

Certified Scrum Developer

Technical Track

M. Mizanur rahman
mizan@techmastersbd.com

- 3 day technical track for CSD
- Divided into 3 sprints (1 daily) with sprint planning, review and retrospective
- Language to cover (with examples)
 - Java, .Net (C#), PHP, JavaScript

Soft copy of the slide will be provided to participants.

I am your co-learner



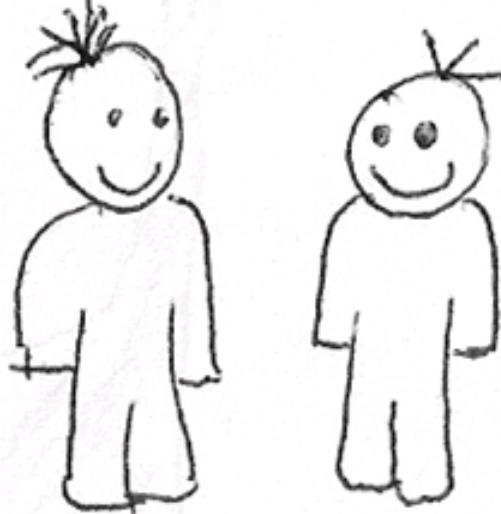
M. Mizanur Rahman
Head of Engineering
Telenor Health A/S

Lead Coach & Trainer
TechMasters Consulting LLC

Bengali, Learner, Novice Programmer,
problem solver, agile coach, practitioner
and community manager

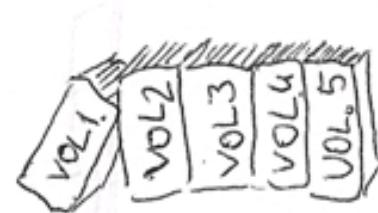


Agile Manifesto



Individuals and interactions over processes and tools

Agile Manifesto



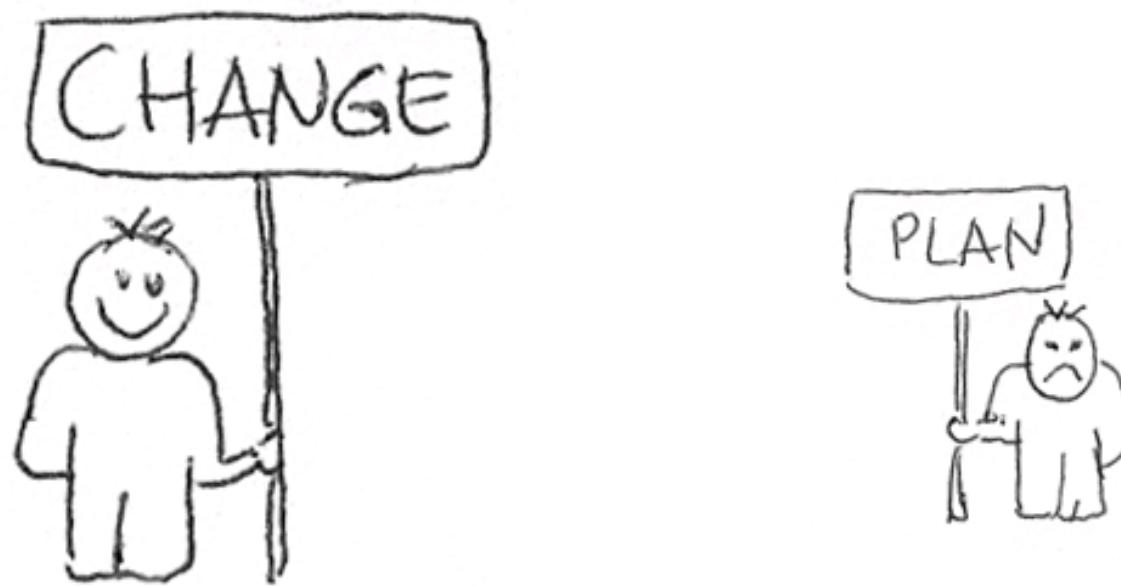
Working software over comprehensive documentation

Agile Manifesto



Customer collaboration over contract negotiation

Agile Manifesto



Responding to change over following a plan

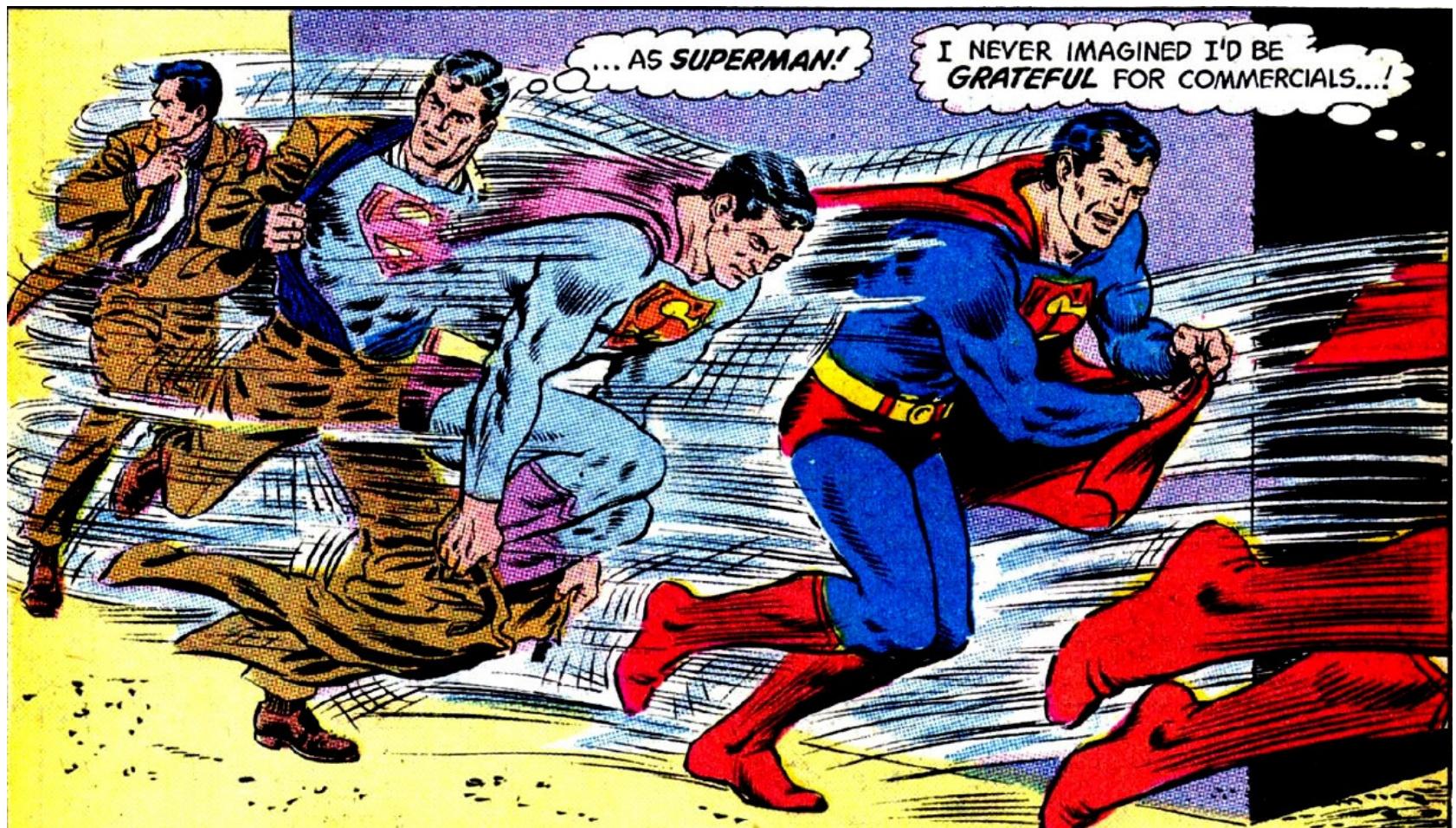
12 Agile Principles



#1: Customer satisfaction by rapid delivery of useful software



#2: Welcome changing requirements, even late in development



#3: Working software is delivered frequently (weeks rather than months)



#4: Close, daily cooperation between business people and developers



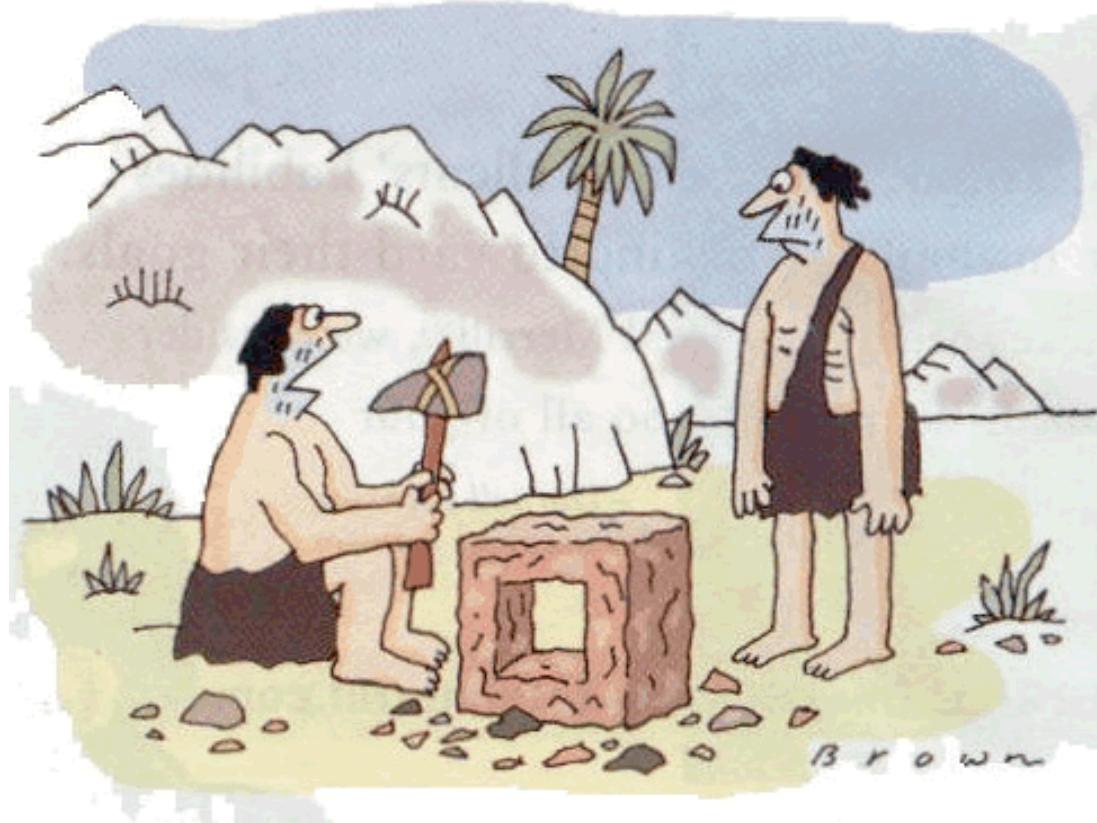
#5: Projects are built around motivated individuals, who should be trusted



#6: Face-to-face conversation is the best form of communication (co-location)



#7: Working software is the principal measure of progress



#8: Sustainable development, able to maintain a constant pace



#9: Continuous attention to technical excellence and good design



#10: Simplicity—the art of maximizing the amount of work not done—is essential



#11: Self-organizing teams



#12: Regular adaptation to changing circumstance



SCRUM AT A GLANCE

The Agile: Scrum Framework at a glance

Inputs from Executives,
Team, Stakeholders,
Customers, Users



Product Backlog



The Team



Sprint Planning Meeting



SCRUM VALUES



Collaboration

Necessity of Collaboration

3 out of 12 agile principles talks about collaboration, here are those

- Business people and developers must work together daily throughout the project.
- Projects need to be built around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Collaboration factors

- Feedback
- Communication
- Motivation

Feedback

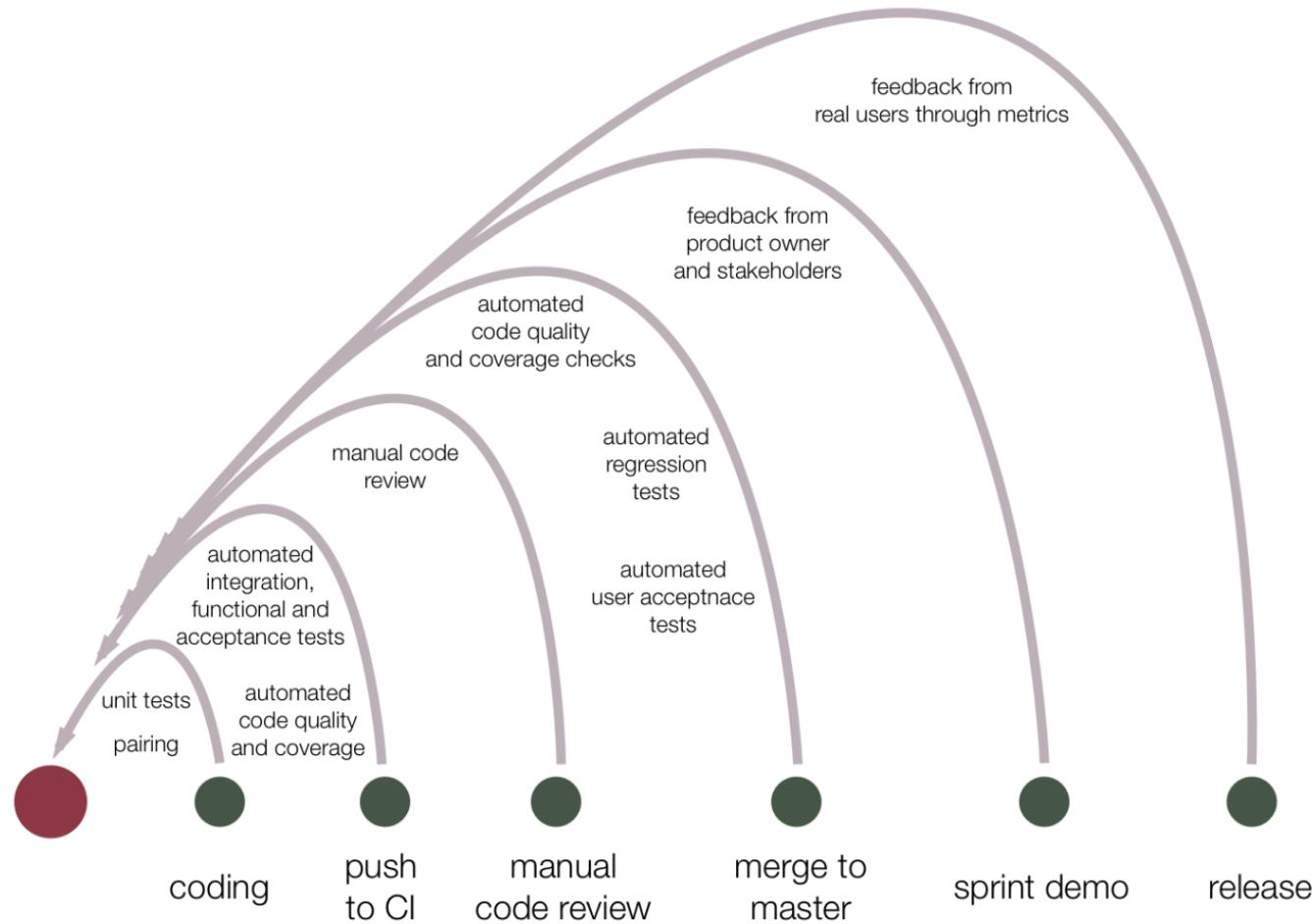
Agile team creates the environment for continuous feedback. Feedback should be provided such a way so that feedback

- Contains a clear purpose
- Is specific and descriptive
- Offers positive alternatives

Feedback events

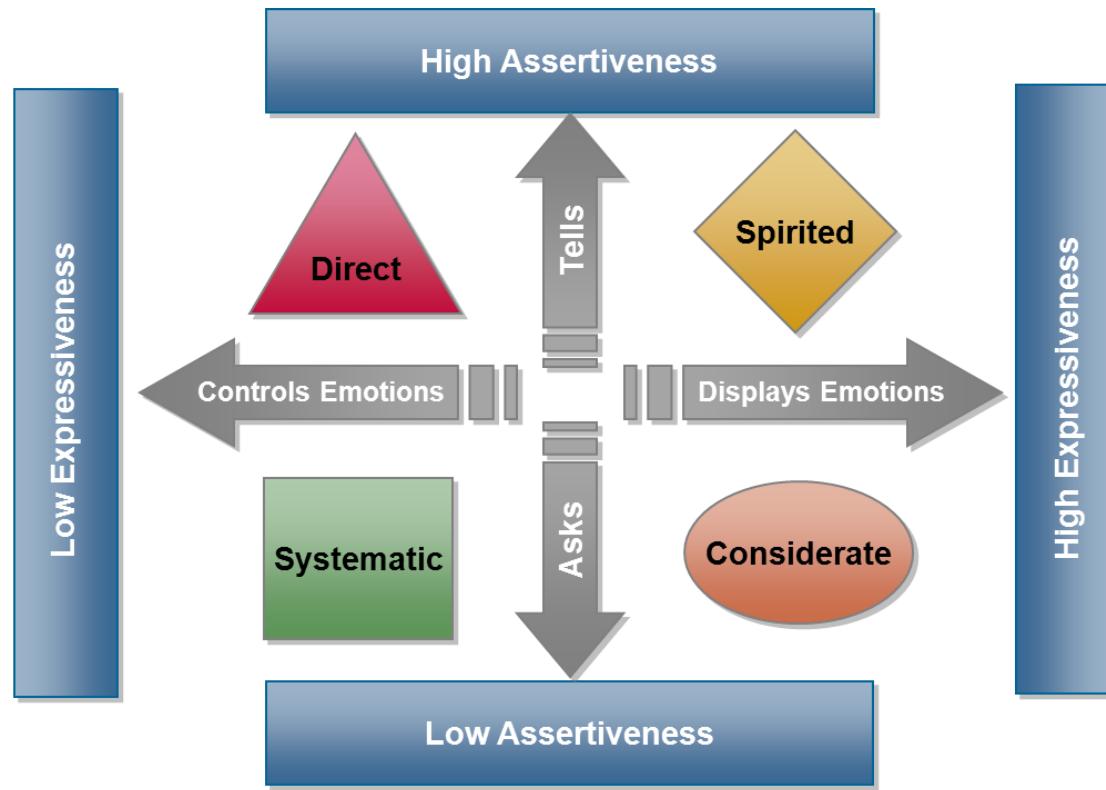
- Pair programming
- Unit tests
- Continuous integration
- Code review
- ATDD & BDD
- Daily standup
- Sprint review
- Sprint retrospective

Feedback for development team



Communication

it is important to deliver information in a manner that is understood by the receiver



Motivation

Motivation is built on

- Encouragement
- Partnership
- Compromise
- Trust
- Empowerment
 - Authority
 - Responsibility
 - Self-directedness

Include customer in the process

- Customer becomes a part of the development process
- Customer must attend events where feedback is generated (planning, review etc)
- Customer should not just write the specs and throw it to the team. It is a collaboration between team and customer to make a great product
- Customer should be involved from the inception of the project till the end.

Opportunity for team and customer to collaborate

- Sprint planning
- Daily Scrum
- Sprint review
- Sprint retrospective
- Product backlog refinement

GIT (VCS)

Collaboration within the team

What is a Version Control System (VCS)?

- A way to keep track of changes to files (& folders)
- Between multiple authors (developers)
- A record of who did what, when
- Why is provided by commit messages!
- Non-distributed (Subversion, CVS)
 - Server has the master repo, all commits go to the server
- Distributed (Git, Mercurial)
 - Server has the master repo, but you have a copy (clone) of the repo on your machine
 - CVS, Visual Source Safe, Perforce are some others

Git is (a VCS that is)...

- An open source VCS designed for speed and efficiency
- Better than competing tools
- Created by Linus Torvalds (for managing Linux kernel)
- Your best insurance policy against:
 - Accidental mistakes like deleting work
 - Remembering what you changed, when, why
 - Your hard drive crashes

- The version control solution I recommend to manage code of all types
- Hosted solutions include Github.com (more widespread, more expensive) or Bitbucket.com (cheaper, integrates with Jira)
- or running your own Git server

Getting Git

- Download the software - it's free
 - <http://git-scm.com/downloads>
 - or on Mac ([homebrew](#)), \$ brew install git
- Download a GUI, optional
 - <http://git-scm.com/downloads/guis>
- Read the manual and/or the book
 - <http://git-scm.com/docs>
 - <http://git-scm.com/book>

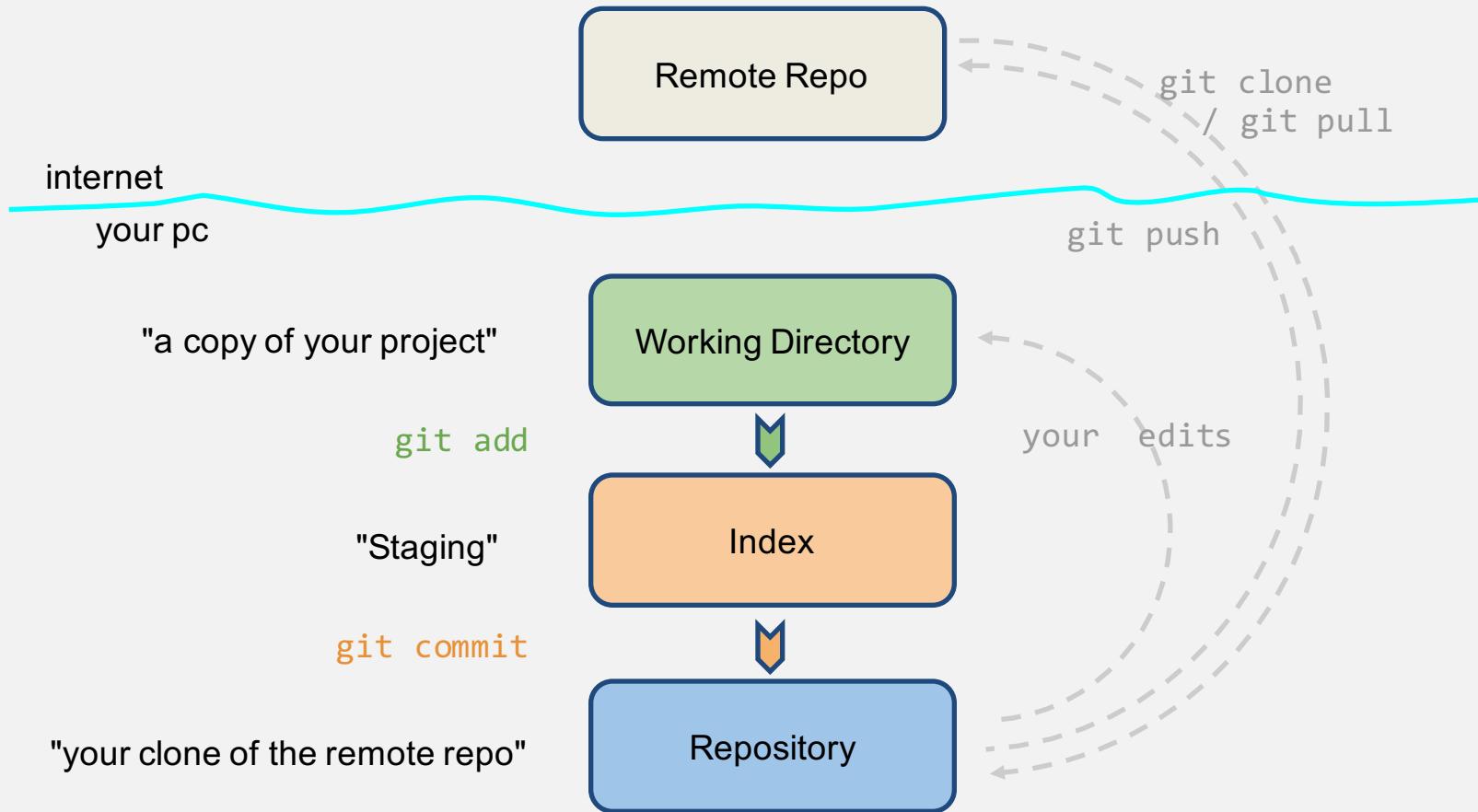
How does Git compare?

- An open source, distributed version control software designed for speed and efficiency
- Unlike Subversion, Git is distributed.
- This means that you can use Git on your local machine without being connected to the internet.
- Revisions (commits) you make to your local repository are available to you only
- The next time you connect to the internet, push your changes to a remote repository to share them & back them up offsite & off your computer

- The distributed nature of Git makes it insanely fast, because most things you do happen on your local machine
- The local nature of Git makes it effortless to create branches to isolate your work
- The local nature of Git makes it possible to coalesce a series of changes (local commits) into a single commit on the remote branch

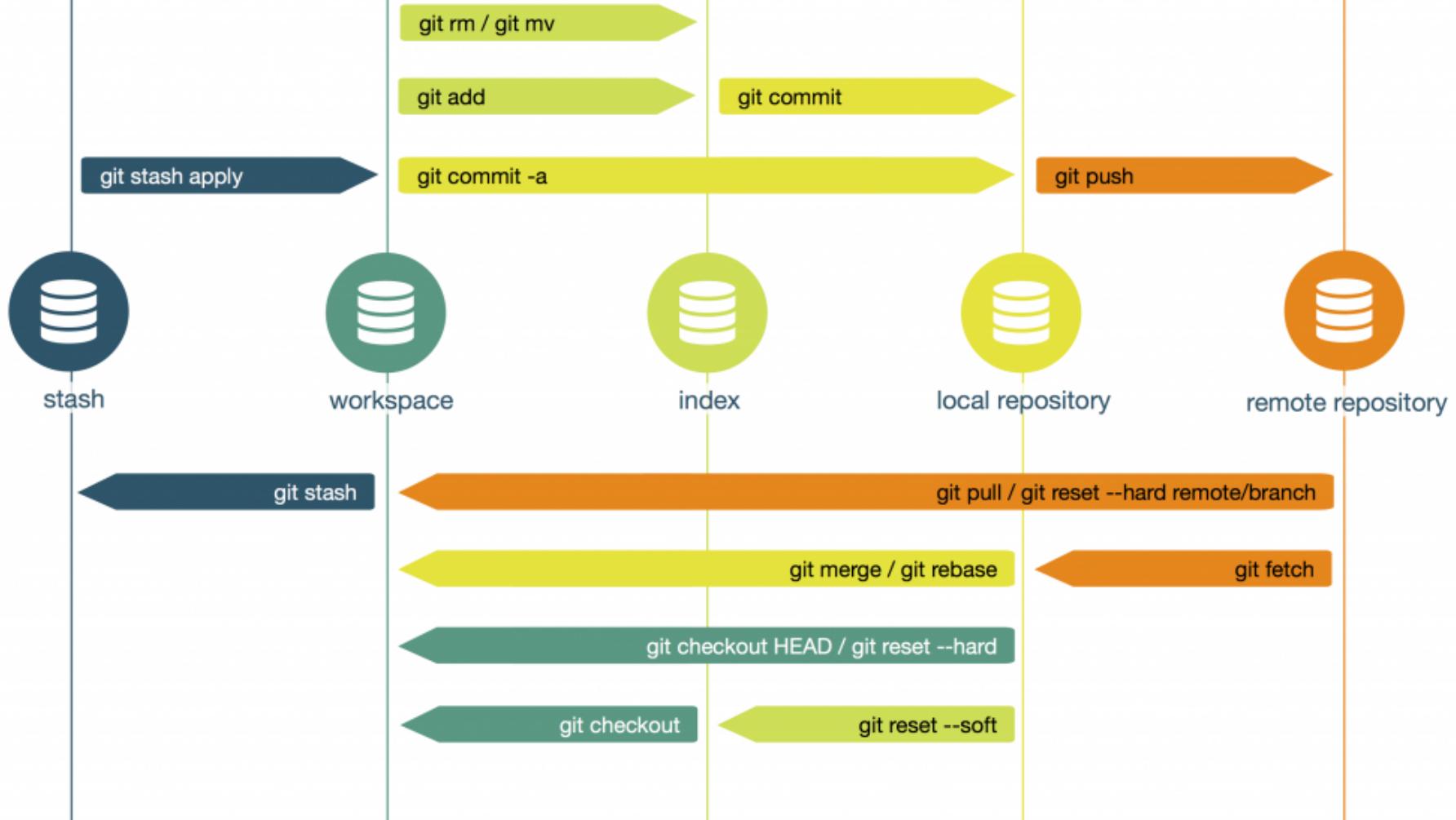
Understanding Git Workflow

- Obtain a repository
 - Either via git init, or git clone, or if you already have the repo, pull changes!
- Make some edits
 - Use your favorite text editor or source code IDE
 - Most IDEs have Git integration, including NetBeans
 - git tracks changes to binary files too: images, pdf, etc.
 - Less useful though, than text-based files
- Stage your changes
 - using git add
- Commit your work
 - git commit -m "Always write clear commit messages!"
- Push to remote
 - git push remotename localbranch:remotebranch



git data transport commands

patrickzahnd.ch



First Steps: Check installation

- Check if Git is installed:
 - \$ git --version
 - git version 1.7.10.2 (Apple Git-33)
- If not, brew install git or download the installer
- Set your Git username and email to your global config:
 - \$ git config --global user.name "your name"
 - \$ git config --global user.email "your email"
- How to enter your username and password only once:
 - <https://help.github.com/articles/generating-ssh-keys>

First Steps: Obtaining a repository

- Creating a repository (if one does not exist remotely):
 - \$ git init
 - @see <http://git-scm.com/docs/git-init>
- Or, cloning a remote repository:
 - \$ git clone https://reposite.tld/account/repo.git localdir
 - \$ git clone https://github.com/phpguru/phredis.git
 - *localdir* is optional. The repo name is used in this case.
 - @see <http://git-scm.com/docs/git-clone>
 - \$ cd phredis
 - \$ ls -la
- Inspect the repo configuration:
 - \$ vim .git/config

First Steps: Inspecting your repo

- Change Directory into the repository & issue a git status
 - \$ cd path/to/repo
 - \$ git status
 - @see <http://git-scm.com/docs/git-status>
 - "Displays paths that have differences between the index file and the current HEAD commit, paths that have differences between the working tree and the index file, and paths in the working tree that are not tracked by Git (and are not ignored by gitignore(5))"
 - @see <http://git-scm.com/docs/gitignore>
- In other words, shows what you added, modified or deleted since your last commit.

First Steps: Inspecting your repo

- Change Directory into the repository & issue a git log
 - \$ cd path/to/repo
 - \$ git log
 - \$ git log --pretty=oneline
 - \$ git log -1
 - @see <http://git-scm.com/docs/git-log>
- "Shows the commit logs."
- In other words, why you committed something
 - Hopefully you wrote descriptive commit messages in the past!

First Steps: Making Edits & Committing

- You can use whatever program you normally use to edit files.
- Make a new file - newfile.txt
- Edit the file as desired
- Save it in your working directory
- \$ git status
 - will show the file as "Changes not staged for commit"
- \$ git add newfile.txt
- \$ git status
 - will show newfile.txt as "Changes to be committed"
 - git commit -m "Added newfile.txt as an example"
 - [Master 1e77a20] Added newfile.txt as an example
- 1 file changed, 0 insertions(+), 0 deletions(-)

First Steps: Adding (staging) changes

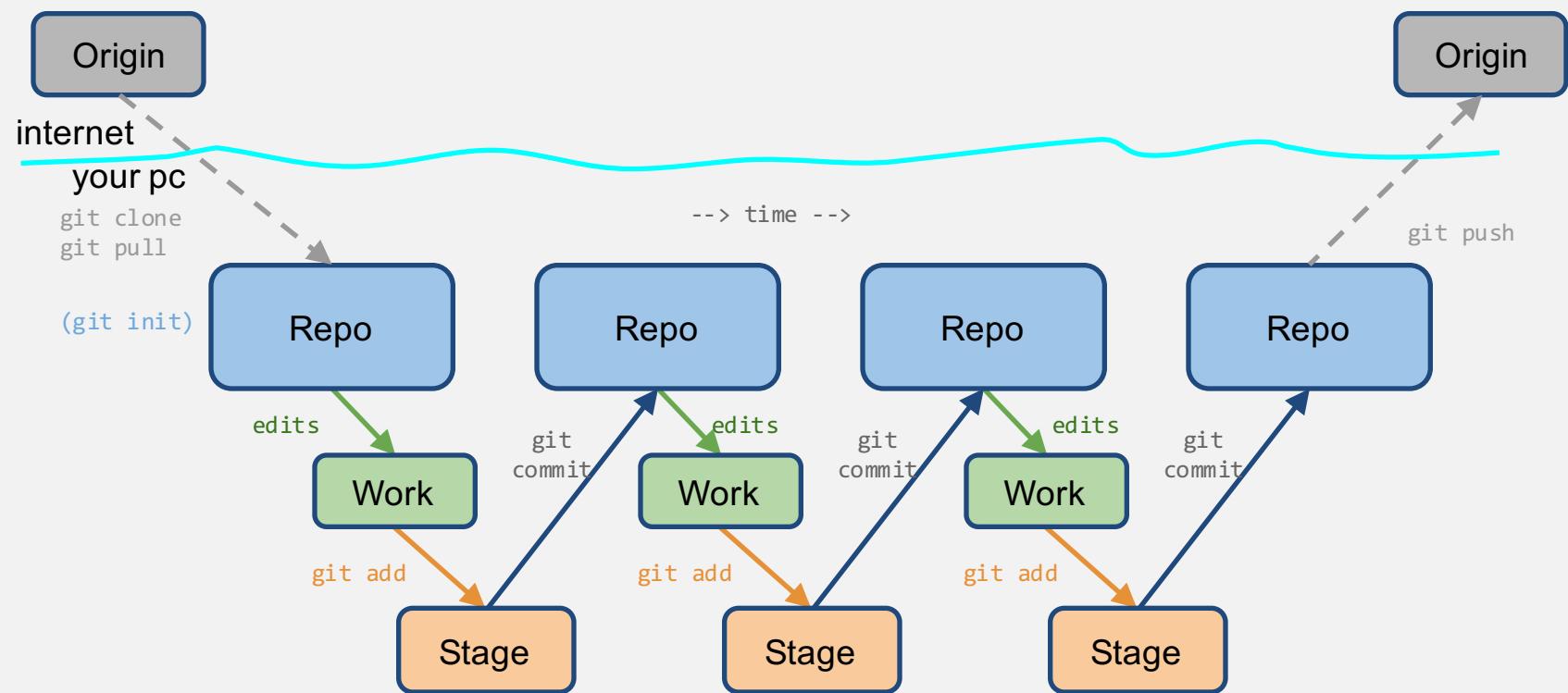
- Change Directory into the repository & issue a git status, then add new/modified files
 - \$ cd path/to/repo
 - \$ git add (file)
 - \$ git add –A
 - @see <http://git-scm.com/docs/git-add>
 - "This command updates the index using the current content found in the working tree, to prepare the content staged for the next commit."
 - In other words, stage your added, modified or deleted files for committing.

First Steps: Committing (staged) changes

- Change Directory into the repository & issue a git status, then add new/modified files, then commit them to your local repository
 - \$ cd path/to/repo
 - \$ git add (file)
 - \$ git commit -m "What did you change? Log messages matter."
 - @see <http://git-scm.com/docs/git-commit>
 - "Stores the current contents of the index in a new commit along with a log message from the user describing the changes."
 - In other words, store the added, modified or deleted files you staged in your repository.

Git workflow visualization

Standard Workflow



First Steps: Pushing

- At this point, none of those changes you've made have left your computer. (You still have local "infinite undo".) Let's push those changes to the remote repo.
 - `git push <repository> <refspec>`
 - `git push origin master`
 - `git push origin master:master`
- You now have an annotated, offsite backup of the work you performed!
 - REMEMBER:
 - If you never commit, you have no history.
 - If you never push, you have no offsite backup.
 - As a rule of thumb, commit every hour, push every day.
 - There is no reason not to commit after every "logical stopping-point" and push when a good section of work is complete, unit tests pass, etc.
 - After you commit, your colleagues can pull changes down from the origin repository to their local working repository.

First Steps: Pushing your changes offsite (local -> remote)

- Change Directory into the repository & issue a git push
 - \$ cd path/to/repo
 - \$ git push origin master
 - @see <http://git-scm.com/docs/git-push>
 - "Updates remote refs using local refs, while sending objects necessary to complete the given refs."
 - In other words, send your local changes back to the remote repo you cloned from.

First Steps: Pulling someone else's changes (remote -> local)

- Change Directory into the repository & issue a git pull
 - \$ cd path/to/repo
 - \$ git pull origin master
 - @see <http://git-scm.com/docs/git-pull>
 - "Incorporates changes from a remote repository into the current branch. In its default mode, git pull is shorthand for git fetch followed by git merge FETCH_HEAD."
 - In other words, retrieve changes made to the remote repo and merge them into your local repo, updating your working copy to match.

Working smart(er)

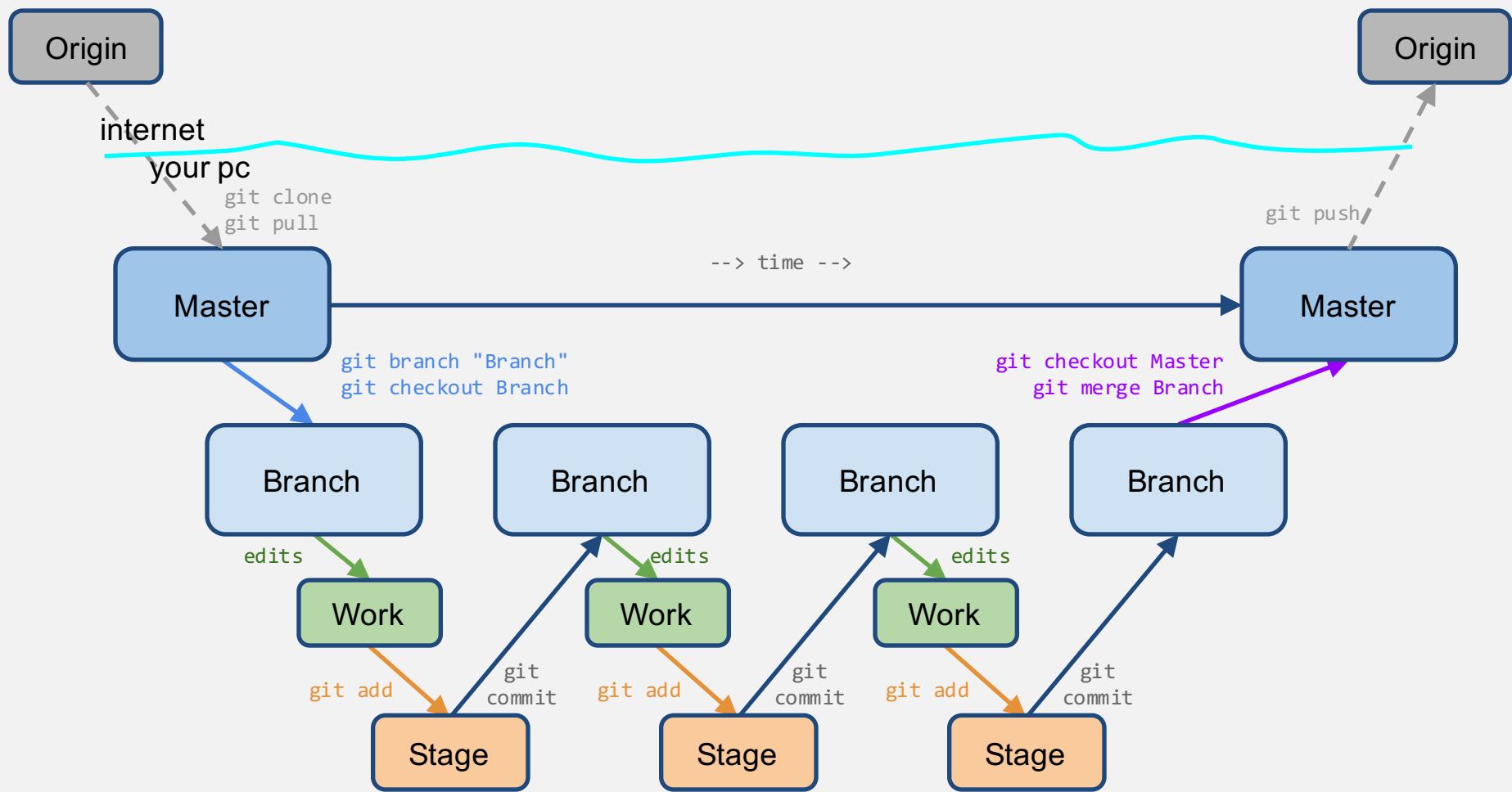
- Branches in Git can be created very quickly (because they happen locally). Creating a branch, and doing work in a branch, is smart because branches allow you to:
 - Try out an idea - experiment
 - Isolate work units - keep unfinished work separate
 - Work on long-running topics - come back to things later
 - Share your work in progress with others without altering the code in production or in development (remote feature branches)

Why do branches matter? An example to illustrate...

- Monday: You put all the client's code into a repo, clone it locally
- Tuesday: You create a branch to implement a new feature, making several commits
- Wednesday: You realize that it would be better to split out the feature into a couple different include files, so you create 2 new files and delete the old one. You commit your work.
- Thursday: The client contacts you frantically wanting a hot fix for an issue discovered on the version of the code on the live site.
- What do you do now?
 - You deleted their version of the file on Wednesday, right?
 - No! You made a branch.
 - Their original file is in the master branch still!
 - If you hadn't branched, you would be either downloading their original file again, or reverting your changes back to the version you had on Monday.

Working smart(er)

Branch Workflow



- Change Directory into the repository, create a branch and check it out
 - \$ cd path/to/repo
 - \$ git branch featurename
 - \$ git checkout featurename
 - @see <http://git-scm.com/docs/git-branch>
 - "If --list is given, or if there are no non-option arguments, existing branches are listed; the current branch will be highlighted with an asterisk. Option -r causes the remote-tracking branches to be listed, and option -a shows both local and remote branches. "
 - In other words, list or create local or remote branches depending on arguments provided.

- Merge changes made in a feature branch into another branch, typically development, to then be tested, and ultimately merged to the master branch and tagged.
 - \$ cd path/to/repo
 - \$ git branch
 - \$ git checkout master
 - \$ git merge featurename
 - @see <http://git-scm.com/docs/git-merge>
 - "Incorporates changes from the named commits (since the time their histories diverged from the current branch) into the current branch. This command is used by git pull to incorporate changes from another repository and can be used by hand to merge changes from one branch into another."
 - In other words, apply changes from another branch to the current one.

Branch Best Practices

- "master" is generally accepted to mean the main branch, that tracks what is deployed on the client's site. This is the default; when creating a new repository it will have one branch named master.
- "develop" or "development" is typically the name of a single branch that has newer code than what is in master. Un-named bug fixes and improvements.
- "featureXYZ" or "issue123" - Branches named for specific features or specific issue tickets in a 3rd party bug tracker.
- You can have branches in your local repository that are not tracked, meaning they do not exist in the remote repository. It's up to you.
- Even if you never make a feature branch, it's recommended to have at least one "development" branch separate from "master"

Final Steps: Deploying Code

- When we're satisfied with our work, everything is tested and it's ready to provide to the client, what should we do? Tag it!
 - Git has the ability to tag specific points in history as being important
 - Use this functionality to mark release points (v1.0, and so on).
 - A Tag should be used anytime you provide code to the client.
 - It should be annotated both in Git and in code comments somewhere, what you changed and why, who asked for the change, their email address and anything else that might be important.
 - Remember that the next person who works on the tag code may not be you! Pay it forward!
 - The client should be provided with access to download the Tagged version of the code
 - Github makes this a zip/tgz automatically

Final Steps: Deploying Code

- Change Directory into the repository, list the current tags, check the branch you're on, and tag the current branch/revision
 - \$ cd path/to/repo
 - \$ git tag -l
 - \$ git branch
 - \$ git tag -a v1.1 -m "Tagging version 1.1 with featurename"
 - @see <http://git-scm.com/docs/git-tag>
 - @see <http://git-scm.com/book/en/Git-Basics-Tagging>
 - "Add a tag reference in refs/tags/, unless -d/-l/-v is given to delete, list or verify tags. Unless -f is given, the named tag must not yet exist. If one of -a, -s, or -u <key-id> is passed, the command creates a tag object, and requires a tag message."
 - In other words, create a named/numbered reference to a specific revision of the code, or list prior tags created

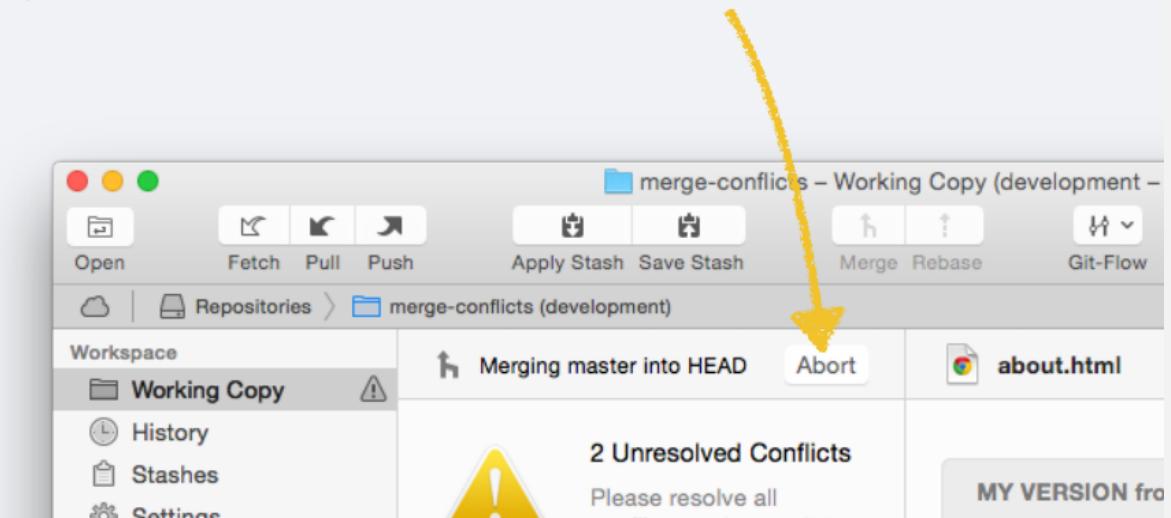
Final Steps: Deploying Code

- A tag is a special type of commit... as such, just like any other commit, the tag only exists locally. The final step in tagging is to push the tag to the remote repository, just like we do with regular commits.
 - \$ cd path/to/repo
 - \$ git push origin v1.1
 - \$ git push origin --tags
 - @see <http://git-scm.com/book/en/Git-Basics-Tagging>
 - "By default, the git push command doesn't transfer tags to remote servers. You will have to explicitly push tags to a shared server after you have created them."
 - In other words, don't forget to push after tagging, otherwise the tag only exists on your local machine.

Dealing with Merge Conflicts

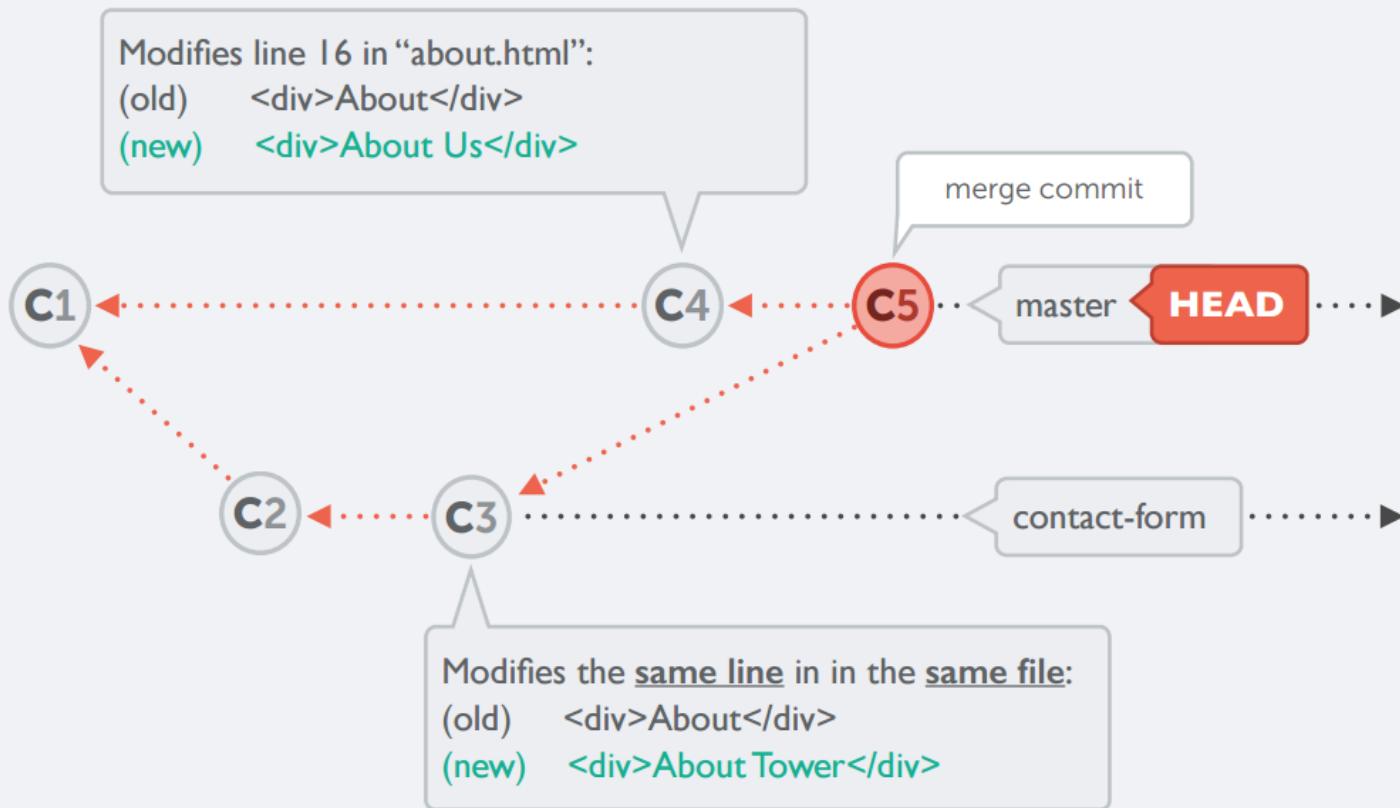
You Can't Break Anything!

1. Git handles most cases automatically for you.
2. Conflicts only happen locally, on your machine.
You only ever handicap yourself (not your team).
3. You can always undo and start fresh. Don't panic!



Dealing with Merge Conflicts

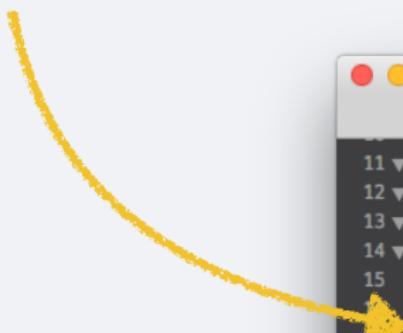
How a Conflict Happens



Dealing with Merge Conflicts

What a Conflict Really Is

1. Two contradictory versions ("A" and "B") of a file.
2. ...just funny letters in a text file.



```
about.html — merge-conflicts (git: development)
about.html +
```

```
11 ▼
12 ▼
13 ▼
14 ▼
15 <body>
   <div id="contentWrapper">
     <div id="navigation">
       <ul>
         <li><a href="index.html">Home</a></li>
<===== HEAD
      <li><a href="about.html">About Tower</a></li>
=====
      <li><a href="about.html">About Us</a></li>
>>>>> master
      <li><a href="imprint.html">Imprint</a></li>
    </ul>
  </div>
</body>
```

Line: 1 | HTML | Soft Tabs: 2 | Symbols

Dealing with Merge Conflicts

Understanding What Happened

```
$ git status
# On branch contact-form
# You have unmerged paths.
#   (fix conflicts and run "git commit")
#
# Unmerged paths:
#   (use "git add <file>..." to mark resolution)
#
#       both modified: contact.html
#       deleted by them: error.html
```

Two developers modified the same file on the same line(s).



The 90% case...

Other scenarios happen less frequently (here: one developer modified a file, while the other deleted it).

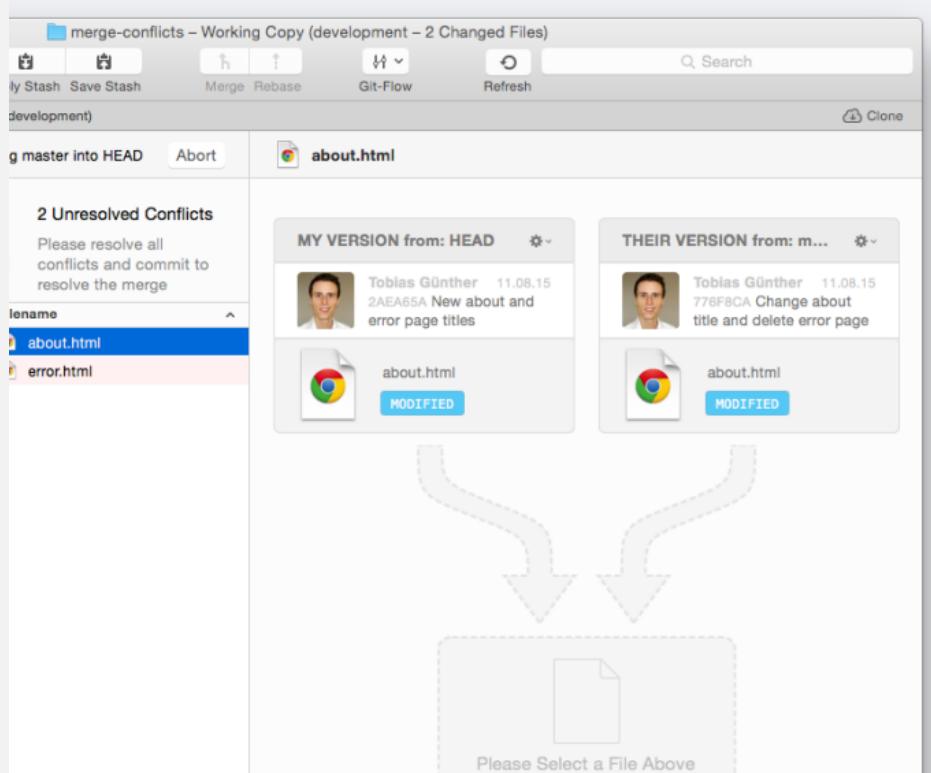


Dealing with Merge Conflicts

Solving a Conflict



= deciding, how the file should finally look



(a) Use your editor / IDE to modify the file by hand. Save it in the state you want it to be.

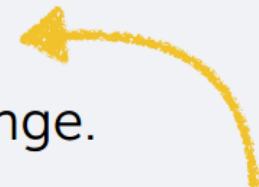
(b) Use a dedicated Merge Tool.

(c) Use a GUI client like Tower to make things easier.

Dealing with Merge Conflicts

Finishing a Conflict Resolution

1. Save the resolved file.
2. Stage (git add) this change.
3. Commit like any other change.



= marks as resolved
(might already be
done by external tool)

The most common commands you will use during a normal day/week working with Git to manage versions of code:

Download/Create a local repo

\$ git clone

\$ git init

Checking what's changed

\$ git status

\$ git log

Storing edits

\$ git add

\$ git commit

Updating your local repo

\$ git pull (git fetch)

\$ git branch

\$ git merge

Storing changes offsite/offbox

\$ git commit

\$ git push

Storing changes offsite/offbox

\$ git tag

\$ git push

Pair Programming



What do we see?

- Two person working on the same task
- One is executing the task
- Another is navigating (watches for external factors, evaluates the situation, corrects him and validates success after execution)
- Two different expertise
- Working as team

What is pair programming

- Pair programming consists of two programmers sharing a single workstation (one screen, keyboard and mouse among the pair).
- The programmer at the keyboard is usually called the - **driver**
- the other, also actively involved in the programming task but focusing more on overall direction is the - **navigator**
- it is expected that the programmers swap roles every few minutes or so.

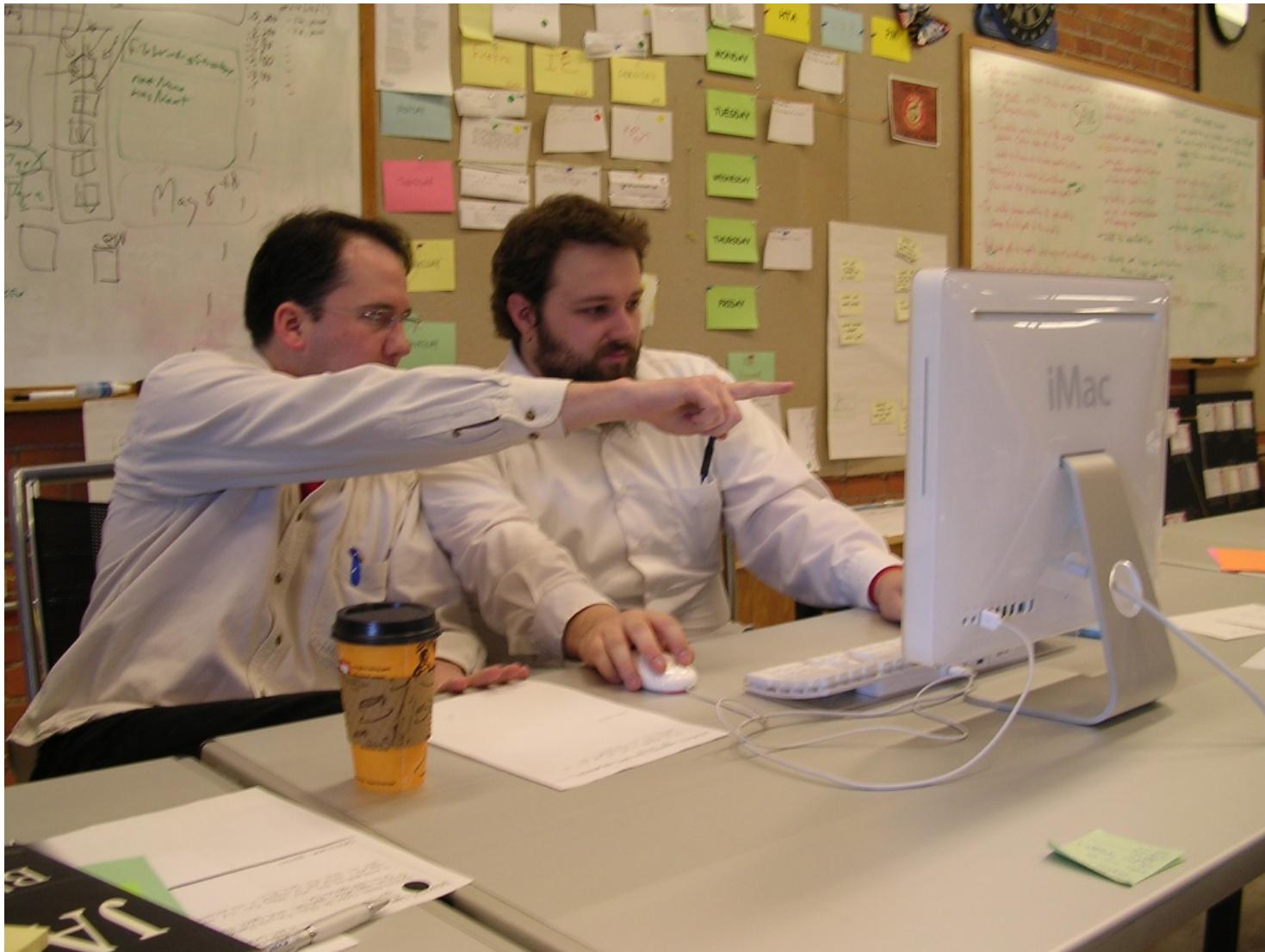
Key principles

- Communication
- Learning
- Knowledge sharing

Guidelines

- Share everything
- Play fair
- Don't hit people
- Put things back where you found them
- Clean up your own mess
- Don't take things that aren't yours
- Say you're sorry when you hurt somebody
- Wash your hands before you eat
- Flush
- Live a balanced life
- Hold hands and stick together
- Be aware of wonder.

Perfect setup for pair



POTENTIAL COSTS

- A quiet navigator will create a dysfunctional pair. Both programmers must be able to maintain a steady conversation.
- An initial decline in productivity is sometimes observed when engineers start to pair program.
- Noise from a pair can disturb others who work alone

- There can be a conflict of egos or personality depending on the programmers involved. This could result in uncomfortable feelings and reduced productivity, negating the potential benefits of pair programming.
- Everyone has to want to pair. Some programmers are very intrinsic and have a deep sense of personal ownership about the code they write, these feelings can be difficult to overcome at first.
- Software engineers are conditioned to work alone. It may feel like they are wasting their time with slower programmers or they may feel inadequate compared to their peers. Attempting pair programming with these types of feelings can lead to a loss in the benefits offered by the practice.

- Pairing an inexperienced programmer with someone more competent can result in feelings of intimidation, resulting in less participation between the two.
- Economics: organizations can be concerned that pair programming will double the development costs as two programmers are working on one task. The result is that it can be difficult to convince management to come onboard with the practice.
- Coordination can be a challenge as the team must decide who works with whom every day. Also, the team needs to decide which portions of the day should be paired sessions during which no meetings would take place.

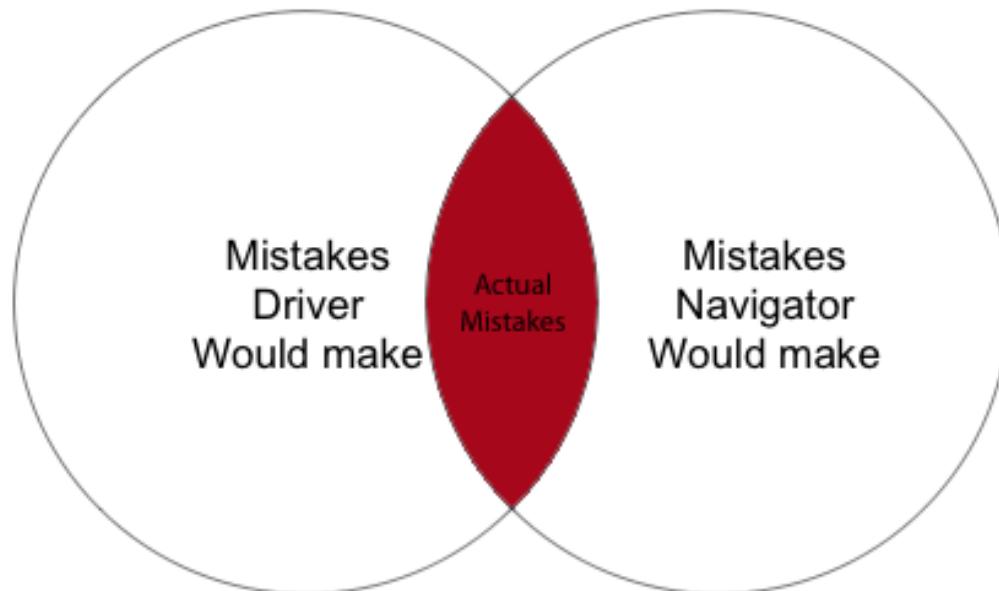
EXPECTED BENEFITS

- improved product quality: pairs produce shorter programs than solo peers, indicating superior designs.
- Programs written by paired developers pass more test cases than those written by solo programmers.

	% test cases pass by solo students	% of test cases passed by paired students
Program 1	73.4%	86.4%
Program 2	78.1%	88.6%
Program 3	70.4%	87.1%
Program 4	78.1%	94.4%

Table 1: Percentage of Test Cases Passed on Average

Reducing mistakes



- Positive effect of improved knowledge sharing. If only one person understands an area of code, the team can suffer if this person is unavailable or leaves the team. With more than one person familiar with each area of the code, that risk is reduced.
- Code is more understandable: code written by the driver must be understandable by the navigator, motivating the driver to be more clear.
- Increases in discipline in the use of other prescribed practices (test-driven development, use of coding standards, frequent integration). Developers are more likely to follow the proper practices when someone else is watching.

- Continuous reviews: the navigator is continually reviewing the driver's code in real-time, leading to more effective defect removal rates
- Despite an initial decline in productivity, pair programming usually has a positive effect on productivity with sustained user on one development team. Each partner is more engaged and focused on the task at hand with less interruptions like checking email
- People learn to discuss and work together, increasing team communication and effectiveness

If you want to go fast, go alone.
If you want to go far,
GO TOGETHER.

African Proverb



SYMPHONY OF LOVE
Photo by Nisha Gill

Understanding User stories

User stories

AS AN AGENT OR ADMINISTRATOR

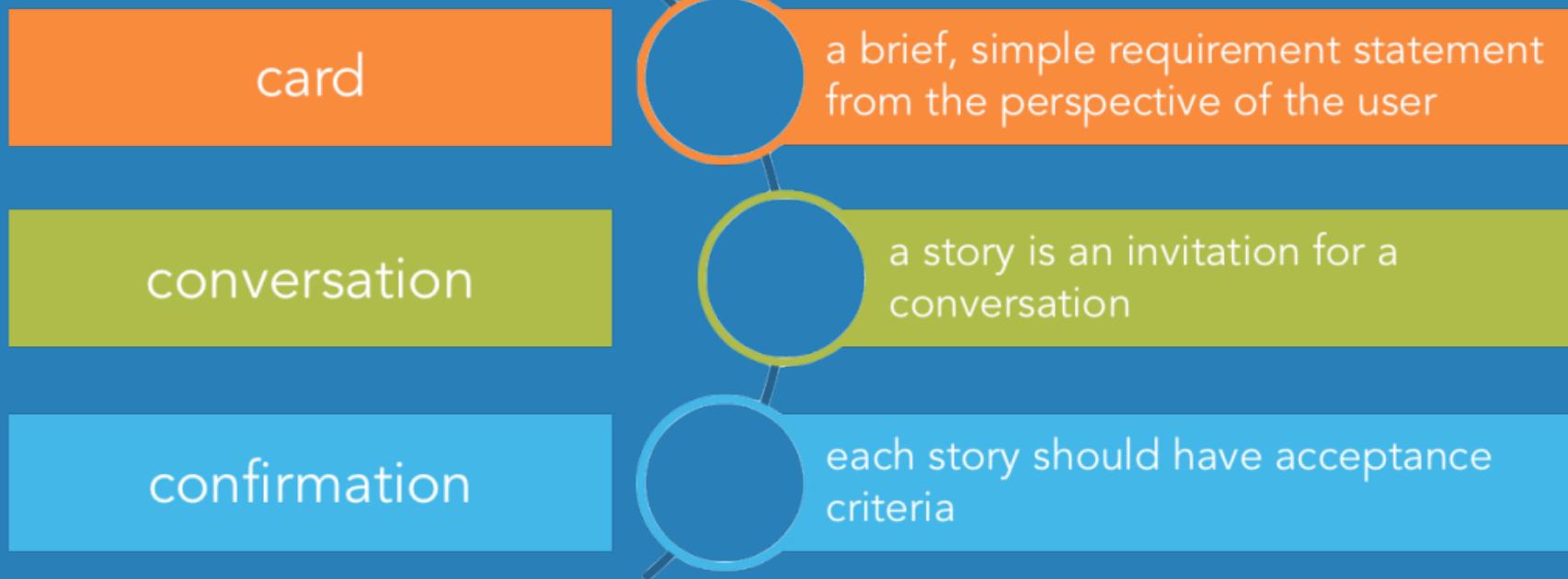
I WANT A PASSWORD
RESET/REMINDER VIA EMAIL

SO THAT I CAN RESET MY PASSWORD
IF I FORGET IT

What are user stories?

- Expresses functionality in terms of value to customers or users
- Uses the language of the customer
- Cross-cuts technical concerns
- Emphasizes shared understanding

What makes a user story?



Card

user stories need to be short and concise enough so that they can fit on a single card and in simplified language so that everyone understand

Conversation

- The collaborative conversation facilitated by the Product Owner which involves all stakeholders and the team.
- As much as possible, this is an in-person conversation.
- The conversation is where the real value of the story lies and the written Card should be adjusted to reflect the current shared understanding of this conversation.
- This conversation is mostly verbal but most often supported by documentation and ideally automated tests of various sorts

Confirmation

- The Product Owner must confirm that the story is complete before it can be considered “done”
- The team and the Product Owner check the “doneness” of each story in light of the Team’s current definition of “done”
- Specific acceptance criteria that is different from the current definition of “done” can be established for individual stories, but the current criteria must be well understood and agreed to by the Team. All associated acceptance tests should be in a passing state.

Different types of User stories

- Feature
- Epic
- Theme
- Goals

Epic

- Product Backlog item or User Story too big to complete within a Sprint
- **Simple Epic**
 - may be small enough to be completed in as few as two Sprints
 - need to be broken down so that the team can deliver value in a given Sprint – Done at Backlog Refinement
- **Large Epic**
 - might take the entire company several Quarters or Years
 - Requires the PO to work with Leadership and the Team to create Road Map, so most valuable features are created first

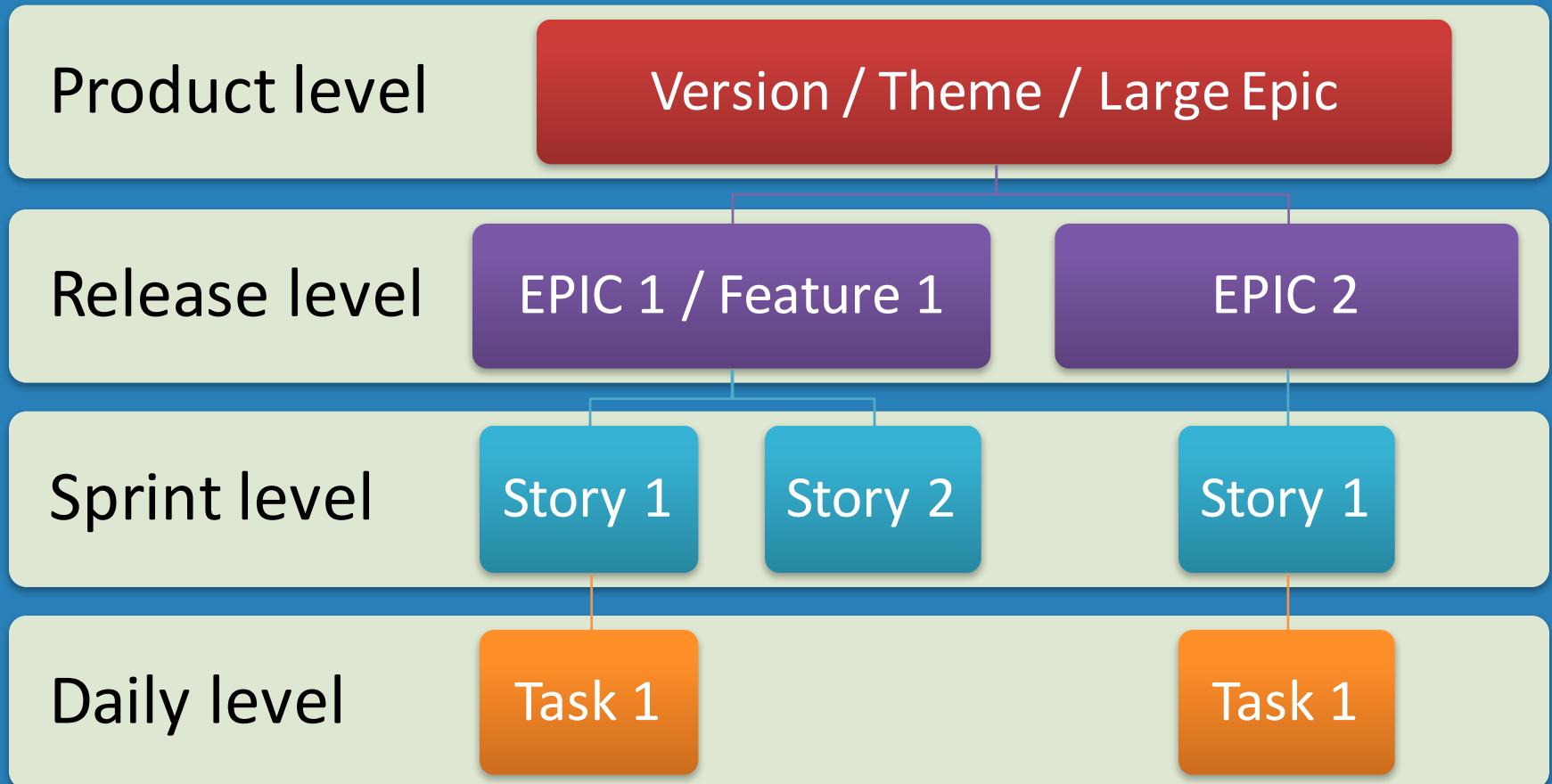
Theme

- Theme is a collection of related user stories.
- Used for grouping/categorizing the user stories

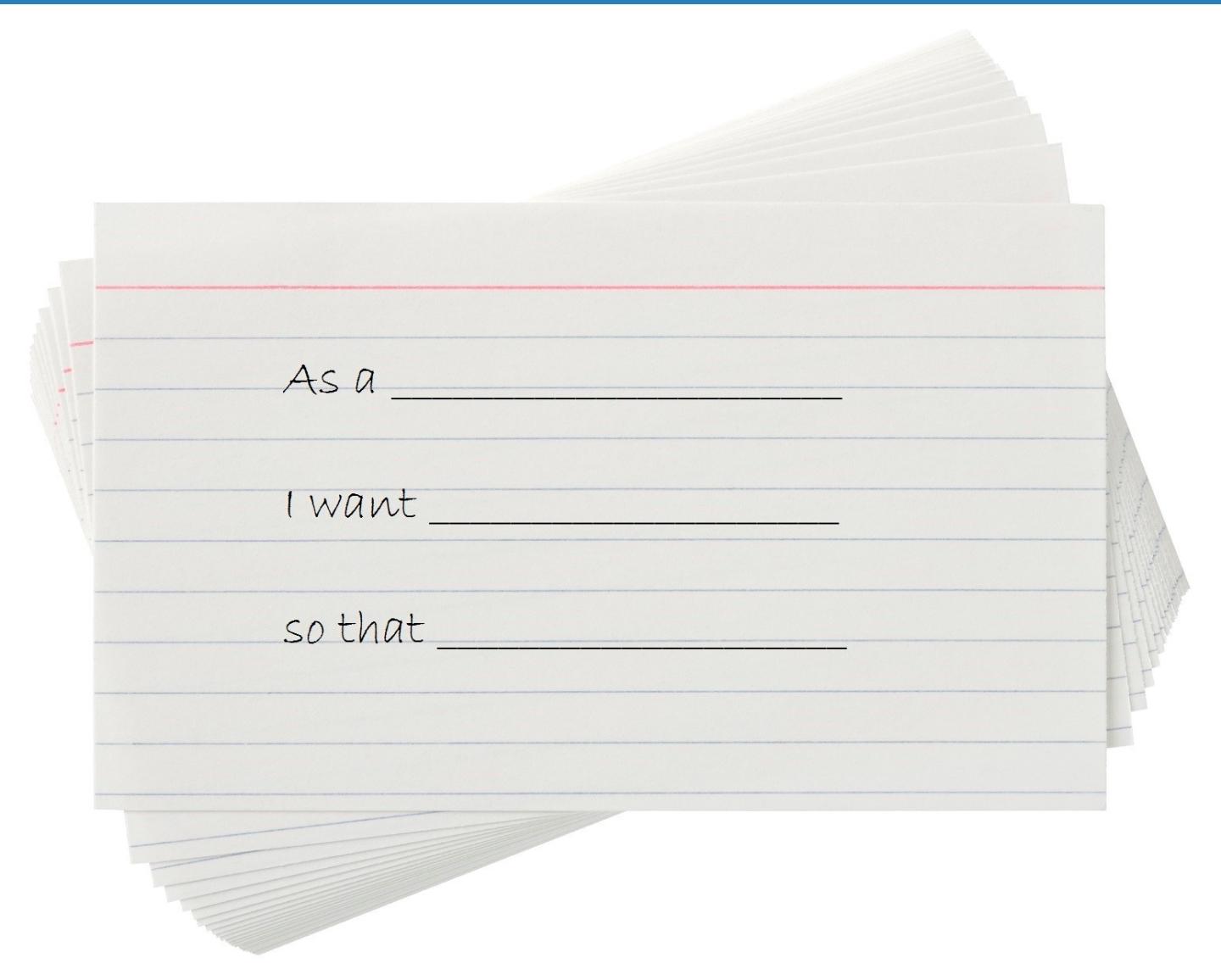
Feature

- An epic can consist of several features.
- For example, ‘*As a user, I want to search for a job, so I can find my next career move*‘ might include various features, such as
 - simple search
 - advanced search
 - Sort options
 - Filtering
 - Summary view of the jobs
 - Options to go to detail view or save for future view

Levels



Good format for user story



Why this format?

As who I want
what so that
why

Good example of user stories

- As a seller I would like to list my item so that I can sell them online
- As an admin I would like to see all pending items so that I can approve them for selling
- As a buyer I would like to add items in my shopping cart so that I can purchase them
- As a buyer I would like to have online payment facility so that I can pay for my items online.

Acceptance Test form/template

The **Given-When-Then formula** is a template intended to guide the writing of acceptance tests for a User Story:

(Given) some context

(When) some action is carried out

(Then) a particular set of observable consequences should be obtained

Given my bank account is in credit,

When I attempt to withdraw an amount less than my card's limit,

Then the withdrawal should complete without errors or warnings

Preparing the user story card

Front of the card

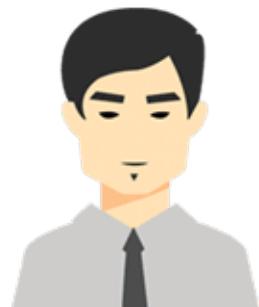
<Title>	#<ID>
As A:	
I Want To:	
So That:	
CONVERSATION	
•	
•	
•	
•	

Back of the card

What is not user story

- Functional requirements
- Use cases
- Interaction scenario
- Wireframes

Who writes user stories?



Product Owner



Scrum Master



Development Team Members

INVEST MODEL

Follow the INVEST
guidelines for good
user stories!



Independent

- Dependencies lead to problems estimating and prioritizing. Ideally you can work on a single story without pulling in lots of other stories.

Negotiable

- Stories are not contracts, they leave (or imply) some flexibility.

Valuable

- Valuable to users or customers, not developers.
- Rewrite developer stories to reflect value to users or customers

Estimable

- We need to be able to estimate our user stories so that we can use them to create a plan.

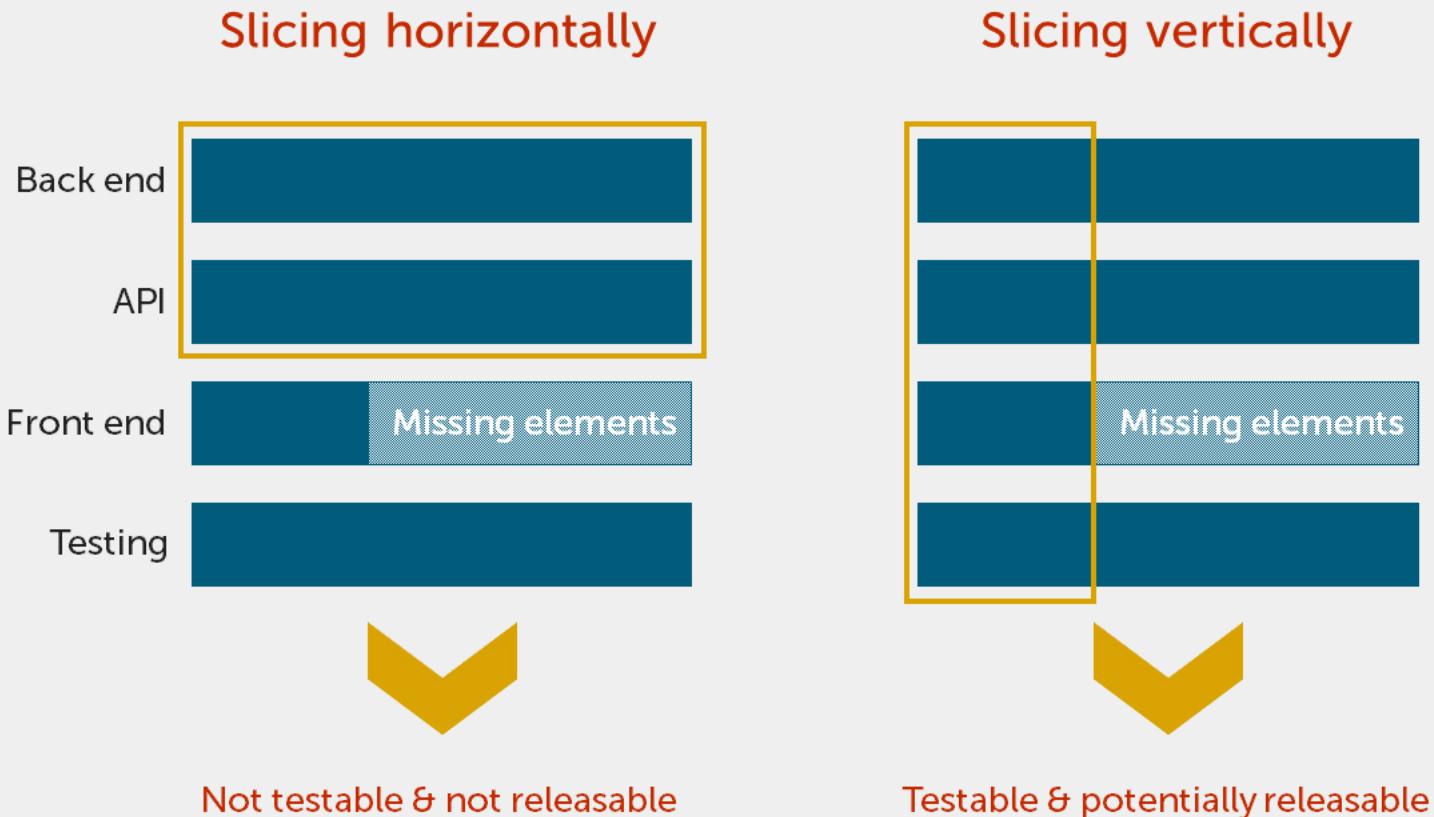
Small/sized appropriately

- A story is small or sized appropriately when it can be completed in one iteration.

Testable

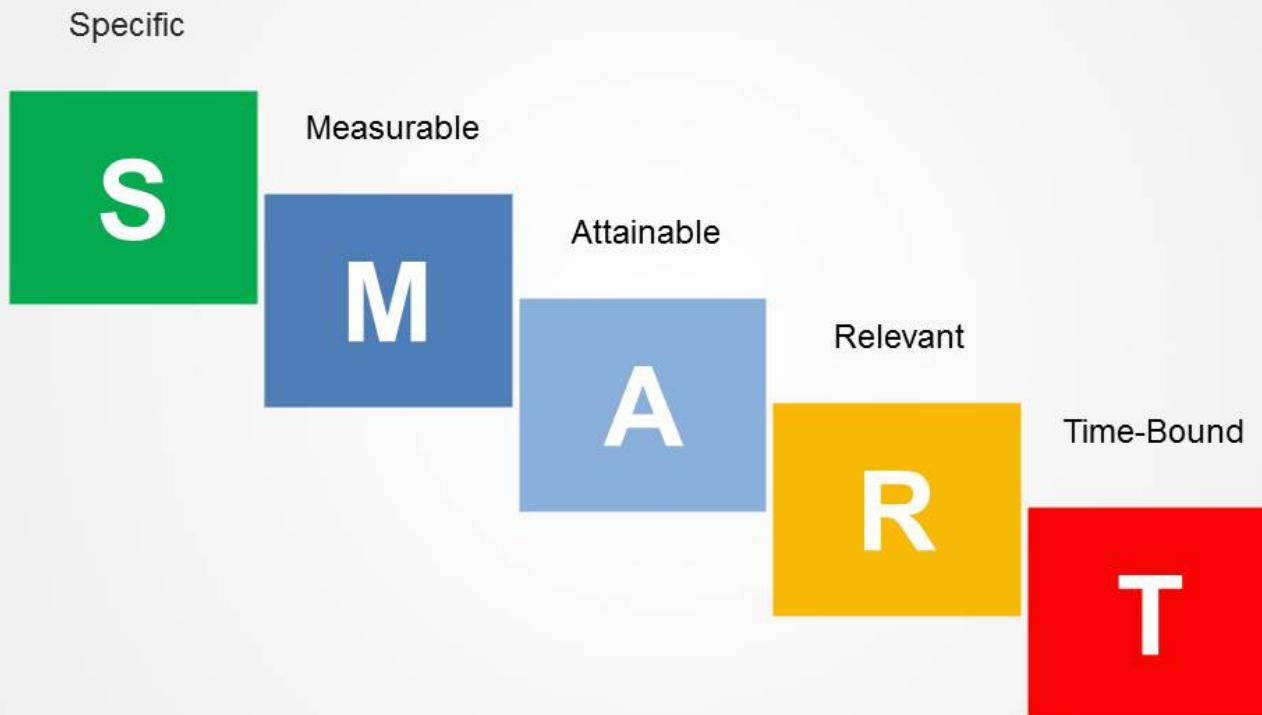
- You should have an easy and binary way of knowing when a story is finished. Done or not done; not partially finished or “done... except...”

Slicing vertically vs horizontally



SMART Tasks

SMART Objectives



Breaking down user stories

As a user, I want to cancel a reservation



As a premium member, I can cancel a reservation up to the last minute

As a non-premium member, I can cancel a reservation up to 24 hours in advance

As a user I am e-mailed a confirmation when I cancel a reservation

Adding Roles

- Adding proper roles can improve user stories.
- Identifying all roles is very important
- We can write/map user stories based on roles

Smelly user stories

- Stories are too small
- Interdependent stories
- Gold plated stories
- Too many details
- Splitting too many stories
- Including user interface details too soon
- Thinking too far ahead
- Trouble prioritizing
- Customer won't write and prioritize stories

Bad example of user stories

- As a customer I would like to have online banking option so that I can manage all my banking needs from internet
- Need a registration button in the website
- Complete payment system

Scenarios, User Case, User Story

Scenario

Josh is a 30 something mid-level manager for an ad agency, metrosexual and beer aficionado. He likes to try new and exotic beers in trendy locations. He also enjoys using a variety of social apps on his smart phone. He reads a review on Yelp of a new burger & beer joint downtown with over 100 beers on tap, and decides to go walk over after work and check it out.

Use Case

- Customer walks to the restaurant
- Customer enters the restaurant
- Customer finds a seat at the bar
- Customer scans the menu
- Customer selects a beer
- Customer orders selected beer
- Bartender takes order
- Bartender pours beer
- Bartender delivers beer
- User drinks beer
- User pays for beer

User story: A user wants to find a bar, to drink a beer

Definition of Done (DoD)



Sample DoD

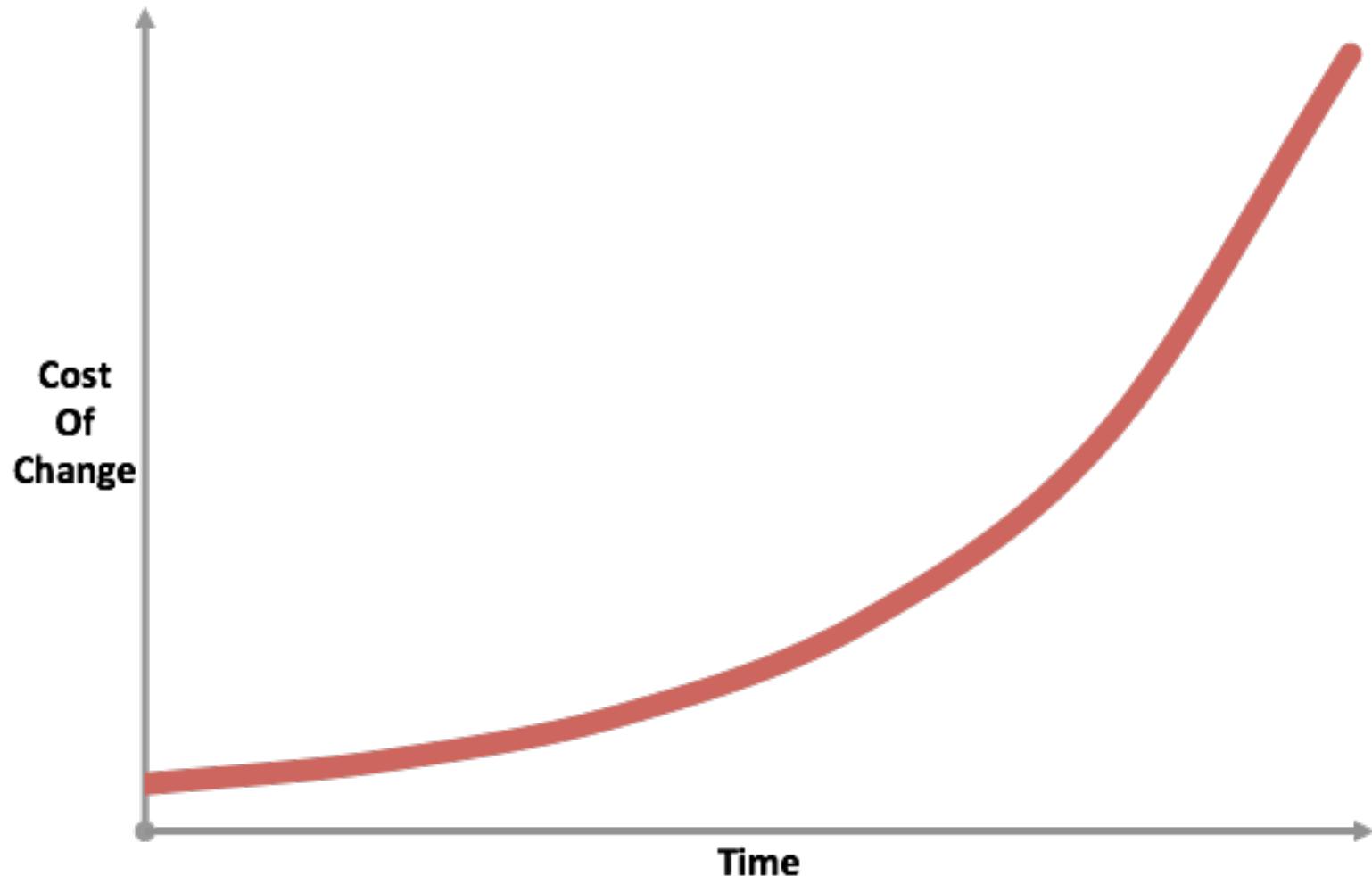
- Code changes
 - BDD tests written and pass
 - Unit tests written and pass
 - Code peer reviewed & approved
 - Code committed to repository
 - QA testing done
 - Product Owner signed off
- Documentation
 - Changes has been properly documented
 - Documentation has been peer reviewed & approved
 - Documentation approved by Product Owner
 - Version number updated

DoD vs Acceptance Criteria

Definition of Done	Acceptance Criteria (condition of satisfaction for User Story)
It serves the purpose of making unambiguous understanding of what all is needed before any product backlog item can be declared complete.	It serves the purpose of clarifying business requirements / conditions which must be met to satisfy the user for given requirement.
Definition of Done uniformly applied to all product backlog items.	It applies to specific product backlog item since it clarifies one item
Development team owns Definition of Done and it is understood and agreed by complete scrum team.	Product Owner owns Acceptance criteria, and development team understands them
Definition of Done does not change frequently, it is not expected to change during the sprint	Acceptance criteria are negotiable between product owner and development team.
Meeting Definition of Done ensures one meets the acceptance criteria	Just meeting Acceptance Criteria may not necessarily mean that Definition of Done is also met.

MANAGING SPECIFICATIONS

Cost of Change



- Are you tired of mindless manual regression tests?
- Does your testing get continually squeezed between development delays and unmovable ship dates?
- Are you frustrated that software delivered to you simply does not work or has major gaps that make testing futile?

Then **Specification By Example** can be a good approach.

What is specification by example

- Specification by example is a practice that expresses desired functionality as scenarios that are sufficiently clear that they can be automated.
- This allows a testing mindset to be applied early in the development cycle and helps drive development towards making these scenarios work.
- it builds a valuable regression suite that is focused on the most important system scenarios.

Build it right

OR

Build the right thing

Build it Right

Business Failure

Business Success

Specification By Example

Build the Right Thing

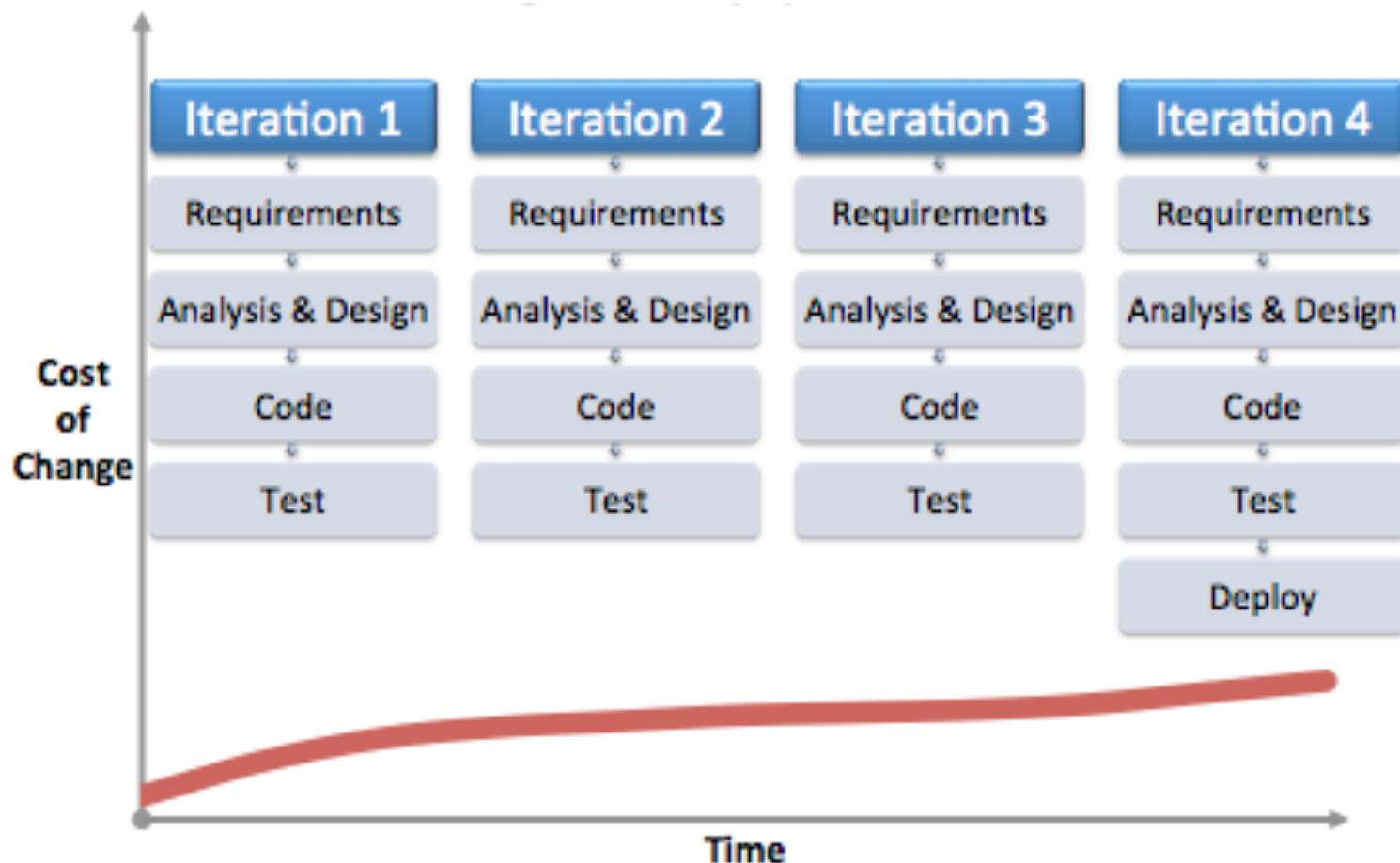
Useless Crap

Maintenance Nightmare

Specification By Example

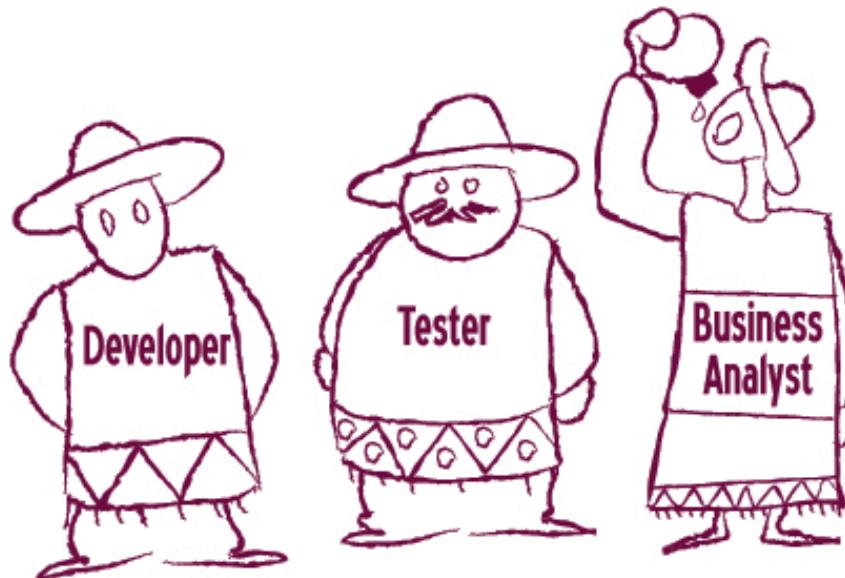


Cost of change in Agile



Specify Collaboratively through Workshops

- Hold regular product backlog workshops
- Full team workshops - when starting
- Three amigo workshops:
 - One developer
 - One tester
 - One analyst



Refining the Specification

- Be precise and make sure spec is testable
- Avoid “scripts” and “flows”
- Focus on business functionality not design
- Avoid UI details
- Avoid covering every possible combination

An example

Customers buying 5 KG or more organic food gets free delivery. Regular customer with customer card gets free delivery for any items weighting 2 KG.

Customer type	Item bought	Weight (KG)	Delivery
Regular Customer	Meat	1	No Free Delivery
Regular Customer	Flour	3	Free
Customer	Potato	4	No Free Delivery
Customer	Vegetables	6	Free
Customer	Potato , Meat	4, 2	Free

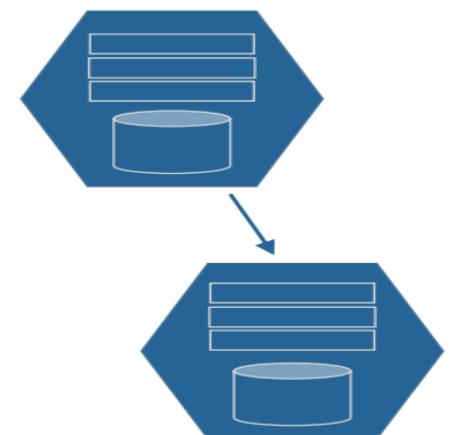
Agile Architecture

MICRO-SERVICES

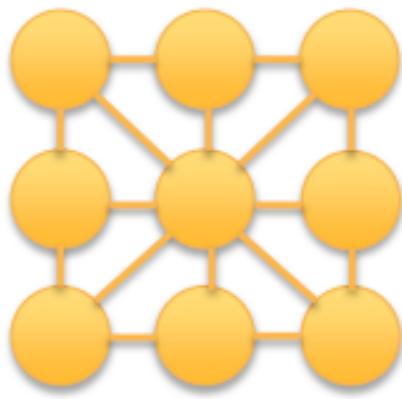
Micro-Services Architecture

- Defined as:
functional system decomposition into manageable and independently deployable components
- “micro” refers to the sizing
 - a micro service must be manageable by a single development team (5-9 developers)
- Functional system decomposition means vertical slicing
 - in contrast to horizontal slicing through layers

- Each micro-service is functionally complete with
 - Resource representation
 - Data management
- Each micro-service handles one resource (or verb), e.g.
 - Clients
 - Shop Items
 - Carts
 - Checkout



Micro-services architecture is an **evolutionary design**
ideal for evolutionary systems where you can't
fully anticipate the types of devices that may
one day be accessing your application



vs



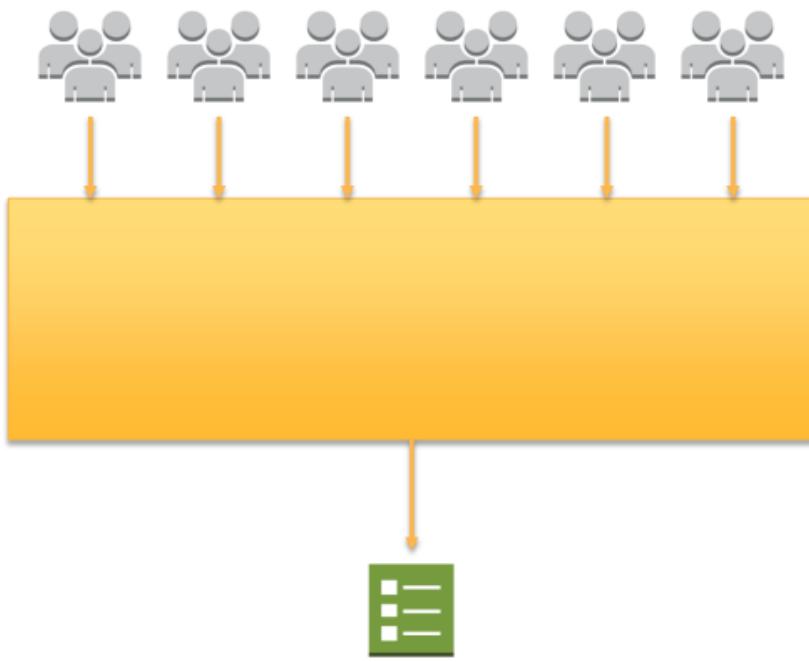
Microservices

Monolith

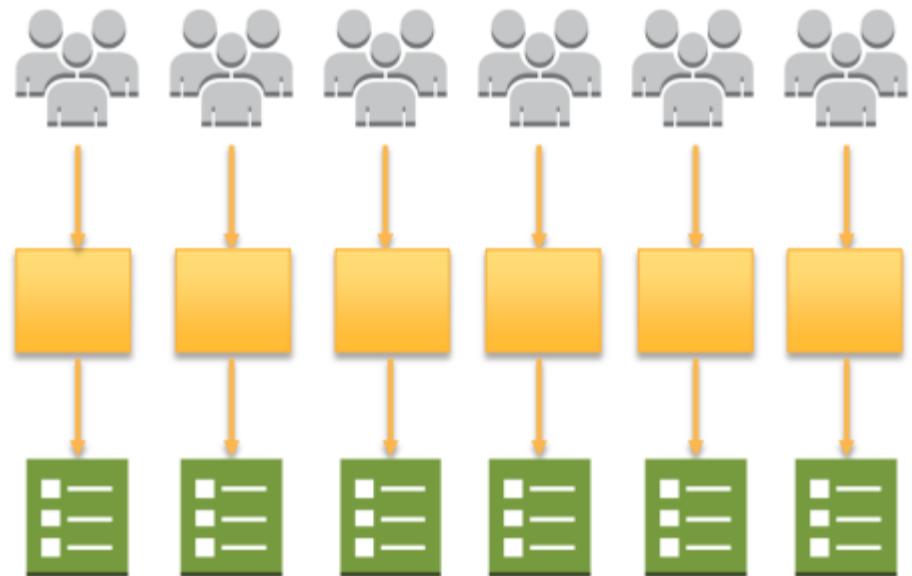
Problems of Monolithic Architectures

- Code complexity and maintainability
- Deployment becomes the bottleneck
- Fear to change
- Lack of ownership
- Failure dependencies
- One size doesn't fit all (ex: relational DB)
- Hard to scale out

Monolithic



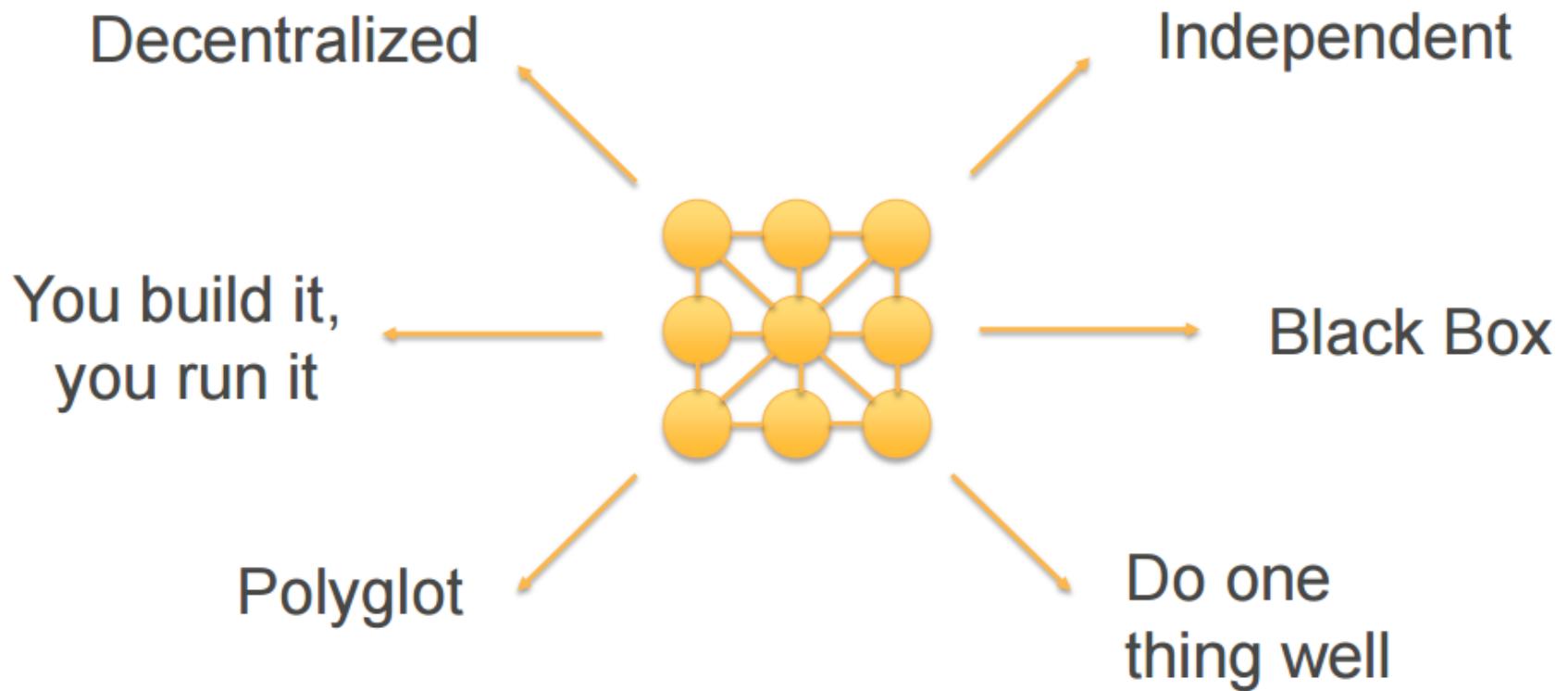
Micro Services



Components & Connectors

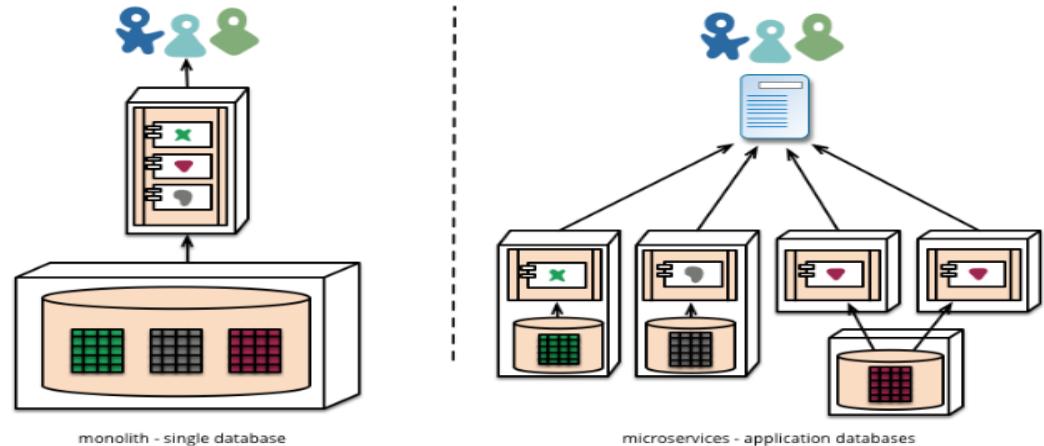
- **Components** – individual programs
- **Connectors** are discussed below:
 - Communication between micro services is often standardized using
 - HTTP(S) – broadly available transport protocol
 - REST – uniform interfaces on data as resources with known manipulation means
 - JSON – simple data representation format
 - REST and JSON are convenient because they simplify interface evolution

Quality Characteristics



Decentralization

- Micro-services decentralize data storage decisions
 - Monolithic applications prefer a single logical database for persistent data
- Micro-services prefer letting each service manage its own database
 - either different instances or entirely different database systems

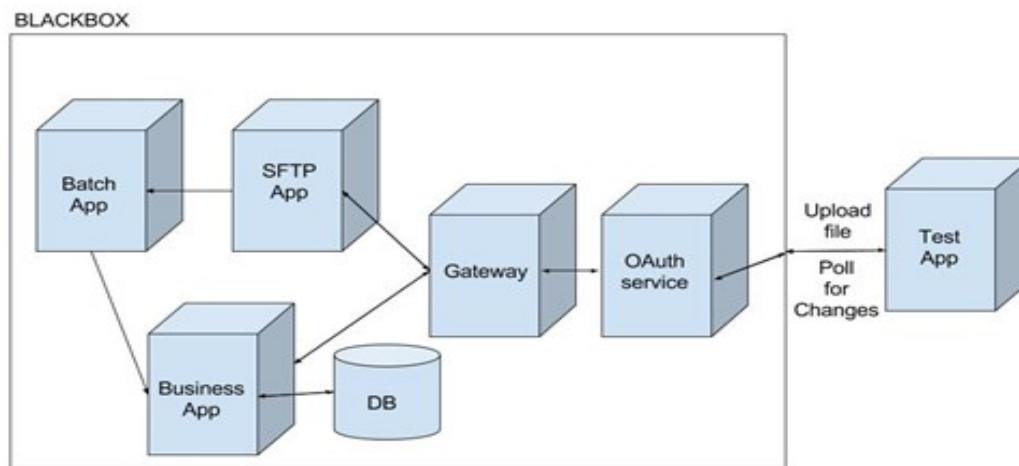


Independence

- Independent deployability is key
- Each service has its own software repository
- Each micro service can be scaled independently
 - Data sharing can be applied to micro-services as needed
- Independent evolution of Features
 - Micro-services can be extended without affecting other services

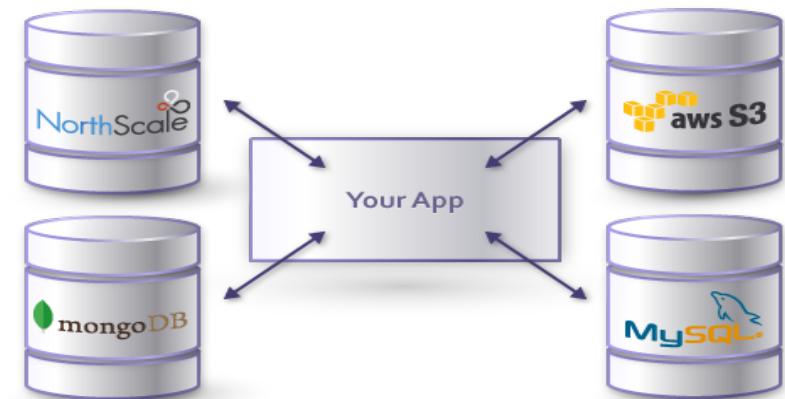
Black Box

- **Black box testing** is a type of component interface testing
- verifies that data being passed among components in the system is done right and it verifies application flow



Polyglot

- Polyglot persistence is an approach
- Used in monolithic but frequently in micro services
- Using multiple data storage technologies for an individual application
- chosen based upon the best way data can be stored and retrieved for the application

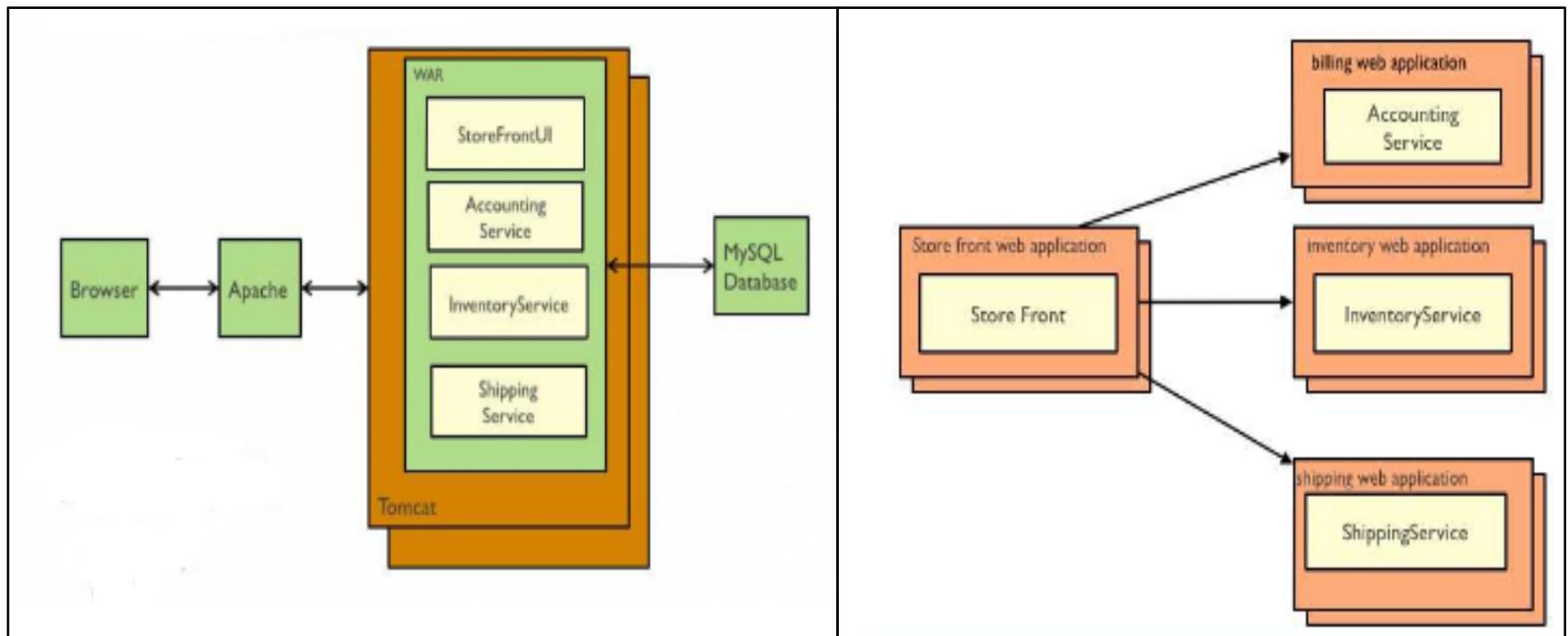


Typical Usage and Applications

- Money Transfer/Transactional processing in Financial Systems
- Most large scale web sites including Netflix, Amazon and eBay have evolved from a monolithic architecture to a **Micro-Service** architecture.

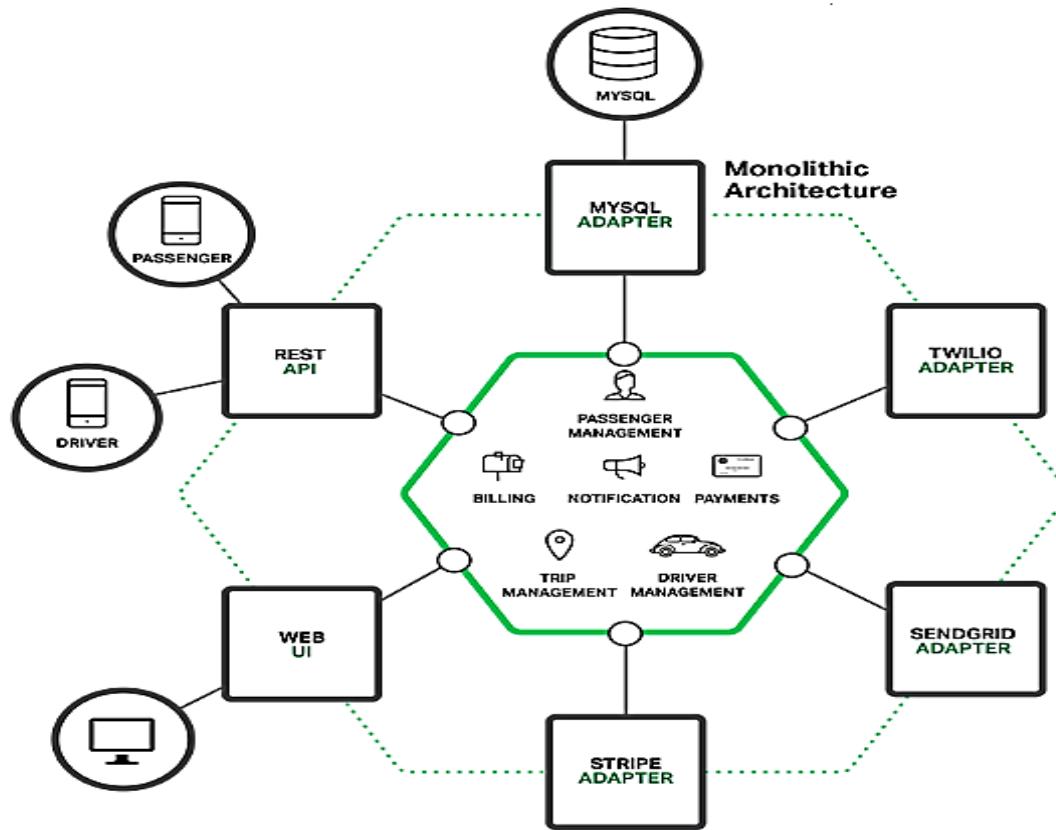
Examples

- Traditional Web Application architecture



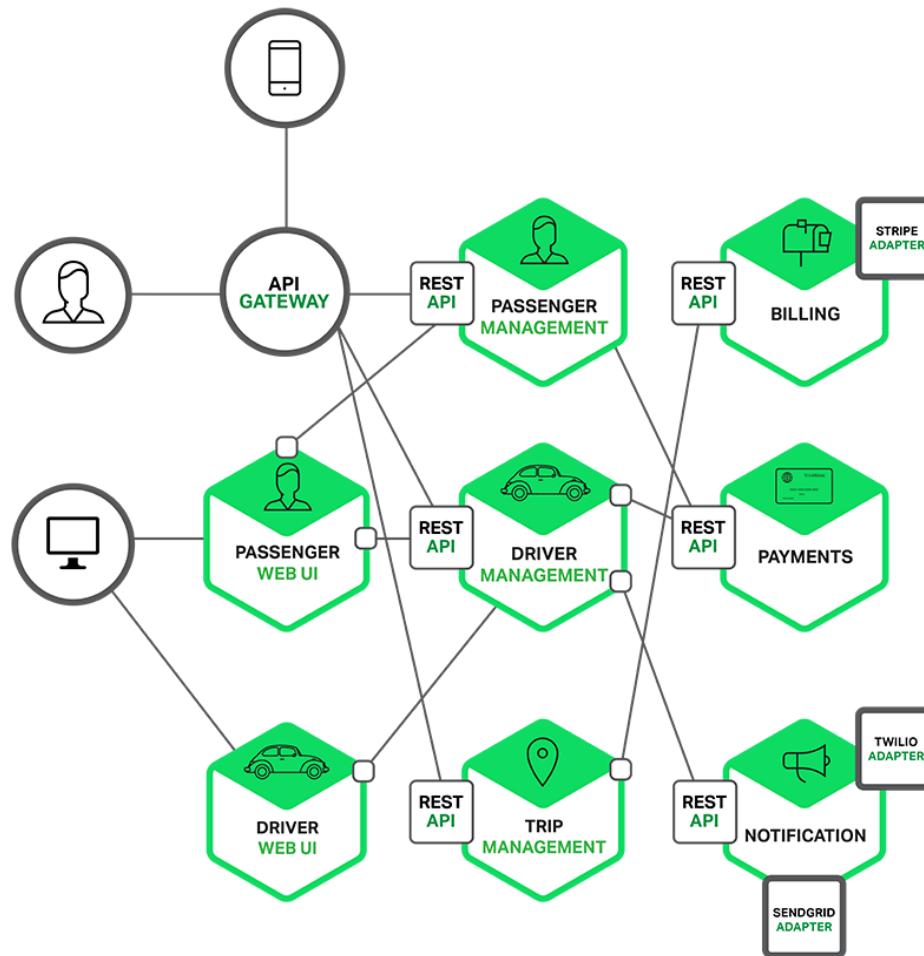
Cont...

- Taxi Hailing Company (monolithic architecture)

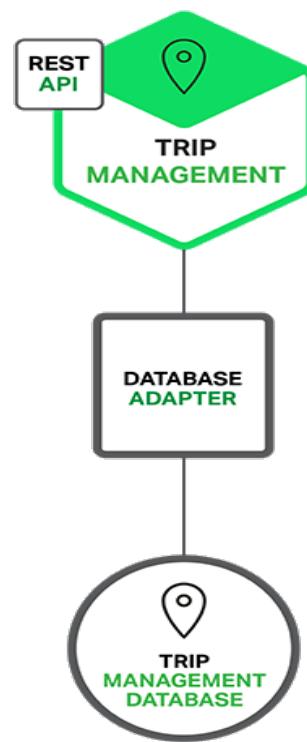
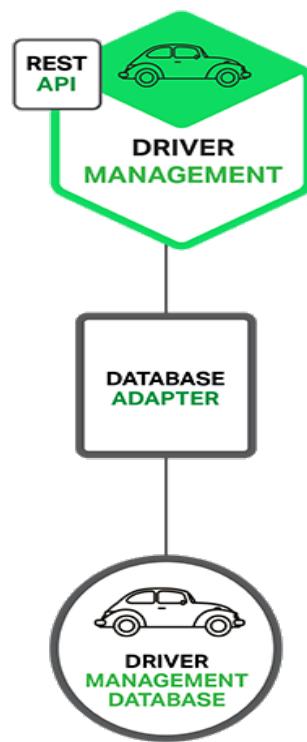


Cont...

Taxi Hailing Company (Micro-Services Architecture)



Cont...



Constraints and Limitations

Constraints

(principles that drive desired properties)

- Elastic
- Resilient
- Composable
- Minimal
- Complete

Limitations

- Complexity
 - Creational (in case of Distributed Systems)
 - Operational
 - Deployment
- Increased Memory Consumption

Micro-Service Constraint #1 - Elastic

- A Micro-Service must be able to scale, up or down, independently of other services in the same application.
- This constraint implies that based on load, or other factors, applications' performance and availability couldn't be affected.
- Multiple instances of each micro service can run ineffectively

Micro-Service Constraint #2 - Resilient

- A Micro-Service must fail without impacting other services in the same application.
- A failure of a **single service instance** should have minimal impact on the application.
 - Failure of all instances of a micro service, should impact a single application function & users should be able to continue using the rest of the application without impact.
- **Micro-Services** are Loosely Coupled

Micro-Service Constraint #3 - Composable

- A Micro-Service must offer an interface that is uniform and is designed to support service composition.
- Micro-Service APIs should be designed with a common way of identifying, representing, and manipulating resources.

Micro-Service Constraint #4 - Minimal

- A **Micro-Service** must only contain highly cohesive entities
- In software, cohesion is a measure of whether things belong together.
- High cohesion – all objects and functions in it are focused on the same tasks
- A **Micro-Service** should perform a single business function, which implies that all of its components are highly cohesive.
- Higher cohesion – more maintainable software.

Micro-Service Constraint #5 - Complete

- **A Micro-Service must be functionally complete**
- A Micro-Service must offer a complete function, with minimal dependencies (loose coupling) to other services in the application.
- **Minimal but Complete.**

Limitation #1 - Complexity

- Too many **Micro-Services**
- Developers must deal with the additional complexity of creating a distributed system.
 - Testing is more difficult
 - Developers must implement the inter-service communication mechanism.
- Developers must implement mechanisms for handling operations/functionality of many different types of services.
- Obviously, the deployment complexity will be increased – need to deploy and manage many different service types.

Limitation #2 – Increased Memory Consumption

- The micro-services approach leads to the increased memory consumption. It simply, due to own address space for the each service.

Design patterns

Refactor to patterns

What is design pattern?

In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

Design Pattern Definitions

- Design patterns are recurring solutions to design problems you see over and over.
- Design patterns constitute a set of rules describing how to accomplish certain tasks in the realm of software development.
- Design patterns focus more on reuse of recurring architectural design themes, while frameworks focus on detailed design... and implementation
- A pattern addresses a recurring design problem that arises in specific design situations and presents a solution to it.
- Patterns identify and specify abstractions that are above the level of single classes and instances, or of components.
- Design patterns are not just about the design of objects; they are also about interaction between objects. One possible view of some of these patterns is to consider them as communication patterns

The design patterns divides into three types:

- **Creational patterns:** create objects for you, rather than your having to instantiate objects directly. Your program gains more flexibility in deciding which objects need to be created for a given case.
- **Structural patterns:** help you compose groups of objects into larger structures, such as complex user interfaces and accounting data.
- **Behavioral patterns:** help you to define the communication between objects in your system and how the flow is controlled in a complex program.

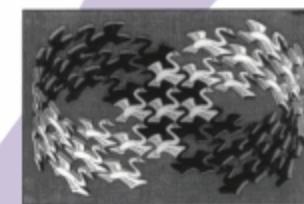
The Gang of Four



Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Cover art © 1994 McGraw-Hill Book Company. All rights reserved.

Foreword by Grady Booch



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

THE 23 GANG OF FOUR DESIGN PATTERNS

C	Abstract Factory	S	Facade	S	Proxy
S	Adapter	C	Factory Method	B	Observer
S	Bridge	S	Flyweight	C	Singleton
C	Builder	B	Interpreter	B	State
B	Chain of Responsibility	B	Iterator	B	Strategy
B	Command	B	Mediator	B	Template Method
S	Composite	B	Memento	B	Visitor
S	Decorator	C	Prototype		

By purpose and scope

By Purpose				
		Creational	Structural	Behavioral
By Scope	Class	<ul style="list-style-type: none">• Factory Method	<ul style="list-style-type: none">• Adapter (class)	<ul style="list-style-type: none">• Interpreter• Template Method
	Object	<ul style="list-style-type: none">• Abstract Factory• Builder• Prototype• Singleton	<ul style="list-style-type: none">• Adapter (object)• Bridge• Composite• Decorator• Façade• Flyweight• Proxy	<ul style="list-style-type: none">• Chain of Responsibility• Command• Iterator• Mediator• Memento• Observer• State• Strategy• Visitor

Abstract Factory Design Pattern

Intent:

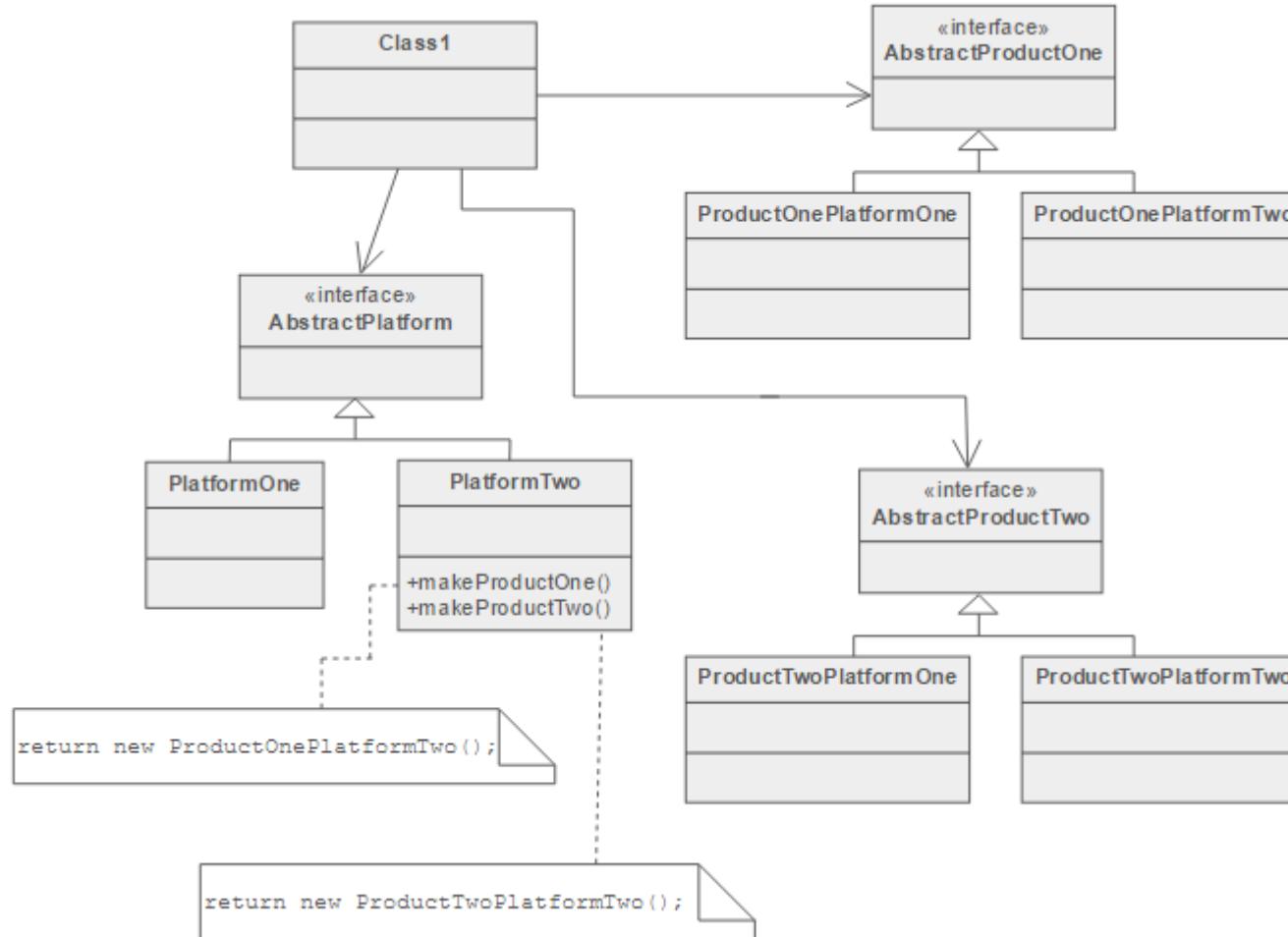
- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- A hierarchy that encapsulates many possible "platforms", and the construction of a suite of "products".
- The new operator considered harmful.

Problem:

If an application is to be portable, it needs to encapsulate platform dependencies. These "platforms" might include: windowing system, operating system, database, etc. Too often, this encapsulation is not engineered in advance, and lots of #ifdef case statements with options for all currently supported platforms begin to procreate like rabbits throughout the code.

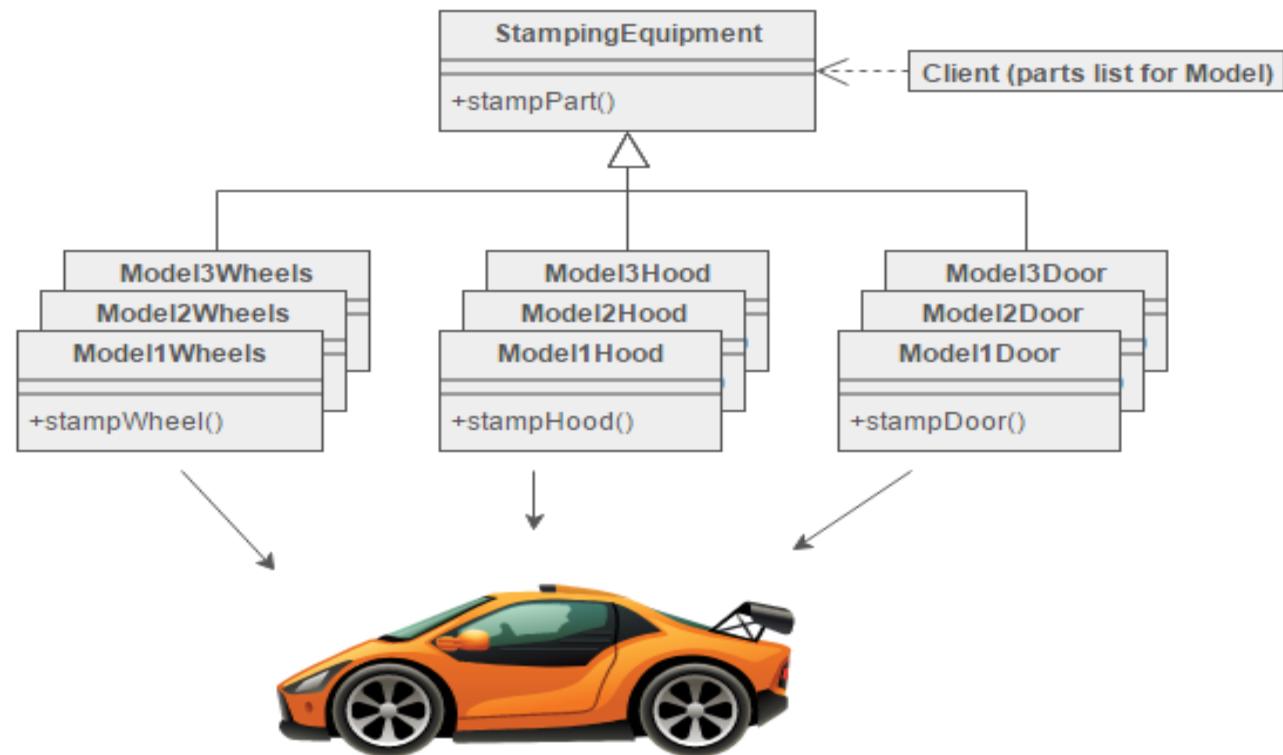
Structure

The Abstract Factory defines a Factory Method per product. Each Factory Method encapsulates the new operator and the concrete, platform-specific, product classes. Each "platform" is then modeled with a Factory derived class.



Example

The purpose of the Abstract Factory is to provide an interface for creating families of related objects, without specifying concrete classes. This pattern is found in the sheet metal stamping equipment used in the manufacture of Japanese automobiles. The stamping equipment is an Abstract Factory which creates auto body parts. The same machinery is used to stamp right hand doors, left hand doors, right front fenders, left front fenders, hoods, etc. for different models of cars. Through the use of rollers to change the stamping dies, the concrete classes produced by the machinery can be changed within three minutes.



Check list

- Decide if "platform independence" and creation services are the current source of pain.
- Map out a matrix of "platforms" versus "products".
- Define a factory interface that consists of a factory method per product.
- Define a factory derived class for each platform that encapsulates all references to the new operator.
- The client should retire all references to new, and use the factory methods to create the product objects.

Builder Design Pattern

Intent:

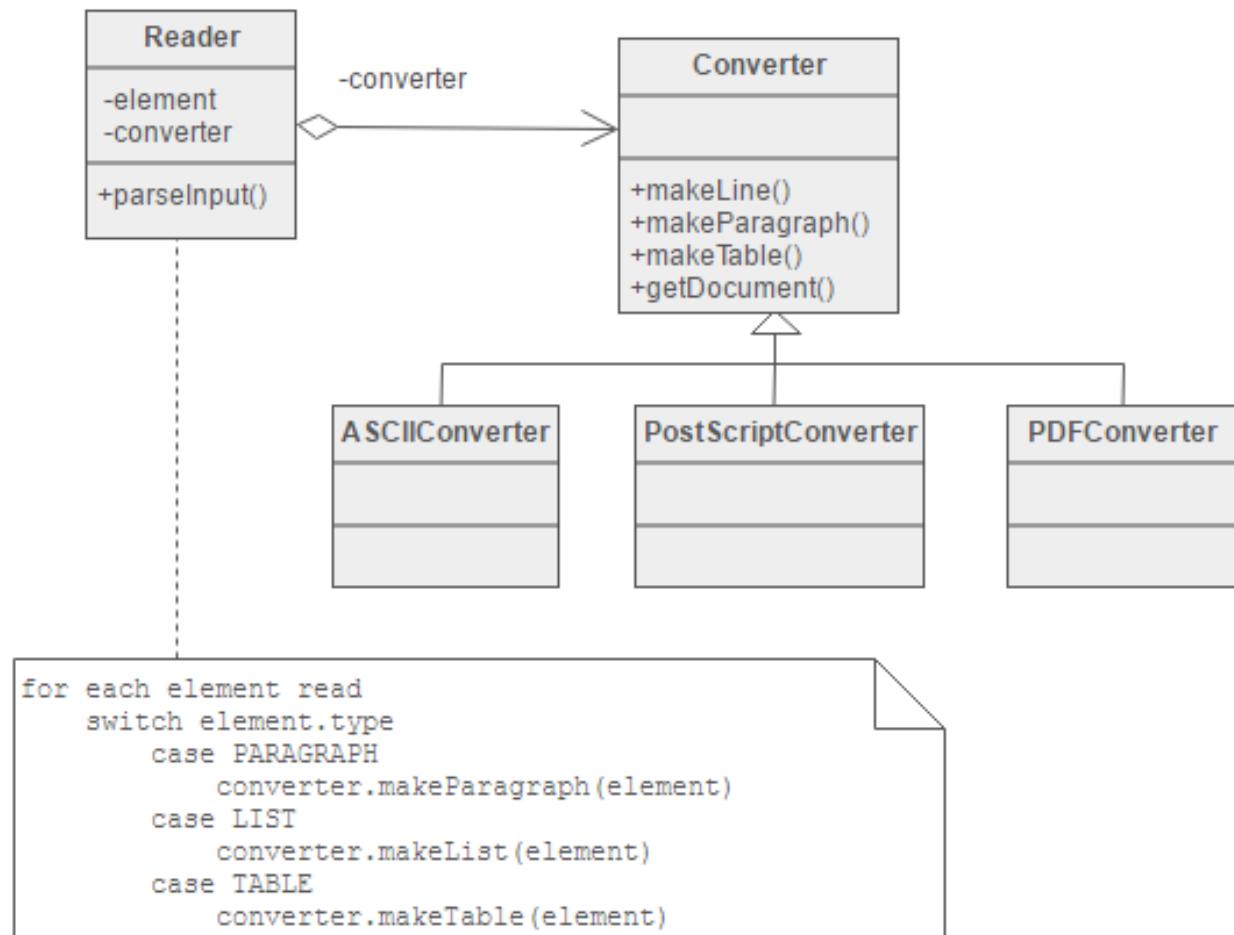
- Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- Parse a complex representation, create one of several targets.

Problem:

An application needs to create the elements of a complex aggregate. The specification for the aggregate exists on secondary storage and one of many representations needs to be built in primary storage.

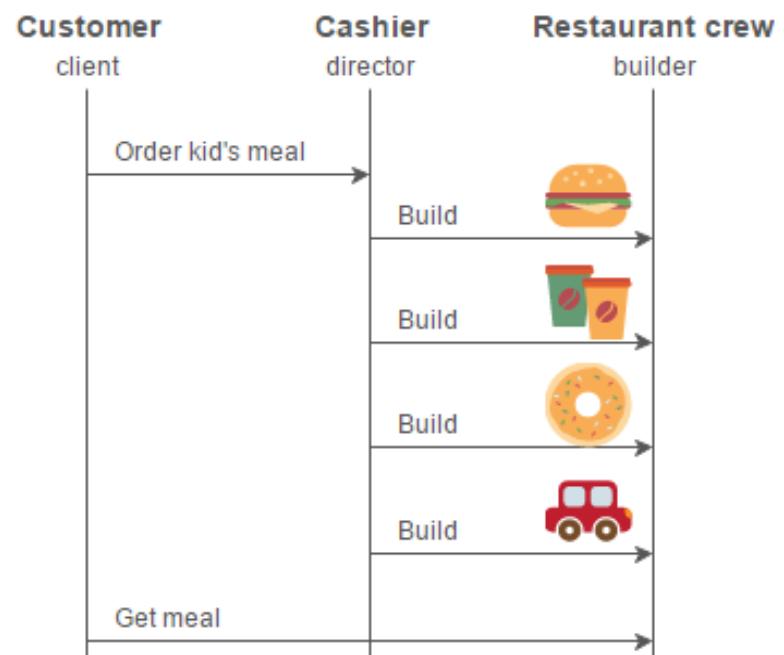
Structure

The Reader encapsulates the parsing of the common input. The Builder hierarchy makes possible the polymorphic creation of many peculiar representations or targets.



Example

The Builder pattern separates the construction of a complex object from its representation so that the same construction process can create different representations. This pattern is used by fast food restaurants to construct children's meals. Children's meals typically consist of a main item, a side item, a drink, and a toy (e.g., a hamburger, fries, Coke, and toy dinosaur). Note that there can be variation in the content of the children's meal, but the construction process is the same. Whether a customer orders a hamburger, cheeseburger, or chicken, the process is the same. The employee at the counter directs the crew to assemble a main item, side item, and toy. These items are then placed in a bag. The drink is placed in a cup and remains outside of the bag. This same process is used at competing restaurants.



Check list

- Decide if a common input and many possible representations (or outputs) is the problem at hand.
- Encapsulate the parsing of the common input in a Reader class.
- Design a standard protocol for creating all possible output representations. Capture the steps of this protocol in a Builder interface.
- Define a Builder derived class for each target representation.
- The client creates a Reader object and a Builder object, and registers the latter with the former.
- The client asks the Reader to "construct".
- The client asks the Builder to return the result.

Factory Method

Intent

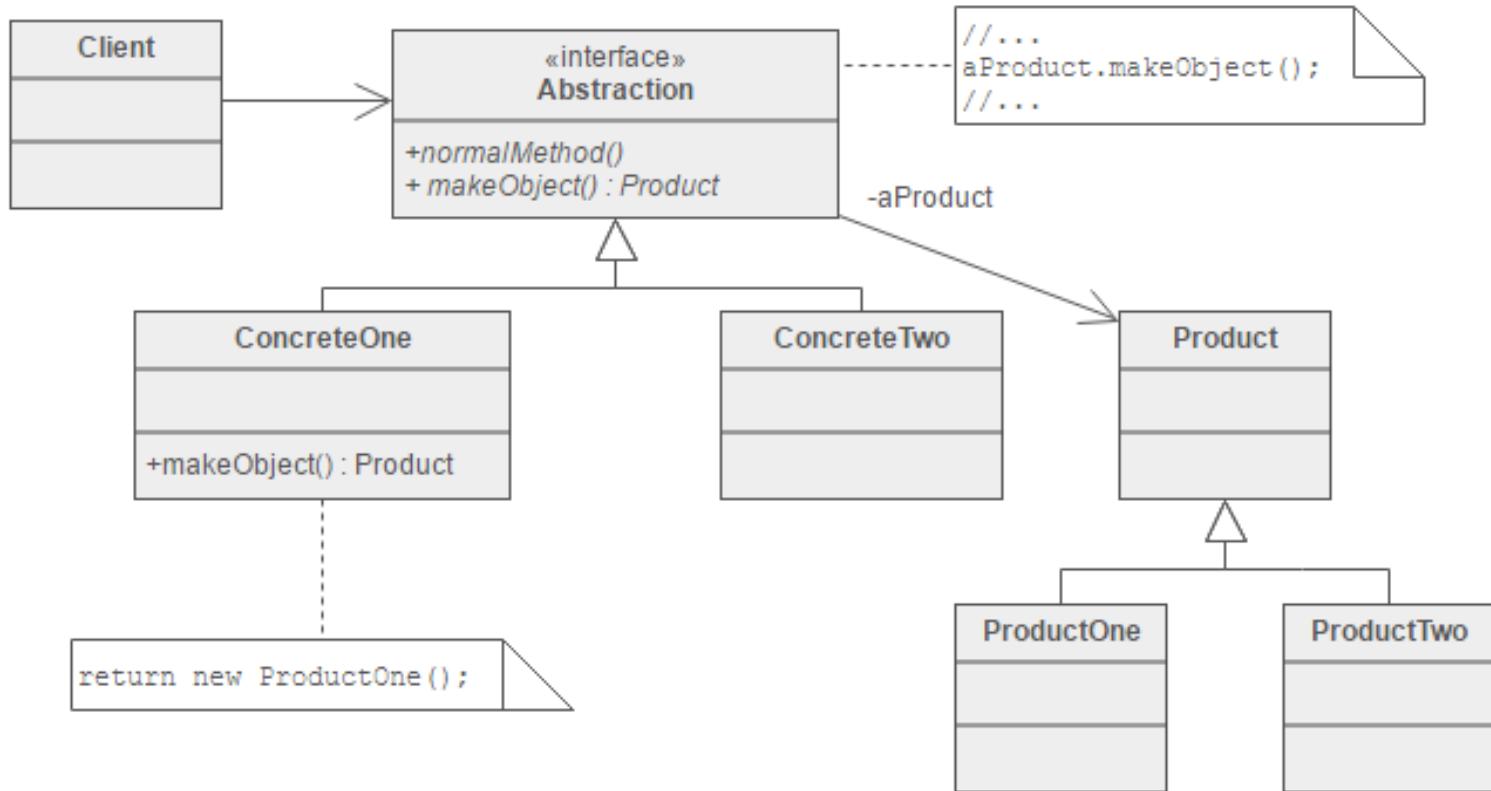
- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Defining a "virtual" constructor.
- The new operator considered harmful.

Problem

A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.

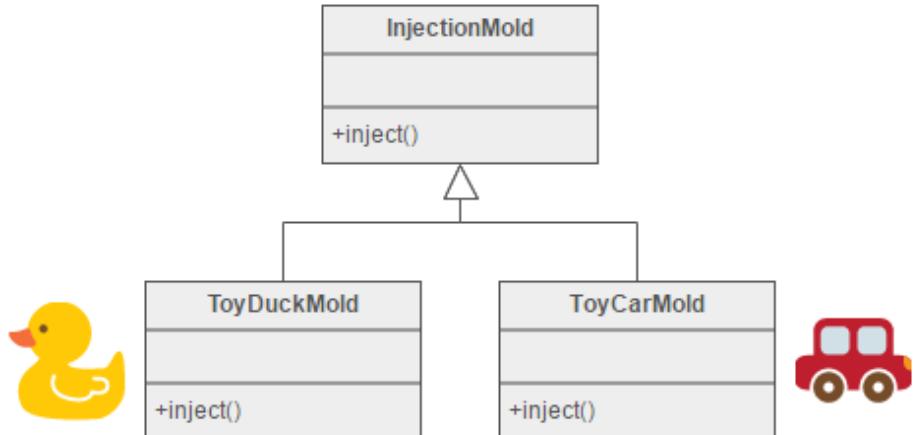
Structure

The implementation of Factory Method discussed in the Gang of Four (below) largely overlaps with that of Abstract Factory. For that reason, the presentation here focuses on the approach that has become popular since.



Example

The Factory Method defines an interface for creating objects, but lets subclasses decide which classes to instantiate. Injection molding presses demonstrate this pattern. Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes. The class of toy (car, action figure, etc.) is determined by the mold.



Check list

- If you have an inheritance hierarchy that exercises polymorphism, consider adding a polymorphic creation capability by defining a static factory method in the base class.
- Design the arguments to the factory method. What qualities or characteristics are necessary and sufficient to identify the correct derived class to instantiate?
- Consider designing an internal "object pool" that will allow objects to be reused instead of created from scratch.
- Consider making all constructors private or protected.

Prototype Design Pattern

Intent

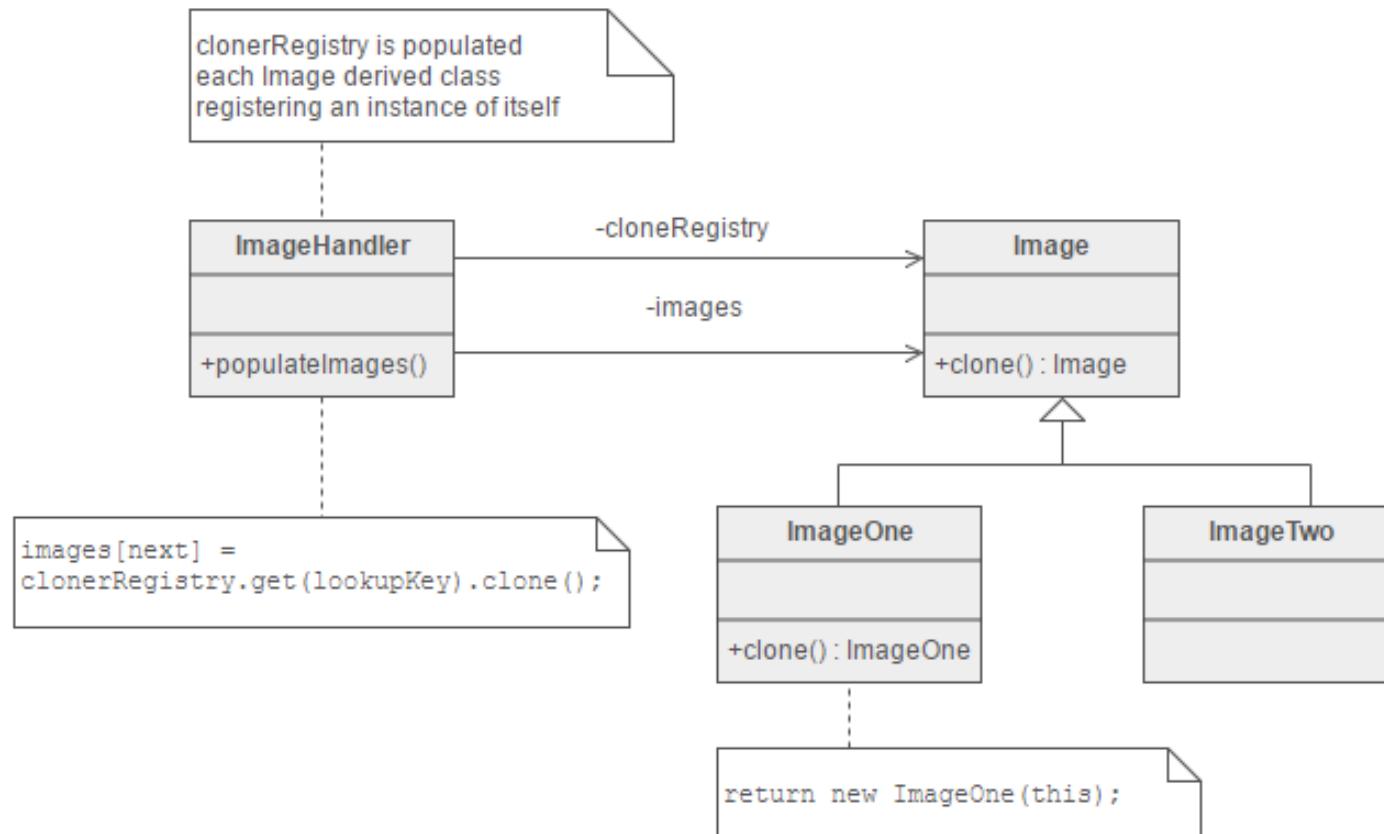
- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- Co-opt one instance of a class for use as a breeder of all future instances.
- The new operator considered harmful.

Problem

Application "hard wires" the class of object to create in each "new" expression.

Structure

The Factory knows how to find the correct Prototype, and each Product knows how to spawn new instances of itself.

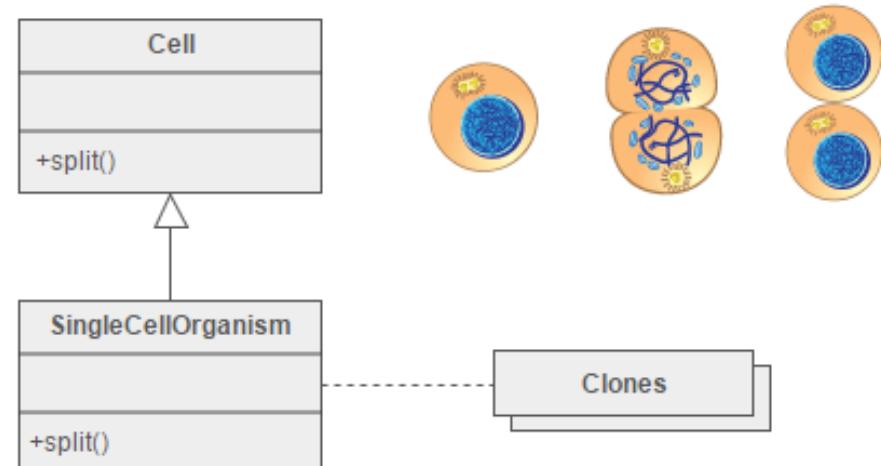


Example

The Prototype pattern specifies the kind of objects to create using a prototypical instance.

Prototypes of new products are often built prior to full production, but in this example, the prototype is passive and does not participate in copying itself.

The mitotic division of a cell - resulting in two identical cells - is an example of a prototype that plays an active role in copying itself and thus, demonstrates the Prototype pattern. When a cell splits, two cells of identical genotype result. In other words, the cell clones itself.



Check list

- Add a `clone()` method to the existing "product" hierarchy.
- Design a "registry" that maintains a cache of prototypical objects. The registry could be encapsulated in a new Factory class, or in the base class of the "product" hierarchy.
- Design a factory method that: may (or may not) accept arguments, finds the correct prototype object, calls `clone()` on that object, and returns the result.
- The client replaces all references to the new operator with calls to the factory method.

Singleton Design Pattern

Intent

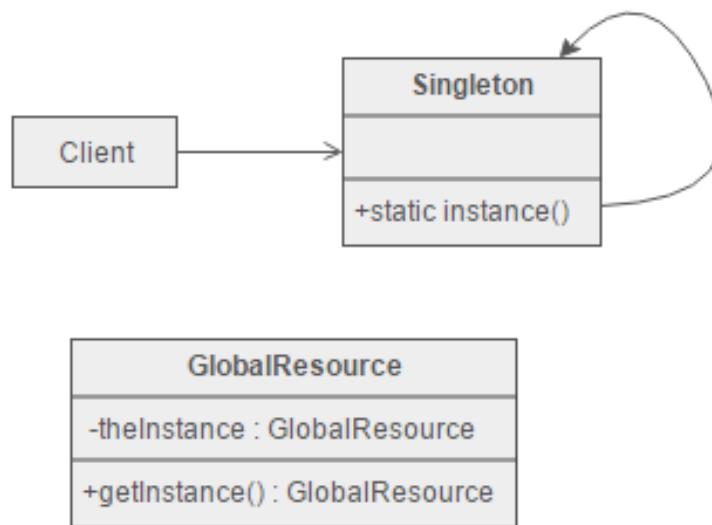
- Ensure a class has only one instance, and provide a global point of access to it.
- Encapsulated "just-in-time initialization" or "initialization on first use".

Problem

Application needs one, and only one, instance of an object.
Additionally, lazy initialization and global access are necessary.

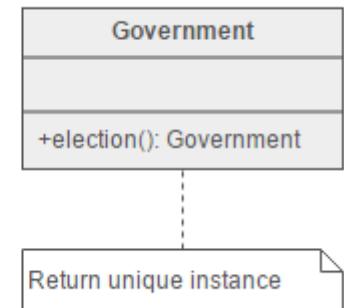
Structure

Make the class of the single instance responsible for access and "initialization on first use". The single instance is a private static attribute. The accessor function is a public static method.



Example

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. It is named after the singleton set, which is defined to be a set containing one element. The office of the President of the United States is a Singleton. The United States Constitution specifies the means by which a president is elected, limits the term of office, and defines the order of succession. As a result, there can be at most one active president at any given time. Regardless of the personal identity of the active president, the title, "The President of the United States" is a global point of access that identifies the person in the office.



Check list

- Define a private static attribute in the "single instance" class.
- Define a public static accessor function in the class.
- Do "lazy initialization" (creation on first use) in the accessor function.
- Define all constructors to be protected or private.
- Clients may only use the accessor function to manipulate the Singleton.

Structural patterns

Adapter Design Pattern

Intent

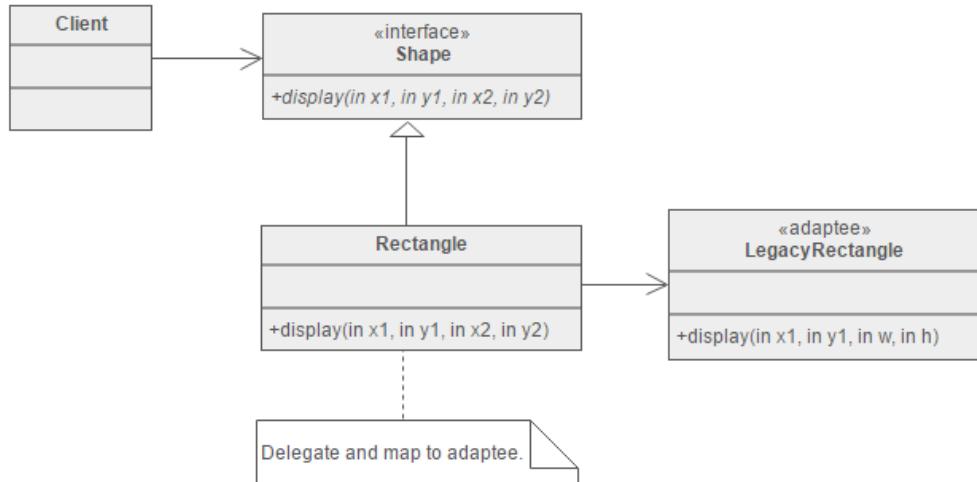
- Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Wrap an existing class with a new interface.
- Impedance match an old component to a new system

Problem

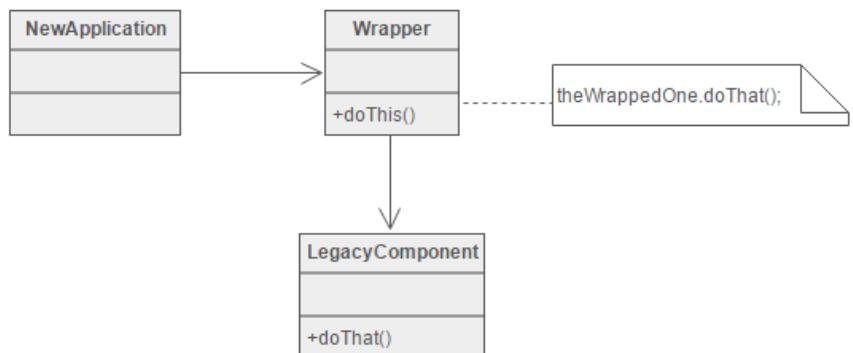
An "off the shelf" component offers compelling functionality that you would like to reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed.

Structure

Below, a legacy Rectangle component's display() method expects to receive "x, y, w, h" parameters. But the client wants to pass "upper left x and y" and "lower right x and y". This incongruity can be reconciled by adding an additional level of indirection – i.e. an Adapter object.

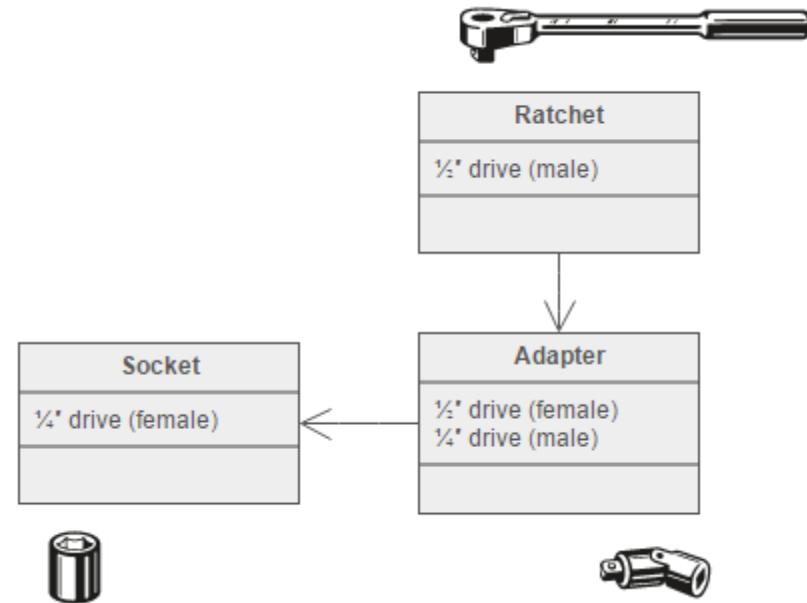


The Adapter could also be thought of as a "wrapper".



Example

The Adapter pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients. Socket wrenches provide an example of the Adapter. A socket attaches to a ratchet, provided that the size of the drive is the same. Typical drive sizes in the United States are 1/2" and 1/4". Obviously, a 1/2" drive ratchet will not fit into a 1/4" drive socket unless an adapter is used. A 1/2" to 1/4" adapter has a 1/2" female connection to fit on the 1/2" drive ratchet, and a 1/4" male connection to fit in the 1/4" drive socket.



Check list

- Identify the players: the component(s) that want to be accommodated (i.e. the client), and the component that needs to adapt (i.e. the adaptee).
- Identify the interface that the client requires.
- Design a "wrapper" class that can "impedance match" the adaptee to the client.
- The adapter/wrapper class "has a" instance of the adaptee class.
- The adapter/wrapper class "maps" the client interface to the adaptee interface.
- The client uses (is coupled to) the new interface

Bridge Design Pattern

Intent

- Decouple an abstraction from its implementation so that the two can vary independently.
- Publish interface in an inheritance hierarchy, and bury implementation in its own inheritance hierarchy.
- Beyond encapsulation, to insulation

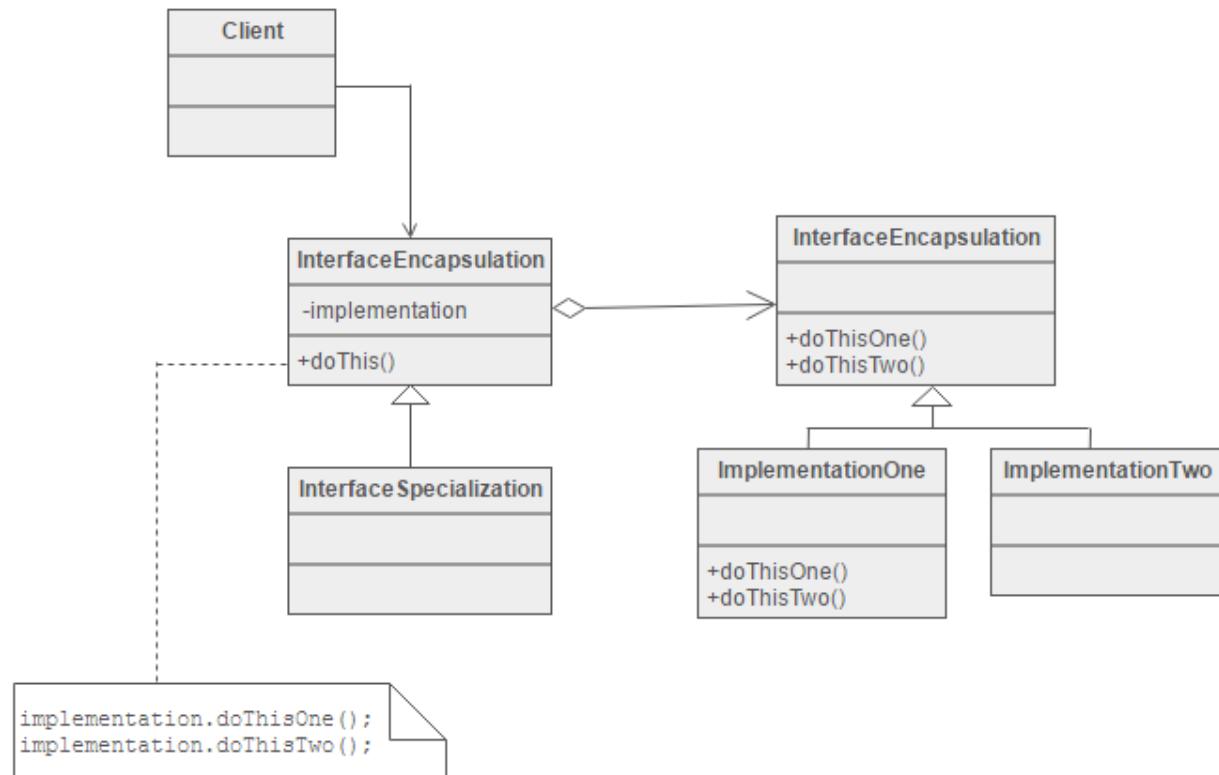
Problem

"Hardening of the software arteries" has occurred by using subclassing of an abstract base class to provide alternative implementations. This locks in compile-time binding between interface and implementation. The abstraction and implementation cannot be independently extended or composed.

Structure

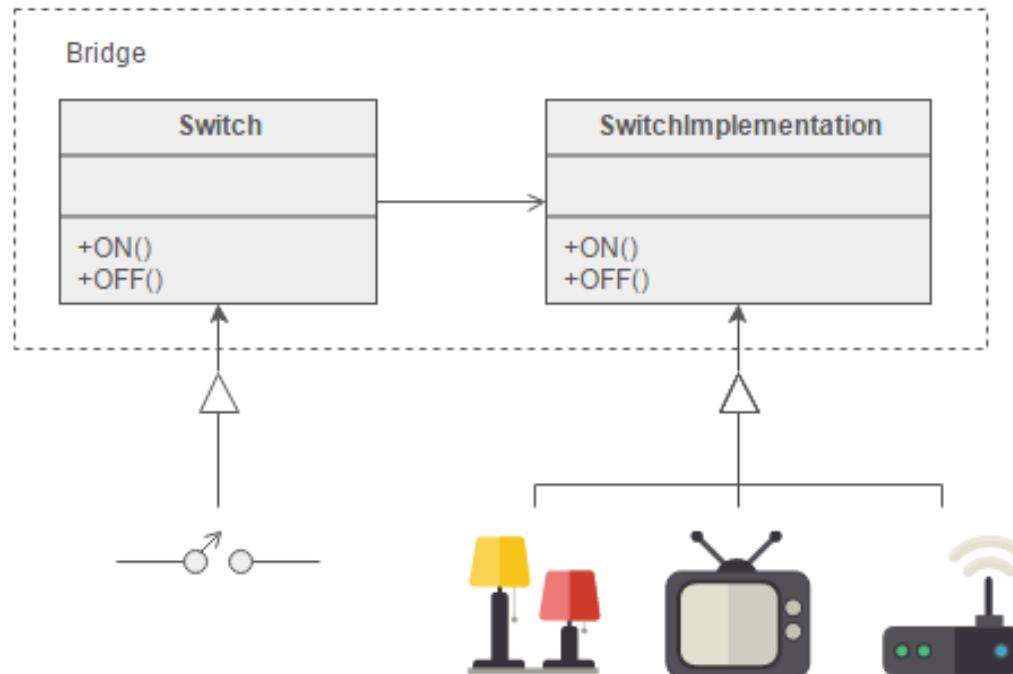
The Client doesn't want to deal with platform-dependent details. The Bridge pattern encapsulates this complexity behind an abstraction "wrapper".

Bridge emphasizes identifying and decoupling "interface" abstraction from "implementation" abstraction



Example

The Bridge pattern decouples an abstraction from its implementation, so that the two can vary independently. A household switch controlling lights, ceiling fans, etc. is an example of the Bridge. The purpose of the switch is to turn a device on or off. The actual switch can be implemented as a pull chain, simple two position switch, or a variety of dimmer switches.



Check list

- Decide if two orthogonal dimensions exist in the domain. These independent concepts could be: abstraction/platform, or domain/infrastructure, or front-end/back-end, or interface/implementation.
- Design the separation of concerns: what does the client want, and what do the platforms provide.
- Design a platform-oriented interface that is minimal, necessary, and sufficient. Its goal is to decouple the abstraction from the platform.
- Define a derived class of that interface for each platform.
- Create the abstraction base class that "has a" platform object and delegates the platform-oriented functionality to it.
- Define specializations of the abstraction class if desired.

Composite Design Pattern

Intent

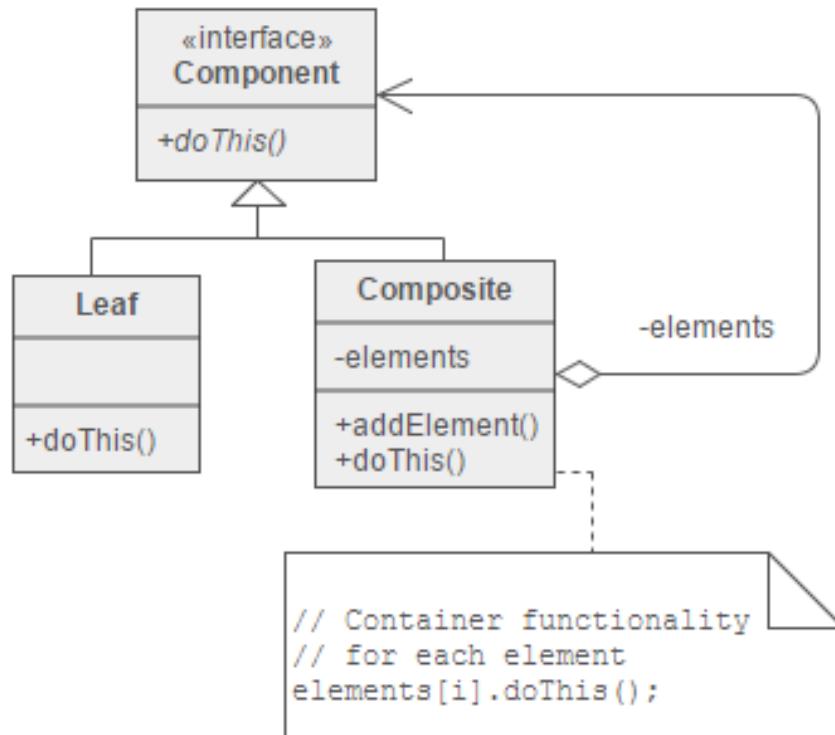
- Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Recursive composition
- "Directories contain entries, each of which could be a directory."
- 1-to-many "has a" up the "is a" hierarchy

Problem

Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently. Having to query the "type" of each object before attempting to process it is not desirable.

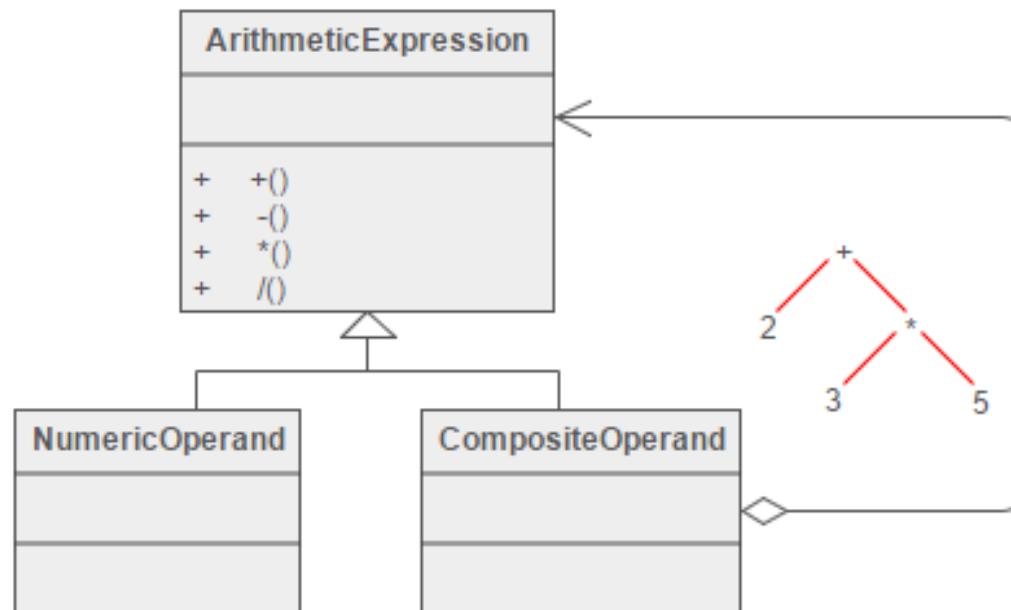
Structure

Composites that contain Components, each of which could be a Composite



Example

The Composite composes objects into tree structures and lets clients treat individual objects and compositions uniformly. Although the example is abstract, arithmetic expressions are Composites. An arithmetic expression consists of an operand, an operator (+ - * /), and another operand. The operand can be a number, or another arithmetic expression. Thus, $2 + 3$ and $(2 + 3) + (4 * 6)$ are both valid expressions.



Check list

- Ensure that your problem is about representing "whole-part" hierarchical relationships.
- Consider the heuristic, "Containers that contain containees, each of which could be a container." For example, "Assemblies that contain components, each of which could be an assembly." Divide your domain concepts into container classes, and containee classes.
- Create a "lowest common denominator" interface that makes your containers and containees interchangeable. It should specify the behavior that needs to be exercised uniformly across all containee and container objects.
- All container and containee classes declare an "is a" relationship to the interface.
- All container classes declare a one-to-many "has a" relationship to the interface.
- Container classes leverage polymorphism to delegate to their containee objects.
- Child management methods [e.g. addChild(), removeChild()] should normally be defined in the Composite class. Unfortunately, the desire to treat Leaf and Composite objects uniformly may require that these methods be promoted to the abstract Component class. See the Gang of Four for a discussion of these "safety" versus "transparency" trade-offs.

Decorator Design Pattern

Intent

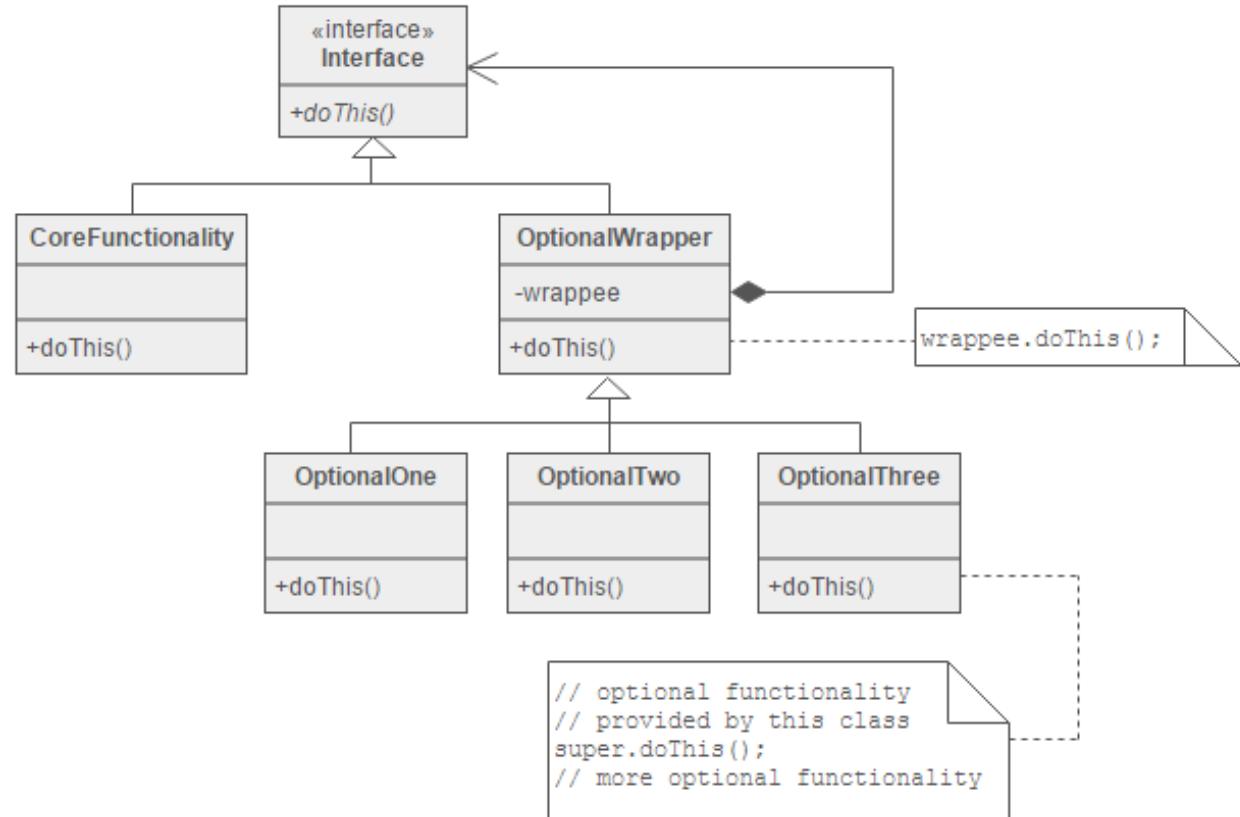
- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Client-specified embellishment of a core object by recursively wrapping it.
- Wrapping a gift, putting it in a box, and wrapping the box.

Problem

You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.

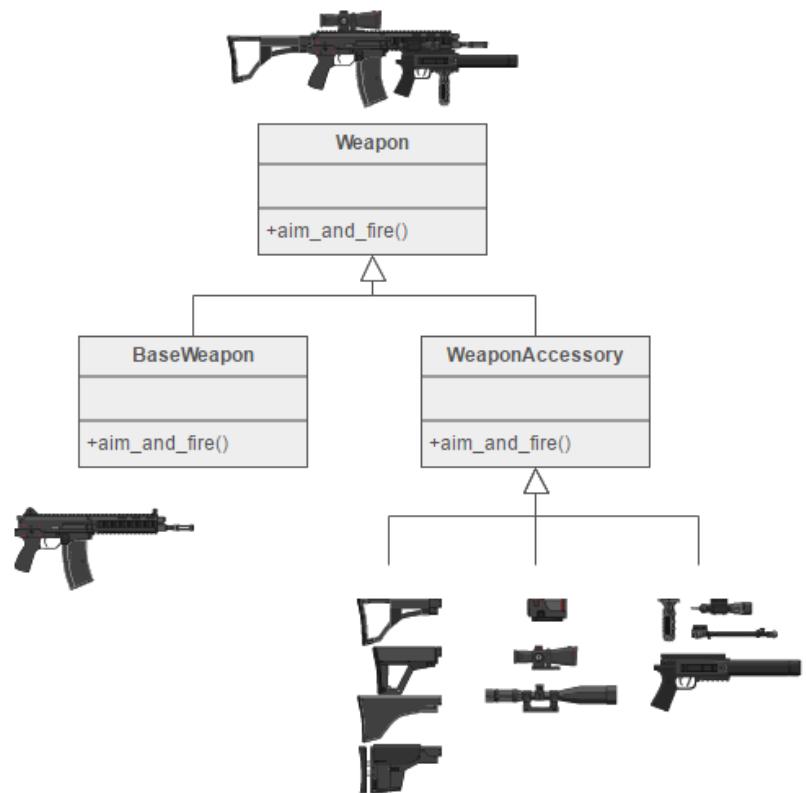
Structure

The client is always interested in CoreFunctionality.doThis(). The client may, or may not, be interested in OptionalOne.doThis() and OptionalTwo.doThis(). Each of these classes always delegate to the Decorator base class, and that class always delegates to the contained "wrappee" object.



Example

The Decorator attaches additional responsibilities to an object dynamically. The ornaments that are added to pine or fir trees are examples of Decorators. Lights, garland, candy canes, glass ornaments, etc., can be added to a tree to give it a festive look. The ornaments do not change the tree itself which is recognizable as a Christmas tree regardless of particular ornaments used. As an example of additional functionality, the addition of lights allows one to "light up" a Christmas tree.



Another example: assault gun is a deadly weapon on its own. But you can apply certain "decorations" to make it more accurate, silent and devastating.

Check list

- Ensure the context is: a single core (or non-optional) component, several optional embellishments or wrappers, and an interface that is common to all.
- Create a "Lowest Common Denominator" interface that makes all classes interchangeable.
- Create a second level base class (Decorator) to support the optional wrapper classes.
- The Core class and Decorator class inherit from the LCD interface.
- The Decorator class declares a composition relationship to the LCD interface, and this data member is initialized in its constructor.
- The Decorator class delegates to the LCD object.
- Define a Decorator derived class for each optional embellishment.
- Decorator derived classes implement their wrapper functionality - and - delegate to the Decorator base class.
- The client configures the type and ordering of Core and Decorator objects.

Facade Design Pattern

Intent

- Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Wrap a complicated subsystem with a simpler interface.

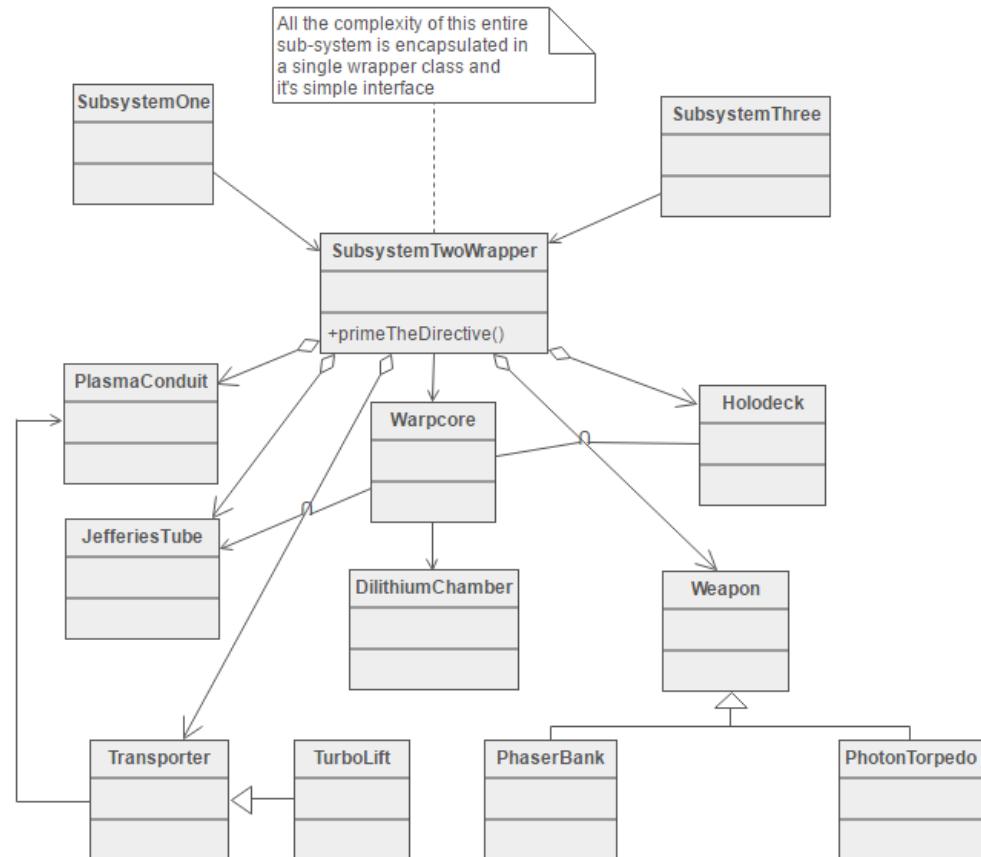
Problem

A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.

Structure

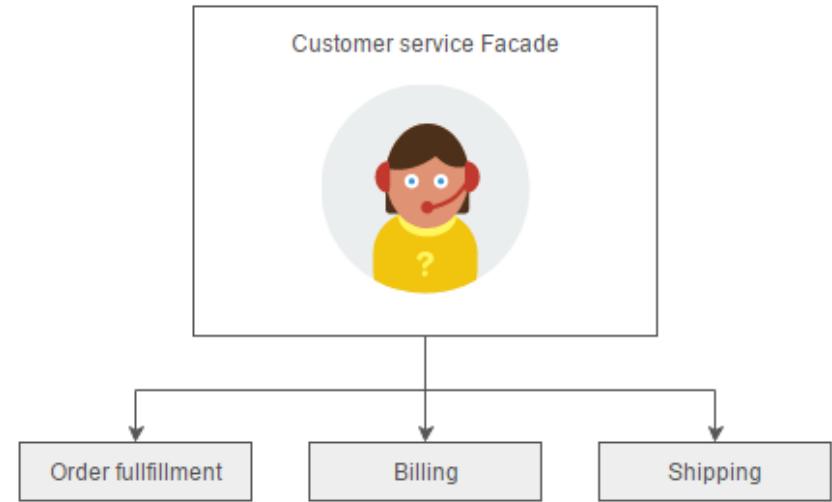
Facade takes a "riddle wrapped in an enigma shrouded in mystery", and interjects a wrapper that tames the amorphous and inscrutable mass of software.

SubsystemOne and SubsystemThree do not interact with the internal components of SubsystemTwo. They use the SubsystemTwoWrapper "facade" (i.e. the higher level abstraction).



Example

The Facade defines a unified, higher level interface to a subsystem that makes it easier to use. Consumers encounter a Facade when ordering from a catalog. The consumer calls one number and speaks with a customer service representative. The customer service representative acts as a Facade, providing an interface to the order fulfillment department, the billing department, and the shipping department.



Check list

- Identify a simpler, unified interface for the subsystem or component.
- Design a 'wrapper' class that encapsulates the subsystem.
- The facade/wrapper captures the complexity and collaborations of the component, and delegates to the appropriate methods.
- The client uses (is coupled to) the Facade only.
- Consider whether additional Facades would add value.

Proxy Design Pattern

Intent

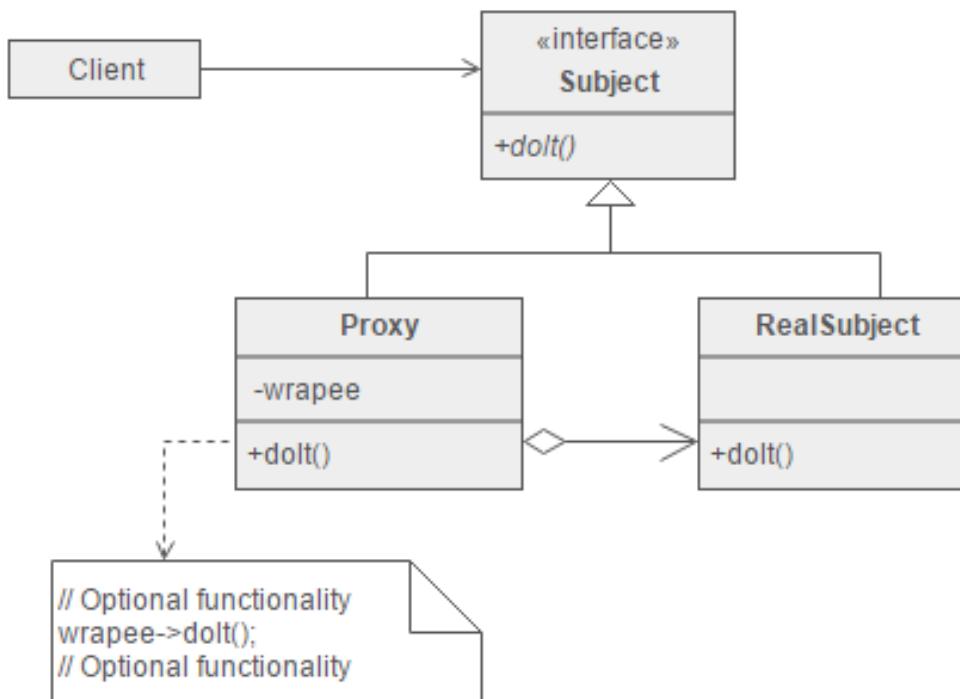
- Provide a surrogate or placeholder for another object to control access to it.
- Use an extra level of indirection to support distributed, controlled, or intelligent access.
- Add a wrapper and delegation to protect the real component from undue complexity.

Problem

You need to support resource-hungry objects, and you do not want to instantiate such objects unless and until they are actually requested by the client.

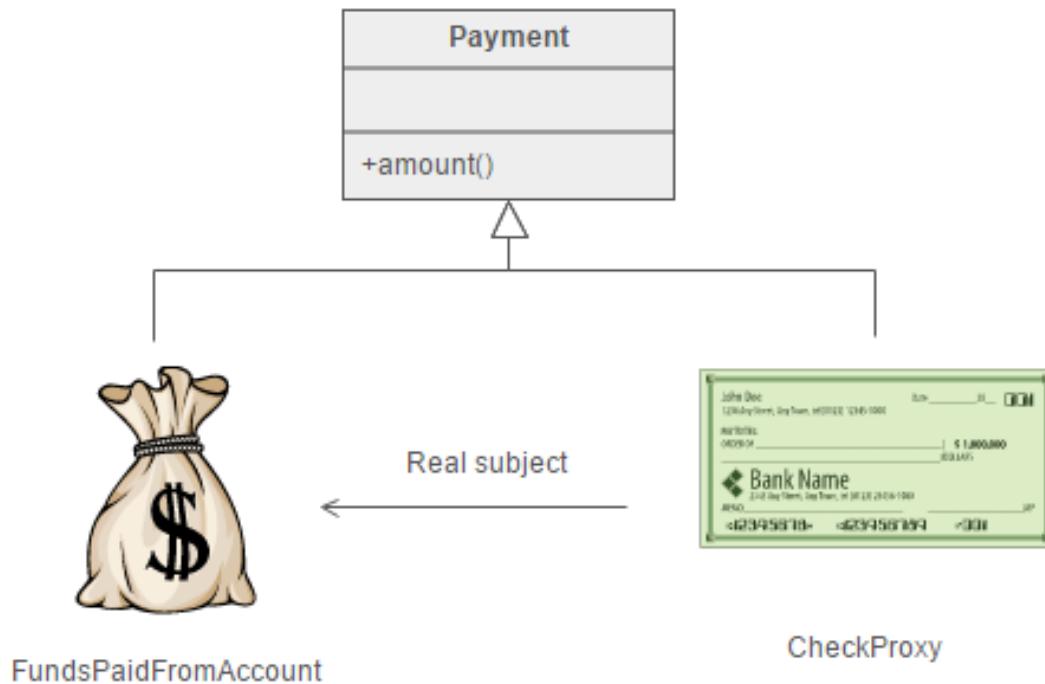
Structure

By defining a Subject interface, the presence of the Proxy object standing in place of the RealSubject is transparent to the client.



Example

The Proxy provides a surrogate or place holder to provide access to an object. A check or bank draft is a proxy for funds in an account. A check can be used in place of cash for making purchases and ultimately controls access to cash in the issuer's account.



Check list

- Identify the leverage or "aspect" that is best implemented as a wrapper or surrogate.
- Define an interface that will make the proxy and the original component interchangeable.
- Consider defining a Factory that can encapsulate the decision of whether a proxy or original object is desirable.
- The wrapper class holds a pointer to the real class and implements the interface.
- The pointer may be initialized at construction, or on first use.
- Each wrapper method contributes its leverage, and delegates to the wrappee object.

Behavioral patterns

Chain of Responsibility

Intent

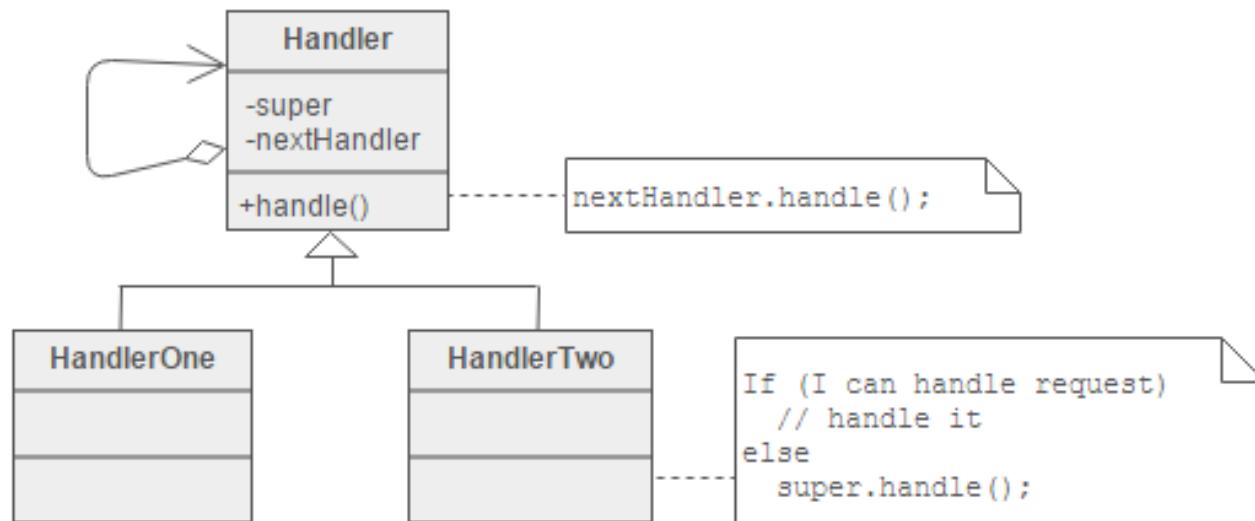
- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- Launch-and-leave requests with a single processing pipeline that contains many possible handlers.
- An object-oriented linked list with recursive traversal.

Problem

There is a potentially variable number of "handler" or "processing element" or "node" objects, and a stream of requests that must be handled. Need to efficiently process the requests without hard-wiring handler relationships and precedence, or request-to-handler mappings.

Structure

The derived classes know how to satisfy Client requests. If the "current" object is not available or sufficient, then it delegates to the base class, which delegates to the "next" object, and the circle of life continues.



Example

The Chain of Responsibility pattern avoids coupling the sender of a request to the receiver by giving more than one object a chance to handle the request. ATM use the Chain of Responsibility in money giving mechanism.



Check list

- The base class maintains a "next" pointer.
- Each derived class implements its contribution for handling the request.
- If the request needs to be "passed on", then the derived class "calls back" to the base class, which delegates to the "next" pointer.
- The client (or some third party) creates and links the chain (which may include a link from the last node to the root node).
- The client "launches and leaves" each request with the root of the chain.
- Recursive delegation produces the illusion of magic.

Iterator Design Pattern

Intent

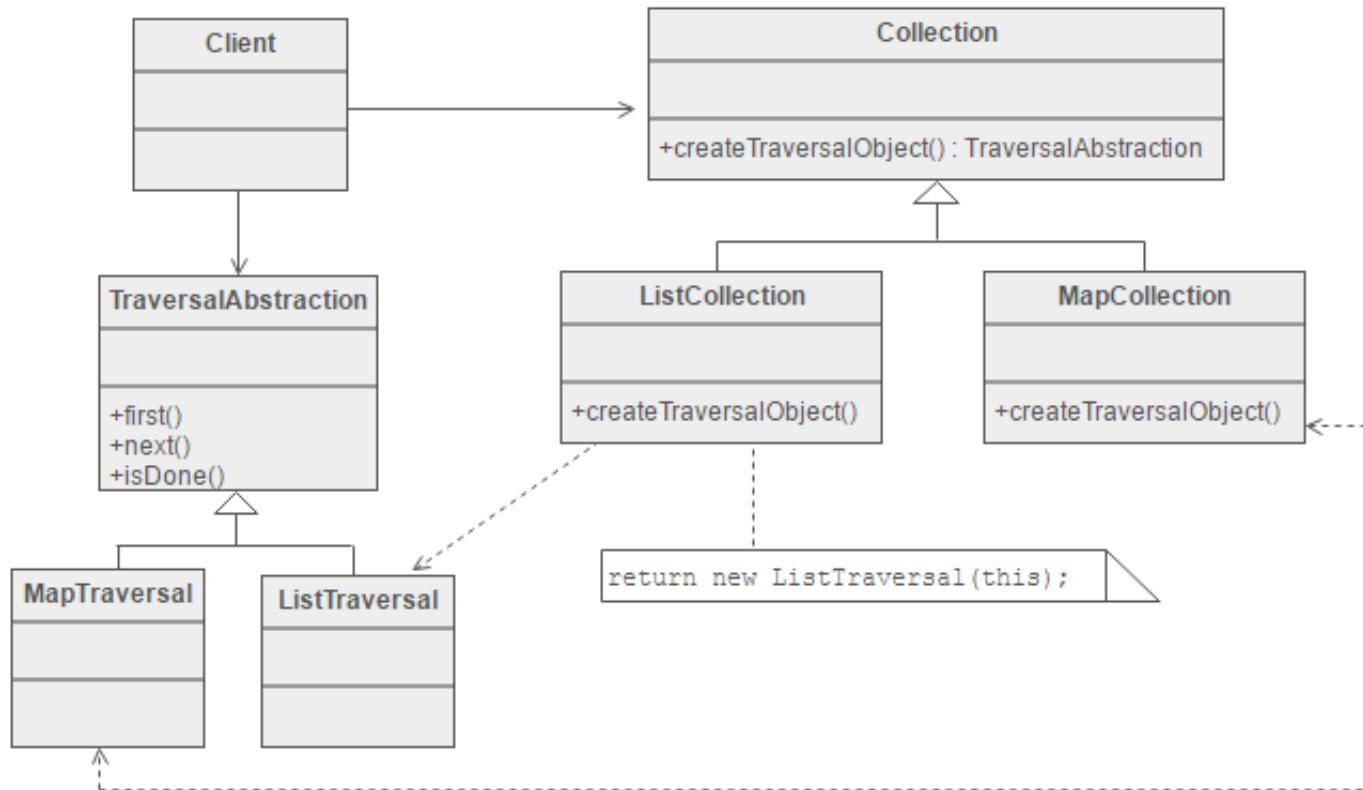
- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- The C++ and Java standard library abstraction that makes it possible to decouple collection classes and algorithms.
- Promote to "full object status" the traversal of a collection.
- Polymorphic traversal

Problem

Need to "abstract" the traversal of wildly different data structures so that algorithms can be defined that are capable of interfacing with each transparently.

Structure

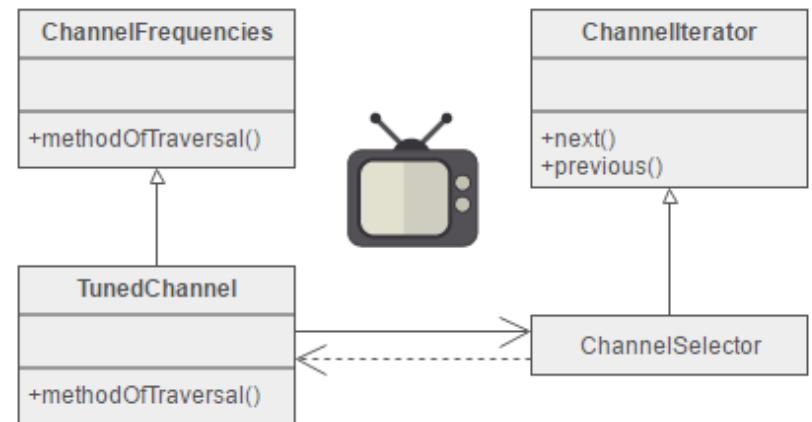
The Client uses the Collection class' public interface directly. But access to the Collection's elements is encapsulated behind the additional level of abstraction called Iterator. Each Collection derived class knows which Iterator derived class to create and return. After that, the Client relies on the interface defined in the Iterator base class.



Example

The Iterator provides ways to access elements of an aggregate object sequentially without exposing the underlying structure of the object. Files are aggregate objects. In office settings where access to files is made through administrative or secretarial staff, the Iterator pattern is demonstrated with the secretary acting as the Iterator. Several television comedy skits have been developed around the premise of an executive trying to understand the secretary's filing system. To the executive, the filing system is confusing and illogical, but the secretary is able to access files quickly and efficiently.

On early television sets, a dial was used to change channels. When channel surfing, the viewer was required to move the dial through each channel position, regardless of whether or not that channel had reception. On modern television sets, a next and previous button are used. When the viewer selects the "next" button, the next tuned channel will be displayed. Consider watching television in a hotel room in a strange city. When surfing through channels, the channel number is not important, but the programming is. If the programming on one channel is not of interest, the viewer can request the next channel, without knowing its number.



Check list

- Add a `create_iterator()` method to the "collection" class, and grant the "iterator" class privileged access.
- Design an "iterator" class that can encapsulate traversal of the "collection" class.
- Clients ask the collection object to create an iterator object.
- Clients use the `first()`, `is_done()`, `next()`, and `current_item()` protocol to access the elements of the collection class.

Mediator Design Pattern

Intent

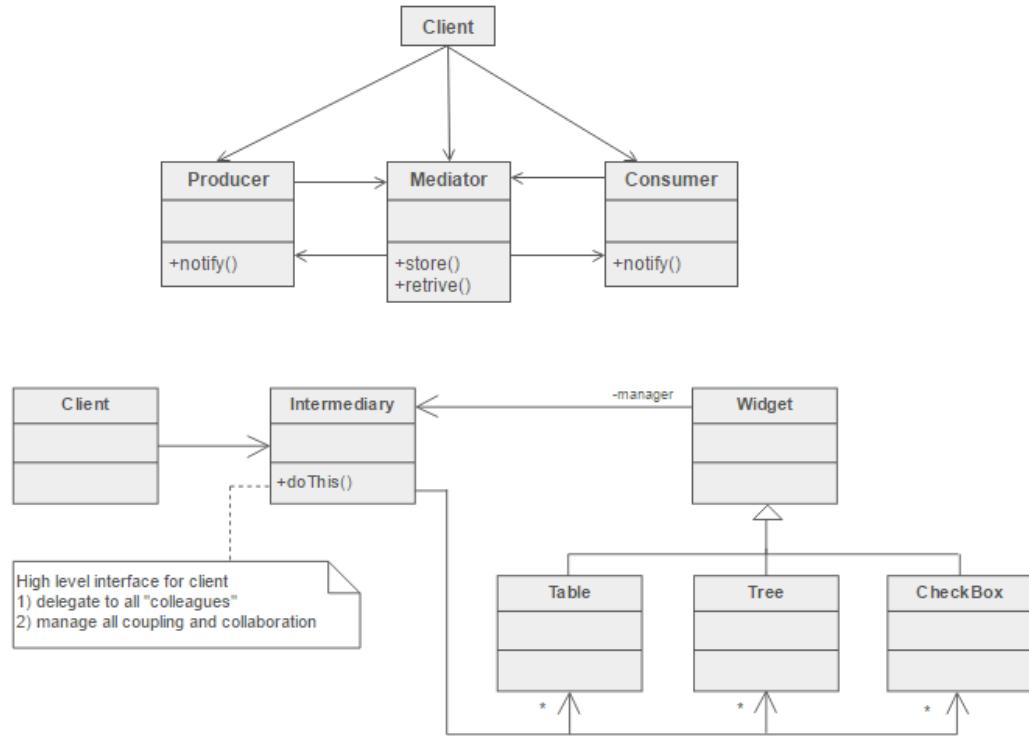
- Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- Design an intermediary to decouple many peers.
- Promote the many-to-many relationships between interacting peers to "full object status".

Problem

We want to design reusable components, but dependencies between the potentially reusable pieces demonstrates the "spaghetti code" phenomenon (trying to scoop a single serving results in an "all or nothing clump").

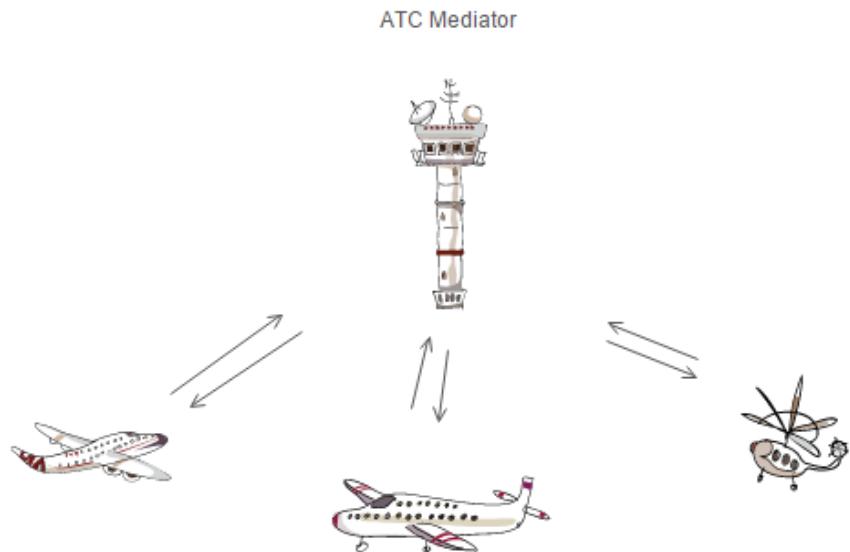
Structure

Colleagues (or peers) are not coupled to one another. Each talks to the Mediator, which in turn knows and conducts the orchestration of the others. The "many to many" mapping between colleagues that would otherwise exist, has been "promoted to full object status". This new abstraction provides a locus of indirection where additional leverage can be hosted.



Example

The Mediator defines an object that controls how a set of objects interact. Loose coupling between colleague objects is achieved by having colleagues communicate with the Mediator, rather than with each other. The control tower at a controlled airport demonstrates this pattern very well. The pilots of the planes approaching or departing the terminal area communicate with the tower rather than explicitly communicating with one another. The constraints on who can take off or land are enforced by the tower. It is important to note that the tower does not control the whole flight. It exists only to enforce constraints in the terminal area.



Check list

- Identify a collection of interacting objects that would benefit from mutual decoupling.
- Encapsulate those interactions in the abstraction of a new class.
- Create an instance of that new class and rework all "peer" objects to interact with the Mediator only.
- Balance the principle of decoupling with the principle of distributing responsibility evenly.
- Be careful not to create a "controller" or "god" object.

Observer Design Pattern

Intent

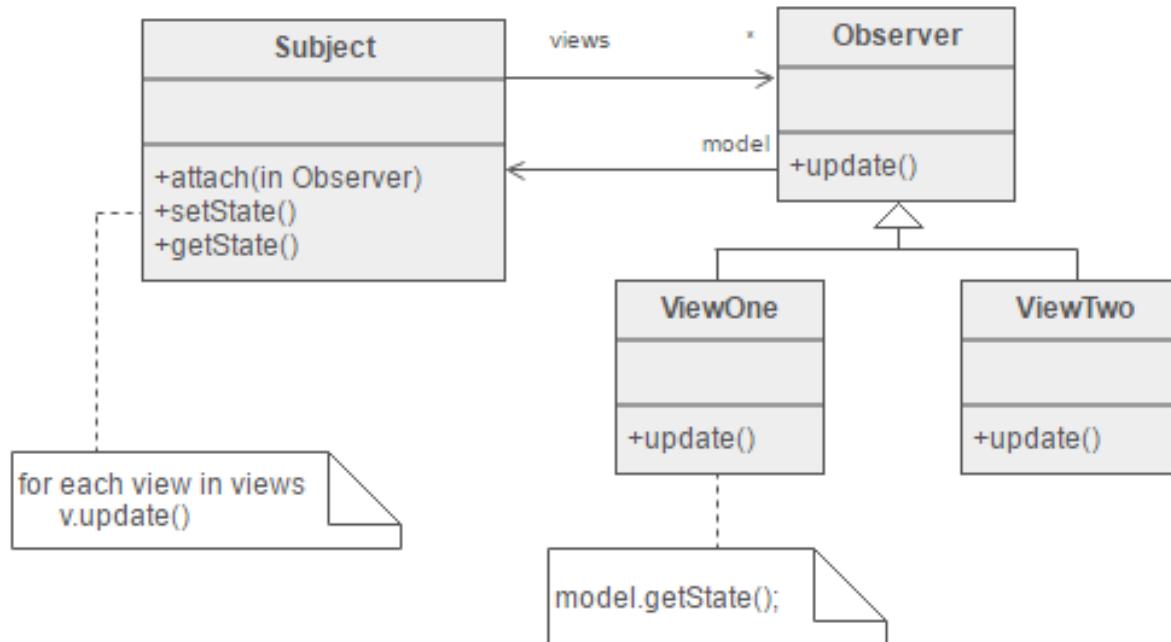
- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.
- The "View" part of Model-View-Controller.

Problem

A large monolithic design does not scale well as new graphing or monitoring requirements are levied.

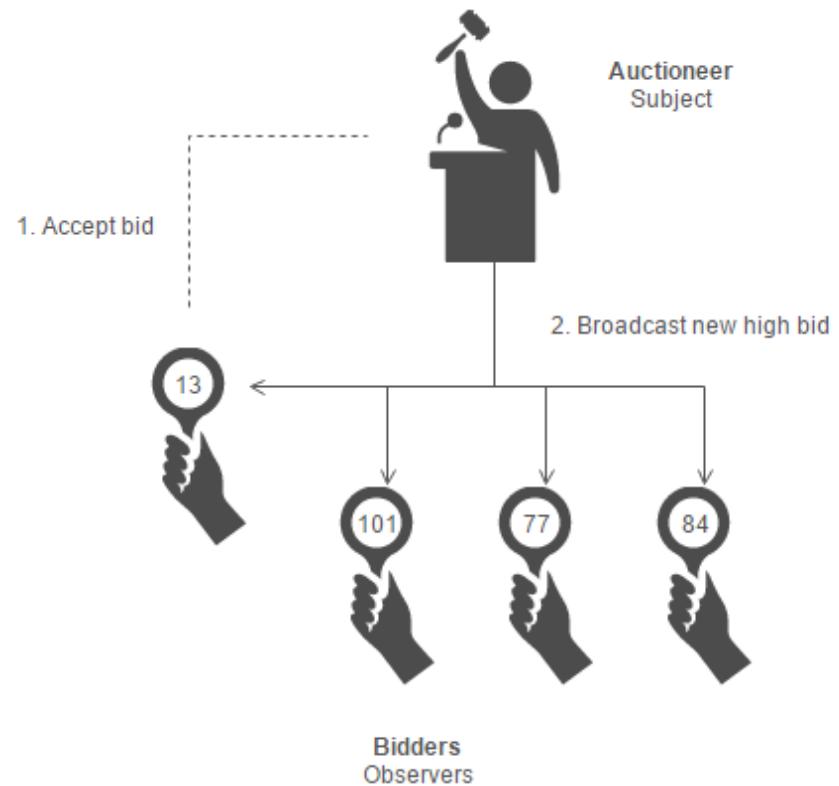
Structure

Subject represents the core (or independent or common or engine) abstraction. Observer represents the variable (or dependent or optional or user interface) abstraction. The Subject prompts the Observer objects to do their thing. Each Observer can call back to the Subject as needed.



Example

The Observer defines a one-to-many relationship so that when one object changes state, the others are notified and updated automatically. Some auctions demonstrate this pattern. Each bidder possesses a numbered paddle that is used to indicate a bid. The auctioneer starts the bidding, and "observes" when a paddle is raised to accept the bid. The acceptance of the bid changes the bid price which is broadcast to all of the bidders in the form of a new bid.



Check list

- Differentiate between the core (or independent) functionality and the optional (or dependent) functionality.
- Model the independent functionality with a "subject" abstraction.
- Model the dependent functionality with an "observer" hierarchy.
- The Subject is coupled only to the Observer base class.
- The client configures the number and type of Observers.
- Observers register themselves with the Subject.
- The Subject broadcasts events to all registered Observers.
- The Subject may "push" information at the Observers, or, the Observers may "pull" the information they need from the Subject.

Strategy Design Pattern

Intent

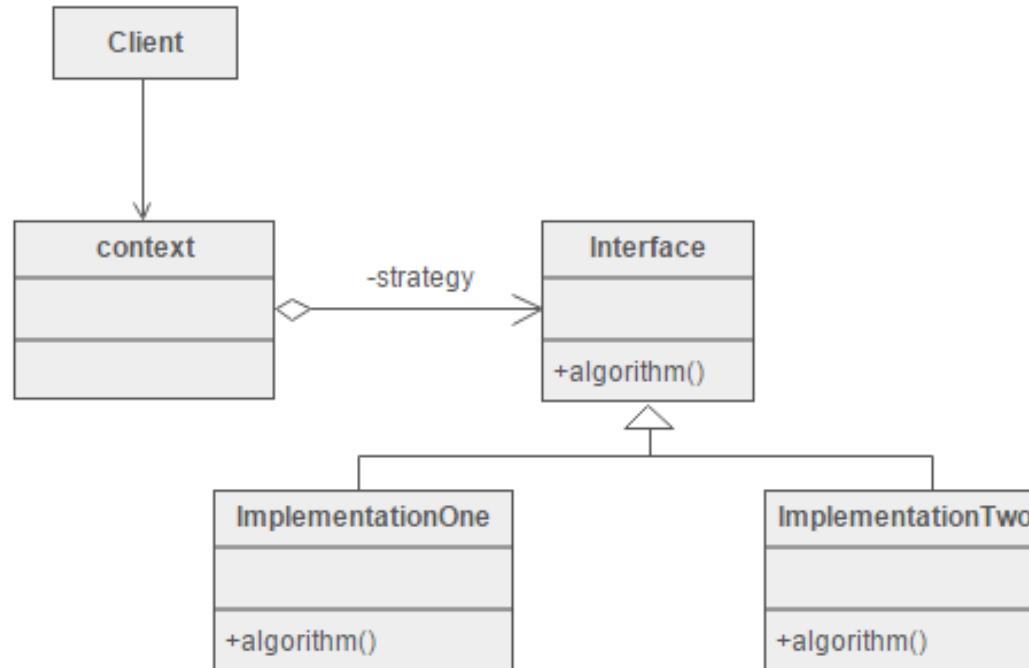
- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
- Capture the abstraction in an interface, bury implementation details in derived classes.

Problem

A generic value of the software community for years has been, "maximize cohesion and minimize coupling". The object-oriented design approach shown in figure is all about minimizing coupling. Since the client is coupled only to an abstraction (i.e. a useful fiction), and not a particular realization of that abstraction, the client could be said to be practicing "abstract coupling" . an object-oriented variant of the more generic exhortation "minimize coupling".

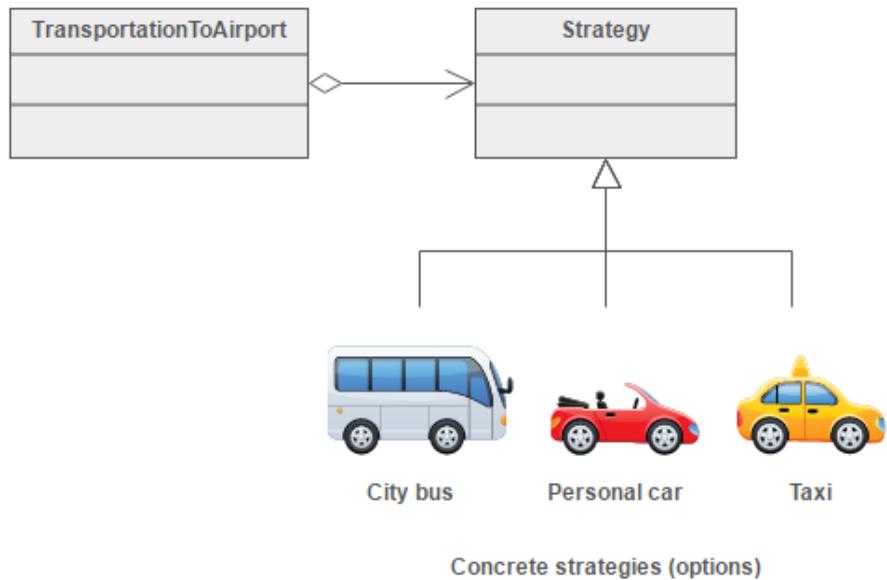
Structure

The Interface entity could represent either an abstract base class, or the method signature expectations by the client. In the former case, the inheritance hierarchy represents dynamic polymorphism. In the latter case, the Interface entity represents template code in the client and the inheritance hierarchy represents static polymorphism.



Example

A Strategy defines a set of algorithms that can be used interchangeably. Modes of transportation to an airport is an example of a Strategy. Several options exist such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service. For some airports, subways and helicopters are also available as a mode of transportation to the airport. Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably. The traveler must chose the Strategy based on trade-offs between cost, convenience, and time.



Check list

- Identify an algorithm (i.e. a behavior) that the client would prefer to access through a "flex point".
- Specify the signature for that algorithm in an interface.
- Bury the alternative implementation details in derived classes.
- Clients of the algorithm couple themselves to the interface.

SOLID Principle

What is SOLID

- Single responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Single Responsibility Principle

*“A class should have one and
only one reason to change”*

Single Responsibility Principle

```
6  public class Employee  
7  {  
8      public double CalculatePay(Money money)  
9      {  
10         //business logic for payment here  
11     }  
12  
13     public Employee Save(Employee employee)  
14     {  
15         //store employee here  
16     }  
17 }
```

Business logic

Persistence

There are **two** responsibilities

Single Responsibility Principle

How to solve this?



Single Responsibility Principle

```
21  public class Employee
22  {
23      public double CalculatePay(Money money)
24      {
25          //business logic for payment here
26      }
27  }
28
29  public class EmployeeRepository
30  {
31      public Employee Save(Employee employee)
32      {
33          //store employee here
34      }
35  }
```

Just create two
different classes

Open/Closed Principle

“Software entities should be open for extension, but closed for modification.”

Open/Closed Principle

```
40 public enum PaymentType = { Cash, CreditCard };
41
42 public class PaymentManager
43 {
44     public PaymentType PaymentType { get; set; }
45
46     public void Pay(Money money)
47     {
48         if(PaymentType == PaymentType.Cash)
49         {
50             //some code here - pay with cash
51         }
52         else
53         {
54             //some code here - pay with credit card
55         }
56     }
57 }
```



Humm...and if I
need to add a new
payment type?

You need to
modificate this
class.

Open/Closed Principle

open for
extension

close for
modification

```
60  public class Payment
61  {
62      public virtual void Pay(Money money)
63      {
64          // from base
65      }
66  }
```



```
68  public class CashPayment : Payment
69  {
70      public override void Pay(Money money)
71      {
72          //some code here - pay with cash
73      }
74  }
```

```
76  public class CreditCardPayment : Payment
77  {
78      public override void Pay(Money money)
79      {
80          //some code here - pay with credit card
81      }
82  }
```

Liskov Substitution Principle

“Let $q(x)$ be a property provable about objects x of type T . Then $q(y)$ should be provable for objects y of type S where S is a subtype of T ”

What do you say?

Liskov Substitution Principle

“A subclass should behave in such a way that it will not cause problems when used instead of the superclass.”

Liskov Substitution Principle

```
5  public class Employee
6  {
7      public virtual string GetProjectDetails(int employeeId)
8      {
9          Console.WriteLine("base project details");
10     }
11 }
```

```
13 public class CasualEmployee : Employee
14 {
15     public override string GetProjectDetails(int employeeId)
16     {
17         base.GetProjectDetails(employeeId);
18         Console.WriteLine("casual employee project details");
19     }
20 }
```

```
public class ContractualEmployee : Employee
{
    //broken your base class here
    public override string GetProjectDetails(int employeeId)
    {
        Console.WriteLine("contractual employee project details");
    }
}
```



Liskov Substitution Principle

```
5  public class Employee
6  {
7      public virtual string GetProjectDetails(int employeeId)
8      {
9          Console.WriteLine("base project details");
10     }
11 }
```

```
13 public class CasualEmployee : Employee
14 {
15     public override string GetProjectDetails(int employeeId)
16     {
17         base.GetProjectDetails(employeeId);
18         Console.WriteLine("casual employee project details");
19     }
20 }
```



Much better

```
public class ContractualEmployee : Employee
{
    public override string GetProjectDetails(int employeeId)
    {
        base.GetProjectDetails(employeeId);
        Console.WriteLine("contractual employee project details");
    }
}
```

Interface Segregation Principle

“Clients should not be forced to depend upon interfaces that they don't use”

Interface Segregation Principle

```
62  public interface IEmployee
63  {
64      string GetProjectDetails(int employeeId);
65
66      string GetEmployeeDetails(int employeeId);
67  }
68
```

Interface Segregation Principle

```
69  public class CasualEmployee : IEmployee  
70  {  
71      public string GetProjectDetails(int employeeId)  
72      {  
73          //code here - return base project details  
74      }  
75  
76      public string GetEmployeeDetails(int employeeId)  
77      {  
78          //code here - specific casual employee details  
79      }
```



WHY?????
I don't need you!!

```
82  public class ContractualEmployee : IEmployee  
83  {  
84      public string GetProjectDetails(int employeeId)  
85      {  
86          //code here - specific project details  
87      }  
88  
89      public string GetEmployeeDetails(int employeeId)  
90      {  
91          throw new System.NotImplementedException();  
92      }  
93  }
```

Interface Segregation Principle

How to solve this?



Interface Segregation Principle

```
106  public interface IEmployee  
107  {  
108      string GetEmployeeDetails(int employeeId);  
109  }  
110
```



You need to
create two
interfaces

```
111  public interface IProject  
112  {  
113      string GetProjectDetails(int employeeId);  
114  }
```

Interface Segregation Principle

```
116 public class CasualEmployee : IEmployee, IProject  
117 {  
118     public string GetEmployeeDetails(int employeeId)  
119     {  
120         //code here - specific casual employee details  
121     }  
122  
123     public string GetProjectDetails(int employeeId)  
124     {  
125         //code here - specific contractual employee details  
126     }  
127 }
```

```
129 public class ContractualEmployee : IProject  
130 {  
131     public string GetProjectDetails(int employeeId)  
132     {  
133         //code here - specific project details  
134     }  
135 }
```

Dependency Inversion Principle

“High-level modules should not depend on low-level modules. Both should depend on abstractions.”

“Abstractions should not depend upon details. Details should depend upon abstractions.”

Dependency Inversion Principle

```
175 public class Email  
176 {  
177     public void SendEmail()  
178     {  
179         // code to send mail  
180     }  
181 }  
182
```

```
183 public class Notification  
184 {  
185     private Email _email;  
186     public Notification()  
187     {  
188         _email = new Email();  
189     }  
190  
191     public void PromotionalNotification()  
192     {  
193         _email.SendEmail();  
194     }  
195 }  
196
```

And if you need to send a notification by SMS?
You need to change this.

Dependency Inversion Principle

```
199  public interface IMessenger  
200  {  
201      void SendMessage();  
202  }
```

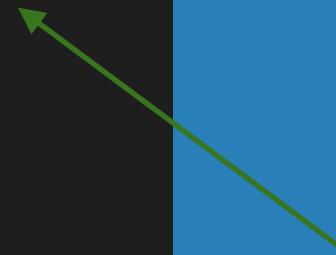
So, I create an interface and now?

```
204  public class Email : IMessenger  
205  {  
206      public void SendMessage()  
207      {  
208          // code to send email  
209      }  
210  }
```

```
212  public class SMS : IMessenger  
213  {  
214      public void SendMessage()  
215      {  
216          // code to send SMS  
217      }  
218  }
```

Dependency Inversion Principle

```
220  public class Notification
221  {
222      private IMessenger _iMessenger;
223      public Notification()
224      {
225          _ iMessenger = new Email();
226      }
227      public void DoNotify()
228      {
229          _ iMessenger.SendMessage();
230      }
231  }
```



Dependency Inversion Principle

Constructor injection:

```
235  public class Notification
236  {
237      private IMessenger _iMessenger;
238      public Notification(IMessenger pMessenger)
239      {
240          _iMessenger = pMessenger;
241      }
242      public void DoNotify()
243      {
244          _iMessenger.SendMessage();
245      }
246  }
```

Dependency Inversion Principle

Property injection:

```
248     public class Notification
249     {
250         private IMessenger _iMessenger;
251
252         public IMessenger MessageService
253         {
254             private get;
255             set
256             {
257                 _iMessenger = value;
258             }
259         }
260
261         public void DoNotify()
262         {
263             _iMessenger.SendMessage();
264         }
265     }
```

Dependency Inversion Principle

Method injection:

```
268  public class Notification
269  {
270      public void DoNotify(IMessenger pMessenger)
271      {
272          pMessenger.SendMessage();
273      }
274  }
```

Clean Coding & Refactoring

When to refactor?

Simple answer: all the time

- Refactoring should not be a phase that is executed e.g. every two weeks, but an integral part of design/programming.
- More specifically, think refactoring when ...
- **Rule of three**
- first time you do something, just do it. Second time you do something similar, you may do it with duplication. Third time something similar, refactor.

- **Refactor when you add functionality**
 - Refactor the code that is going to change before you make the change in order to deeply understand the code
 - If the change does not fit in easily, refactor the design to enable smooth addition of the new feature
- **Refactor when you need to fix a bug**
 - The fact that there was a bug states that the code is not easy to understand, otherwise the bug would not have born
- **Refactor in a code review**
 - If you are going to go through the code, why not go through the design as well, and to deeply understand the code

Code Quality Control Law

Improving code quality reduces development costs

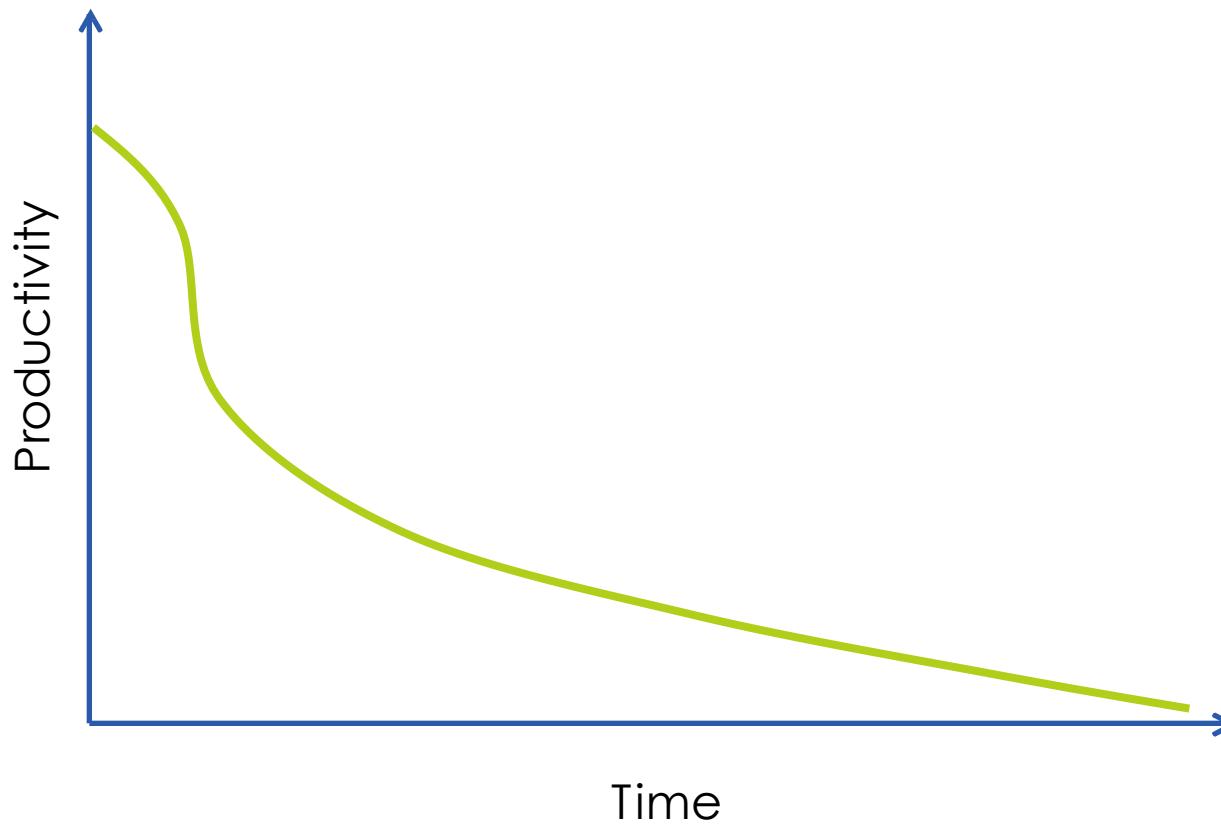
Why?

Developer average productivity is 10-50 lines per day

Writing code – 10%

Reading code, testing, debugging – 90%

Productivity vs Time



Software Development Paradox

1. Low-quality code slows down development
2. Devs reduce code quality to meet a deadline

Paradox: “Devs have no time to work rapidly”

The clue: #2 is wrong

Impossible to successfully meet a deadline messing up the code

Continuously maintain high code quality

The Boy Scout Rule

- Leave the campground cleaner than you found it

Keep Code Simple

KISS (Keep It Simple, Stupid)

Simplicity is a key goal

Components should be integrated tightly or weakly
(?)

Try to reduce influence between components
Follow “Divide and Conquer” principle

Division of Responsibilities

- System -> packages
- Package -> classes
- Class -> methods
- Method -> control statements

Low coupling on each level

Use Patterns

Don't reinvent the square wheel!

- Reduce complexity!
- Less mistakes!
- Simplify communication!

Examples: Repository, Factory, Adapter, Strategy, Observer

No Unnecessary Code

- **YAGNI (You Aren't Gonna Need It)**
- Implement things only when you really need them

Avoid Duplication

- DRY (Don't Repeat Yourself)
- Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

Object-Oriented Design

SOLID Principle

1. Single Responsibility (SRP)
2. Open-Closed (OCP)
3. Liskov Substitution (LSP)
4. Interface Segregation (ISP)
5. Dependency Inversion (DIP)

Meaningful Names

- Naming principles
- Naming conventions
- Name length
- Variable names
- Class names
- Method names

Naming Principles

Name should show your intention

```
int d; //elapsed time
```

Is it good enough?

Which name is better?

```
int elapsedTimelnDays;  
int daysSinceCreation;
```

Naming Principles

What does the following code do?

```
function getItems(items) {  
    var list1 = [];  
    for(var i = 0; i < items.length; i++) {  
        var x = items[i];  
        if(items[i]["Field"] === 12) {  
            list1.push(x);  
        }  
    }  
    return list1;  
}
```

What is about this one?

```
function getMarkedCells(cells) {  
    var result = [];  
    for(var i = 0; i < cells.length; i++) {  
        var cell = cells[i];  
        if(cell.Status === CellStatus.Marked) {  
            result.push(cell);  
        }  
    }  
    return result;  
}
```

No implementation details

```
HashSet<Account> accountHashSet;
```

Is it good enough? Why? What if we
decide to use List? Which name is better?

```
HashSet<Account> accounts;
```

Choose easy-to-pronounce name

```
class Cstmr {  
    private int prsnid;  
    private DateTime genTstmp;  
    private DateTime modTstmp;  
}
```

```
class Customer {  
    private int personId;  
    private DateTime generationTimestamp;  
    private DateTime modificationTimestamp;  
}
```

Single word for concept (avoid synonyms)

- fetch / retrieve / get controller / manager / driver

Names from patterns and algos

- RenderStrategy, JobQueue

Names from user stories

- Credit, Debit, Asset

Antonyms

- begin / end
- first / last
- min / max
- next / previous
- old / new
- opened / closed
- source / target
- up / down

Naming Convention

The most important is to follow to the convention

Example:

- CONSTANT
- ClassName
- localVariable
- _privateMember

Popular Naming Conventions

- **Camel case:** camelCaseExample
- **Snake case:** snake_case_example
- **Kebab case:** kebab-case-example
- **Pascal case/Uppercase camel case:** PascalCaseExample

Name Length

What is the optimal length for a name?

x → maximumNumberOfPointsInEachLine

short / long name pros & cons

short names – lacks of information

long names – harder to read and write

10-16 characters is the optimal length

if < 10 ask: is meaningful enough?

if > 16 ask: isn't verbose?

Loop Variable Names

- **Simple Loop**

```
for(int i = 0; i < data.length; i++) {  
    data[i] = 0;  
}
```

- **Complex loop**

```
for(int teamIndex = 0; teamIndex < teamCount; teamIndex++) {  
    for(int playerIndex = 0; playerIndex < count; playerIndex++) {  
        playerStore[teamIndex][playerIndex] = 0;  
    }  
}
```

- **Variable used outside the loop**

```
int recordCount = 0;  
while(moreRecords()) {  
    data[recordCount] = getNextRecord(); recordCount++;  
}  
// ... recordCount is accessed here
```

Temporary Variable Names

Avoid pointless names

```
temp = sqrt(b^2 - 4*a*c);  
root[0] = (--b + temp) / (2*a);  
root[1] = (--b - temp) / (2*a);
```

How to improve?

```
discriminant = sqrt(b^2 - 4*a*c);  
root[0] = (--b + discriminant) / (2*a);  
root[1] = (--b - discriminant) / (2*a);
```

Boolean Variables and Methods

Use verbs **is**, **has**, **can** for booleans

- isSupported
- hasNext
- canExecute

Class Names

Use nouns and noun combinations for classes

- Client
- Window
- UserAccount
- FileLoader

Method Names

Use verbs and verb phrases for methods

- Save
- Close
- GetAccount
- LoadFile

Variables - Declaration

Declare all variables

- Turn off implicit variable declaration

JavaScript:

“use strict”

VB:

Option Explicit On

php:

```
error_reporting(E_STRICT);
```

Variables - initialization

Initialize variables at declaration

```
var total = 0;
```

or at first occurrence in code

```
Dim total As Double
```

```
' another code
```

```
total = 0.0
```

```
' code using total
```

Variables – lifetime and scope

Variables lifetime and scope should be minimal

Why?

Does code follow the principle?

```
function summarizeData() {  
  var oldData = getOldData();  
  var totalOldData = summarize(oldData);  
  printData(totalOldData);  
  saveData(totalOldData);  
  
  var newData = getNewData();  
  var totalNewData = summarize(newData);  
  printData(totalNewData);  
  saveData(totalNewData);  
}
```

Code might look like

```
function summarizeData() {  
  var oldData = getOldData();  
  var newData = getNewData();  
  
  var totalOldData = summarize(oldData);  
  var totalNewData = summarize(newData);  
  
  printData(totalOldData);  
  printData(totalNewData);  
  
  saveData(totalOldData);  
  saveData(totalNewData);  
}
```

Single purpose principle

```
int value = getValue();  
int result = calcResult(value);
```

//some code

```
value = val1;  
val1 = val2;           →  
val2 = value;
```

```
var swap = val1;  
val1 = val2;  
val2 = swap;
```

- Avoid global variables
- Prefer the most restrictive access level

private → public

Methods - Purposes

Simplify code

- Improve readability
- Documentation
- Eliminating duplication
- Inheritance support

defines abstraction
most popular
method overriding

Methods - Principles

Method should be small

How many lines?

not more than **20** lines and better even less

Example:

```
public ReportContainer BuildReport(List<Order> orders) {  
    var report = new ReportContainer();  
    report.Lines = BuildReportLines(orders);  
    report.Total = BuildReportTotal(report.Lines);  
    return report;  
}
```

Method should have

- blocks (if, else, for, while) 1-3 lines long
- indent level not more than 2

```
private IList<ReportLine> BuildReportLines(IList<Order> orders) {  
    IList<ReportLine> result = new List<ReportLine>();  
    foreach (Order order in orders) {  
        ReportLine reportLine = BuildReportLine(order);  
        if (reportLine.Visible)  
            result.Add(reportLine);  
    }  
    return result;  
}
```

Single Thing Rule

Function should do one thing, do it only, and do it well

Examples:

- BuildReport – makes report
- BuildReportLines – prepares report lines
- BuildReportTotal – summarizes report

Criteria: function cannot be divided into sections

Command-Query Separation (CQS)

Method performs an action **OR** returns data

Does code follow the principle?

```
if (SetAttribute("username", "Admin")) {  
    // code  
}
```

How to fix?

```
if (HasAttribute("username")) {  
    SetAttribute("username", "Admin");  
    // code  
}
```

Class - Purposes

- Simplify code
- Improve readability
- Limit the scope of changes
- Hide implementation details (encapsulation)
- Allow to prepare generic solution (abstraction)
- Eliminate duplication (delegation & inheritance)
- Provide extensibility (inheritance)
- Allow to work with real domain entities

Principles

- Encapsulation

Hide as much as possible

- Compactness

Reduce class responsibilities (SRP)

- Abstraction

Program to an interface, not an implementation

- High Cohesion

Fields and methods should belong together

Class - Inheritance

Avoid following situations:

- Parent class has only one child
- Child overrides method leaving it empty
- Too many hierarchy levels better not more than 3 levels

Commenting Principles

- Comments do not redress bad code!

clean code much better than bad code with comments!

- Do not comment tricky code – rewrite it!

tricky code = bad code!

- Try to explain with code

```
// is employee eligible for bonus
if ((employee.Schedule == HOURLY_SCHEDULE) &&
    employee.Age > 65) { ... }
```

Better

```
if (employee.IsEligibleForBonus()) { ... }
```

Bad Comments

Redundant comments

- obvious and repeating code

Commented code!

- remove unnecessary code,
VCS is for the rescue

Good Comments

Legal comments
license, copyright, author

Explanations
provide info about inevitable tricks

```
// NOTE: postpone execution to refresh UI
setTimeout(function() {
    // code
});
```

TODOs

notes for future to avoid “broken windows”

```
// TODO: get rid of tricky workaround
function badFunction() {
    // tricky workaround here
};
```

Comment says “why” not “how”

How

```
// if flag equals to zero  
if (accountFlag == 0)
```

Why

```
// if new account created  
if (accountFlag == 0)
```

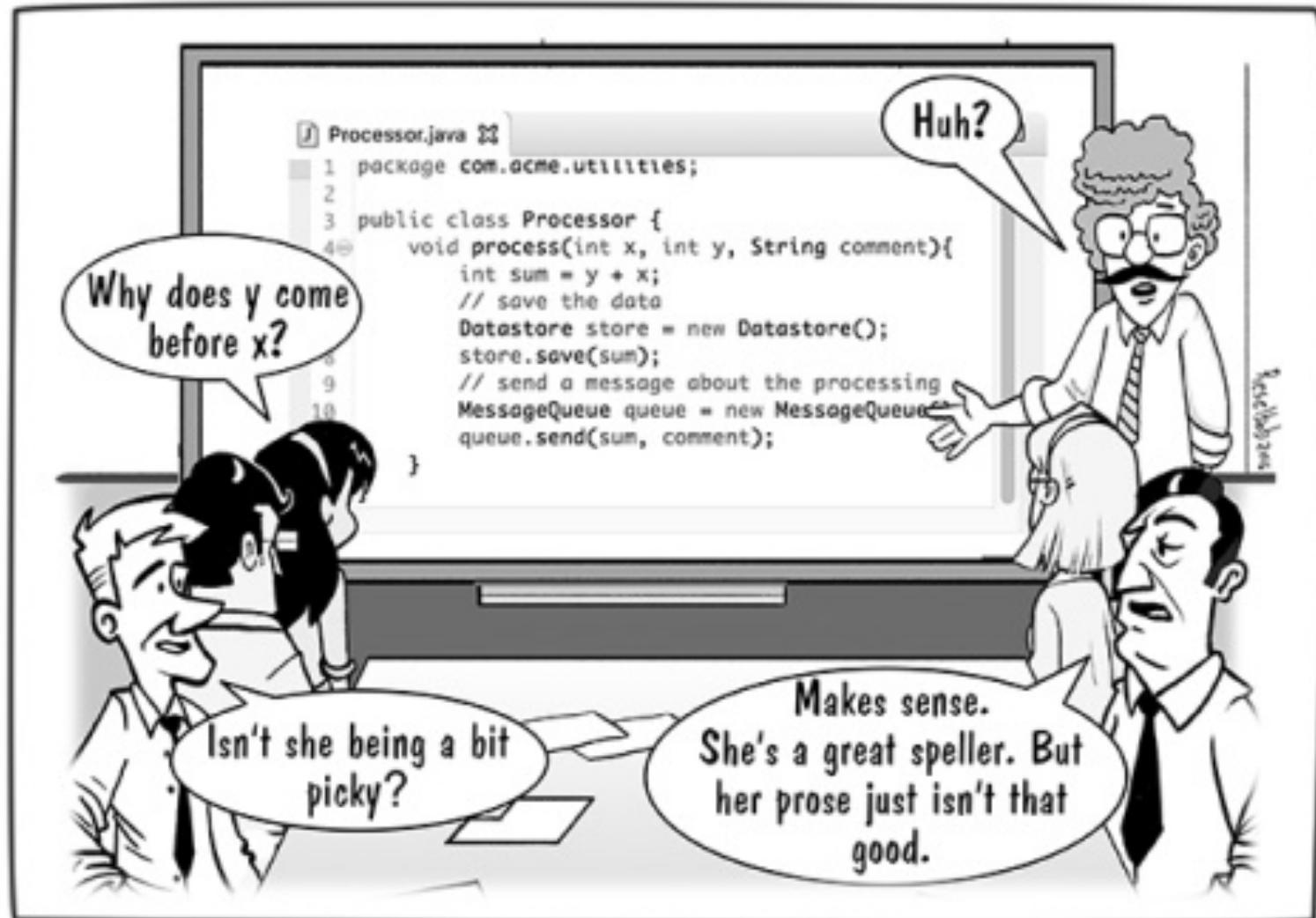
Code instead of comment

```
if (accountType == AccountType.NewAccount)
```

Refactoring class work – by students

- Please form groups and go through the sample git repo and attached code to refactor the code
- We have to consider each possible opportunities
 - Naming principle
 - Refactor to patterns
 - Identify duplicate codes

Code review & process



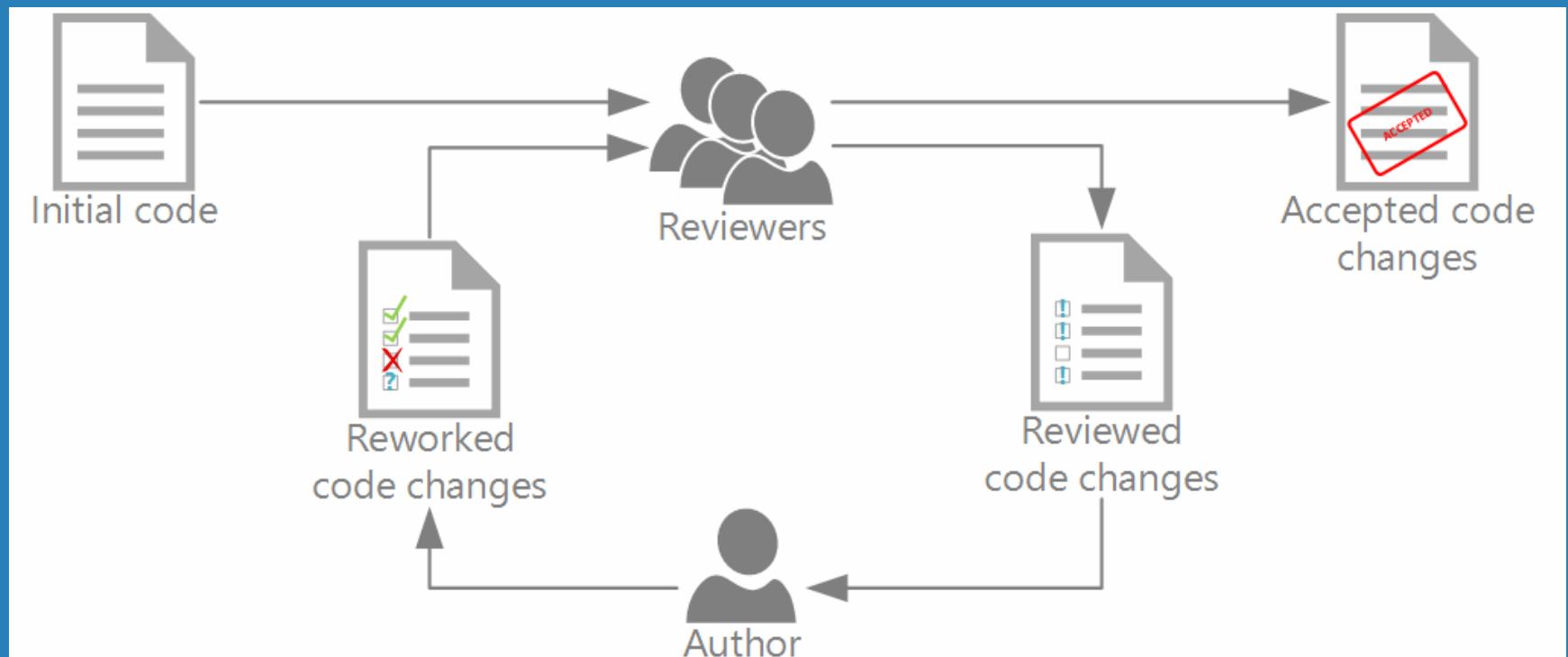
How come code review is agile?

- Code review ensures
 - Within the team feedback
 - Quality code
 - Best practices
 - Great deliveries
 - Possible refactoring

When to do code review?

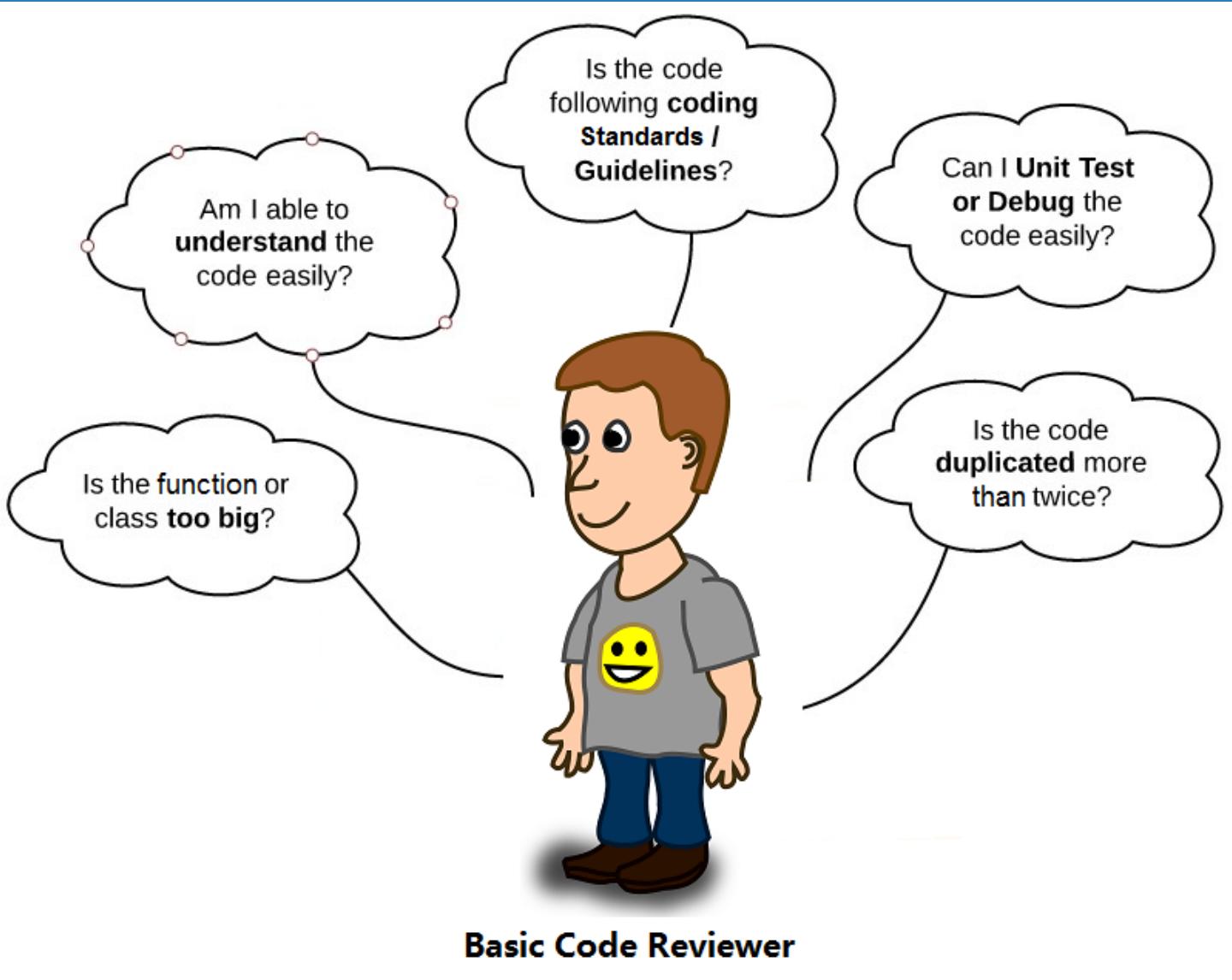
- Before commit?
- After commit?
- After release?

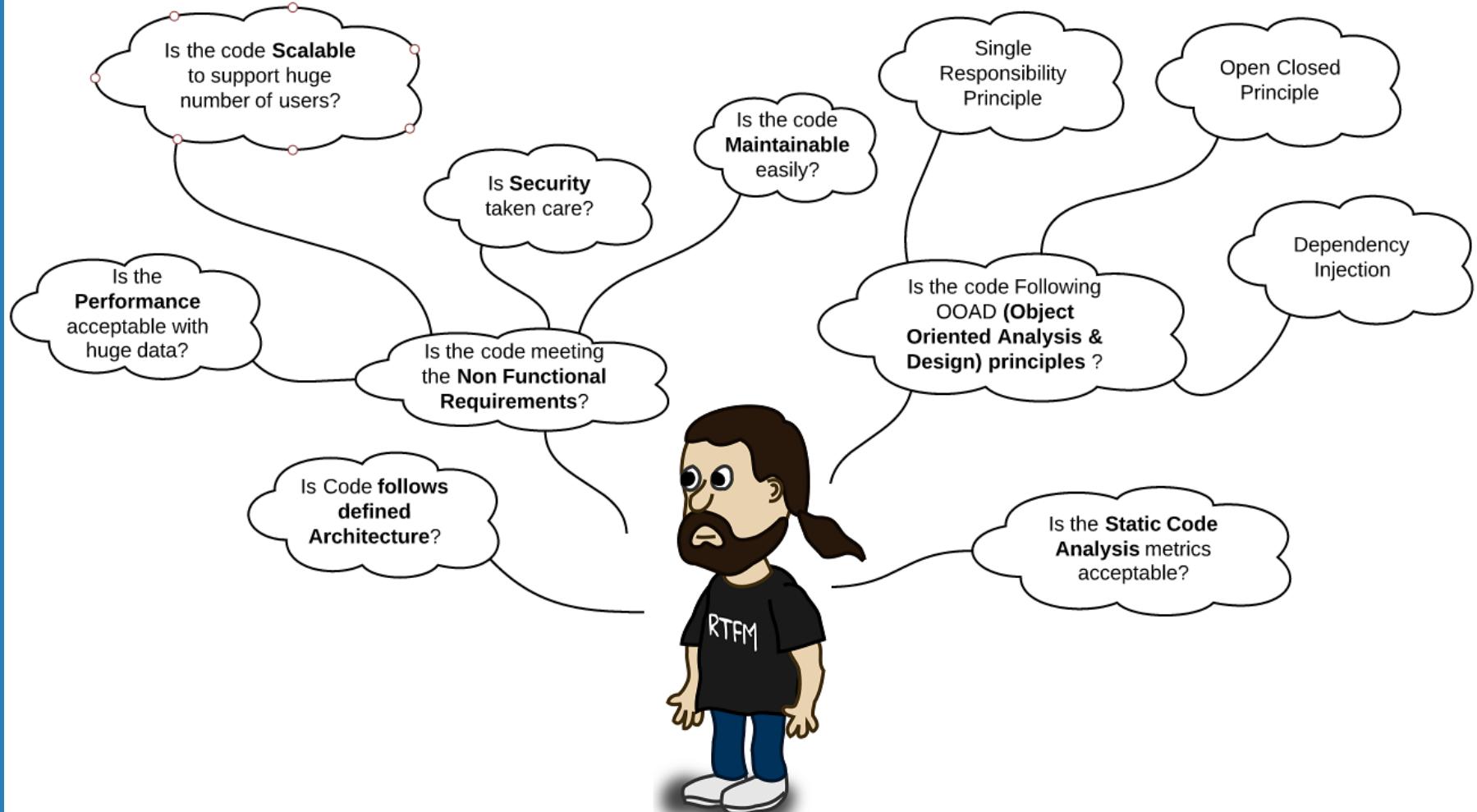
Ideal flow



Be constructive in review

- Review should not be a blaming place
- Team members must feel comfortable to do peer review
- Review should focus on best design and practices
- Review can be very detail like finding spelling mistakes and not following coding guidelines
- Reviewer must also suggest how to do things better





Expert Code Reviewer

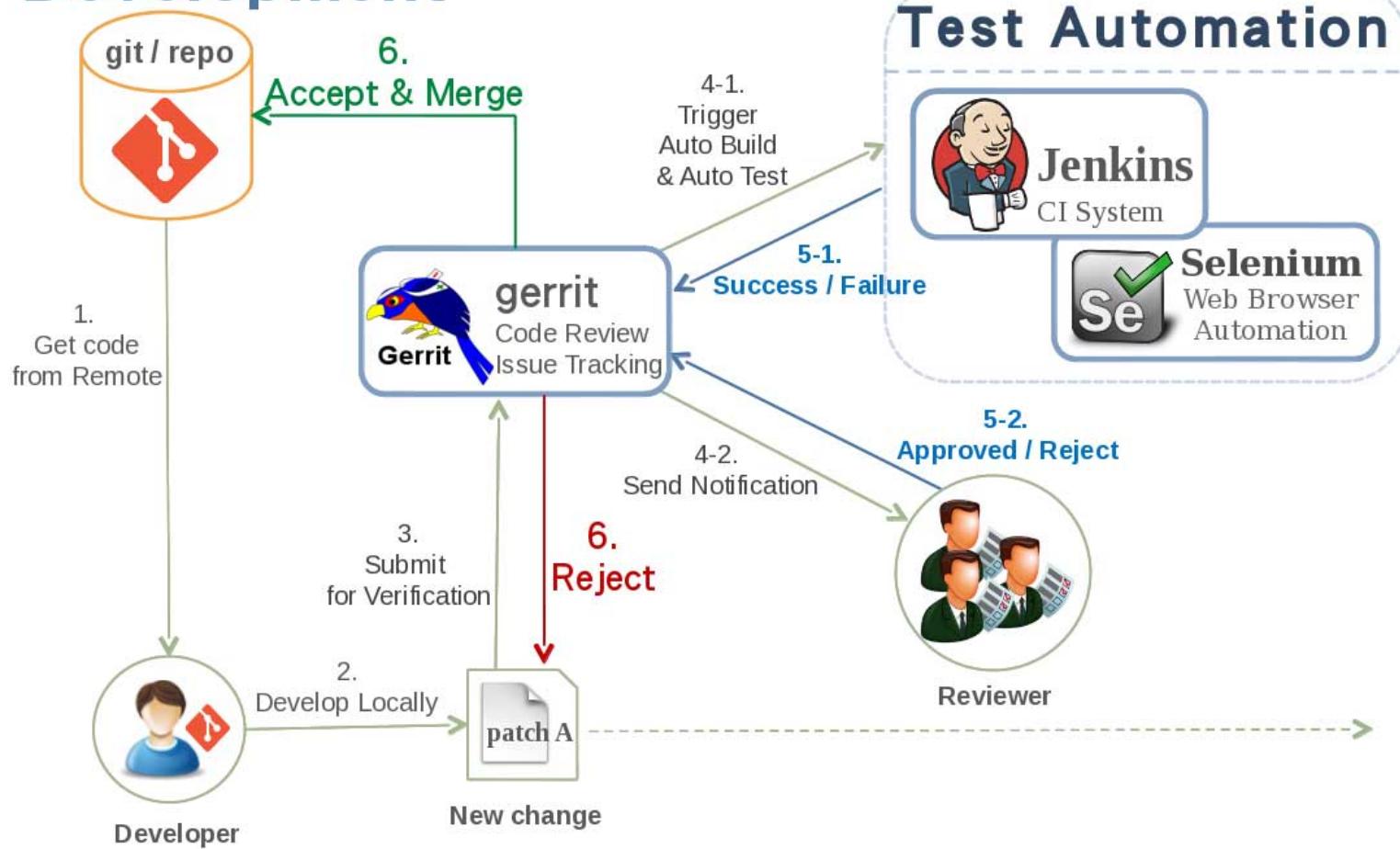
Code review check list

<i>Code Review Checklist</i>	
<input checked="" type="checkbox"/>	Coding standards
<input type="checkbox"/>	Coding Best practices
<input checked="" type="checkbox"/>	Non Functional Requirements
<input checked="" type="checkbox"/>	OOAD Principles
<input checked="" type="checkbox"/>	Static Code Analysis Metrics
<input type="checkbox"/>

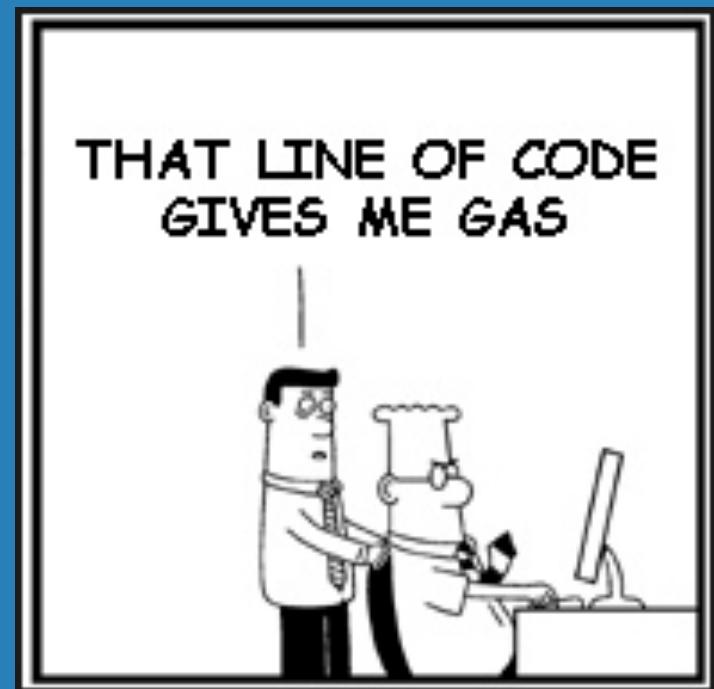
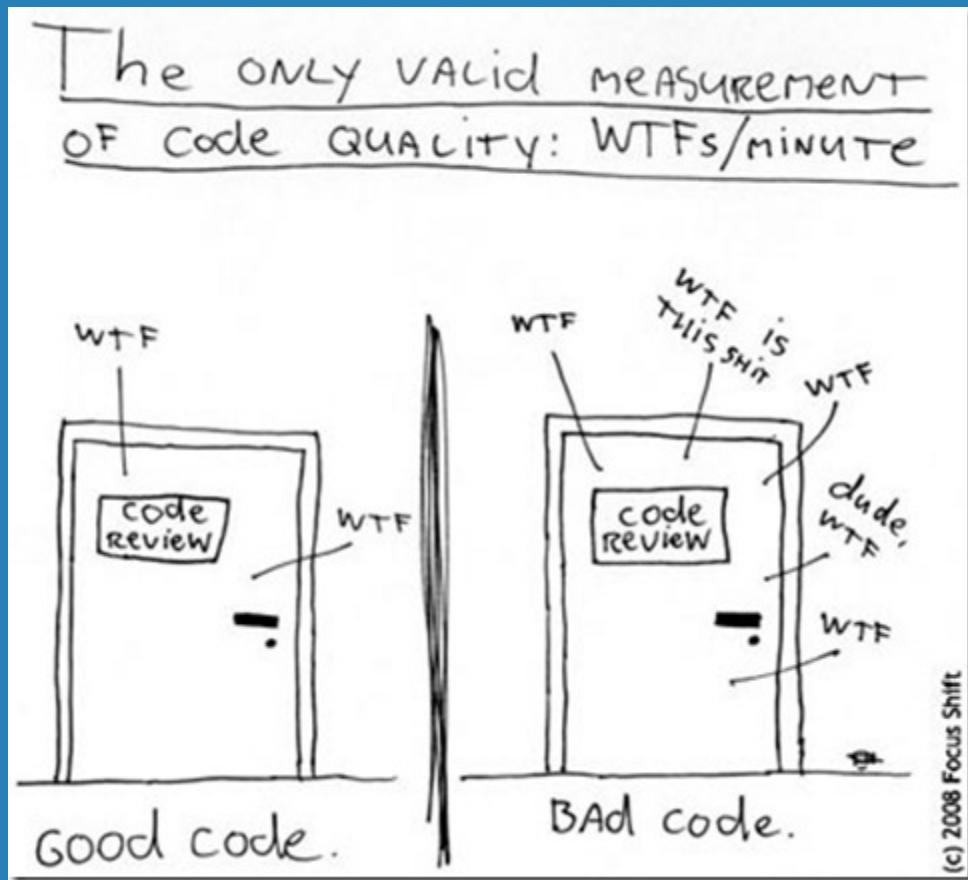
Code review tools

- Github and other SCM has code review option
- Gerrit, sonarcube , review board, rhode code
- Wiki for more detail feedback and coding guidelines

Web Application Test-driven Development

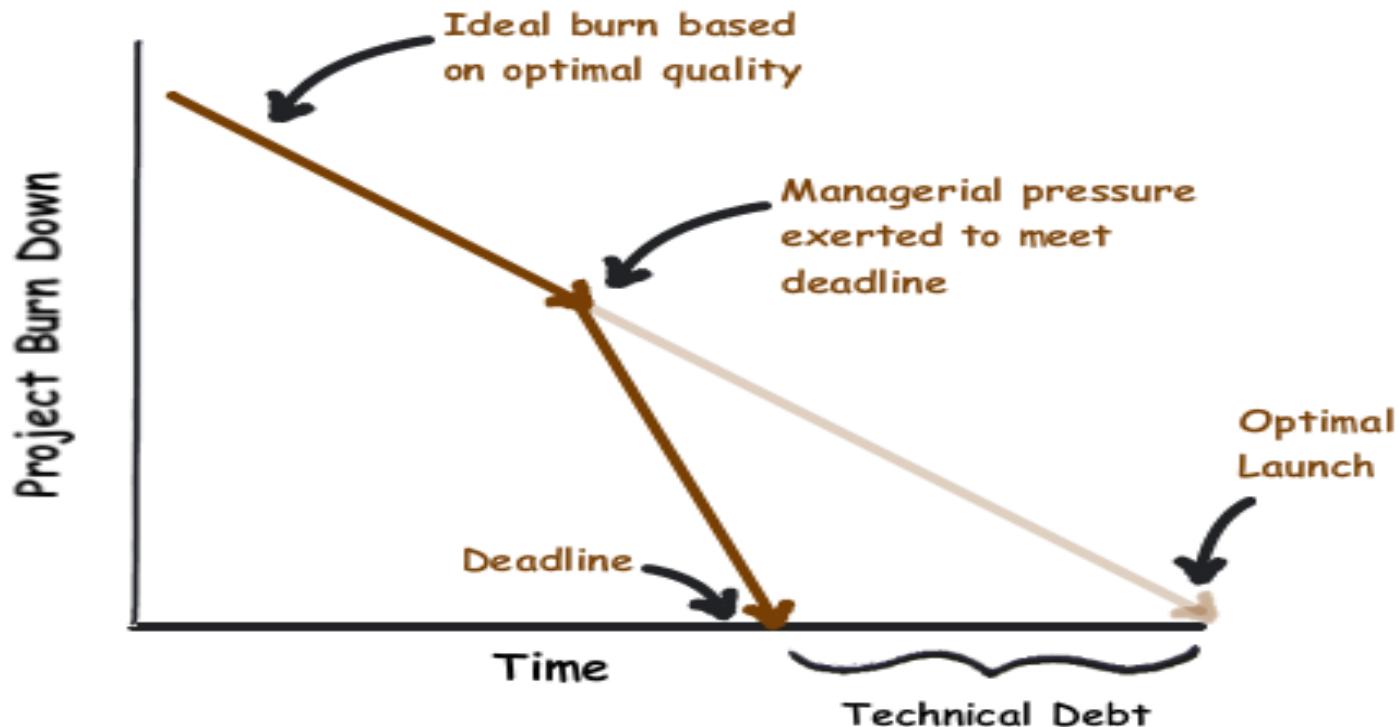


Not to do this



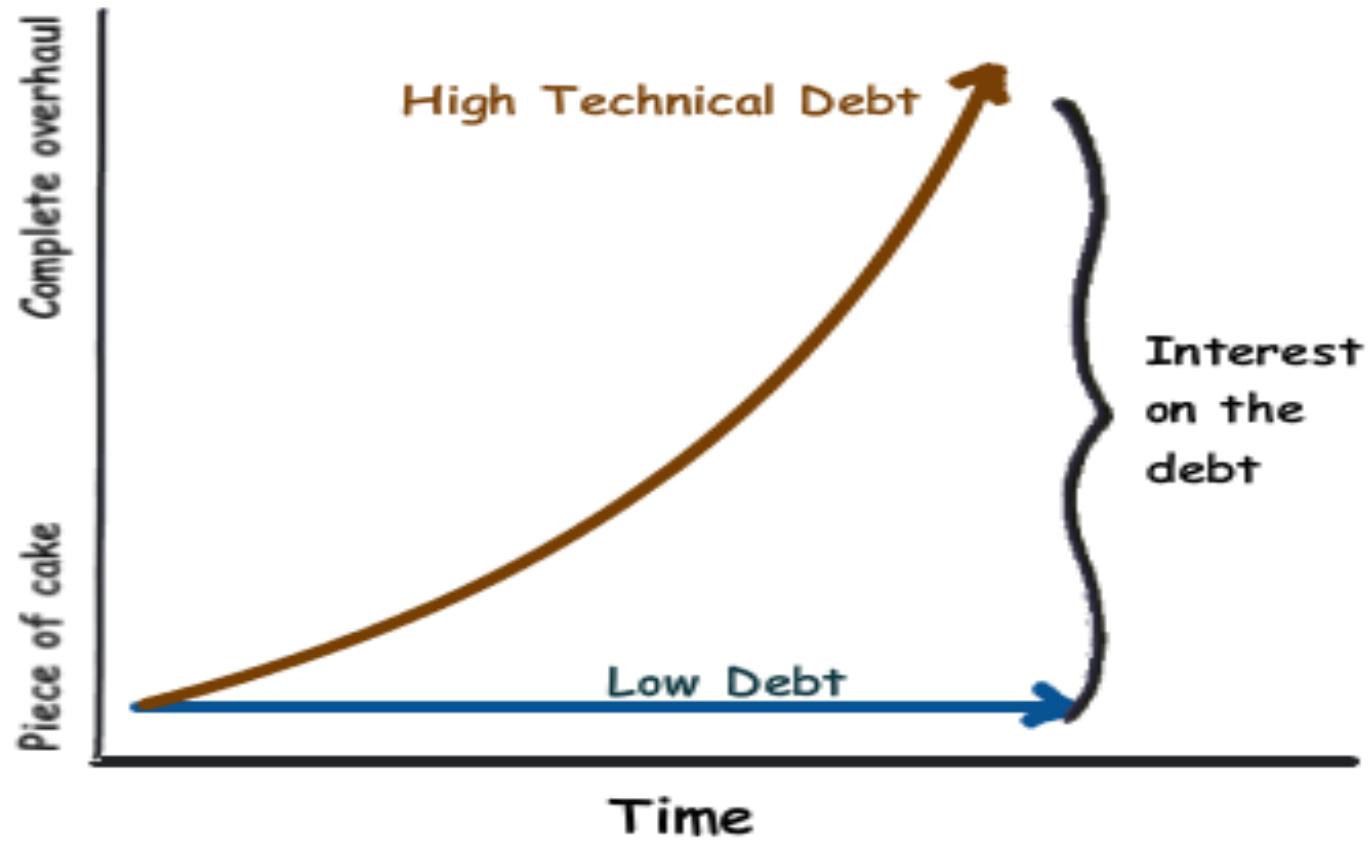
Technical Debt

Technical Debt

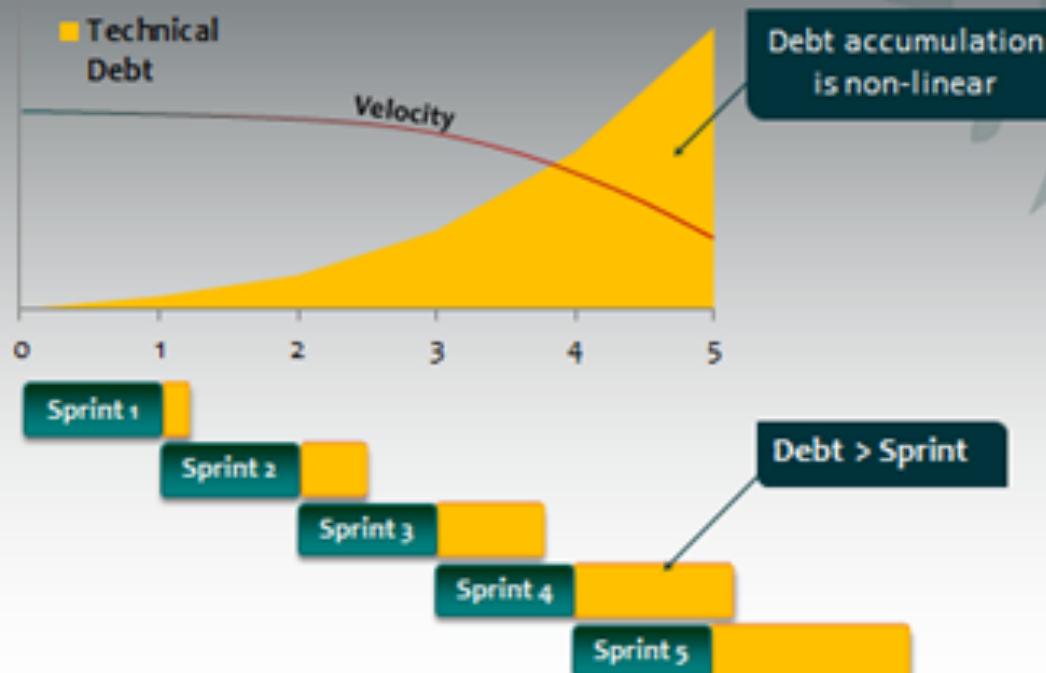


Technical Debt

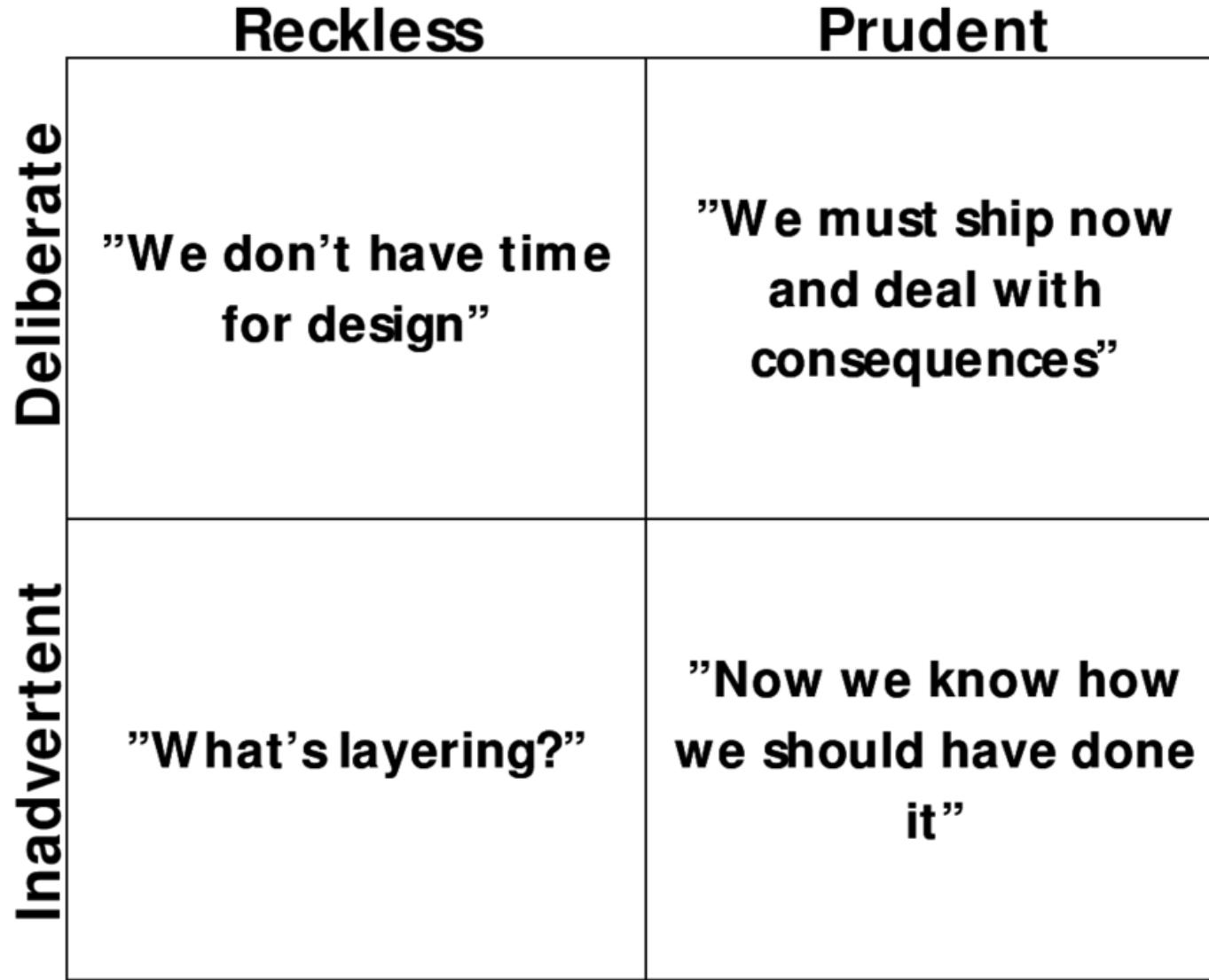
How hard to add features?



Mounting Technical Debt



Martin Fowler's debt quadrant



DELIBERATE

**Selfish
Debt**

**Acknowledged
Debt**

INADVERTENT

**Ignorance
Debt**

**Immature
Debt**

RECKLESS

PRUDENT

Test driven development - TDD

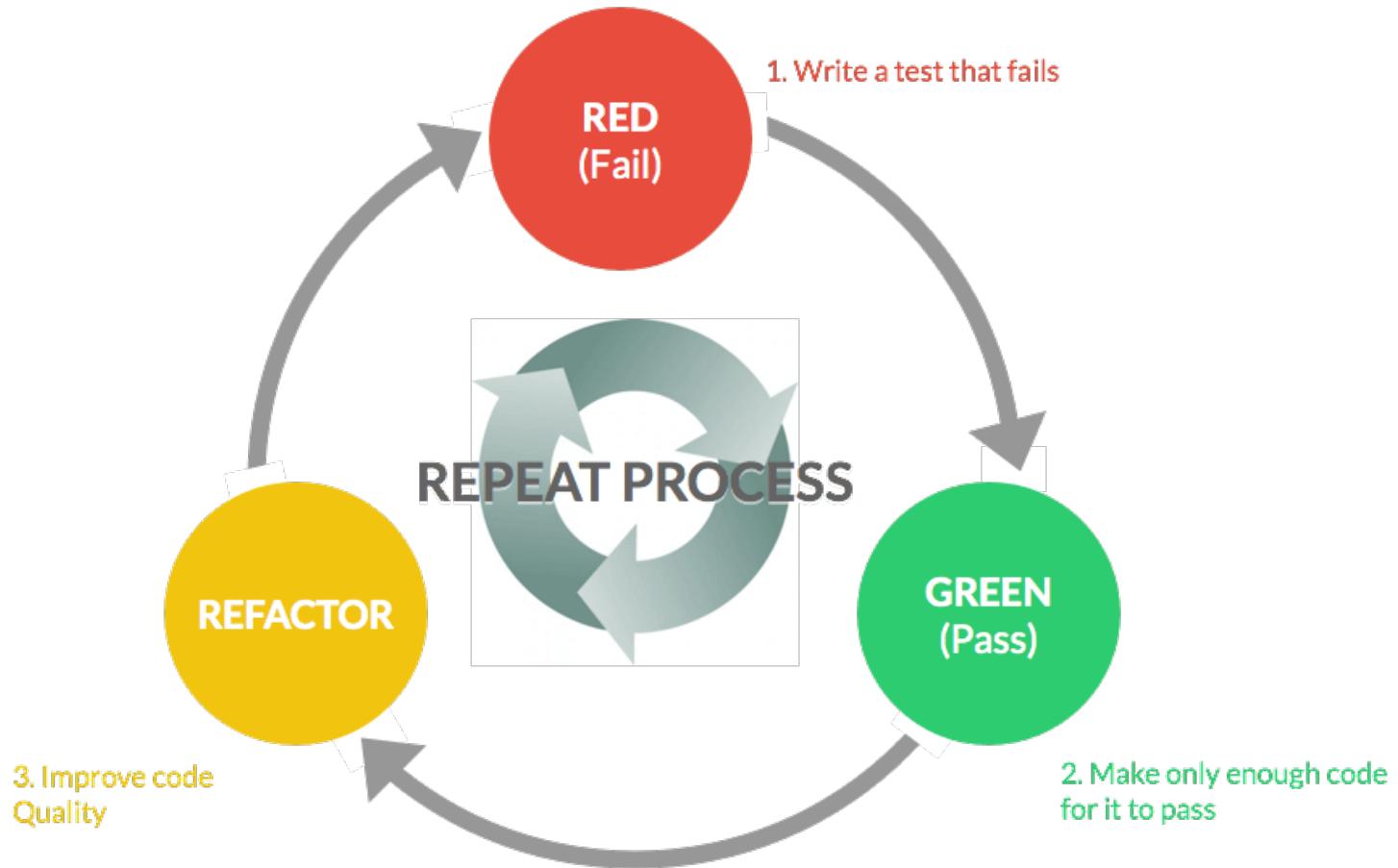
A PERSPECTIVE ON TDD

- TDD is Mostly about Design
- Gives Confidence
- Enables Change
- Is Automated
- Validates Your Design
- Is Documentation By Example
 - As Well As an Up To Date, “Live”, And Executable Specification
- Provides Rapid Feedback
 - Quality of Design
 - Quality of Implementation
- Forces Constant Integration
- Requires More Discipline
- Brings Professionalism to Software Development
- Isn’t The Only Testing You’ll Need To Do
- Developers Write Tests

Why TDD?

- “Clean code that works,”—Ron Jeffries
- Predictable, no worry of mounting bugs
- Better quality of code
- Makes you dependable
- You are confident... feel good

TDD Mantra



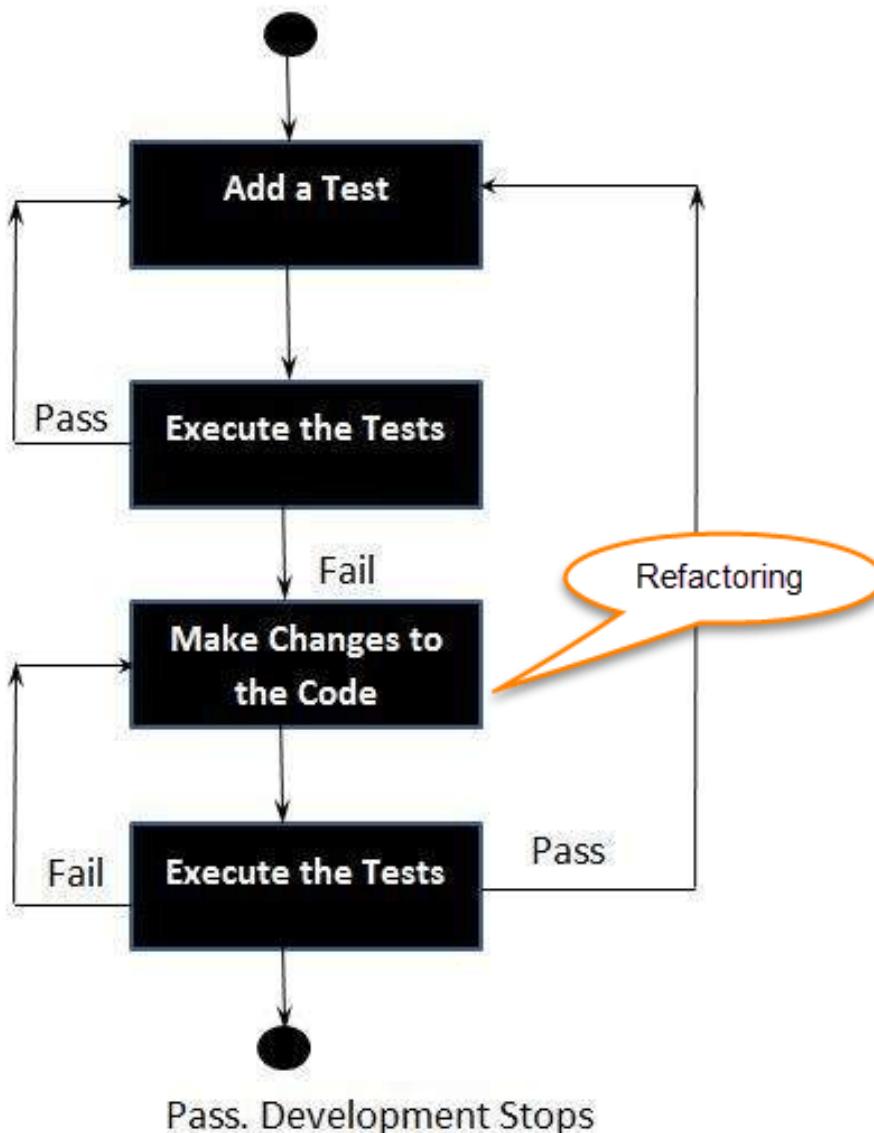
THE THREE LAWS OF TDD

1: You May Not Write Production Code Until You Have Written A Failing Unit (or Acceptance) Test

2: You May Not Write More Of A Unit (or Acceptance) Test Than Is Sufficient To Fail, And Compiling Is Failing

3: You May Not Write More Production Code Than Is Sufficient To Pass The Currently Failing Test

TDD Cycle

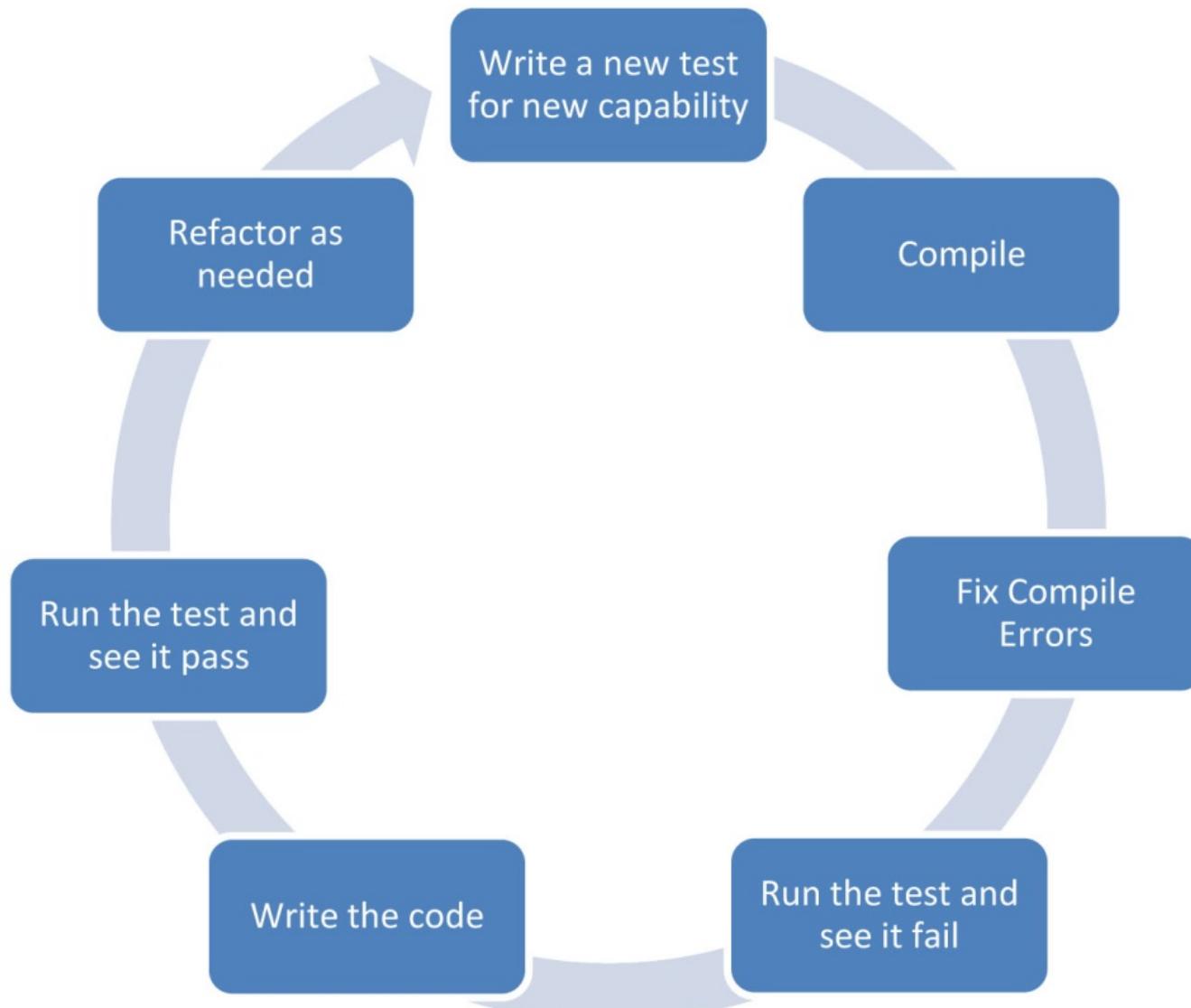


FUNDAMENTAL PRINCIPLES

- Think About What You Are Trying To Do
- Follow The TDD Cycle And The Three Laws
- Never Write New Functionality Without A Failing Test
- Continually Make Small, Incremental Changes
- Keep The System Running At All Times
 - No One Can Make A Change That Breaks The System
 - Failures Must Be Address Immediately

In reality, unwavering adherence the cycle and laws can sometimes be very hard. In my opinion, it doesn't always have to be all or nothing (especially when you are just learning). But I've found that when I don't write tests first, I usually regret it afterwards. Also, TDD isn't applicable for every single scenario or line of code.

TDD steps



WHAT ABOUT ARCHITECTURE AND DESIGN?

- TDD Does Not Replace Architecture Or Design
- You Need To Do Up Front Design And Have A Vision
- But, You Should Not Do “Big Up Front Design”
- Defer Architectural Decisions As Long As Is “Responsibly” Possible
- TDD Will Inform And Validate (or Invalidate) Your Design Decisions
- TDD Will Almost Undoubtedly Uncover Weaknesses And Flaws In Your Design...Listen To Them

CATEGORIES OF TESTS

- Unit
- Integration
- Acceptance
 - Can Encompass “system” Or “functional” Tests
 - End-to-end Is Desirable When Feasible
- Learning Or Exploratory

BEST PRACTICES: F.I.R.S.T

- Fast
- Independent
- Repeatable
- Self-validating
- Timely

Right B.I.C.E.P.

- It's essential to understand what's important to test.
- Right BICEP helps you ask the right questions about what to test.

Right B.I.C.E.P.

- Guidelines of some areas that are important to test:
 - Right - Are the results right?
 - B - Are all the **boundary** conditions CORRECT?
 - I - Can you check **inverse** relationships?
 - C - Can you **cross-check** results using other means?
 - E - Can you force **error** conditions to happen?
 - P - Are **performance** characteristics within bounds?

[Right] – B.I.C.E.P

Key question : If the code ran correctly, how would the developer know?

- If this question cannot be answered satisfactorily, then writing the code or the test may be a complete waste of time.

Does that mean code cannot be written until all the requirements are in?

- Nothing stops you from proceeding without answers to every last question.
- Use your best judgment to make a choice about how to code things, and later refine the code when answers do come.

The definition of correct may change over the lifetime of the code in question, but at any point, developer should be able to prove that it's doing what he/she thinks it should be doing.

B - Are all the boundary conditions CORRECT?

Identifying boundary conditions is one of the most valuable parts of unit testing, because this is where most bugs generally live - at the edges.

```
public void testOrder ()  
{  
    assertEquals(9, Largest.largest(new int[] { 9, 8, 7 }));  
    assertEquals(9, Largest.largest(new int[] { 8, 9, 7 }));  
    assertEquals(9, Largest.largest(new int[] { 7, 8, 9 }));  
}  
  
public void testDups ()  
{  
    assertEquals(9, Largest.largest(new int[] { 9, 7, 9, 8 }));  
}  
  
public void testOne ()  
{  
    assertEquals(1, Largest.largest(new int[] { 1 }));  
}  
  
public void testNegative ()  
{  
    int[] negList = new int[] { -9, -8, -7 };  
    assertEquals(-7, Largest.largest(negList));  
}  
  
public void testEmpty ()  
{  
    try  
    {  
        Largest.largest(new int[] {});  
        fail("Should have thrown an exception");  
    }  
    catch (RuntimeException e)  
    {  
        assertTrue(true);  
    }  
}
```

Example Boundaries

- Totally bogus or inconsistent input values, such as a file name of "!*W:Xn&Gi/w>g/h#WQ@".
- Badly formatted data, such as an e-mail address without a top-level domain ("fred@foobar.").
- Computations that can result in numeric overflow.
- Empty or missing values (such as 0, 0:0, "", or null).
- Values far in excess of reasonable expectations, such as a person's age of 150 years.
- Duplicates in lists that shouldn't have duplicates e.g. class attendance.
- Ordered lists that aren't, and vice-versa. Try handing a pre-sorted list to a sort algorithm, for instance, or even a reverse-sorted list.
- Things that arrive out of order, or happen out of expected order, such as trying to print a document before logging in.

Remember Boundary Conditions with C.O.R.R.E.C.T

- Conformance - Does the value conform to an expected format?
- Ordering - Is the set of values ordered or unordered as appropriate?
- Range - Is the value within reasonable minimum and maximum values?
- Reference - Does the code reference anything external that isn't under direct control of the code itself?
- Existence - Does the value exist (e.g., is non-null, nonzero, present in a set, etc.)?
- Cardinality - Are there exactly enough values?
- Time (absolute and relative) - Is everything happening in order? At the right time? In time?

I. Check Inverse Relationships

- Some methods can be checked by applying their logical inverse.
- e.g. check a method that calculates a square root by squaring the result, and testing that it is tolerably close to the original number.
- or – verify division by performing multiplication.
- or - check that some data was successfully inserted into a database by then searching for it.

```
public void testSquareRootUsingInverse()
{
    double x = mySquareRoot(4.0);
    assertEquals(4.0, x * x, 0.0001);
}
```

C. Cross-check Using Other Means

- Where possible, use a different source for the inverse test (bug could be in original and in inverse).
- Usually there is more than one way to calculate some quantity;
 - pick one algorithm over the others because it performs better, or has other desirable characteristics - use that one in production.
 - use one of the other versions to crosscheck our results in the test system.
- Especially helpful when there's a proven, known way of accomplishing the task that happens to be too slow or too complex to use in production code.

```
public void testSquareRootUsingStd()
{
    double number = 3880900.0;
    double root1 = mySquareRoot(number);
    double root2 = Math.sqrt(number);
    assertEquals(root2, root1, 0.0001);
}
```

C. Cross-check Using Other Means

Another example - a library database system:

The number of copies of a particular book should always balance:

e.g. number of copies that are checked out + number of copies sitting on the shelves should always equal the total number of copies.

These are separate pieces of data, and they may even be reported by objects of different classes, but they still have to agree, and so can be used to cross-check one another.

E - Can you force error conditions to happen?

- In the real world, errors happen:
 - disks fill up,
 - network lines drop,
 - e-mail goes down,
 - and programs crash.
- Developers should test that code handles many of these real world problems by forcing errors to occur.
- That's easy enough to do with invalid parameters and the like, but to simulate specific network errors without unplugging any cables takes some special techniques

E. Force Error Conditions

Some scenarios you might consider when writing tests:

- Running out of memory.
- Running out of disk space.
- Issues with wall-clock time.
- Network availability and errors.
- System load.
- Limited color palette.
- Very high or very low video resolution.

P. Performance Characteristics

- Performance characteristics - does not necessarily mean measuring performance itself - but rather performance trends as input sizes grow, as problems become more complex.
- The approach is not to objectively measure performance, but to incorporate general tests just to make sure that the performance curve remains stable.

Performance example

- A filter that identifies web sites to block.
- The code may work well with a few dozen sample sites, but will it work as well with 10,000? 100,000?
- This test may take 6-7 seconds to run, so may run only nightly.
- See JUnitPerf for tools to simplify unit-level performance measurement

```
public void testURLFilter()
{
    Timer timer = new Timer();
    String naughty_url = "http://wwwxxxxxxxxx.com";

    // First, check a bad URL against a small list
    URLFilter filter = new URLFilter(small_list);
    timer.start();
    filter.check(naughty_url);
    timer.end();
    assertTrue(timer.elapsedTime() < 1.0);

    // Next, check a bad URL against a big list
    URLFilter f = new URLFilter(big_list);
    timer.start();
    filter.check(naughty_url);
    timer.end();
    assertTrue(timer.elapsedTime() < 2.0);

    // Finally, check a bad URL against a huge list
    URLFilter f = new URLFilter(huge_list);
    timer.start();
    filter.check(naughty_url);
    timer.end();
    assertTrue(timer.elapsedTime() < 3.0);
}
```

P. Performance Characteristics

A better use of a unit-level performance measurement is to provide baseline information for purposes of making changes.

- Suppose you suspect that a Java 8 lambda-oriented solution is suboptimal. You'd like to replace it with a more classic solution to see if the performance improves.

Approach:

1. Before making optimizations, first write a performance "test" that simply captures the current elapsed time as a baseline. (Run it a few times and grab the average.)
2. Change the code, run the performance test again, and compare results. You're seeking relative improvement—the actual numbers themselves don't matter.

A-TRIP

- Automatic: Running the test and checking the results should be automatic
- Thorough: Well-tested methods may have 4-5 asserts.
- Repeatable: Run over and over again, in any order produce the same results.
- Independent: Keep tests tight, focused, test only one thing at once.
- Professional: Write real code, refactor.

TDD CHALLENGES

- Discipline Is Required
- Developers Are Often Stubborn And Lazy
- It Is Hard For Developers To Drop Bad Habits
- Management Doesn't Often Understand
“Internal Quality”

The Three Laws of TDD

- You are not allowed to write any production code unless it is to make a failing unit test pass.
- You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
- You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

principles that enable testability and ease refactoring

- Using Design patterns
- SOLID principles
- Test Driven Development

Measuring Test Effectiveness

Test Coverage: Test coverage is one of the most important effectiveness measurement for test managers. The metric is determined by comparing the number of successful tests to the total number of tests that need to be completed for a given project. If you have a poor test coverage is poor, then that's something you will need to address immediately.

Defect Removal efficiency: This metric is an indicator of the filtering ability of quality control (Testing) and quality assurance (Process). The target is related to the severity of the defect. The objective is to determine the efficiency of defects found by internal review and testing process before releasing the product to the customer. Testing teams should be aiming for a 100 percent removal rate.

Measuring Test Effectiveness

Defect Detection Percentage (DDP) it gives a measure of the testing effectiveness. It is calculated as a ratio of defects found prior to release and after release by customers.

Calculation

To be able to calculate that metric, it is important that in your defect tracking system you track:

affected version, version of software in which this defect was found.
release date, date when version was released

You can also calculate DDP from sub-metrics:

Number of escaped defects

Number of defects (at the moment of software version release)

Measuring Test Effectiveness - DDP

DDP = Number of defects at the moment of software version release /
Number of defects at the moment of software release + escaped
defects found.

As DDP ratio is changing over time as more defects are found by customers working with the version best visualization is using a line chart that starts with 100% at moment of software version release and a line representing a trend of how fast DDP is declining.

Example

For example, suppose that 90 defects were found during QA/testing stage and 20 defects were found by customers after the release. The DDP would be calculated as $90 \div (90 + 20) = 81.8\%$

UNIT TESTING

What is unit testing?

Unit testing is a method by which individual units of source code are tested to determine if they are fit for use

Unit test is not

- Integration testing in which individual software modules are combined and tested as a group in their runtime environment
- Validation testing which is the final process of checking that a software system meets specifications and that it fulfills its intended purpose

Unit test goal

- So the goal of unit testing is to isolate each part of the program and show that the individual parts are correct
- A unit test provides a strict, written contract that the piece of code must satisfy

few terminology

Assertion

a statement that something should be true

Test Suite

a collection of tests

Pass / Success

test that runs ok

Fail / Failure
test that does not pass

Fixture

all the things we need to have in place in order to run a test and expect a particular outcome. Some people call this the test context

Test Double / Fake Object / Mock Objects
simulated objects that mimic the behavior of real
(complex) objects in controlled ways

Know what you're testing

A test written without a clear objective in mind is easy to spot. This type of test is long, hard to understand, and usually tests more than one thing

(When a developer has a problem naming a test, that probably means the test lacks focus)

Unit Test should be self-sufficient

A good unit test should be isolated. Avoid static variables usage and dependencies on external data (i.e. database, env settings)

A single test should not depend on running other tests before it, nor should it be affected by the order of execution of other tests

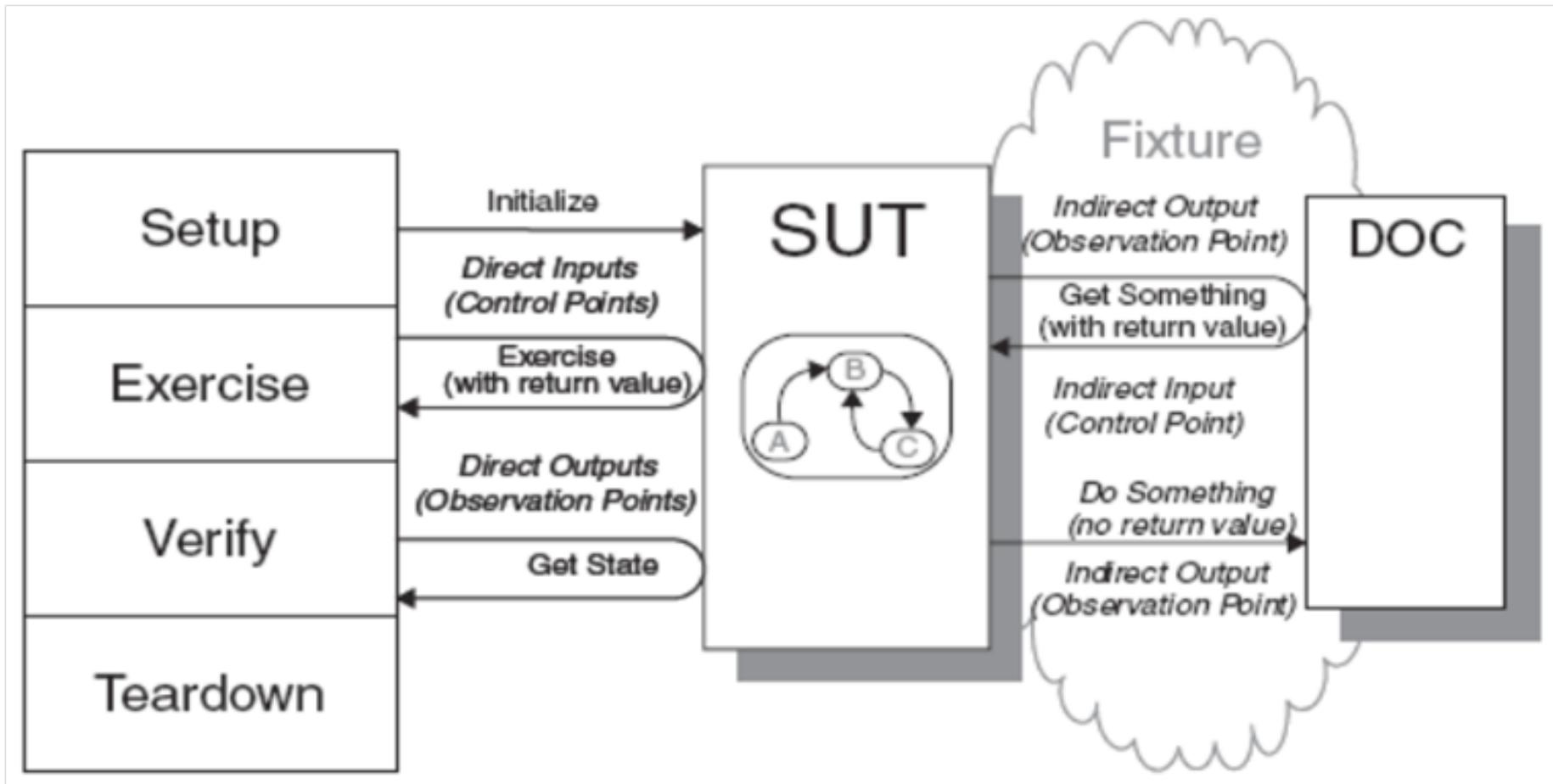
Writing unit tests can be hard when the class has complex dependencies. Fake objects can help

Unit Test should be deterministic

The worst test is the one that passes some of the time. A test should either pass all the time or fail until fixed

Avoid writing tests with random input, that introduces uncertainty and prevent to reproduce the failure

Unit test flow



Four Phases Pattern

Setup (Fixture)

```
1 @Test
2 public void subsequentNumber() {
3     NumberRangeCounter counter = new NumberRangeCounter();
4
5     int first = counter.next();
6     int second = counter.next();
7
8     assertEquals( first + 1, second );
9 }
```

Exercise

Verify

Teardown: cleaning up the fixture in case it is persistent.

SUT - System Under Test

- Definition: part of the system being tested.
- Example: single class, single module or whole application (different granularity).

DOC - Depended On Component (or Collaborator)

- Definition: entity required by a SUT to achieve a goal, same granularity as the SUT.
- Example: service class to persist data, authentication module, etc.

Code/Test coverage

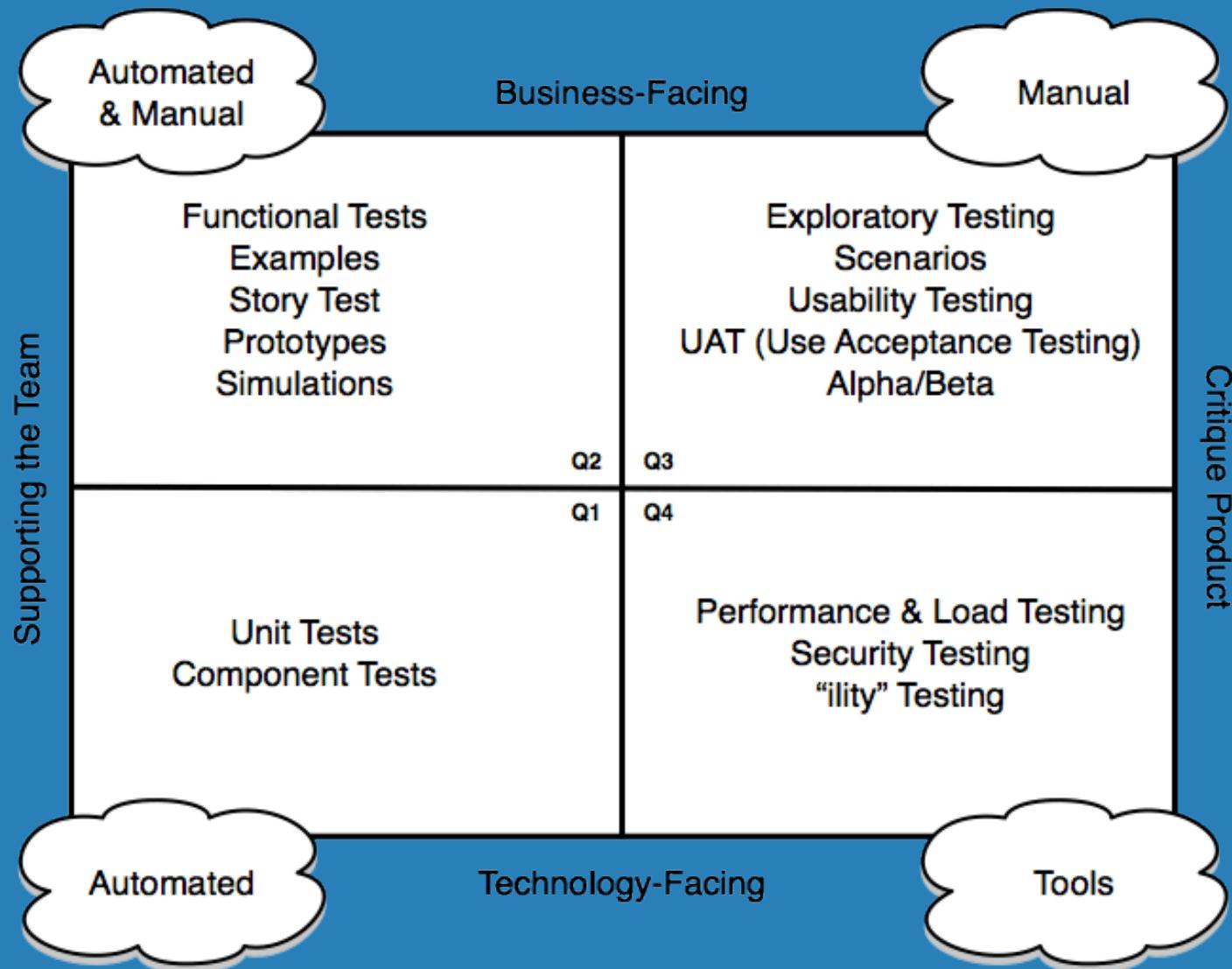
Code coverage is a measure used to describe the degree to which the source code of a program is tested by a particular test suite. A program with high code coverage has been more thoroughly tested and has a lower chance of containing software bugs than a program with low code coverage.

Note: higher code coverage does not mean better quality code.

TDD, BDD, ATDD tools

Language	TDD	BDD	ATDD
PHP	PHPUnit, CodeCeption, Simple test	Behat, PHPSpec, CodeCeption	CodeCeption
JavaScript	Qunit, Karma, chai	Jasmine, chai, Mocha	CasperJS, Protractor, NightWatch, TestCafe
.NET	Nunit, MSTest, xUnit	Nbehave, NSpec SpecsFor,xBehave.net	SpecFlow
Java	JUnit	Jbehave, Jgiven, Cuppa	Concordion
Python	pyUnit	Behave, radish, pytest-bdd	
Ruby	Test-unit	Rspec	watir
Generic	Concordion, Robot Framework, Fitness, Selenium, Cucumber, Spectacular,		

Agile testing quadrants



TEST DOUBLE

SPY, MOCK, STUB, FAKE, Dummy Object

Dependency in the Test

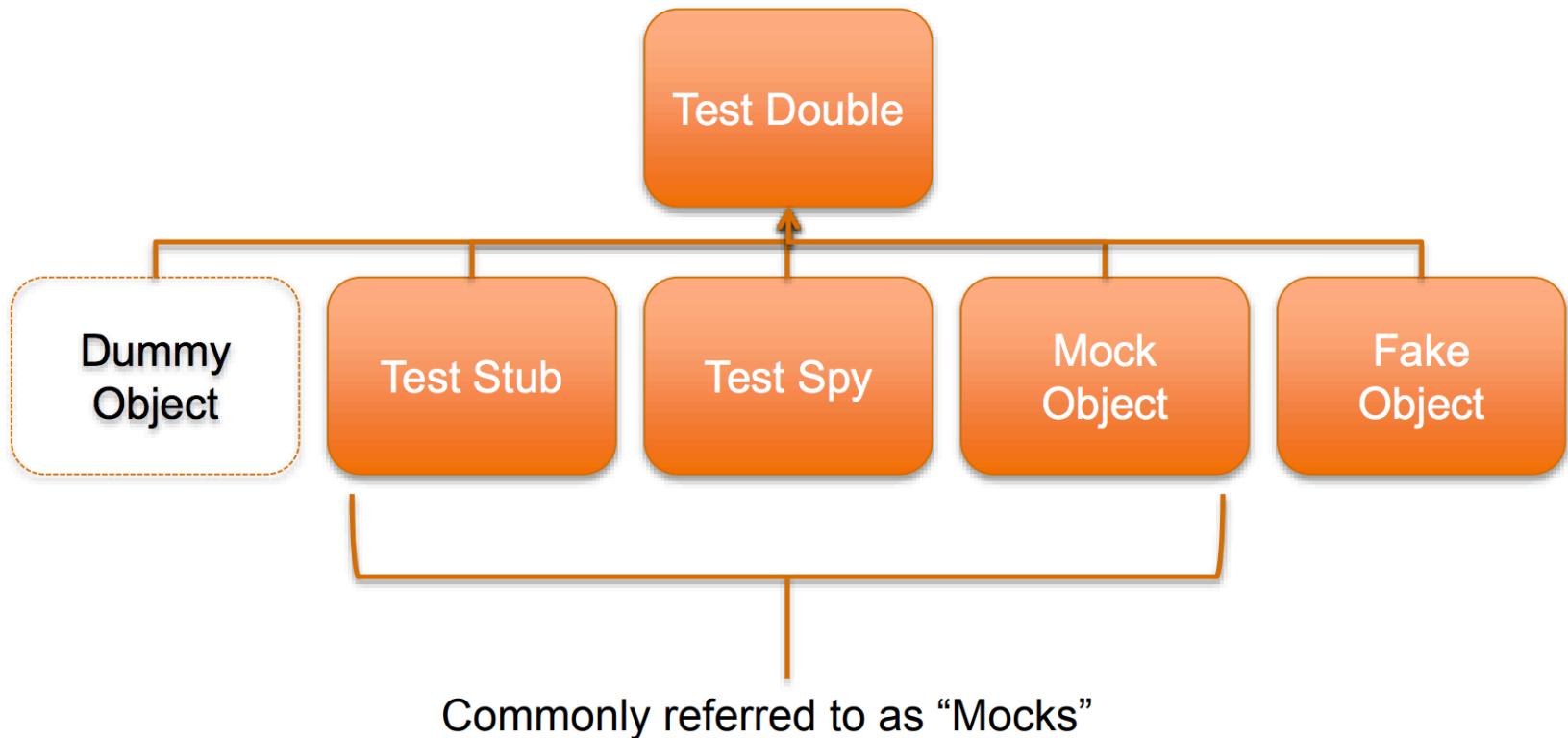
- System Under Test (SUT) Dependency
 - SUT need **another module** to perform its task
 - Handler need a \$dbmodel to perform query
 - `$handler = new Handler($dbmodel);`

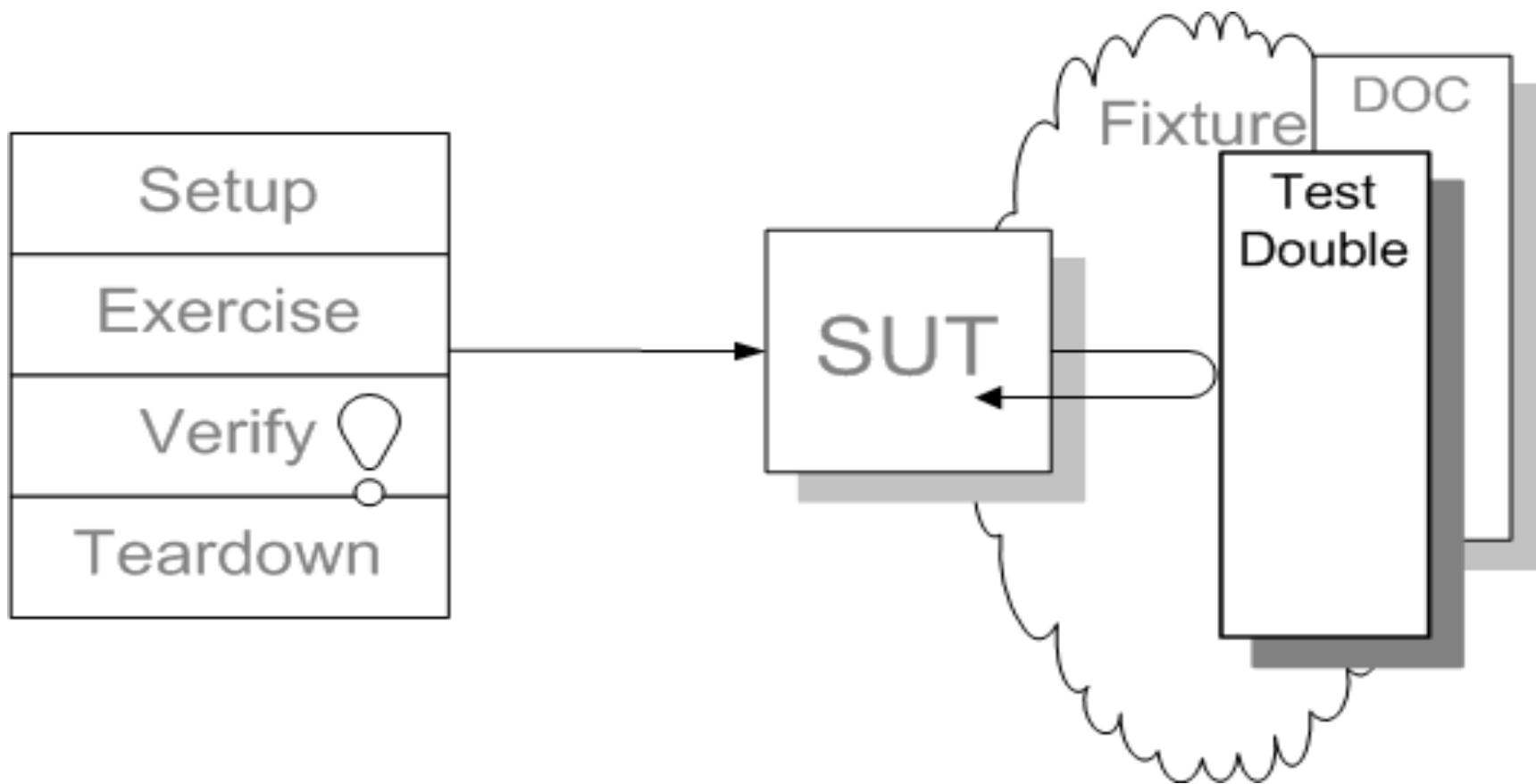
Real Object is Hard to Test

- Real object
 - Supplies non-deterministic results
 - (e.g. the current temperature)
 - Difficult to create or reproduce (e.g. network fail)
 - Slow (e.g. database)
 - Does not yet exist or may change behavior

Substitute Dependency

- Using Fake
 - Fake is anything **not real**
- Fake techniques
 - Mock object
 - Stub object
 - Fake object
 - Spy
 - Dummy Object

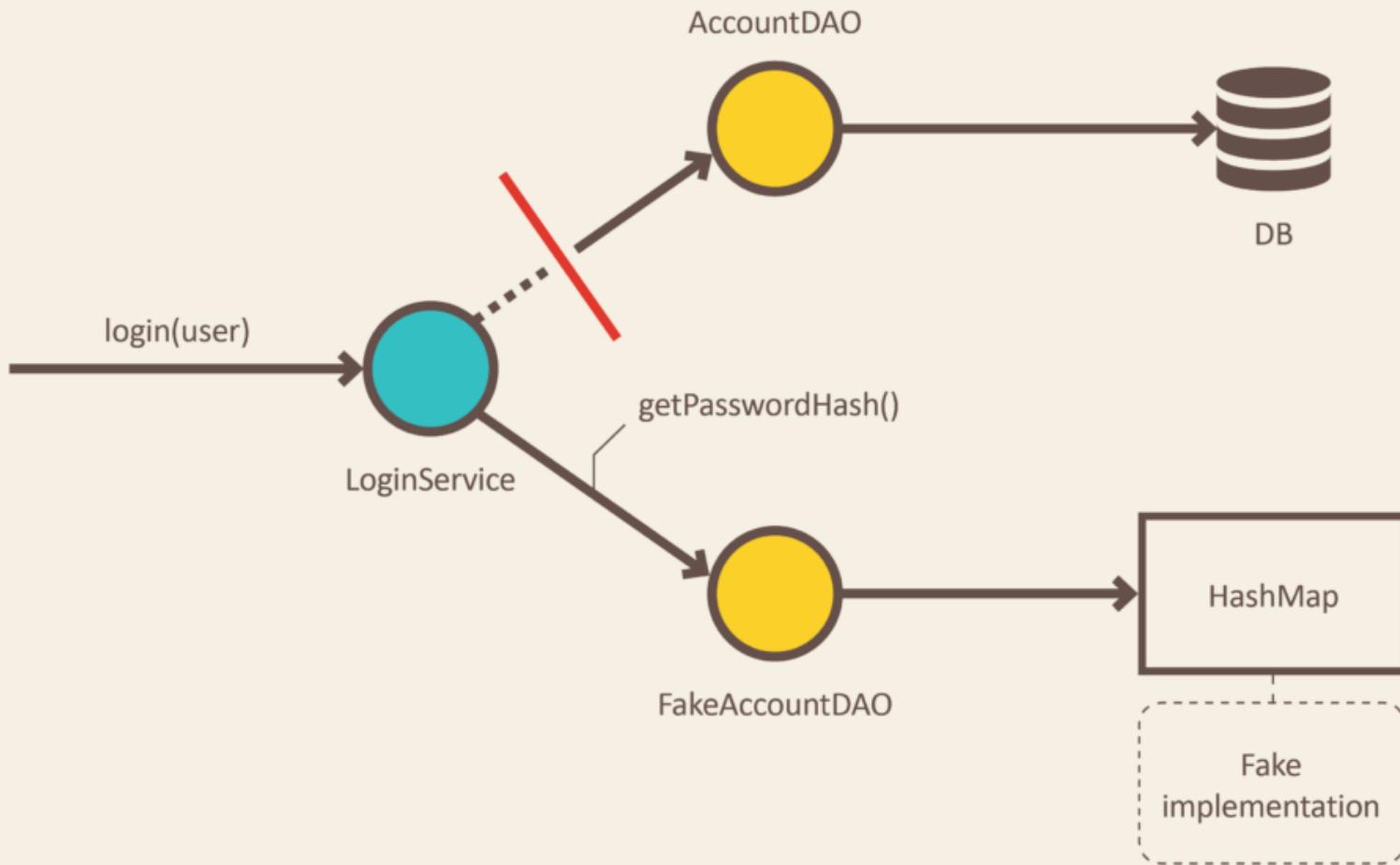




Fake

- *Fakes are objects that have working implementations, but not same as production one. Usually they take some shortcut and have simplified version of production code.*
- An example of this shortcut, can be an in-memory implementation of Data Access Object or Repository. This fake implementation will not engage database, but will use a simple collection to store data. This allows us to do integration test of services without starting up a database and performing time consuming requests.

Fake



Example

```
1
2 @Profile("transient")
3 public class FakeAccountRepository implements AccountRepository {
4
5     Map<User, Account> accounts = new HashMap<>();
6
7     public FakeAccountRepository() {
8         this.accounts.put(new User("john@bmail.com"), new UserAccount());
9         this.accounts.put(new User("boby@bmail.com"), new AdminAccount());
10    }
11
12    String getPasswordHash(User user) {
13        return accounts.get(user).getPasswordHash();
14    }
15 }
```

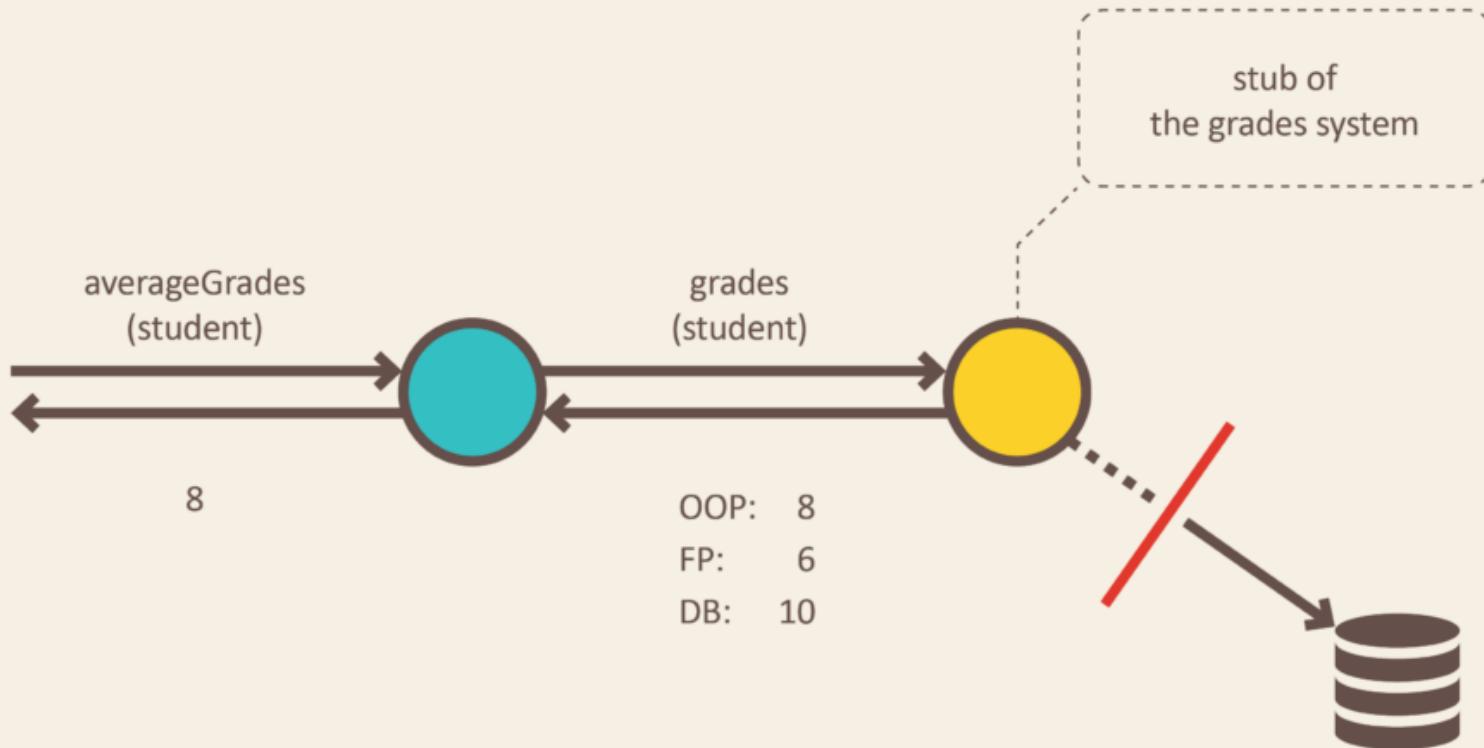
Apart from testing, fake implementation can come handy for prototyping and spikes. We can quickly implement and run our system with in-memory store, deferring decisions about database design. Another example can be also a fake payment system, that will always return successful payments.

STUB

Stub is an object that holds predefined data and uses it to answer calls during tests. It is used when we cannot or don't want to involve objects that would answer with real data or have undesirable side effects.

An example can be an object that needs to grab some data from the database to respond to a method call. Instead of the real object, we introduced a stub and defined what data should be returned

Stub




```
1
2  public class GradesService {
3      private final Gradebook gradebook;
4
5      public GradesService(Gradebook gradebook) {
6          this.gradebook = gradebook;
7      }
8
9      Double averageGrades(Student student) {
10         return average(gradebook.gradesFor(student));
11     }
12 }
```

```
1
2  public class GradesService {
3      private final Gradebook gradebook;
4
5      public GradesService(Gradebook gradebook) {
6          this.gradebook = gradebook;
7      }
8
9      Double averageGrades(Student student) {
10         return average(gradebook.gradesFor(student));
11     }
12 }
```

Instead of calling database from Gradebook store to get real students grades, we preconfigure stub with grades that will be returned. We define just enough data to test average calculation algorithm.

```
1
2  public class GradesServiceTest {
3      private Student student;
4      private Gradebook gradebook;
5
6      @Before
7      public void setUp() throws Exception {
8          gradebook = mock(Gradebook.class);
9          student = new Student();
10     }
11
12     @Test
13     public void calculates_grades_average_for_student() {
14         when(gradebook.gradesFor(student)).thenReturn(grades(8, 6, 10)); //stubbing grade
15         double averageGrades = new GradesService(gradebook).averageGrades(student);
16         assertThat(averageGrades).isEqualTo(8.0);
17     }
18 }
```

CQS – Command Query Separation

Methods that return some result and do not change the state of the system, are called **Query**. Method *averageGrades*, that returns average of student grades is a good example.

```
Double averageGrades(Student student);
```

It returns a value and is free of side effects. As we have seen in students grading example, for testing this type of method we use Stubs. We are replacing real functionality to provide values needed for method to perform its job. Then, values returned by the method can be used for assertions.

There is also another category of methods called **Command**. This is when a method performs some actions, that changes the system state, but we don't expect any return value from it.

```
void sendReminderEmail(Student student);
```

A good practice is to divide an object's methods into those two separated categories.

This practice was named: Command Query separation by Bertrand Meyer in his book "[Object Oriented Software Construction](#)".

For testing **Query type methods** we should prefer use of **Stubs** as we can verify method's return value. But what about **Command type** of methods, like method sending an e-mail? How to test them when they do not return any values? The answer is **Mock**.

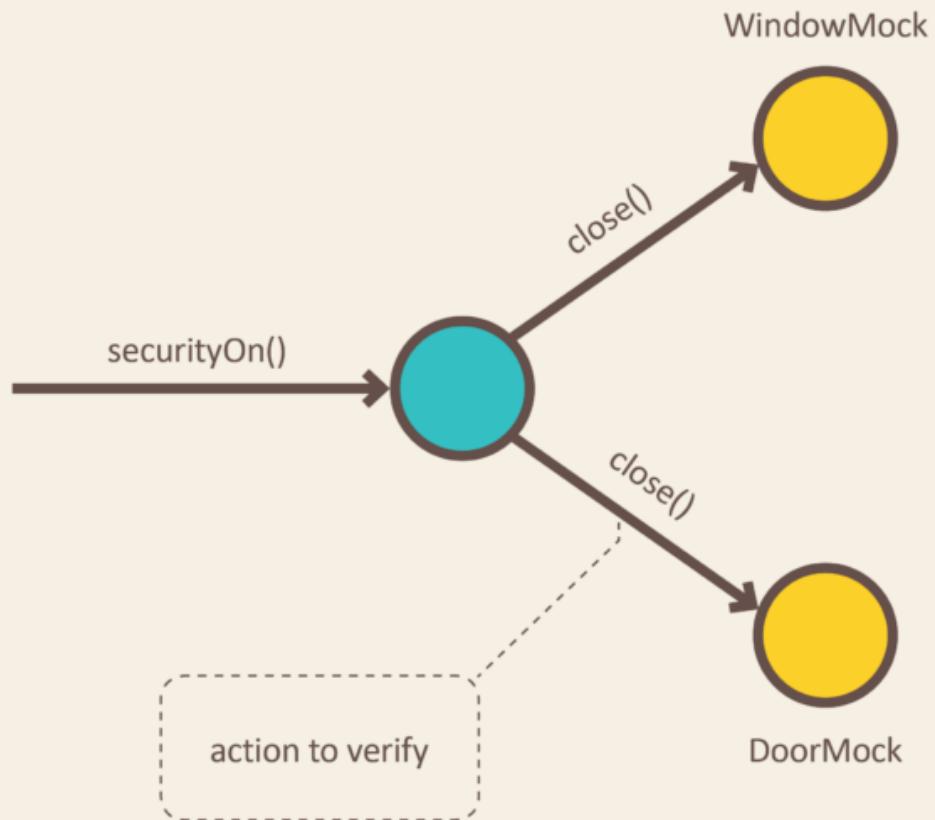
Mock

*Mocks are objects that register calls they receive.
In test assertion we can verify on Mocks that all expected actions were performed.*

We use mocks when we don't want to invoke production code or when there is no easy way to verify, that intended code was executed. There is no return value and no easy way to check system state change. An example can be a functionality that calls e-mail sending service.

We don't want to send e-mails each time we run a test. Moreover, it is not easy to verify in tests that a right email was send. Only thing we can do is to verify the outputs of the functionality that is exercised in our test. In other worlds, verify that e-mail sending service was called.

Mock



```
1
2  public class SecurityCentral {
3      private final Window window;
4      private final Door door;
5
6      public SecurityCentral(Window window, Door door) {
7          this.window = window;
8          this.door = door;
9      }
10
11     void securityOn() {
12         window.close();
13         door.close();
14     }
15 }
```

We don't want to close real doors to test that security method is working, right? Instead, we place door and window mocks objects in the test code.

```
1
2  public class SecurityCentralTest {
3      Window windowMock = mock(Window.class);
4      Door doorMock = mock(Door.class);
5
6      @Test
7      public void enabling_security_locks_windows_and_doors() {
8          SecurityCentral securityCentral = new SecurityCentral(windowMock, doorMock);
9          securityCentral.securityOn();
10         verify(doorMock).close();
11         verify(windowMock).close();
12     }
13 }
```

After execution of securityOn method, window and door mocks recorded all interactions. This lets us verify that window and door objects were instructed to close themselves. That's all we need to test from SecurityCentral perspective.

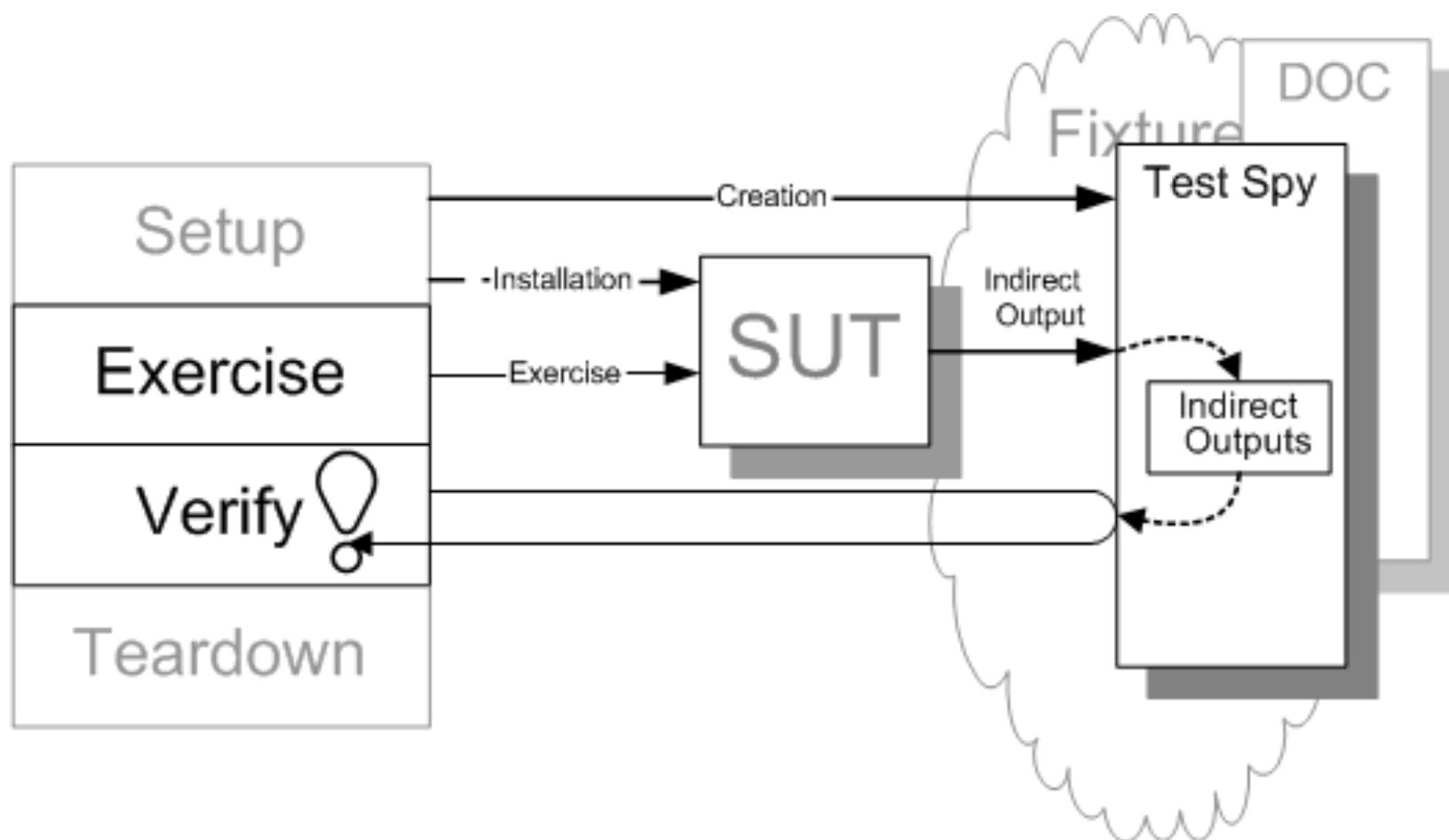
You may ask how can we tell if door and window will be closed for real if we use mock? The answer is that we can't. But we don't care about it. This is not responsibility of SecurityCentral. This is responsibility of Door and Window alone to close itself when they get proper signal. We can test it independently in different unit test.

SPY

Test Spy is a simple and intuitive way to implement an observation point that exposes the indirect outputs of the SUT so they can be verified.

We should use a Test Spy if any of the following are true:

- We are verifying the indirect outputs of the SUT and we cannot predict the value of all attributes of the interactions with the SUT ahead of time.
- We want the assertions to be visible in the test and we don't think the setting up of the Mock Object (page X) expectations is sufficiently intent-revealing.
- Our test requires test-specific equality therefore we cannot use the standard definition of equality as implemented in the SUT and we are using tools that generate the Mock Object but which do not give us control over the Assertion Methods that are being called.
- A failed assertion cannot be reported effectively back to the Test Runner (page X). This might occur if the SUT is running inside a container that catches all exceptions and makes it difficult to report the results or if the logic of the SUT runs in a different thread or process from the test which invokes it. (Both of these cases really beg refactoring to allow us to test the SUT logic directly, but that is the subject of another chapter.)



Or in simple words

As the name might suggest, spies are used to get information about function calls. For example, a spy can tell us how many times a function was called, what arguments each call had, what values were returned, what errors were thrown, etc.

As such, a spy is a good choice whenever the goal of a test is to verify something happened. Combined with assertions, we can check many different results by using a simple spy.

The most common scenarios with spies involve...

- Checking how many times a function was called
- Checking what arguments were passed to a function

```
it('should call save once', function() {  
  var save = sinon.spy(Database, 'save');  
  
  setupNewUser({ name: 'test' }, function() { });  
  
  save.restore();  
  sinon.assert.calledOnce(save);  
});
```

```
it('should pass object with correct values to save', function() {
  var save = sinon.spy(Database, 'save');
  var info = { name: 'test' };
  var expectedUser = {
    name: info.name,
    nameLowercase: info.name.toLowerCase()
  };

  setupNewUser(info, function() { });

  save.restore();
  sinon.assert.calledWith(save, expectedUser);
});
```

Dummy object

a *Dummy Object* can be passed as an argument removing the need to build a real object.

```
1  Invoice inv = new Invoice(new DummyCustomer());
2
3  public class DummyCustomer implements ICustomer {
4
5    public DummyCustomer() {
6      // Real simple; nothing to initialize!
7    }
8
9    public int getZone() {
10      throw new RuntimeException("This should never be called!");
11    }
12 }
```

Not only for testing

- Single Responsibility Principle
 - Should alert system holds notifier and data provider logic?
 - Ex. Should the class read registry directly?
- Dependency Inversion Principle
- Open Close Principle

The Law of Demeter

- Definition
 - Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.
 - Each unit should only talk to its friends; don't talk to strangers.
 - Only talk to your immediate friends.

Violation of the Law

- How do you test for?
 - Mock for mock object, for another mock object
 - Like Looking for a Needle in the Haystack

```
class Monitor {  
    SparkPlug sparkPlug;  
    Monitor(Context context) {  
        this.sparkPlug = context.  
            getCar().getEngine().  
            getPiston().getSparkPlug();  
    }  
}
```

Law of Demeter - Explicit

- Explicit
 - We should not need to know the details of collaborators

```
class Mechanic {  
    Engine engine;  
    Mechanic(Engine engine) {  
        this.engine = engine;  
    }  
}
```

Guide – Write Testable Code

- Bad smell for non-testable Code
 - Constructor does real work
 - Digging into collaborators
 - Brittle Global State & Singletons
 - Class Does Too Much
- References
 - Guide – Write Testable Code
 - <http://misko.hevery.com/attachments/Guide-Writing%20Testable%20Code.pdf>

Mocking frameworks

Language	Frameworks
PHP	Phake, PHPUnit, Mockery, Phony
JavaScript	Sinon.js, Jest.js, JSMockito, JS-Mock
Python	Mock, FlexMock, Mox, Mocker, Dingus, Fudge
.NET	Nsubstitute, Rhino Mocks, Moq, Fakelteeasy, Nmoq 3
Java	Mockito, Jmockit, EasyMock
Ruby	Mocha, rspec-mocks, flexmock

Conclusion

- Writing good unit tests is hard
- Good OO design brings good testability
- Using stub/mock from mocking framework

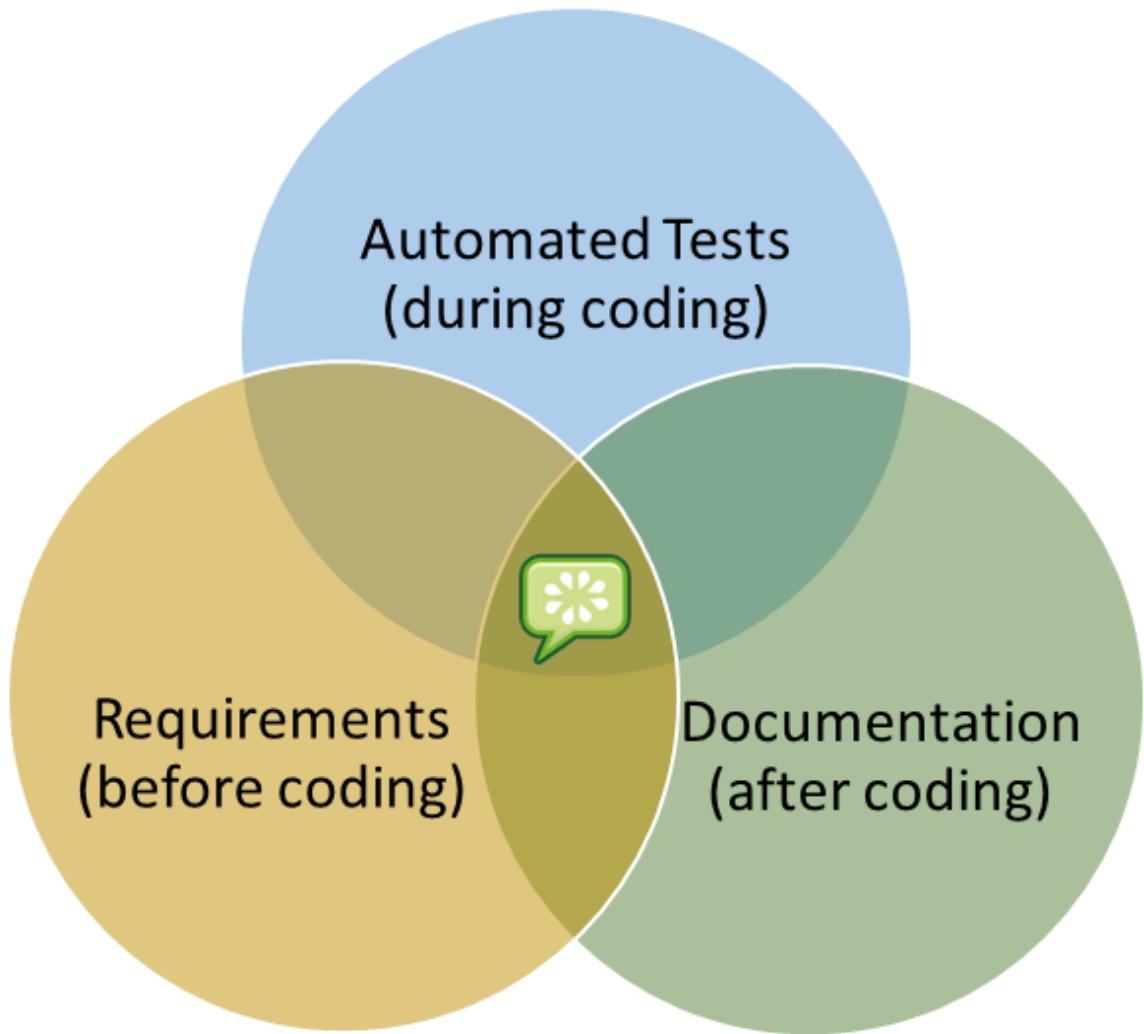
Behavior driven development- BDD

Defining specifications

With Cucumber

Cucumber tool
supports Behavior
Driven Development
and is as cool as a
"Cucumber"





Have vast support for

- Ruby
- Java
- PHP
- JavaScript
- Python
- .NET
- C++
- Lua

Installation - Mac

Step 1 – Install RVM and Ruby

```
curl -L get.rvm.io | bash -s stable  
source ~/.rvm/scripts/rvm  
$ rvm list known  
$rvm install 1.9.3  
# This installs Ruby 1.9.3
```

Step 2 – Install the required gems

Execute:

```
gem update --system  
gem install rspec --no-ri --no-rdoc  
gem install watir-webdriver --no-ri --no-rdoc  
gem install cucumber --no-ri --no-rdoc
```

Cucumber Basics

- For every cucumber project there is a single directory at the root of the project named "**features**".
- This is where all of our cucumber features will reside.
- In this directory we will find additional directories, which is **step_definition** and **support directories**
- Each feature file has a **.feature** extension

What is "Feature File"?

Feature File consist of following components -

- **Feature:** A feature would describe the current test script which has to be executed.
- **Scenario:** Scenario describes the steps and expected outcome for a particular test case.
- **Scenario Outline:** Same scenario can be executed for multiple sets of data using scenario outline. The data is provided by a tabular structure separated by (| |).
- **Given:** It specifies the context of the text to be executed. By using datatables "Given", step can also be parameterized.
- **When:** "When" specifies the test action that has to performed
- **Then:** The expected outcome of the test can be represented by "Then"

Example

Feature: Visit what is scrum page in scrumalliance.org

Scenario : Visit scrumalliance.org

Given: I am on scrumalliance.org

When: I click on what is scrum on home page

Then: I should see learn about scrum page

Gherkin

Gherkin is the language that Cucumber understands. It is a Business Readable, Domain Specific Language that lets you describe software's behavior without detailing how that behavior is implemented.

Gherkin serves two purposes

- documentation and automated tests. The third is a bonus feature
- when it yells in red it's talking to you, telling you what code you should write.

Gherkin's grammar is defined in the Treetop grammar that is part of the Cucumber codebase. The grammar exists in different flavors for many spoken languages (60+), so that your team can use the keywords in your own language.

There are a few conventions.

- Single Gherkin source file contains a description of a single feature.
- Source files have .feature extension.

Keywords

keyword	localized
name	'English'
native	'English'
encoding	'UTF-8'
feature	'Feature'
background	'Background'
scenario	'Scenario'
scenario_outline	'Scenario Outline'
examples	'Examples' / 'Scenarios'
given	'Given'
when	'When'
then	'Then'
and	'And'
but	'But'

Languages

cucumber --language help

ar	Arabic	العربية
bg	Bulgarian	български
cat	Catalan	català
cy	Welsh	Cymraeg
cz	Czech	Česky
da	Danish	dansk
de	German	Deutsch
en	English	English
en-au	Australian	Australian
en-lol	LOLCAT	LOLCAT
en-tx	Texan	Texan
es	Spanish	español
et	Estonian	eesti keel
fi	Finnish	suomi
fr	French	français
he	Hebrew	עברית
hr	Croatian	hrvatski
hu	Hungarian	magyar
id	Indonesian	Bahasa Indonesia

In Gherkin, each line that isn't blank has to start with a Gherkin *keyword*, followed by any text you like. The main keywords are:

- Feature
- Scenario
- Given, When, Then, And, But (Steps)
- Background
- Scenario Outline
- Examples

There are a few extra keywords as well:

- """ (Doc Strings)
- | (Data Tables)
- @ (Tags)
- # (Comments)

Feature

A .feature file is supposed to describe a single feature of the system, or a particular aspect of a feature. It's just a way to provide a high-level description of a software feature, and to group related scenarios.

A feature has three basic elements---

- the Feature: keyword,
- a name (on the same line)
- and an optional (but highly recommended) description that can span multiple lines.

Cucumber does not care about the name or the description---the purpose is simply to provide a place where you can document important aspects of the feature, such as a brief explanation and a list of business rules (general acceptance criteria).

example

Feature: Refund item

Sales assistants should be able to refund customers' purchases.

This is required by the law, and is also essential in order to keep customers happy.

Rules:

- Customer must present proof of purchase
- Purchase must be less than 30 days ago

Scenario

A scenario is a *concrete example* that *illustrates* a business rule. It consists of a list of steps.

We can have as many steps as we like, but we recommend to keep the number at 3-5 per scenario. If they become longer than that they lose their expressive power as specification and documentation.

In addition to being a specification and documentation, a scenario is also a *test*. As a whole, your scenarios are an *executable specification* of the system.

Scenarios follow the same pattern:

- Describe an initial context
- Describe an event
- Describe an expected outcome

Given When Then

- Cucumber scenarios consist of steps, also known as Givens, Whens and Thens.
- These words have been carefully selected for their purpose, and we should know what the purpose is to get into the BDD mindset.

Given

The purpose of givens is to **put the system in a known state** before the user (or external system) starts interacting with the system (in the When steps). Avoid talking about user interaction in givens. If we were creating use cases, givens would be our preconditions.

Examples:

- Create records (model instances) / set up the database state.
- It's ok to call into the layer "inside" the UI layer here (in Rails: talk to the models).
- Log in a user (An exception to the no-interaction recommendation. Things that "happened earlier" are ok).

When

The purpose of When steps is to **describe the key action** the user performs (or, using Robert C. Martin's metaphor, the state transition).

Examples:

- Interact with a web page
(Webrat/Watir/Selenium *interaction* etc should mostly go into When steps).
- Interact with some other user interface element.
- Developing a library? Kicking off some kind of action that has an observable effect somewhere else.

Then

The purpose of Then steps is to **observe outcomes**. The observations should be related to the business value/benefit in your feature description. The observations should also be on some kind of *output* – that is something that comes *out* of the system (report, user interface, message) and not something that is deeply buried inside it (that has no business value).

Examples:

- Verify that something related to the Given+When is (or is not) in the output
- Check that some external system has received the expected message (was an email with specific content sent?)
- While it might be tempting to implement Then steps to just look in the database – resist the temptation. You should only verify outcome that is observable for the user (or external system) and databases usually are not.

And, But

Scenario: Multiple Givens

Given one thing

Given another thing

Given yet another thing

When I open my eyes

Then I see something

Then I don't see something

else

Scenario: Multiple Givens

Given one thing

And another thing

And yet another thing

When I open my eyes

Then I see something

But I don't see something

else

Step definitions

For each step Cucumber will look for a matching step definition. A step definition is written in Ruby. Each step definition consists of a keyword, a string or regular expression, and a block. Example:

```
# features/step_definitions/coffee_steps.rb  
  
Then "I should be served coffee" do  
  @machine.dispensed_drink.should == "coffee"  
End
```

Step definitions can also take parameters if we use regular expressions:

```
# features/step_definitions/coffee_steps.rb
```

Given /there are (\d+) coffees left in the machine/

do |n|

```
  @machine = Machine.new(n.to_i)
```

end

A step definition is analogous to a method definition / function definition in any kind of OO/procedural programming language. Step definitions can take 0 or more arguments, identified by groups in the Regexp (and an equal number of arguments to the Proc).

Some people are uncomfortable with Regular Expressions. It's also possible to define Step Definitions using strings and \$variables like this:

```
Given "I have $n cucumbers in my belly" do |cukes|
  # Some Ruby code here
end
```

In this case the String gets compiled to a Regular Expression behind the scenes: /^I have (.*) cucumbers in my belly\$/.

Then there are Steps. Steps are declared in our features/*.feature files. Here is an example:

Given I have 93 cucumbers in my belly

A step is analogous to a method or function invocation. In this example, we are “calling” the step definition above with one argument — the string “93”. Cucumber matches the Step against the Step Definition’s Regexp and takes all of the captures from that match and passes them to the Proc.

Step status

Successful steps: When Cucumber finds a matching Step Definition it will execute it. If the block in the step definition doesn't raise an Exception, the step is marked as successful (green). What you return from a Step Definition has no significance what so ever.

Undefined steps: When Cucumber can't find a matching Step Definition the step gets marked as yellow, and all subsequent steps in the scenario are skipped. If you use --strict this will cause Cucumber to exit with 1.

Pending steps: When a Step Definition's Proc invokes the pending method, the step is marked as yellow (as with undefined ones), reminding you that you have work to do. If you use --strict this will cause Cucumber to exit with 1.

Failed steps: When a Step Definition's Proc is executed and raises an error, the step is marked as red. What you return from a Step Definition has no significance what so ever. Returning nil or false will **not** cause a step definition to fail.

Skipped steps: Steps that follow undefined, pending or failed steps are never executed (even if there is a matching Step Definition), and are marked cyan.

String steps: steps can be defined using strings rather than regular expressions. Instead of writing

Given /`I have (.*) cucumbers in my belly$`/ do |cukes|

You could write

Given "I have \$count cucumbers in my belly" do |cukes|

Note that a word preceded by a \$ sign is taken to be a placeholder, and will be converted to match *. The text matched by the wildcard becomes an argument to the block, and the word that appeared in the step definition is disregarded.

Scenario Outline

When we have a complex business rule with severable variable inputs or outputs we might end up creating several scenarios that only differ by their values.

Let's take an example from feed planning for cattle and sheep:

Scenario: feeding a small suckler cow

Given the cow weighs 450 kg

When we calculate the feeding requirements

Then the energy should be 26500 MJ

And the protein should be 215 kg

Scenario: feeding a medium suckler cow

Given the cow weighs 500 kg

When we calculate the feeding requirements

Then the energy should be 29500 MJ

And the protein should be 245 kg

suppose there are 2 more examples !!!

Can be written as

Scenario Outline: feeding a suckler cow

Given the cow weighs <weight> kg

When we calculate the feeding requirements

Then the energy should be <energy> MJ

And the protein should be <protein> kg

Examples:

weight	energy	protein
450	26500	215
500	29500	245
575	31500	255
600	37000	305

Examples

A Scenario Outline section is always followed by one or more Examples sections, which are a container for a table.

The table must have a header row corresponding to the variables in the Scenario Outline steps.

Each of the rows below will create a new Scenario, filling in the variable values.

Background

Background allows us to add some context to the scenarios in a single feature. A Background is much like a scenario containing a number of steps. The difference is when it is run. The background is run before each of your scenarios but after any of your Before Hooks.

Feature: Multiple site support

As a Mephisto site owner

I want to host blogs for different people

In order to make gigantic piles of money

Background:

Given a global administrator named "Greg"

And a blog named "Greg's anti-tax rants"

And a customer named "Dr. Bill"

And a blog named "Expensive Therapy" owned by "Dr. Bill"

Background best practices

- Don't use "Background" to set up complicated state unless that state is actually something the client needs to know.
- Keep "Background" section short.
- Make "Background" section vivid.
- Keep scenarios short, and don't have too many.

Step Arguments

In some cases we might want to pass a larger chunk of text or a table of data to a step---something that doesn't fit on a single line.

For this purpose Gherkin has Doc Strings and Data Tables.

Doc Strings

Doc Strings are handy for passing a larger piece of text to a step definition. The syntax is inspired from Python's Docstring syntax.

The text should be offset by delimiters consisting of three double-quote marks on lines of their own:

Given a blog post named "Random" with Markdown body

"""

Some Title, Eh?

=====

Here is the first paragraph of my blog post. Lorem ipsum dolor sit amet,
consectetur adipiscing elit.

"""

In our Step Definition, there's no need to find this text and match it in our pattern. It will automatically be passed as the last parameter in the step definition.

Indentation of the opening """ is unimportant, although common practice is two spaces in from the enclosing step. The indentation inside the triple quotes, however, is significant. Each line of the Doc String will be de-indented according to the opening """. Indentation beyond the column of the opening """ will therefore be preserved.

Data Tables

Data Tables are handy for passing a list of values to a step definition:

Given the following users exist:

name	email	twitter
Aslak	aslak@cucumber.io	@aslak_hellesoy
Julien	julien@cucumber.io	@jbpros
Matt	matt@cucumber.io	@mattwynne

Tags

Tags are a way to group Scenarios. They are @-prefixed strings and we can place as many tags as we like above Feature, Scenario, Scenario Outline or Examples keywords. Space character are invalid in tags and may separate them.

Tags are inherited from parent elements. For example, if we place a tag above a Feature, all scenarios in that feature will get that tag.

Similarly, if we place a tag above a Scenario Outline or Examples keyword, all scenarios derived from examples rows will inherit the tags.

We can tell Cucumber to only run scenarios with certain tags, or to exclude scenarios with certain tags.

Cucumber can perform different operations before and after each scenario based on what tags are present on a scenario.

Hooks

```
Before do
  # do something before first step of scenario
end
```

```
After do
  # do something after each scenario
end
```

```
#tagged hook
Before('@billing', '@calculations') do
  # This will only run before scenarios tagged
  # with @billing or @calculations.
end
```

Reports

Cucumber can report results in several different formats, using formatter plugins. The available formatters plugins are:

- **Pretty:** Prints the Gherkin source to STDOUT along with additional colors and stack traces for errors
- **HTML:** Generates a HTML file, suitable for publishing.
- **JSON:** Generates a JSON file, suitable for post-processing to generate custom reports.
- **Progress:** This report prints results to STDOUT, one character at a time. It looks like this:
....F--U.....
- **Usage:** Prints statistics to STDOUT. Programmers may find it useful to find slow or unused Step Definitions
- **Junit:** Generates XML files just like Apache Ant's junitreport task. This XML format is understood by most Continuous Integration servers, who will use it to generate visual reports.
- **Rerun:** The rerun report is a file that lists the location of failed Scenarios. This can be picked up by subsequent Cucumber runs:

cucumber @rerun.txt

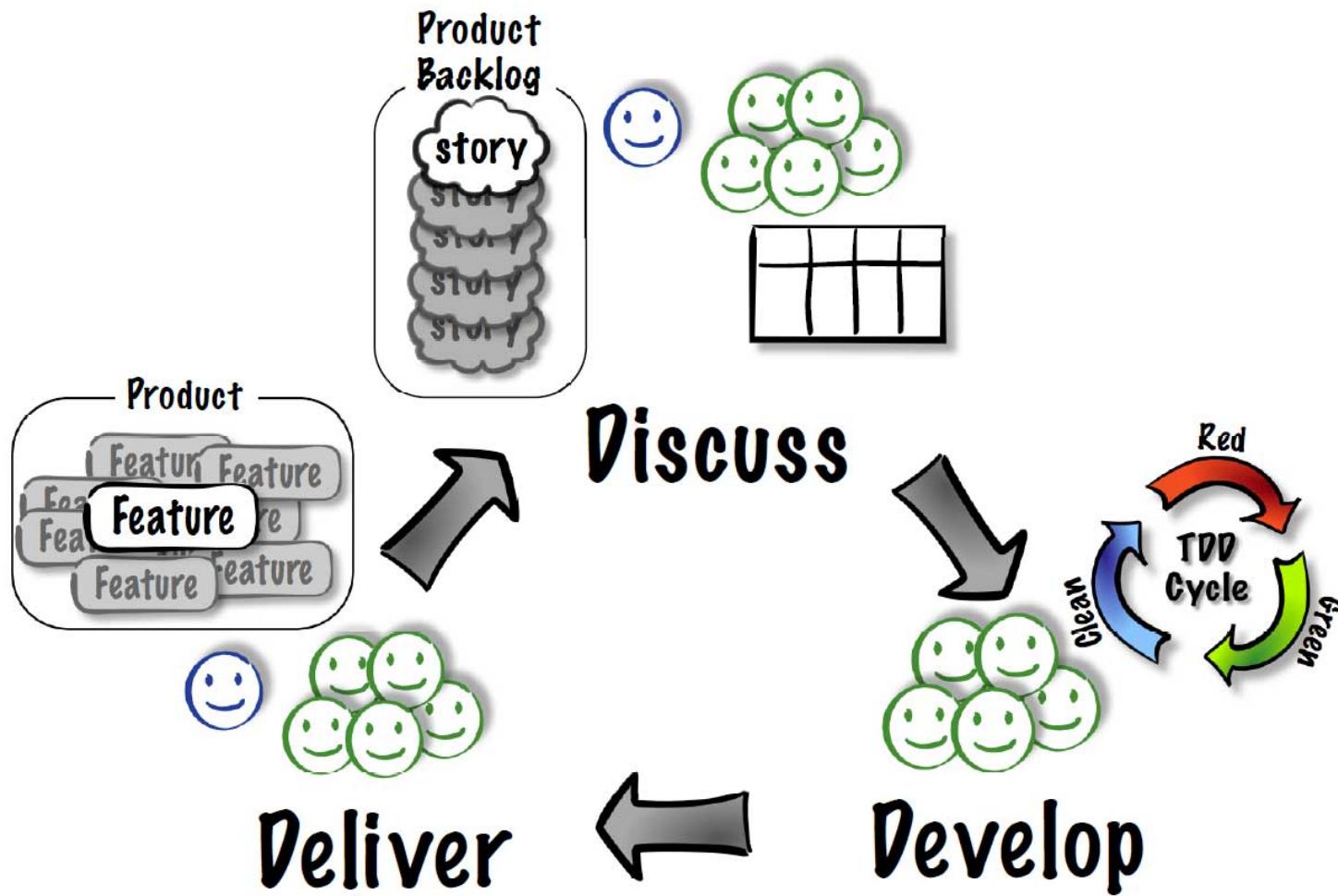
This is useful while fixing broken scenarios, as only the scenarios that failed in the previous run will be run again. This can reduce time spent fixing a bug when running all scenarios is time-consuming.

BDD WORKSHOP

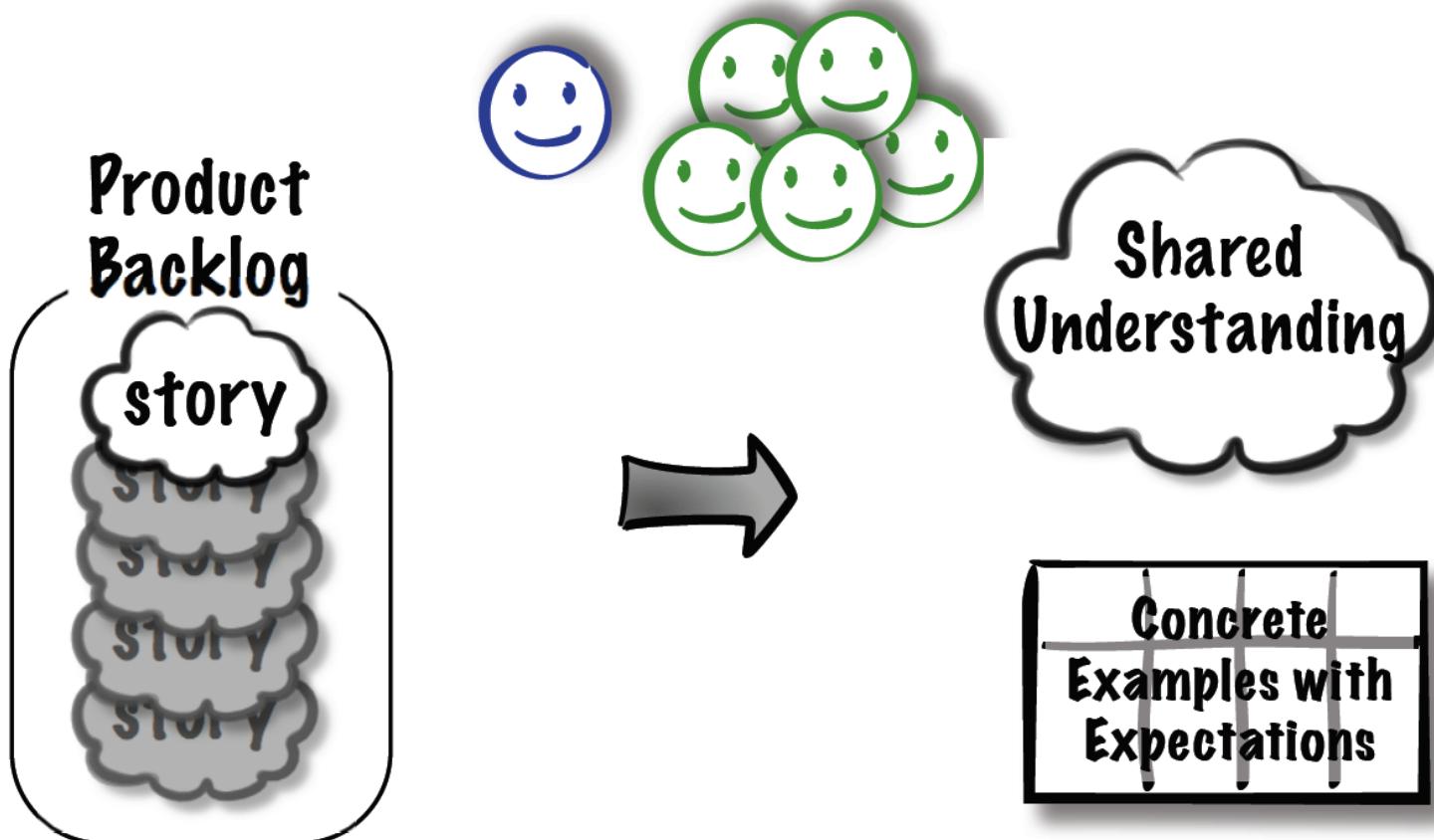
PRACTICAL TIME (2 HOURS)

Acceptance test driven development - ATDD

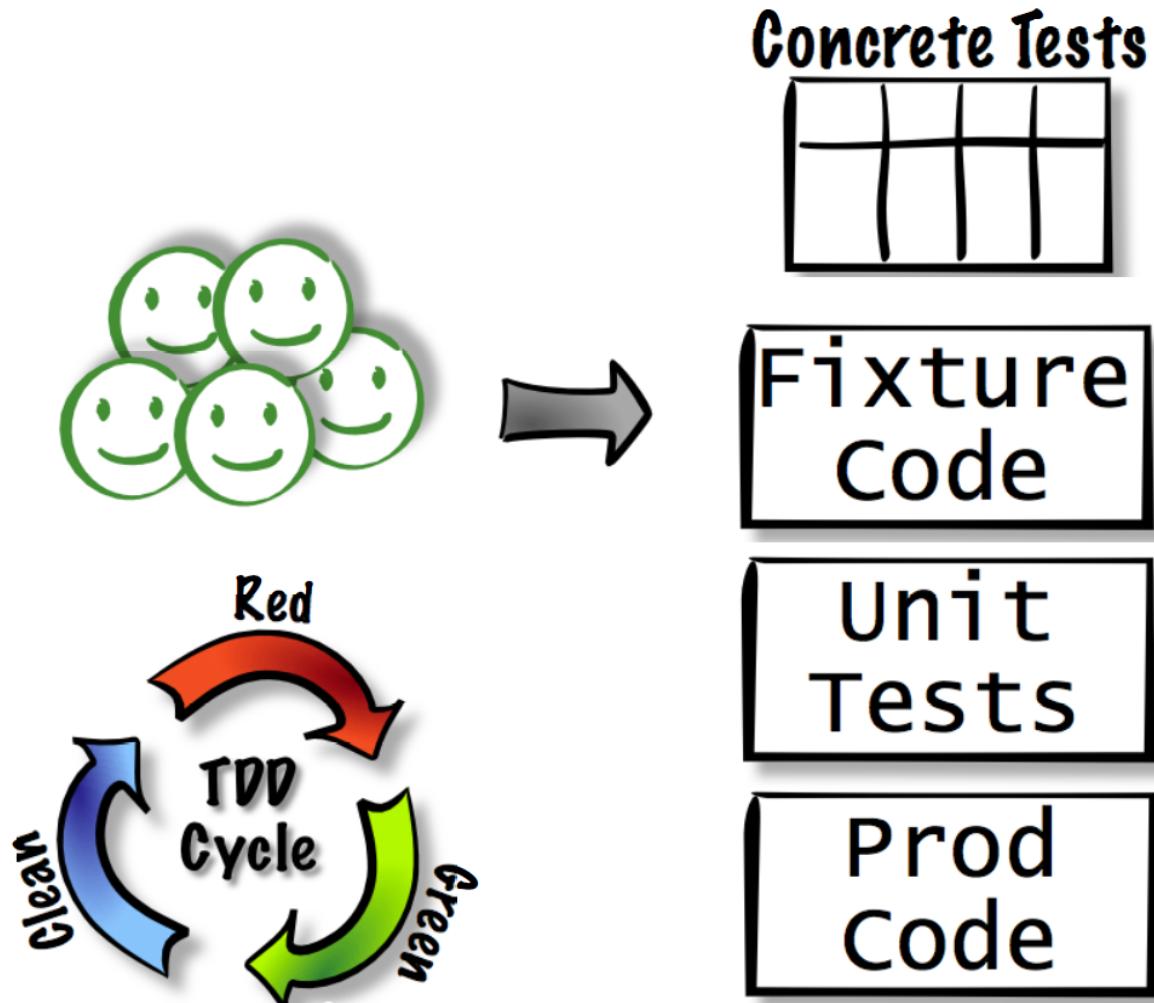
What is ATDD?



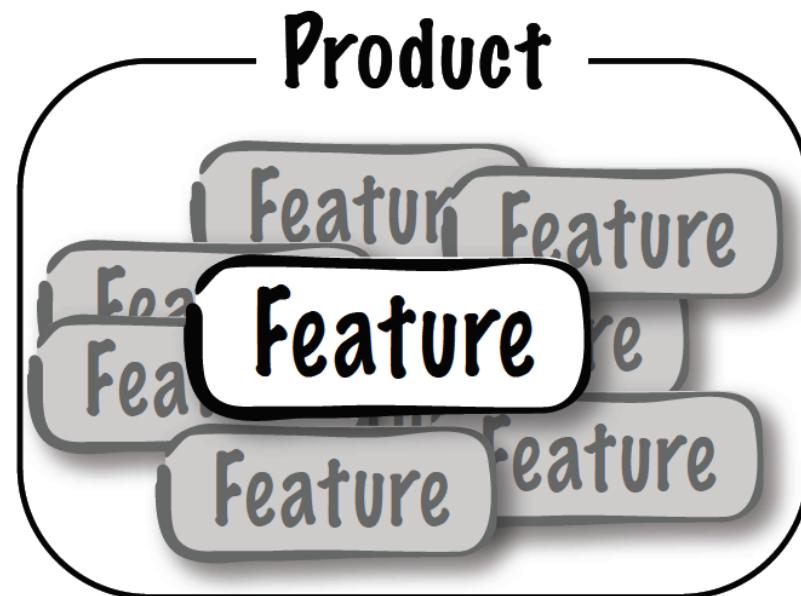
ATDD: Discuss



ATDD: Develop



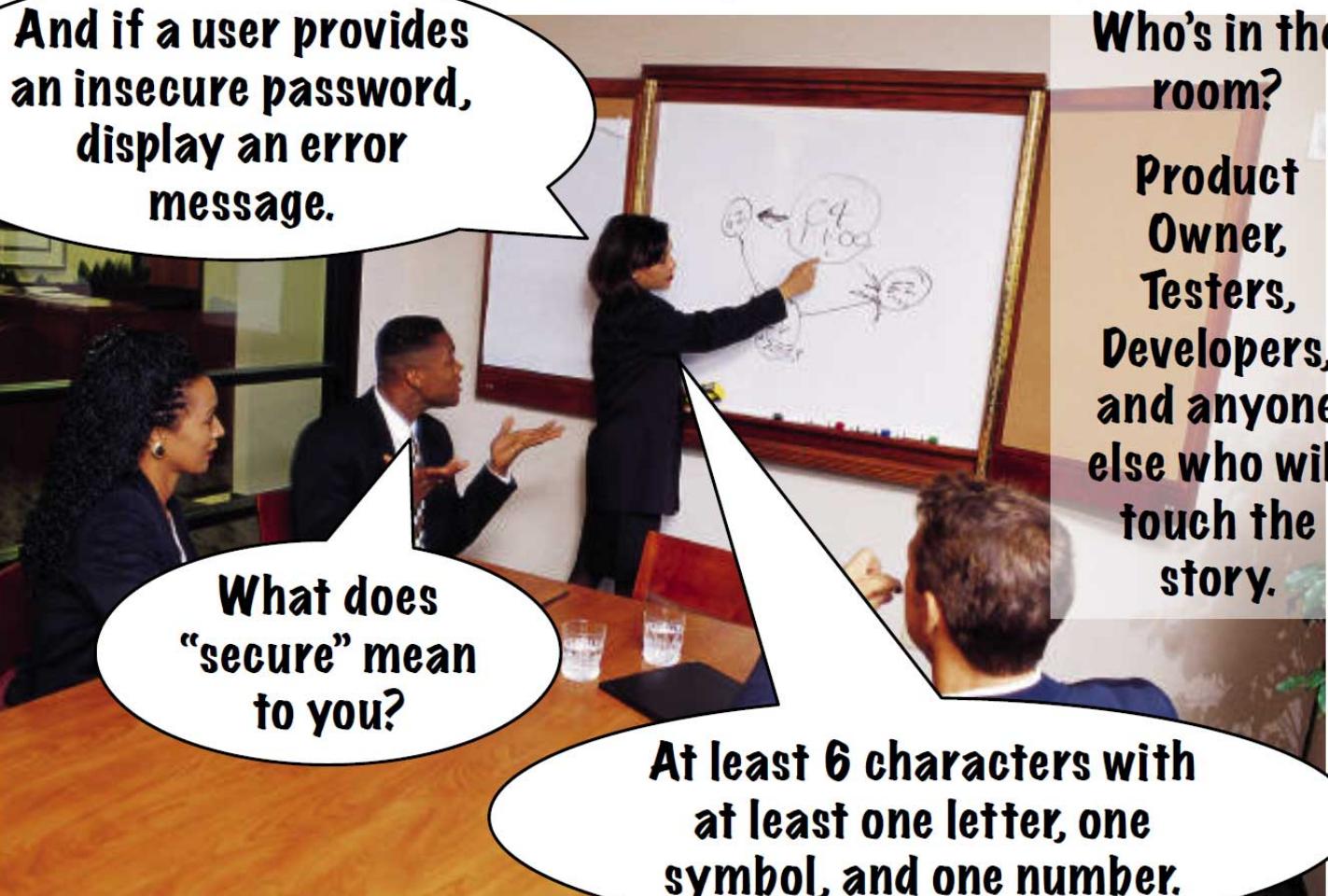
ATDD: Deliver



An example

As an administrator, I want users creating accounts to be required to choose secure passwords so that their accounts cannot be hacked by someone using a password guessing program.

Discuss



And if a user provides an insecure password, display an error message.

What does “secure” mean to you?

At least 6 characters with at least one letter, one symbol, and one number.

Who's in the room?
Product Owner, Testers, Developers, and anyone else who will touch the story.

Capture Concrete Expectations and Examples

Password	Valid?	Can be expressed as "Given - When - Then"
"p@ssw0rd"	Yes	
"p@s5"	No	
"passw0rd"	No	<u>Given</u> a user is creating an account
"p@ssword"	No	<u>When</u> they specify an insecure password
"@#§%1234"	No	<u>Then</u> they see a message, "Passwords must be at least 6 characters long with at least one letter, one number, and one symbol."

*Or can be
expressed in tables*

*Or in other formats
depending on the
Framework*

Why ATDD?

Drive Out Ambiguity and Clarify
Expectations

This is not an Argument about a Bug

"Bug Triage Meeting"

The Tuesday before release.

It's a bug.

No it's not.

Is too.

IS NOT.

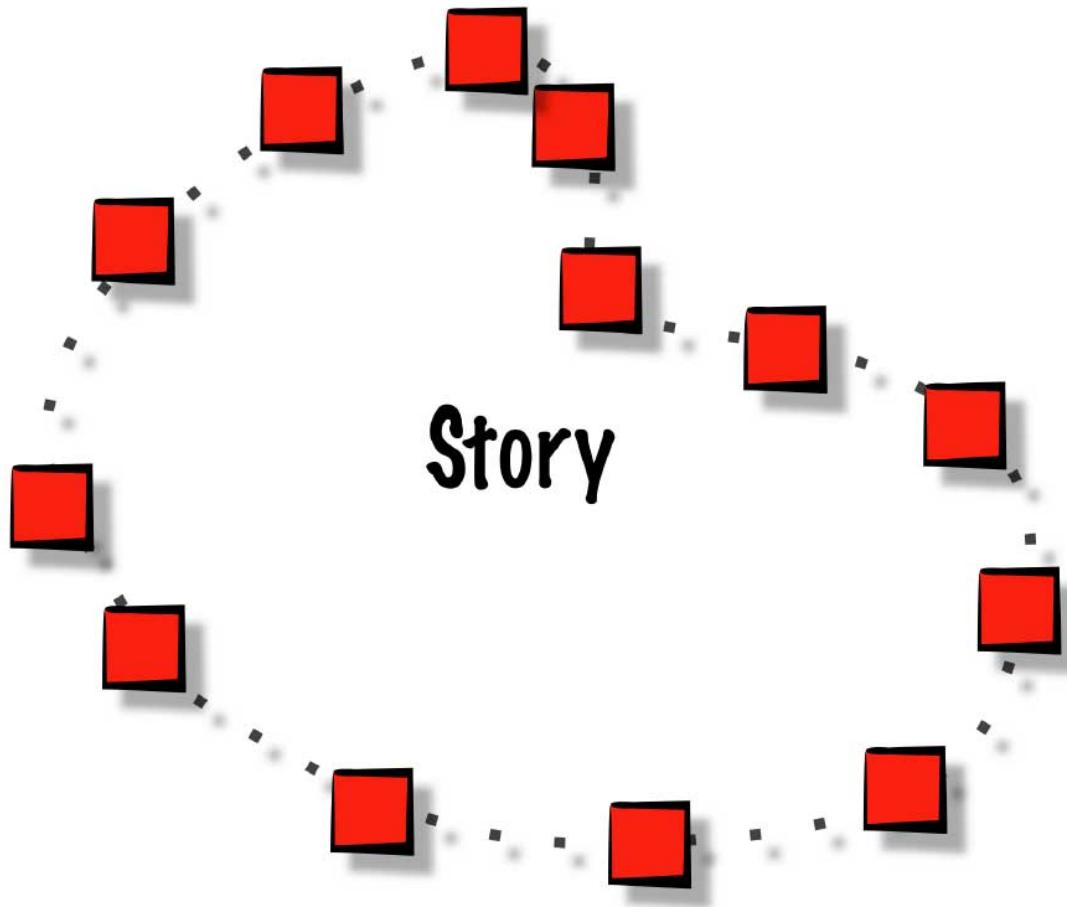
IS TOO!

NOT NOT NOT!



**Whether or not
it's a bug, if we
make a change
we'll blow the
schedule.**

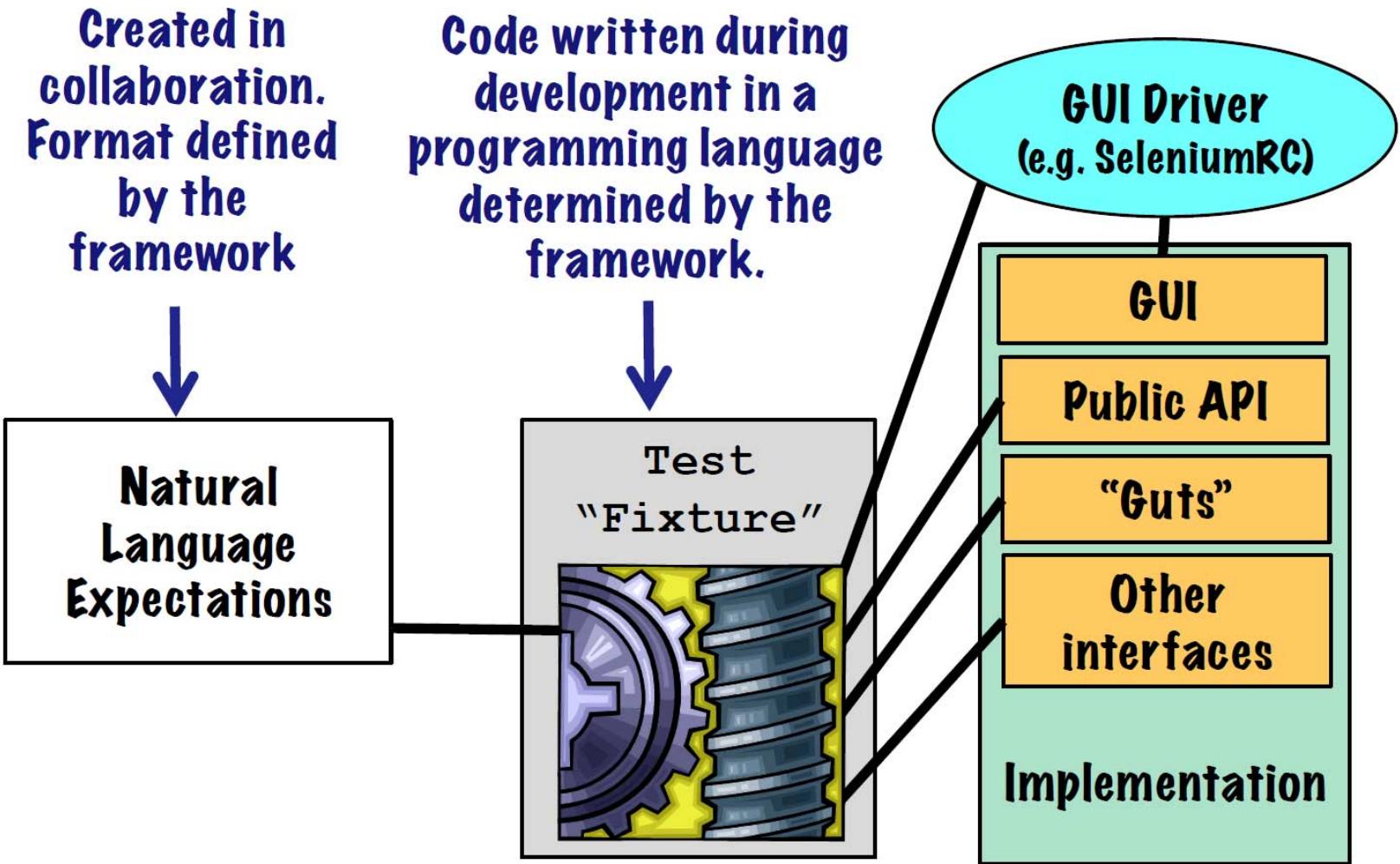
Acceptance Tests Define Scope



Examples of ATDD-Friendly Frameworks

- Cucumber: a Ruby-based BDD tool that supports “Given-When-Then”
- Fitnesse: a table-driven framework that uses a wiki for displaying and editing tests
- Robot Framework: keyword-driven framework that supports text or tables
- Concordion: Java-based framework for expressing expectations in prose

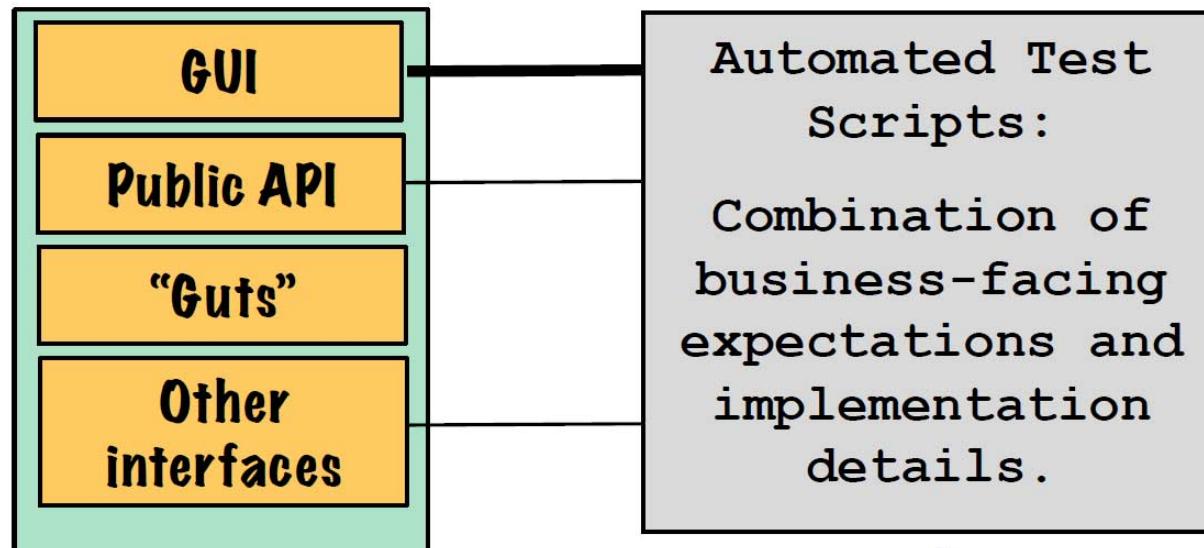
Frameworks, Interfaces, and Drivers



Characteristics of ATDD-Friendly Frameworks

- Support expressing expectations in a language and format that fits the context
- Support collaboration among the whole team including developers, testers, & the product owner
- Connect expectations to the system under test with a minimum of test code (“fixtures,” “libraries,” “steps”) to leverage expectations as executable requirements
- Play nicely with source control systems and continuous integration
- Pluggable to support a variety of interfaces

Contrasting View: Traditional Test Automation



↑
Written or recorded after the fact.
Expectations are translated, not leveraged.

Back to the example

Password	Valid?
“p@ssw0rd”	Yes
“p@s5”	No
“passw0rd”	No
“p@ssword”	No
“@#\$.%1234”	No

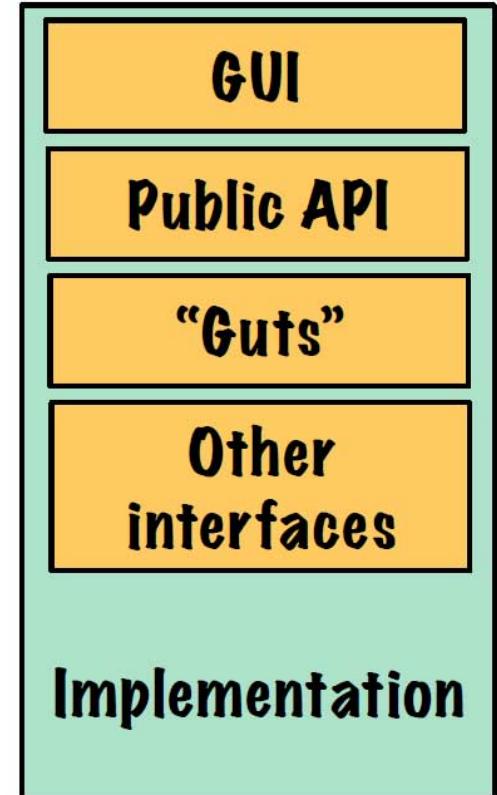
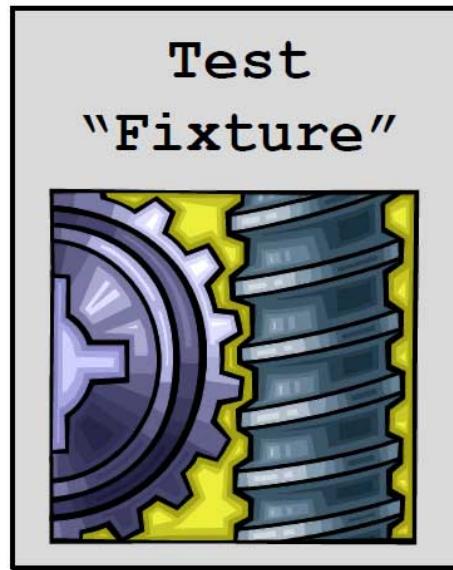
(Note that these expectations are implementation-agnostic and express just the essence of the expectation.)

Given a user is creating an account
When they specify an insecure password

Then they see a message, “Passwords must be at least 6 characters long with at least one letter, one number, and one symbol.”

and Write the Code

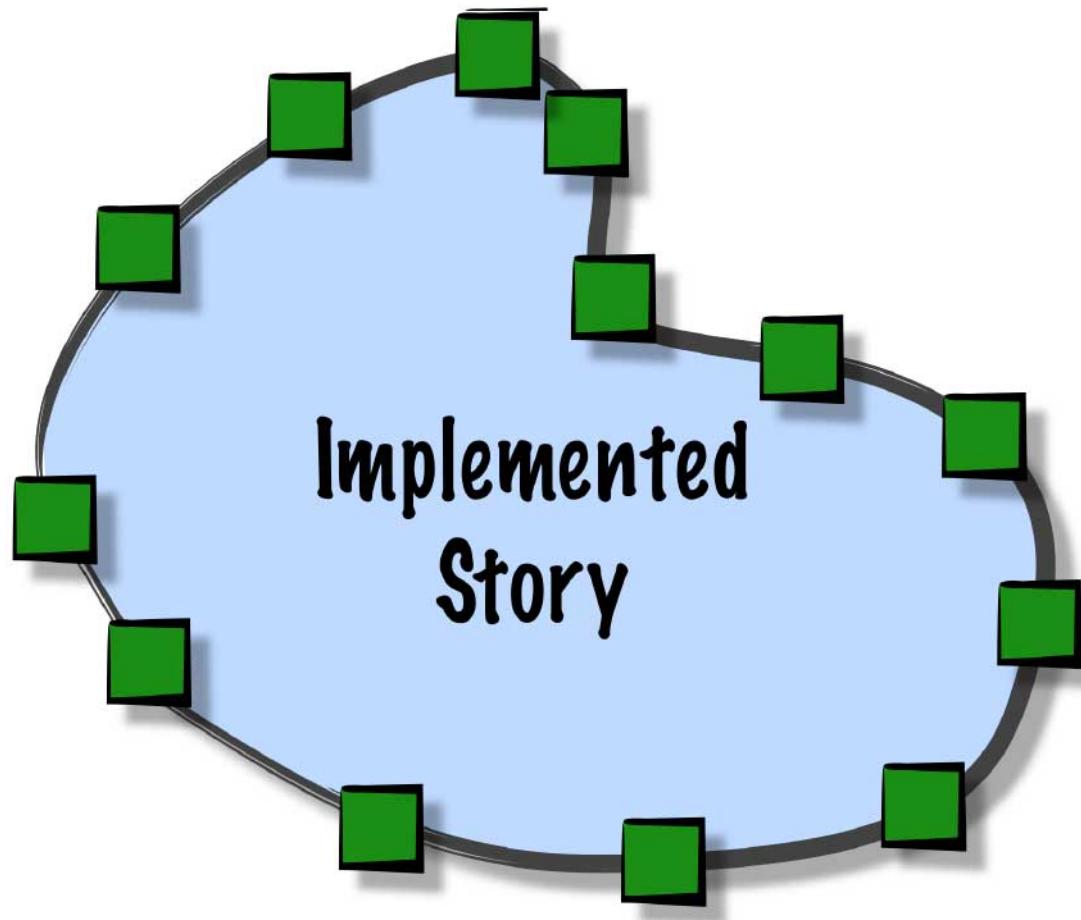
Password	Valid?
"p@ssw0rd"	Yes
"p@s5"	No
"passw0rd"	No
"p@ssword"	No
"@#\$.%1234"	No



Why ATDD?

Make progress visible

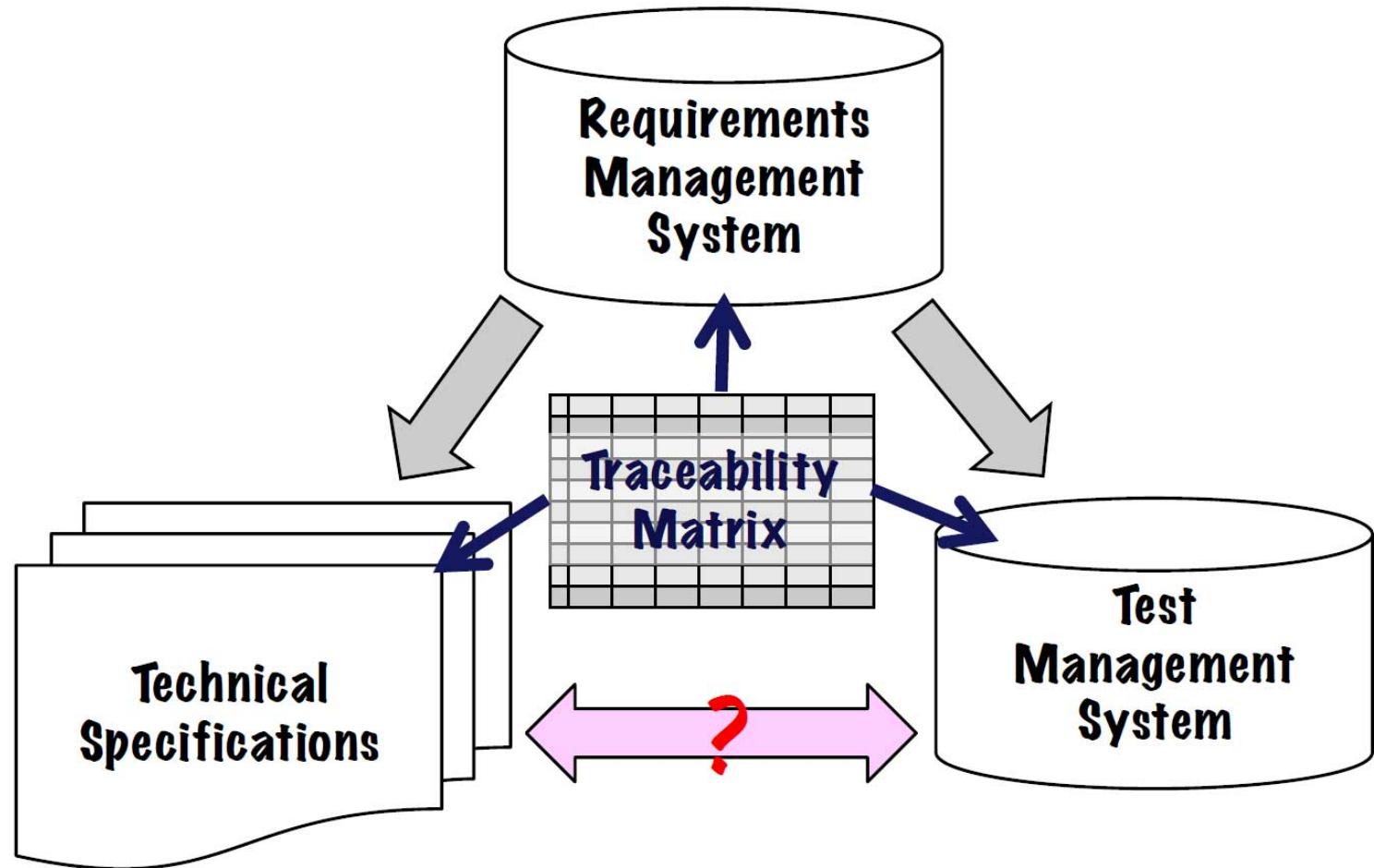
Are We There Yet?



Why ATDD?

Leverage, Efficiency, and Executable Specifications

Traditional Approaches



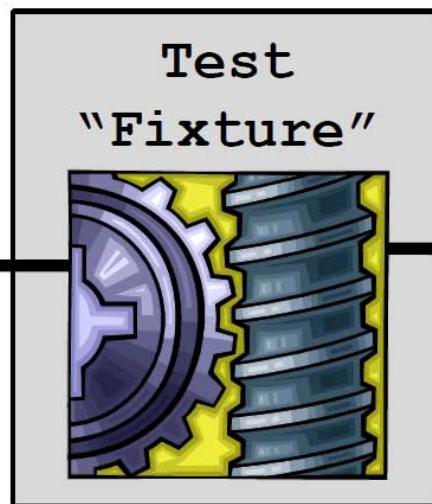
Efficiency, Reusability, Maintainability

The tests
define the
requirements.

Natural
Language
Expectations

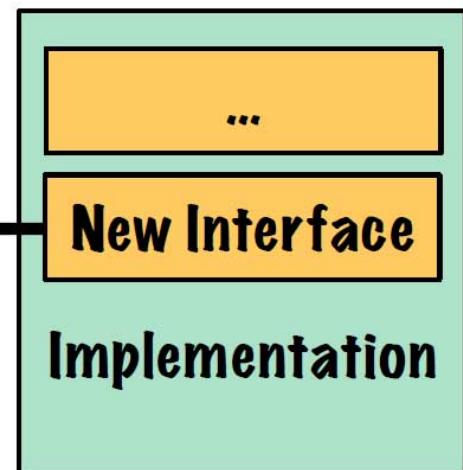
*(No reconciling
multiple, duplicate
artifacts.)*

Implementation
change? Make a
localized update to
the test fixture.



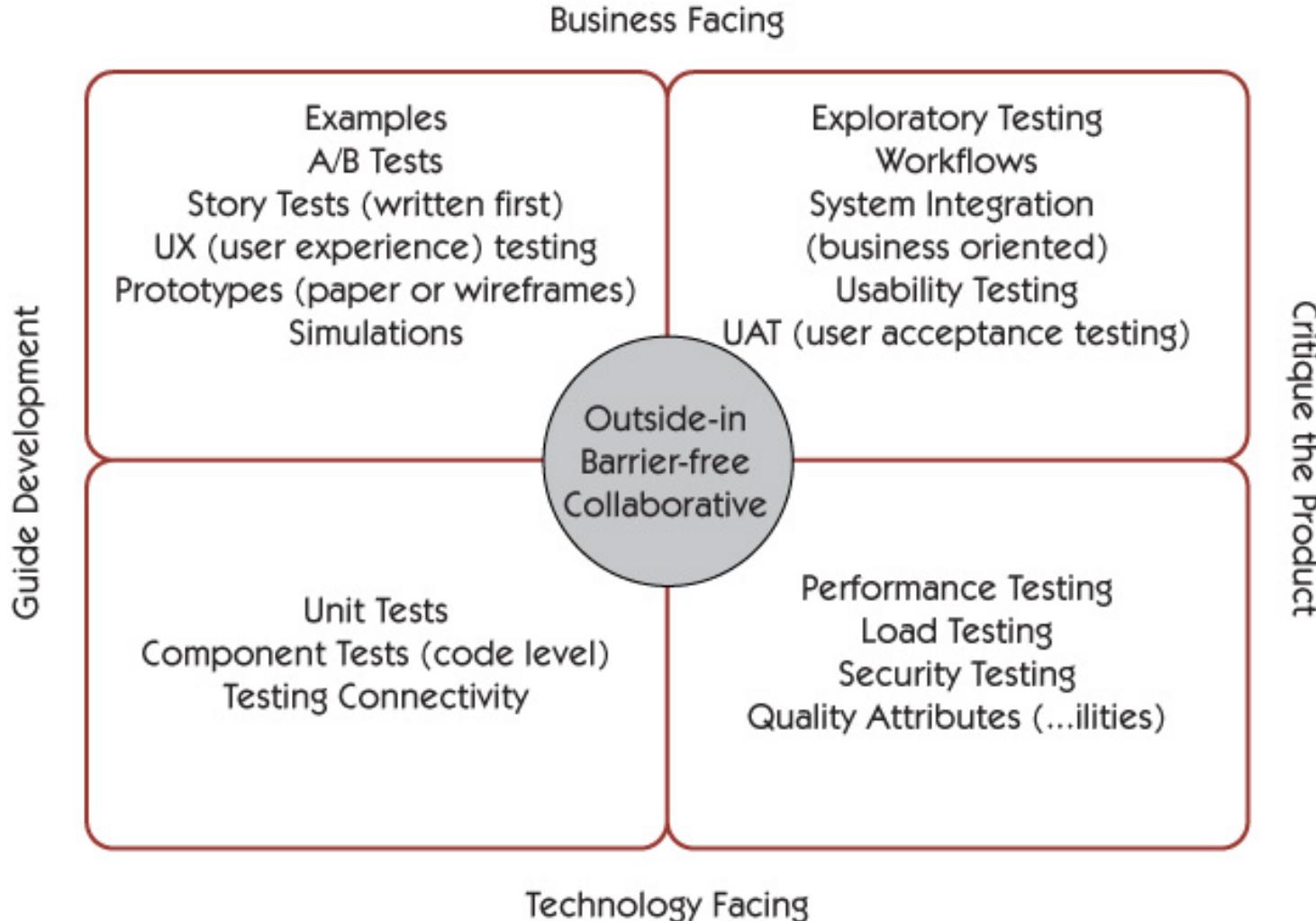
*(Preserve valid
expectations.)*

New interface?
Just add a test
fixture

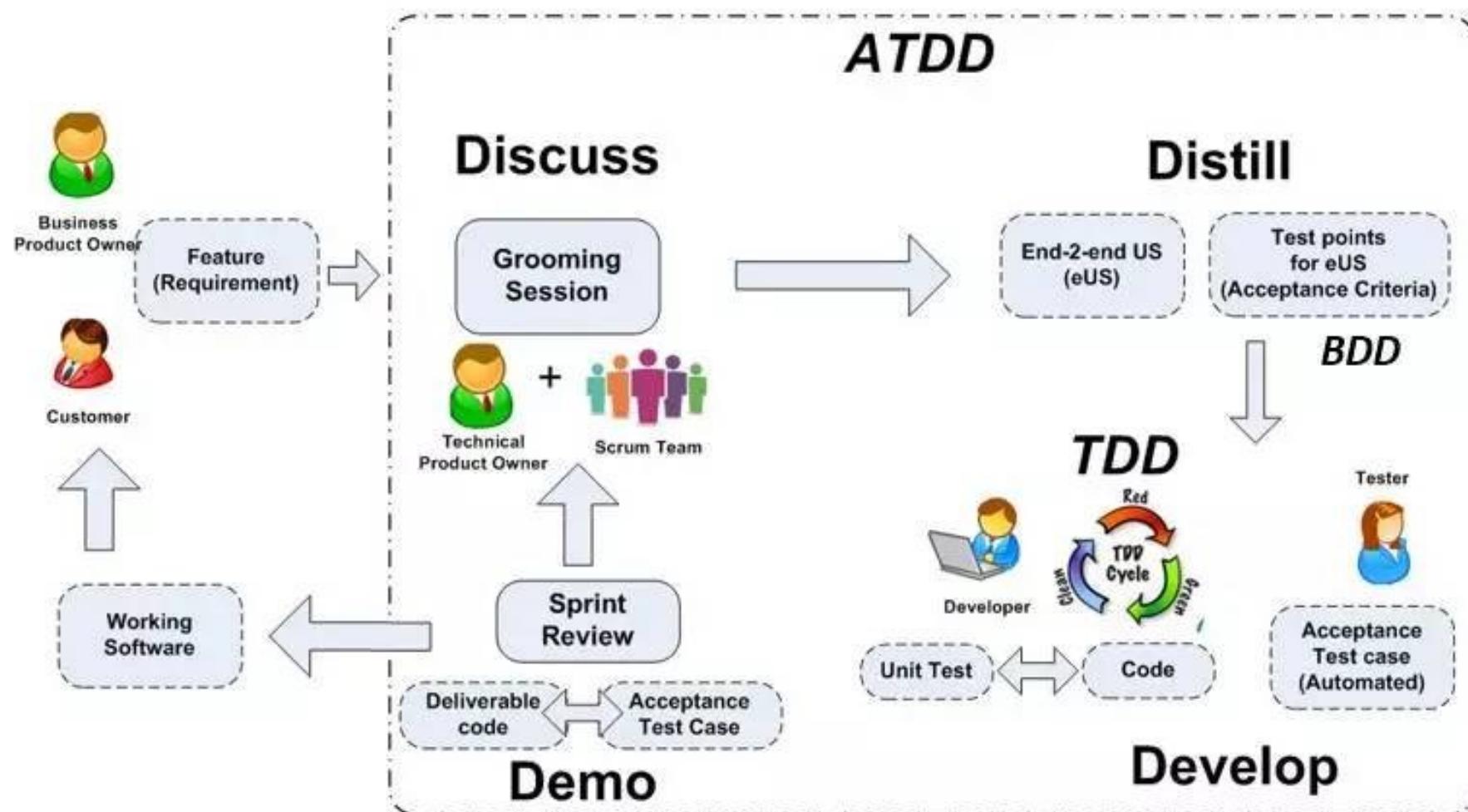


*(Leverage relevant
expectations.)*

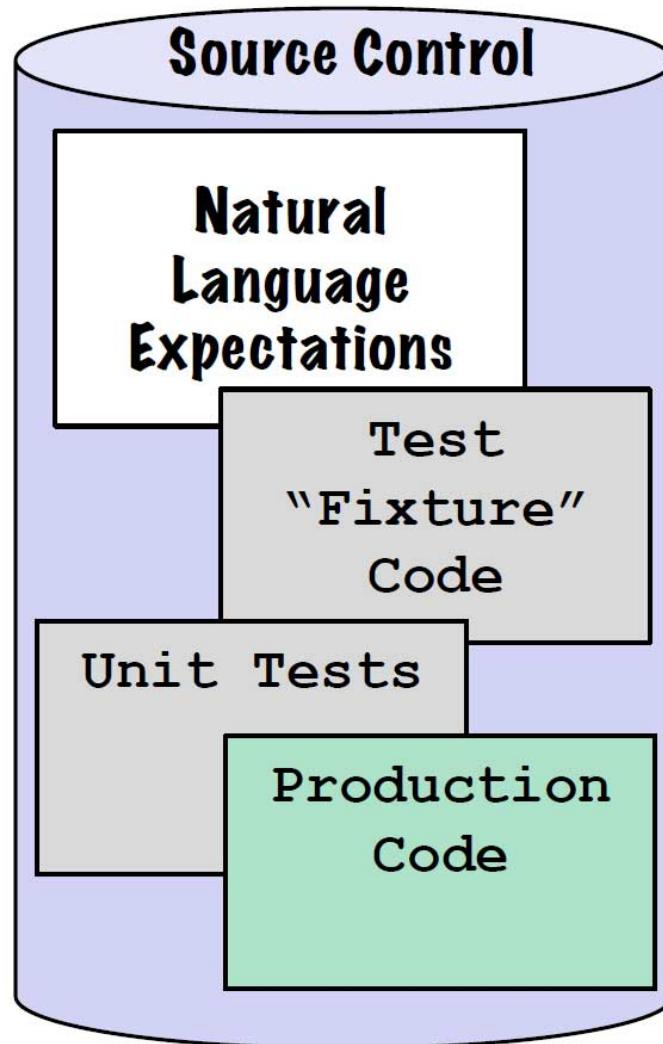
ATDD fitness with other processes



ATDD + BDD+TDD



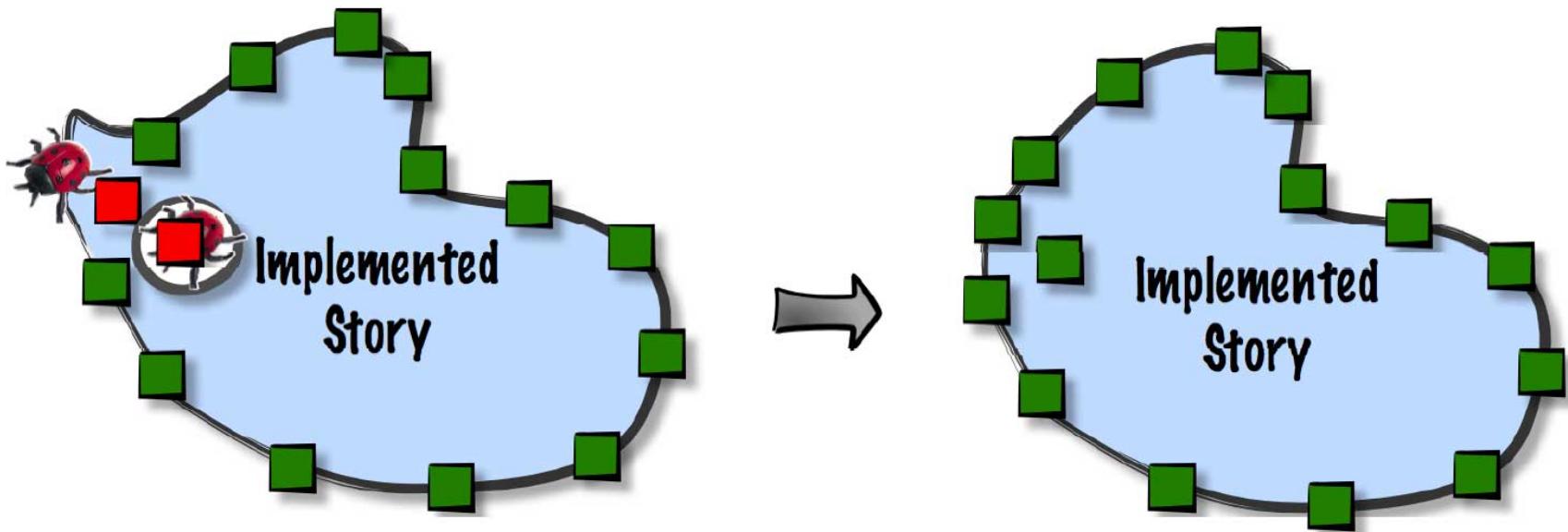
Tests are Versioned with the Code



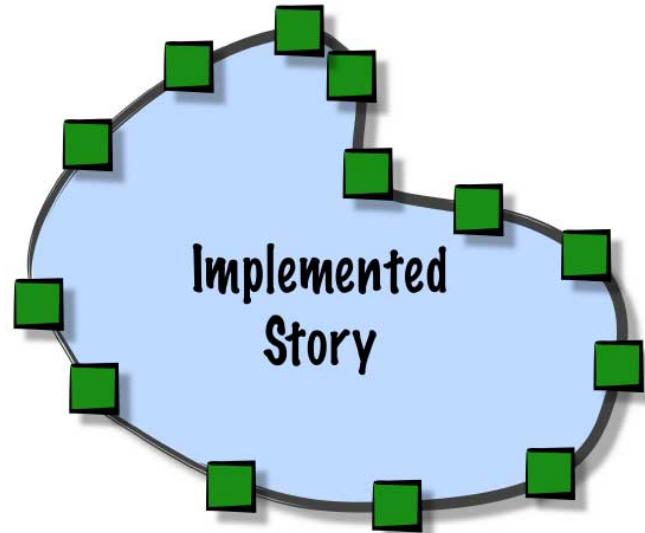
Why ATDD?

No more bugs !!

Zero Tolerance for Bugs



But Not Everything is a Bug



*In this context,
a **BUG**
is behavior that
violates the letter
or spirit of the
Product Owner's
expectations for the
implemented story.*



If the behavior does not violate expectations related to the implemented stories, it's an item for the backlog.

Selenium

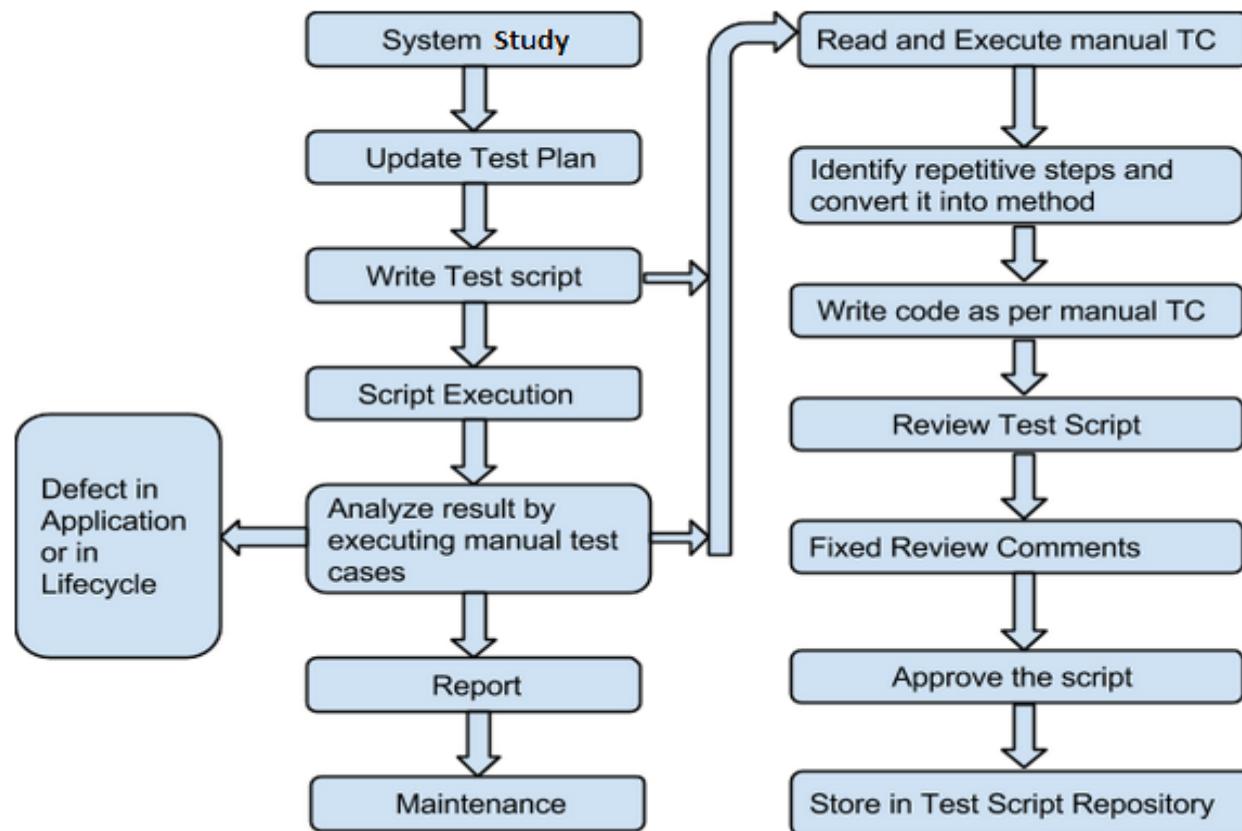
Why automate testing?

Test automation has specific advantages for improving the long-term efficiency of a software team's testing processes. Test automation supports:

- Frequent regression testing
- Rapid feedback to developers
- Virtually unlimited iterations of test case execution
- Support for Agile and extreme development methodologies
- Disciplined documentation of test cases
- Customized defect reporting
- Finding defects missed by manual testing

Automation Test Life Cycle

Automation Test Life Cycle:



History

- Developed in 2004 by Jason Huggins as a JavaScript library used to automate his manual testing routines
- Selenium Core is born whose functionality underlies the Selenium RC (Remote Control) and Selenium IDE tools
- The Limitation of having a JavaScript based automation engine and browser security restricted Selenium to specific functionality
- Google, who has been a long time user of Selenium, had a developer named Simon Stewart who developed WebDriver. This tool circumvented Selenium's JavaScript sandbox to allow it to communicate with the Browser and Operating System directly using native methods
- In 2008, Selenium and WebDriver merged technologies and intellectual intelligence to provide the best possible test automation framework

Introduction

- Selenium is a suite of testing automation tools used for Web-Base applications: Selenium IDE, Selenium RC, Selenium WebDriver and Selenium Grid
- These tools provide a rich set of testing functions specifically geared to varied testing scenarios of all types of Web applications
- The operations provided by these tools are highly flexible and afford many options for comparing UI elements to expected application behavior
- Selenium tests can be executed on multiple browser platforms

Why Use/Learn Selenium

- Increases you marketability
- Has a lot of Java planks
- Growing Industry standard
- Assist with the deployment of defective-free code
- Open source, web-based testing automation tool and cross-browser compliant
- Multi-language backend support (Java, Ruby, Python, C#, PHP, etc...)

Selenium Tools

- Selenium IDE

- Rapid prototyping tool for building test scripts
- Firefox plugin
- Can be used by developers with little to no programming experience to write simple tests quickly and gain familiarity with the Selenese commands
- Has a recording feature that records a user's live actions that can be exported in one of many programming languages
- Does not provide iteration or conditional statements for test scripts
- Can only run tests against FireFox
- Developed tests can be run against other browsers, using a simple command-line interface that invokes the Selenium RC server
- Can export WebDriver or Remote Control scripts (these scripts should be in PageObject structure)
- Allows you the option to select a language for saving and displaying test cases

Selenium Tools

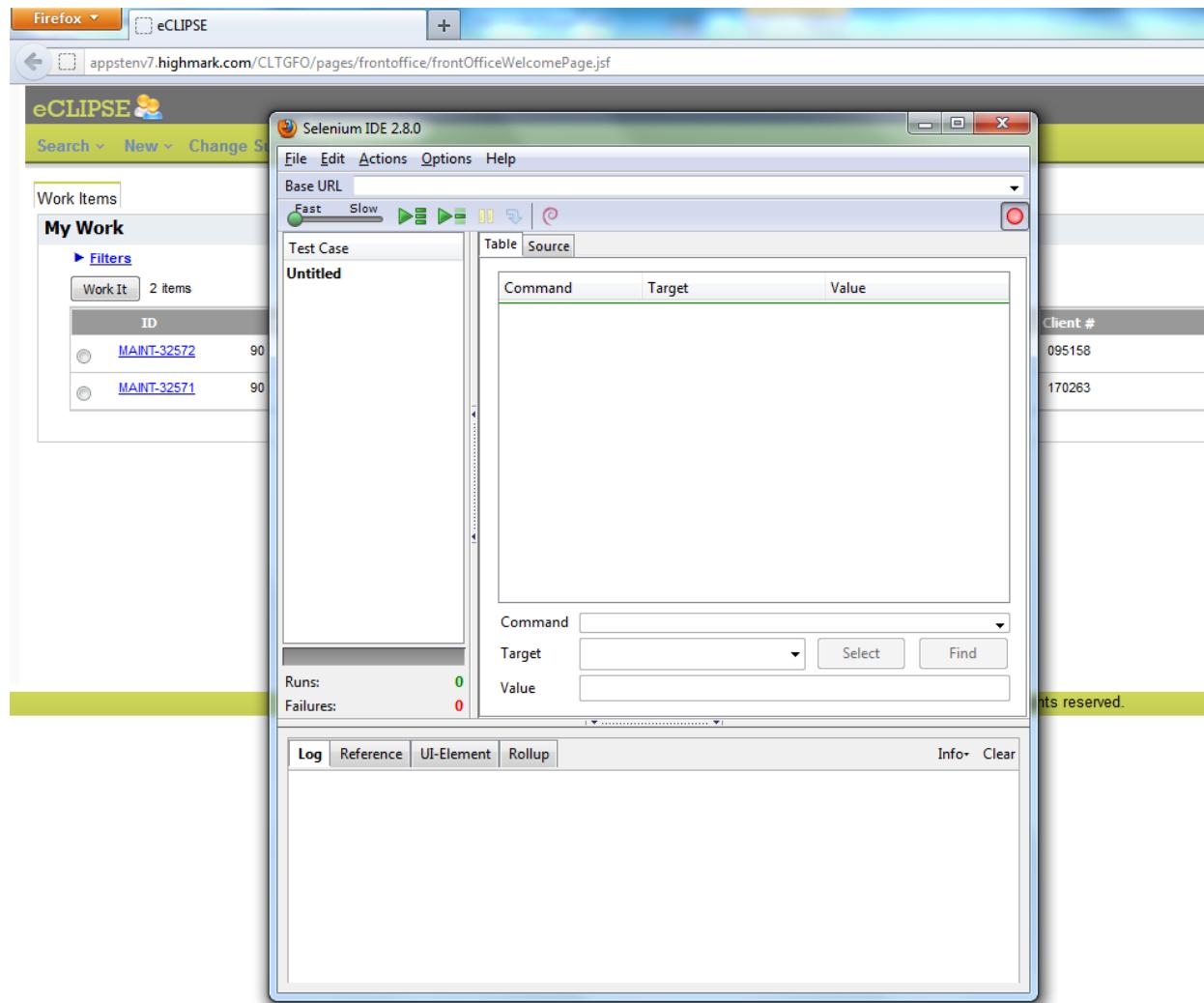
- Selenium RC aka Selenium 1
 - It ‘inject’ JavaScript functions into the browser when the browser is loaded and then uses its JavaScript to drive the AUT within the browser
 - Mainly supported in maintenance mode
 - Provides support for several programming languages
- Selenium WebDriver
 - Designed to provide a simpler, more concise programming interface in addition to addressing some limitations in the Selenium-RC API
 - Developed to better support dynamic web pages where elements of a page may change without the page itself being reloaded
 - Makes direct calls to the browser using each browser’s native support for automation.
 - Has the Selenium 1 (aka Selenium RC) underlying technology for flexibility and Portability
 - [*Migrating From Selenium RC to Selenium WebDriver*](#)
 - Not tied to any particular test framework, so it can be used equally well in unit testing or from a plain old “main” method.

Selenium Tools

- **Selenium Grid**

- Scales the Selenium RC solution for large test suites and test that must be run in multiple environments
- Tests can be run in parallel with simultaneous execution (different tests on different remote machines)
- It allows for running your tests in a *distributed test execution* environment
- Used to run your tests against multiple browsers, multiple versions of browser, and browsers running on different operating systems
- It reduces the time it takes for the test suite to complete a test pass

Selenium IDE



Selenium IDE

- Selenese Commands
 - Action
 - Manipulate the state of the application
 - Used with “AndWait” (clickAndWait)
 - Accessors
 - Examines the application state and stores the results in variables
 - Used to auto generate Assertions
 - Assertions
 - Similar to Accessors but verifies the state of the application to what is expected
 - Modes: assert, verify and waitFor

Selenium IDE

– Syntax

- Has two parameters (both are not required)
- Can view command requirements from the command reference tab

– Parameters

- Locators identify a UI Element on a page
- Test Patterns are used for asserting or verifying
- Selenium variable or Text Patterns that can be entered in input fields or drop down selections

Selenium IDE

- Selenese Command (continued)
 - AndWait
 - Tells Selenium to wait for the page to load after an action has been performed
 - Used when triggering navigation/page refresh (test will fail otherwise)
 - Command: clickAndWait
 - WaitFor
 - No set time period
 - Dynamically waits for the desired condition, checking every second
 - Commands: waitForElementPresent, waitForVisible, etc...
 - Echo
 - Used to display information progress notes that is displayed to the console during test execution
 - Informational notes can be used to provide context within your test results report
 - Used to print the contents of Selenium variables

Selenium IDE

- Store Commands and Selenium Variables
 - Selenium variables can be used to store constants at the beginning of scripts and values passed to a test script from the command-line, another program or file
 - Store Command
 - Two parameters (text value and Selenium variable)
 - Uses \${} to access the value of a variable
 - Can be used in the first or second parameter or within a locator expression
 - Other Store Commands: storeElementPresent, storeText, storeEval, ect...

Selenium IDE

- JavaScript and Selenese Parameters
 - JavaScript uses script and non-script Selenese parameters
 - Parameters can be accessed via a JavaScript associative array “storedVars[]”
 - Script Parameters
 - Specified by assertEval, verifyEval, storeEval and waitForEval
 - A snippet of JavaScript code is placed into the appropriate field, normally the Target field (script parameters are usually the first or only parameter)
 - Non-Script Parameters
 - Uses JavaScript to generate values for elements with non-script parameters
 - JavaScript must be enclosed in curly braces and preceded by the label “javascript”

Selenium IDE

- Commonly Used Selenium Commands
 - open
 - click/clickAndWait
 - verifyTitle/assertTitle
 - verifyTextPresent
 - verifyElementPresent
 - verifyText
 - verifyTable
 - waitForPageToLoad
 - waitForElementPresent

Selenium IDE

- Locators
 - By Identifier
 - Used by default
 - Locator type is “identifier”
 - First element with id attribute value matching the location will be used
 - First element with a name attribute matching the location will be used if there are no id matches
 - By ID
 - More limited than the “identifier” type
 - Locator type is “id”
 - Use this type when you know the element’s id
 - By Name
 - Locates an element with a matching name attribute
 - Filters can be applied for elements with the same name attribute
 - Locator type is “name”

Selenium IDE

– X-Path

- Used for locating nodes in an XML document
- Elements can be located in regards to absolute terms or a relative position to an element that has a specified id or name attribute
- Can locate elements via attributes other than id or name
- Starts with “//”

– By DOM

- Can be accessed using Javascript
- Locator type is “document”

– By CSS

- Uses the style binding of selectors to elements in a document as a locating strategy
- Faster than X-Path and can find the most complicated objects in an intrinsic HTML document
- Locator type is “css”

Selenium IDE

- Matching Patterns
 - Text Patterns
 - A parameter required by following Selenese commands: verifyText, verifyText, verifyTitle, verifyAlert, assertConfirmation, verifyPrompt, ect...
 - Globbing Patterns
 - Pattern matching based on wild card characters (*, [], -)
 - Uses the “glob:” label
 - Default pattern matching scheme
 - Regular Expressions
 - The most powerful pattern matching scheme
 - Prefixed with “regexp:” label
 - Exact Patterns
 - Uses no special characters, no need to escape characters
 - Prefixed with “exact:” label

Selenium IDE

- Alerts, Pop-ups and Multiple Windows
- In Selenium, JavaScript alert and confirmation pop-ups will not appear, they are overridden at runtime by Selenium's own JavaScript
- Alert pop-ups, however, still have a presence and would need to be asserted with one of the various assertFoo functions (assertFoo(pattern), assertFooPresent(), assertFooNotPresent(), storeFoo(variable), storeFooPresent(variable), ect...)
- Confirmation pop-ups select “Ok” by default and use assertConfirmation, assertConfirmationPresent, ect.. functions

Selenium IDE

- Debugging and Start Points
- Set a debug start point by right-clicking a command and toggle “break point/start point”
- The Find button highlights the currently selected UI element on the displayed page. From the Table view, select any command that has a locator parameter and click the Find button
- To view portions of the Page Source, select the respective portion of the web page, right-click, select view selection source
- In recording a locator-type argument, Selenium IDE stores additional information that presents the user with alternative locator-type arguments

Selenium IDE

- User Extensions
- JavaScript files created for customizations and features to add additional functionality to Selenium IDE
- For Flow Control, install the `goto_sel_ide.js` extension

Selenium IDE

- Java Test Script Example

```
public void testGoogleTestSearch() throws Exception
```

```
{
```

```
    selenium.open("http://www.google.com/webhp");
    assertEquals("Google", selenium.getTitle());
    selenium.type("q", "Selenium OpenQA");
    selenium.click("btnG");
    selenium.waitForPageToLoad("5000");
    assertEquals("Selenium OpenQA - Google Search",
    selenium.getTitle());
```

```
}
```

Selenium WebDriver

- Project Setup
 - Java
 - The easiest way is use Maven. Maven will download the java bindings (the Selenium 2.0 java client library) and all its dependencies, and will create the project for you, using a maven pom.xml (project configuration) file
 - You can then import the maven project into your preferred IDE, IntelliJ IDEA or Eclipse.
 - From a command-line, CD into the project directory and run maven as follows: mvn clean install

Selenium WebDriver

Commands and Operations

To fetch a page you would use the “get” command

```
driver.get("http://www.google.com");
```

Locating UI Elements

Language bindings expose a “findElement” and “Find Elements” method

The “Find” methods take a locator or query object called “By”

```
WebElement element= driver.findElement(By.id("coolestWidgetEvah"));
List<WebElement>cheeses = driver.findElements(By.className("cheese"));
WebElement frame = driver.findElement(By.tagName("iframe"));
WebElement cheese = driver.findElement(By.name("cheese"));
WebElement cheese = driver.findElement(By.linkText("cheese"));
WebElement cheese = driver.findElement(By.partialLinkText("cheese"));

Web Element cheese =
driver.findElement(By.cssSelector("#food.span.dairy.aged")) List<WebElement>
inputs = driver.findElements(By.xpath("//input"));

WebElement element = (WebElement)
((JavascriptExecutor)driver).executeScript("return $('.cheese')[0]");
```

- Input and Navigation
- Select select = new Select(driver.findElement(By.tagName("select")));
select.deselectAll(); select.selectByVisibleText("Edam");
- driver.findElement(By.id("submit")).click();
- driver.switchTo().window("windowName");
- for (String handle : driver.getWindowHandles()){
driver.switchTo().window(handle); }
- driver.switchTo().frame("frameName");
- Alert alert = driver.switchTo().alert();
- driver.navigate().to("http://www.example.com");
- driver.navigate().forward(); driver.navigate().back();

Selenium WebDriver

- Page Objects
 - OO Library that separates test code into a MVC pattern bringing OOP to test scripts
 - Language neutral pattern for representing a complete page or position of a page in an OO manner
 - Requires Language specific coding
 - Used for maintenance, script cascading, enhanced script readability/functionality

Selenium WebDriver

- Scripts and Page Objects
 - Scripts are more procedural while Page Objects are detail oriented
 - Locators appear once in all Page Objects of a page and do not cross Page Object boundaries
 - Uses Elements, Actions and Synchronization
 - Order of Operation
 - Locator
 - Element Implementation
 - Add Elements to Page Objects
 - Actions

Selenium WebDriver

- Do not create the Page Object all at once, build test incrementally
- Scripts Should
 - Not contain any synchronization code
 - Not contain any Driver API calls (promotes changes to Selenium or other technology without changing the scripts)
 - Has asserts (determination of results)

Selenium WebDriver

- Driver Implementations
 - HtmlUnitDriver
 - The fastest and most lightweight implementation of WebDriver
 - HtmlUnit is a java based implementation of a WebBrowser without a GUI
 - For any language binding (other than java) the Selenium Server is required to use this driver
 - A pure Java solution and so it is platform independent
 - Supports JavaScript but emulates other browsers' JavaScript behaviour
 - FireFox Driver
 - Controls the Firefox browser using a Firefox plugin
 - Runs in a real browser and supports JavaScript
 - Faster than the Internet Explorer Driver but slower than HtmlUnitDriver

Selenium WebDriver

– Internet Explorer Driver

- This driver is controlled by a .dll files and is thus only available on Windows OS
- Each Selenium release has its core functionality tested against versions 6, 7 and 8 on XP, and 9 on Windows7
- Runs in a real browser and supports JavaScript
- XPath is not natively supported in most versions
- CSS is not natively supported in versions 6 and 7
- CSS selectors in IE 8 and 9 are native, but those browsers don't fully support CSS3

Selenium WebDriver

- Driver Implementation
 - Chrome Driver
 - Chrome Driver is maintained / supported by the Chromium Project
 - WebDriver works with Chrome through the chromedriver binary (found on the chromium project's download page)
 - Runs in a real browser and supports JavaScript
 - Because Chrome is a Webkit-based browser, the Chrome Driver may allow you to verify that your site works in Safari. Note that since Chrome uses its own V8 JavaScript engine rather than Safari's Nitro engine, JavaScript execution may differ
 - Slower than the HtmlUnit Driver

Selenium WebDriver

- Opera Driver
 - See the [Opera Driver wiki article](#) in the Selenium Wiki for information on using the Opera Driver
- iOS Driver
 - See either the [ios-driver](#) or [appium](#) projects
- Android Driver
 - See the [Selendroid project](#)

Selenium WebDriver

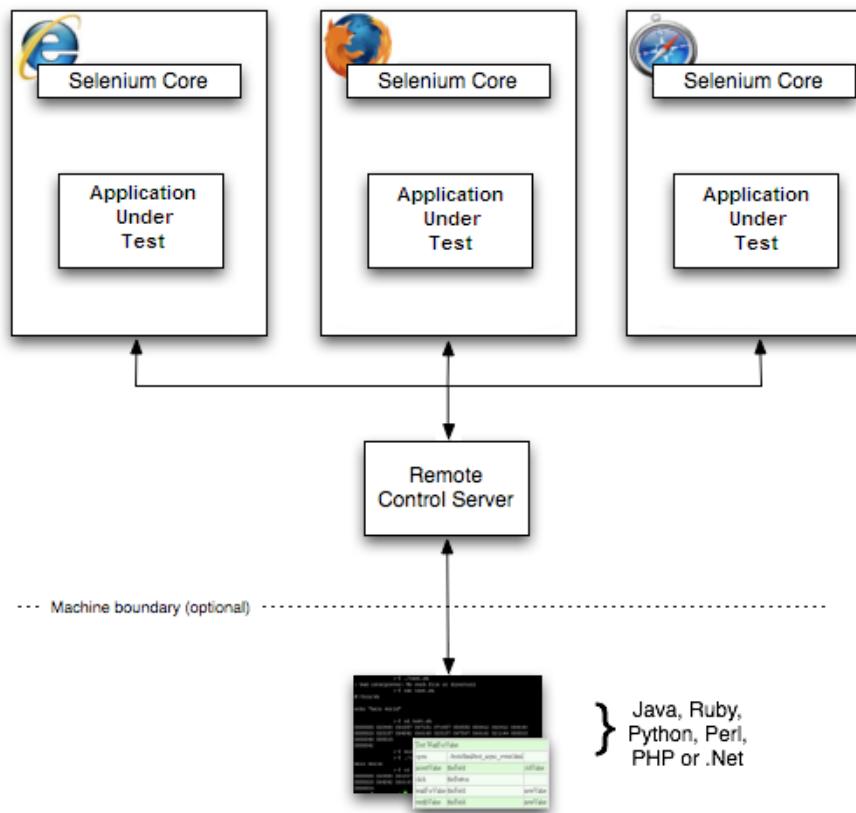
```
const {Builder, By, until} = require('selenium-webdriver');
|
(async function example() {
  let driver = await new Builder().forBrowser('chrome').build();
  try {
    await driver.get('https://www.seleniumhq.org/');
    await driver.findElement(By.id('q')).sendKeys('webdriver');
    await driver.findElement(By.id('submit')).click();
    await driver.wait(until.titleIs('webdriver - Google Search'), 1000);
  } finally {
    await driver.quit();
  }
})();
```

Selenium RC

- In Selenium RC, the Selenium Server launches and kills browsers, interprets and runs the Selenese commands passed from the test program, and acts as an *HTTP proxy*, intercepting and verifying HTTP messages passed between the browser and the AUT
- Client libraries which provide the interface between each programming language and the Selenium RC Server.
- The primary task for using Selenium RC is to convert your Selenese into a programming language

Selenium RC

Windows, Linux, or Mac (as appropriate)...



Selenium RC

Sample Test Script

open	/	
type	q	selenium rc
clickAndWait	btnG	
assertTextPresent	Results *	for selenium rc

Selenium RC

```
/** Add JUnit framework to your classpath if not already there
 * for this example to work
 */
package com.example.tests;

import com.thoughtworks.selenium.*;
import java.util.regex.Pattern;

public class NewTest extends SeleneseTestCase {
    public void setUp() throws Exception {
        setUp("http://www.google.com/", "*firefox");
    }
    public void testNew() throws Exception {
        selenium.open("/");
        selenium.type("q", "selenium rc");
        selenium.click("btnG");
        selenium.waitForPageToLoad("30000");
        assertTrue(selenium.isTextPresent("Results * for selenium rc"));
    }
}
```

Selenium RC

- Learning the API
 - `setUp("http://www.google.com/", "*firefox");`
 - The Browser is manipulated by a Browser Instance that is assigned to a program variable
 - This program variable is then used to call methods from the browser
 - Selenese commands are then ran by calling the respective methods from the browser variable - `selenium.type("field-id","string to type")`
 - To utilize iteration and conditional logic, Selenium RC uses program language specific methods in conjunction with Selenese commands
 - Use the **getEval** method of selenium API to execute JavaScript from selenium RC

Selenium RC

- Server, Security and Browsers Configurations
- Command line options can be used to change the default server behaviour.
- If your AUT is behind an HTTP proxy which requires authentication then you should configure http.proxyHost, http.proxyPort, http.proxyUser and http.proxyPassword
- You can run Selenese html files directly within the Selenium Server by passing the html file to the server's command line
- When launching selenium server the **-log** option can be used to record valuable debugging information reported by the Selenium Server to a text file
- When dealing with HTTPS in a Selenium RC test, there is a run mode that supports handling security pop-ups and processes the security certificate for you
- When a browser is not directly supported, you may still run your Selenium tests against a browser of your choice by using the “*custom” run-mode (i.e. in place of *firefox or *iexplore) when your test application starts the browser.

Selenium Grid

- A Grid consists of a single Hub, and one or more Nodes. Both are started using a selenium-server.jar executable.
- The Hub receives a test to be executed along with information on which browser and ‘platform’ (i.e. WINDOWS, LINUX, etc) where the test should be run.
- Since the Hub knows the configuration for each registered Node, it selects an available Node that has the requested browser-platform combination
- Selenium commands initiated by the test are sent to the Hub, which passes them to the Node assigned to that test
- The Node runs the browser, and executes the Selenium commands within that browser against the application under test

- Selenium-Grid allows the Selenium-RC solution to scale for test suites or test suites to be run in multiple environments.
- With Selenium-Grid multiple instances of Selenium-RC are running on various operating system and browser configurations, each of these when launching register with a hub. When tests are sent to the hub they are then redirected to an available Selenium-RC, which will launch the browser and run the test.
- This allows for running tests in parallel, with the entire test suite theoretically taking only as long to run as the longest individual test.

How to grid

- Download Selenium Grid latest binary distribution and unpack it on your computer.
- Go at the root of selenium grid directory and validate your installation:
 - cd selenium-grid-1.0
 - **ant sanity-check**
- Go to the selenium distribution directory and launch in a new terminal:
 - ant launch-hub
- Check out that the Hub is running by looking at its console in a browser:
 - <http://localhost:4444/console> (See the Browser in the next slide)

File Edit View Favorites Tools Help

Selenium Grid - Run the Demo Selenium Grid Hub Console

Live Search

Selenium Grid Hub

[Documentation](#) | [FAQ](#)

Configured Environments

Target	Browser
*firefox3	*firefox3
Safari on OS X	*safari
*chrome	*chrome
*firefox2	*firefox2
*firefoxproxy	*firefoxproxy
*pifirefox	*pifirefox
Firefox on Windows	*firefox
*iehta	*iehta
*piiexplore	*piiexplore
*iexploreproxy	*iexploreproxy
*opera	*opera
Firefox on OS X	*firefox
*safariproxy	*safariproxy
*firefox	*firefox
*safari	*safari

Available Remote Controls

Host Port Environment

Active Remote Controls

Host Port Environment

How to run

- In a new terminal enter the following command
 - ant launch-remote-control
- Based on your target file you can run either in sequence or in parallel

What's good about RC?

- Relatively easy to automate web UI tests
- Record/Replay for regression tests
- RC allows integration with CI and JUnit/FitNesse tests

What's bad?

- Speed: RC->Browser communication is a speed bottleneck (run in grid, overnight)
- UI is brittle, tests depending on the UI break a lot (DSTL might fix this, page abstractions as well)
- Data-backed tests are not easily repeatable (integrate with DB test engines)

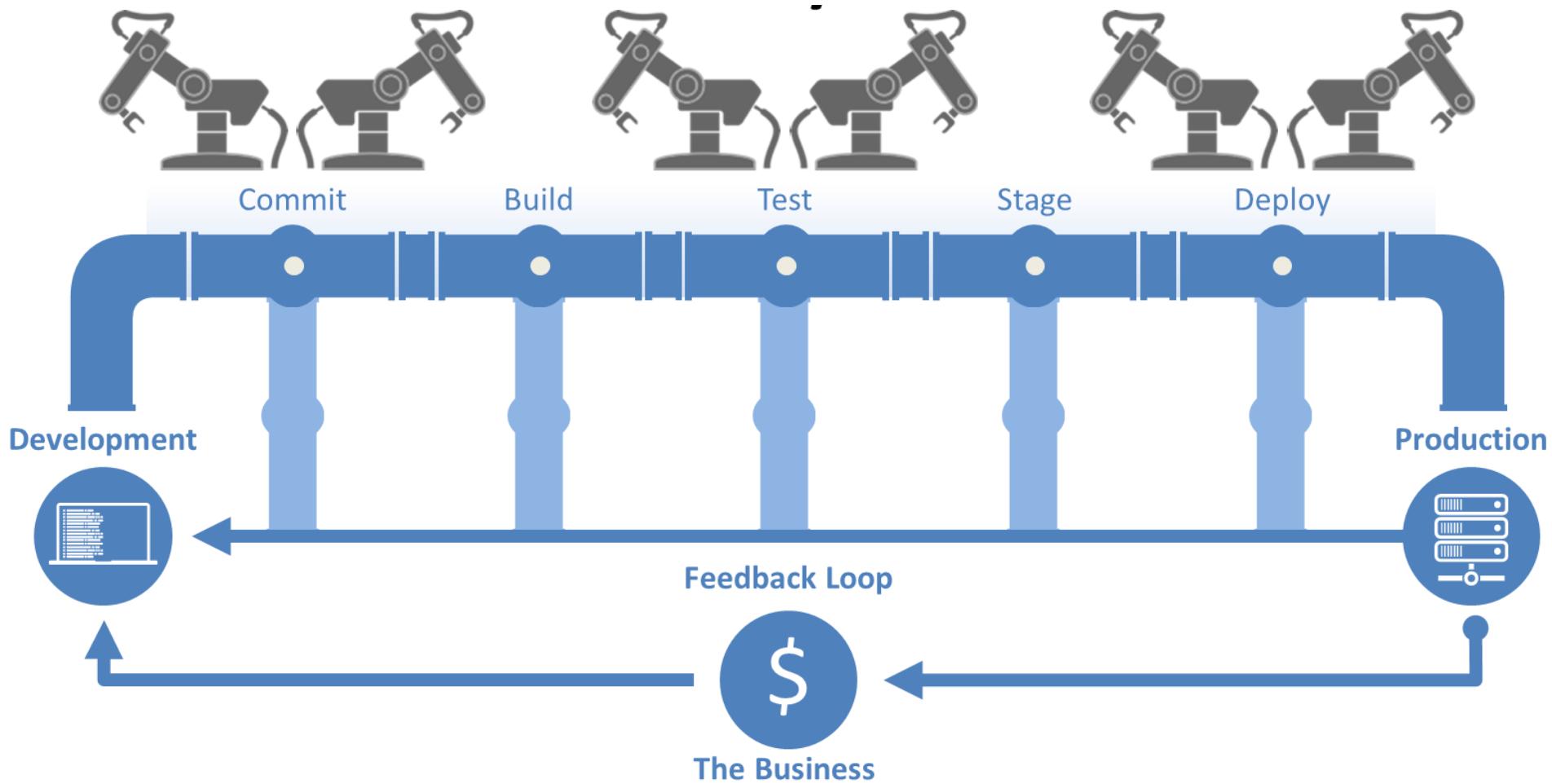
Continuous Integration & Continuous Deployment

The first principle of the Agile Manifesto

*“Our highest priority is to satisfy the customer through early and **continuous delivery** of **valuable software**.”*

<http://www.agilemanifesto.org/principles.html>

Continuous Delivery and Automation are Key



One step at a time

- Source control
- Build automation
- Test automation
- Continuous integration
- Release automation
- Continuous delivery

Source Control

Version control best practices for CI/CD

- Use a descriptive commit message
- Link Commits to Issues (Feature, Bug, Task)
- Make each commit a logical unit
- Incorporate others' changes frequently
- Share your changes frequently
- Don't commit generated files
- Must have single source repository

Build Automation

Make

MSBuild

Ant

Maven

Gradle

Rake

Jenkins Pipeline

Build scripts – single command build

Build script is a single script or set of script which can compile, test, build and deploy software. Using build scripts, we can have a single command build.

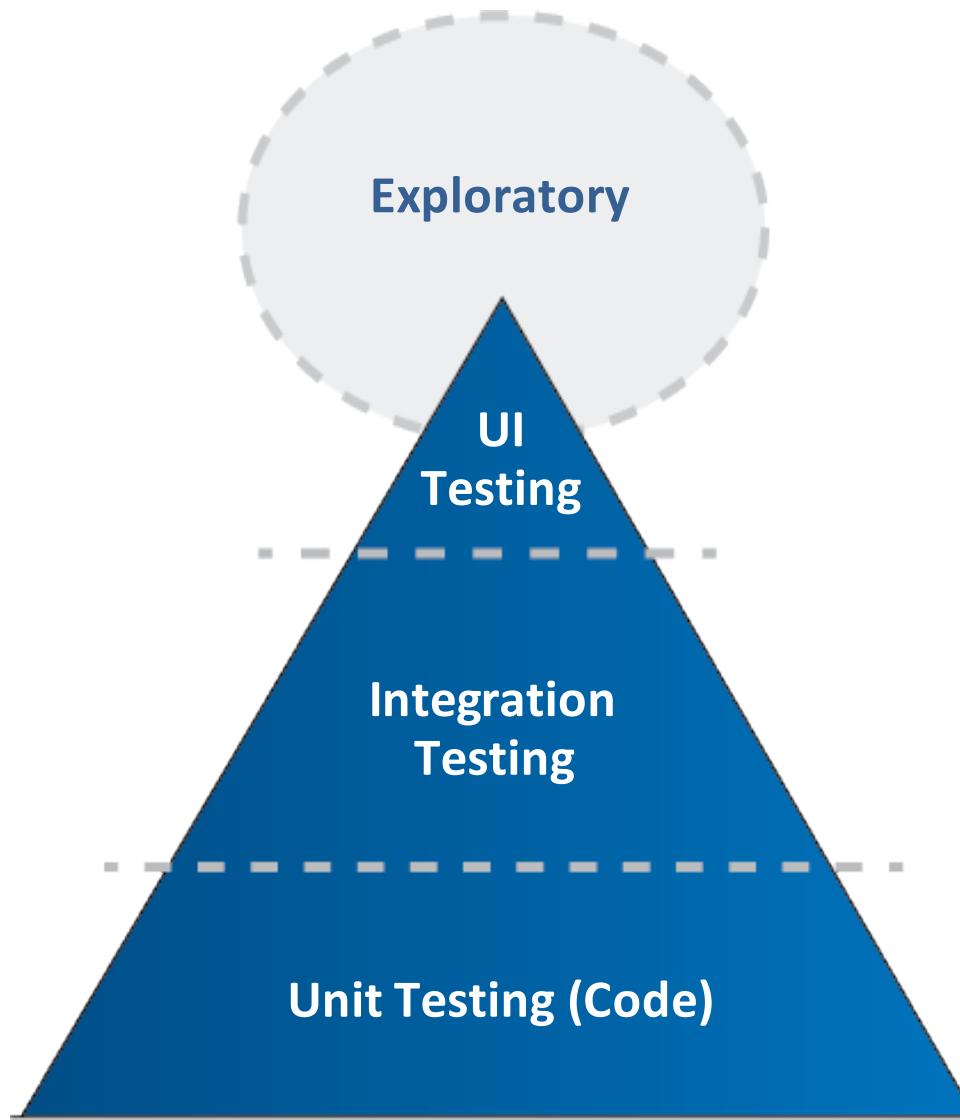
Build tools can work without CI system as well. Apache Ant, Apache Maven, NAnt, MSBuild, Rake, Bazel, Grunt, Gulp, Gradle are examples of build tools.

Most of the IDEs have built in build tools or options to integrate 3rd party build tools.

Build script – let's try some examples

- Apache Ant
- Gulp
- Grunt
- Maven

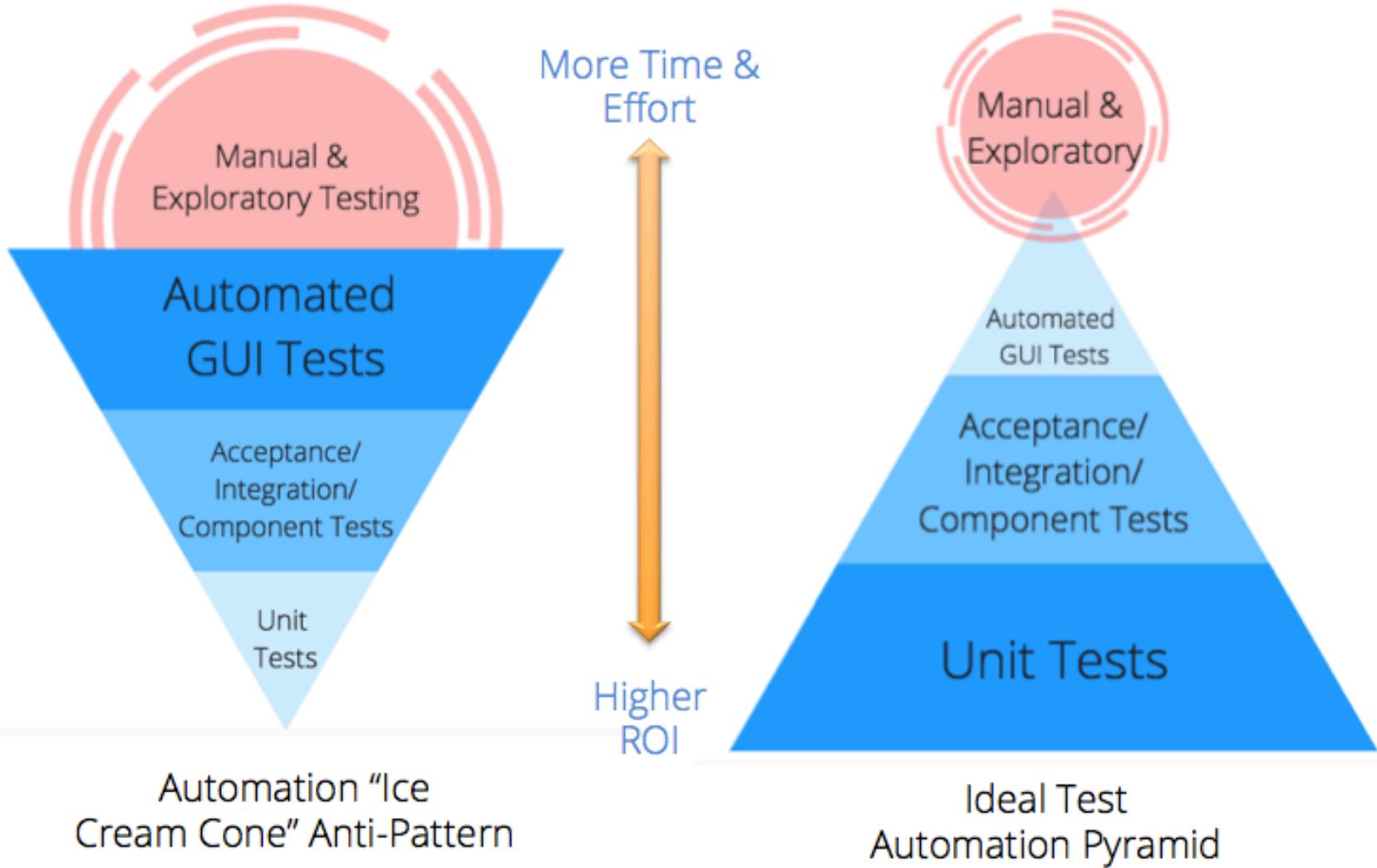
Test Automation



Automating Examples

- Start small
- Select important examples for automation
- Plan up-front to automate
- Be prepared to go slower at the start
- Treat automation code as a first class citizen
- Avoid record and playback
- Avoid using pre-populated data

How automation helps in Agile



Continuous Integration

- Make the build self-testing
- Everyone commits to the baseline every day
- Every commit (to baseline) should be built
- Keep the build fast
- Test in a clone of the production environment
- Make it easy to get the latest deliverables
- Everyone can see the results of the latest build
- Automate deployment

Release Automation

- Produce Artifacts once and store them in an artifact repository
- Promote these artifacts rather than recreating them at each step.
- Use a configuration management tool like Chef, Puppet, or Ansible to manage your environments
- Treat your infrastructure like code and include your playbooks and configuration management modules with your code repository

Continuous Delivery

- Software is confirmed to be in a shippable state.
- Stakeholders have instant visibility into production readiness.
- Teams can perform push-button deployments on-demand.

<http://martinfowler.com/bliki/ContinuousDelivery.html>

Jenkins Pipeline (Workflow)

Single Job - Entire CD Pipeline in a single workflow (or job)

Complex logic - Support complex logic like for-loops,
if-then-else, try-catch, fork-join etc...

Survive restarts - While the workflow is running

Human approval/input - Integrated human interaction into the
Workflow

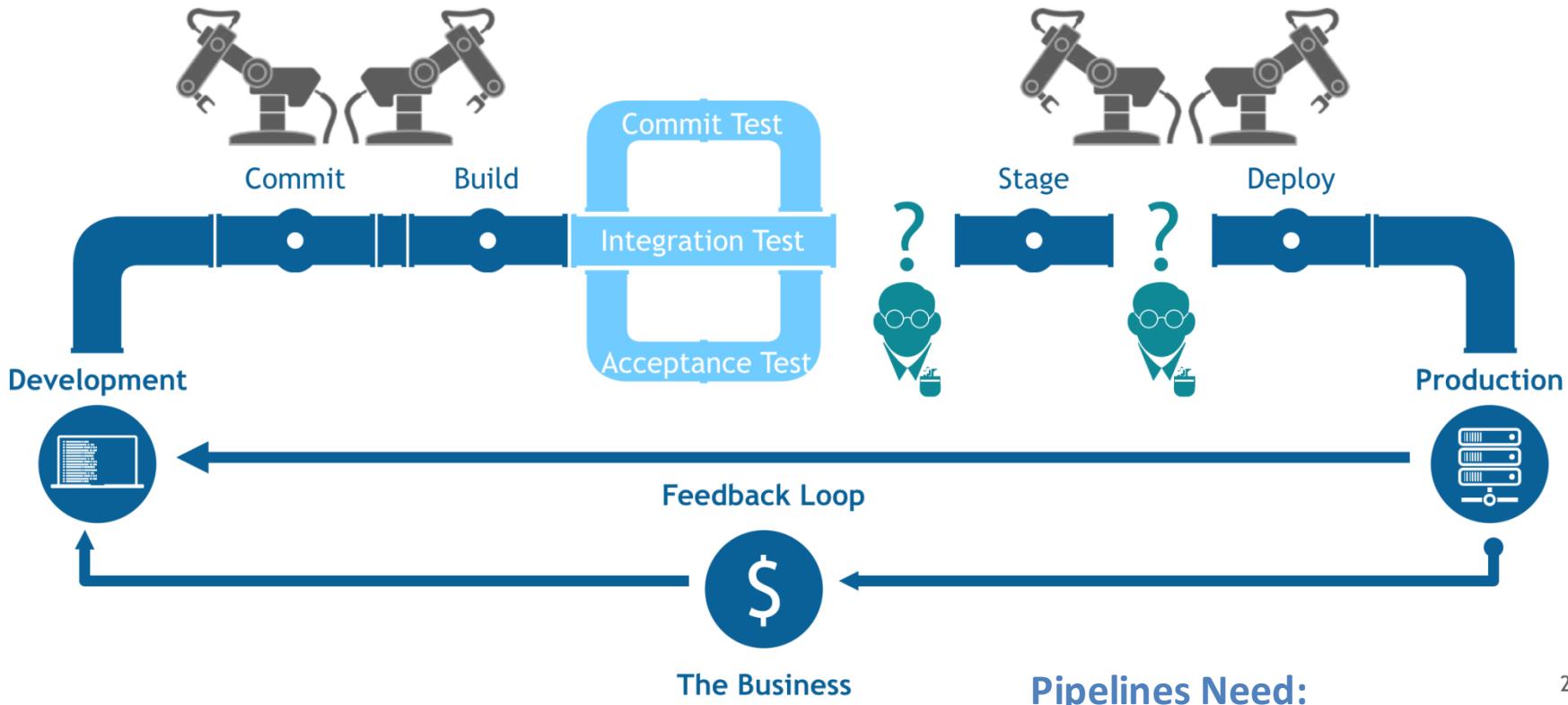
Allocate resources - Dynamically allocate slaves and workspaces

Versioning - Supports checking workflow into version control system

Checkpoints - CloudBees feature to be able to resume from a safe point

Visualization - Stage view of workflow

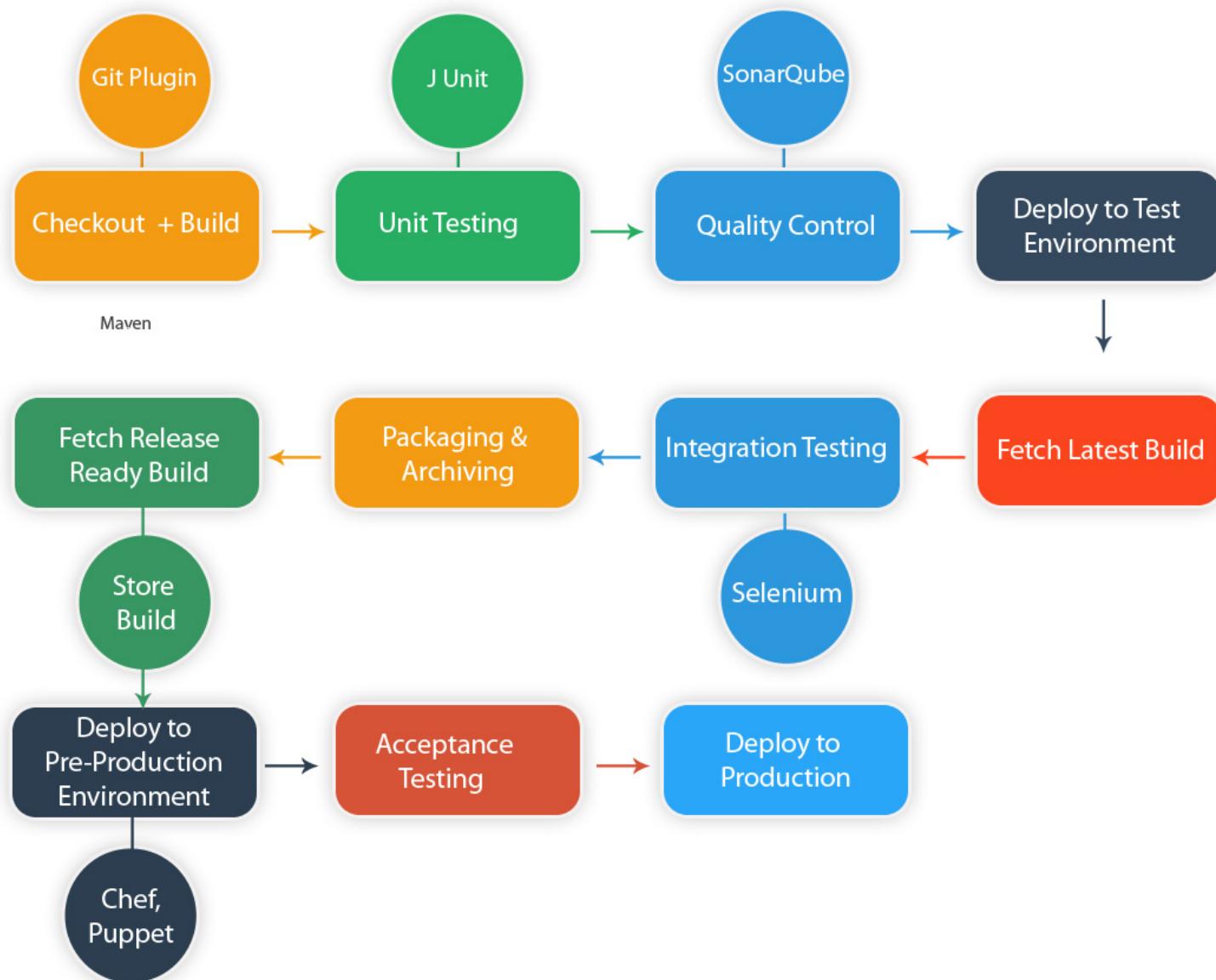
Jenkins Powered CD Pipelines



Pipelines Need:

- ✓ Branching
- ✓ Looping
- ✓ Restarts
- ✓ Checkpoints
- ✓ Manual Input

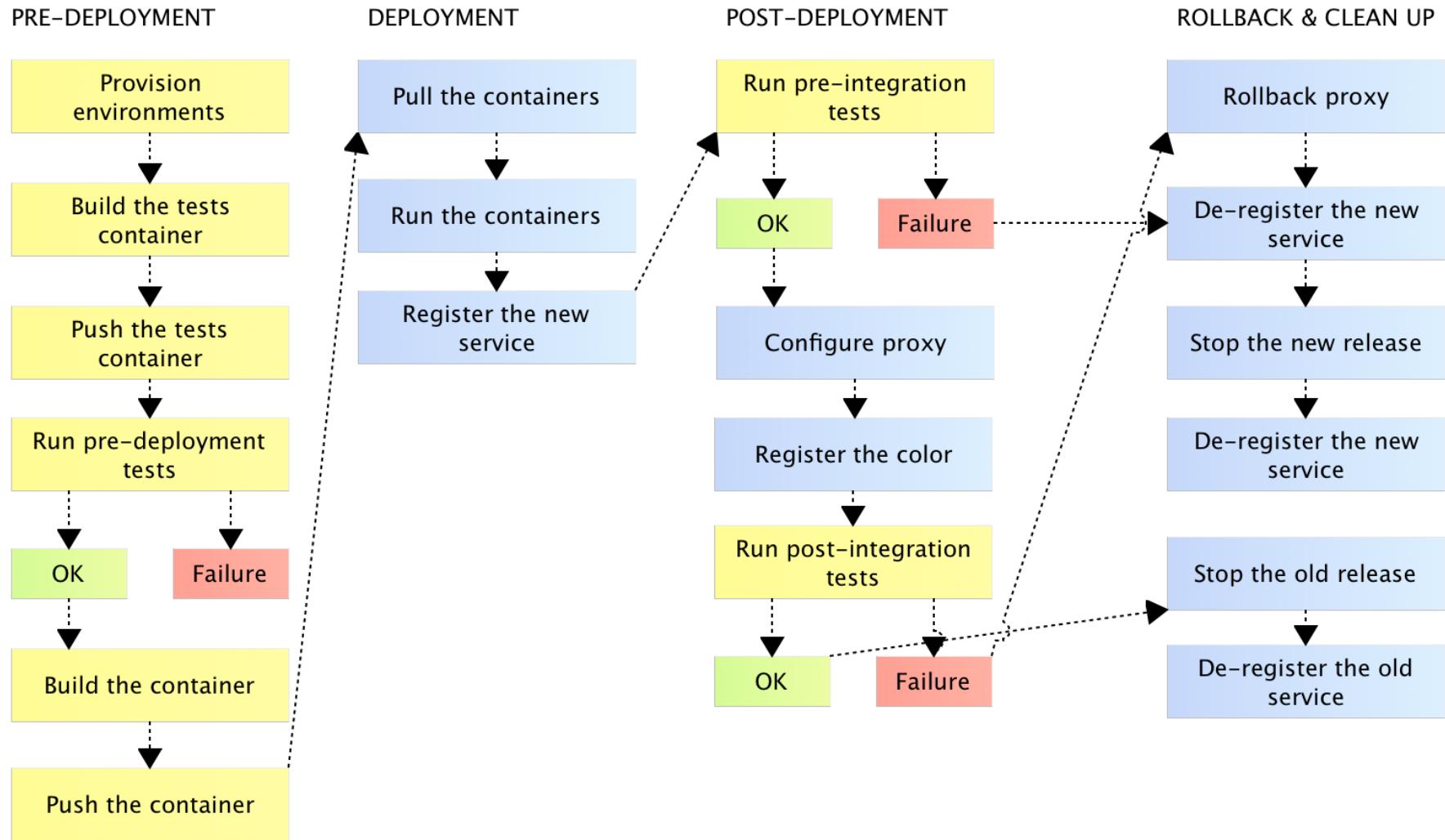
CICD



CD maturity



CI/CD and Containers

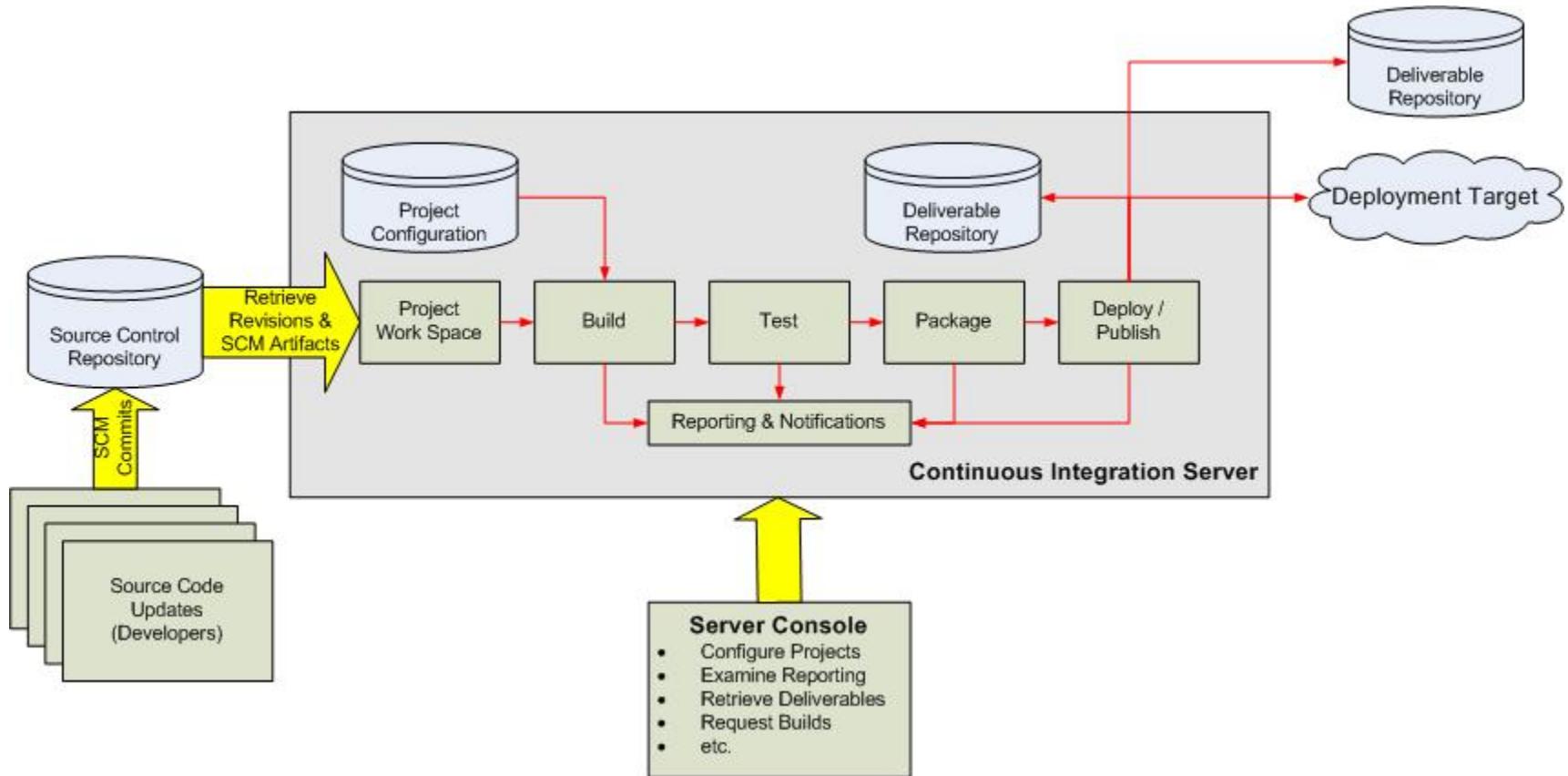


Jenkins

What's Jenkins/Hudson

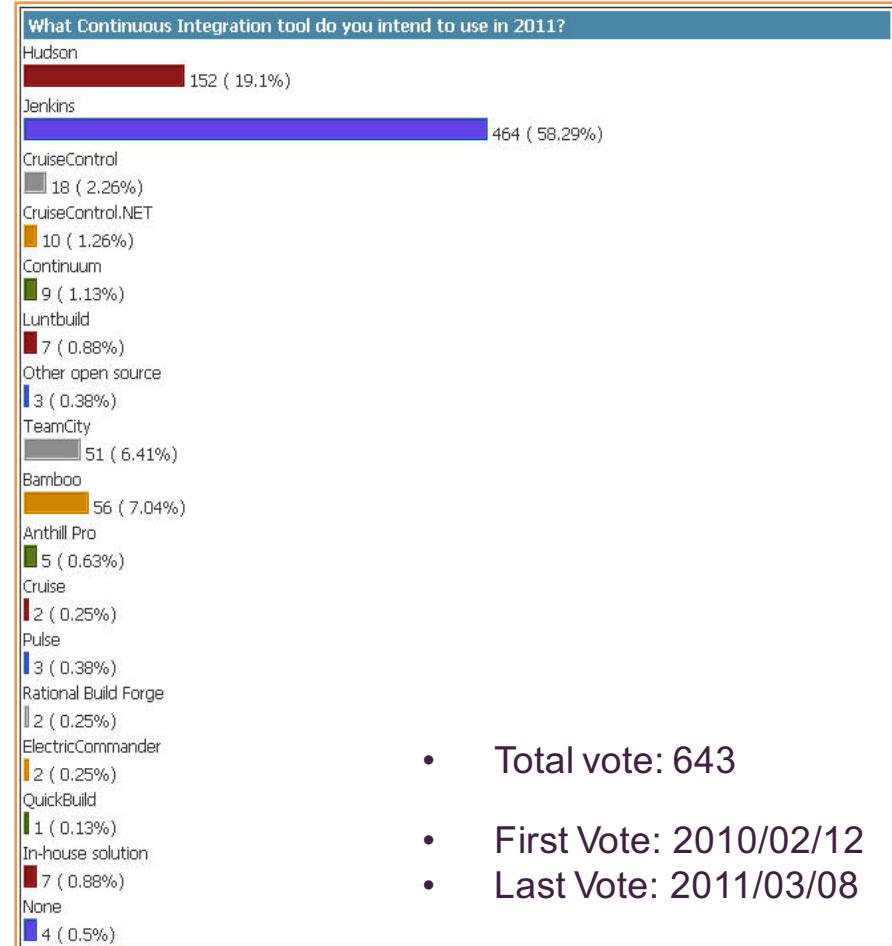
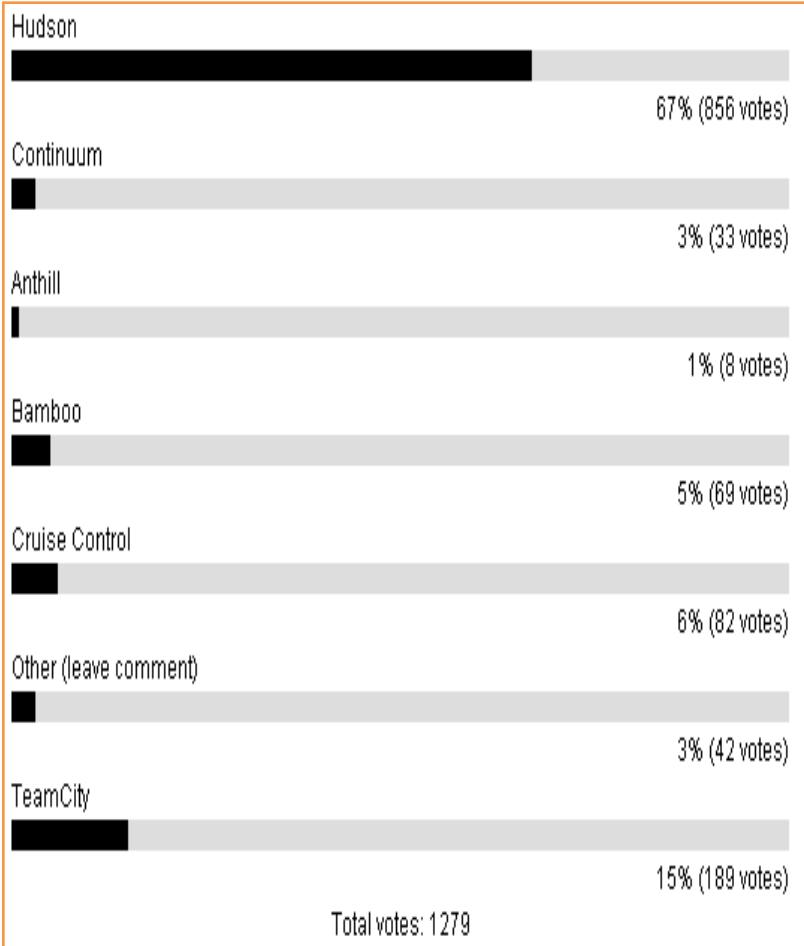
1. An open source CI server
2. Plug-in extensibility
3. MIT license

Continuous Integration Overview



Ref: <http://www.javaworld.com/javaworld/jw-12-2008/images/CIOverview.jpg>

CI Tool Usage



Ref: <http://java.dzone.com/node/28241/results>

Ref: <http://www.wakaleo.com/resources/polls>

Jenkins v.s.Hudson

- Jenkins: Original Hudson team
 - Hudson: Oracle and Sonatype
-
- Ref: <http://jenkins-ci.org/content/hudsons-future>

Jenkins Features

- Trigger a build
- Get source code from repository
- Automatically build and test
- Generate report & notify
- Deploy
- Distributed build

Jenkins Requirement

- Web Server (Tomcat, WebLogic, ...)
- Build tool (Maven, Ant)
- SCM (Git, Svn, Cvs, ...)

Jenkins Plugins

- Build triggers
- Source code management
- Build tools
- Build wrappers
- Build notifiers
- Build reports
- Artifact uploaders
- UI plugins
- Authentication and user management

Build Trigger

- Manually click build button
- Build periodically
- Build whenever a SNAPSHOT dependency is built
- Build after other projects are built
- Poll SCM
- IRC, Jabber, ...

Get Source Code (1/2)

- CVS (build-in)
- SVN (build-in)
- GIT (requires Git)
- ClearCase (requires ClearCase)
- Mercurial, PVCS, VSS, ...

Get Source Code (2/2)

- Get current snapshot
- Get baseline (tag)

Hudson > A

[Back to Dashboard](#)

[Status](#)

[Changes](#)

[Workspace](#)

[Build Now](#)

[Delete Project](#)

[Configure](#)

Project A

This build requires parameters:

ClearCase UCM baseline

fa4_web_INITIAL
fa4_web_INITIAL
fa4_web_INITIAL_4P
hudson-LEM-03-UploadArtifactsIntoClearCase-37.2707
hudson-LEM-03-UploadArtifactsIntoClearCase-38.2501
hudson-LEM-03-UploadArtifactsIntoClearCase-39.8732
fa4_initial_07_01_2010.6927

Build History (trend)

[for all](#) [for failures](#)

Code Change History

Jenkins

Jenkins » Jakarta HTTP Client » #4 允許自動更新頁面

回到專案 狀態 變更 畫面輸出 Configure Tag this build Monitor Maven Process 上次建構

Build #4 (2011/3/11 下午 03:24:05) Progress: Started 4 min 1 sec ago

Revision: 1080422 Changes

1. Minor enhancement to stale-while-revalidate (RFC5861) handling; now we can also serve stale 304s (Not Modified) while asynchronously revalidating. Prior to this, we were always returning a stale 200 response regardless of whether the incoming request was conditional or not. The old behavior was not incorrect, but this is better. ([detail](#))
2. + Fix minor typos in private function names. ([detail](#))
3. Upgraded Jetty; minor benchmark improvements ([detail](#))
4. HttpClient benchmark improvements ([detail](#))
5. Merged fixes from 4.1.x branches ([detail](#))
6. HTTPCLIENT-1051: eliminated reverse DNS lookup when performing hostname verification for secure connections ([detail](#))
7. Changed project version to 4.2-alpha1-SNAPSHOT ([detail](#))
8. Reverted HTTPCLIENT-1051 ([detail](#))
9. HTTPCLIENT-1066: Handling of multiple consecutive slashes in the URI path component ([detail](#))
10. HTTPCLIENT-1051: Default X509 hostname verifier rejects certificates with an IP address as CN ([detail](#))
11. More test cases for cookie protocol interceptors ([detail](#))

[add description](#)

Build Tools

- Java
 - Maven (build-in), Ant, Gradle
- .Net
 - MSBuild, PowerShell
- Shell script
 - Python, Ruby, Groovy

Build Wrapper

- Build name (version no) setter
- Virtual machine (VMWare, Virtual Box)
- Set environment variable
- ClearCase release plugin
- ...

Build Notifier

- E-mail
- Twitter
- Jabber
- IRC
- RSS
- Google calendar
- ...

Build Report

- Static Code Analysis
 - Checkstyle, PMD, Findbugs, Compiler Warning
- Test Report & Code Coverage
 - JUnit, TestNG, Cobertura, Clover
- Open Tasks

Static Code Analysis

All **Dashboard** +

Warnings per Job

Job ↓	Checkstyle	Duplicate Code	FindBugs	PMD	Open Tasks	Compiler Warnings	Total
M-Multi-Freestyle	14	-	0	86	24	0	124
M-Multi-Maven	14	11	12	80	0	4	121
M-Single-Freestyle	0	0	15	18	99	3	135
M-Single-Freestyle-Site	5967	2	-	18	5	0	5992
M-Single-Maven	-	-	-	-	121	0	121
M-Single-Maven-2	-	4	16	243	4	0	267
M-Single-Maven-Site	1784	2	72	2	0	-	1860

The chart displays the count of different warning types across ten builds. The Y-axis represents the count, ranging from 0 to 100. The X-axis represents the build numbers from #91 to #100. The legend identifies five warning types: Checkstyle (red), Compiler (blue), Duplicate Code (green), FindBugs (orange), and PMD (purple). The PMD count shows significant fluctuations, peaking at 100 in builds #92 and #94, and dropping to near zero in build #95. The FindBugs count is relatively stable around 20. The other three warning types (Checkstyle, Compiler, and Duplicate Code) have much lower counts, generally below 20, with some minor fluctuations.

Build	Checkstyle	Compiler	Duplicate Code	FindBugs	Open Task	PMD
#91	0	0	0	0	0	0
#92	0	22	12	20	0	100
#93	0	22	12	20	0	100
#94	0	22	12	20	0	100
#95	0	22	12	20	0	0
#96	0	22	12	20	0	95
#97	0	22	12	20	0	95
#98	0	22	12	20	0	95
#99	0	22	12	20	0	95
#100	0	22	12	20	0	95

Legend: Checkstyle, Compiler, Duplicate Code, FindBugs, Open Task, PMD

CheckStyle

CheckStyle Result

Warnings Trend

All Warnings	New Warnings	Fixed Warnings
6766	2	0

Summary

Total	High Priority	Normal Priority	Low Priority
6766	6766	0	0

Details

Modules	Package	Files	Categories	Types	Warnings	Details	New
---------	---------	-------	------------	-----------------------	----------	---------	-----

Type	Total	Distribution
AvoidInlineConditionalsCheck	47	
AvoidNestedBlocksCheck	2	
AvoidStarImportCheck	1	
ConstantNameCheck	11	
DesignForExtensionCheck	789	
EmptyBlockCheck	29	
FinalClassCheck	5	
FinalParametersCheck	980	
HiddenFieldCheck	288	
HideUtilityClassConstructorCheck	9	
InnerAssignmentCheck	1	
InterfaceIsTypeCheck	8	
JavadocMethodCheck	1472	
JavadocDocBlockCheck	1	

FindBugs

FindBugs Result

Warnings Trend

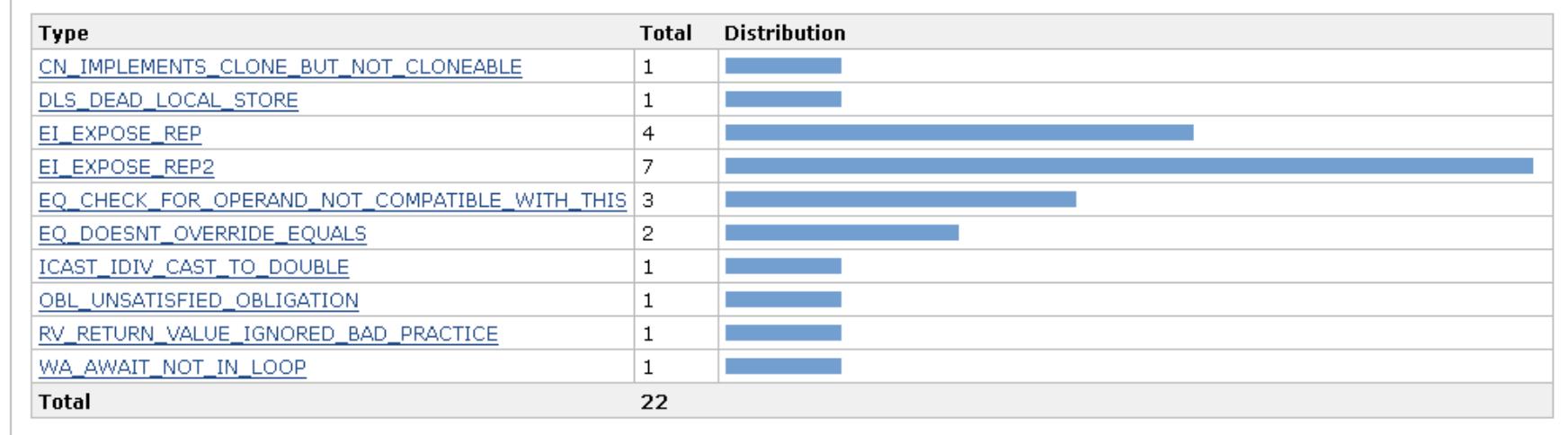
All Warnings	New this build	Fixed Warnings
22	0	0

Summary

Total	High Priority	Normal Priority	Low Priority
22	0	0	<u>22</u>

Details

Modules	Package	Files	Categories	Types	Warnings	Details
---------	---------	-------	------------	-------	----------	---------



Open Tag

Open Tasks

Open Tasks Trend

All Open Tasks	New Tasks	Fixed Tasks
19	19	0

Summary

Total	High Priority	Normal Priority
19	0	19

Details

Package	Files	Warnings		New		
File	Package		Line	Priority	Type	Category
AbstractClearCaseScm.java	hudson.plugins.clearcase		255	Normal	TODO	
AbstractClearCaseScm.java	hudson.plugins.clearcase		256	Normal	TODO	
AbstractClearCaseScm.java	hudson.plugins.clearcase		257	Normal	TODO	
ClearTool.java	hudson.plugins.clearcase		196	Normal	TODO	
ClearTool.java	hudson.plugins.clearcase		237	Normal	TODO	
ClearTool.java	hudson.plugins.clearcase		238	Normal	TODO	

Duplicate Code

Duplicate Code Result

Warnings Trend

All Warnings	New Warnings	Fixed Warnings
16	0	0

Summary

Total	High Priority	Normal Priority	Low Priority
16	0	10	6

Details

Package	Files	Warnings	Details	Normal	Low

File	Number of lines	Duplicated in
NetscapeDraftSpec.java:120	17	BestMatchSpec.java:117
BestMatchSpec.java:117	17	NetscapeDraftSpec.java:120
SSLocketFactory.java:462	28	PlainSocketFactory.java:154
AbstractAuthenticationHandler.java:82	23	AuthSchemeBase.java:88
RequestProxyAuthentication.java:95	28	RequestTargetAuthentication.java:86
RFC2109DomainHandler.java:47	32	BasicDomainHandler.java:45
AuthSchemeBase.java:88	23	AbstractAuthenticationHandler.java:82
NetscapeDraftSpec.java:137	19	BrowserCompatSpec.java:163
PlainSocketFactory.java:154	28	SSLocketFactory.java:462
RequestTargetAuthentication.java:86	28	RequestProxyAuthentication.java:95
BrowserCompatSpec.java:163	19	NetscapeDraftSpec.java:137
DefaultRedirectHandler.java:141	44	DefaultRedirectStrategy.java:133
BrowserCompatSpec.java:128	34	BestMatchSpec.java:101
BasicDomainHandler.java:45	32	RFC2109DomainHandler.java:47
DefaultRedirectStrategy.java:133	44	DefaultRedirectHandler.java:141
BestMatchSpec.java:101	34	BrowserCompatSpec.java:128

Test Report

Tests							
Job	Success		Failed		Skipped		Total
	#	%	#	%	#	%	#
Eden-ActiveWorlds	127	100%	0	0%	0	0%	127
Eden-Eden	266	100%	0	0%	0	0%	266
Eden-MondiDinamiciWebservice	155	100%	0	0%	0	0%	155
Eden-Palm	20	100%	0	0%	0	0%	20
Total	568	100%	0	0%	0	0%	568

Test Result

0 failures (± 0) , 2 skipped (± 0)

1,513 tests (± 0)

Module	Fail	(diff)	Total	(diff)
org.apache.httpcomponents:httpclient	0		582	
org.apache.httpcomponents:httpclient-cache	0		920	
org.apache.httpcomponents:httppmime	0		11	

Test Code Coverage

Packages

All

[net.sourceforge.cobertura.ant](#)

[net.sourceforge.cobertura.check](#)

[net.sourceforge.cobertura.cover](#)

[net.sourceforge.cobertura.instru](#)

[net.sourceforge.cobertura.merg](#)

[net.sourceforge.cobertura.repor](#)

[net.sourceforge.cobertura.repor](#)

[net.sourceforge.cobertura.repor](#)

All Packages

Classes

[AntUtil \(88%\)](#)

[Archive \(100%\)](#)

[ArchiveUtil \(80%\)](#)

[BranchCoverageData \(N/A\)](#)

[CheckTask \(0%\)](#)

[ClassData \(N/A\)](#)

Coverage Report - All Packages

Package	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	55	75% 1625/2179	64% 472/738	2.319
net.sourceforge.cobertura.ant	11	52% 170/330	43% 40/94	1.848
net.sourceforge.cobertura.check	3	0% 0/150	0% 0/76	2.429
net.sourceforge.cobertura.coveragedata	13	N/A N/A	N/A N/A	2.277
net.sourceforge.cobertura.instrument	10	90% 460/510	75% 123/164	1.854
net.sourceforge.cobertura.merge	1	86% 30/35	88% 14/16	5.5
net.sourceforge.cobertura.reporting	3	87% 116/134	80% 43/54	2.882
net.sourceforge.cobertura.reporting.html	4	91% 475/523	77% 156/202	4.444
net.sourceforge.cobertura.reporting.html.files	1	87% 39/45	62% 5/8	4.5
net.sourceforge.cobertura.reporting.xml	1	100% 155/155	95% 21/22	1.524
net.sourceforge.cobertura.util	9	60% 175/291	69% 70/102	2.892
someotherpackage	1	83% 5/6	N/A N/A	1.2

Report generated by Cobertura 1.9 on 6/9/07 12:37 AM.

Ref: <http://cobertura.sourceforge.net/sample/>

Artifact uploaders

- Tomcat
- JBoss
- Glassfish
- WebSphere
- FTP
- SSH

UI Enhancement

- Dashboard
- Sectioned view
- iPhone/Android

Security Management

- Security Realm
 - LDAP
 - Jenkins's own user database
 - Delegate to servlet container
- Authorization
 - Anyone can do anything
 - Logged-in users can do anything
 - Matrix-based security
 - Project-based Matrix Authorization Strategy
 - Legacy mode

Security Management

- Matrix-based security

• Matrix-based security

User/group to add: 新增

- Project-based Matrix Authorization

Enable project-based security

User/group to add: 新增

Security Management Plugins

- Active directory, OpenID, MySQL, ...
 - Role based privilege control

Role to add

Add

Project roles		Job					Run		
Role	Pattern	Delete	Configure	Read	Build	Workspace	Delete	Update	Artifacts
[Red]	[Red]	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
[Red]	[Red]	<input checked="" type="checkbox"/>							
[Red]	[Red]	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Sample	Sample.*	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
[Red]	[Red]	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

IT'S PRACTICAL TIME FOR JENKINS

DevOps

The Problem In A Nutshell

- Everything needs software.
- Software runs on a server to become a service.
- Delivering a service from inception to its users is too slow and error-prone.
- There are internal friction points that make this the case.
- This loses you money. (Delay = loss)
- Therefore IT is frequently the bottleneck in the transition of “concept to cash.”

Symptoms

- Defects released into production, causing outage
- Inability to diagnose production issues quickly
- Problems appear in some environments only
- Blame shifting/finger pointing
- Long delays while dev, QA, or another team waits on resource or response from other teams
- “Manual error” is a commonly cited root cause
- Releases slip/fail
- Quality of life issues in IT

Why Does This Problem Exist?

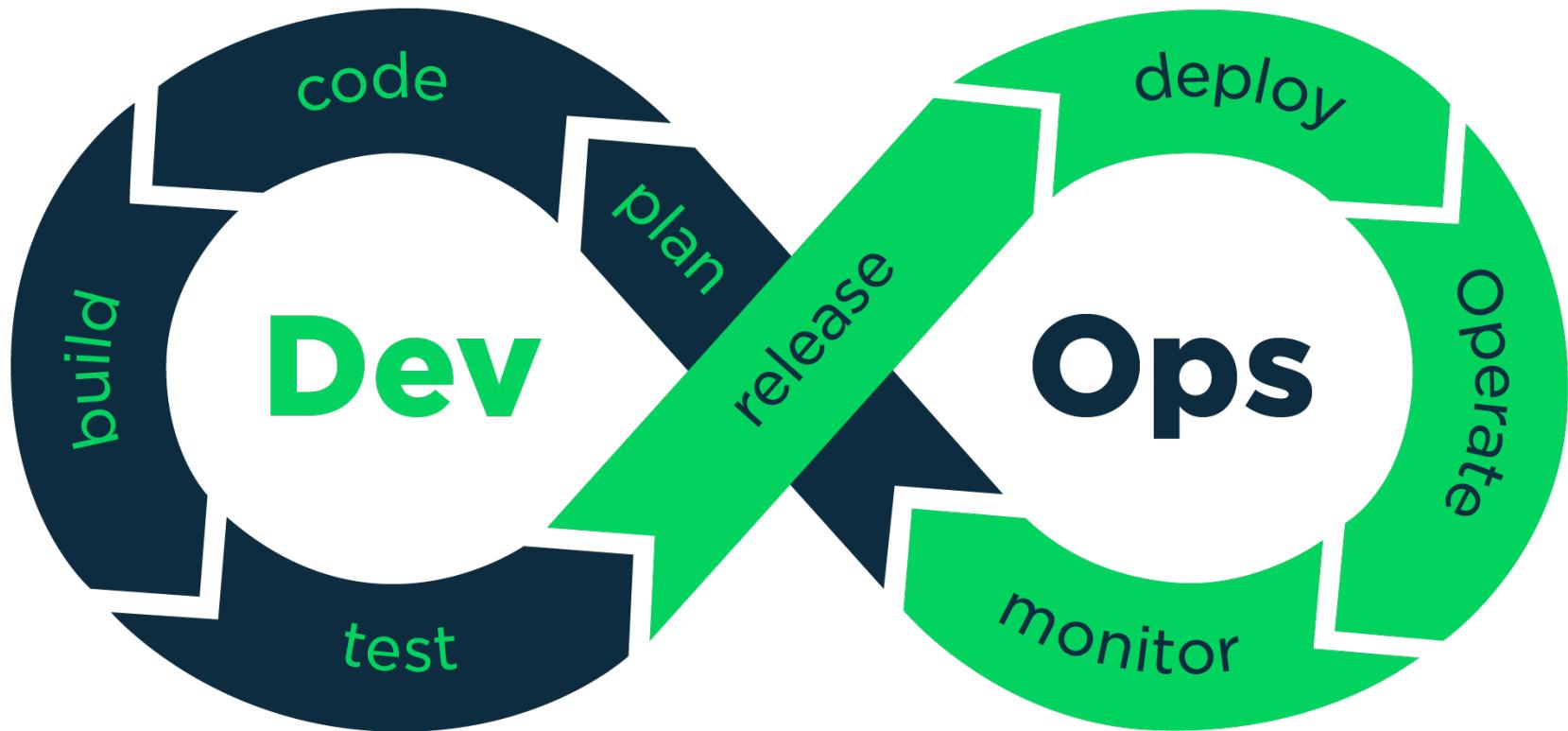
- “Business-IT Alignment?”
- The business has demanded the wrong things out of IT
 - Cost sensitive
 - Risk averse
- IT has metastasized over time into a form to give the business what it’s said it wants
 - Centralized and monolithic
 - Slow and penny wise, pound foolish

DevOps

A culture and mindset for collaborating between
developers and operations

DevOps Defined

- DevOps is the practice of operations and development engineers participating together in the entire service lifecycle, from design through the development process to production support.
- DevOps is also characterized by operations staff making use many of the same techniques as developers for their systems work.



DevOps

- ✖ Tools ✓ Communication
- ✖ Automation ✓ Understanding
- ✖ Access rights ✓ Integration
- ✖ Teams ✓ Relationships

DevOps Concepts



- DevOps Principles

- DevOps Practices

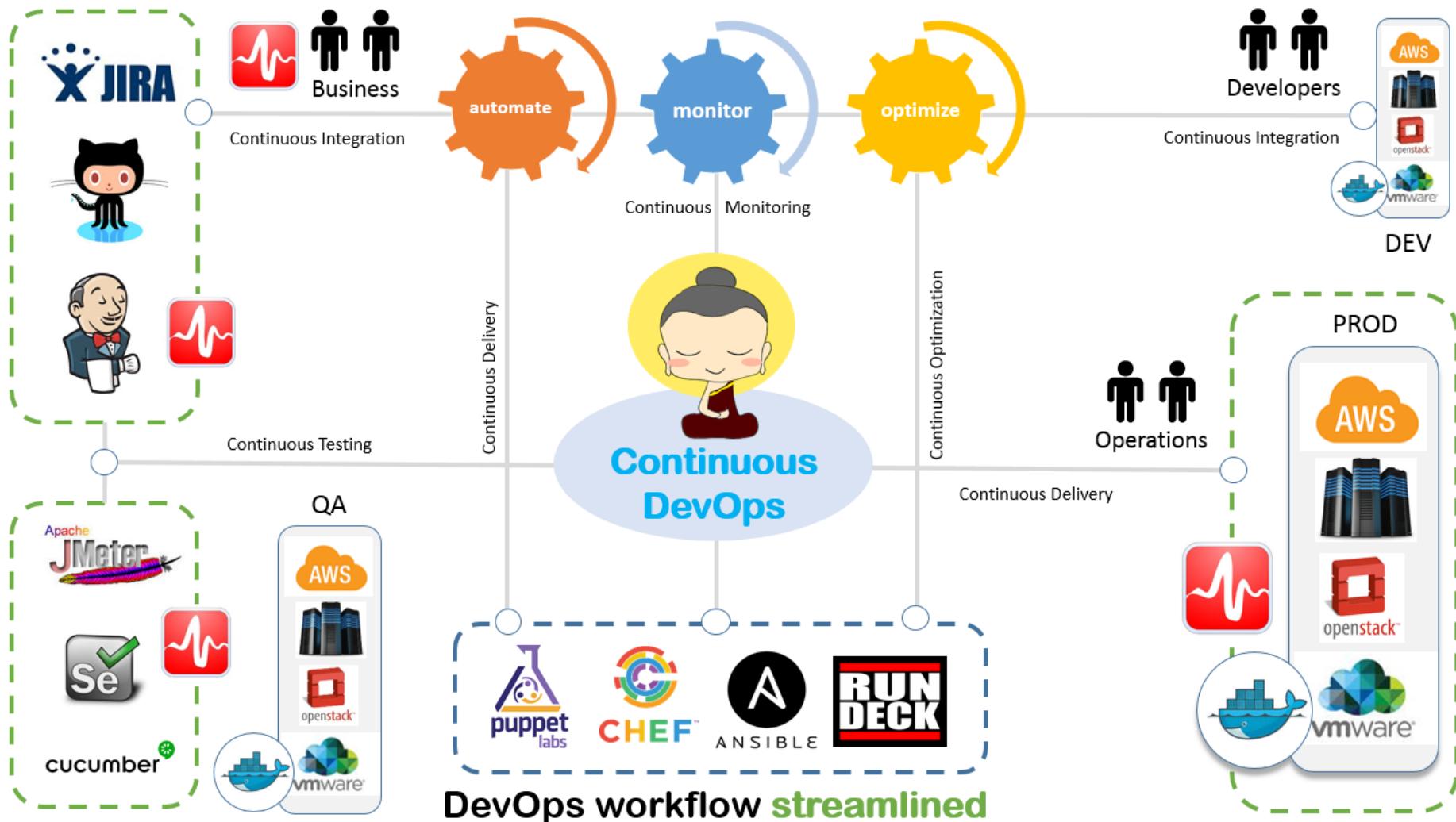
- DevOps Tools

DevOps Principles

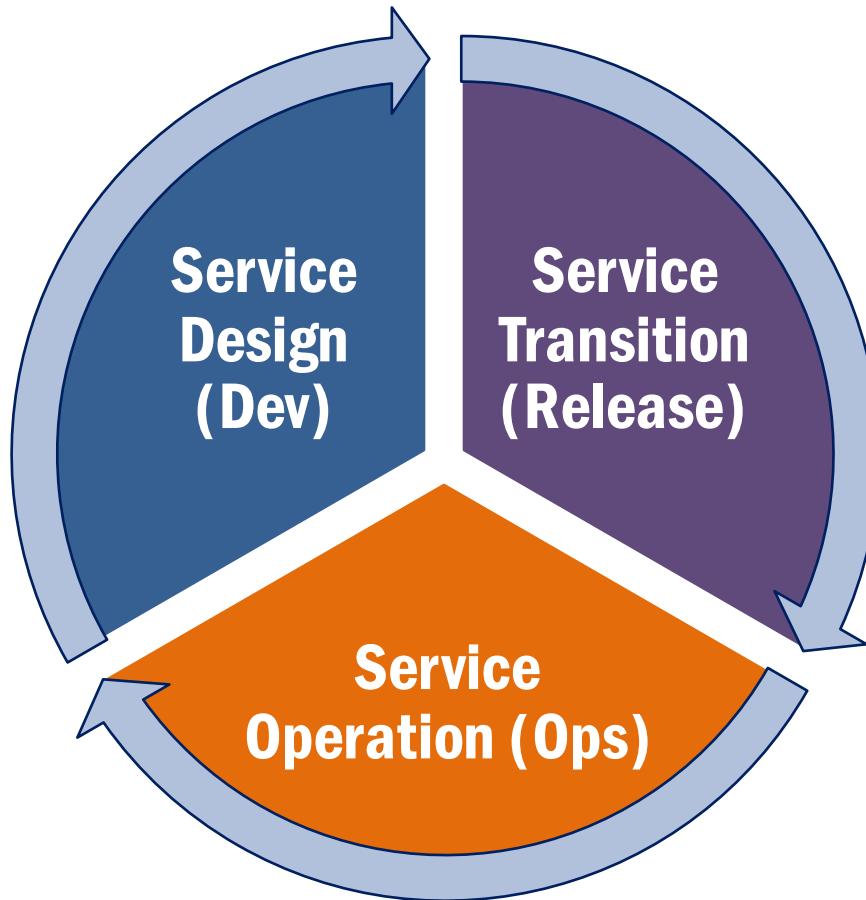
- **The Three Ways**
 - Systems Thinking
 - Amplify Feedback Loops
 - Culture of Continual Experimentation
- **CAMS**
 - Culture – People > Process > Tools
 - Automation – Infrastructure as Code
 - Measurement – Measure Everything
 - Sharing – Collaboration/Feedback
- Informed by the values in the Agile Manifesto and Lean Theory of Constraints

DevOps Practices

- Version Control For All
- Automated Testing
- Proactive Monitoring and Metrics
- Kanban/Scrum
- Visible Ops/Change Management
- Configuration Management
- Incident Command System
- Continuous Integration/Deployment/Delivery
- “Put Developers On Call”
- Virtualization/Cloud/Containers
- Toolchain Approach
- Transparent Uptime/Incident Retrospectives



An Implementation Model



Add Ops Into Dev

- Enhance Service Design With Operational Knowledge
 - Reliability
 - Performance
 - Security
 - Test Them
- Build Feedback Paths Back from Production
 - Monitoring and metrics
 - Postmortems
- Foster a Culture of Responsibility
 - Whether your code passes test, gets deployed, and stays up for users is your responsibility – not someone else's
- Make Development Better With Ops
 - Production like environments
 - Power tooling

Accelerate Flow To Production

- Reduce batch size
- Automated environments mean identical dev/test/prod environments
- Create safety through automation
 - Continuous Integration/Testing
 - Automated Regression Testing
 - Continuous Delivery
 - Continuous Deployment
 - Feature Flags (A/B testing)
 - Security Testing

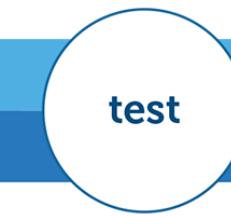
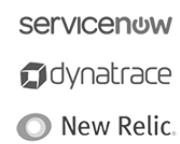
Add Dev Into Ops

- Don't do tasks for people. Build tools so they can do their own work.
- Monitoring/logging/metrics feeds back into dev (and the business)
- Blameless Incident Postmortems
- Developers Do Production Support/Empower Ops Acceptance

DEVOPS INTELLIGENCE

XL RELEASE | Release Orchestration

XL DEPLOY | Deployment Automation



dev » test » prod

PROVISION



Grass Roots Checklist

- Find ways to **collaborate** – involve others early
- Find ways to **automate** and make self-service
- Become **metrics** driven
- Learn new things, **continually improve**
- Understand the larger **business goals**, metrics, and priorities you support
- **Communicate**
- Work in parallel with **small batches**
- Allow **refactoring**
- **Prove the business value to management**

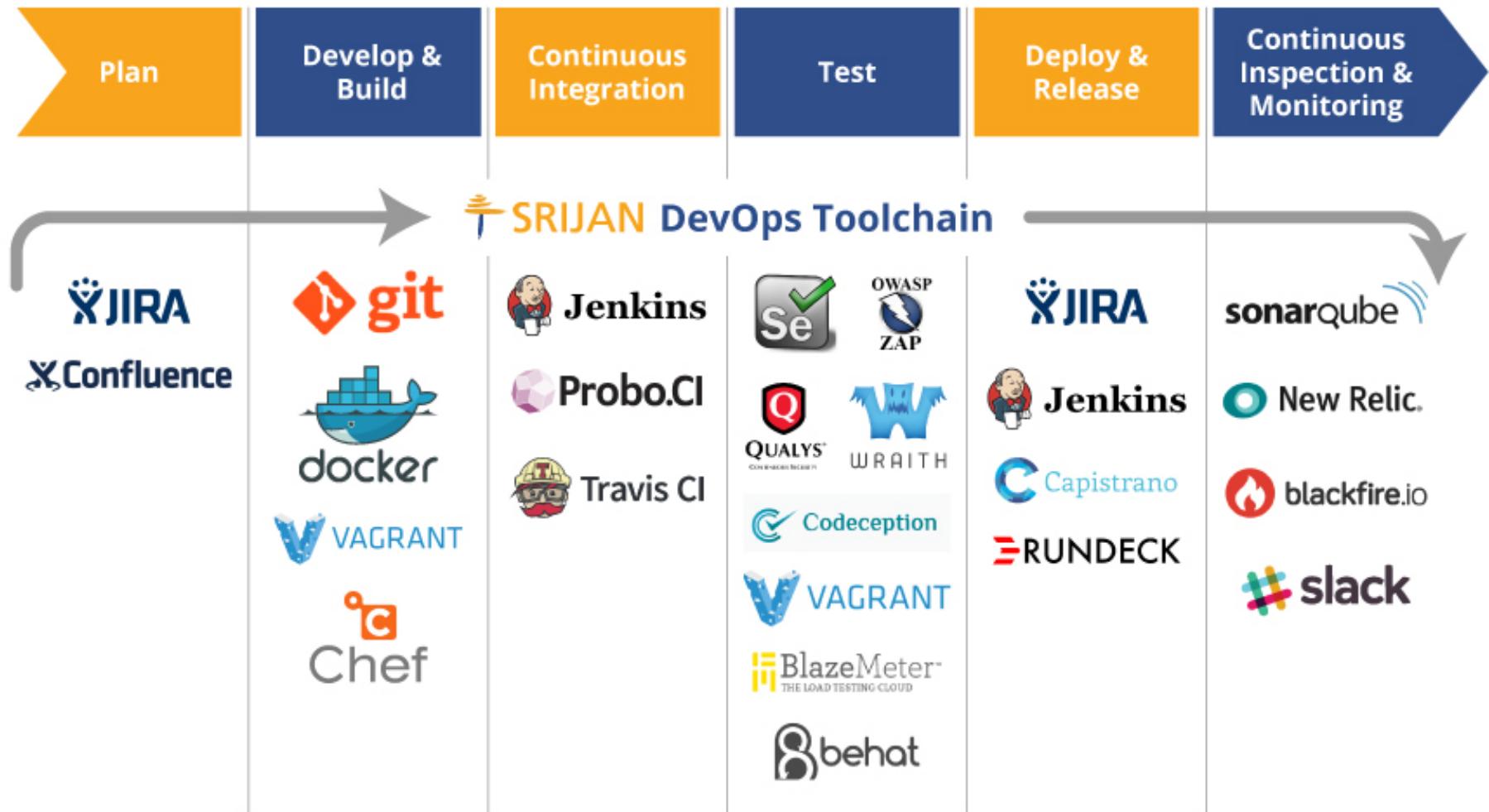
Management Checklist

- **Experiment** – choose a test case as a pilot
- Then document and spread **best practices**
- Empower your teams, but guide their **values**
- **Metrics** are your friend – demand measurable outcomes
- **Don't accept excuses** when the old baseline isn't good enough
- Fail fast, **continually improve**
- **Build** on small successes to gain broad support for more substantive change.
- **Align** roles and responsibilities across groups –enable collaboration even if it seems “inefficient”

Things Not To Do

- Only Token Gestures
 - “Ops team, change your name to DevOps team!”
 - “Put DevOps in those job titles!”
- Only Implement Tools
 - Changing tools without changing tactics leaves the battlefield strewn with bodies
- Create More Silos
- Devalue Operations Or Development Knowledge
- Anything You’re Not Measuring The Impact Of

DevOps Tools



SCRUM - OVERVIEW

SCRUM AT A GLANCE

The Agile: Scrum Framework at a glance

Inputs from Executives,
Team, Stakeholders,
Customers, Users



Product Owner



The Team



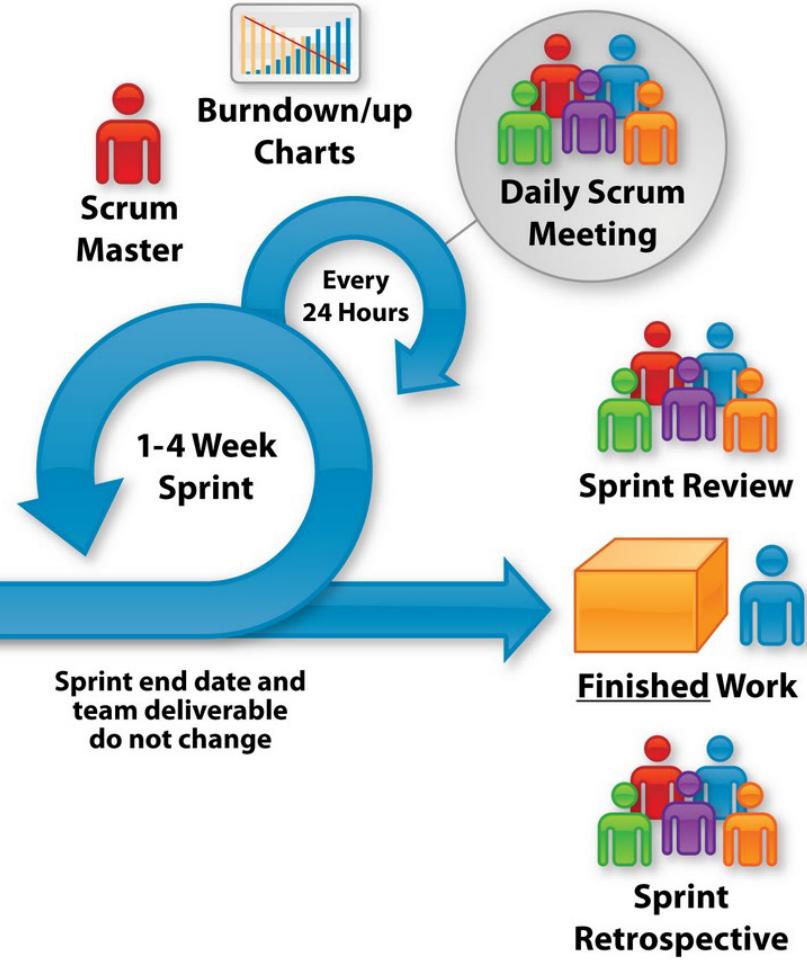
Product Backlog

Team selects starting at top as much as it can commit to deliver by end of Sprint

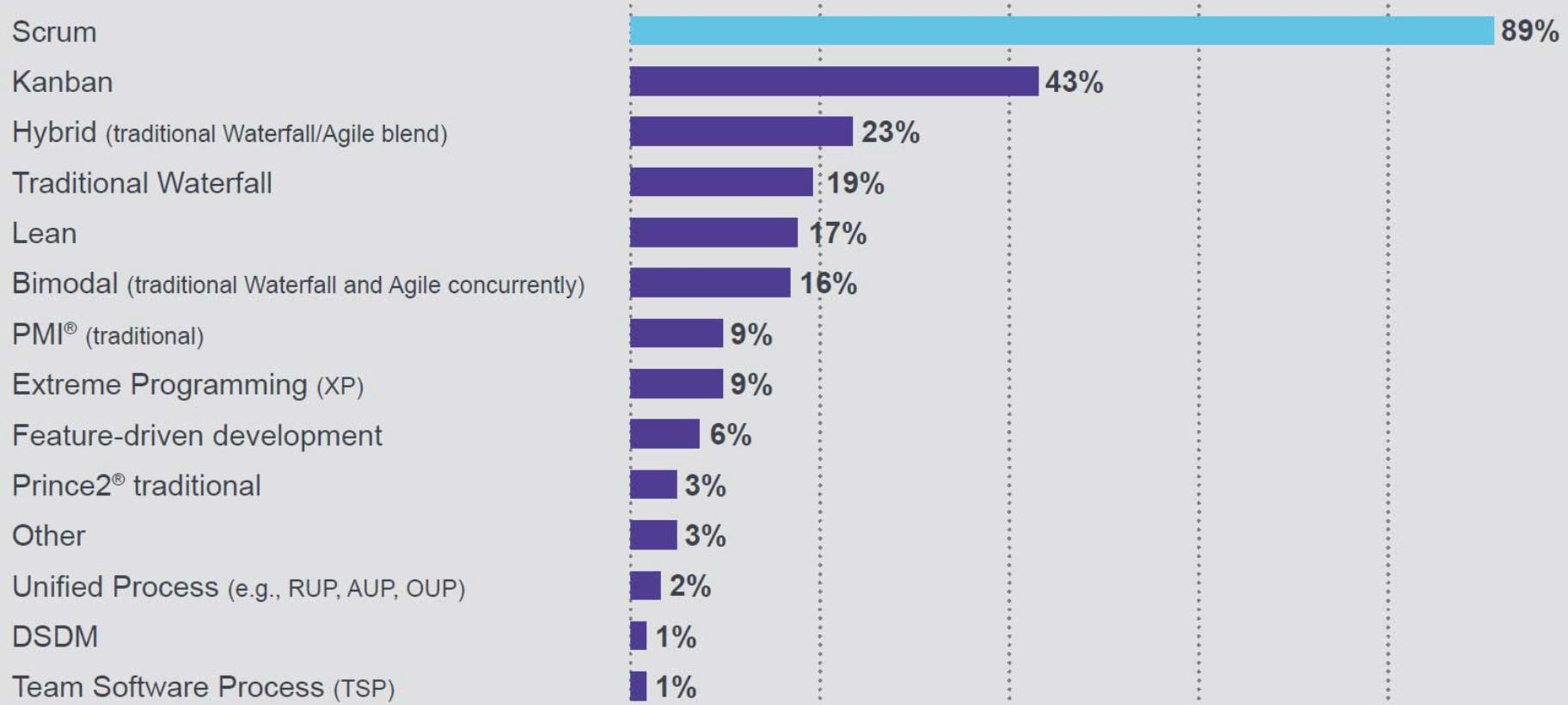
Sprint Planning Meeting



Sprint Backlog



SCRUM IN COMPANIES



The Foundations of Scrum

- There is a cross-functional Development Team – as a team, they have the skills needed to produce “done” product in a Sprint.
- The Development Team is self-organizing – the team organizes itself to get the work done.
- The Dev Team plans its Sprints one at a time, at the start of the Sprint.
- The Product Owner decides what needs to be produced.
- The Dev Team decides how much to target in each Sprint.
- The Dev Team’s goal is shared, clear, and does not change in Sprint.
- The Dev Team tries to hit its Sprint goal, but in some Sprints it will under-deliver, and in other Sprints it will over-deliver.
- Each Sprint is made up of timeboxes, which are not extended.
- The Dev Team aims for “done” each Sprint = tested and defect-free.
- There is a “Definition of Done,” that tells us what “done” means.
- At the end of the Sprint, we inspect and adapt product and process.

SCRUM ROLES

The Basics of Scrum

The Agile: Scrum Framework at a glance

Inputs from Executives,
Team, Stakeholders,
Customers, Users



Product Backlog



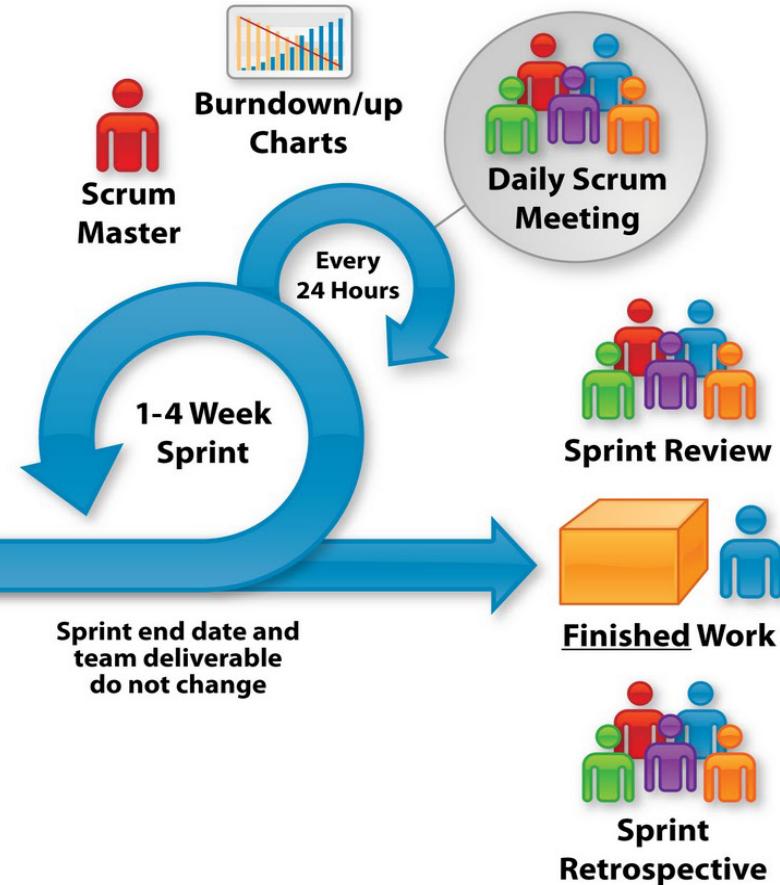
The Team

Team selects starting at top as much as it can commit to deliver by end of Sprint

Sprint Planning Meeting

Task Breakout

Sprint Backlog



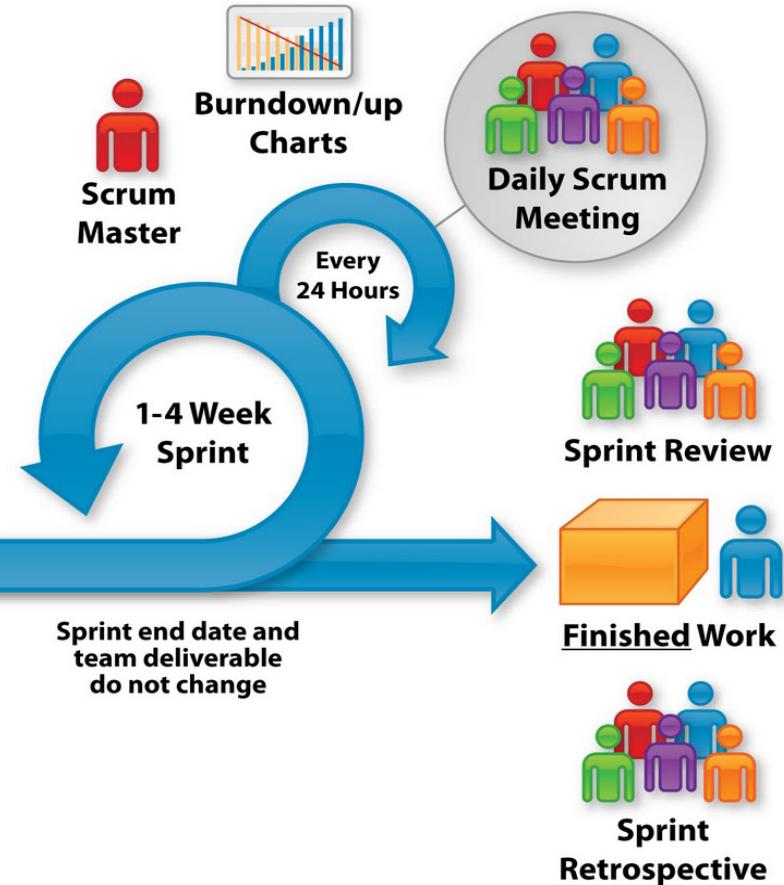
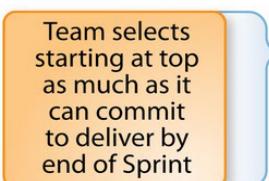
Product Owner

- Responsible for the overall project vision and goals
- Responsible for managing project ROI vs. risk
- Responsible for taking all inputs into what the team should produce, and turning it into a prioritized list (the Product Backlog)
- Participates actively in Sprint Planning and Sprint Review meetings, and is available to team throughout the Sprint
- Determines release plan and communicates it to upper management and the customer

The Basics of Scrum

The Agile: Scrum Framework at a glance

Inputs from Executives,
Team, Stakeholders,
Customers, Users



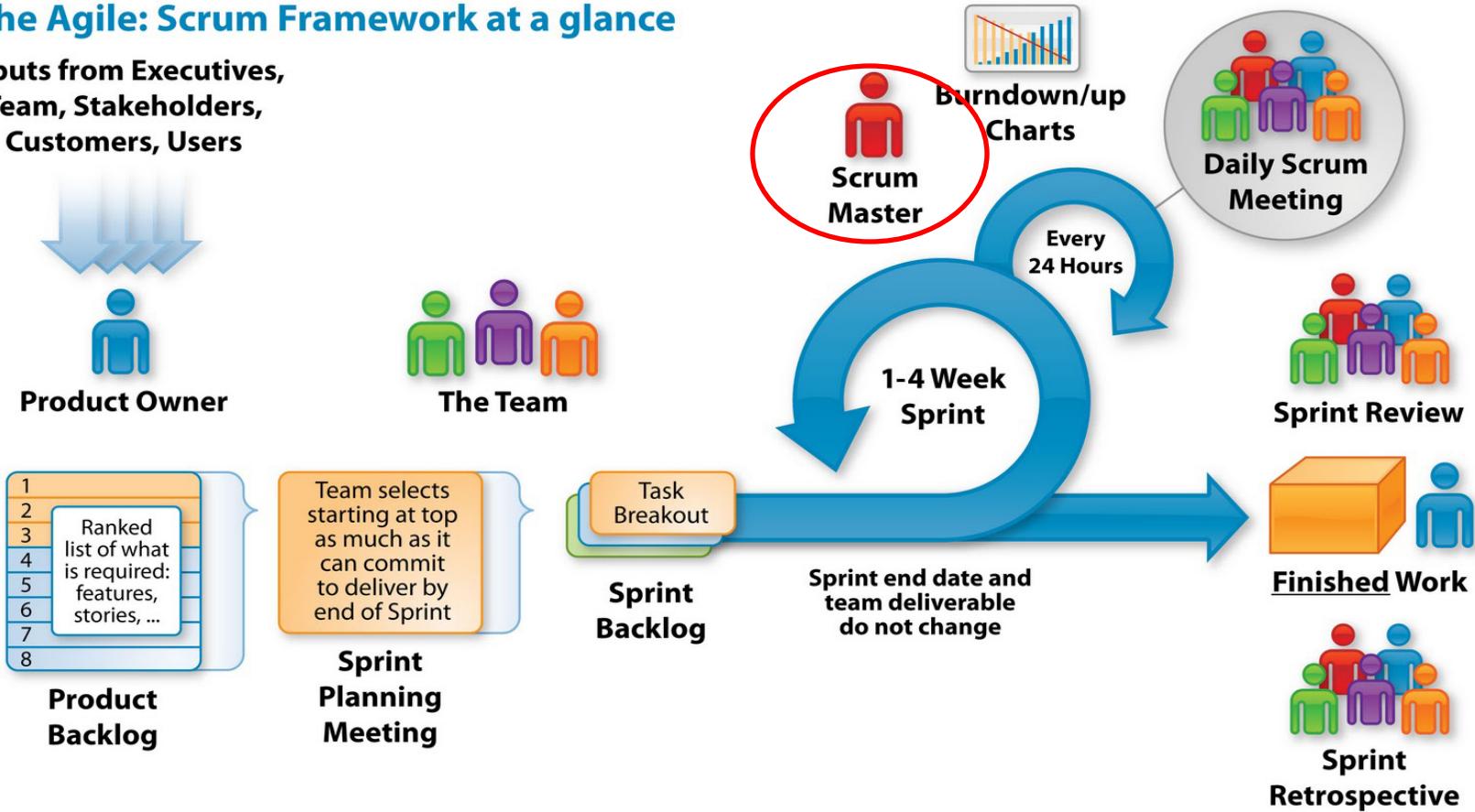
Team

- 6 people, + or – 3
 - Has worked with as high as 12, as few as 2
 - Can be shared with other teams (but better when not)
 - Can change between Sprints (but better when they don't)
 - Can be distributed (but better when co-located)
- Cross-functional
 - Possesses all the skills necessary to produce an increment of potentially shippable product
 - Team takes on tasks based on skills, not just official “role”
- Self-managing
 - Team manages itself to achieve the Sprint commitment

The Basics of Scrum

The Agile: Scrum Framework at a glance

Inputs from Executives,
Team, Stakeholders,
Customers, Users



The Role of the ScrumMaster

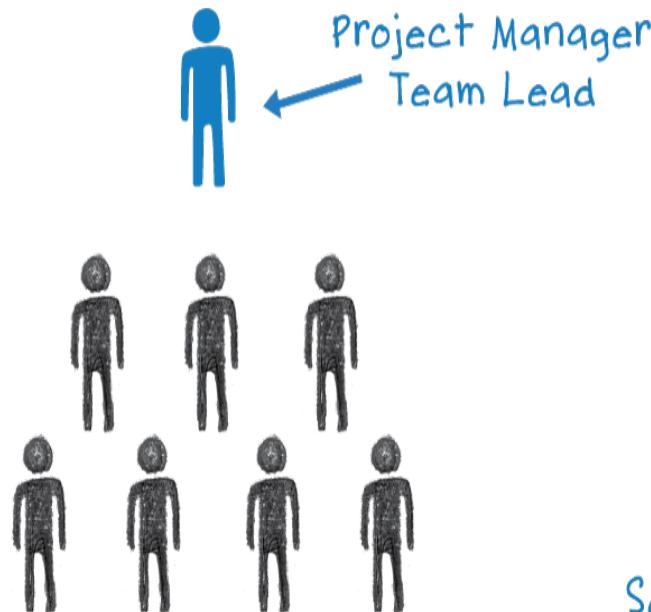
- The ScrumMaster does everything in their power to help the team achieve success
- This includes:
 - Serving the team
 - Protecting the team
 - Guiding the team's use of Scrum

What Does the ScrumMaster NOT Do?

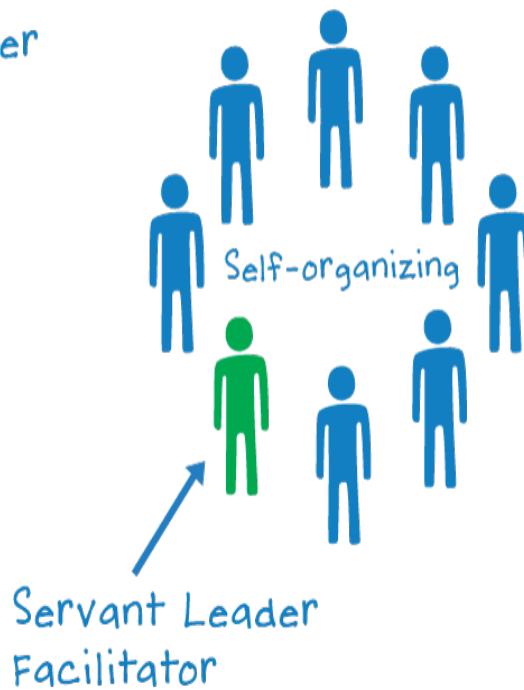
- The ScrumMaster does not manage the team
- The ScrumMaster does not direct team-members
- The ScrumMaster does not assign tasks
- The ScrumMaster does not “drive the team” to hit its goals
- The ScrumMaster does not make decisions for the team
- The ScrumMaster does not overrule team-members
- The ScrumMaster does not direct product strategy, decide technical issues, etc.

The Servant Leader

Traditional Teams



Agile Teams



The Basics of Scrum

The Agile: Scrum Framework at a glance

Inputs from Executives,
Team, Stakeholders,
Customers, Users



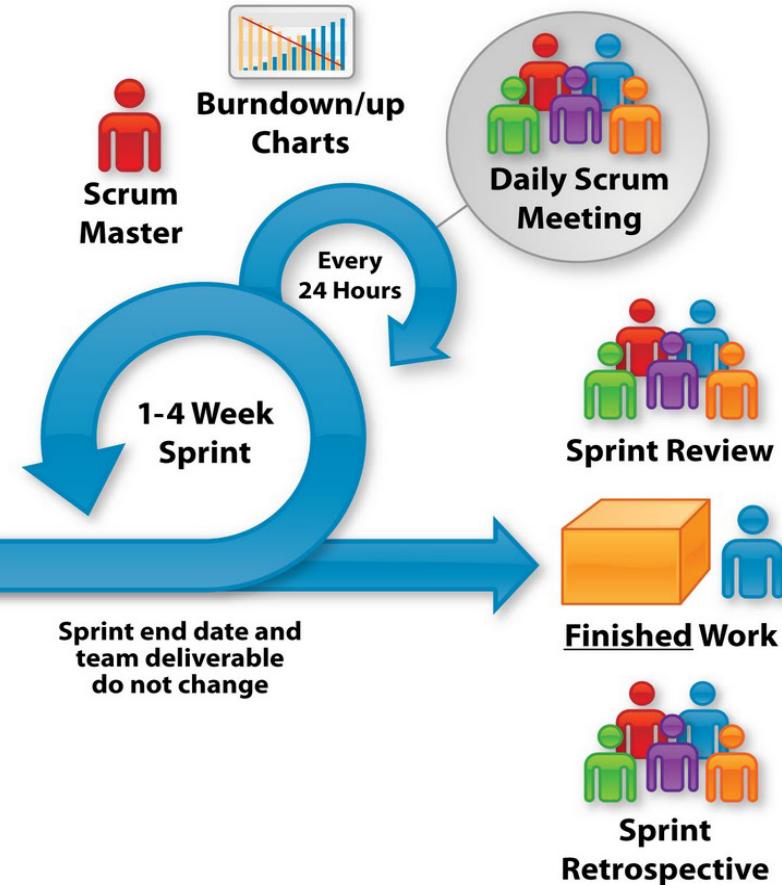
Product Backlog



Team selects starting at top as much as it can commit to deliver by end of Sprint

Sprint Planning Meeting

Task Breakout
Sprint Backlog



Sprint Planning Meeting

- Takes place before the start of every Sprint
- Team decides how much Product Backlog it will commit to complete by the end of the Sprint, and comes up with a plan and list of tasks for how to achieve it
- What's a good commitment?
 - Clearly understood by all
 - Shared among the team
 - Achievable without sacrificing quality
 - Achievable without sacrificing sustainable pace
- Attended by Team, Product Owner, ScrumMaster, Stakeholders
- May require 1-2 hours for each week of Sprint duration
 - 2 week Sprint = 2-4 hours, 4 week Sprint = 4-8 hours

The Basics of Scrum

The Agile: Scrum Framework at a glance

Inputs from Executives,
Team, Stakeholders,
Customers, Users



Product Owner



The Team



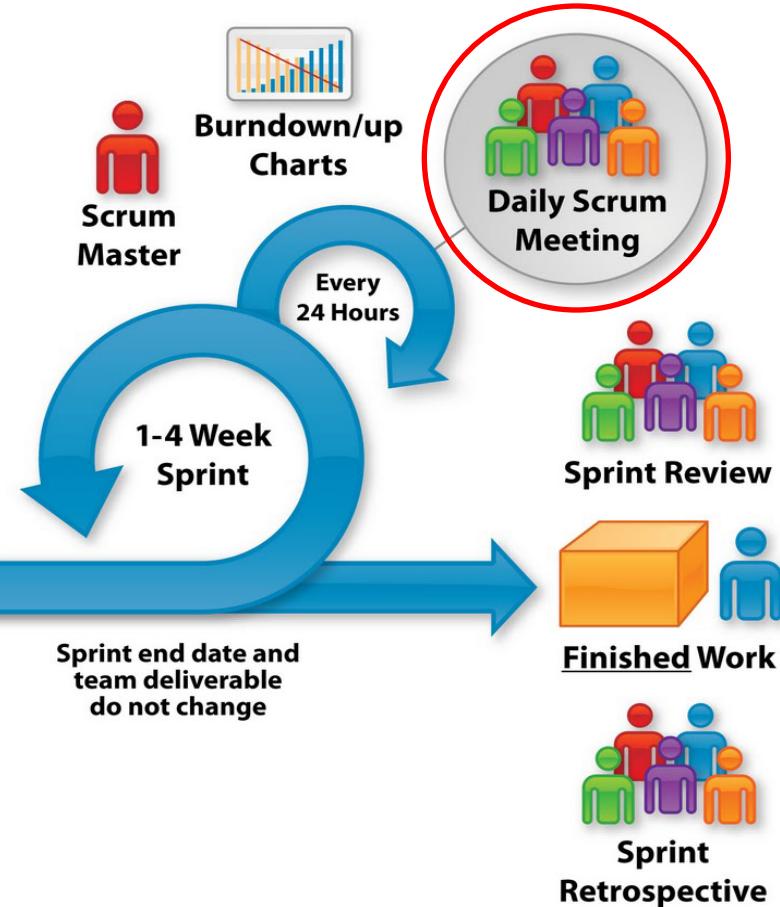
Product Backlog

Team selects starting at top as much as it can commit to deliver by end of Sprint

Sprint Planning Meeting

Task Breakout

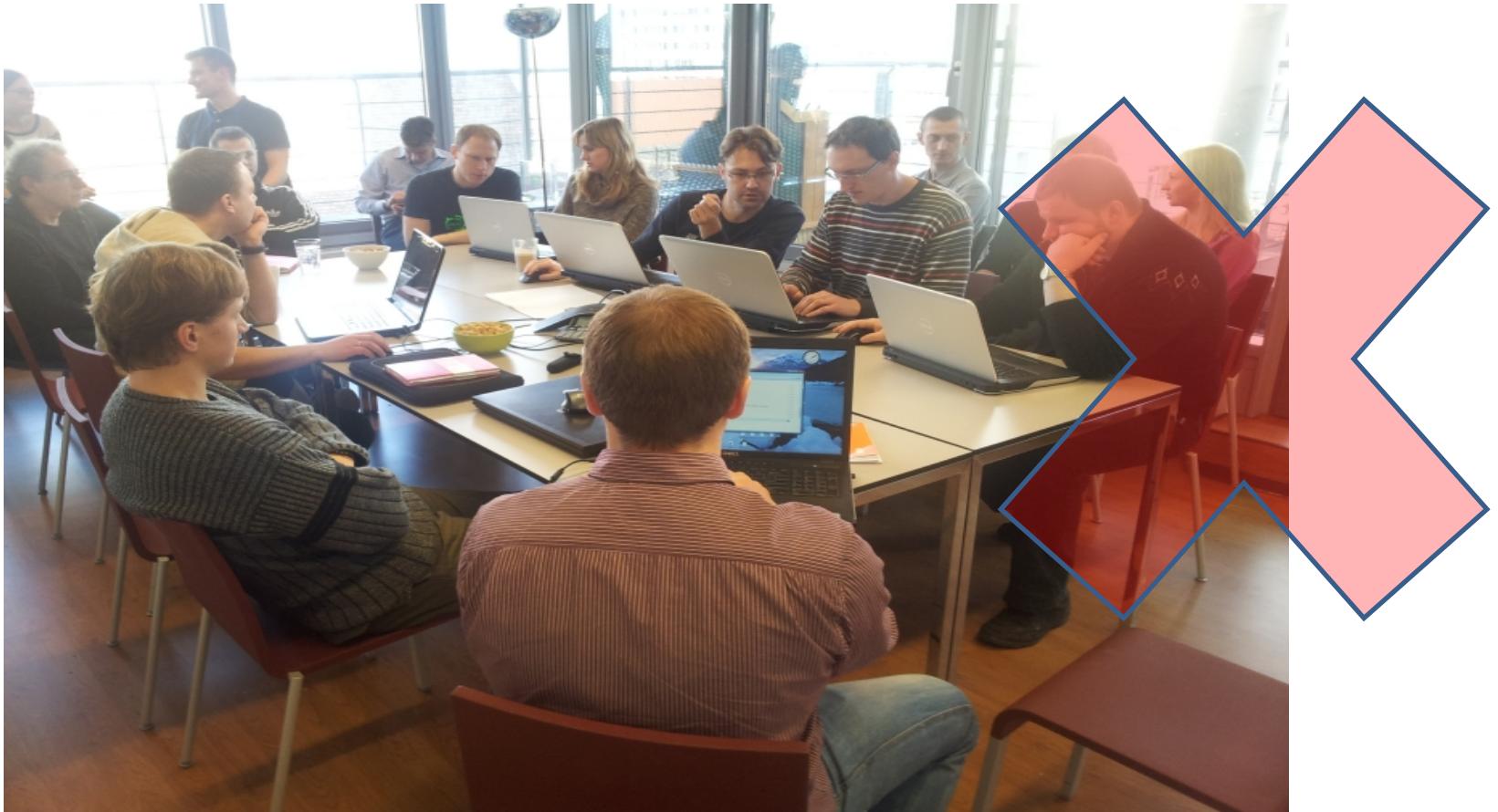
Sprint Backlog



Daily Scrum Meeting

- Purpose of Daily Scrum Meeting
 - Keep team coordinated and up-to-date with each other
 - Surface impediments daily
- How it works
 - Every weekday
 - Whole team attends
 - Team chooses a time that works for everyone
 - Product Owner can attend, but doesn't speak
 - Everyone stands in a circle, facing each other (not facing the SM)
 - Lasts 15 minutes or less
- Everyone reports 3 things only to each other
 - What was I able to accomplish since last meeting
 - What will I try to accomplish by next meeting
 - What are my blocks / problems / difficulties
- No discussion or conversation until meeting ends

Not allowed



Perfect Stand up



Sprint Review

- Purpose of the Sprint Review
 - Demo what the team has built
 - Make visible whether the team completed what they set out to
 - Generate feedback, which the Product Owner can incorporate in the Product Backlog
- Attended by Team, Product Owner, ScrumMaster, functional managers, and any other stakeholders
- A demo of what's been built, not a presentation about what's been built
 - no Powerpoints allowed!
- Usually lasts 1-2 hours
- Followed by Sprint Retrospective

Sprint Retrospective

- **What is it?**
 - 1-2 hour meeting following each Sprint Demo
 - Attended by Product Owner, Team, ScrumMaster
 - Usually a neutral person will be invited in to facilitate
 - What's working and what could work better
- **Why does the Retrospective matter?**
 - Accelerates visibility
 - Accelerates action to improve