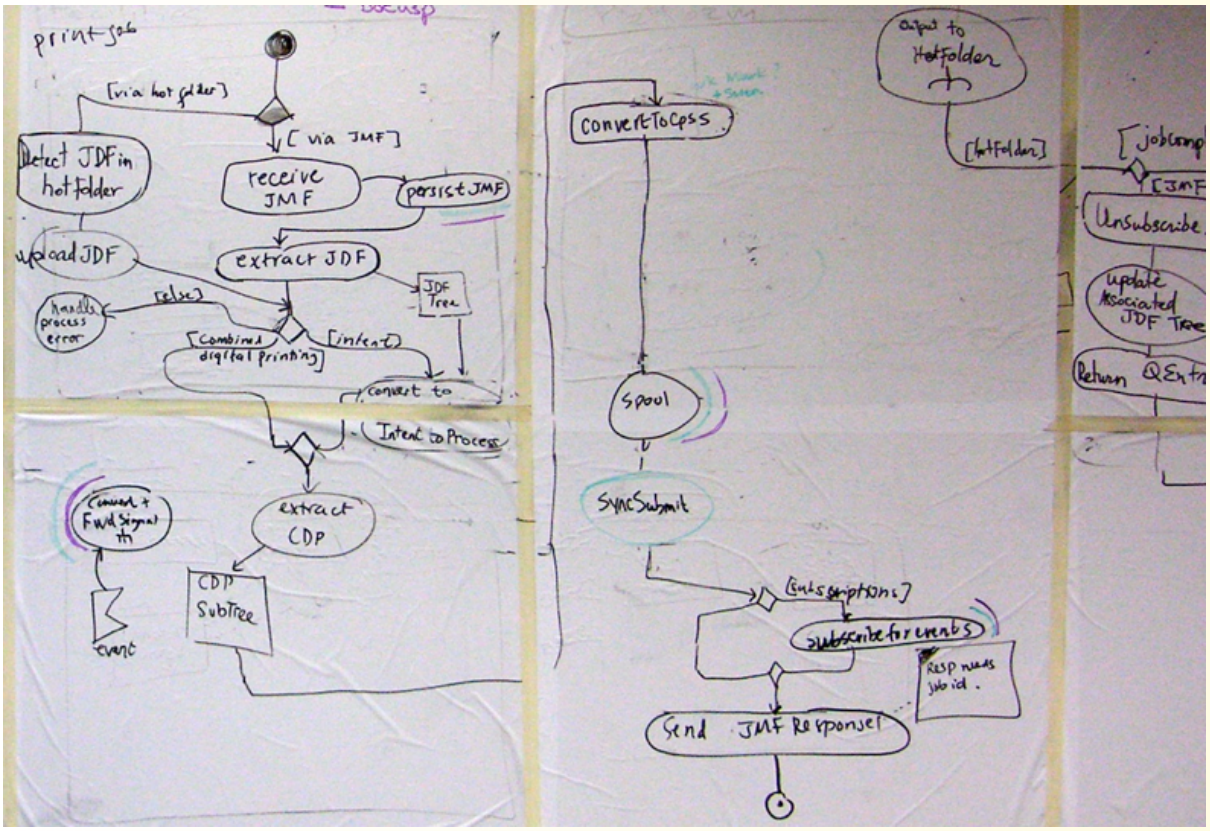
23 Iteration 2—More Patterns 401
24 Quick Analysis Update 407
25 GRASP: More Objects with Responsibilities 413
26 Applying GoF Design Patterns 435

PART V ELABORATION ITERATION 3 — INTERMEDIATE TOPICS

27 Iteration 3—Intermediate Topics 475
28 UML Activity Diagrams and Modeling 477
29 UML State Machine Diagrams and Modeling 485
30 Relating Use Cases 493
31 Domain Model Refinement 501
32 More SSDs and Contracts 535

Learn with Activity Diagrams

EXAMPLE: activity diagram



- useful both for...
- analysis of business or domain processes (what)
- design of algorithms (how)

- in this course, we will defer applying activity diagrams in analysis, though this can be useful
- we will apply them during design (for algorithms), and you will see how they could also apply in analysis

Learn with Specification by Example

- examples are a surprisingly powerful way to understand and probe into requirements

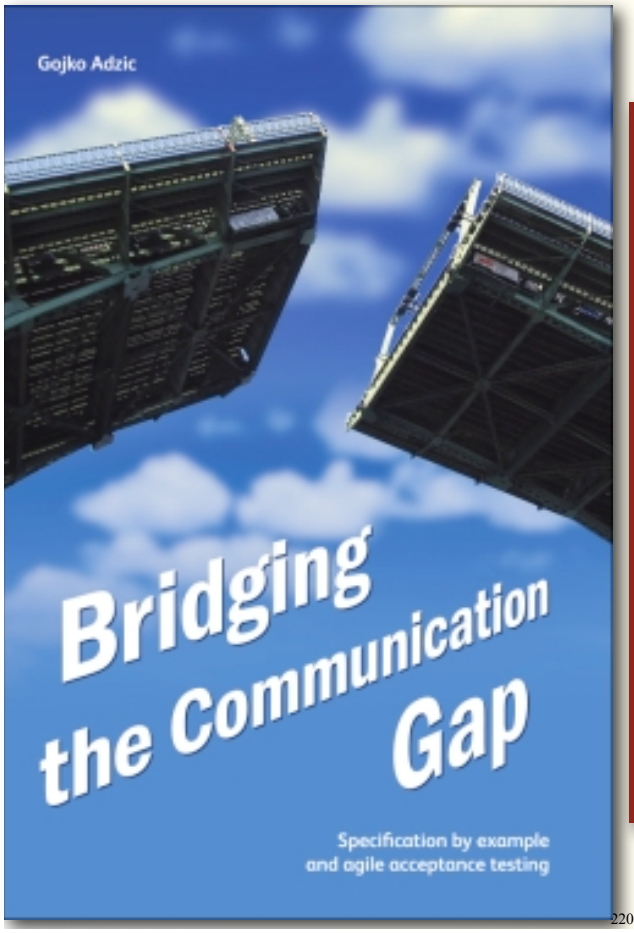
- specifying & learning by examples illustrates inductive reasoning -- a way of learning natural for children... until they are "de-educated" ;)

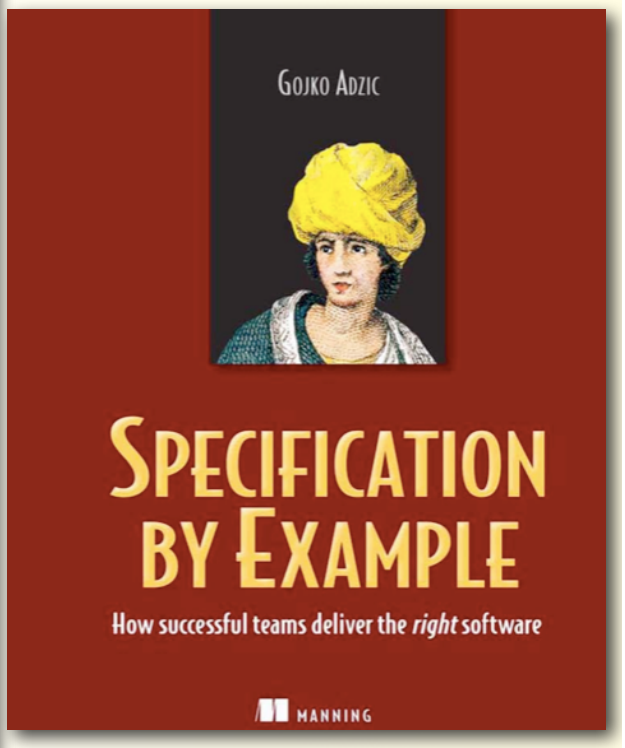
story =

Card +

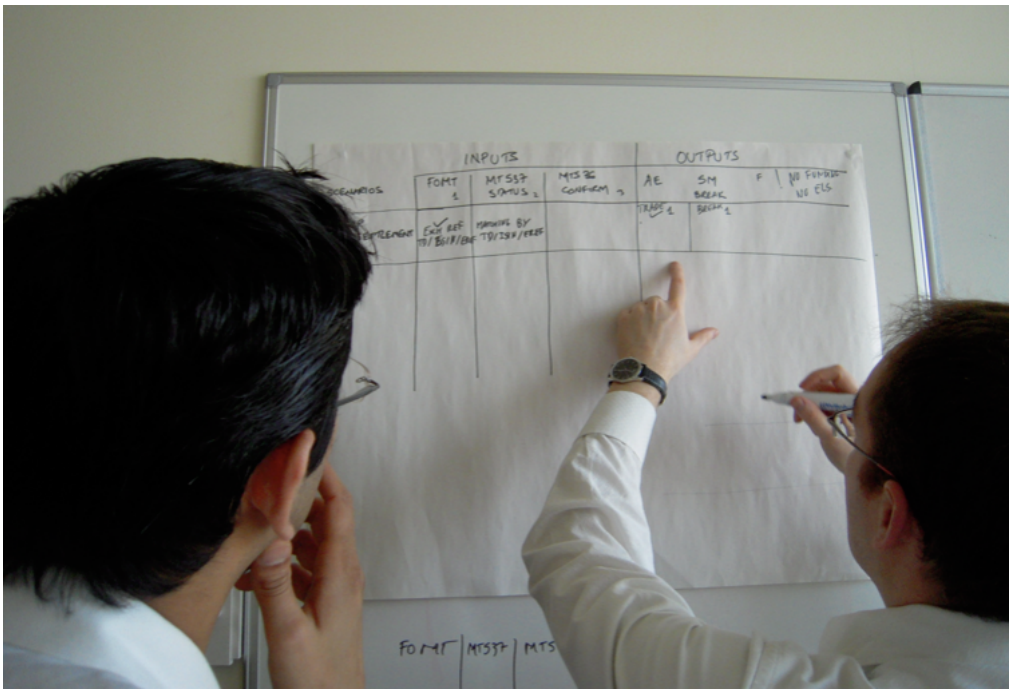
Conversation +

Confirmation





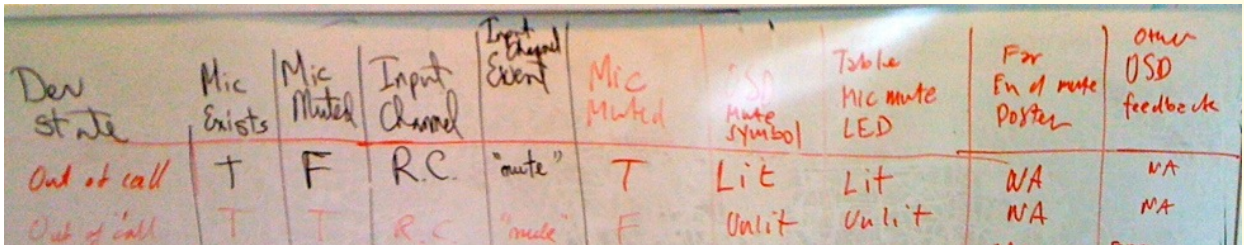
confirm by specification by example



videoconferencing system example

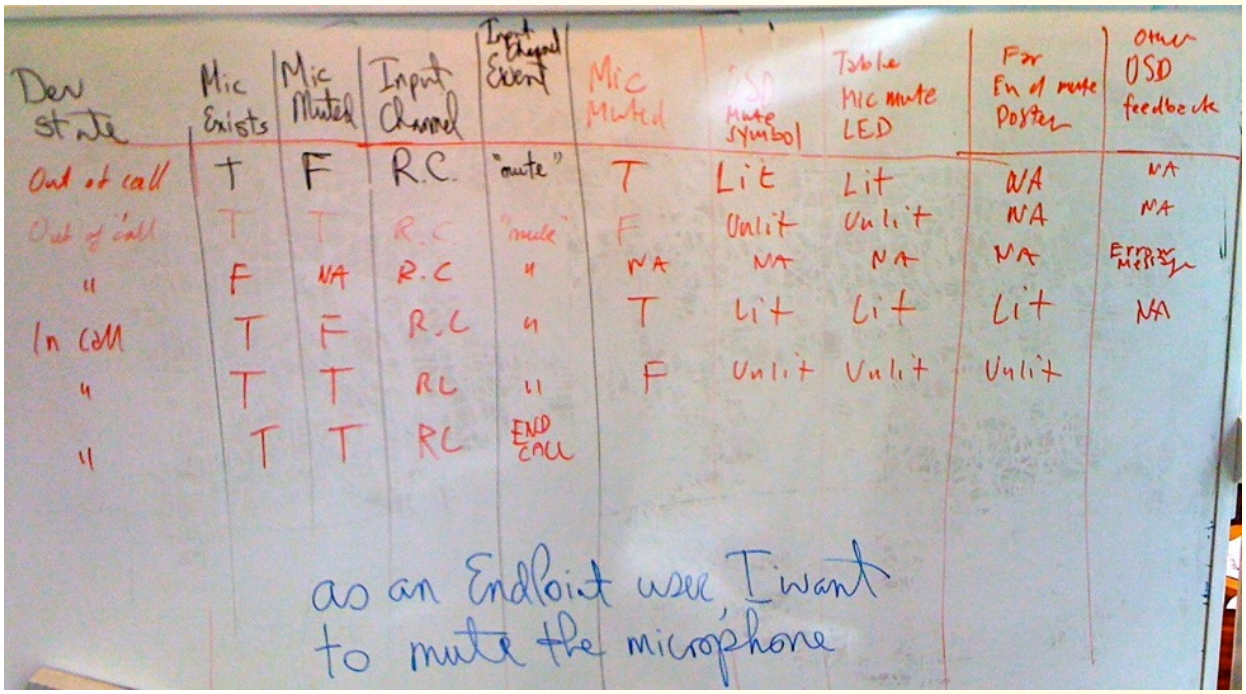
Dev state	Mic Exists	Mic Muted	Input Channel	Input Signal Event	Mic Muted	OSD mute symbol	Table Mic mute LED	For En of mute Poster	other OSD feedback
Out of call	T	F	R.C.	"mute"	T	Lit	Lit	NA	NA

videoconferencing system example



Dev state	Mic Exists	Mic Muted	Input Channel	Input Event	Mic Muted	OSD mute symbol	Table Mic mute LED	For En of mute Poster	Other OSD feedback
Out of call	T	F	R.C.	"mute"	T	Lit	Lit	NA	NA
Out of call	T	T	R.C	"mute"	F	Unlit	Unlit	NA	NA

videoconferencing system example



Dev state	Mic Exists	Mic Muted	Input Channel	Input Event	Mic Muted	OSD mute symbol	Table Mic mute LED	For En of mute Poster	Other OSD feedback
Out of call	T	F	R.C.	"mute"	T	Lit	Lit	NA	NA
Out of call	T	T	R.C	"mute"	F	Unlit	Unlit	NA	NA
"	F	NA	R.C	"	NA	NA	NA	NA	Error Message
In call	T	F	R.C	"	T	Lit	Lit	Lit	NA
"	T	T	RC	"	F	Unlit	Unlit	Unlit	
"	T	T	RC	END CALL					

as an Endpoint user, I want to mute the microphone

financial trading example

auto settlement tracking positions

A	B	C	D	E	F	G	H	I	J	K	L	
auto settlement tracking positions												
contractual trade amt	trade id	start date	security	Settlement Message	Now	contractual position	actual position		contractual position	actual position	comment	application log
1,000,000	A-RE-AAA-1234-1	11/28/2011	BTP3.5 Dec 2012	(tradeID=A-RE-AAA-1234-1, settled=y, settlementDate=Nov 28 2011, leg=S)	11/28/2011	(nov 28 2011, 1,000,000)	(nov 28 2011, 0)		(nov 28 2011, 1,000,000)	(nov 28 2011, 1,000,000) (nov 29, 2011, 1,000,000)	binary status – settled	add success m
1,000,000	A-RE-AAA-1234-1	11/28/2011	BTP3.5 Dec 2012		11/28/2011	(nov 28 2011, 1,000,000)	(nov 28 2011, 0)		(nov 28 2011, 1,000,000)	(nov 28 2011, 0)	no settlement	
1,000,000	A-RE-AAA-1234-1	11/28/2011	BTP3.5 Dec 2012	(tradeID=A-RE-AAA-1234-1, settled=n, settlementDate=Nov 28 2011,leg=S)	11/28/2011	(nov 28 2011, 1,000,000)	(nov 28 2011, 0)		(nov 28 2011, 1,000,000)	(nov 28 2011, 0)	binary status – failed	add failure msg

for what to create examples?

- stories/features, any level of granularity
  - specification by example on coarse-grained items helps later splitting into fine-grained items
- each system operation in an SSD

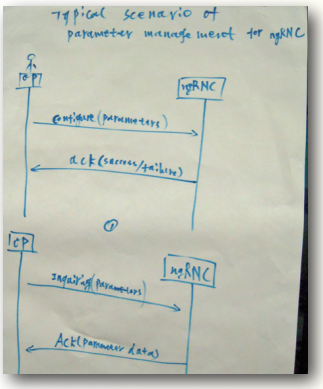
post-conditions or "expectations" in examples

- existence/state of observable output
  - messages, signals, parameters, data elements, ...
- existence/state of persistent things
  - DB, files, logs, ...
- existence/state of non-persistent in-memory things
  - software objects, alarms, timers, ...
- "URPS+" tests
  - duration, capacity, ...

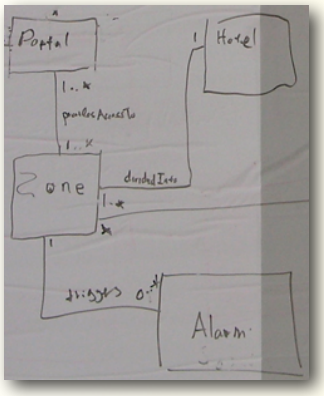
GUIDELINE: inspiration from prior models

- can prior models help us create the examples?

SSD



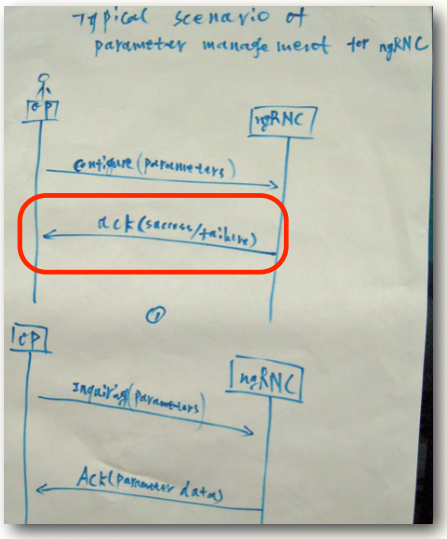
domain model



• • •

GUIDELINE: inspiration from the SSDs

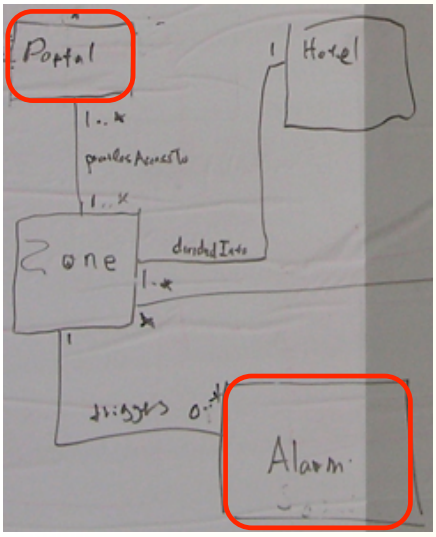
- the SSDs (system operations or output messages) may inspire us to think of useful expectations in the examples...



Test Case	EXPECTS Ack message	EXPECTS ?
1	yes	?
2	no	?

GUIDELINE: inspiration from domain model

- although the domain model is a purely conceptual (non-software) model, it may inspire us to think of useful expectations in the examples...



Test Case	EXPECTS Alarm.status	EXPECTS Portal.isOpen
1	signaling	TRUE
2	silent	FALSE

GUIDELINE: remember FURPS+ in examples

What are the Types and Categories of Requirements?

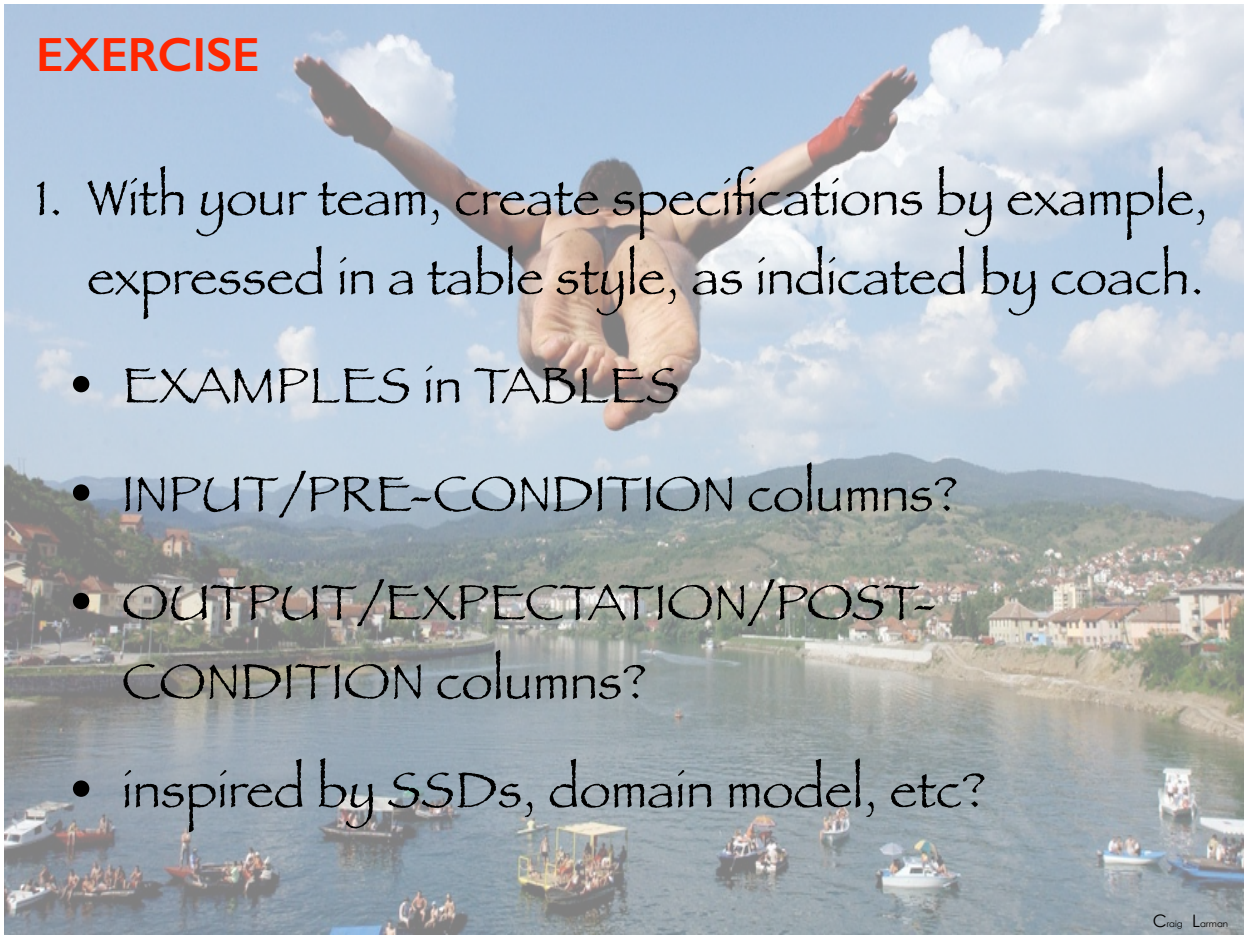
[Grady92], a useful mnemonic with the following meaning:[1]

- **Functional**—features, capabilities, security.
- **Usability**—human factors, help, documentation.
- **Reliability**—frequency of failure, recoverability, predictability.
- **Performance**—response times, throughput, accuracy, availability, resource usage.
- **Supportability**—adaptability, maintainability, internationalization, configurability.

The "+" in FURPS+ indicates ancillary and sub-factors, such as:

- **Implementation**—resource limitations, languages and tools, hardware, ...
- **Interface**—constraints imposed by interfacing with external systems.
- **Operations**—system management in its operational setting.
- **Packaging**—for example, a physical box.
- **Legal**—licensing and so forth.

EXERCISE



1. With your team, create specifications by example, expressed in a table style, as indicated by coach.
   - EXAMPLES in TABLES
   - INPUT/PRE-CONDITION columns?
   - OUTPUT/EXPECTATION/POST-CONDITION columns?
   - inspired by SSDs, domain model, etc?

Models to Have Conversations and Learn? ...

categories & examples of models

	analysis	design
static	story map use case dgm domain object model	design class dgm
dynamic	story map use case dgm stories low-fidelity UI mockups system sequence diagrams activity dgm of process specification by example state machine dgm of domain object	activity dgm of algorithm communication dgm

Acceptance TDD

how to build quality in?

how to reduce some of the wastes?

make the requirement language and the test language almost (or exactly) the same...

Robot Framework

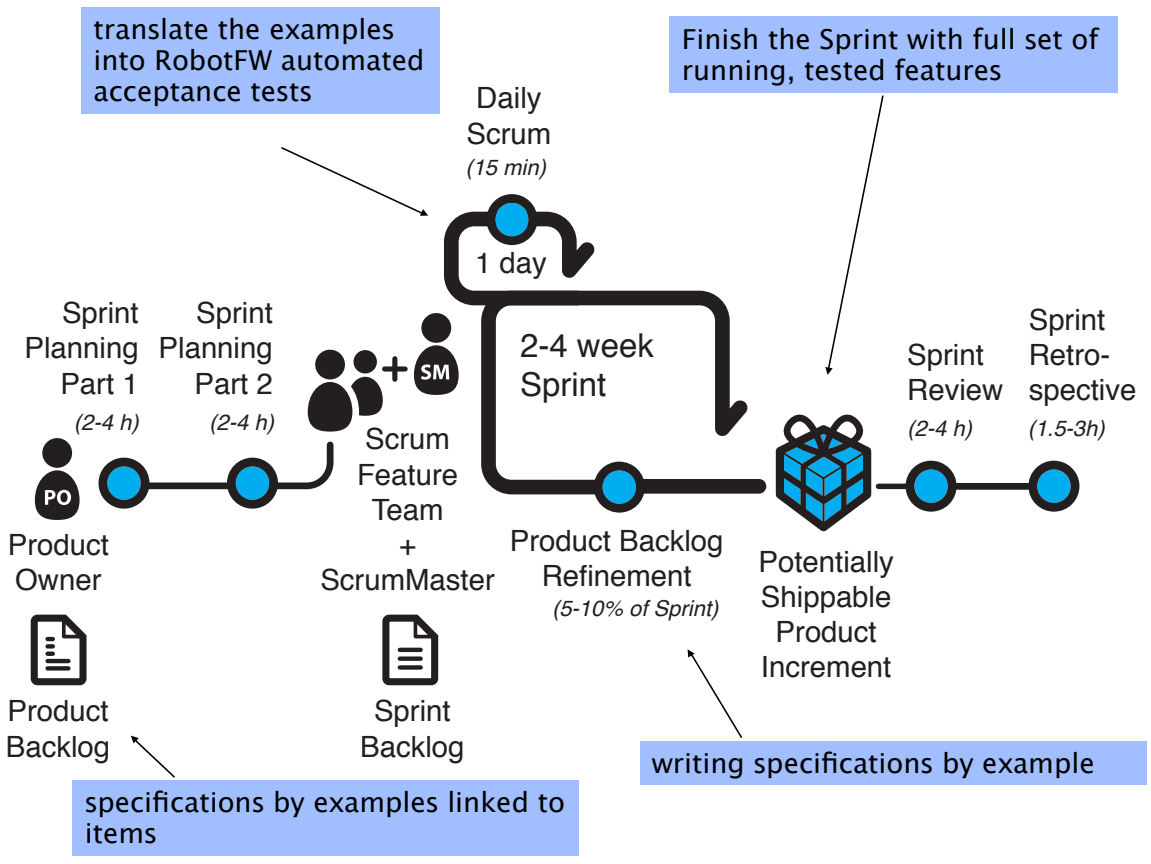
Examples = Specification = Requirement ≈ Automated Acceptance Tests!

has deep implications

	A	B	C	D	E
1	**admin of aggregations – disable aggregate computation**				
2	**aggId**	**adaptation entry**	**isLeaf**		**adaptation entry**
3	call_drop_hour	(..., isEnabled=true)	TRUE		(isEnabled=true)
4	call_succ_hour	(..., isEnabled=true)	FALSE		(isEnabled=false)

Robot Framework Automated Tests					
Test Case	**Action**	**In: aggregationID**	**In: adaptation entry - isEnabled**	**In: adaptation node: isLeaf**	**Post-condition: adaptation entry – isEnabled**
leaf node	disable aggregate computation	call_drop_hour	TRUE	TRUE	TRUE
non-leaf node	disable aggregate computation	call_succ_hour	TRUE	FALSE	FALSE

and, test FIRST...

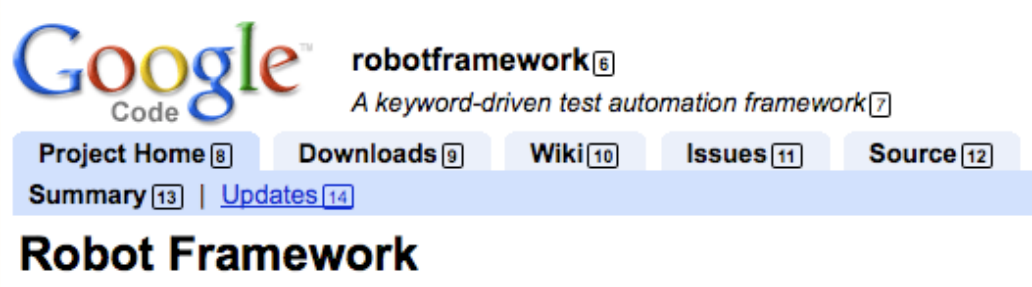


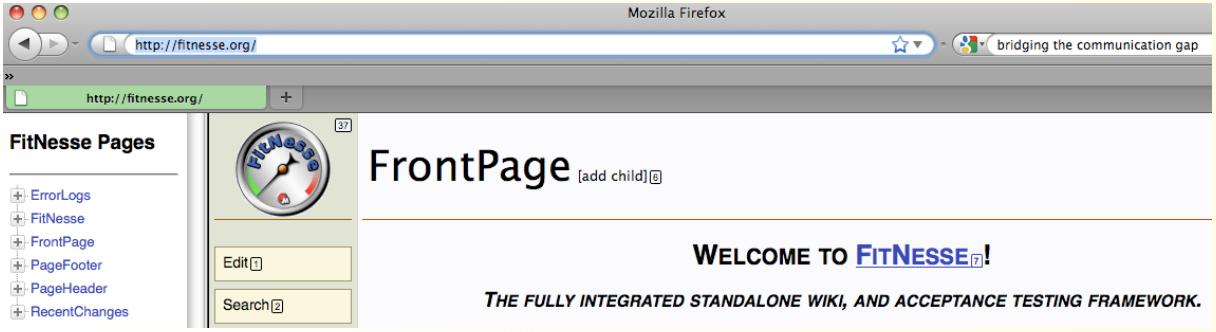
all that supports reducing...

A Lean List of Wastes (remove these)

1. Inventory; overproduction of features, or of elements ahead of the next step
2. Waiting, delay
3. Handoff, conveyance
4. Extra processing (includes extra processes), relearning
5. Work in progress (WIP); partially done work, (specifications not implemented, code not integrated or tested)
6. Task switching, motion between tasks, interrupt-driven multitasking
7. Defects, testing/inspection and correction at the end
8. Not using people's full potential: "working to job title," no multi-skill, no multi-learning, no kaizen, ...
9. Knowledge and information scatter or loss
10. Wishful thinking (e.g., plans and specifications are correct, that estimates can't increase)
11. blaming

to distill the examples to automated acceptance tests



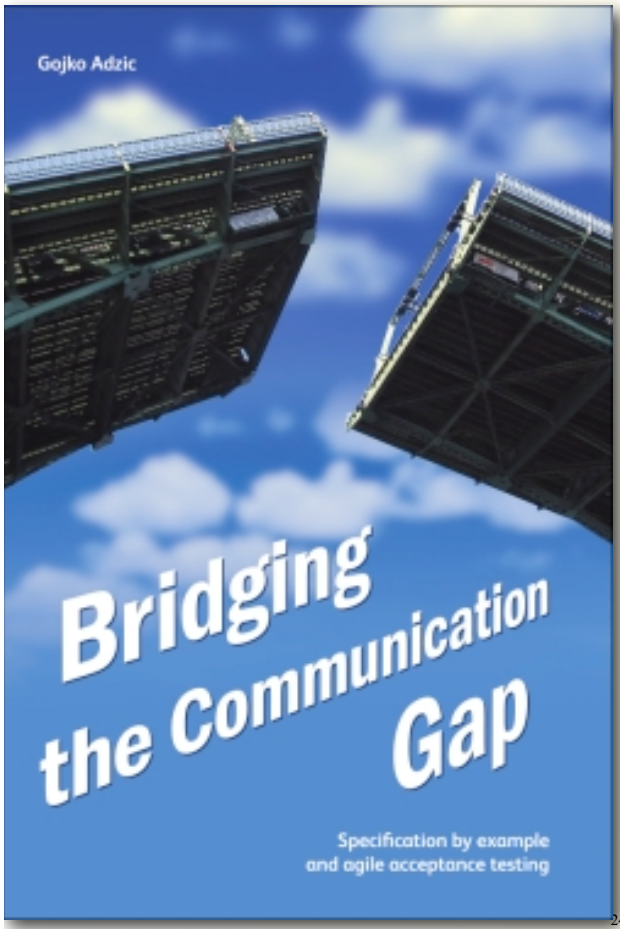


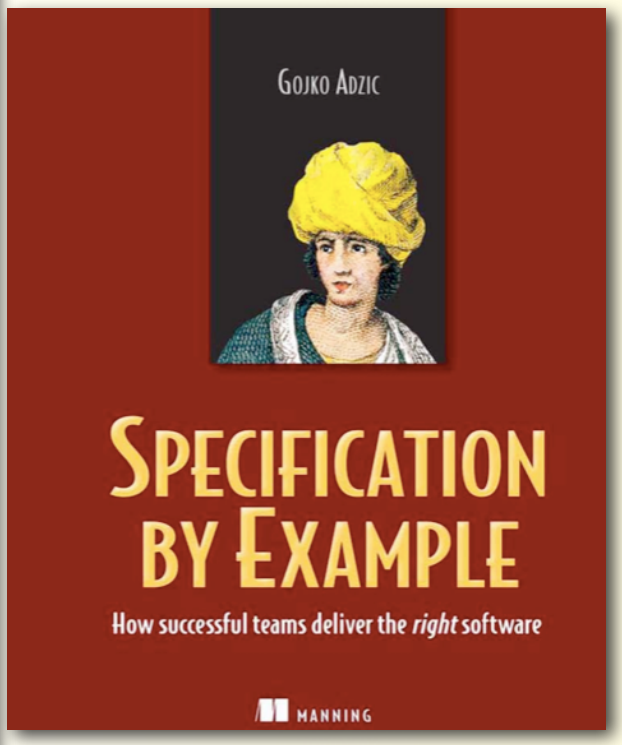
distilling

not every example needs to map to an executable test

not every column in an example is relevant

so, distill into a smaller representative set





BAD vs GOOD Automated Tests

what is the business requirement or goal here? ...

```
if (selenium.isTextPresent("Log in"))
{
 selenium.click("link=Log in");
 selenium.waitForPageToLoad(WellSphere.timeout);
 selenium.type("email", WellSphere.username);
 selenium.type("password", WellSphere.password);
 selenium.click ("//td[1]/table/tbody/tr/td/a/div[1]/div/div/b");
 selenium.waitForPageToLoad(WellSphere.timeout);
 assertTrue(selenium.isTextPresent("My Wellsphere"));
 System.out.println("Log In passed");
 System.out.println("User is logged in successfully");
}
```


contrast the previous with the clarity of goal here...

# of units to create	ETF fund CUSIP	NAV per unit	counter party info	shares per creation unit		op. type	settlement qty	settlement amt
1	00S000001	1234567.12	(0001, DTCYID)	50000		S	50000	1234567.12

how robust is the biz-goal test, if UI changes?

```
if (selenium.isTextPresent("Log in"))
{
 selenium.click("link=Log in");
 selenium.waitForPageToLoad(WellSphere.timeout);
 selenium.type("email", WellSphere.username);
 selenium.type("password", WellSphere.password);
 selenium.click ("//td[1]/table/tbody/tr/td/a/div[1]/div/div/b");
 selenium.waitForPageToLoad(WellSphere.timeout);
 assertTrue(selenium.isTextPresent("My Wellsphere"));
 System.out.println("Log In passed");
 System.out.println("User is logged in successfully");
}
```


contrast the previous with the UI-independence here...

# of units to create	ETF fund CUSIP	NAV per unit	counter party info	shares per creation unit		op. type	settlement qty	settlement amt
1	00S000001	1234567.12	(0001, DTCYID)	50000		S	50000	1234567.12

can business people understand this? biz language?

```
if (selenium.isTextPresent("Log in"))
{
 selenium.click("link=Log in");
 selenium.waitForPageToLoad(WellSphere.timeout);
 selenium.type("email", WellSphere.username);
 selenium.type("password", WellSphere.password);
 selenium.click ("//td[1]/table/tbody/tr/td/a/div[1]/div/div/b");
 selenium.waitForPageToLoad(WellSphere.timeout);
 assertTrue(selenium.isTextPresent("My Wellsphere"));
 System.out.println("Log In passed");
 System.out.println("User is logged in successfully");
}
```


contrast the previous with business understanding...

# of units to create	ETF fund CUSIP	NAV per unit	counter party info	shares per creation unit		op. type	settlement qty	settlement amt
1	00S000001	1234567.12	(0001, DTCYID)	50000		S	50000	1234567.12

look/read just like the business requirement?

```
if (selenium.isTextPresent("Log in"))
{
 selenium.click("link=Log in");
 selenium.waitForPageToLoad(WellSphere.timeout);
 selenium.type("email", WellSphere.username);
 selenium.type("password", WellSphere.password);
 selenium.click ("//td[1]/table/tbody/tr/td/a/div[1]/div/div/b");
 selenium.waitForPageToLoad(WellSphere.timeout);
 assertTrue(selenium.isTextPresent("My Wellsphere"));
 System.out.println("Log In passed");
 System.out.println("User is logged in successfully");
}
```


contrast previous with connection to biz requirement...

# of units to create	ETF fund CUSIP	NAV per unit	counter party info	shares per creation unit		op. type	settlement qty	settlement amt
1	00S000001	1234567.12	(0001, DTCYID)	50000		S	50000	1234567.12

can we write this test *before* the UI exists?

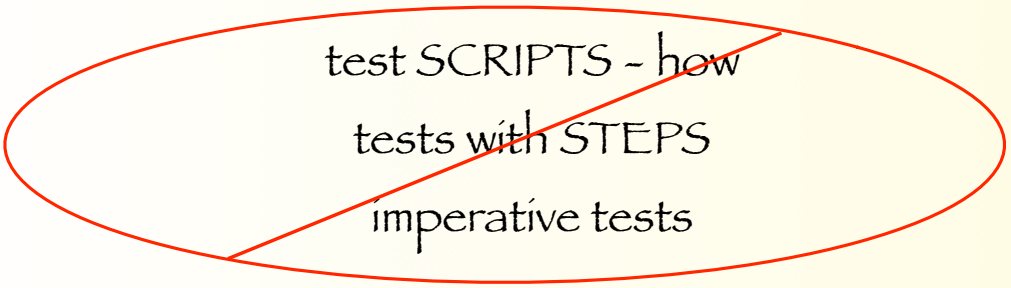
```
if (selenium.isTextPresent("Log in"))
{
 selenium.click("link=Log in");
 selenium.waitForPageToLoad(WellSphere.timeout);
 selenium.type("email", WellSphere.username);
 selenium.type("password", WellSphere.password);
 selenium.click ("//td[1]/table/tbody/tr/td/a/div[1]/div/div/b");
 selenium.waitForPageToLoad(WellSphere.timeout);
 assertTrue(selenium.isTextPresent("My Wellsphere"));
 System.out.println("Log In passed");
 System.out.println("User is logged in successfully");
}
```


can we write this test *before* the UI exists?

# of units to create	ETF fund CUSIP	NAV per unit	counter party info	shares per creation unit		op. type	settlement qty	settlement amt
1	00S000001	1234567.12	(0001, DTCYID)	50000		S	50000	1234567.12

Craig Larman

DECLARATIVE automated tests



declarative - what - intentional - biz rule

Test Case	Action	IN: operation	IN: unit info	EXPECTS: rest channel state	EXPECTS: unit state	EXPECTS: channel to CACU
Remove a channel	Configure channel	remove	SP-CACU	open	TE-TEX	removed
Add a chanel	Configure channel	add	SP-CACUn	open	TE-TEX	open

1-line declarative "what" acceptance tests

# of units to create	ETF fund CUSIP	NAV per unit	counter party info	shares per creation unit		op. type	settlement qty	settlement amt
1	00S000001	1234567.12	(0001, DTCYID)	50000		S	50000	1234567.12

we want the requirement and test almost EQUAL

specification by example

	A	B	C	D	E
1	**admin of aggregations – disable aggregate computation**				
2	**aggId**	**adaptation entry**	**isLeaf**		**adaptation entry**
3	call_drop_hour	(..., isEnabled=true)	TRUE		(isEnabled=true)
4	call_succ_hour	(..., isEnabled=true)	FALSE		(isEnabled=false)

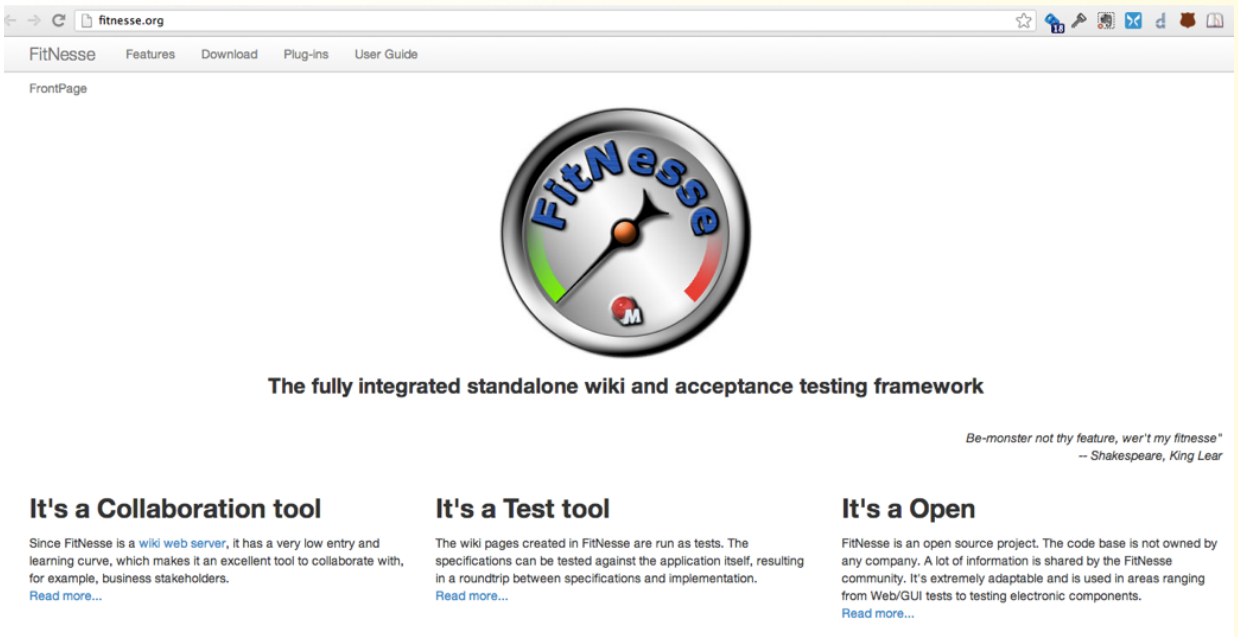
automated acceptance test

Robot Framework Automated Tests					
Test Case	**Action**	**In: aggregationID**	**In: adaptation entry - isEnabled**	**In: adaptation node: isLeaf**	**Post-condition: adaptation entry – isEnabled**
leaf node	disable aggregate computation	call_drop_hour	TRUE	TRUE	TRUE
non-leaf node	disable aggregate computation	call_succ_hour	TRUE	FALSE	FALSE

avoid describing or testing business/domain rules or logic through user interfaces

apply as much "software engineering" skill in the design and maintenance of the automated tests, as to the production code, to minimize all the "code smells" (duplication, hard-coded constants, ...)

Fitnesse



GUIDELINE: embed text and pictures

- living documentation, executable requirements
- important! ... Fitnesse can (and should) contain text and pictures for humans to read, in addition to the executable table-tests
  - Fitnesse page = elaboration of requirement for humans + the precise tests
  - reduce need for other documents
- ...

GUIDELINE: embed text and pictures. LIVING DOCUMENTATION

FitNesse > UserGuide

TwoMinuteExample

Failure Navigator < of 1 >

Tests Executed OK | Test | Tools

Assertions: 5 right, 1 wrong, 0 ignored, 0 exceptions (0.003 seconds)

Precompiled Libraries | Expand All | Collapse All

A One-Minute Description

An Example FitNesse Test

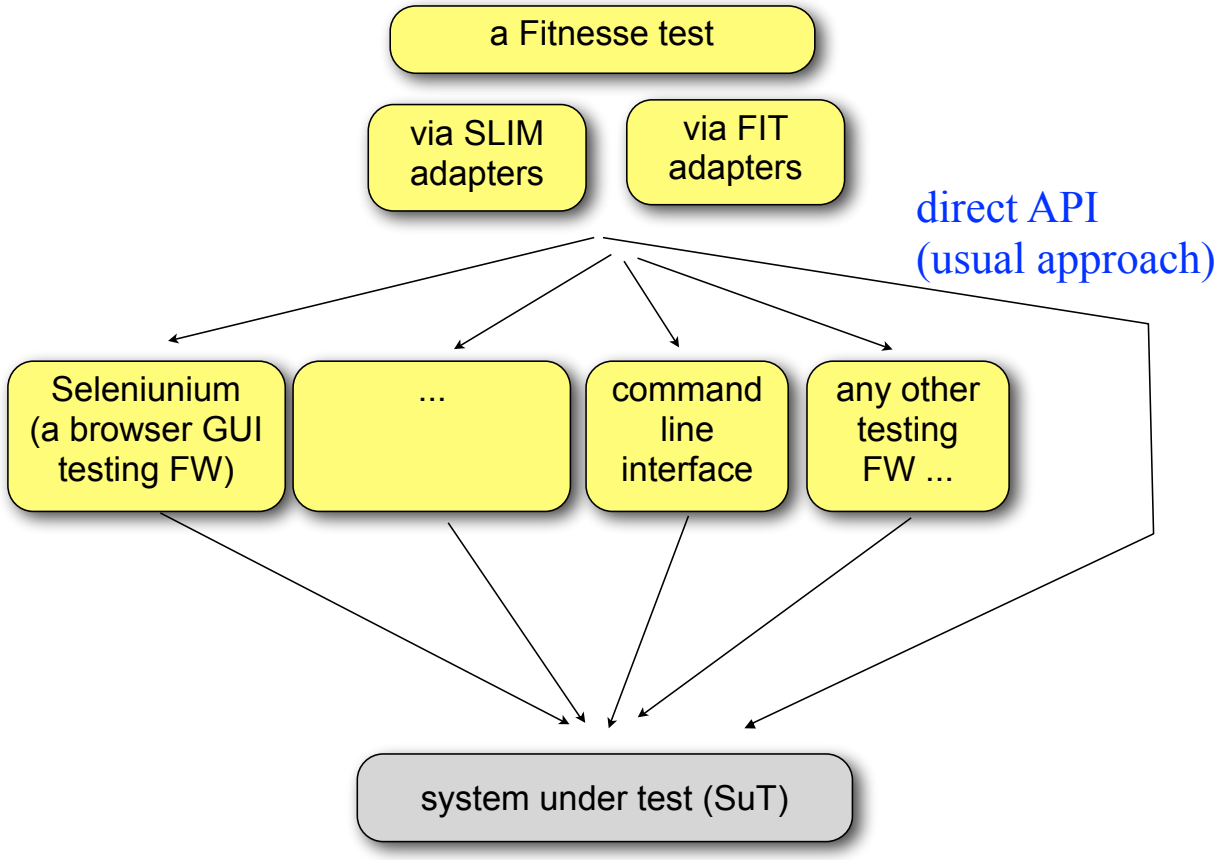
If you were testing the division function of a calculator application, you might like to see some examples working. You might want to see what you get back if you ask it to divide 10 by 2. (You might be hoping for a 5!)
In FitNesse, tests are expressed as tables of **input** data and **expected output** data. Here is one way to specify a few division tests in FitNesse:

eg.Division		
numerator	denominator	quotient?
10	2	5.0
12.6	3	4.2
22	7	3.142857142857143~=3.14
9	3	3.0<5
11	2	4<5.5<6
100	4	[25.0] expected [33]

This style of FitNesse test table is called a Decision Table, each row represents a complete scenario of example inputs and outputs. Here, the "numerator" and "denominator" columns are for inputs, and the question mark in the "quotient?" column tells FitNesse that this is our column of expected outputs. Notice our "10/2 = 5.0" scenario. Try reading it as a question: *"If I give you a numerator of 10 and denominator of 2, do I get back a 5?"*

Running our test table: Click the Test button

Before we do another thing, let's run this test table. See the little blue and white **Test** button in the upper-left, just below the FitNesse logo? Click it and see what happens.



after installation...

```
(first, rename fitnesse-standalone.jar)

>java -jar fitnesse.jar -p 8080
```


fixtures (adapters) to connect to the SUT

- for example...

example SLIM decision table

Foo1Test

Tests Executed OK | Test | Edit

Assertions: 1 right, 1 wrong, 0 ignored, 0 exceptions (0.001 seconds)

Precompiled Libraries

variable defined: TEST_SYSTEM=slim
classpath: /Users/dcl/Documents/workspace/temp1/bin

com.craiglarman.dicegame.domain.DiceGameFixture		
in1	in2	expected1?
1	2	[3] expected [4]
2	3	3

example SLIM decision table

```
!define TEST_SYSTEM {slim}
!path /Users/dcl/Documents/workspace/temp1/bin

| com.craiglarman.dicegame.domain.DiceGameFixture |
| in1 | in2 | expected1? |
| 1   | 2   | 4          |
| 2   | 3   | 3          |
```

example SLIM fixture — shows "connecting the dots"

```
package com.craiglarman.dicegame.domain;

public class DiceGameFixture {
private int in1;
private int in2;

public void setIn1(int in1) {
   this.in1 = in1;
}

public void setIn2(int in2) {
   this.in2 = in2;
}

public int expected1() {
   return 3;
}
}
```


SLIM Fixture Elements — use the EXECUTE method

```
public class SomeFixture {

public void setAttribute1(…) {
   // …
}


// automatically called by SLIM runner
public void execute() {
   // 1. main call to do the system work
   // 2. (optional) collect data for "expectations"
}


public int expected1() {
   return ???;
}
}
```

via ATTD, defining the API for your SuT!

```
public class SomeFixture {

public void setAttribute1(…) {
   // …
}


public void execute() {
  reconciliationSystem.reconcile(attribute1, …);
}


public int countOfFailures() {
   return reconciliationSystem.getCountOfFailures();
}
}
```


immediately define & call the API for your SuT

Discover & write the calls to the system under test immediately. Do not defer making these calls. At first, since the bodies of these methods will be empty, it should cause the Fitnesse test to fail — which is what we want!

```
public class SomeFixture {

public void setAttribute1(…) {
   // …
}


public void execute() {
  reconciliationSystem.reconcile(attribute1, …);
}


public int countOfFailures() {
   return reconciliationSystem.getCountOfFailures();
}
}
```

living documentation!

Foo1Test

Test Edit

variable defined: TEST_SYSTEM=slim
classpath: /Users/dcl/Documents/workspace/temp1/bin

com.craiglarman.dicegame.domain.DiceGameFixture		
in1	in2	expected1?
1	2	4
2	3	3

This is the wonderful dice game!

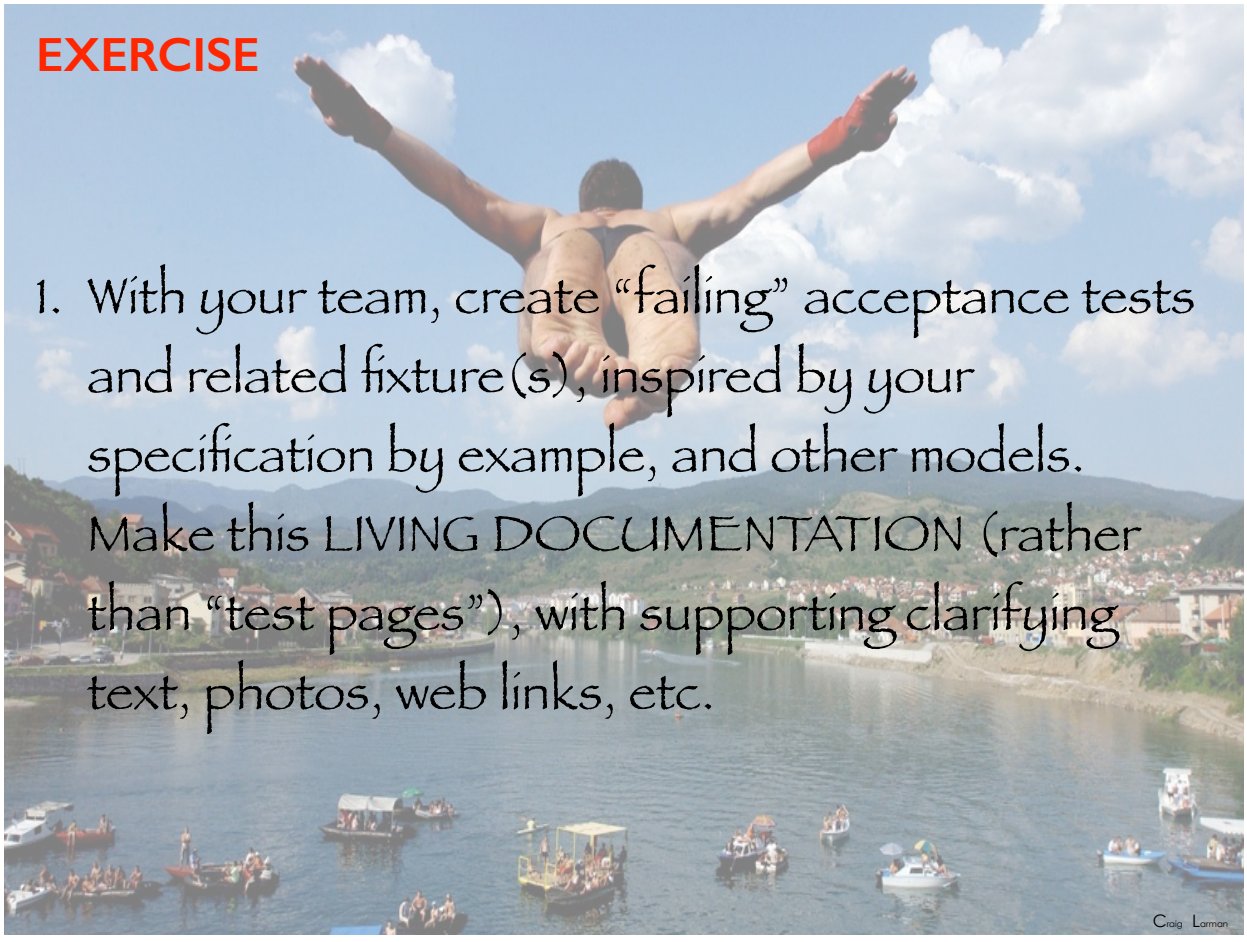
example

variable defined: TEST_SYSTEM=slim
classpath: /Users/dcl/Documents/workspace/temp1/bin

com.craiglarman.dicegame.domain.DiceGameFixture		
in1	in2	expected1?
1	2	[3] expected [4]
2	3	3

```
package com.craiglarman.dicegame.domain;

public class DiceGameFixture {
 private int in1;
 private int in2;

 public void setIn1(int in1) {
    this.in1 = in1;
 }

 public void setIn2(int in2) {
    this.in2 = in2;
 }

 public int expected1() {
    return 3;
 }
}
```

ATDD Exercise



EXERCISE

1. With your team, create "failing" acceptance tests and related fixture(s), inspired by your specification by example, and other models. Make this LIVING DOCUMENTATION (rather than "test pages"), with supporting clarifying text, photos, web links, etc.

Craig Larman

Transition to Design

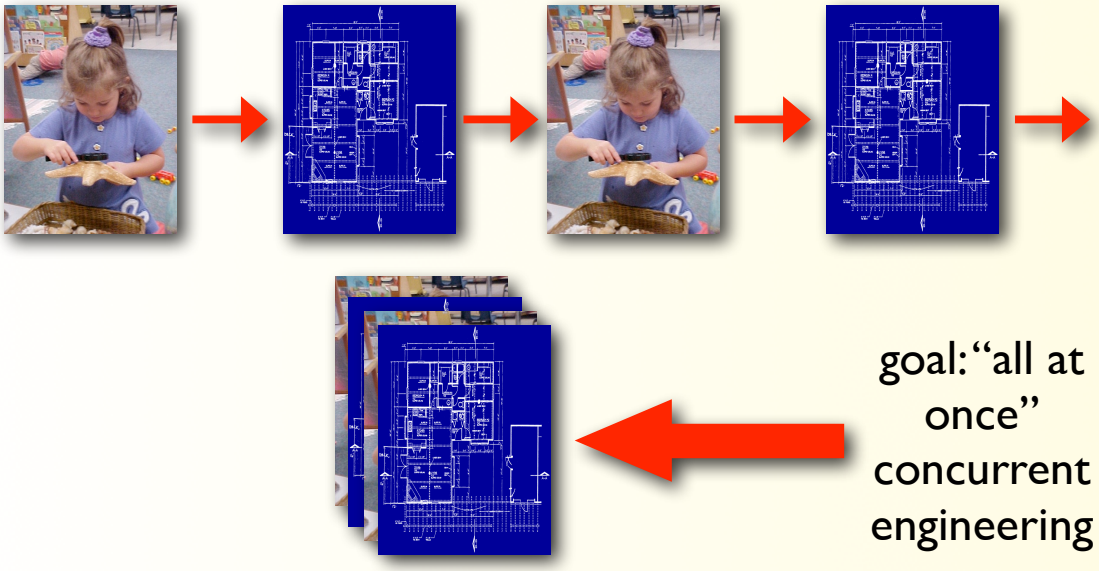
now we make a major shift from exploring requirements to exploring design

the story until now...

Topics

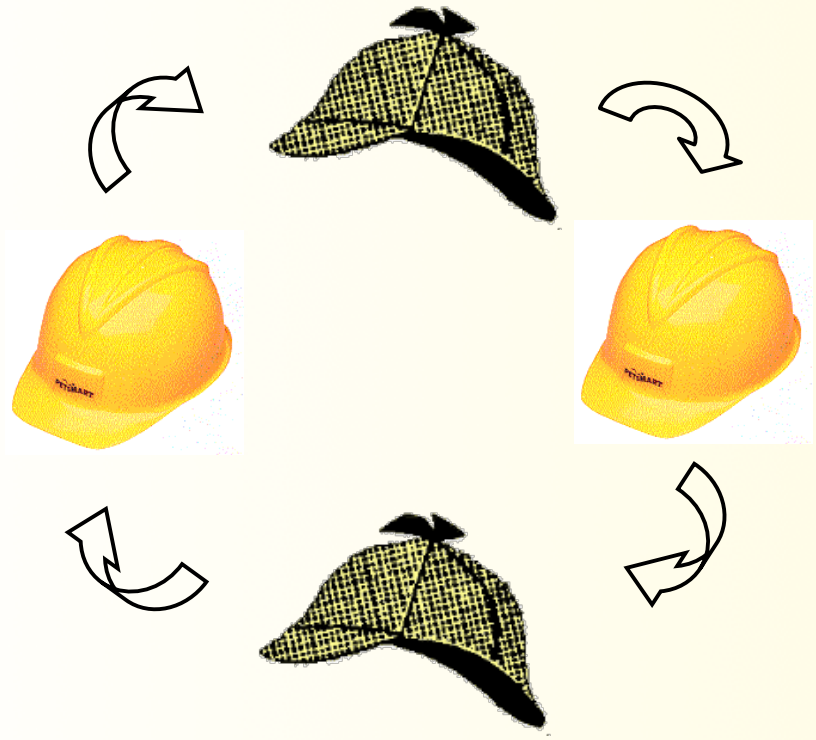
- Lean & Agile Modeling
- Lean & Agile Requirements
- User-Centered Design
- Story Mapping
- Telling Stories
- Splitting Features
- Learn with System-Sequence Diagrams
- Learn with a Domain Model
- Learn with Specification by Example
- Acceptance TDD
- BAD vs GOOD Automated Tests
- Designing with Layers
- Create Algorithms with Activity Diagrams
- GRASPing Object Design Principles
- Create Object Design with Communication Diagrams
- Create Object Design with Design Class Diagrams
- Continuous Integration
- Create Code Driven by Unit Tests (Unit TDD)
- Lean Thinking
- Lean Software Development
- Agile Values & Principles
- SOLID Principles
- Generalization
- Polymorphism & Class Hierarchy Design
- Feature Toggle Patterns
- Simple Patterns (Simple Factory, Null Object, ...)

analysis & design, iteratively & concurrently each iteration...hour...minute



goal: "all at once" concurrent engineering

people & team changes hats, iteratively



Designing with Layers

layers

- GUI windows
- reports
- speech interface
- HTML, XML, XSLT, JSP, Javascript, ...

Presentation
(AKA Interface, UI, View)

- handles presentation layer requests
- workflow
- session state
- window/page transitions
- consolidation/transformation of disparate data for presentation

Application
(AKA Workflow, Process, Mediation, App Controller)

- handles application layer requests
- implementation of domain rules
- domain services (· *POS* · , · *Inventory* ·)
  - services may be used by just one application, but there is also the possibility of multi-application services

Domain(s)
(AKA Business, Business Services, Model)

- very general low-level business services used in many business domains
- *CurrencyConverter*

Business Infrastructure
(AKA Low-level Business Services)

- (relatively) high-level technical services · · and frameworks
- *Persistence* · , · *Security*

Technical Services
(AKA Integration, Technical Infrastructure, High-level Technical Services)

- low-level technical services, utilities, · · and frameworks
- *data structures, threads, math,* · · *file, DB, and network I/O*

Foundation
(AKA Core Services, Base Services, Low-level Technical Services/Infrastructure)

more app specific

dependency

width implies range of applicability

Craig Larman

design with layers

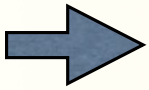
maintain a separation of concerns (high cohesion)

Create & Collaborate

via

Agile Design Modeling

acceptance tests



activity diagrams for algorithms

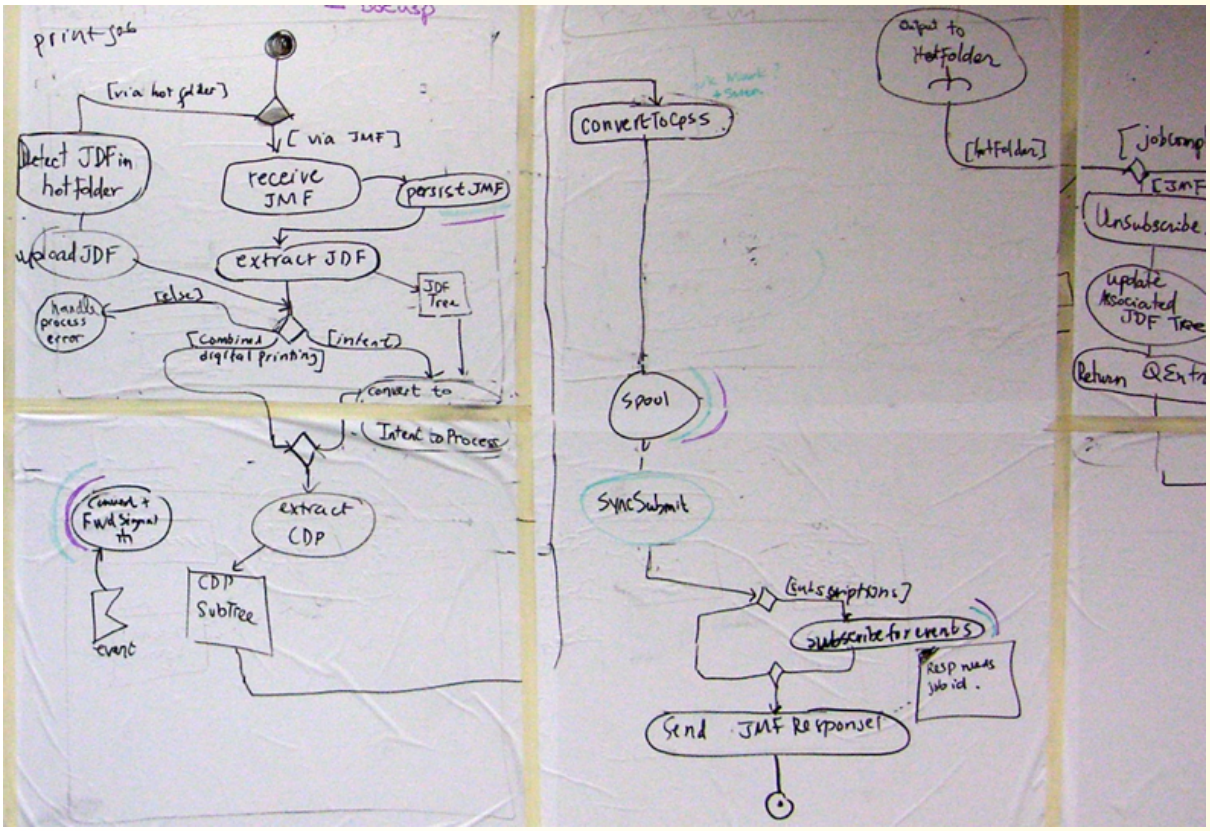
communication diagrams

design class diagrams

Create Algorithms with Activity Diagrams

EXAMPLE: activity diagram

EXAMPLE: activity diagram

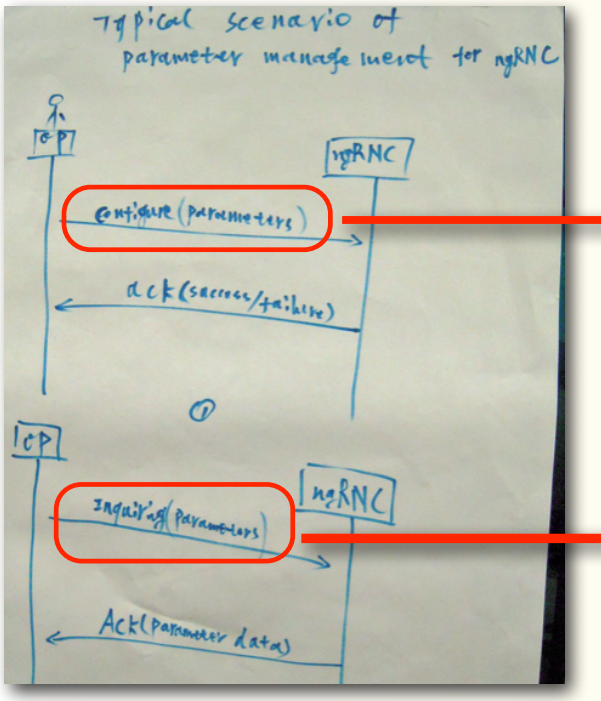


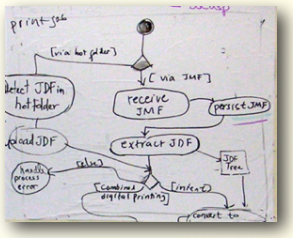
activity diagrams

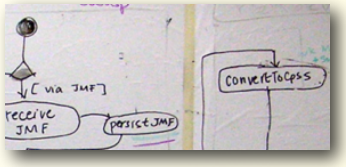
- useful both for...
- analysis of business or domain processes (what)
- design of algorithms (how)

GUIDELINE: consider for each system operation

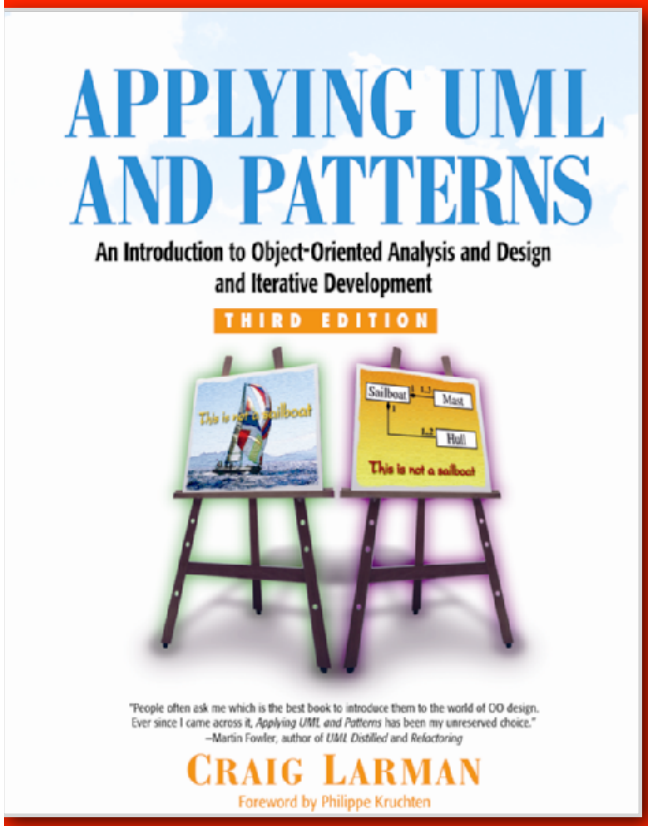
SSDs identify each system operation for which we need to design a solution







EXERCISE



1 Object-Oriented Analysis and Design 3
2 Iterative, Evolutionary, and Agile 17
3 Case Studies 41

PART II INCEPTION

4 Inception is Not the Requirements Phase 47
5 Evolutionary Requirements 53
6 Use Cases 61
7 Other Requirements 101

PART III ELABORATION ITERATION 1 — BASICS

8 Iteration 1—Basics 123
9 Domain Models 131
10 System Sequence Diagrams 173
11 Operation Contracts 181
12 Requirements to Design—Iteratively 195
13 Logical Architecture and UML Package Diagrams 197
14 On to Object Design 213
15 UML Interaction Diagrams 221
16 UML Class Diagrams 249
17 GRASP: Designing Objects with Responsibilities 271
18 Object Design Examples with GRASP 321
19 Designing for Visibility 363
20 Mapping Designs to Code 369
21 Test-Driven Development and Refactoring 385
22 UML Tools and UML as Blueprint 395

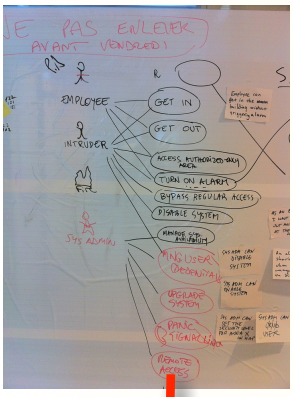
PART IV ELABORATION ITERATION 2 — MORE PATTERNS

23 Iteration 2—More Patterns 401
24 Quick Analysis Update 407
25 GRASP: More Objects with Responsibilities 413
26 Applying GoF Design Patterns 435

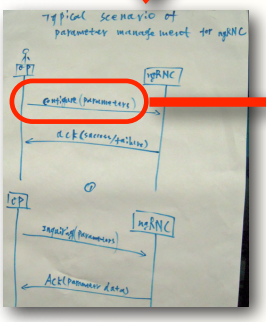
PART V ELABORATION ITERATION 3 — INTERMEDIATE TOPICS

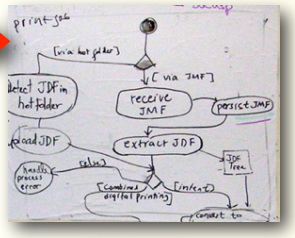
27 Iteration 3—Intermediate Topics 475
28 UML Activity Diagrams and Modeling 477
29 UML State Machine Diagrams and Modeling 485
30 Relating Use Cases 493
31 Domain Model Refinement 501
32 More SSDs and Contracts 535
33 Architectural Analysis 541
34 Logical Architecture Refinement 559
35 Package Design 579
36 More Object Design with GoF Patterns 587
37 Designing a Persistence Framework with Patterns 621
38 UML Deployment and Component Diagrams 651
39 Documenting Architecture: UML & the N+1 View Model 655

where are we?...



Outside-In Development





categories & examples of models

	analysis	design
static	story map use case dgm domain object model	design class dgm
dynamic	story map use case dgm stories low-fidelity UI mockups system sequence diagrams activity dgm of process specification by example state machine dgm of domain object	activity dgm of algorithm communication dgm

GRASPing Object Design Principles

skillful object design

- knowing diagrams != knowing good design
- we need key principles for agile object design
- then we can sketch -- creating a skillful design
- what are some of these principles? ...

the GRASP core object-design principles

1. Information Expert
2. Creator
3. Controller
4. Low Coupling
5. High Cohesion
6. Polymorphism
7. Pure Fabrication
8. Indirection
9. Protected Variations

applying these is useful for good and agile object design

1. Information Expert

- Most basic, general principle of object design?
- "put services with data" (... needed for the services)
- variation: "tell, don't ask"
- find the "information expert" that has this data
- "classic" abstract data type design

1.Information Expert
2.Creator
3.Controller
4.Low Coupling
5.High Cohesion
6.Polymorphism
7.Pure Fabrication
8.Indirection
9.Protected Variations

1. Information Expert and "tell, don't ask"

- variation: tell, don't ask - and tell in ONE call
- avoid queries; use "commands"
- decisions/actions based entirely upon the state of object X should be made 'inside' object X

avoid

```
class Message {
 Header header;
 Body body;

 String toString() {
  if ( body.isEncrypted() )
      body.decrypt();
  return
    header.toString() +
    body.toString();
}
```

better

```
class Message {
 Header header;
 Body body;

 String toString() {
  return
    header.toString() +
    body.toString();
}
```

1. Information Expert and "tell, don't ask"

- tell, don't ask is a good heuristic, but there are times that a query (ask) is still needed
- queries may be OK if they don't violate "decisions/ actions based upon state of X should be made 'inside' X"

OK

```
class Message {
 Header header;
 Body body;

 void send() {
  if ( header.isVerbose() )
      body.compress();

  ...
}
```

Even Better?

```
...
void send() {
  // alternatives:
  body.compress( header );
  body.compressDependingOn(
                header );
  body.compress();
  header.compress( body );

  ...
}
```


1. Information Expert and "tell, don't ask"

- variation: tell, don't ask
- since it implies avoid queries on an object, if followed "dogmatically", it would create artificially "strange" low-cohesion un-reusable objects; it is a guideline, not a rule

```
x = myList.get(3);      // not allowed??

x = die.getFaceValue();// not allowed??
```

2. Creator

- Most basic object creation guideline?
- Choose an object C (the Creator) to create X, when:
  - C contains or aggregates X
  - C closely uses X
  - C has the initializing data for X
- The more, the better.

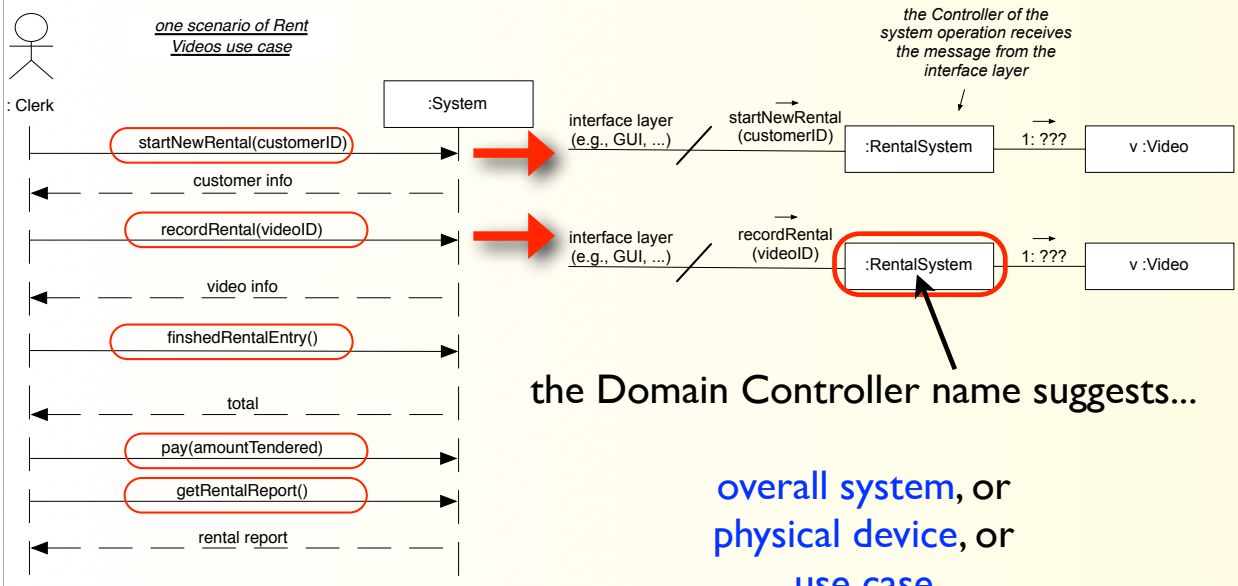
1.Information Expert
2.Creator
3.Controller
4.Low Coupling
5.High Cohesion
6.Polymorphism
7.Pure Fabrication
8.Indirection
9.Protected Variations

3. Controller (AKA Domain Controller or Domain 'Facade')

- At the "top" of the "domain" layer, (1) add a front-end "facade object" that receives system operation messages from the interface layer, and (2) consider a naming convention such as...
- the overall "system" or "root object"
  - RentalSystem, Game
- the device (if embedded software)
  - Phone, Printer
- the use case name
  - <UsecaseName>Handler
  - ManageNetworkAddressesHandler

1.Information Expert
2.Creator
3.Controller
4.Low Coupling
5.High Cohesion
6.Polymorphism
7.Pure Fabrication
8.Indirection
9.Protected Variations

3. (Domain) Controller -- example



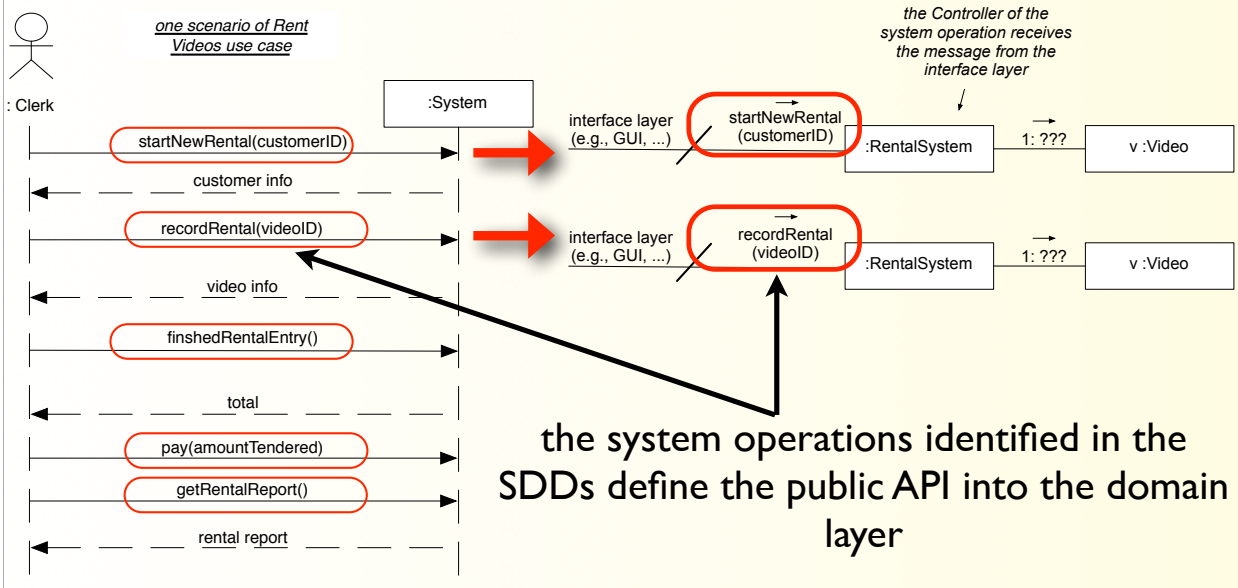
the Domain Controller name suggests...

overall system, or
physical device, or
use case

SSD with system operations

communication diagrams for system operations

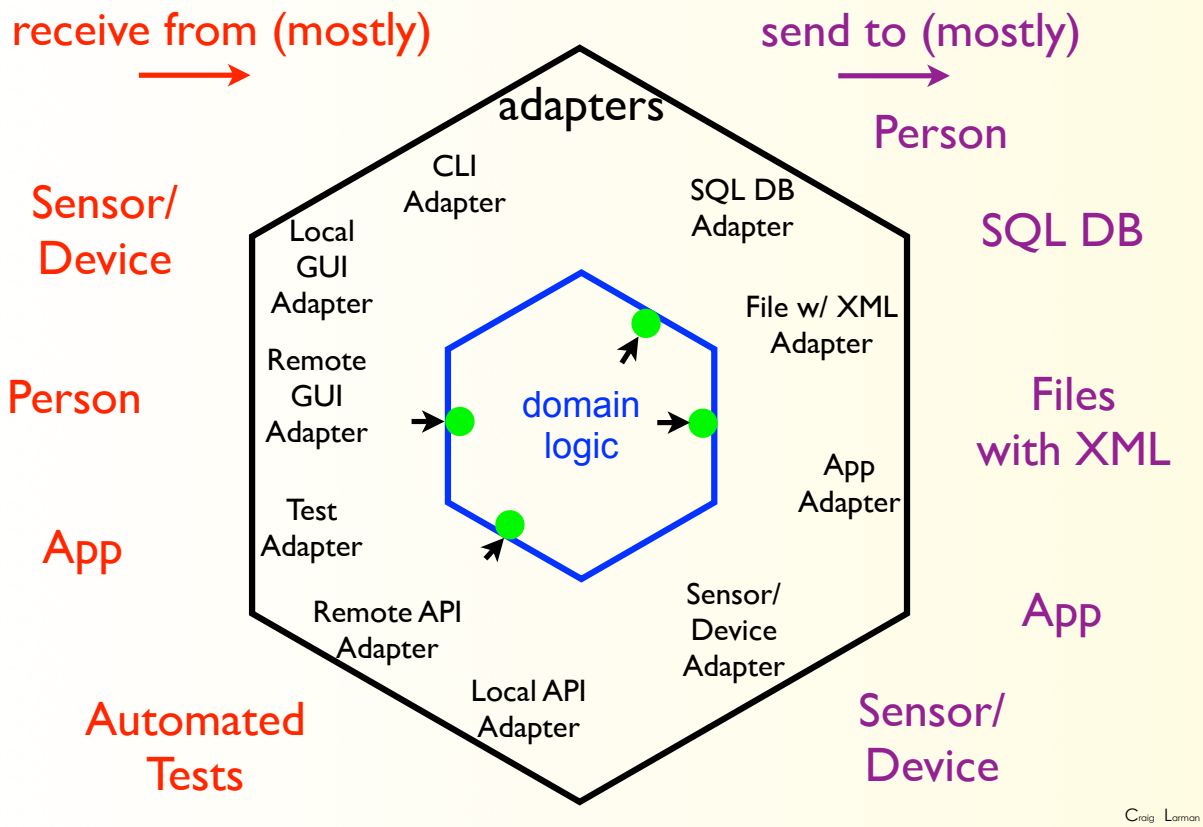
3. (Domain) Controller -- example



the system operations identified in the SDDs define the public API into the domain layer

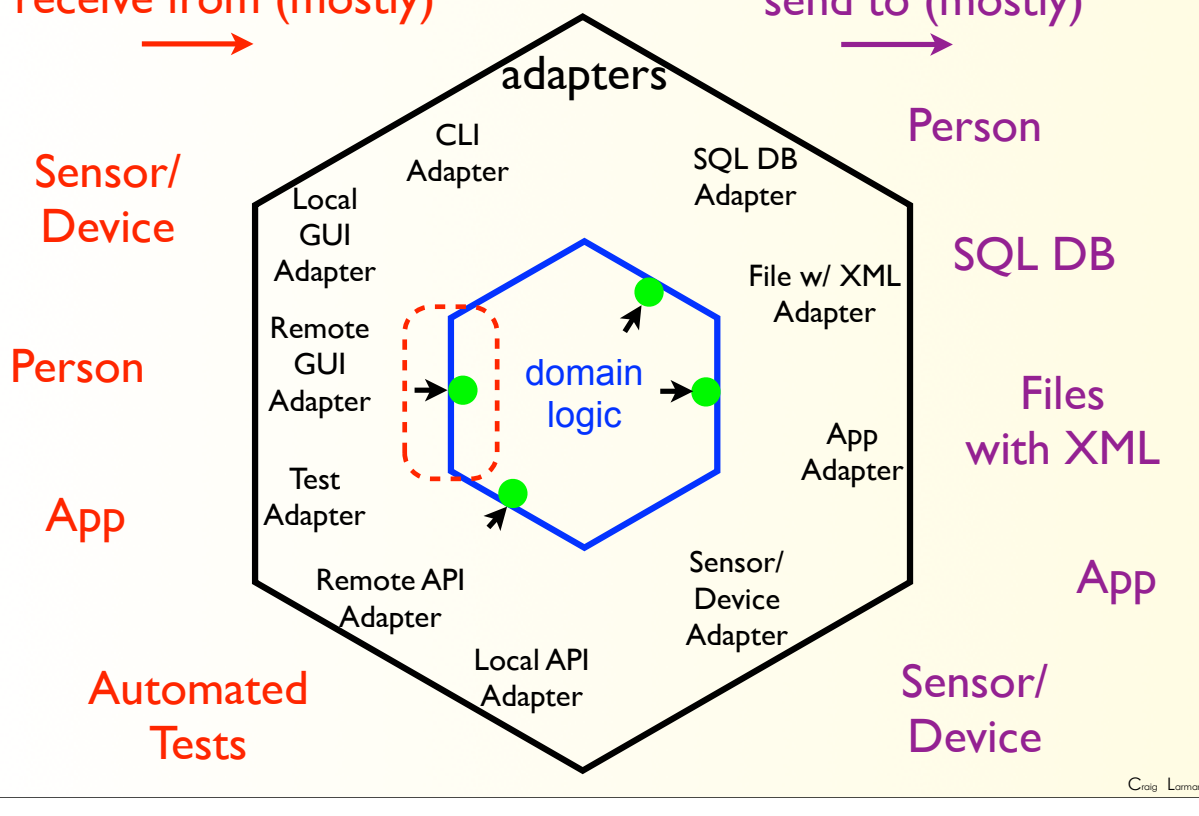
SSD with system operations

architecture metaphor: hexagon, adapters & ports



Craig Larman

a Domain Controller is a port to the domain logic



Craig Larman

3. Controller (AKA Domain Controller or Domain 'Facade')

- a classic "Facade" object does not contain meaningful application logic (it just forwards and/or choreographs). But a Domain Controller 'Facade' *may* contain application logic. e.g., a "PokerGame" class could be Controller on to the Domain layer, and contain behavior related to being a Poker Game.

1.Information Expert
2.Creator
3.Controller
4.Low Coupling
5.High Cohesion
6.Polymorphism
7.Pure Fabrication
8.Indirection
9.Protected Variations

4. Low Coupling

- Low Coupling does not mean NO coupling, and is most relevant when the dependent element is unstable
- applies at many levels
- key question: Is the dependent unstable?

1.Information Expert
2.Creator
3.Controller
4.Low Coupling
5.High Cohesion
6.Polymorphism
7.Pure Fabrication
8.Indirection
9.Protected Variations

- How do Information Expert and Creator support Low Coupling?

1.Information Expert
2.Creator
3.Controller
4.Low Coupling
5.High Cohesion
6.Polymorphism
7.Pure Fabrication
8.Indirection
9.Protected Variations

5. High Cohesion

- applies at many levels...
  - method
  - class
  - package
  - layer
  - subsystem
  - service
  - ....

1.Information Expert
2.Creator
3.Controller
4.Low Coupling
5.High Cohesion
6.Polymorphism
7.Pure Fabrication
8.Indirection
9.Protected Variations

- example specializations of High Cohesion:
  - "Single Responsibility principle"
  - "separation of concerns"
  - "layers"
  - "command/query separation principle"

Command/Query Separation (supports High Cohesion)

- violation:

```
public int roll() {
      faceValue = Math.random(1,MAX); // command
      return faceValue;               // query
}
```

- support:

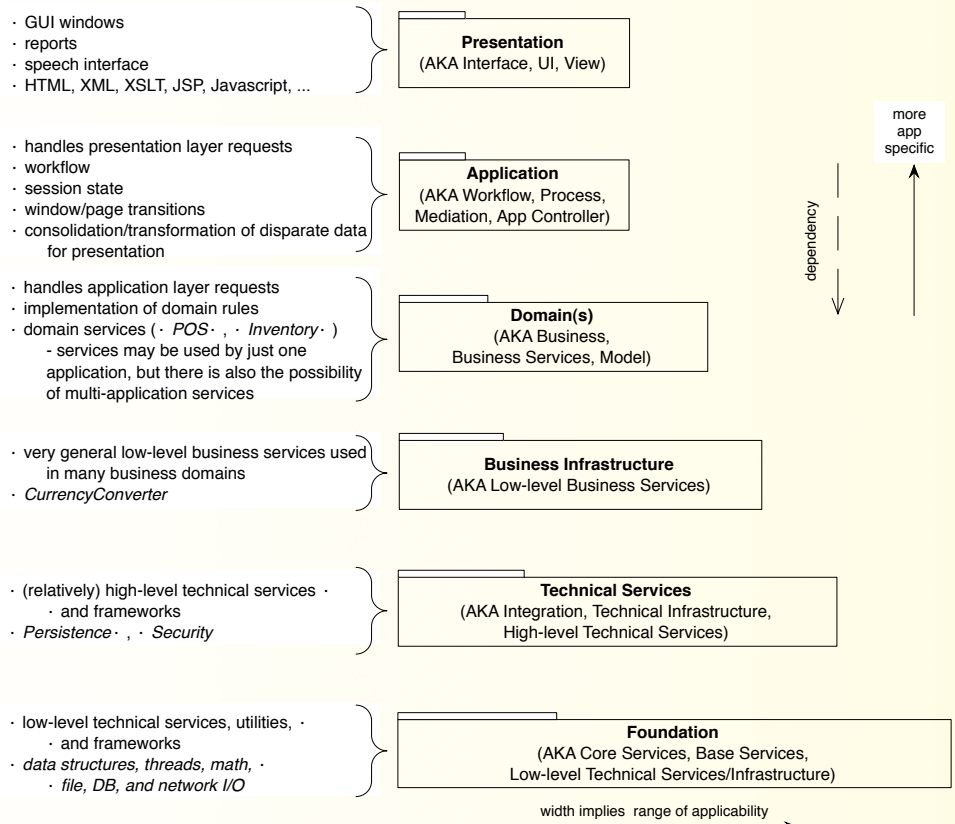
asking a question should not change the answer

```
public void roll() {
      faceValue = Math.random(1,MAX); // command
}

public int getFaceValue() {
      return faceValue;               // query
}
```

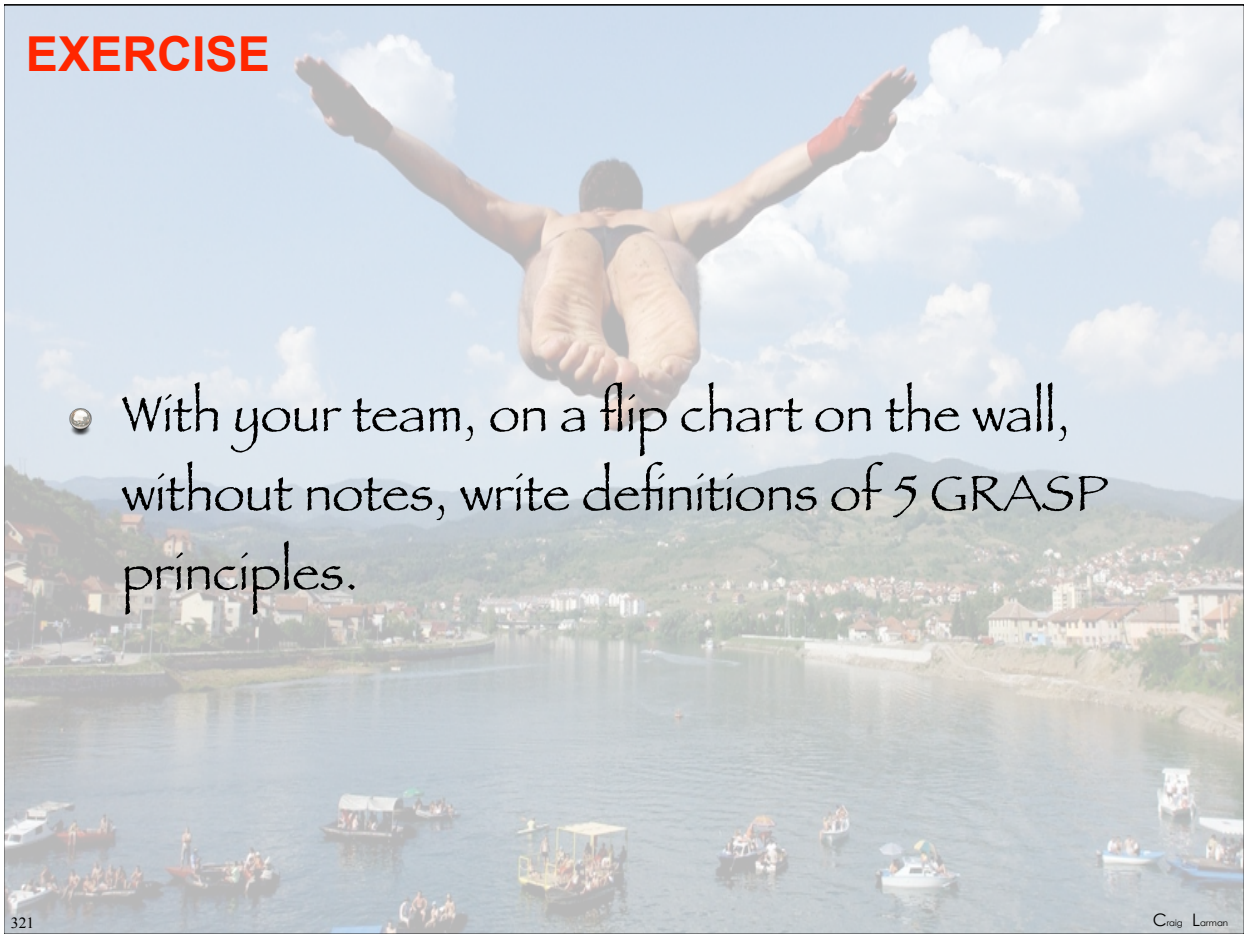
Command/Query Separation (supports High Cohesion)

- typically acceptable violations:
  - remote calls (performance motivation)
  - "fluent interface" (commands return this)
  - status return codes in C

layers (supports High Cohesion)



EXERCISE



- With your team, on a flip chart on the wall, without notes, write definitions of 5 GRASP principles.

GUIDELINES

Fact: When we apply Creator or Information Expert, we may need to choose names of software classes

Question: How to choose name? …

GUIDELINE: low representational gap

- create software classes whose names & data reflects the domain model -- low representational gap

domain model

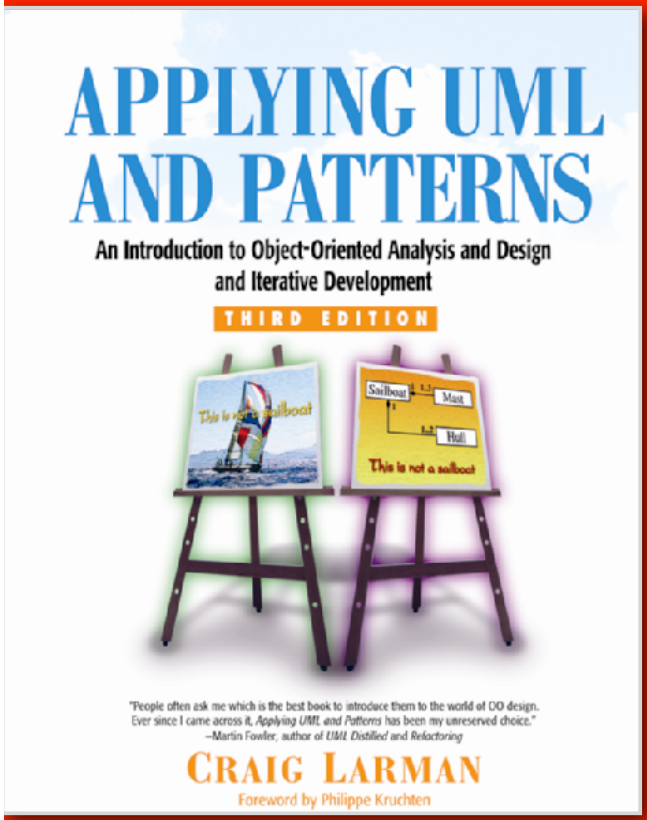
VideoStore
address : Address name : Text

design model

VideoStore
address : Address name : Text
addRental

the software classes may be inspired by the domain model

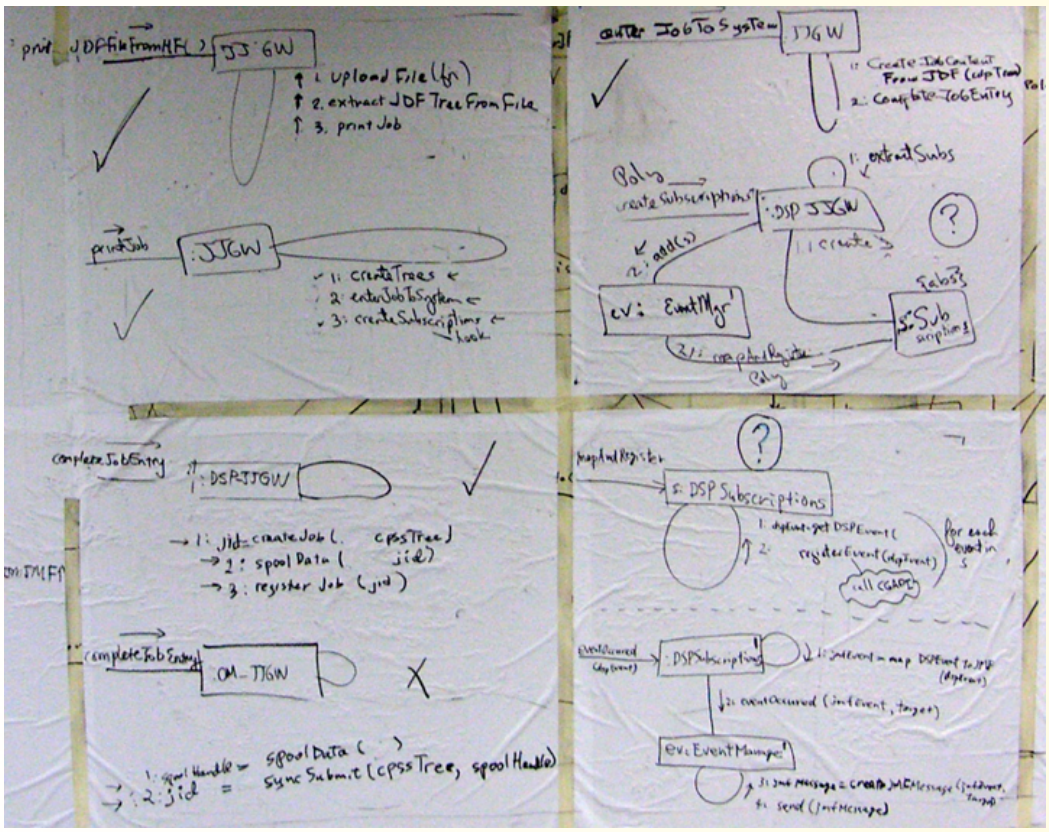
reduced representational gap



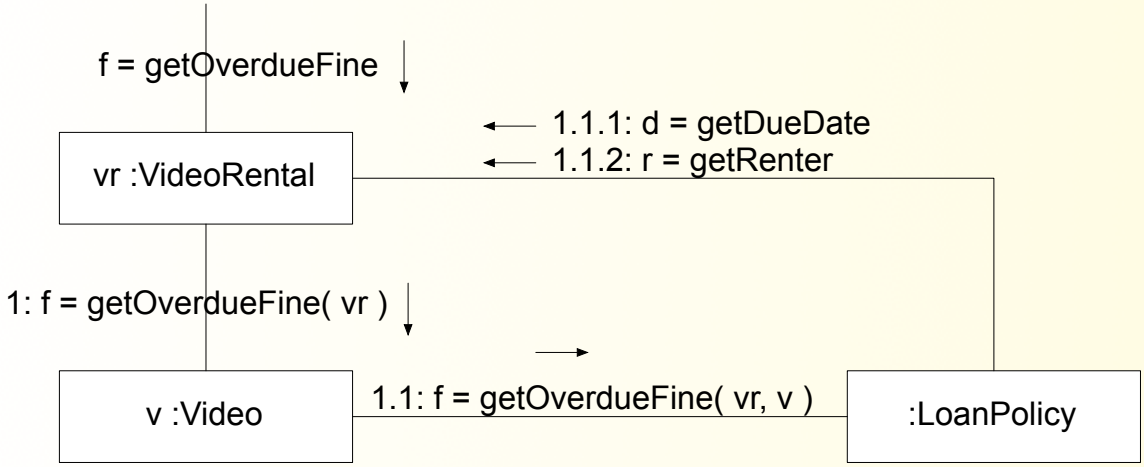
1 Object-Oriented Analysis and Design 3
2 Iterative, Evolutionary, and Agile 17
3 Case Studies 41
PART II INCEPTION
4 Inception is Not the Requirements Phase 47
5 Evolutionary Requirements 53
6 Use Cases 61
7 Other Requirements 101
PART III ELABORATION ITERATION 1 — BASICS
8 Iteration 1—Basics 123
9 Domain Models 131
10 System Sequence Diagrams 173
11 Operation Contracts 181
12 Requirements to Design—Iteratively 195
13 Logical Architecture and UML Package Diagrams 197
14 On to Object Design 213
15 UML Interaction Diagrams 221
16 UML Class Diagrams 249
17 GRASP: Designing Objects with Responsibilities 271
18 Object Design Examples with GRASP 321
19 Designing for Visibility 363
20 Mapping Designs to Code 369
21 Test-Driven Development and Refactoring 385
22 UML Tools and UML as Blueprint 395
PART IV ELABORATION ITERATION 2 — MORE PATTERNS
23 Iteration 2—More Patterns 401
24 Quick Analysis Update 407
25 GRASP: More Objects with Responsibilities 413
26 Applying GoF Design Patterns 435
PART V ELABORATION ITERATION 3 — INTERMEDIATE TOPICS
27 Iteration 3—Intermediate Topics 475
28 UML Activity Diagrams and Modeling 477
29 UML State Machine Diagrams and Modeling 485
30 Relating Use Cases 493
31 Domain Model Refinement 501
32 More SSDs and Contracts 535
33 Architectural Analysis 541
34 Logical Architecture Refinement 559
35 Package Design 579
36 More Object Design with GoF Patterns 587
37 Designing a Persistence Framework with Patterns 621
38 UML Deployment and Component Diagrams 651
39 Documenting Architecture: UML & the N+1 View Model 655

Create Object Design with Communication Diagrams

EXAMPLE: communication diagram

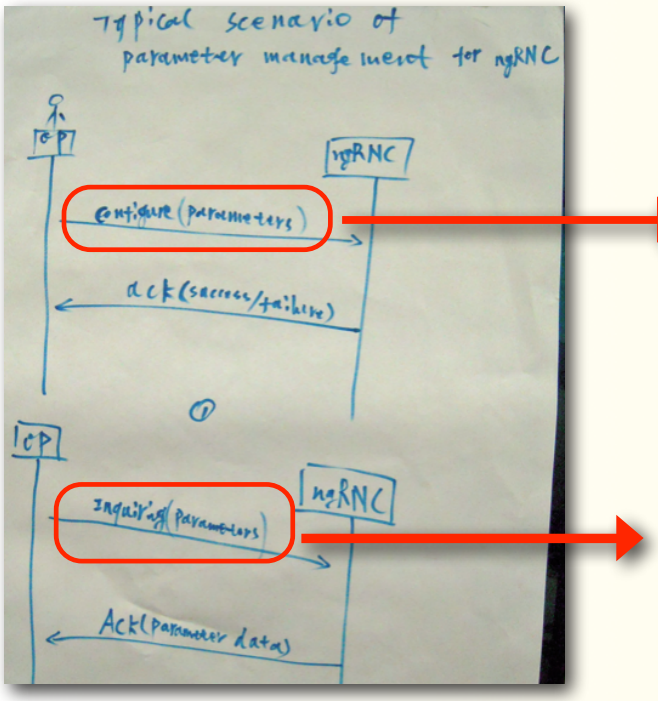


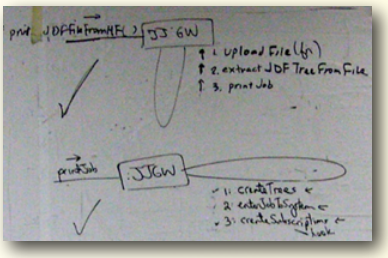
NOTATION: communication diagram

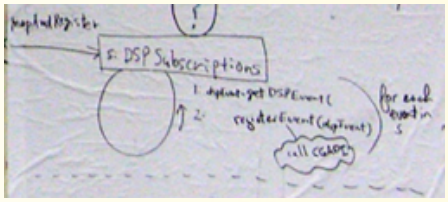


GUIDELINE: consider for each system operation

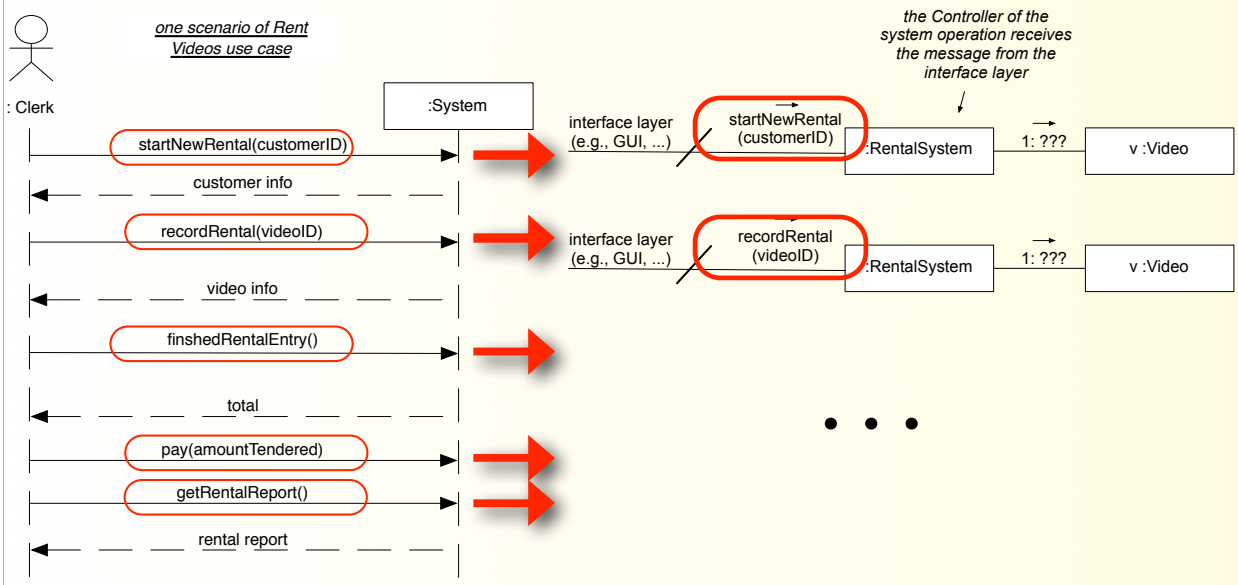
SSDs identify each system operation for which we need to design a solution





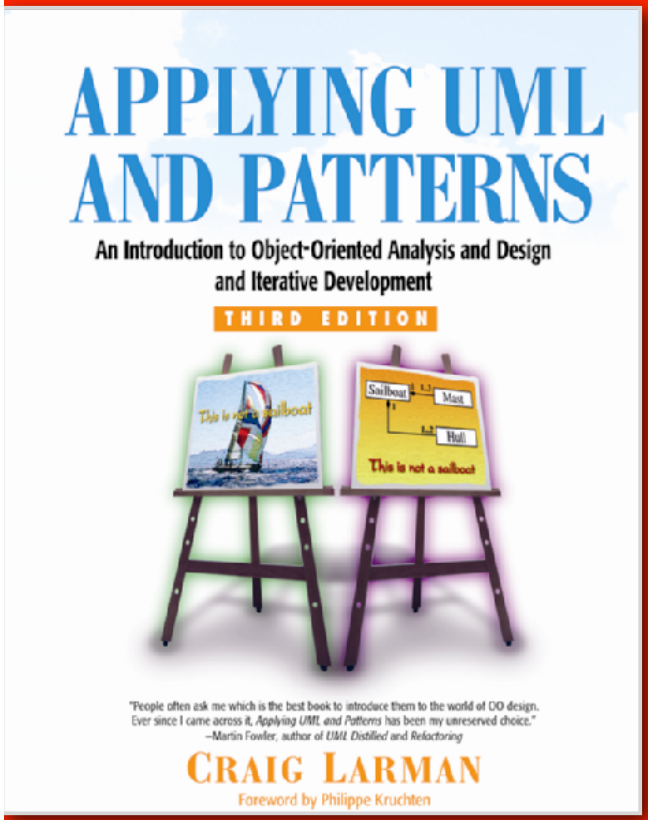


EXAMPLE: consider for each system operation



SSD with system operations

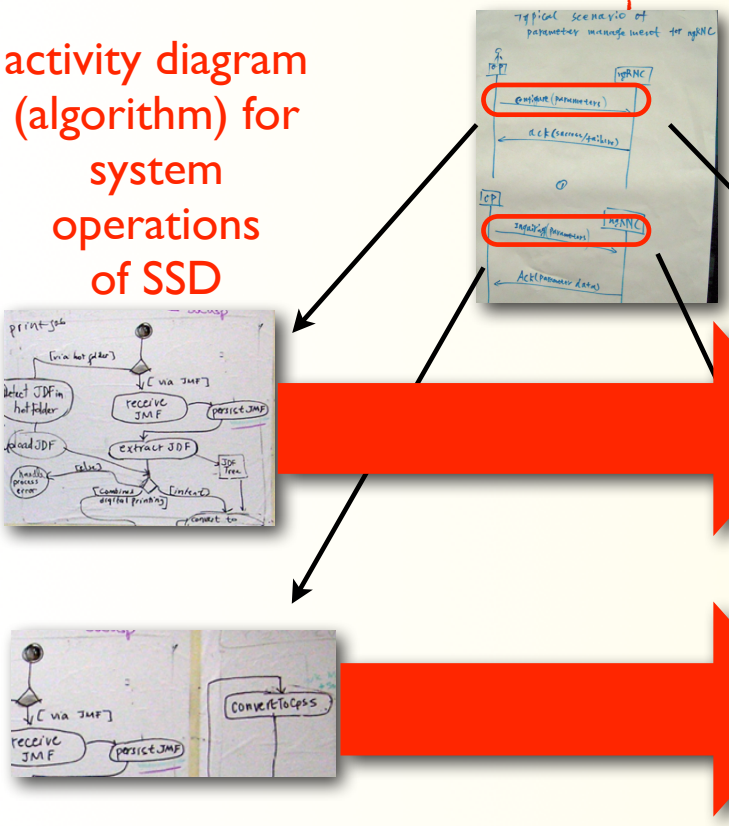
communication diagrams for system operations



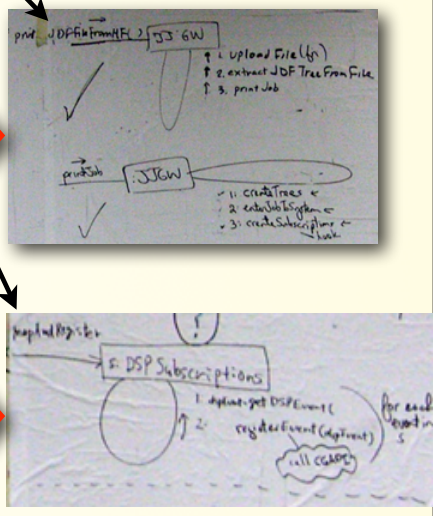
1 Object-Oriented Analysis and Design 3
2 Iterative, Evolutionary, and Agile 17
3 Case Studies 41
PART II INCEPTION
4 Inception is Not the Requirements Phase 47
5 Evolutionary Requirements 53
6 Use Cases 61
7 Other Requirements 101
PART III ELABORATION ITERATION 1 — BASICS
8 Iteration 1—Basics 123
9 Domain Models 131
10 System Sequence Diagrams 173
11 Operation Contracts 181
12 Requirements to Design—Iteratively 195
13 Logical Architecture and UML Package Diagrams 197
14 On to Object Design 213
15 UML Interaction Diagrams 221
16 UML Class Diagrams 249
17 GRASP: Designing Objects with Responsibilities 271
18 Object Design Examples with GRASP 321
19 Designing for Visibility 363
20 Mapping Designs to Code 369
21 Test-Driven Development and Refactoring 385
22 UML Tools and UML as Blueprint 395
PART IV ELABORATION ITERATION 2 — MORE PATTERNS
23 Iteration 2—More Patterns 401
24 Quick Analysis Update 407
25 GRASP: More Objects with Responsibilities 413
26 Applying GoF Design Patterns 435
PART V ELABORATION ITERATION 3 — INTERMEDIATE TOPICS
27 Iteration 3—Intermediate Topics 475
28 UML Activity Diagrams and Modeling 477
29 UML State Machine Diagrams and Modeling 485
30 Relating Use Cases 493
31 Domain Model Refinement 501
32 More SSDs and Contracts 535
33 Architectural Analysis 541
34 Logical Architecture Refinement 559
35 Package Design 579
36 More Object Design with GoF Patterns 587
37 Designing a Persistence Framework with Patterns 621
38 UML Deployment and Component Diagrams 651
39 Documenting Architecture: UML & the N+1 View Model 655

GUIDELINE: inspiration from activity diagrams

activity diagram (algorithm) for system operations of SSD

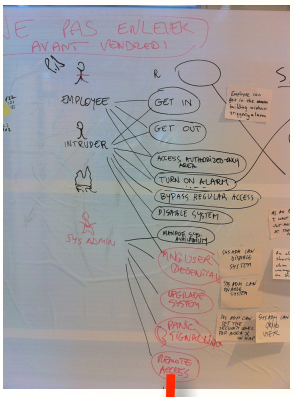


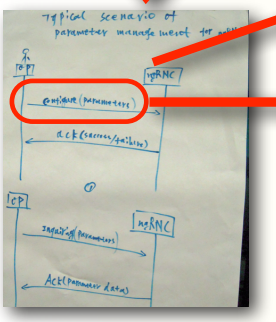
detailed object design for system operations of SSD

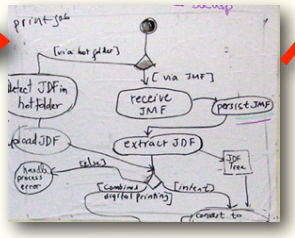


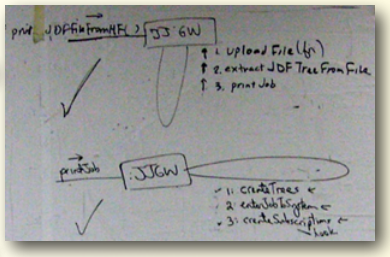
Craig Larman

Outside-In Development









Craig Larman

GUIDELINE: low representational gap

- create software classes whose names & data reflects the domain model -- low representational gap

domain model

VideoStore
address : Address name : Text

design model

VideoStore
address : Address name : Text
addRental

the software classes may be inspired by the domain model

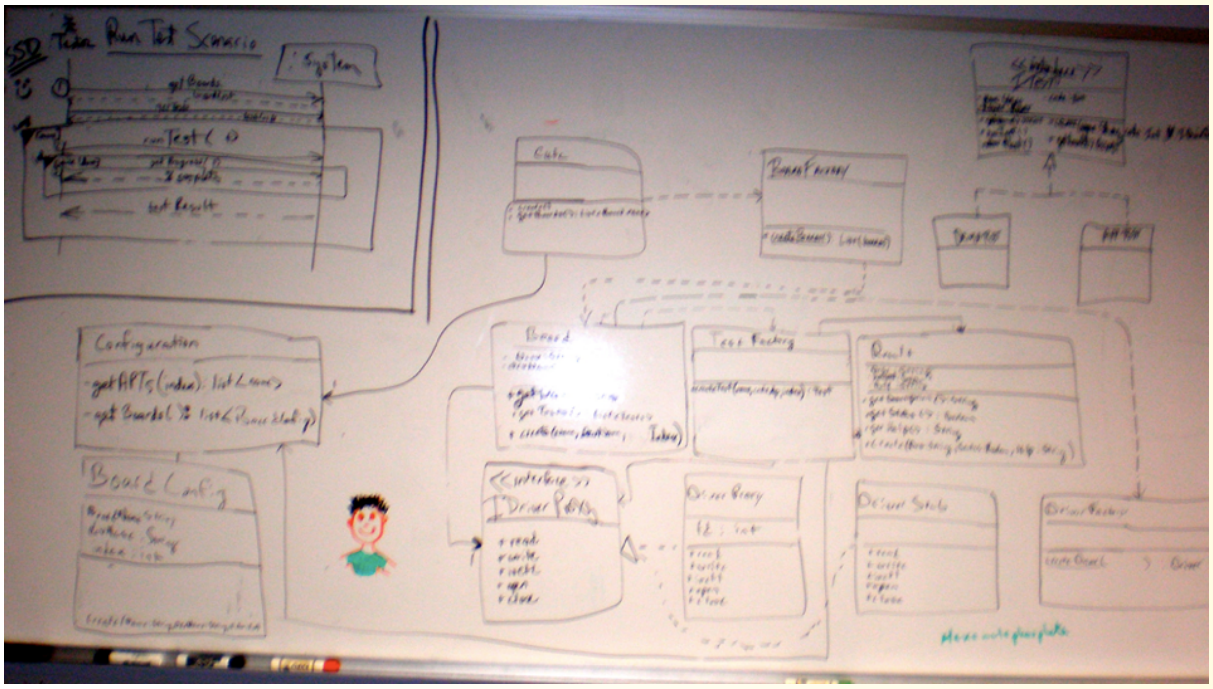
reduced representational gap

EXERCISE

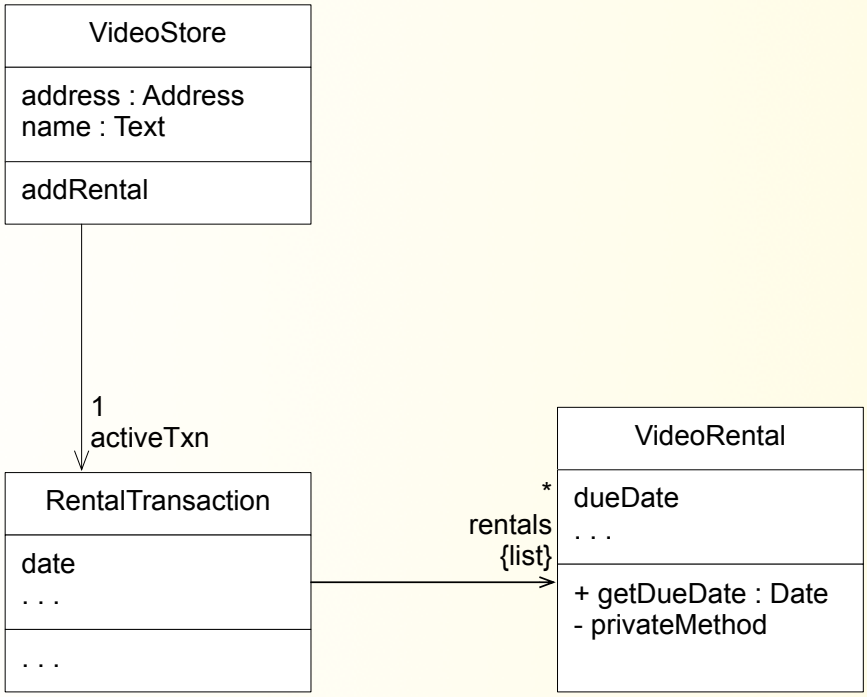
- With your team at the whiteboard, create an object design for the system operations indicated by the coach.
  - sketch with communication diagrams
  - consider inspiration or constraints from prior models: domain model, activity diagrams, ...
  - apply the GRASP principles

Create Object Design with Design Class Diagrams

EXAMPLE: design class diagram (DCD)



EXAMPLE: design class diagram (DCD)



Craig Larman

EXERCISE

1. With your team at the whiteboard, draw the design class diagram corresponding to the communication diagrams previously created.

Craig Larman

Agile Modeling for Shared Design Discovery

visual modeling on whiteboards is a great learning aid to learn how to "think in objects"

it has another value...

there is an important context here that is not appreciated by *single* solo workers...

agile modeling easily encourages a *team* to collaborate and learn from each other, and develop shared understanding of the design, for the *common* feature they will be working together on, over the following days.

if you don't have real teams, but OTOH have *individual* people working on different features at the same time, the value of this *team* alignment via agile modeling will not be appreciated

now we make a major shift from
exploring design
to *real* design — in code!

the story until now…

Topics

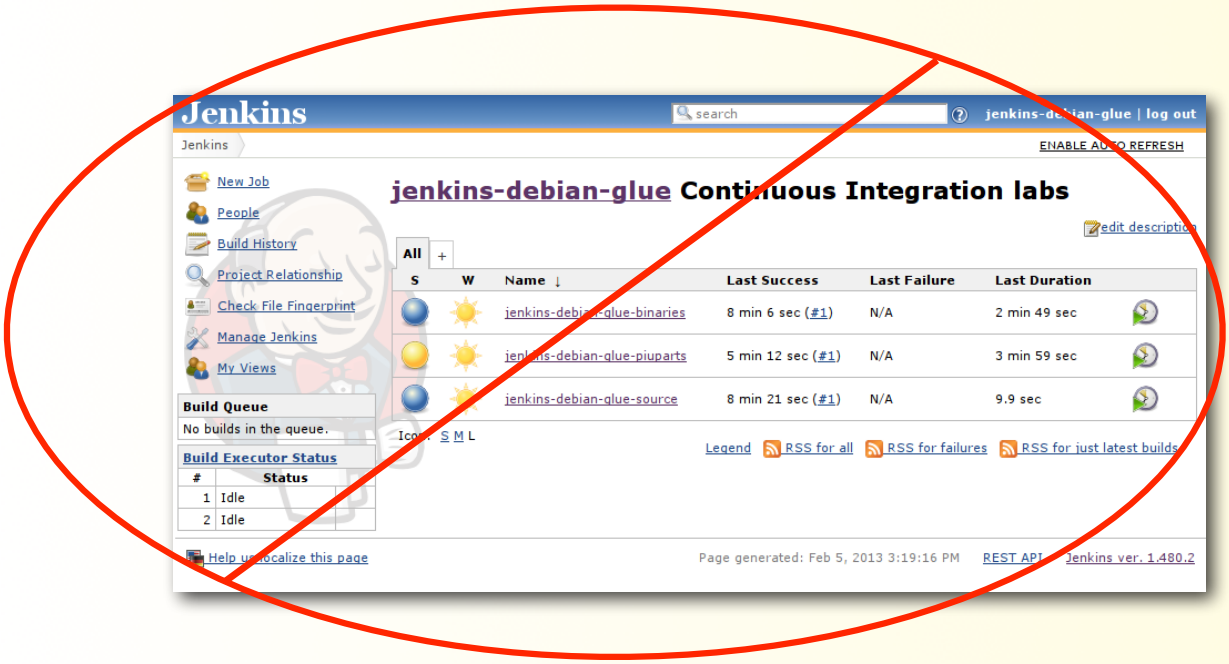
- Lean & Agile Modeling
- Lean & Agile Requirements
- User-Centered Design
- Story Mapping
- Telling Stories
- Splitting Features
- Learn with System-Sequence Diagrams
- Learn with a Domain Model
- Learn with Specification by Example
- Acceptance TDD
- BAD vs GOOD Automated Tests
- Designing with Layers
- Create Algorithms with Activity Diagrams
- GRASPing Object Design Principles
- Create Object Design with Communication Diagrams
- Create Object Design with Design Class Diagrams
- Continuous Integration
- Create Code Driven by Unit Tests (Unit TDD)
- Lean Thinking
- Lean Software Development
- Agile Values & Principles
- SOLID Principles
- Generalization
- Polymorphism & Class Hierarchy Design
- Feature Toggle Patterns
- Simple Patterns (Simple Factory, Null Object, ...)

Continuous Integration

we will now reveal the surprising meaning of "continuous integration". it is ...

... to integrate continuously

real continuous integration has little to nothing to do with build servers or Jenkins/etc



real continuous integration

means to…

integrate continuously

are you checking back in every 5 minutes?
if not, you are delaying integration

are you all on the head of the trunk?
(avoiding branching)
if not, you are delaying integration

CI is a programmer behavior

Continuous Integration (CI)

CI is a developer practice (behavior)
with the goal to always have a working system
by small changes, slowly growing the system
and integrating them many times daily
on the mainline (trunk)
typically supported by a CI system
with lots of automated tests

- Increases transparency
- Increases cooperation and communication
- Enables people to work on same code

Create Code Driven by Unit Tests (Unit TDD)

only write code to fix a failing test

development is driven by tests...

test-driven development

= test-first development

RED GREEN REFACTOR

1. Red – First, write one test and then a "1 second" do-nothing "solution" (which compiles but fails test)
2. Green – Write a passing solution
3. Refactor – refactor the code (and test that you're still green)
   - DRY, ...

1. Red < 2 min
2. Green < 2 min
3. Refactor < 2 min

4. Integrate with *all* product code

CI -> Check-in early. Check-in often. ✓

- duplicate code or data
- long method
- unclear names
- magic constants
- high coupling (e.g., "data envy")
- low cohesion
- case logic rather than polymorphism
- data object (record object, data transfer object)
- comments that explain what code does (rather than why)

GUIDELINES

- avoid unit tests for trivial getters/setters
- use a standard variable name for your instance under test; e.g., testObj
- start with a testInitialize, that checks construction post-conditions

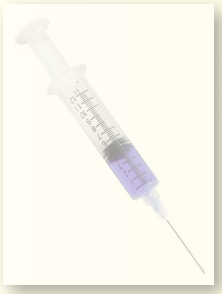
demo

- the coach will demonstrate unit TDD (which includes refactoring)

GUIDELINE: low coupling to objects not under test

- provide dependency injection or control via...

- object seams (polymorphism) or link seams
- constructor parameters
- setters
- factories
- dependency-injection framework (e.g., Spring); it uses factories, etc.



demo

- (the coach may defer this until a later iteration)

- the coach will demonstrate unit TDD and dependency injection or control

GUIDELINE: test FIRST...

- F - fast (< 5 milliseconds)
- I - independent
  - do not have to talk to file system, database, network, ...
  - tests do not depend on other tests
- R - repeatable
  - do not permanently change state (of shared objects, files, database, ...)
- S - self-validating (no output to view & verify)
- T - timely (before the solution)

GUIDELINE: solutions inspired by agile models

```
import static org.junit.Assert.*;
import org.junit.*;
public class FooTest {
 @Before public void setUp() { }

 @After public void tearDown()  { }

 @Test public void test1() {
    assertEquals(1, 1);
 }
}
```


Implementation Patterns

if there is a UI in the feature, split it into 3 children, and implement in this order:

1. feature with an API to drive it
2. feature with a command-line interface (CLI) driving the API
3. feature with the intended UI technology

this will force a good architecture with no application logic in the UI layer, and the CLI will sometimes be useful for quick exploration or configuration

CLI frameworks

clamshell-cli

A framework to build command-line console applications in Java

CLI toolkit | Changes | Download | Documentation | Contact

CLI toolkit

Natural CLI home | Downloads | Examples | Changelog | Javadocs | Contact

Natural CLI

Natural CLI is a Java library providing to developers command line interfaces with human readable sentences. It means, your software can understand easily command lines like the following:

jline.sourceforge.net

jline

```
 .d8888b.  888                        .d8888b.  888               888 888
d88P  Y88b 888                       d88P  Y88b 888               888 888
888    888 888                       Y88b.      888               888 888
888        888  8888b.  88888b.d88b.   :Y888b.   88888b.   .d88b.  888 888
888        888     :88b 888 :888 :88b     :Y88b. 888 :88b d8P  Y8b 888 888
888    888 888 .d888888 888  888  888       :888 888  888 88888888 888 888
Y88b  d88P 888 888  888 888  888  888 Y88b  d88P 888  888 Y8b.     888 888
 :Y8888P:  888 :Y888888 888  888  888  :Y8888P:  888  888  :Y8888  888 888

                                                    Command-Line Interpreter

Java version: 1.6.0_22
Java Home: /usr/lib/jvm/java-6-openjdk/jre
OS: Linux, Version: 2.6.38-10-generic

prompt> _
```

Exercise

before the exercise specifics, some details/practices you need to consider…

the IDE keyboard challenge!

(great programmers use short cuts)

if pair programming…

1. person1: 'red'
2. person2: 'green'
3. person1: 'refactor'
4. person2: 'red'
5. …

if pair programming...

promiscuous pairing

continuous integration behavior...

1. check out
2. create & run tests
3. *update*
4. if merging, re-run tests
5. commit

continuous integration behavior…

1. *update* (into each local computer) from the *entire* product codebase, *every* TDD cycle

"test double" or "mocking" frameworks are useful, but in this course we will keep it simple and use hand-crafted test doubles

where will you place tests and test doubles?

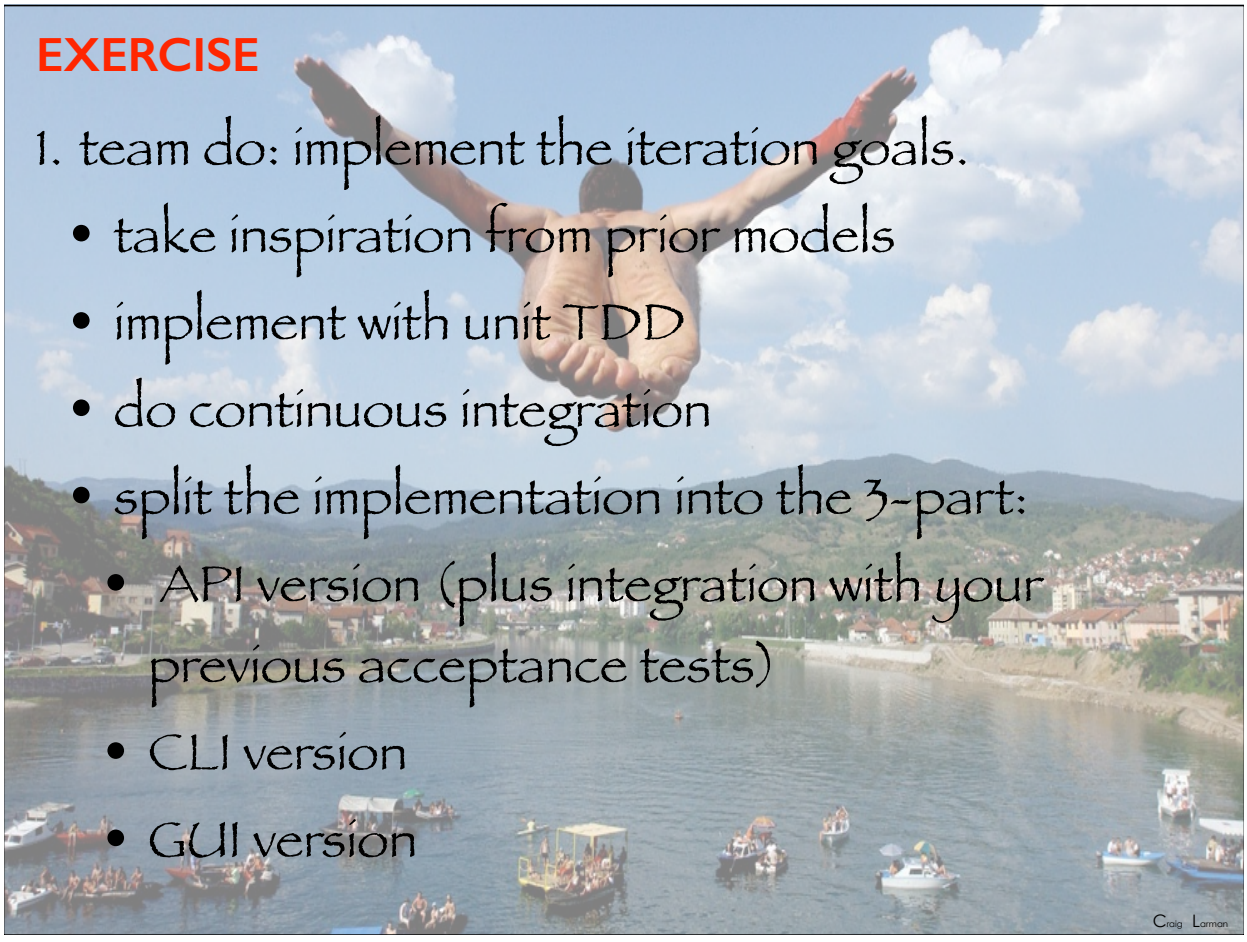
test doubles: keep it simple and just subclass a regular class for the test double; don't introduce interfaces unless it is needed for the "regular" design goals

are you using a test coverage "color coding" tool such as EclEmma?

might be useful… and then everyone should use it

before your team starts, spend a few minutes at your design class diagram, and discuss coordination and division of tasks

As you go along and discover the need for coding conventions & standards (pattern for naming tests, ...), talk about and agree on these — it is important for a product group with internal open source to follow common standards. You don't need to decide these up front, but do incrementally discuss & decide them.



EXERCISE

1. Bind the solution to the acceptance tests, and ensure they pass.
2. If have not yet done so, evolve the "test" page into "living documentation" by adding an image, and some explanatory text

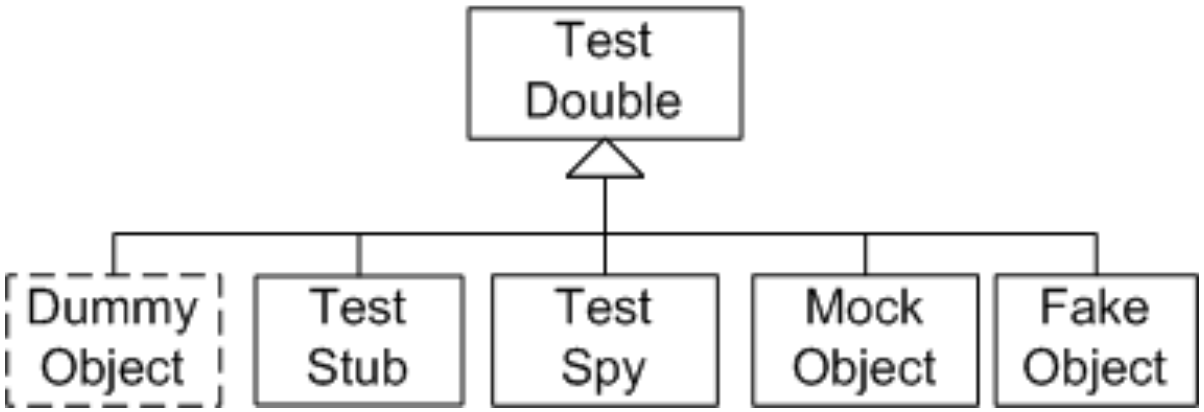
Craig Larman

now that you have the visceral experience, notice that in real *team* based development, with the whole team working together on 1 feature at a time (the Scrum & Lean guidelines to keep WIP low, and "scrum together")… the critical importance of the *entire* team together literally at the same table — as your work environment

Test Doubles, Fakes, & Mocks

many neophytes use the term "mock" incorrectly…

MARTIN FOWLER
Intro Design Agile Refactoring NoSQL DSL
Mocks Aren't Stubs


Test Double
Dummy Object
Test Stub
Test Spy
Mock Object
Fake Object

what are *real* mocks for?

state-based versus
interaction-based (behavior)
testing

which supports better encapsulation (the vision of object-orientation and ADTs)?

case study:

an appropriate use of both state-based testing, and interaction-based testing with mocks…

case study:

Player.takeTurn()

versus

MonopolyGame.playRound()

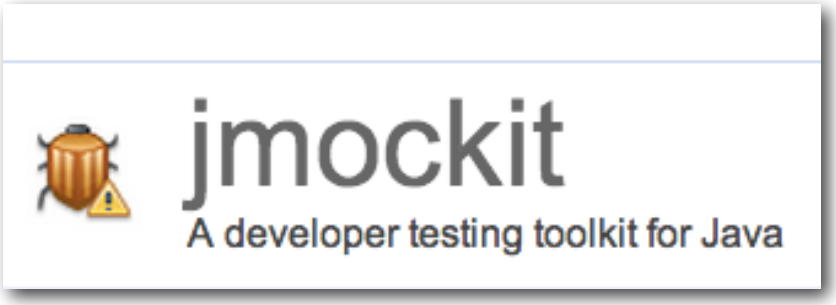
summary:

When testing MonopolyGame.playRound(), we don't want to interact with the real Player.takeTurn() — that would be an unnecessary integration test. We already know that Player.takeTurn() works correctly independently, via normal state-based testing. For the MonopolyGame.playRound() test of "all players are told to takeTurn" the only test we need now to make is if all MockPlayers are being called.

Model-View-Presenter example

similarly, once we have tested the functionality of 'P' and 'M' layers (but not the 'V' layer), we can connect the 'V' layer to a mock of the 'P' layer to simply ensure the "connections are correct"

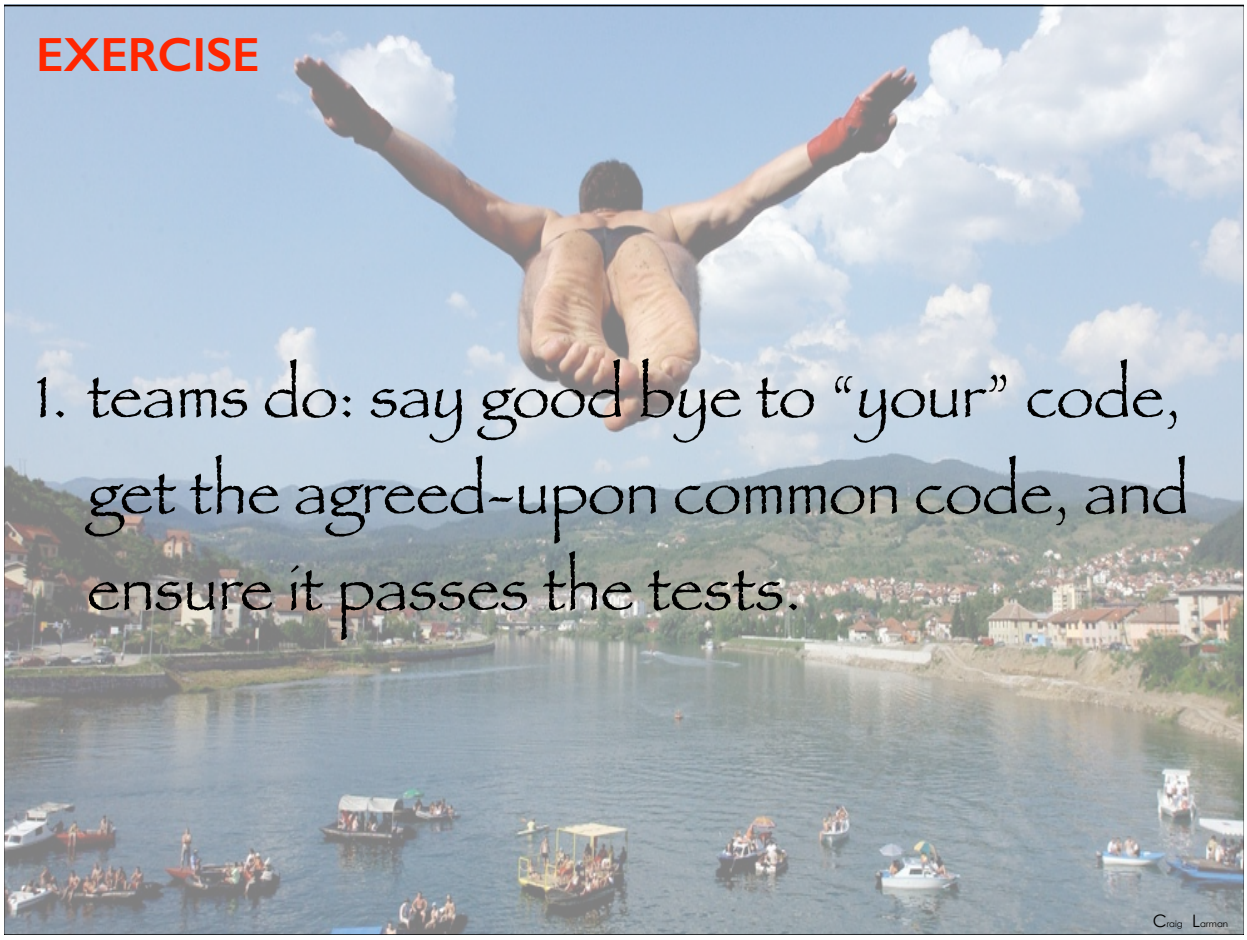
a current favorite



Internal Open Source (Collective Code Ownership)

with discussion with the coach, choose one of the product groups' code base, and have all other groups use that code base, rather than their prior "own" code

that might feel uncomfortable or scary, but notice that if we have clean code, with great unit test coverage and acceptance tests, you will find it relatively painless to use this open source code, rather than "your" code

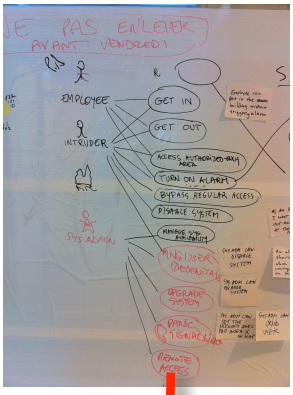


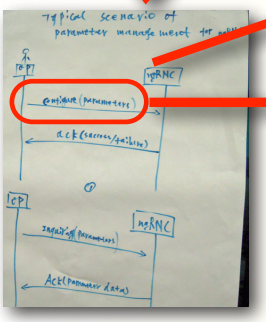
where are we?...

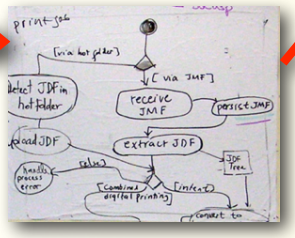
done the first iteration!

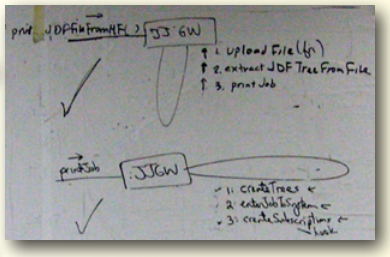
Outside-In Development

models from outside-in



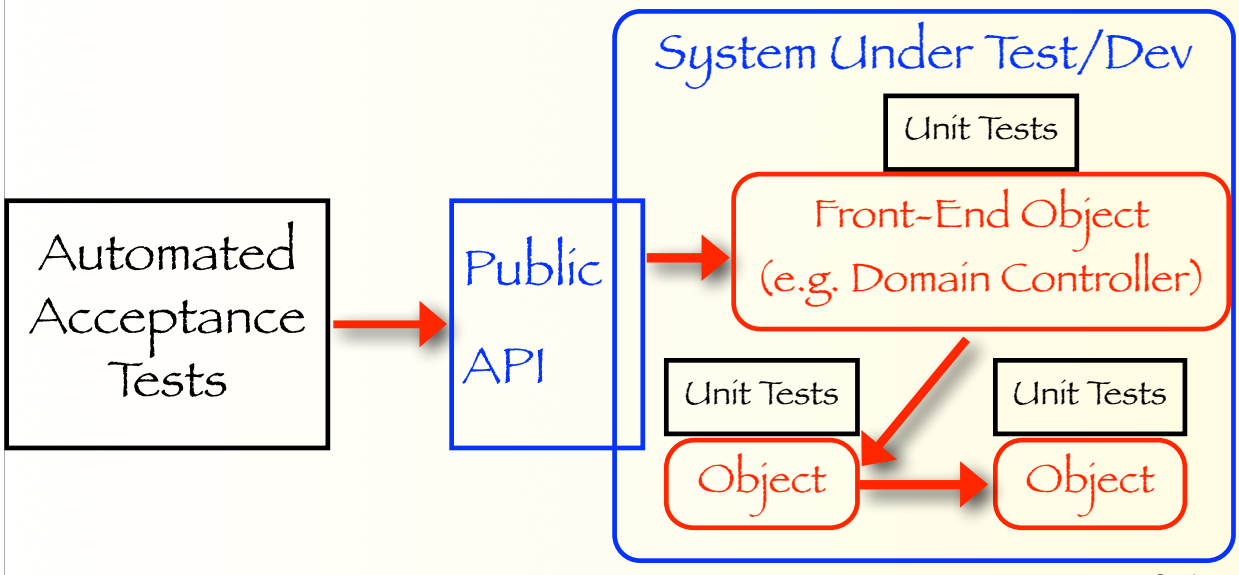






Outside-In Development

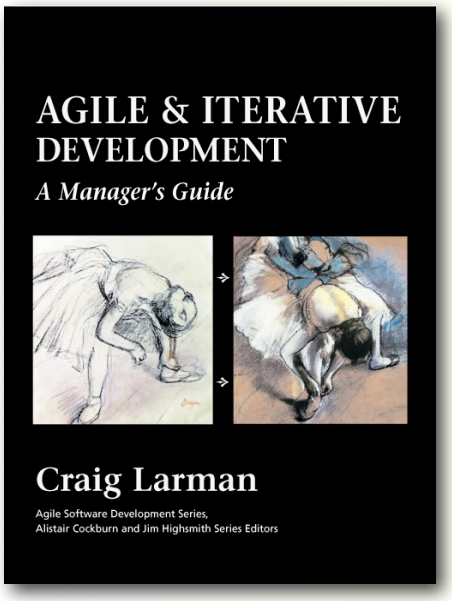
automated tests from outside-in



agile delivery:

build a thin end-2-end solution, then "thicken it" incrementally

-> early integration, early feedback, early delivery of smaller units of value, early confidence in completing something



the story until now...

Topics

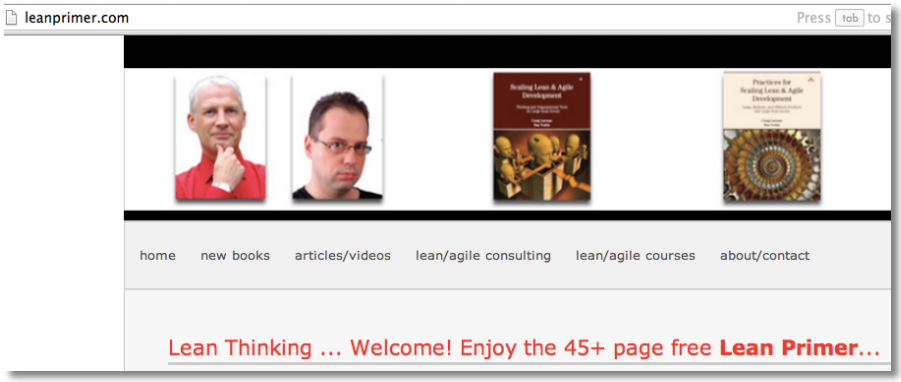
- Lean & Agile Modeling
- Lean & Agile Requirements
- User-Centered Design
- Story Mapping
- Telling Stories
- Splitting Features
- Learn with System-Sequence Diagrams
- Learn with a Domain Model
- Learn with Specification by Example
- Acceptance TDD
- BAD vs GOOD Automated Tests
- Designing with Layers
- Create Algorithms with Activity Diagrams
- GRASPing Object Design Principles
- Create Object Design with Communication Diagrams
- Create Object Design with Design Class Diagrams
- Continuous Integration
- Create Code Driven by Unit Tests (Unit TDD)
- Lean Thinking
- Lean Software Development
- Agile Values & Principles
- SOLID Principles
- Generalization
- Polymorphism & Class Hierarchy Design
- Feature Toggle Patterns
- Simple Patterns (Simple Factory, Null Object, ...)

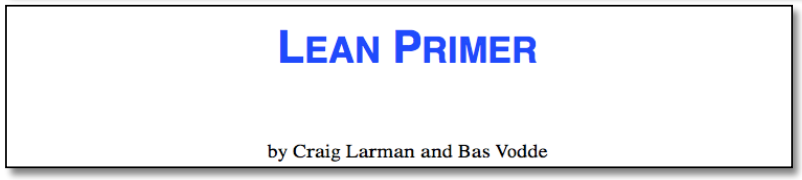
Code Sharing

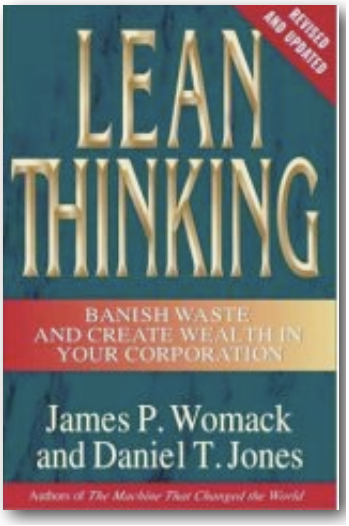
- (we probably did lots of code review during the first iteration, but if not…) let us look at some of each other's code (and test code), to learn and make suggestions

Lean Thinking

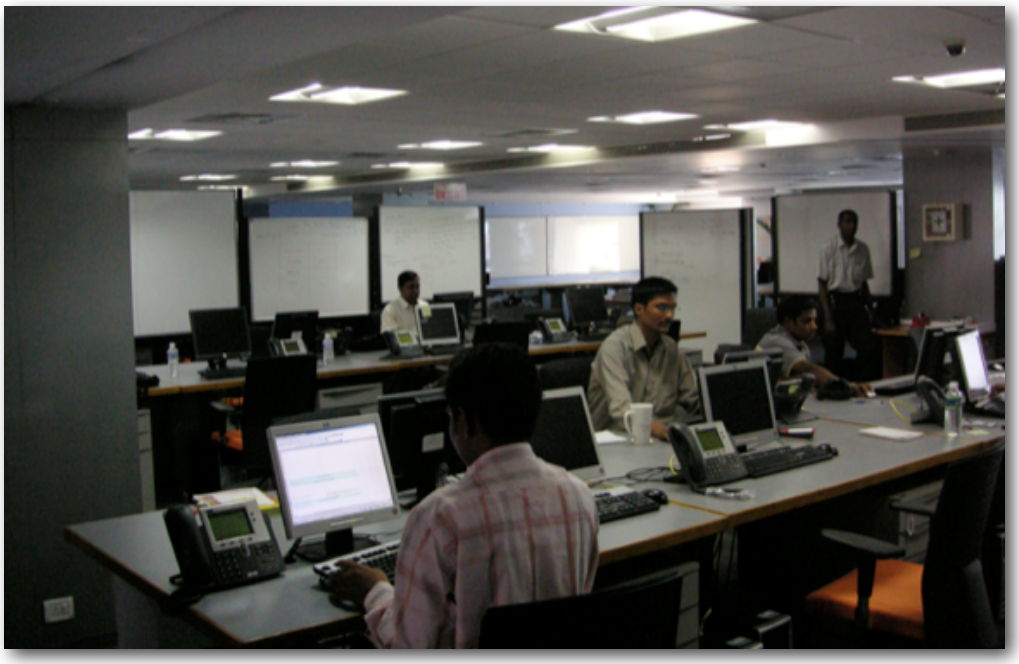
back to concepts…







Lean Principle: Go See and Help at "Gemba"



Japanese: "genchi genbutsu"

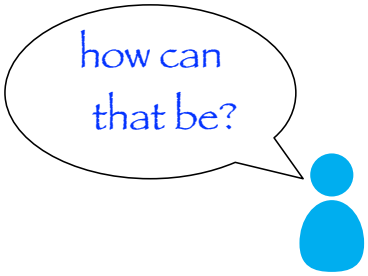
value stream

perfection goal:
value flow to customer without pause or impediment

Local Optimization

Low value-throughput happens because:

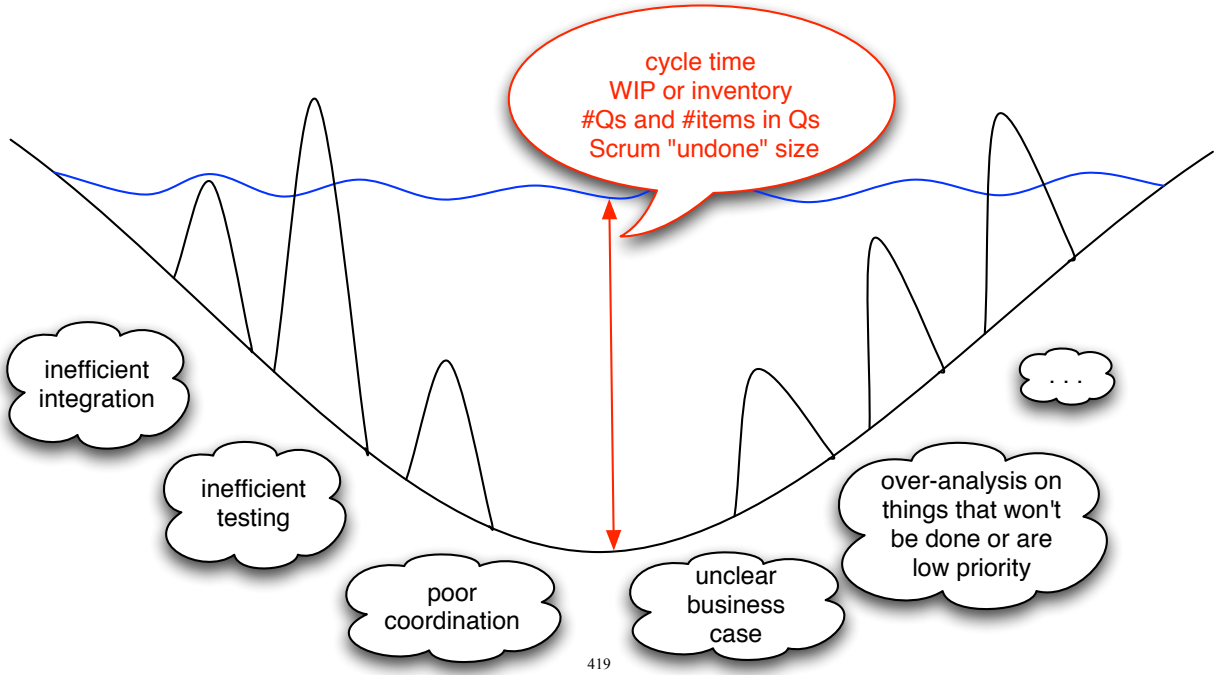
- we are optimizing all the parts
- everyone is doing their "best" and is "busy"
- thinking mistake: "we optimize the whole by optimizing the parts"



EXERCISE

- class discuss: what are some examples of when people behaved or organized thinking "this is best" or "this is fastest" and this was actually a local-best rather than a system-throughput best, and ironically, the "local best" behavior actually reduced/delayed overall throughput (usually due to the creation of bottlenecks, queues, waiting, or more hand off)

Increasing feedback & visibility by "lowering the waters"... lower the waters to see the waste and weakness (rocks), and deal with it, not because of need to ship in short cycles



EXERCISE

- class discuss... Fact: introducing "lowering the water" will "make things worse". Therefore, what does management need to support to move from "it's worse" to "it's better"?

Lean Principle: Stop and Fix

(this is not a principle for broken builds; this is for *every* weakness)

Lean Principle: Zero Quality Control

(...via Shingo. AKA Zero Defects via Crosby)

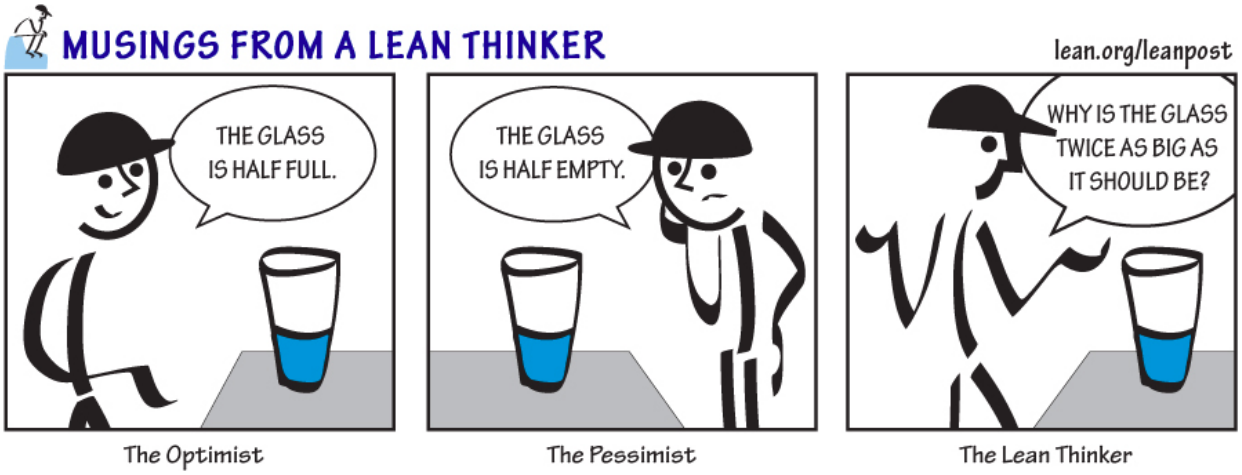
build quality in

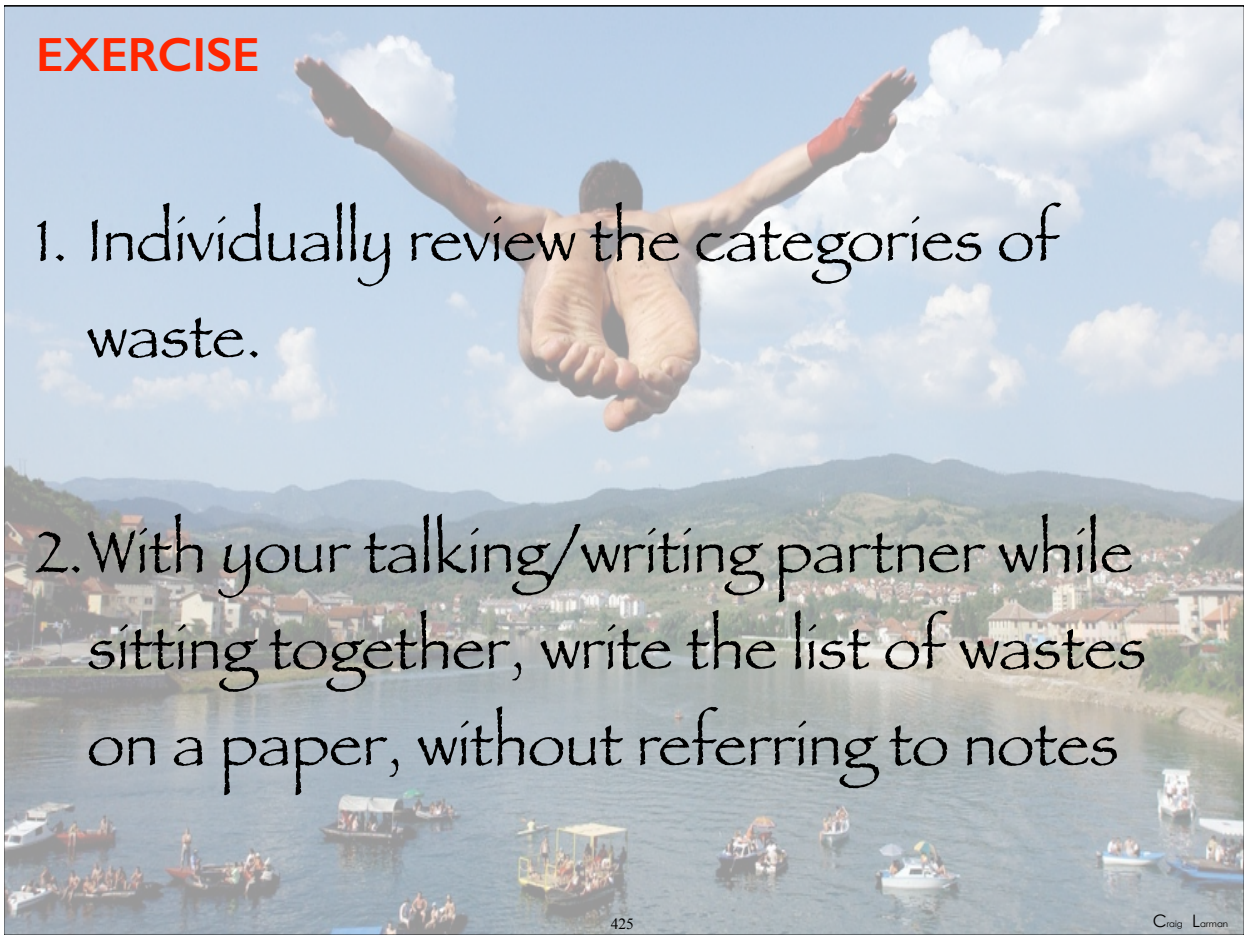
1. create
2. delay (transfer to another, ...)
3. test/inspect/review to find defects and correct them

"hardening", "QA phase", "system verification phase", "review the doc", "review the code", UAT ,..integration testing

A Lean List of Wastes (remove these)

1. Inventory; overproduction of features, or of elements ahead of the next step
2. Waiting, delay
3. Handoff, conveyance
4. Extra processing (includes extra processes), relearning
5. Work in progress (WIP); partially done work, (specifications not implemented, code not integrated or tested)
6. Task switching, motion between tasks, interrupt-driven multitasking
7. Defects, testing/inspection and correction at the end
8. Not using people's full potential: "working to job title," no multi-skill, no multi-learning, no kaizen, ...
9. Knowledge and information scatter or loss
10. Wishful thinking (e.g., plans and specifications are correct, that estimates can't increase)
11. blaming





EXERCISE

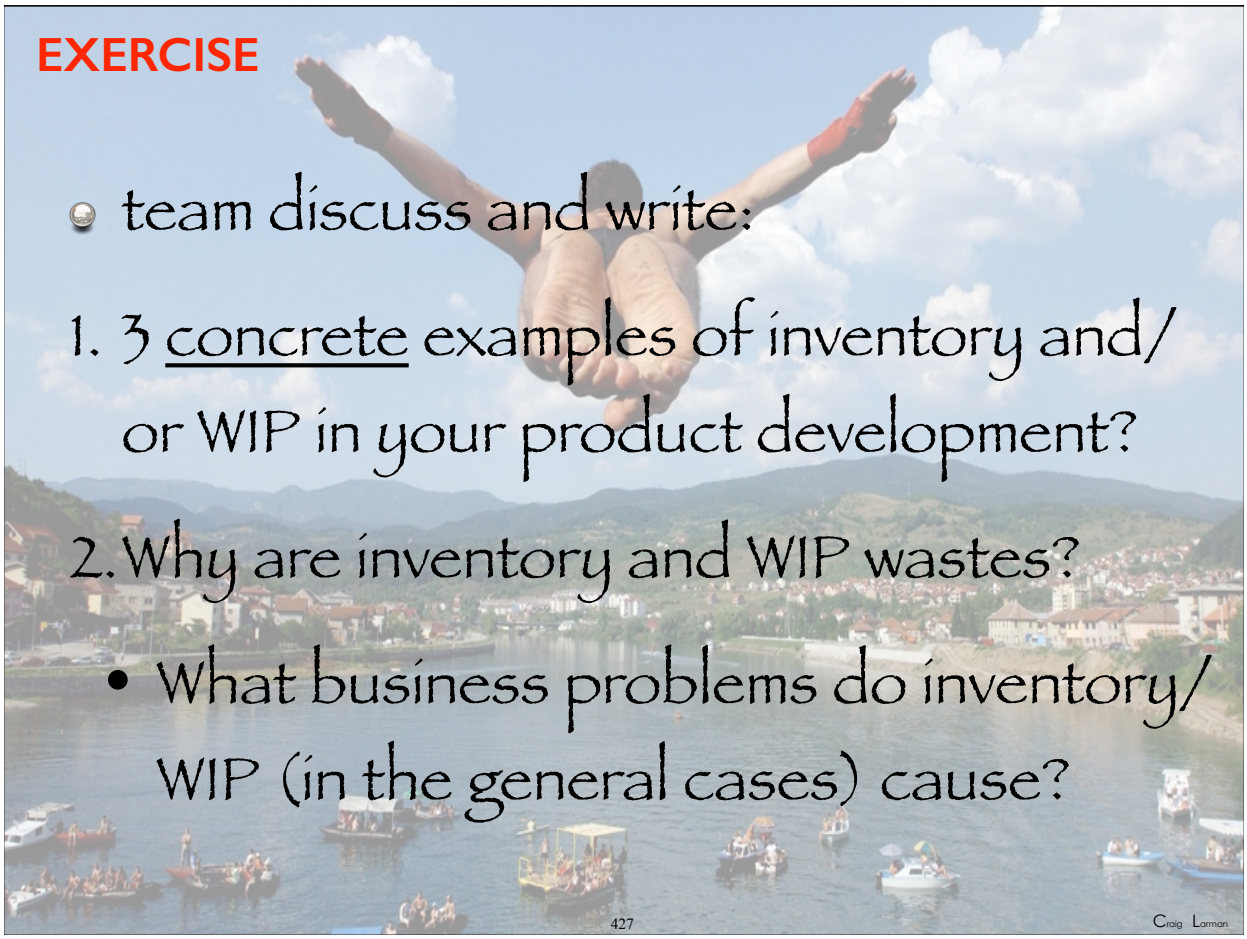
1. Individually review the categories of waste.

2. With your talking/writing partner while sitting together, write the list of wastes on a paper, without referring to notes

Craig Larman

pure waste

temporary necessary waste



EXERCISE

- team discuss and write:

1. 3 concrete examples of inventory and/or WIP in your product development?
2. Why are inventory and WIP wastes?
   - What business problems do inventory/WIP (in the general cases) cause?

Business Problems of Inventory/WIP

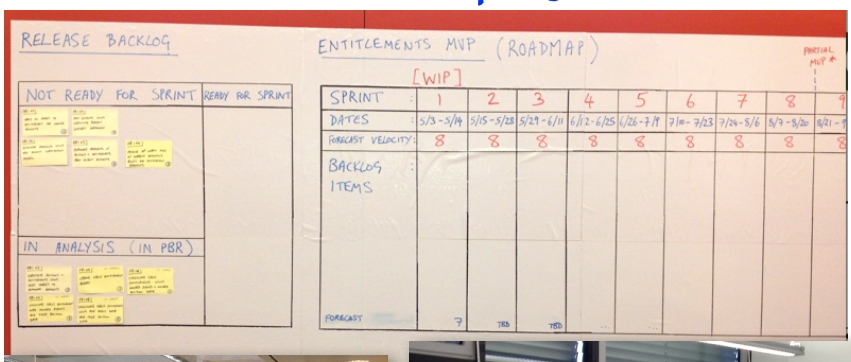
- no or delayed ROI
- lost market opportunity -> lower or no ROI
- hidden defects -> ...
- "care & feeding" costs
- reduction/stop of investment as a consequence of lack of visible progress -> no ROI
- opportunity cost
- scattered, loss of "focus"
- reduction in morale -> increase in attrition -> more costs; reduction in confidence in mgmt -> frictions -> more costs
- reduction in goodwill value
- increased capital carrying costs
- time value of money
- ...

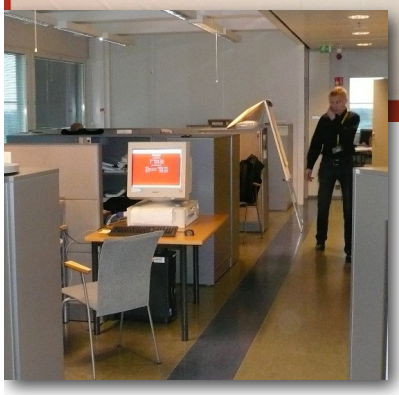
EXERCISE

- class discuss:
- How do single-function groups that can do one step "efficiently" and "fast" lead to more WIP/inventory?

Craig Larman

Lean Practices: Visual Management (with physical tokens)





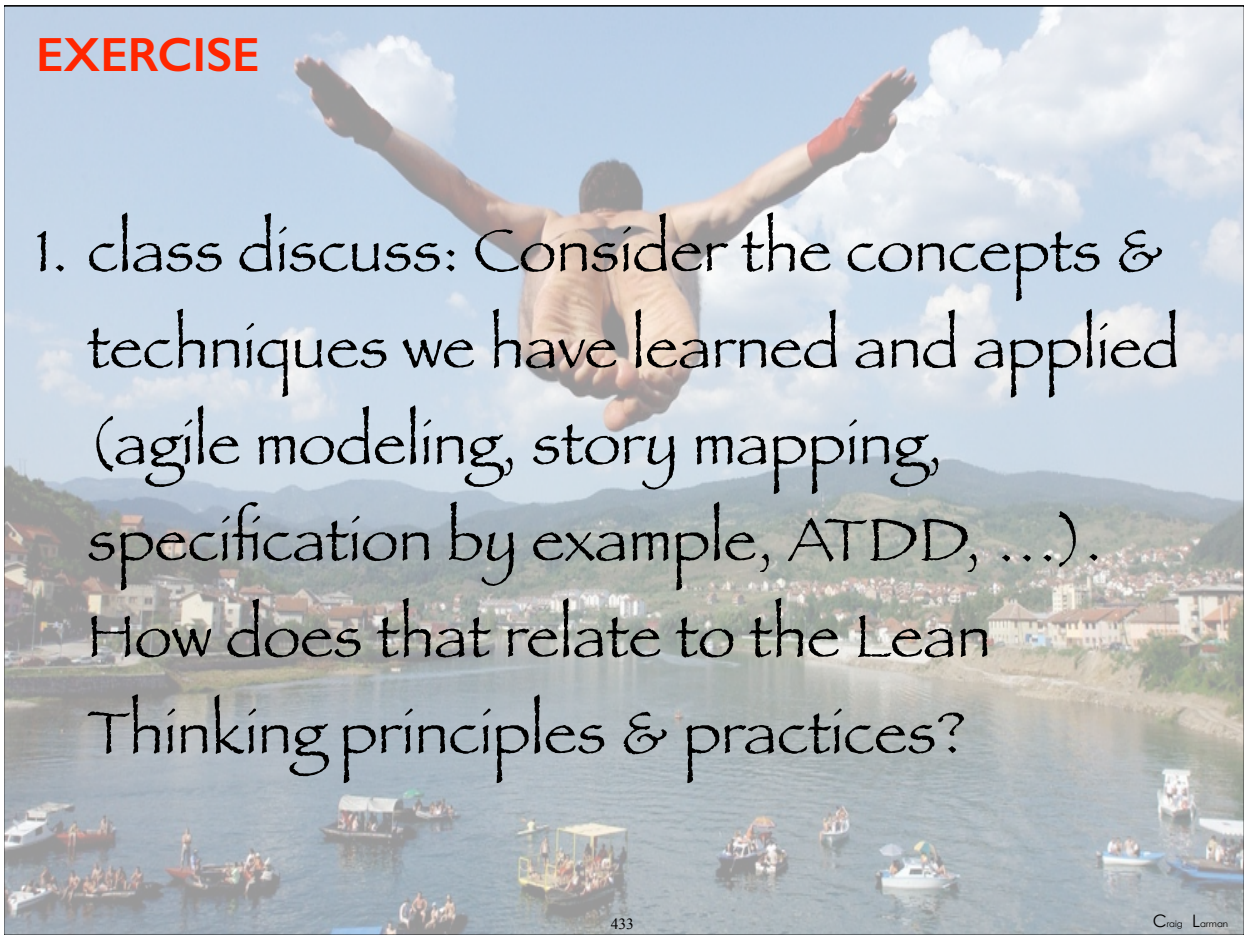
Lean Practices: "Big Team Room" (Obeya)
(& where there is Visual Management)

team members face each other

EXERCISE

1. team discuss: Without referring to notes, "round robin", each person in the team share one of the big ideas in lean thinking

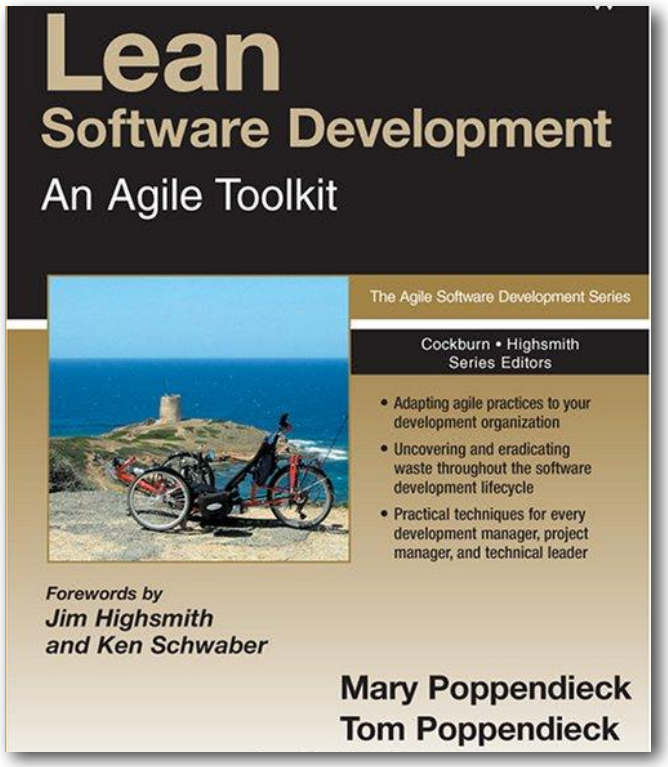
Craig Larman



EXERCISE

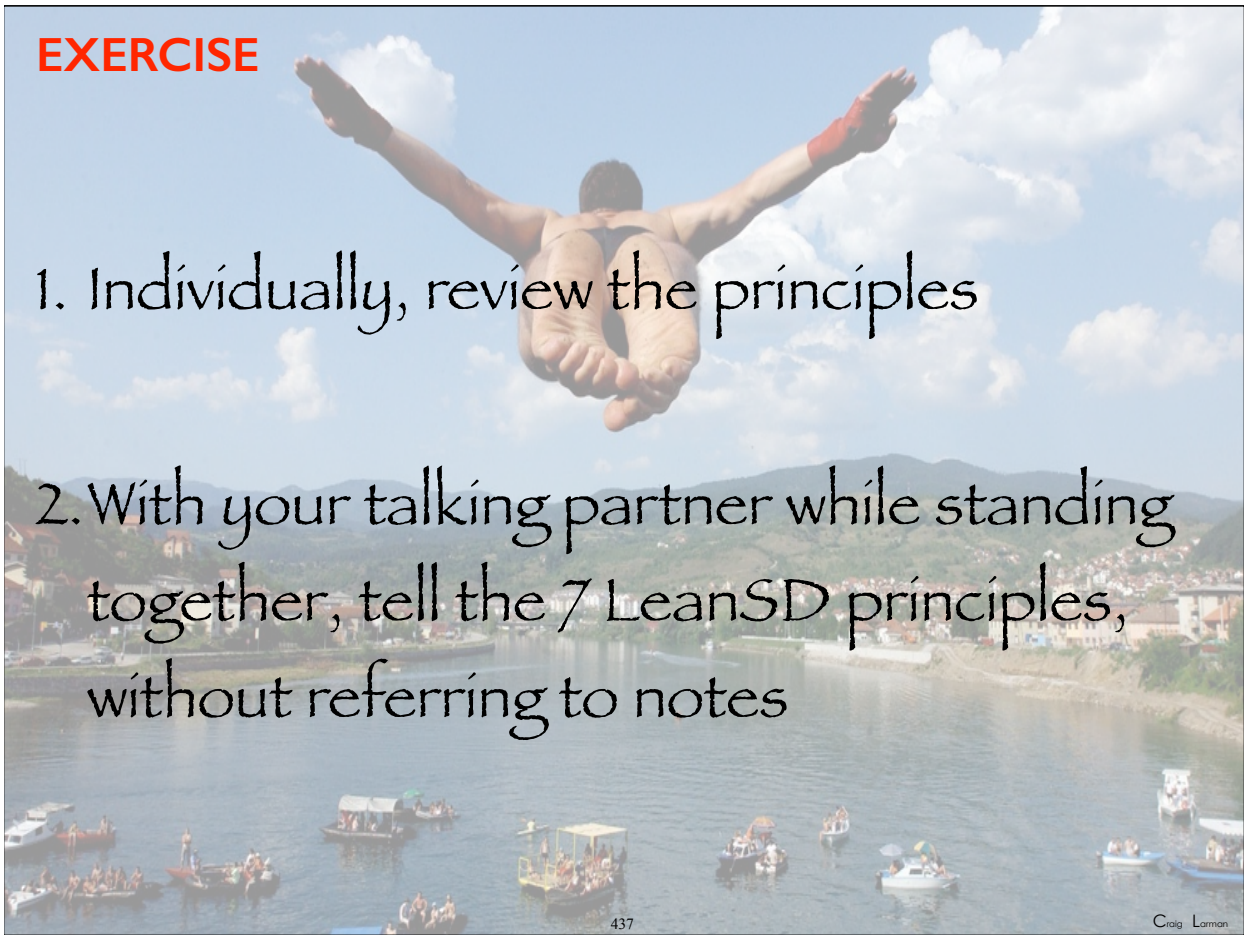
1. class discuss: Consider the concepts & techniques we have learned and applied (agile modeling, story mapping, specification by example, ATDD, ...). How does that relate to the Lean Thinking principles & practices?

Lean Software Development



LeanSD: 7 principles

- Eliminate waste
- Amplify learning
- Decide as late as possible
- Deliver as fast as possible
- Empower the team
- Build integrity in (= Zero Quality Control)
- See the whole



EXERCISE

1. Individually, review the principles

2. With your talking partner while standing together, tell the 7 LeanSD principles, without referring to notes

LeanSD Tools

- Eliminate waste
  - knowing & seeing the wastes
  - value-stream mapping
    - from when customer request enters & exits your company — not your silo

- Amplify learning
  - increase feedback and feedback loops
  - develop in short iterations, to "done"
  - synchronization
  - set-based development

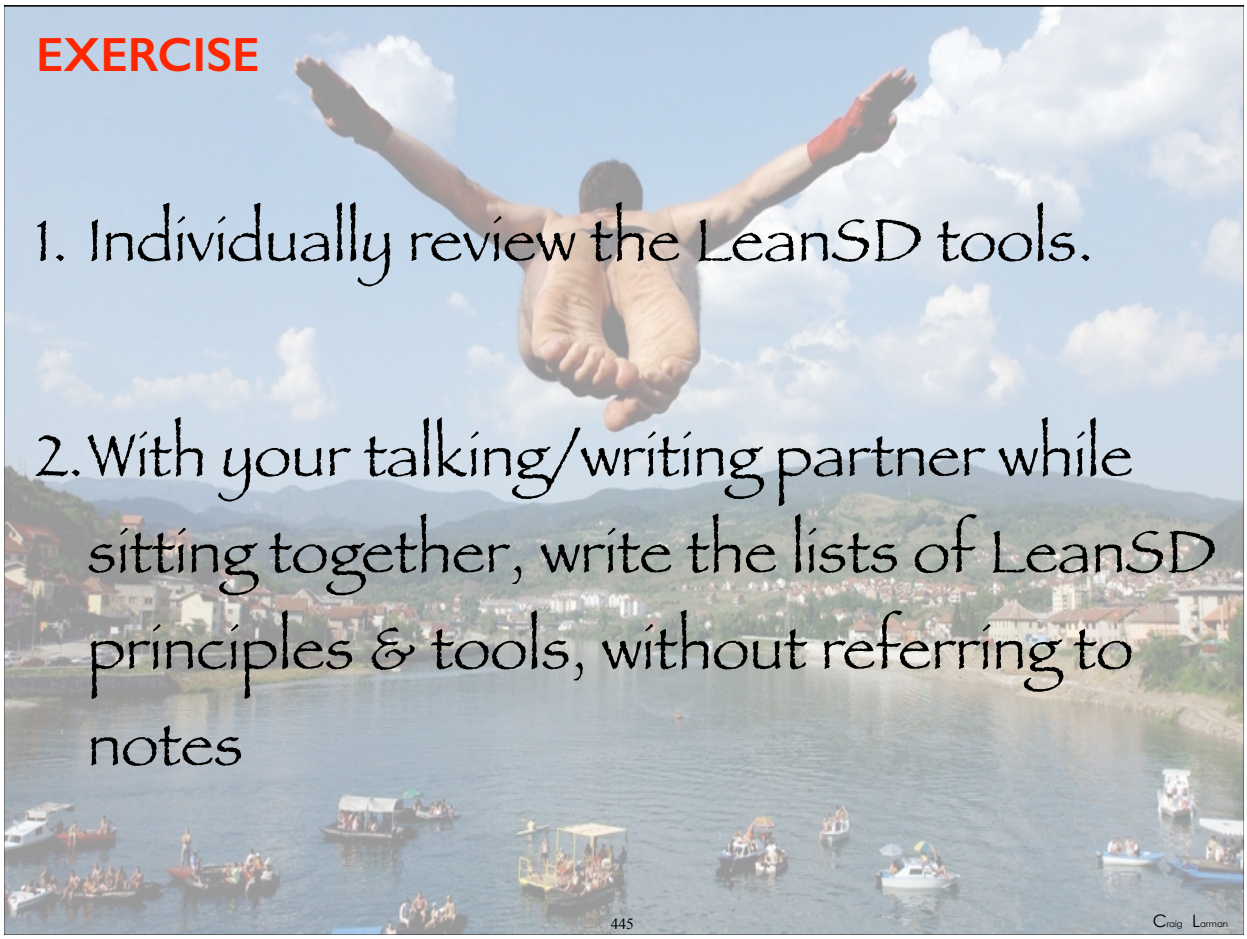
- Decide as late as possible
  - options thinking
  - identify the last responsible moment
  - learn decision-making concepts & practices

- Deliver as fast as possible
  - pull rather than push work, with visual management
  - apply the insights of queuing theory:
    - small equal-sized work packages
    - low WIP
    - steady rate of service
    - slack for each person & team

- Empower the team
  - self-determination (self-management)
  - understand motivation
  - respected leaders, not managers
  - culture of long-lived and valued master developers (not, "become a manager")
  - foster expertise (in programming)

- Build integrity in
  - perceived integrity via:
    - customer-written Acceptance TDD
    - short iterations of "done" + feedback from users
    - modeling (for feedback & alignment)
  - conceptual (architectural) integrity via:
    - user-centered design
    - design patterns
    - early & often architectural testing
  - constant refactoring
  - constant (automated)testing

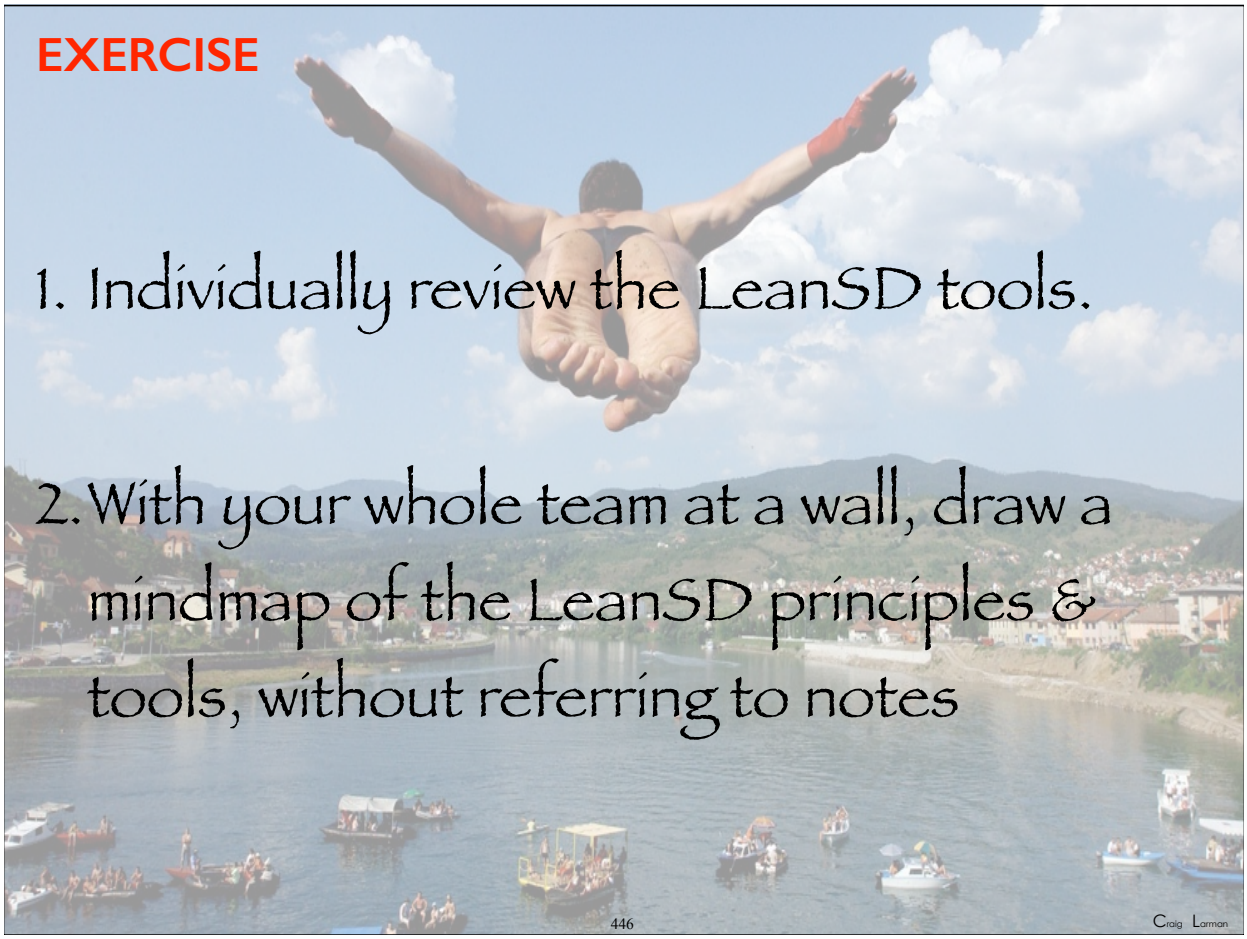
- See the whole
  - learn & practice systems thinking
  - beware bad measurements
    - local optimizing, meeting targets, …
    - measuring "performance" of people
  - value-stream mapping
  - know when a customer request enters & exits your company — not your silo



EXERCISE

1. Individually review the LeanSD tools.

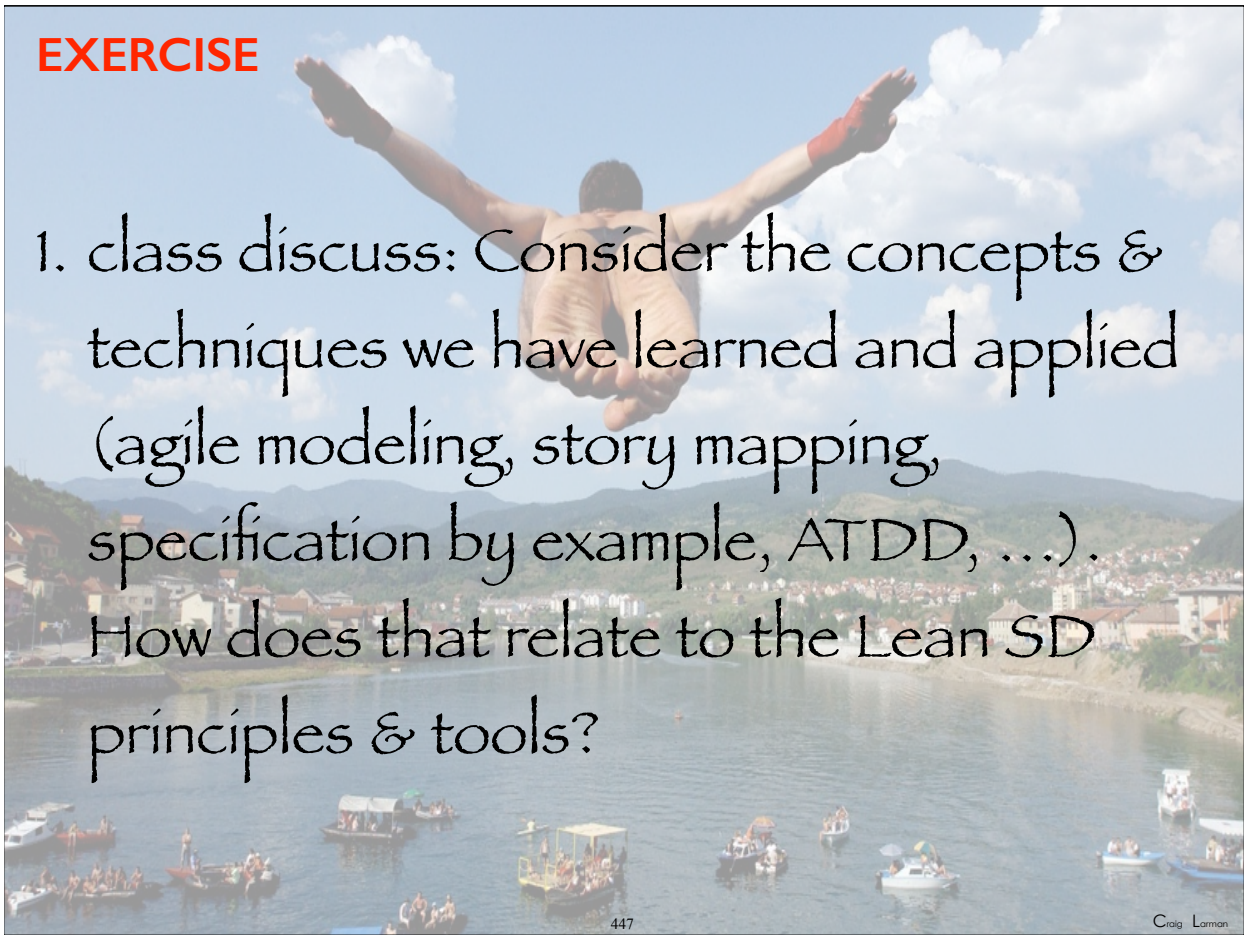
2. With your talking/writing partner while sitting together, write the lists of LeanSD principles & tools, without referring to notes



EXERCISE

1. Individually review the LeanSD tools.

2. With your whole team at a wall, draw a mindmap of the LeanSD principles & tools, without referring to notes



EXERCISE

1. class discuss: Consider the concepts & techniques we have learned and applied (agile modeling, story mapping, specification by example, ATDD, ...). How does that relate to the Lean SD principles & tools?

Agile Values & Principles

"adopt/do" Scrum

be Agile

"Agile"

➥ values & principles

not

practices

but many practices support these

Manifesto for Agile Software Development

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

Individuals and interactions over processes and 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.

What does a typical old-generation PROCESS ENGINEER, MANAGER, or AUDITOR look for?

- Processes and tools *over* individuals and interactions
- Comprehensive documentation *over* working software
- Contract negotiation *over* customer collaboration
- Following a plan *over* responding to change

the 4 agile values then expand into the 12 agile principles...

The 12 Agile Principles

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most effective method of conveying information to and within a development team is face-to-face conversation.

The 12 Agile Principles

7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self-organizing (self-managing) teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

agile approaches are

value-based

not practice-based

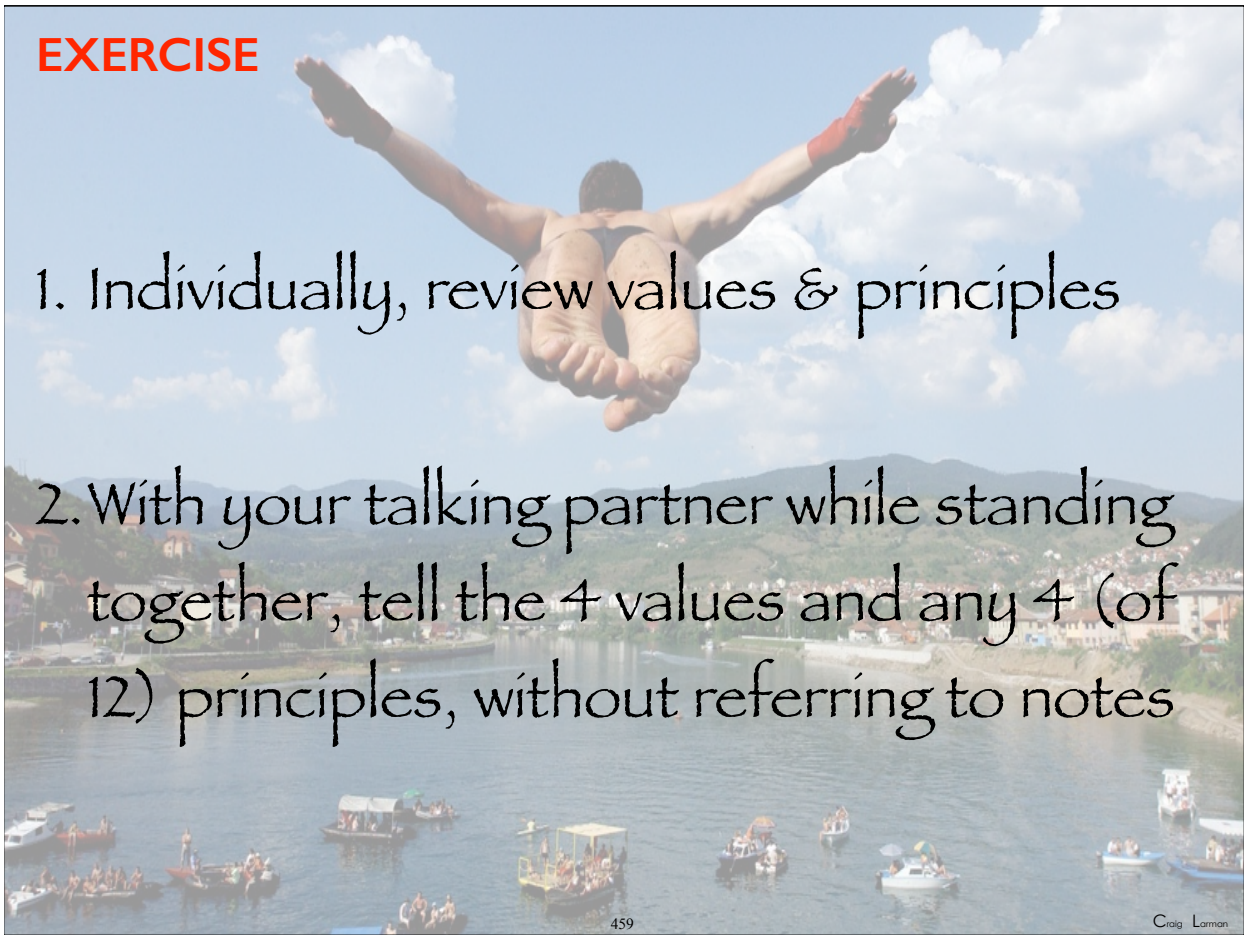
say “agile values”

avoid saying “agile practices”

Business Agility

enabled through

Organizational Agility



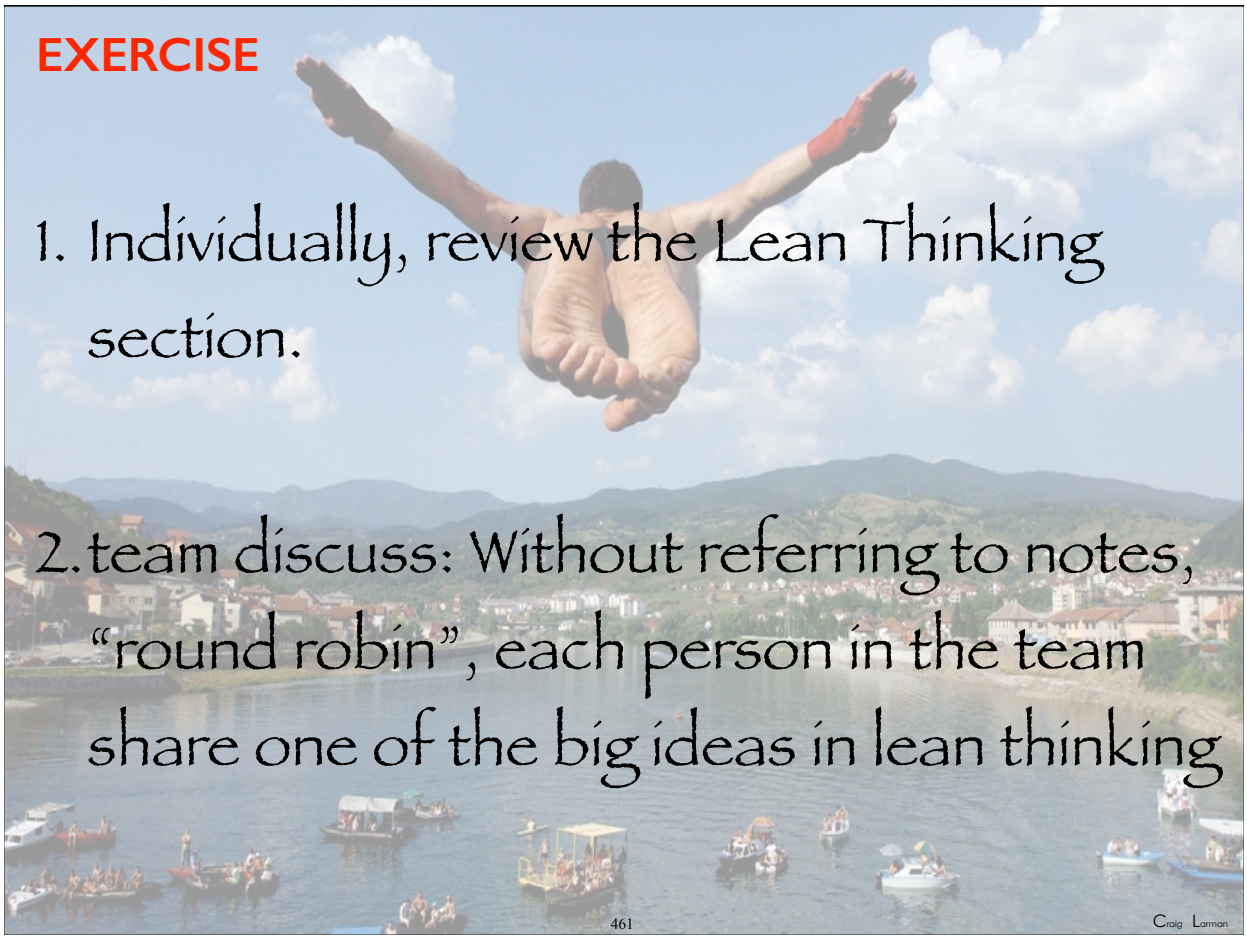
EXERCISE

1. Individually, review values & principles

2. With your talking partner while standing together, tell the 4 values and any 4 (of 12) principles, without referring to notes

Craig Larman

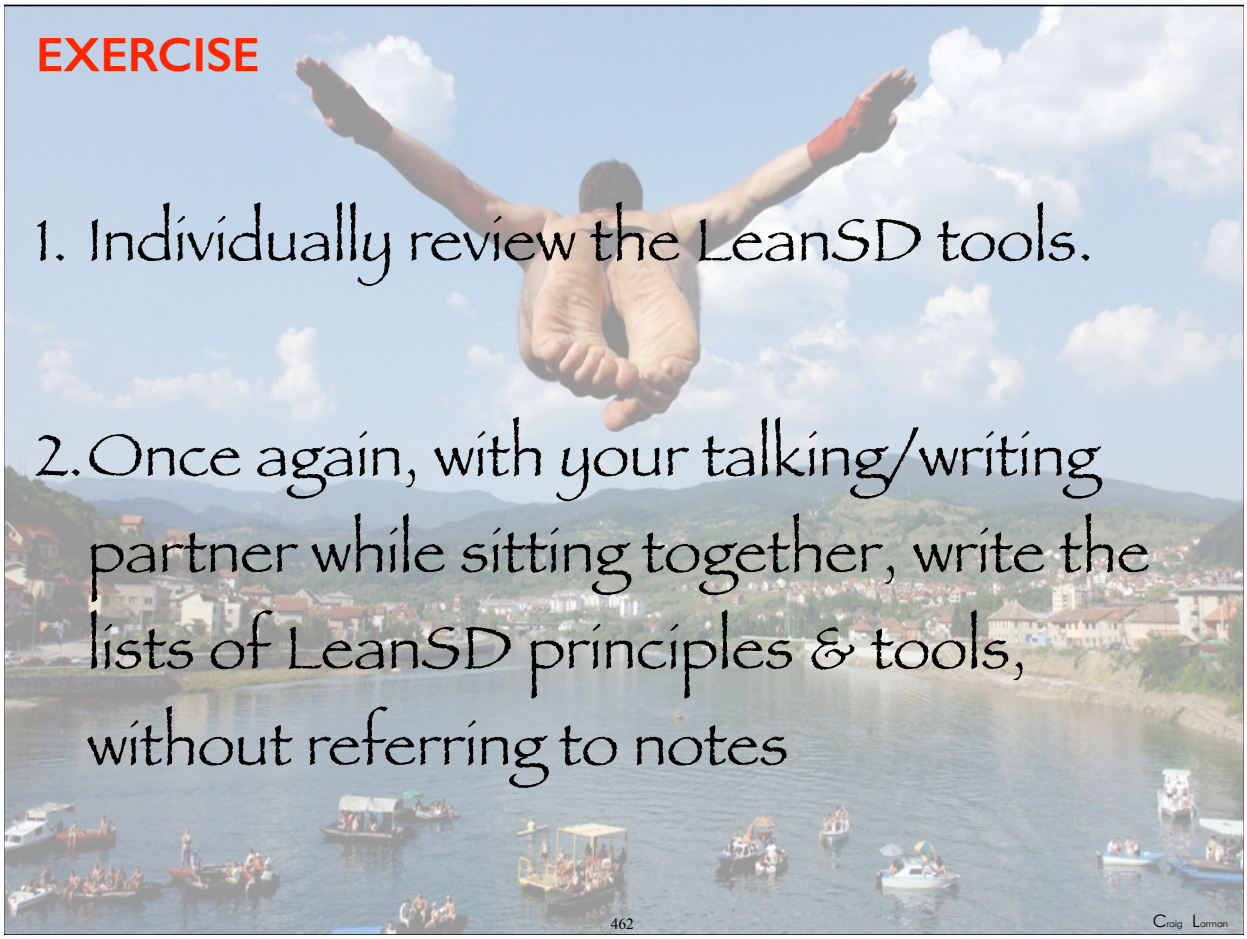
Agile & Lean Thinking Review



EXERCISE

1. Individually, review the Lean Thinking section.

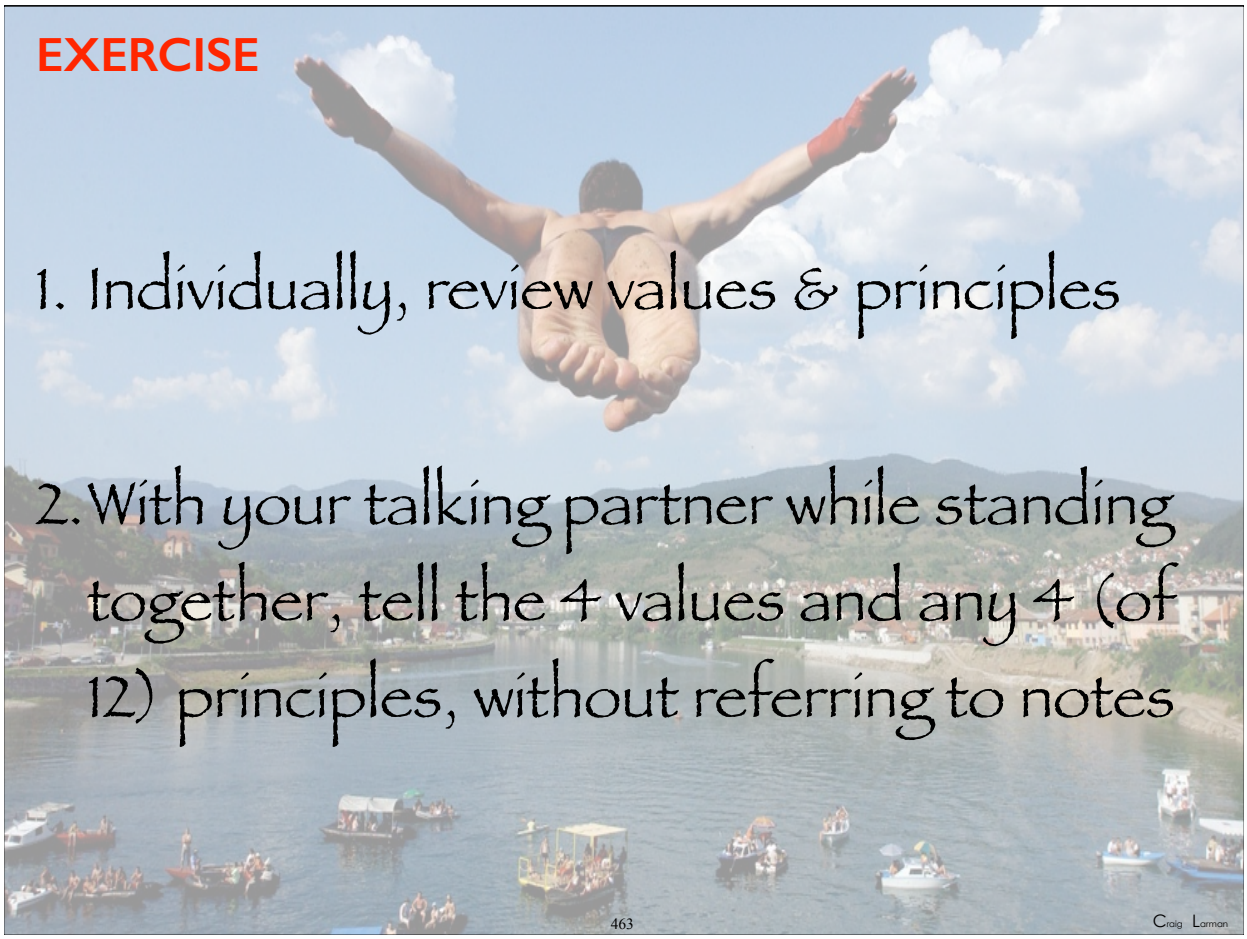
2. team discuss: Without referring to notes, "round robin", each person in the team share one of the big ideas in lean thinking



EXERCISE

1. Individually review the LeanSD tools.

2. Once again, with your talking/writing partner while sitting together, write the lists of LeanSD principles & tools, without referring to notes



EXERCISE

1. Individually, review values & principles

2. With your talking partner while standing together, tell the 4 values and any 4 (of 12) principles, without referring to notes

back to code and the next iteration …

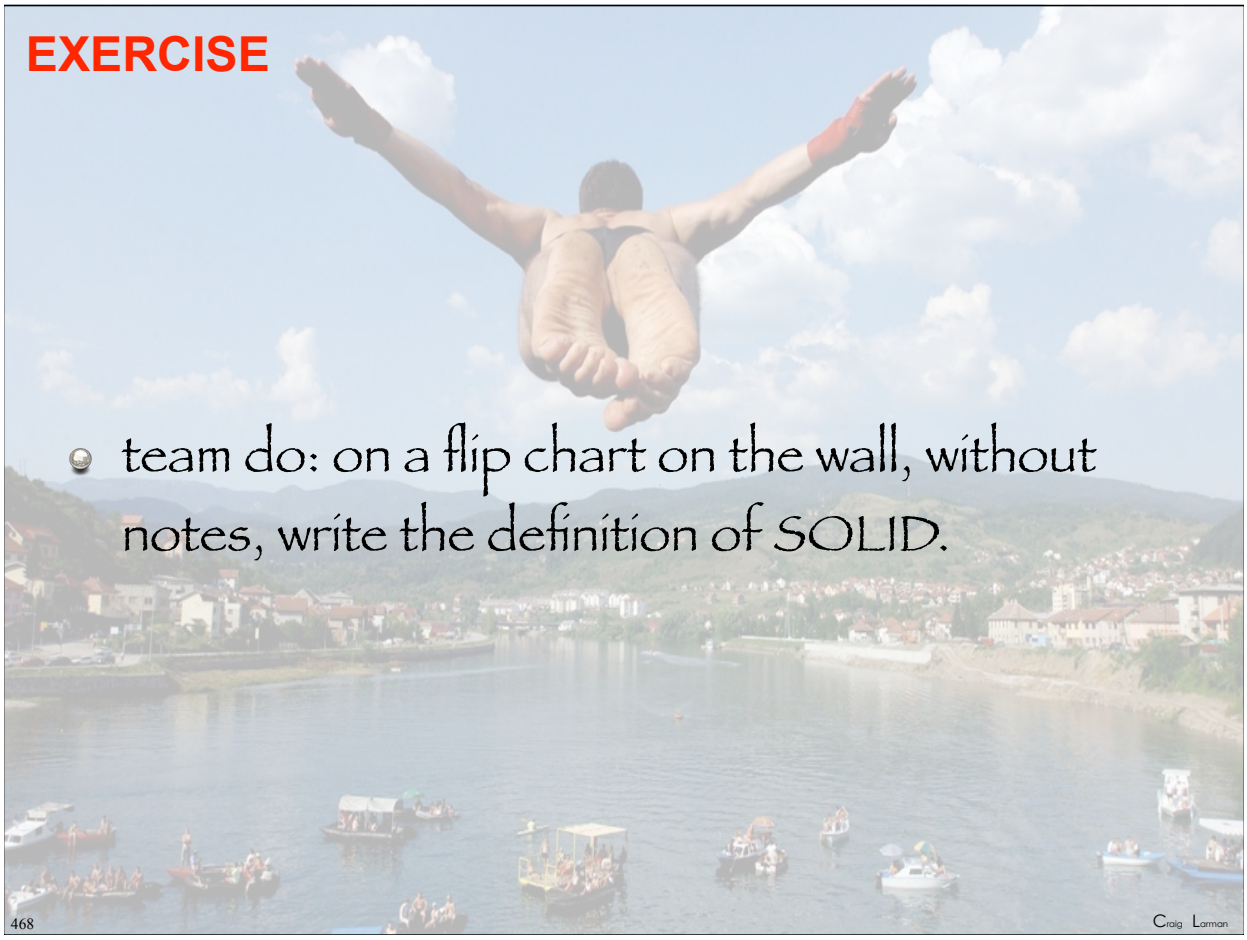
SOLID Principles

we have already informally visited some of the "SOLID" principles, and are introducing them formally because they are relatively well known and you may see or hear reference to them…

the SOLID principles (Uncle Bob Martin)

- S - Single responsibility principle
  - a class (and method) should have only a single responsibility.
- O - Open/closed principle
  - classes: open for extension, but closed for modification.
- L - Liskov substitution principle
  - objects should be replaceable with instances of subtypes without altering program correctness
- I - Interface segregation principle
  - many client-specific interfaces; not 1 general interface.
- D - Dependency inversion principle
  - Depend upon abstractions, not on concretions.
  - e.g., via dependency injection

EXERCISE

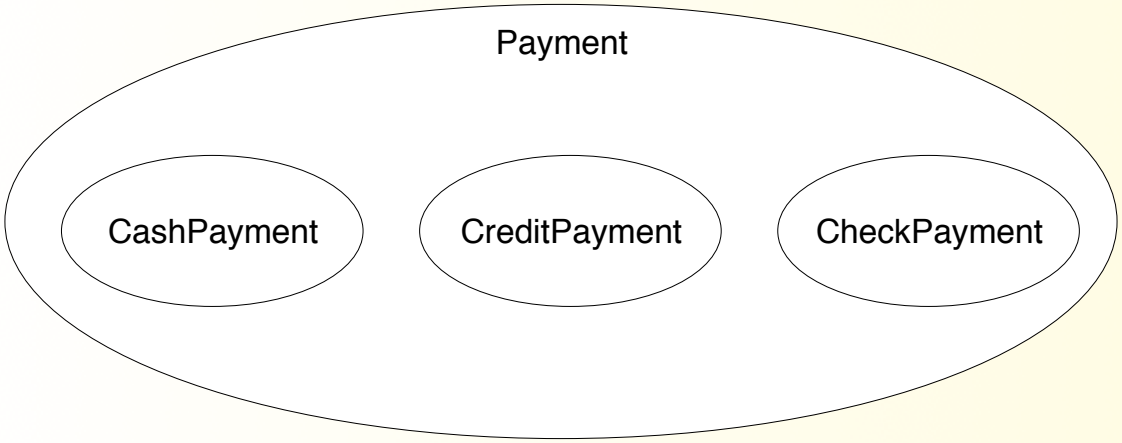


- team do: on a flip chart on the wall, without notes, write the definition of SOLID.

Generalization

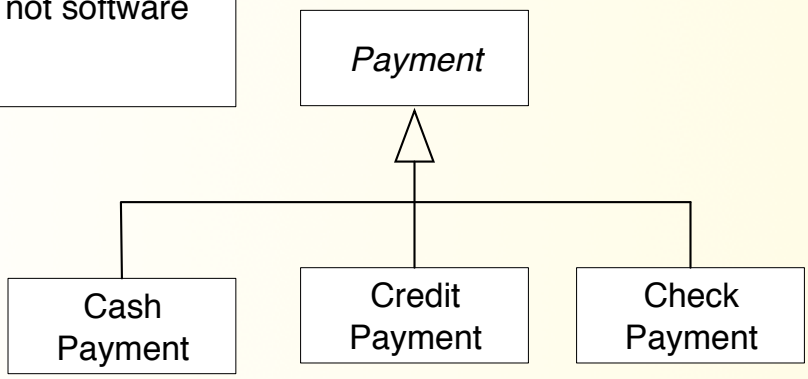
DEFINITION: generalization

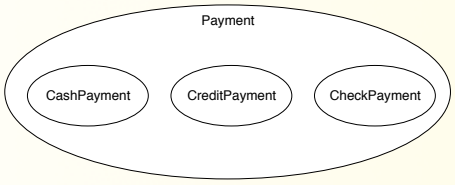
- Payment is a generalization
  - CashPayment is a specialization
- A Venn diagram of sets:



generalization in the domain model

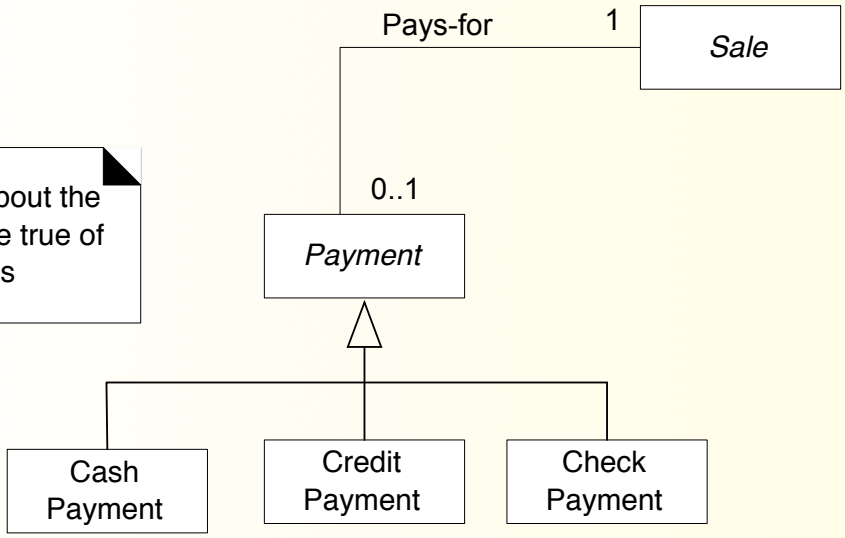
these are conceptual classes, not software classes





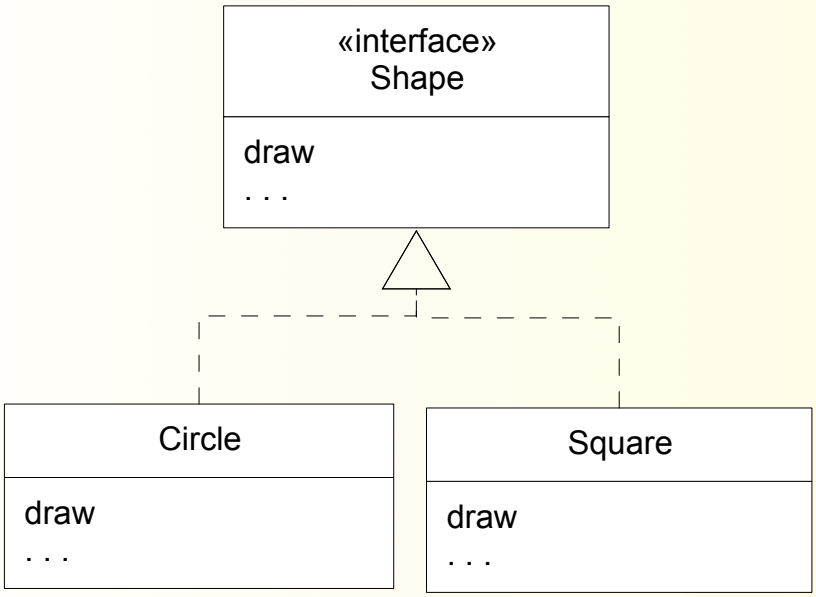
generalization in the domain model

Statements about the superclass are true of the subclasses



Polymorphism & Class Hierarchy Design

polymorphism as a design principle



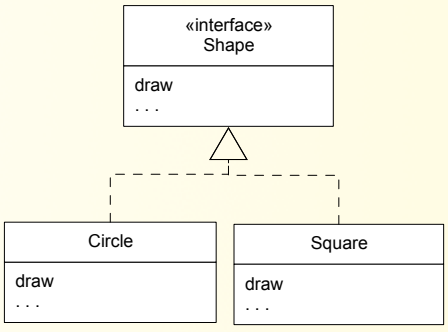
the GRASP core object-design principles

1. Information Expert
2. Creator
3. Controller
4. Low Coupling
5. High Cohesion
6. Polymorphism
7. Pure Fabrication
8. Indirection
9. Protected Variations

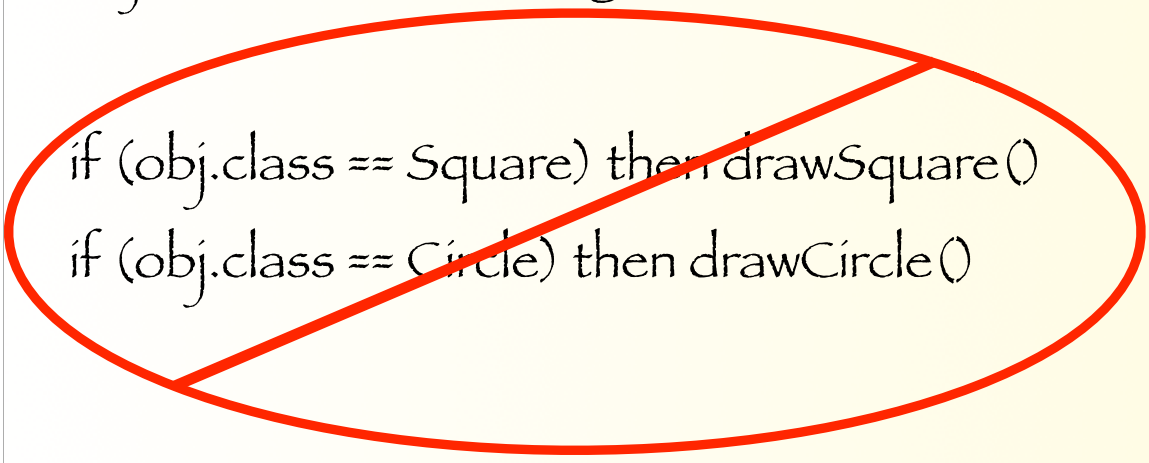
6. Polymorphism

- How to design for varying, but similar behaviors, that vary by type?
- Solution: Assign responsibilities for the similar behaviors using polymorphic operations in each type for which the behavior varies.
- for example…

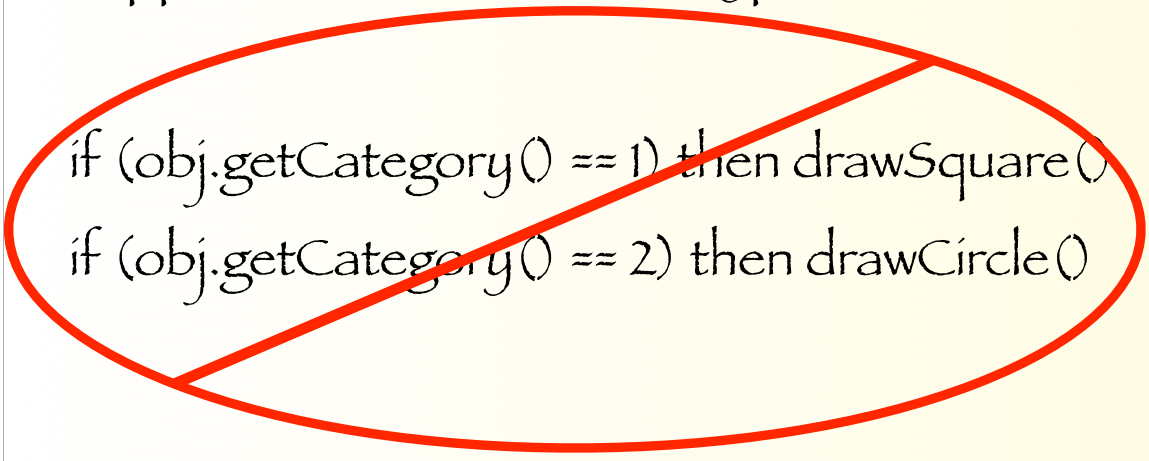
1.Information Expert
2.Creator
3.Controller
4.Low Coupling
5.High Cohesion
6.Polymorphism
7.Pure Fabrication
8.Indirection
9.Protected Variations



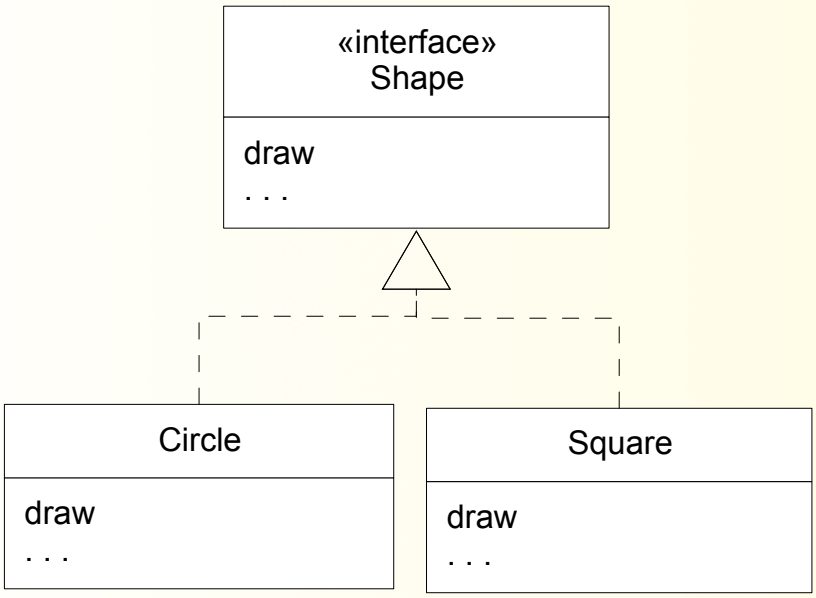
- Corollary: Do not test the type (class) of an object, and use case logic to handle the cases.



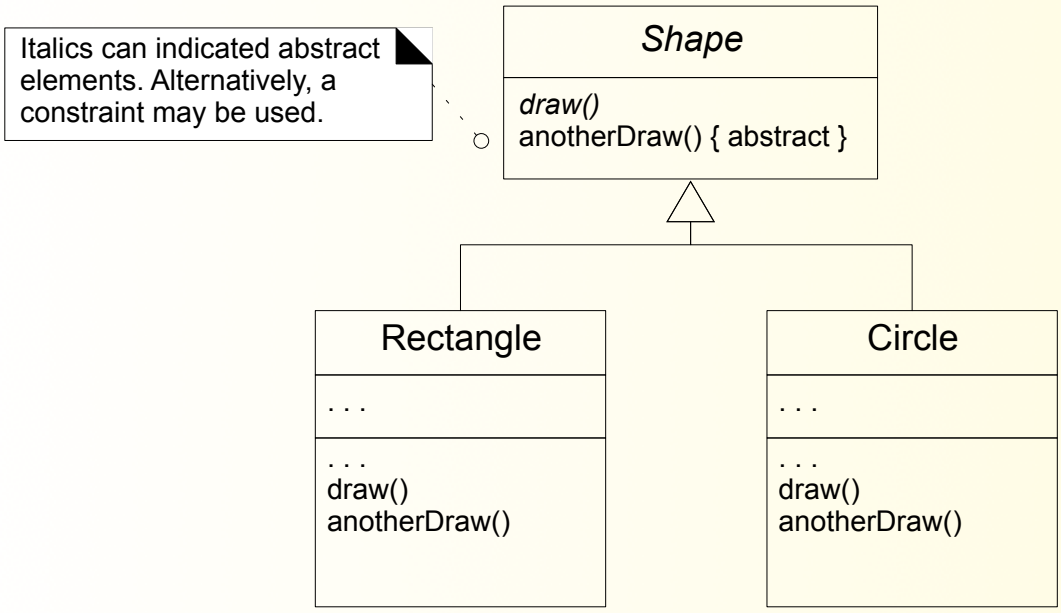
- "testing the type of the object" also (more subtly) happens with variations of "type codes"



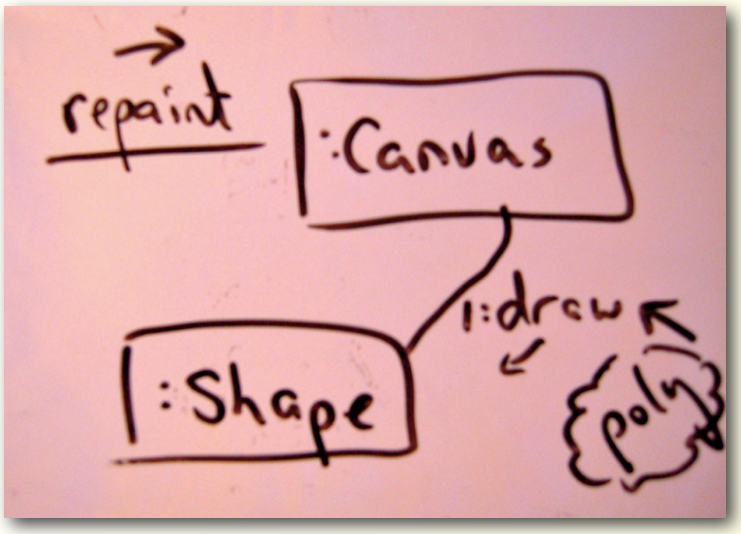
UML for polymorphism: interfaces



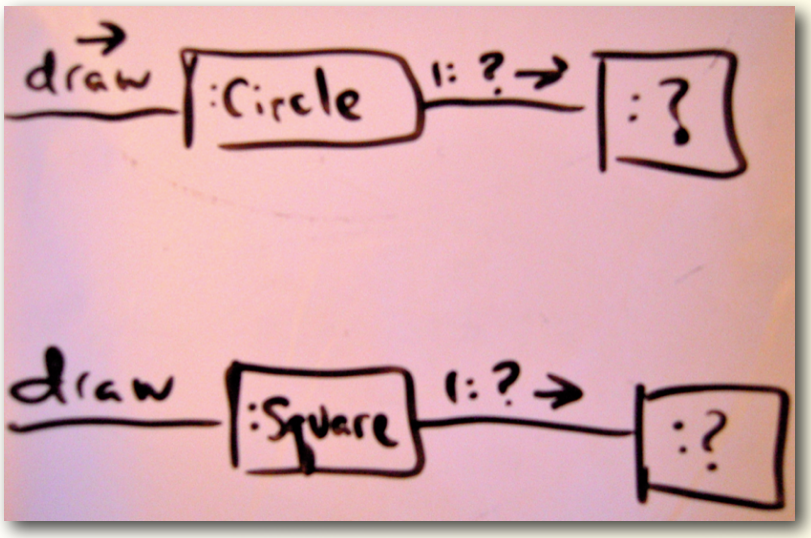
UML for polymorphism: abstract elements



UML for polymorphism



UML for polymorphism



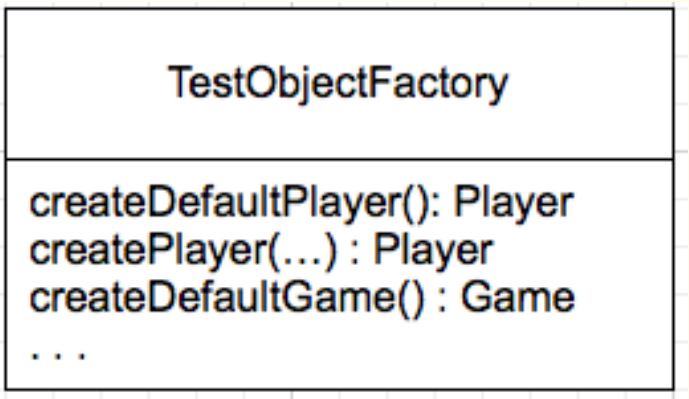
100% rule

only leafs are concrete

Simple Factory Pattern

pattern: Simple Factory

- Problem: duplication of object setup code in many tests:
- Solution: Define a "factory" helper class for creation and to eliminate duplication

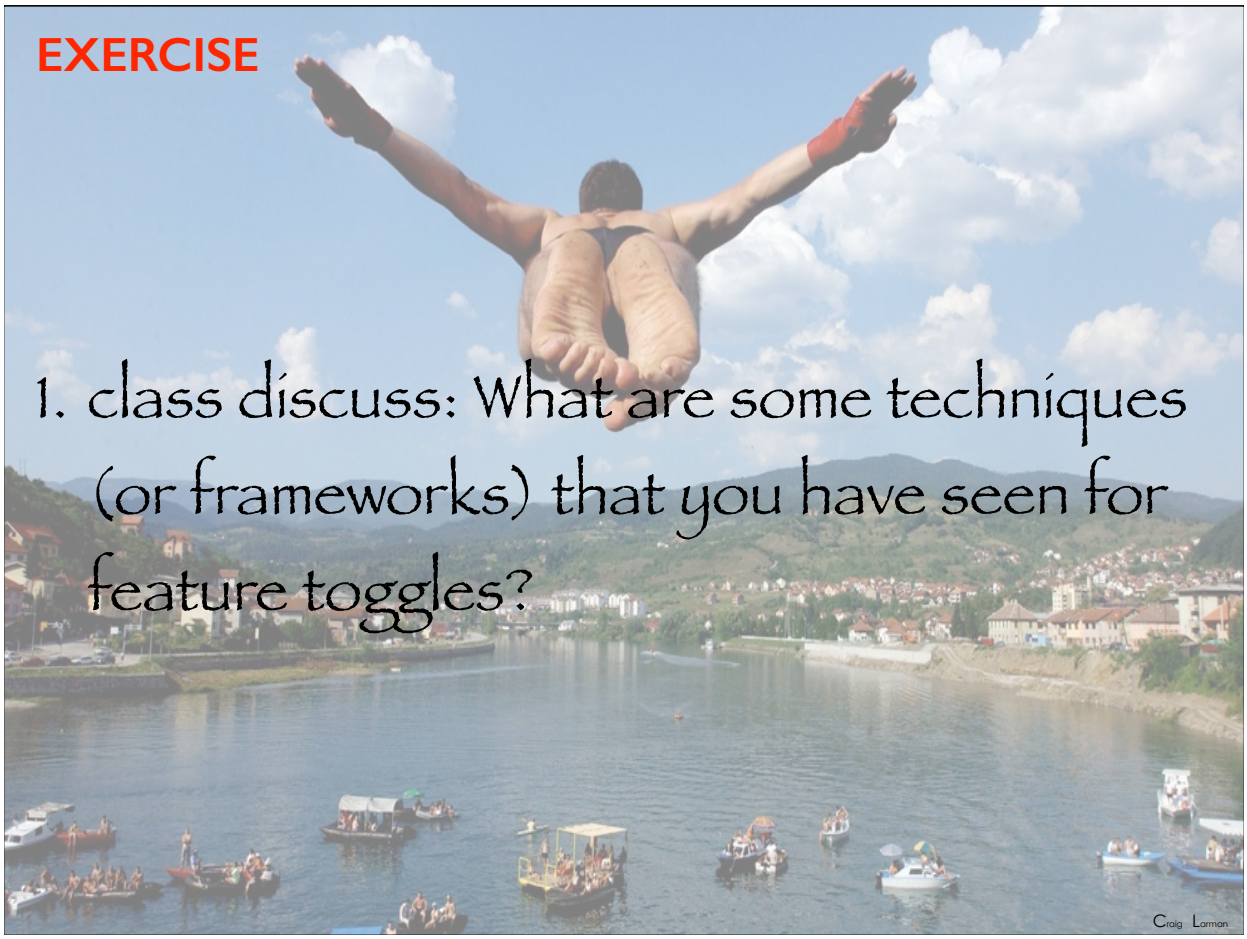


Feature Toggle Patterns

- important in incremental agile delivery
- important when avoiding branching (CI)
- also requires hiding UI elements

- AKA
  - feature configuration
  - feature/code switches/flippers/bits
  - "latent code" or "latent code patterns"

EXERCISE

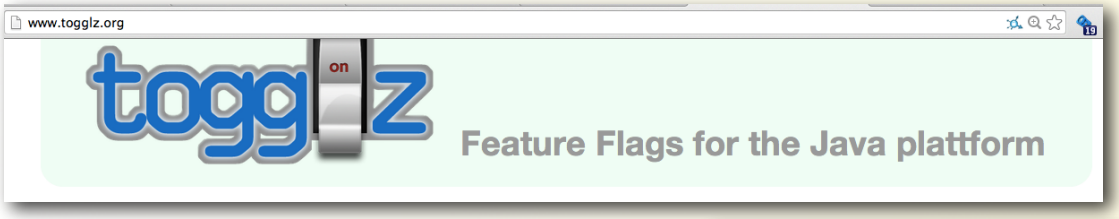


1. class discuss: What are some techniques (or frameworks) that you have seen for feature toggles?

Craig Larman

Feature Toggle Solutions

- there are a gazillion ways to skin this cat
- there are also FOSS frameworks…

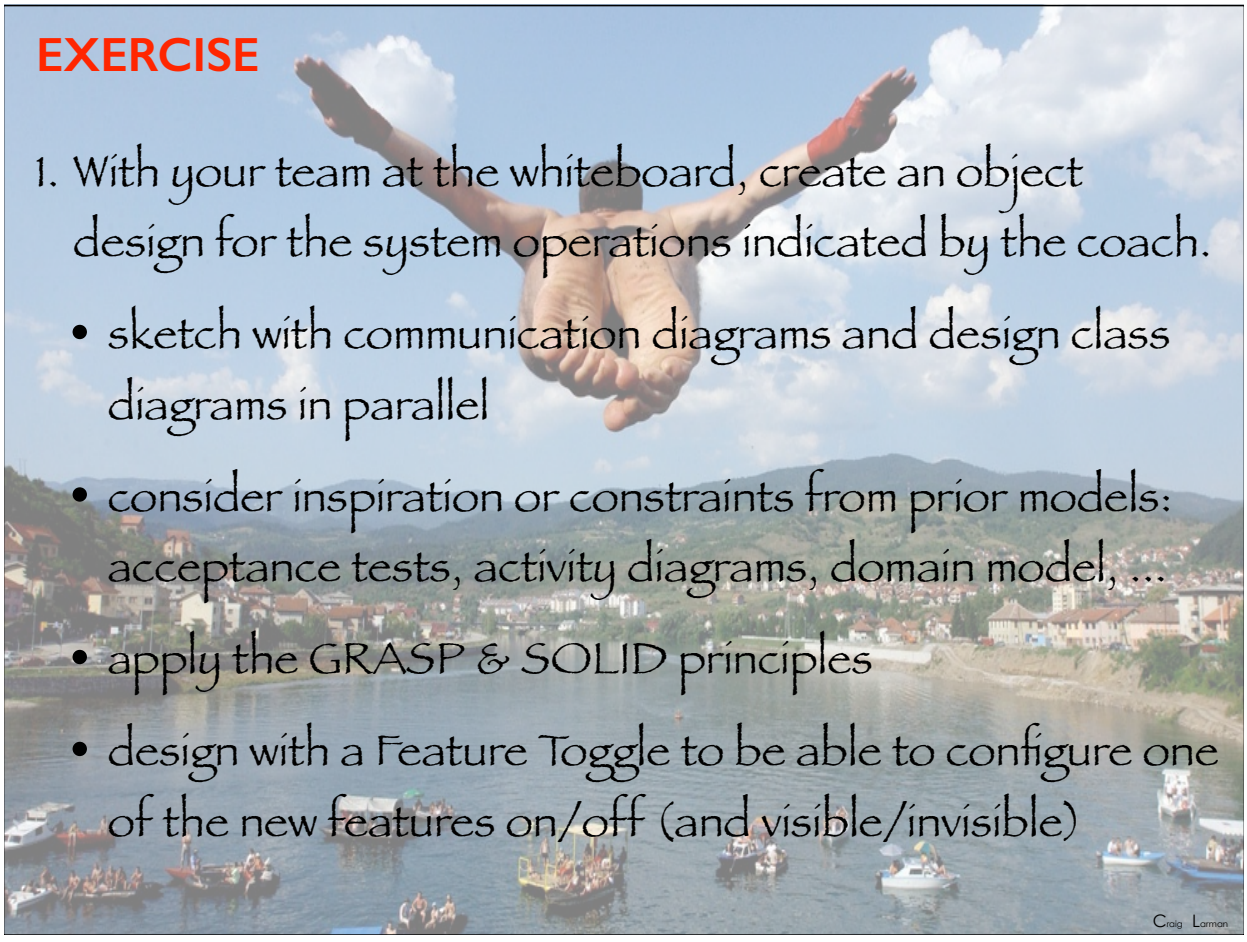


Spring profiles and feature toggle

Craig Larman

Exercise

EXERCISE



1. With your team at the whiteboard, create an object design for the system operations indicated by the coach.
   - sketch with communication diagrams and design class diagrams in parallel
   - consider inspiration or constraints from prior models: acceptance tests, activity diagrams, domain model, ...
   - apply the GRASP & SOLID principles
   - design with a Feature Toggle to be able to configure one of the new features on/off (and visible/invisible)

EXERCISE

GRASPing the rest

the GRASP core object-design principles

1. Information Expert
2. Creator
3. Controller
4. Low Coupling
5. High Cohesion
6. Polymorphism
7. Pure Fabrication
8. Indirection
9. Protected Variations

7. Pure Fabrication

- sometimes, Information Expert ("put services with data") leads to design with cohesion or coupling problems
- e.g., putting database persistence responsibilities inside "domain objects" is consistent with Information Expert, but are there cohesion/coupling problems?

1.Information Expert
2.Creator
3.Controller
4.Low Coupling
5.High Cohesion
6.Polymorphism
7.Pure Fabrication
8.Indirection
9.Protected Variations

7. Pure Fabrication

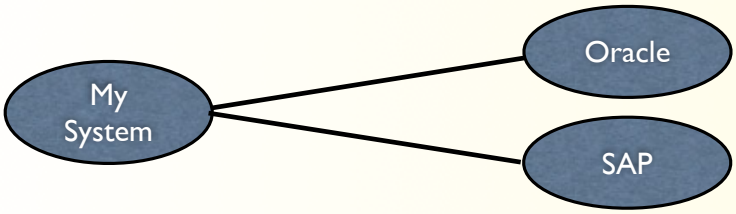
- so, we "make something up" -- a pure fabrication of our imagination to collect related responsibilities, usually supporting High Cohesion
- the class name will NOT be found in the Domain Model
- e.g., class DataAccessObject

1.Information Expert
2.Creator
3.Controller
4.Low Coupling
5.High Cohesion
6.Polymorphism
7.Pure Fabrication
8.Indirection
9.Protected Variations

8. Indirection

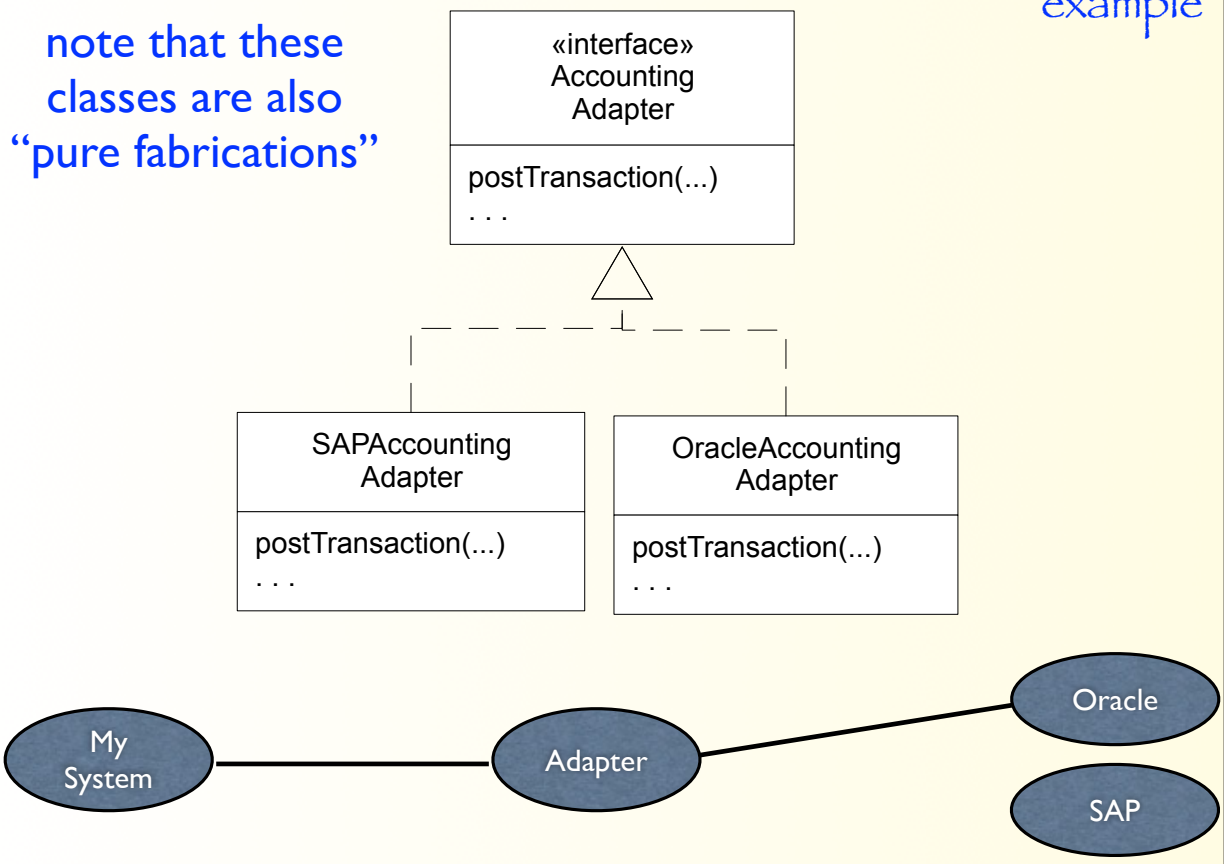
- when direct coupling is a problem (usually due to a variation point), consider a level of indirection

1.Information Expert
2.Creator
3.Controller
4.Low Coupling
5.High Cohesion
6.Polymorphism
7.Pure Fabrication
8.Indirection
9.Protected Variations



example

note that these classes are also "pure fabrications"



8. Indirection

- "Any problem in computer science can be solved with another layer of indirection." – David Wheeler
- It's an extremely common mechanism:
  - Operating System is an indirection
  - Virtual Machine is an indirection
  - . . .

1. Information Expert
2. Creator
3. Controller
4. Low Coupling
5. High Cohesion
6. Polymorphism
7. Pure Fabrication
8. Indirection
9. Protected Variations

8. Indirection

- many people don't know the 2nd sentence in Wheeler's quote...
- "Any problem in computer science can be solved with another layer of indirection. But that usually creates another problem."

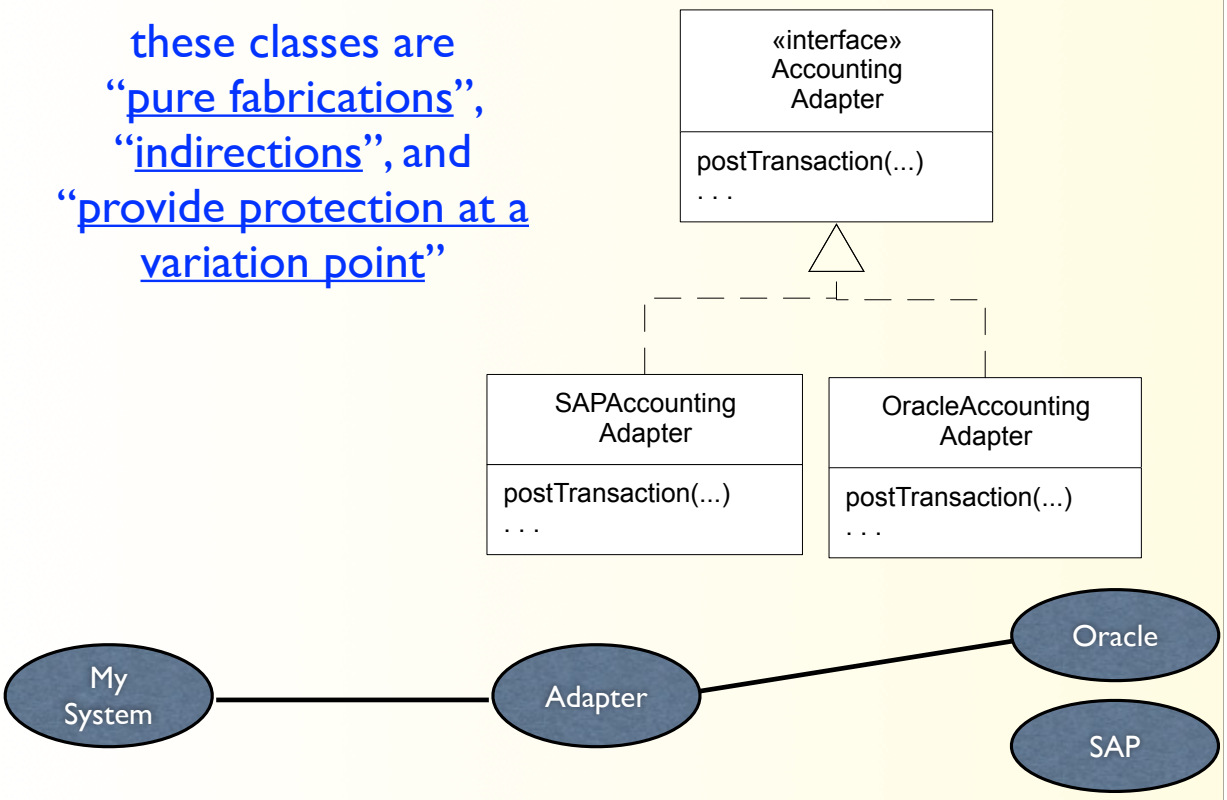
1.Information Expert
2.Creator
3.Controller
4.Low Coupling
5.High Cohesion
6.Polymorphism
7.Pure Fabrication
8.Indirection
9.Protected Variations

9. Protected Variations

- Problem: How to design components so that the variation or evolution points do not have undesirable impact on other components?
- Solution: Identify points of predicted variation and create a stable "interface" around them.
- key 'architectural' design pattern

1.Information Expert
2.Creator
3.Controller
4.Low Coupling
5.High Cohesion
6.Polymorphism
7.Pure Fabrication
8.Indirection
9.Protected Variations

these classes are "pure fabrications", "indirections", and "provide protection at a variation point"



- it's an extremely common mechanism:
  - most forms of indirection exist to protect at a variation point
  - low level: private attributes
  - medium: interfaces & polymorphism
  - high: virtual machines
  - global: standards

1. Information Expert
2. Creator
3. Controller
4. Low Coupling
5. High Cohesion
6. Polymorphism
7. Pure Fabrication
8. Indirection
9. Protected Variations

law of demeter (a protected variation)

- "law of demeter" is a guideline that provides protection when the variation point is "unstable object structure"...

1. Information Expert
2. Creator
3. Controller
4. Low Coupling
5. High Cohesion
6. Polymorphism
7. Pure Fabrication
8. Indirection
9. Protected Variations

law of demeter (a protected variation)

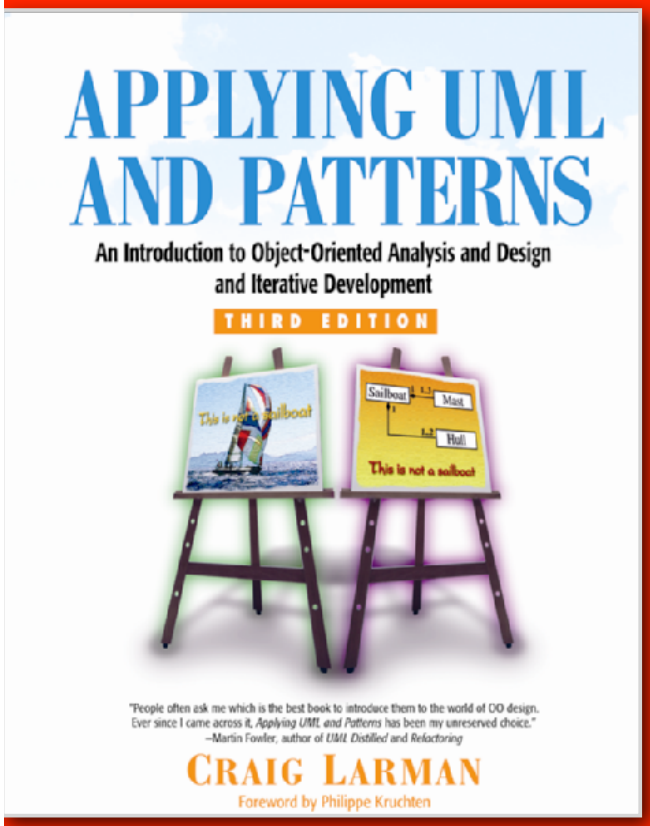
```
class Foo {
private Bar bar;

public void addGrep() {
bar.getX().getY().getZ().doFoo();
}
// ...
```

law of demeter (a protected variation)

```
// AKA "don't talk to strangers"
class Foo {
private Bar bar;

public void addGrep() {
  bar.doFoo();
}
// ...
```


law of demeter (a protected variation)

- Within a method, messages should be sent only to the following objects:
  - receiver object (e.g., "this", "self")
  - parameter of method
  - attribute of receiver
  - an element of a collection, which is an attribute of receiver
  - object created within method
- These are "familiars" of the receiver object.



1 Object-Oriented Analysis and Design 3
2 Iterative, Evolutionary, and Agile 17
3 Case Studies 41

PART II INCEPTION

4 Inception is Not the Requirements Phase 47
5 Evolutionary Requirements 53
6 Use Cases 61
7 Other Requirements 101

PART III ELABORATION ITERATION 1 — BASICS

8 Iteration 1—Basics 123
9 Domain Models 131
10 System Sequence Diagrams 173
11 Operation Contracts 181
12 Requirements to Design—Iteratively 195
13 Logical Architecture and UML Package Diagrams 197
14 On to Object Design 213
15 UML Interaction Diagrams 221
16 UML Class Diagrams 249
17 GRASP: Designing Objects with Responsibilities 271
18 Object Design Examples with GRASP 321
19 Designing for Visibility 363
20 Mapping Designs to Code 369
21 Test-Driven Development and Refactoring 385
22 UML Tools and UML as Blueprint 395

PART IV ELABORATION ITERATION 2 — MORE PATTERNS

23 Iteration 2—More Patterns 401
24 Quick Analysis Update 407
25 GRASP: More Objects with Responsibilities 413
26 Applying GoF Design Patterns 435

PART V ELABORATION ITERATION 3 — INTERMEDIATE TOPICS

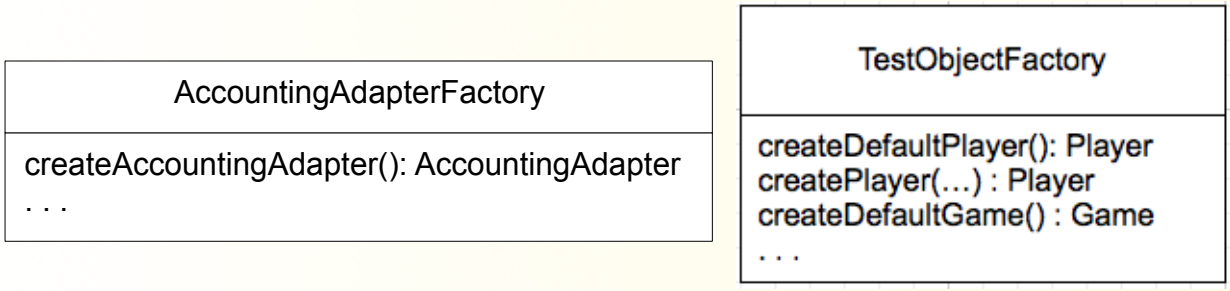
27 Iteration 3—Intermediate Topics 475
28 UML Activity Diagrams and Modeling 477
29 UML State Machine Diagrams and Modeling 485
30 Relating Use Cases 493
31 Domain Model Refinement 501
32 More SSDs and Contracts 535
33 Architectural Analysis 541
34 Logical Architecture Refinement 559
35 Package Design 579
36 More Object Design with GoF Patterns 587
37 Designing a Persistence Framework with Patterns 621
38 UML Deployment and Component Diagrams 651
39 Documenting Architecture: UML & the N+1 View Model 655

Simple Patterns

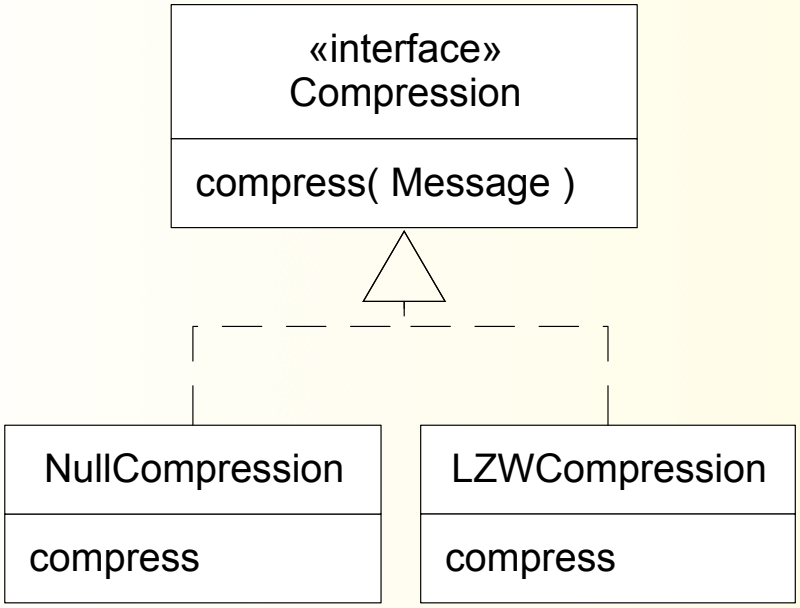
patterns?

pattern: Simple Factory

- Problem: Who should be responsible for creating objects when special considerations, such as:
  - complex or duplicated creation logic
  - creation of one instance from a related class hierarchy
  - pooling, …
- Solution: Define a "factory" class for creation.

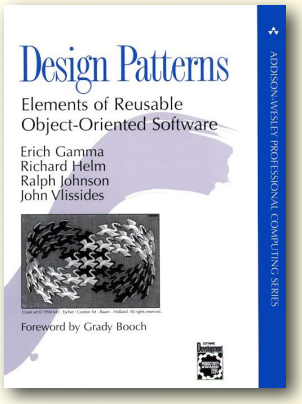


pattern: Null Object

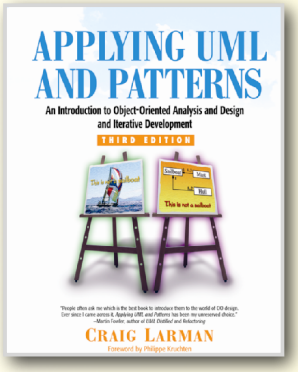


"Gang of Four" design patterns

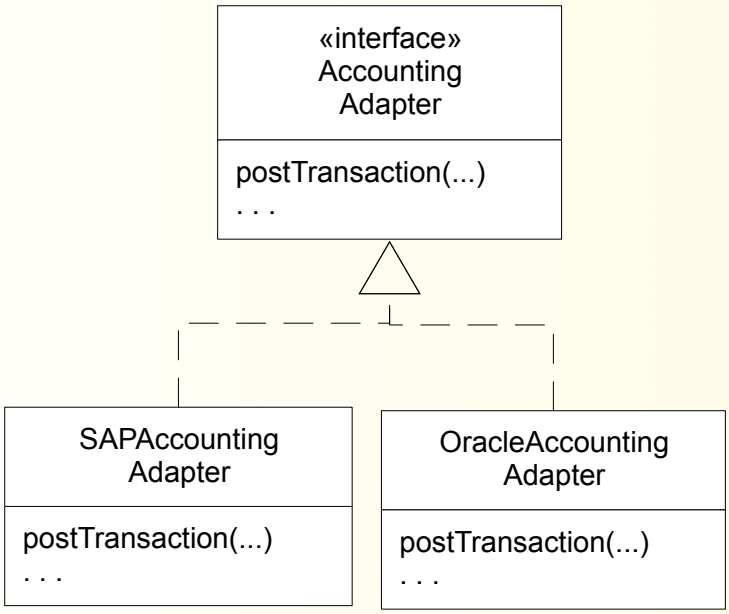
- 23 patterns
- 10-15 are widely used
- We will apply just a few



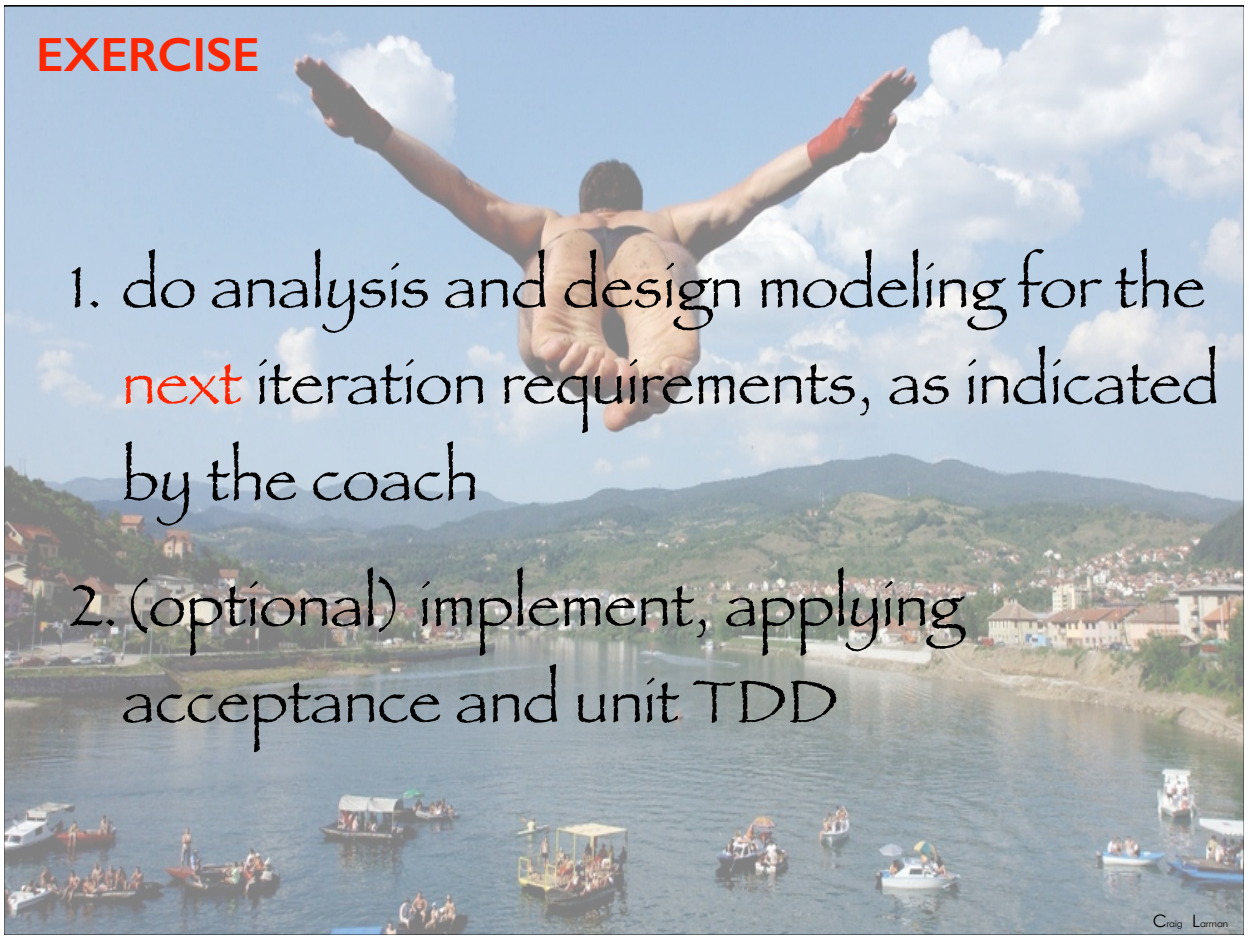
- The Agile Modeling & Advanced Object Design with Patterns workshop explores in detail



PATTERN: Adapter



Exercise


EXERCISE
1. do analysis and design modeling for the next iteration requirements, as indicated by the coach
2. (optional) implement, applying acceptance and unit TDD
Craig Larman

EXERCISE
1. (optional) implement, applying acceptance and unit TDD
Craig Larman

Closing

the story until now…

Topics

- Lean & Agile Modeling
- Lean & Agile Requirements
- User-Centered Design
- Story Mapping
- Telling Stories
- Splitting Features
- Learn with System-Sequence Diagrams
- Learn with a Domain Model
- Learn with Specification by Example
- Acceptance TDD
- BAD vs GOOD Automated Tests
- Designing with Layers
- Create Algorithms with Activity Diagrams
- GRASPing Object Design Principles
- Create Object Design with Communication Diagrams
- Create Object Design with Design Class Diagrams
- Continuous Integration
- Create Code Driven by Unit Tests (Unit TDD)
- Lean Thinking
- Lean Software Development
- Agile Values & Principles
- SOLID Principles
- Generalization
- Polymorphism & Class Hierarchy Design
- Feature Toggle Patterns
- Simple Patterns (Simple Factory, Null Object, …)

www.craiglarman.com