

LEAN SOFTWARE DEVELOPMENT

Part 1: from continuous integration to continuous delivery

Part 2: continuous delivery: software projects & baby steps

Part 3: agile testing

TODAY'S AGENDA

- **Exercise debrief & takeaways**
- **Agile Testing**
 - Software quality and testing
 - Agile testing quadrants and selected testing methods

Exercise debrief & take-aways



Implement and deploy a "plain vanilla web site", with only a link to "meteosuisse.admin.ch"



1 day

Mechanism: make it possible to execute a task in our back-end at a fixed schedule (e.g. every day at the same hour)



2 days

Mechanism: our back-end is able to send an email to a registered user.



1 day

Combine the "scheduling task" and the "send email" mechanisms to send a daily email notification to all users.



1 day

- The team is "**in movement**".
- It **does not feel stuck** in a long analysis phase, during which all problems need to be solved.
- The first deliverable, even if it is very basic, increases the **team confidence**. The way by which the team organizes itself during the project is an important outcome.

- On the **technical side**, we can already validate a couple of important **capabilities** (scheduling, mail).
- The team can already **start using the service**. Some errors (with some e-mail addresses, with devices, might be detected early thanks to this).

Week 2

Add styles and images to make a "plain vanilla web site" more appealing



3 days

Mechanism: our back-end is able to send a SMS to a registered user. We use the SmartSMS commercial service (0.10 per SMS)



2 days

Week 3

Fetch weather data from the WeatherMaster.com API. Quality is high, cost is 49 USD/month.



3 days

Display the raw data provided by a Weather API in the web site



1 day

Implement basic Web analytics to know how many users come back to the website



1 day

- In the **"lean" approach**, it is important to **collect metrics** in order to validate design choices. For this reason, we implement the web analytics tool early in the project.

- At this stage, we should not be concerned with the API price. If we can prove that the product has a market, we will be able to **optimize costs later. For now, speed matters.**

Implement a basic iOS app. The user can register, give his phone number and email address. There is a link to meteosuisse.



5 days

- Should we do the iOS and Android development **in parallel or in sequence?**
- If we do it in sequence, should we start with iOS (**more costly, more users**) or with Android (**cheaper, smaller market**)?
- We are still very early in the process and **we still don't know if there is a market for this product**. So let's pick one platform. The platform that will allow us to get as many users as possible.
- **After 3 weeks, we have a mobile app that we should test with internal and “friendly” users.**

Implement an analytics tools so that we can analyze the poll results.



5 days

Poll feature: in the iOS app, we can ask a question to users and collect their answers.



4 days

Combine the "scheduling task" and the "send SMS" mechanisms to send a daily email notification to all users.



1 day

- As previously stated, we want to be able to **observe users and to gather feedback**.
- Therefore, we invest time to put in the place the **infrastructure** for doing this.

- At the end of this iteration, we have a functional service, which we can reasonably release to “beta” users.
- We don’t have fancy prediction algorithms... but does it matter? Is it a problem if everybody receives the SMS at 5:30 AM?

Mechanism: our back-end is able to send an iOS push notification



4 days

Combine the "scheduling task" and the "push" mechanisms to send a daily notification to iOS users.



1 day

Display the raw data provided by a Weather API in the iOS app



2 days

Implement a basic android app. The user can register, give his phone number and email address. There is a link to meteosuisse.



3 days

Poll feature: in the Android app, we can ask a question to users and collect their answers.



2 days

Mechanism: our back-end is able to send an Android push notification



2 days

Display the raw data provided by a Weather API in the android app



1 day

Implement an algorithm, which computes the probability that it will rain today for a given registered user. We use the data provided by an API.



5 days

Combine the "scheduling task" and the "push" mechanisms to send a daily notification to Android users.



1 day

Use the output of our "forecast" algorithm and display the information on the web site.



2 days

Use the output of our "forecast" algorithm and display the information in iOS



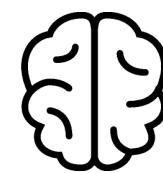
1 day

Use the output of our "forecast" algorithm and display the information in Android



1 day

Implement an algorithm, which predicts at which time a registered user will leave home today.



5 days + 5 days + 5 days

Mechanism: our back-end is able to send a SMS to a phone number. We use the OpenSMS free service



4 days

Implement and deploy a web site visually appealing, with only a link to "meteosuisse.admin.ch"



3 days

This is a typical example of a “feature” that **seems cool and valuable**. It would certainly be exciting for developers... but does it really make the product better?

Ok, it's free. But at this stage, is it not better to **pay a couple of bucks** to have something working quickly? And is it more expensive than a developer's salary?

Is it better to deploy an “ugly” site in week 1 or a “pretty” site in week 2? It is a trade-off, but **setting a rhythm for the team is incredibly valuable**.

SUMMARY

- **Small incremental releases** have **external** and **internal** benefits.
- **Externally**, they make it possible to **observe user behavior and to collect feedback** early and often. Furthermore, the team will be able to respond to new user needs (or to a better understanding of user needs) more quickly.
- **Internally**, they give a **rhythm** to the team and makes progress more visible. They also **drive the team towards continuous delivery**. The software is already in a “releasable” state and new ideas/features can be developed and pushed towards users quickly.
- With very frequent releases, you don’t really have a choice: you need to automate a lot of the validation and deployment procedures. This creates a **virtuous cycle**.

Agile Testing



Your PC ran into a problem that it couldn't handle, and now it needs to restart.

You can search for the error online: [INITIALIZATION_OF_BASS_DROP_FAILED](#)



PRODUCT DEVELOPMENT

Analyze

Specify

Build

Test



So... anything wrong with that?

PRODUCT DEVELOPMENT

- How much **time** do we need to go through all the phases?
- How often do we go through these phases (is it a **cycle**)?
- How much “**functionality**” is moving through these phases?



- **Who is working in each phase?**
- **How do people collaborate?**

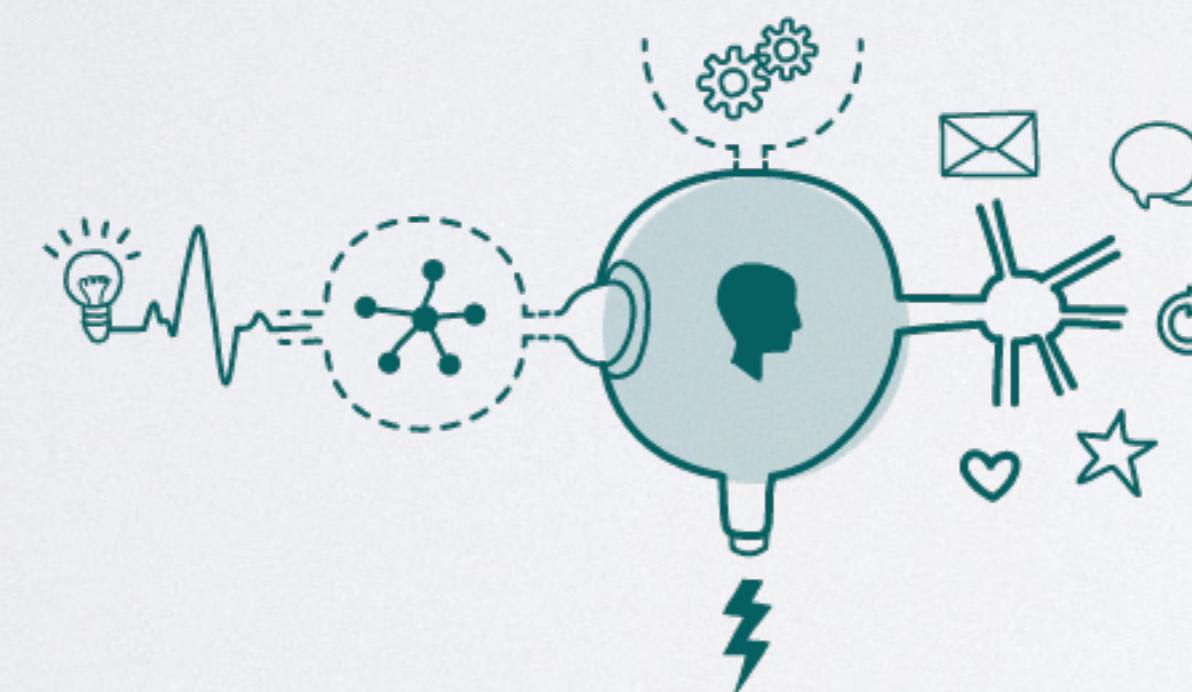
PRODUCT DEVELOPMENT

Analyze

Specify

Build

Test

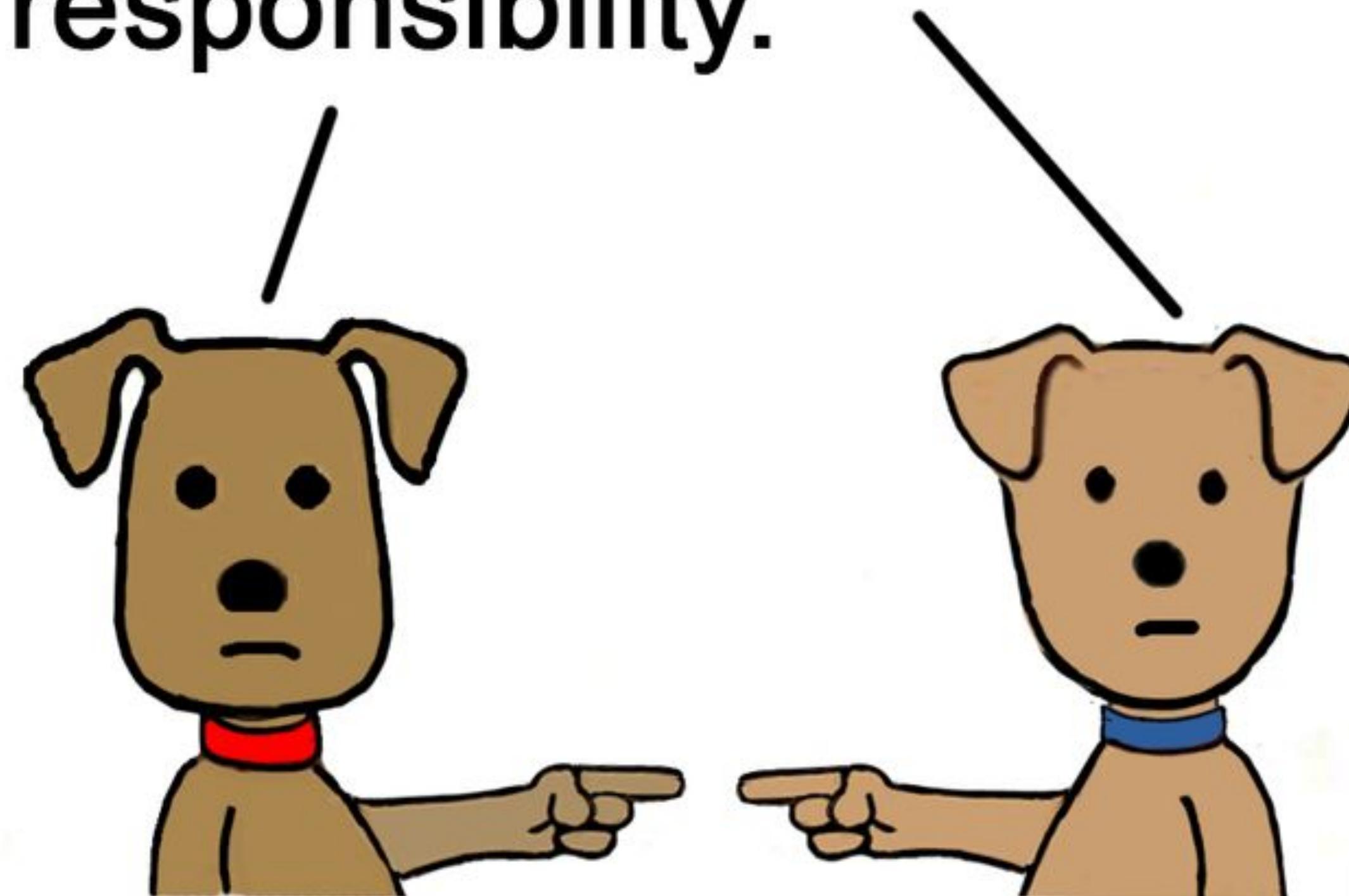








**Don't look at me, it's his
responsibility.**



Browman

*“One of the biggest differences in agile development versus traditional development is the “**whole team**” approach.*

*With agile, it’s not only the testers or a quality assurance team who feel responsible for quality. [...]. **Everyone on an agile team gets “test-infected”**. Tests, from the unit level on up, drive the coding, help the team learn how the application should work, and let us know when we’re “done” with a task or story.”*

*Render by
StylicSh*



RISE of the
SILVER SURFER

Small autonomous teams, which are given the responsibility of a complete “product” are very effective.

Communication, collaboration and coordination are much easier than across **organizational silos**.

People are much more likely to feel **empowered** and **accountable**.

If you can't feed a team with two pizzas, it's too large.

Startup Quote!



JEFF BEZOS
FOUNDER, AMAZON

Agile Testing Quadrants

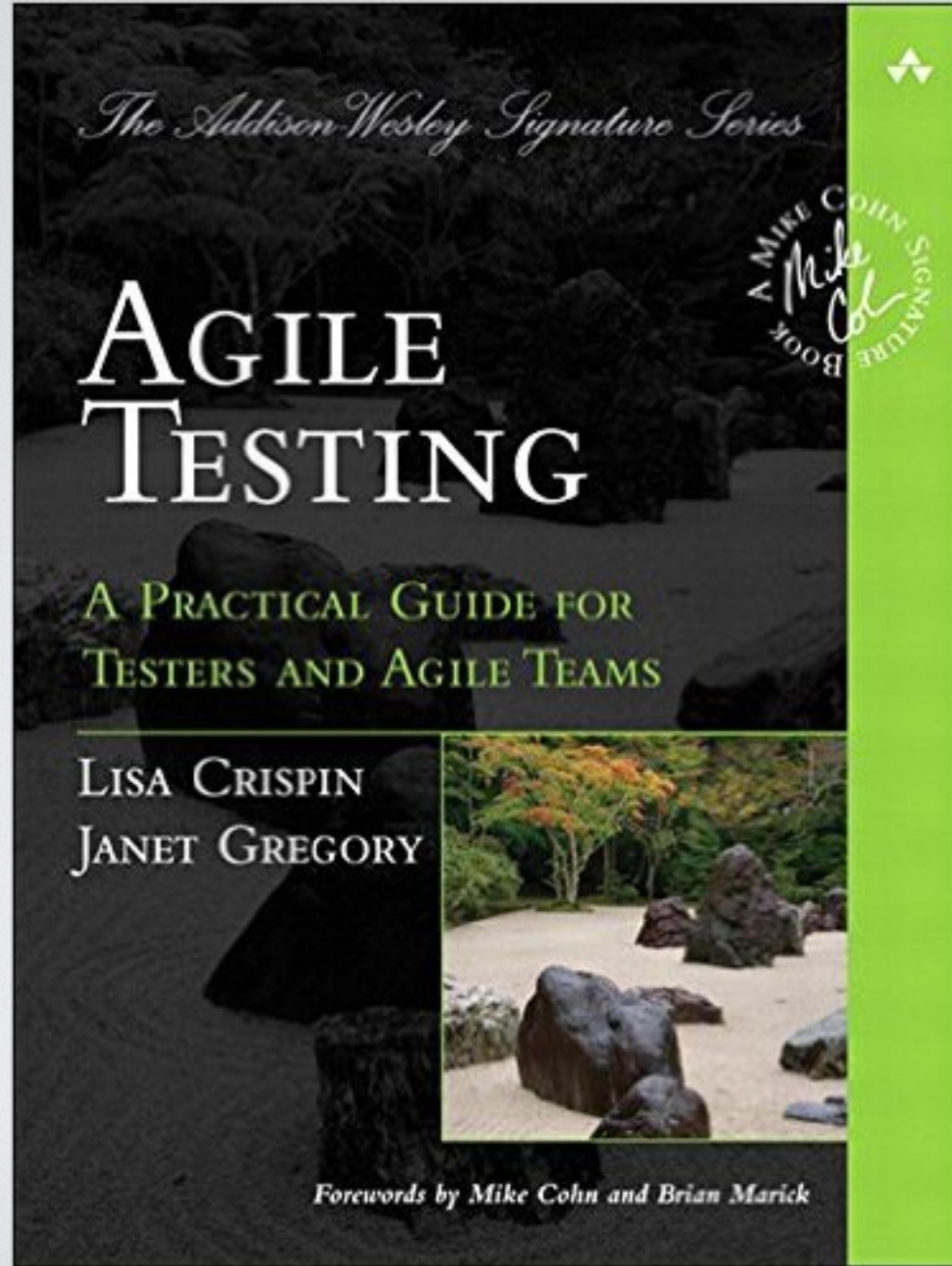
“Software quality” is a **broad concept** and has many aspects (reliability, efficiency, usability, maintainability, etc.).

“Software testing” refers to methods and techniques for **assessing** certain aspects of the quality of a software system. **There are many, many of them.**

Some “**Software testing**” techniques do not only measure quality after the fact, but **help the team to proactively** maintain the quality of the software to an appropriate level.

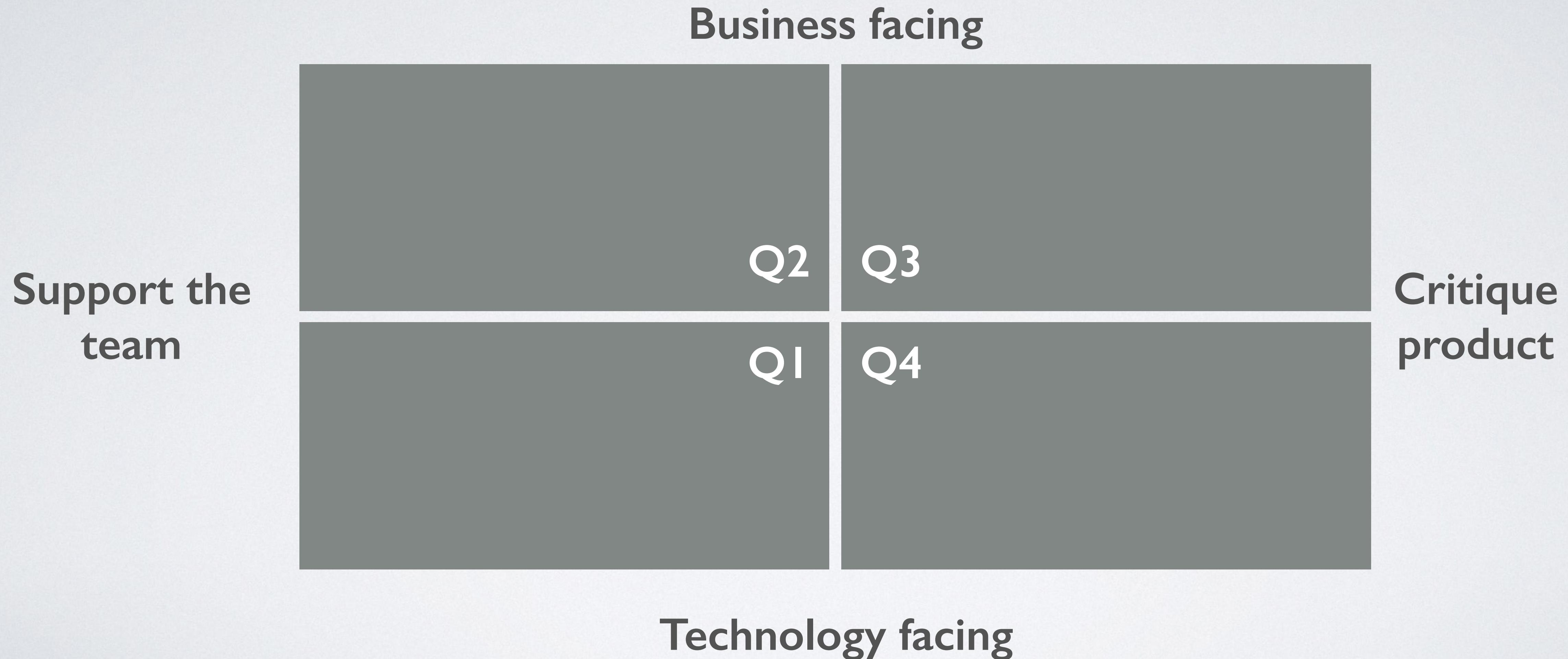
Is there a way to **classify** all these methods, so that we can see how they relate to each other?

These techniques are aligned with the principles of **agile** software development.



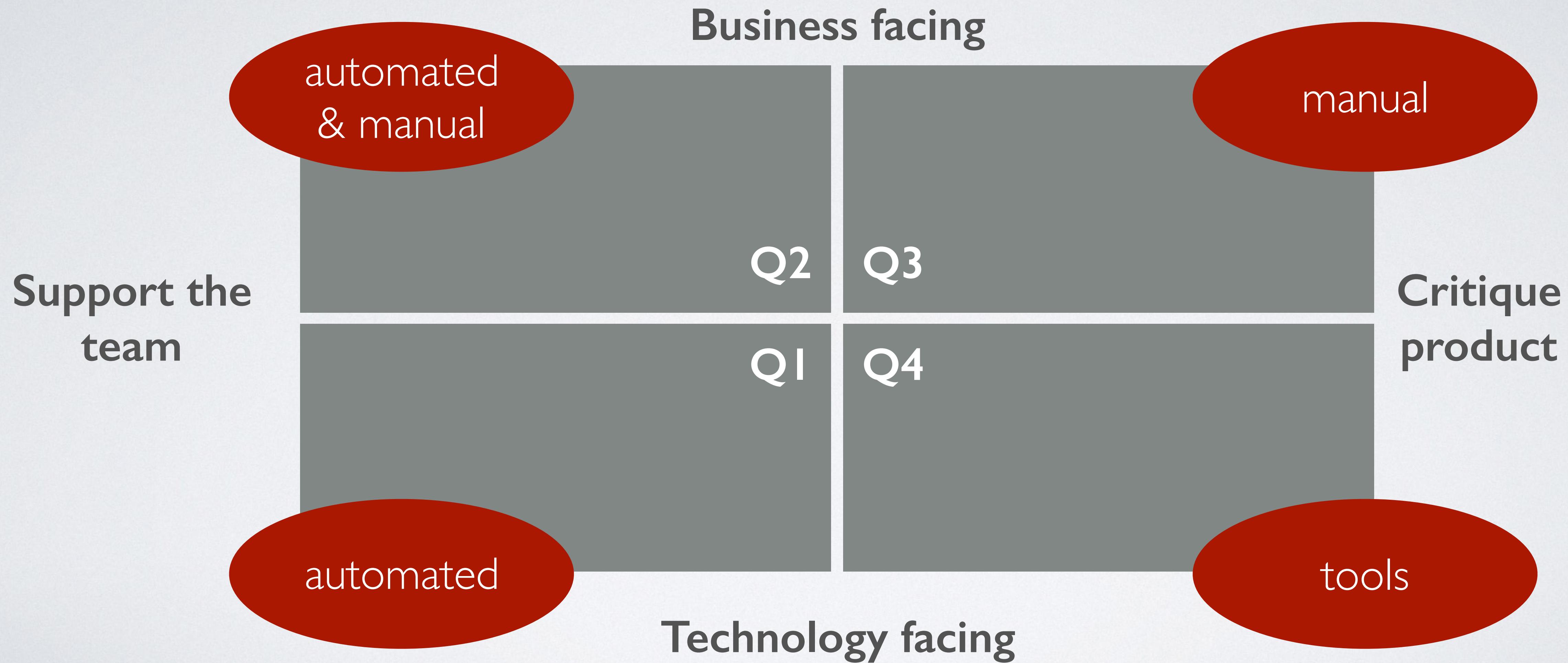
<https://www.youtube.com/watch?v=bqrOnIECCSg>

AGILE TESTING QUADRANTS



Support the team	Some of these tests help individual team members while they do their job. Sometimes, creating a “test” helps me specify and/or design the product. Other tests facilitate team collaboration , especially between “business” and “technical” people (shared language).
Critique product	Some of these tests allow humans to evaluate the quality of a software from the users point of view (is it easy to use? is it easy to learn? does it solve the user’s problem?). Other tests aim to detect issues with non-functional (systemic) qualities .
Technology facing	Some tests are created and executed by technical team members . They are highly automated. They relate to the “Are we building the product right?” question.
Business facing	Some tests are created by (or at least with) business-oriented team members . They also relate to the “Are we building the right product?” question.

AGILE TESTING QUADRANTS



AGILE TESTING QUADRANTS: Q1

**Support the
team**

Business facing

**Unit tests
Integration tests**

**Critique
product**

Technology facing

AGILE TESTING QUADRANTS: Q2

**Support the
team**

Business facing

Functional tests
Examples
Prototypes
Simulations

**Critique
product**

Technology facing

AGILE TESTING QUADRANTS: Q3

Support the
team

Business facing

Exploratory testing
Usability testing
User Acceptance Testing

**Critique
product**

Technology facing

AGILE TESTING QUADRANTS: Q4

Support the
team

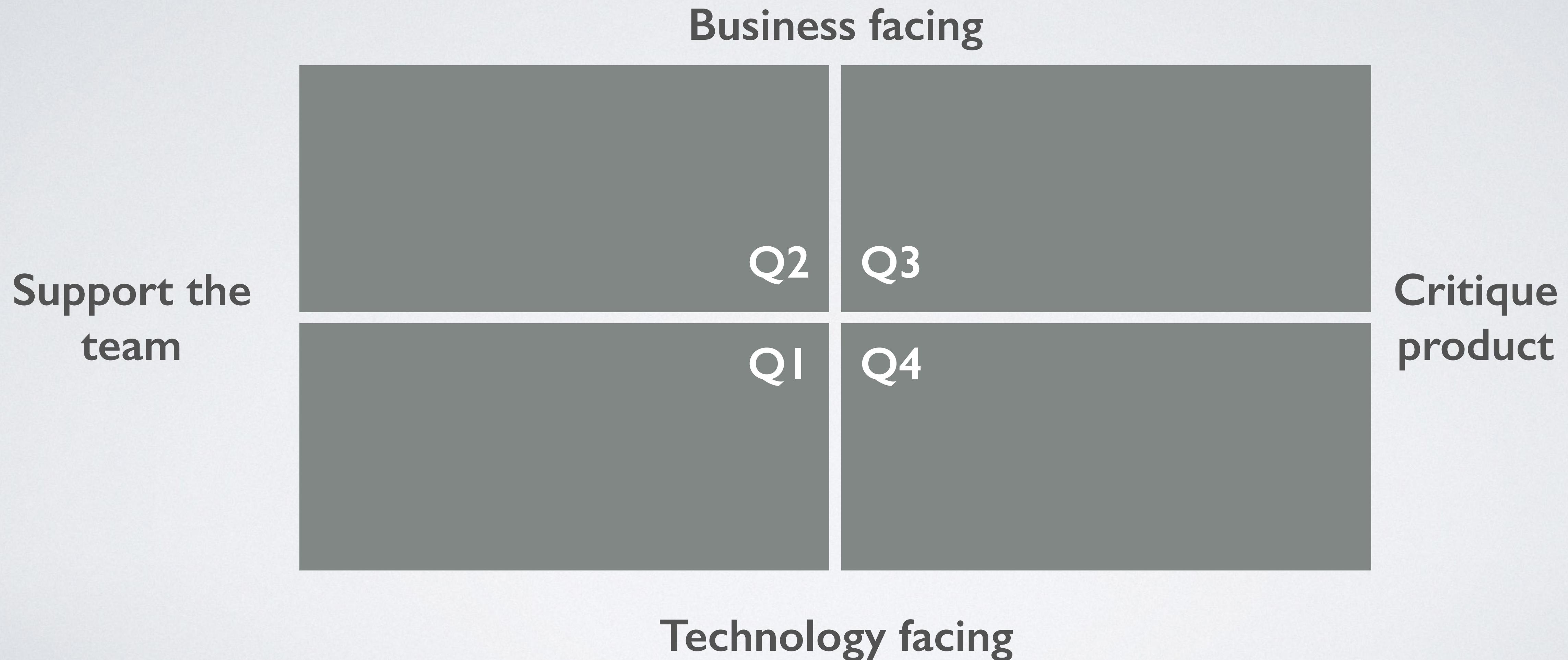
Business facing

Performance tests
Security tests
Fault-tolerance tests

Critique
product

Technology facing

AGILE TESTING QUADRANTS



AGILE TESTING QUADRANTS: Q1

**Support the
team**

Business facing

**Unit tests
Integration tests**

**Critique
product**

Technology facing

UNIT TESTS

- Programmers use a **programming language** to write code in “source files”. These “source files” are later translated in a language that can be understood by a machine.
- In a particular software system, there might hundreds or thousands of source files. **Every source file is made of “blocs”**. Depending on the programming language, we speak of “classes”, “functions”, “methods”, etc.
- A **Unit Test** is a software function that validates one aspect of one specific “bloc” in the tested system.

Unit

```
public int add(int a, int b) {  
    return a + b;  
}
```

Unit Test

```
public int add(int a, int b) {  
    return a + b;  
}
```

```
@Test  
public void testAdd() {  
    int result = add(5, 3);  
    assertEquals(8, result);  
}
```

When you write a unit test, you **invoke the unit** (call the function) and make a number of **assertions** on the result. By doing that, you **compare your expectations with the actual behavior** of the tested unit.

Unit

```
public int add(int a, int b) {  
    return a + b;  
}
```

```
@Test  
public void testAdd() {  
    int result = add(5, 3);  
    assertEquals(8, result);  
}
```

Unit Test #1

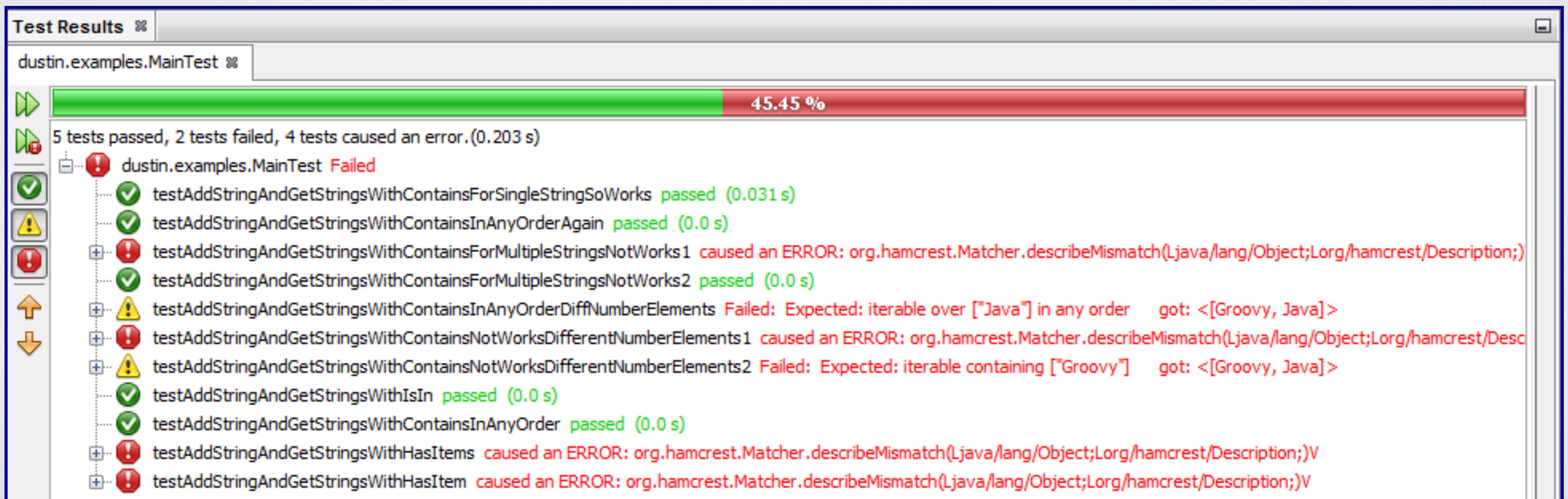
```
@Test  
public void testAdd_2() {  
    int result = add(-2, 2);  
    assertEquals(0, result);  
}
```

Unit Test #2

Developers use **Integrated Development Environment** (IDEs), such as IntelliJ, Netbeans or Eclipse to write their code.

In general, they use the same tool to **code the application** and to **code the unit tests** that validate the application components.

Whenever they modify their code, they can re-run the tests and get a “**green light / red light**” status in **milliseconds or seconds**. Even if there are thousands of unit tests.

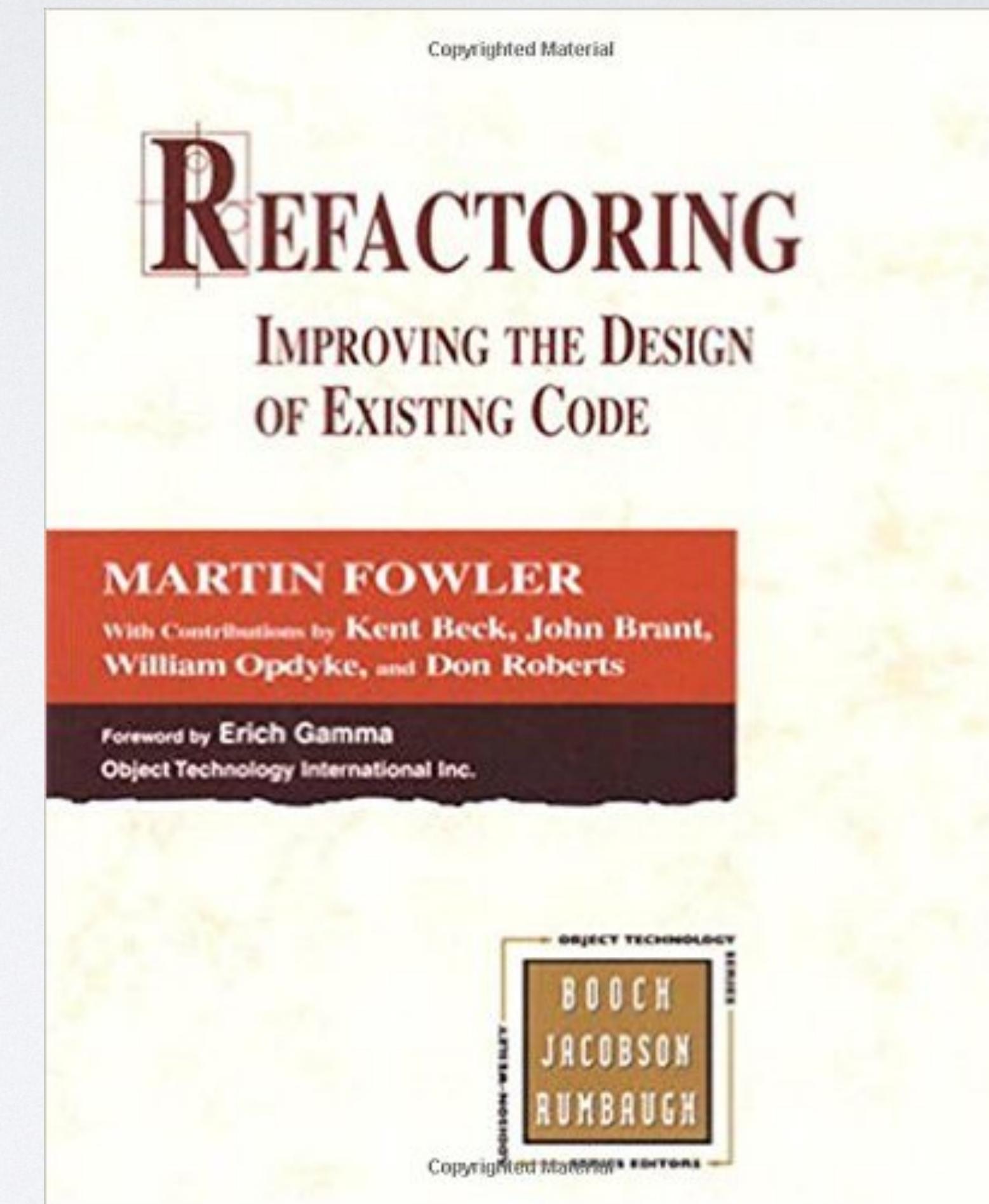




UNIT TESTS ENABLE REFACTORING

*“Refactoring is a disciplined technique for **restructuring an existing body of code**, altering its internal structure **without changing its external behavior.**”*

<http://www.refactoring.com/>



The **code works** (all tests are green) **but is a mess**. It is difficult to maintain and makes the development of new feature slow.



The code is now **nicely structured**. It is meant to do the same thing as before (no behavior change). Because all my unit tests are still green, I have some confidence that this is true.



```
✓ testArguments passed (3.592 s)
✓ testInputOutput passed (0.924 s)
✓ testWelcome passed (2.208 s)
✓ testQuote passed (8.346 s)
✓ testSubProjects passed (3.608 s)
✓ testPi passed (0.598 s)
✓ testFreeway passed (0.012 s)
✓ testFractal passed (15.9 s)
✓ testLexYacc passed (1.363 s)
✓ testMP passed (4.338 s)
✓ testHello passed (0.012 s)
✓ testHelloQtWorld passed (0.009 s)
✓ testProfilingDemo passed (0.025 s)
```

```
✓ testArguments passed (3.592 s)
✓ testInputOutput passed (0.924 s)
✓ testWelcome passed (2.208 s)
✓ testQuote passed (8.346 s)
✓ testSubProjects passed (3.608 s)
✓ testPi passed (0.598 s)
✓ testFreeway passed (0.012 s)
✓ testFractal passed (15.9 s)
✓ testLexYacc passed (1.363 s)
✓ testMP passed (4.338 s)
✓ testHello passed (0.012 s)
✓ testHelloQtWorld passed (0.009 s)
✓ testProfilingDemo passed (0.025 s)
```

INTEGRATION TESTS

- Integration Tests are another type of **automated tests**.
- They validate the **interactions between several units**.
 - Does my “prediction algorithm” work with my “database service”?
 - Does my “web front-end” work with my “REST API”?
- Integration tests are **slower than unit tests**. We do not run them as often as unit tests (not to slow developers down), but we run them very often (at least before sharing one’s code with the team).

AGILE TESTING QUADRANTS: Q2

**Support the
team**

Business facing

Functional tests
Examples
Prototypes
Simulations

**Critique
product**

Technology facing

FUNCTIONAL TESTS

- With **functional tests**, we want to validate that the system does what it is supposed to do **from the users point of view.**
- Very often, this means **defining usage scenarios (test cases)**. We describe the steps to be followed by users and the expected results.
- When we evaluate a software release, we can **check** whether the defined test cases can be executed with success.

MANUAL FUNCTIONAL TESTS

- In many organizations, test cases are documented in **test management software**. They are executed by **human operators**.
- This is a **repetitive process** with little added value.
- This is a **slow process**.
- It creates **overhead** and often gives a **false sense of confidence**.
- If you release every 3 months, it “might” work. If you release on a weekly basis, it is just not possible.



AUTOMATED FUNCTIONAL TESTS

- There are now **tools** that can be used to **simulate human users**.
- With these tools, you write scripts. When the scripts are executed, they **control a web browser** and check that the content of the pages is.
- **It is not a free lunch.** Writing these scripts takes time. Maintaining these scripts (when the UI changes) takes a lot of time.
- Integration tests are slower than unit tests. Automated functional tests are **a lot slower** than integration tests.
- For this reason, they are not executed as often (at a later stage in the continuous delivery pipeline).

Workflow of Selenium Webdriver



© <http://www.helloselenium.com>

BEHAVIOR DRIVEN DEVELOPMENT

- With Unit Tests, developers have a way to **specify and check** the behavior of a tiny piece of code.
- The same principle can be applied with higher-level, **business oriented tests**. This is the idea of “behavior driven development” or BDD.
- BDD is a method that **facilitates the collaboration** between business analysis, developers and testers. It gives them a **common vocabulary**.

- BDD proposes a **template** to describe the intended behavior of a system. The template is used to specify the acceptance criteria for a given user story.

Given some initial context (the givens),
When an event occurs,
then ensure some outcomes.

USER STORY
As a customer,
I want to withdraw cash from an ATM,
so that I don't have to wait in line at
the bank.

ACCEPTANCE CRITERIA

ACCEPTANCE CRITERIA

ACCEPTANCE CRITERIA

Given the account is in credit
And the card is valid
And the dispenser contains cash
When the customer requests cash
Then ensure the account is debited
And ensure cash is dispensed
And ensure the card is returned

- The specification is **human readable**. It can be written, or at least read, by business analysis and product owners.
- But because we use certain keywords and conventions, the specification can also be **interpreted by machines**.
- We can therefore “**execute the specification**” and trigger automated tests for all stories and acceptance criteria.

Scenario: trader is not alerted below threshold

Given a stock of symbol STK1 and a threshold of 10.0

When the stock is traded at 5.0

Then the alert status should be OFF



Executable
Specifications



AGILE TESTING QUADRANTS: Q4

Support the
team

Business facing

Performance tests
Security tests
Fault-tolerance tests

Critique
product

Technology facing

SYSTEMIC QUALITIES

- A common problem is that developers focus on the functionality (**what** the software does), but forget about the non-functional aspects (**how** it does it).
- Things like performance, availability, security, scalability, etc.
- We often talk about “**“systemic qualities”** or “-ilities”

A SAD STORY...

Developer	I have modified the “search feature”, so that it now ranks the results based on the user profile (e.g. preferences, past history, etc.)
Product Owner	Cool, let's release it!
User	Why is it search feature so slow today?
System Administrator	Why is all the CPU and memory in red on the server?

An algorithm may work well if it is applied on a database that contains **1'000** entries. But it may break if it contains **1'000'000** entries.

A feature may be fast if it is used by a single user at the time. But it may extremely slow if is used by 10 **concurrent users**. It may even return wrong results in this case.

A web app may appear to be fast when the browser, the server and the database are all on the same (developer) machine. But may appear very slow when the components are **distributed across the Internet**.

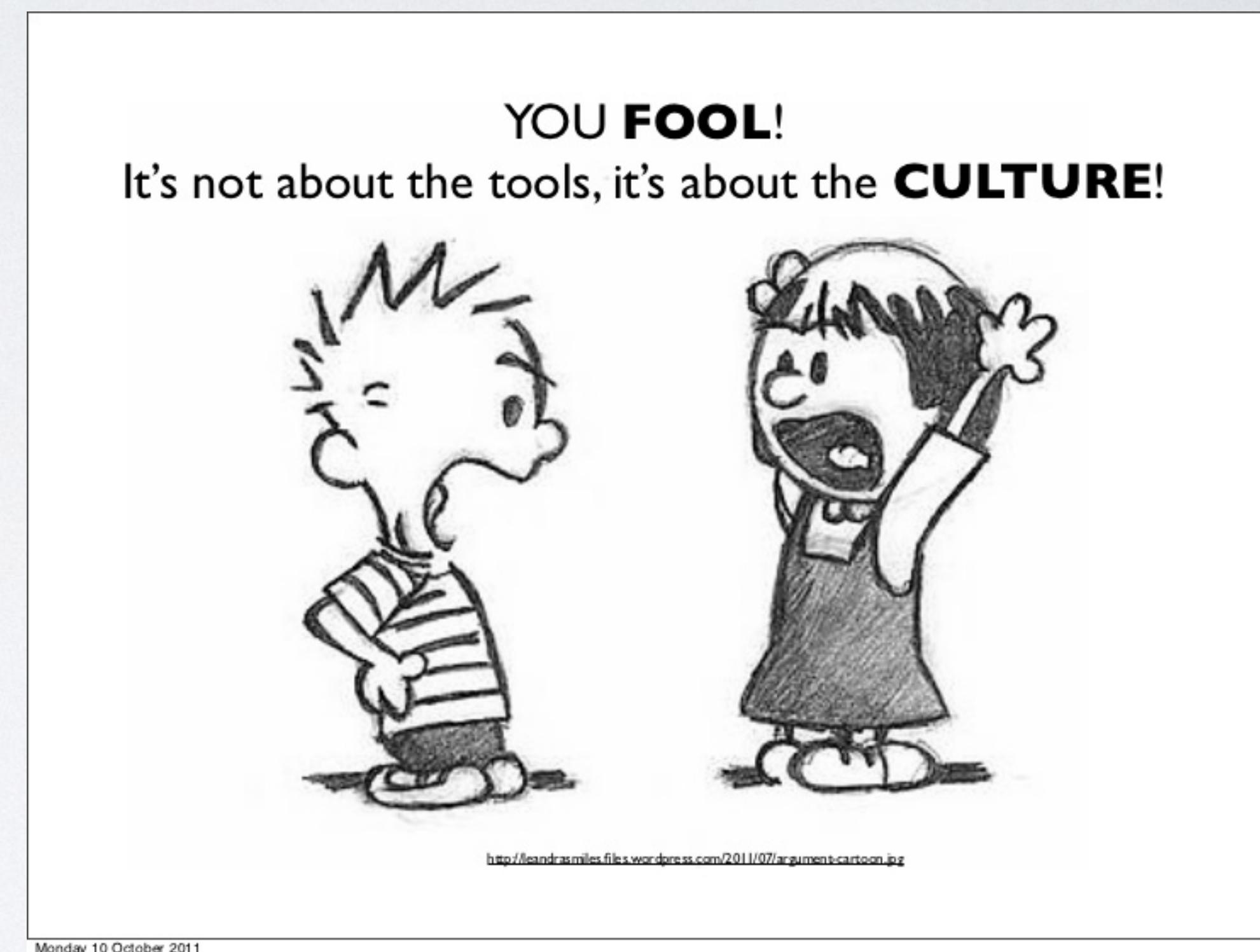


IT WORKS
on my machine

HOW TO AVOID THESE ISSUES (I)

- Increase **team awareness**

- Developers and product owners should be reminded about “**systemic qualities**” and the importance of **non-functional aspects**.
- With a “**whole-team approach**”, you are more likely to hear the point of view of system administrators very early. This will help a lot.



http://codeignition.co/blog/2015/09/17/dev_to_ops/

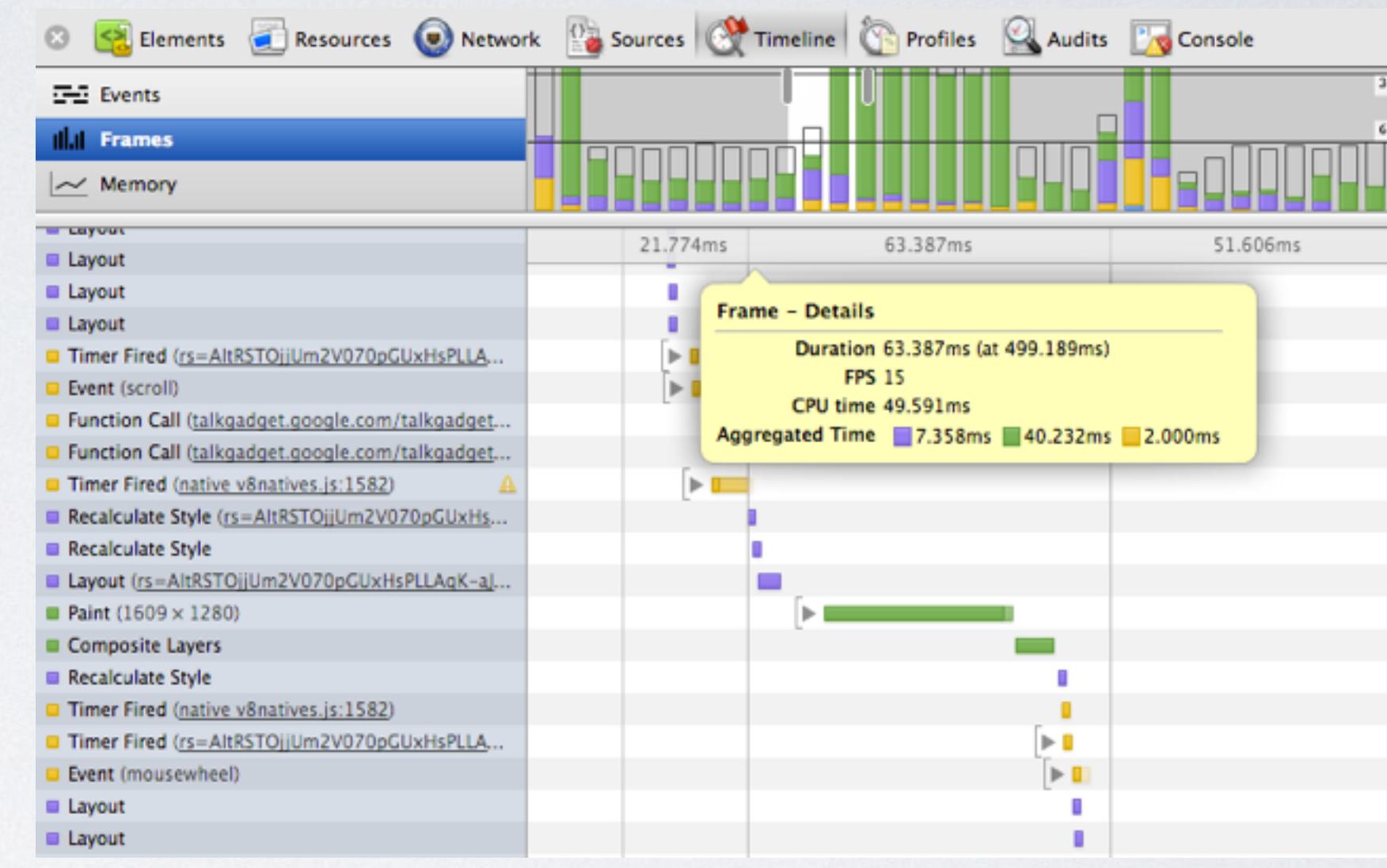
HOW TO AVOID THESE ISSUES (2)

- **Measure** early, measure often

- There are **many tools to measure performance** (response time) of a software system. There are also **many tools that simulate a large number of users** and to generate a heavy load on a test system.

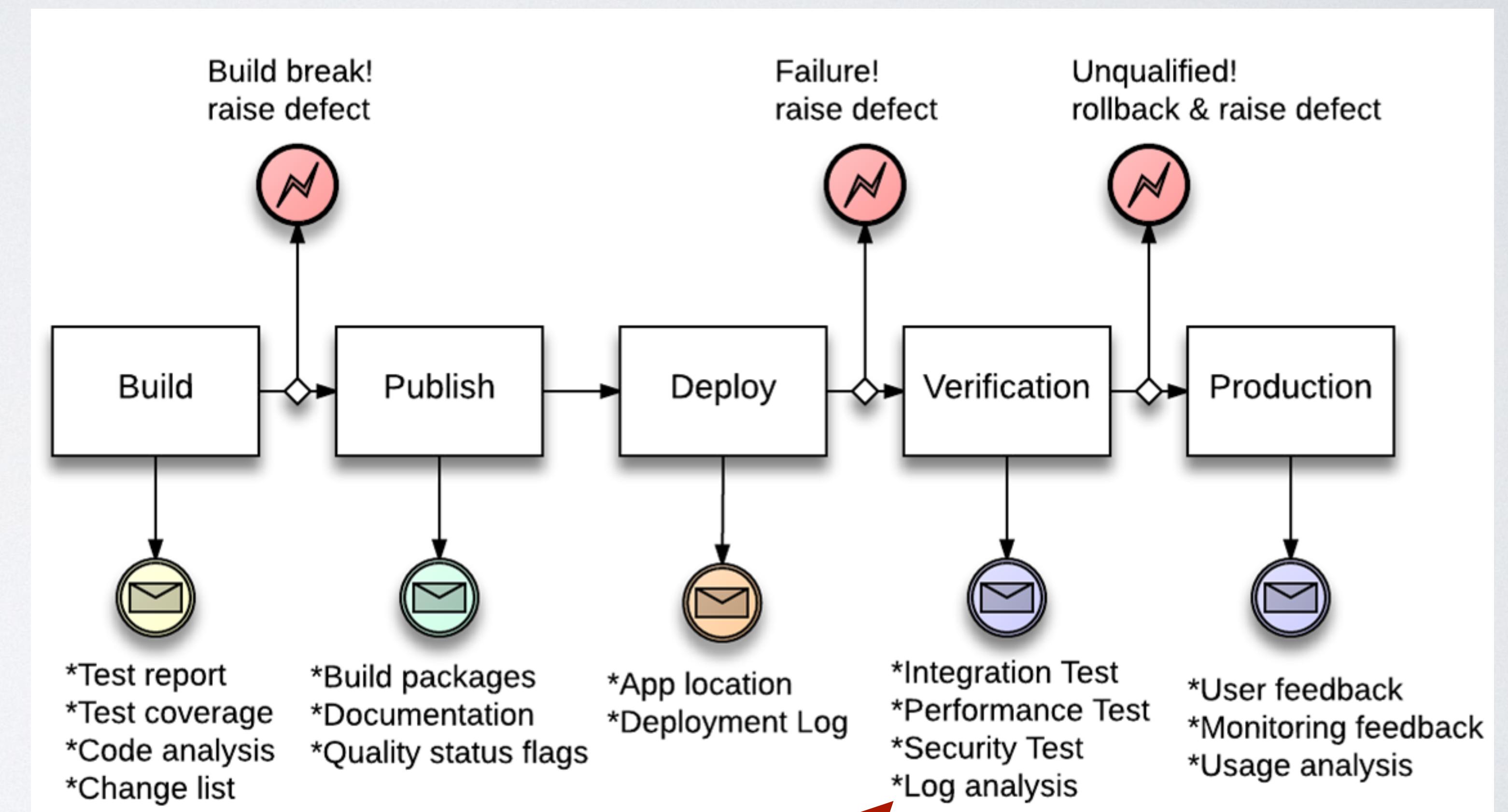
- Engineers can also **write custom programs and scripts** to do these experiments.

- **Allocate time and resources** to support these activities.



HOW TO AVOID THESE ISSUES (3)

- Integrate automated performance tests in your **continuous delivery pipeline**.
- Performance is only one aspect, you should also think about security, fault tolerance, etc.



AGILE TESTING QUADRANTS: Q3

Support the
team

Business facing

Exploratory testing
Usability testing
User Acceptance Testing

**Critique
product**

Technology facing

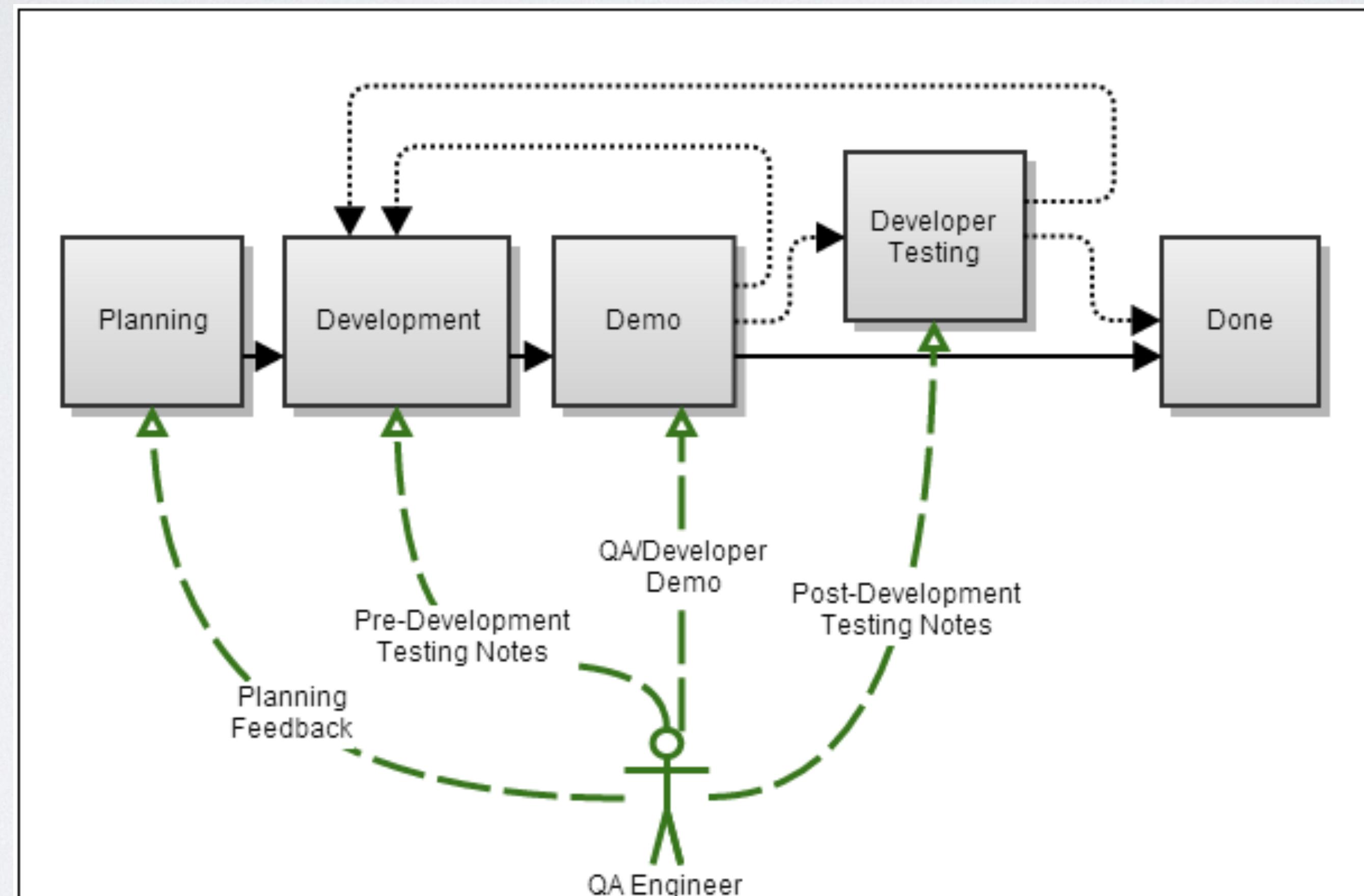
EXPLORATORY TESTING

- Exploratory testing is a **human activity** (which may involve the use of automated tests).
- Unlike scripted manual testing, it requires **strong skills and expertise**.
- It fits well with the “**whole-team**” approach (handing over the system to an external testing team is not efficient).

“A style of software testing that emphasizes the **personal freedom and responsibility** of the individual tester to continually optimize the quality of his/her work by treating **test-related learning, test design, test execution, and test result interpretation** as mutually **supportive activities** that run in parallel throughout the project.”

Cem Kaner

JIRA DEVELOPMENT PROCESS



<http://blogs.atlassian.com/2013/12/jira-qa-process/>

JIRA DEVELOPMENT PROCESS

“Once a developer is happy that they have completed a story, he or she will call over their QA engineer for a **demo session**.

The demo is **a discussion between equals**, and not a test that the developer or story can “pass” or “fail.” However, sometimes concerns are noticed during the demo, and these get quickly noted as comments on the story.

At the end of the demo, the QA engineer and developer will make a **joint decision** on what should happen to the story next.”

JIRA DEVELOPMENT PROCESS

“The Atlassian QA strategy is more about prevention and trust.

As QA, we know the roads really well – we know what causes accidents, and where the accidents have been in the past. **So instead of sitting by the road with a radar gun to catch speeding drivers, we sit in the cab with the drivers and say, “careful in this next bit, there’s a huge rock in the middle of the road after that blind corner.”** In safer stretches, we say, “There’ve been no nasty accidents in this stretch in the past, full speed ahead!””