Conduite de Projet
Cours 9 — Test-Driven Development

Stefano Zacchiroli
zack@irif.fr

Laboratoire IRIF, Université Paris Diderot

2017–2018

URL https://upsilon.cc/zack/teaching/1718/cproj/

Outline

Test-Driven Development (TDD)

Test-Driven Development, or TDD, is an iterative software development process which uses very short development cycles and leverages tests to provide constant feedback to software developers.

Goal: “clean code that works”, i.e., develop better software, less stressfully.

Test-Driven Development (TDD)

Test-Driven Development, or TDD, is an iterative software development process which uses very short development cycles and leverages tests to provide constant feedback to software developers.

Goal: "clean code that works", i.e., develop better software, less stressfully.

The "Test-Driven Development" expression is often (ab)used to talk about 2 distinct things:

- the TDD development process
- the xUnit family of testing frameworks
  - e.g., JUnit, Python's unittest, Check, OUnit, ...

  which have been designed to support the TDD development process, but can also be used when adopting different development process

xUnit

xUnit collectively refers to a set of frameworks for automated unit testing which share a common test coding style.

Each xUnit framework includes:

test case	abstraction used to define tests
test suite	abstraction used to organize test in test suites
assertion API	to implement test case oracles to verify outcomes
test fixture	mechanisms to factorize test initialization and clean up code
test runner	end-user program to discover and (selectively) run test suites
test result formatters	summarize test results and present them for human consumption

xUnit implementations

Teting frameworks that adhere to the xUnit paradigm exist for most languages and platforms, e.g.:[1]

- SUnit (Smalltalk)
- JUnit (Java)
- CppUnit (C++)
- OUnit (OCaml)
- Test::Unit (Ruby)
- HUnit (Haskell)
- NUnit (.NET)
- unittest (Python)
- Check (C)

1. https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

Outline

Development as learning

- every software development project tries *something* that has never been tried before
  - otherwise you wouldn't be doing it...
- due to constraints, developers often use technologies they don't completely master
  - new technologies, old technologies used in new contexts, etc
- all stakeholders (developers, managers, customers) learn as the project progresses

Problem: as we don't know everything at the beginning, there will be unexpected changes during the project.

How do we cope with them?

Nested feedback loops

Idea

1. we use empirical feedback to learn about the system
2. we store what we learn in the system itself, for future use

To do so, we organize development as nested feedback loops with increasing time periods and scopes in the organization (file, unit, product, team, etc.), e.g.:

- pair programming — period: seconds
- unit tests — seconds–1 minute
- acceptance tests — minutes
- daily meeting — 1 day
- iterations — 1 day–1 week
- releases — 1 week–months

We want feedback as quickly as possible. If something slips through an inner loop, it will (hopefully) be catched by an outer one.

Expecting the unexpected

Practices that (empirically) help coping with unexpected changes:

- constant testing
  - when we change something we might introduce regressions
  - to avoid that we need to constantly test our system
  - doing it manually doesn't scale ⇒ automated testing
- simple design

  keep the code as simple as possible
  optimize for simplicity
  - as we will have to change it, we want code that is easy to understand and modify
  - empirical studies show that developers spend more time reading code than writing it
  - clean design doesn't come for free, to achieve it we must constantly refactor
  - test suites give you *courage* to refactor, and apply other changes, thanks to their tight feedback loop

TDD principle

So we have test suites. Why do we need TDD?

TDD principle

So we have test suites. Why do we need TDD? Because:

- developers don't like writing tests
- testing is often seen as a 2nd class development activity

TDD principle

So we have test suites. Why do we need TDD? Because:

- developers don't like writing tests
- testing is often seen as a 2nd class development activity

TDD idea (i.e., a judo move on the above problem)

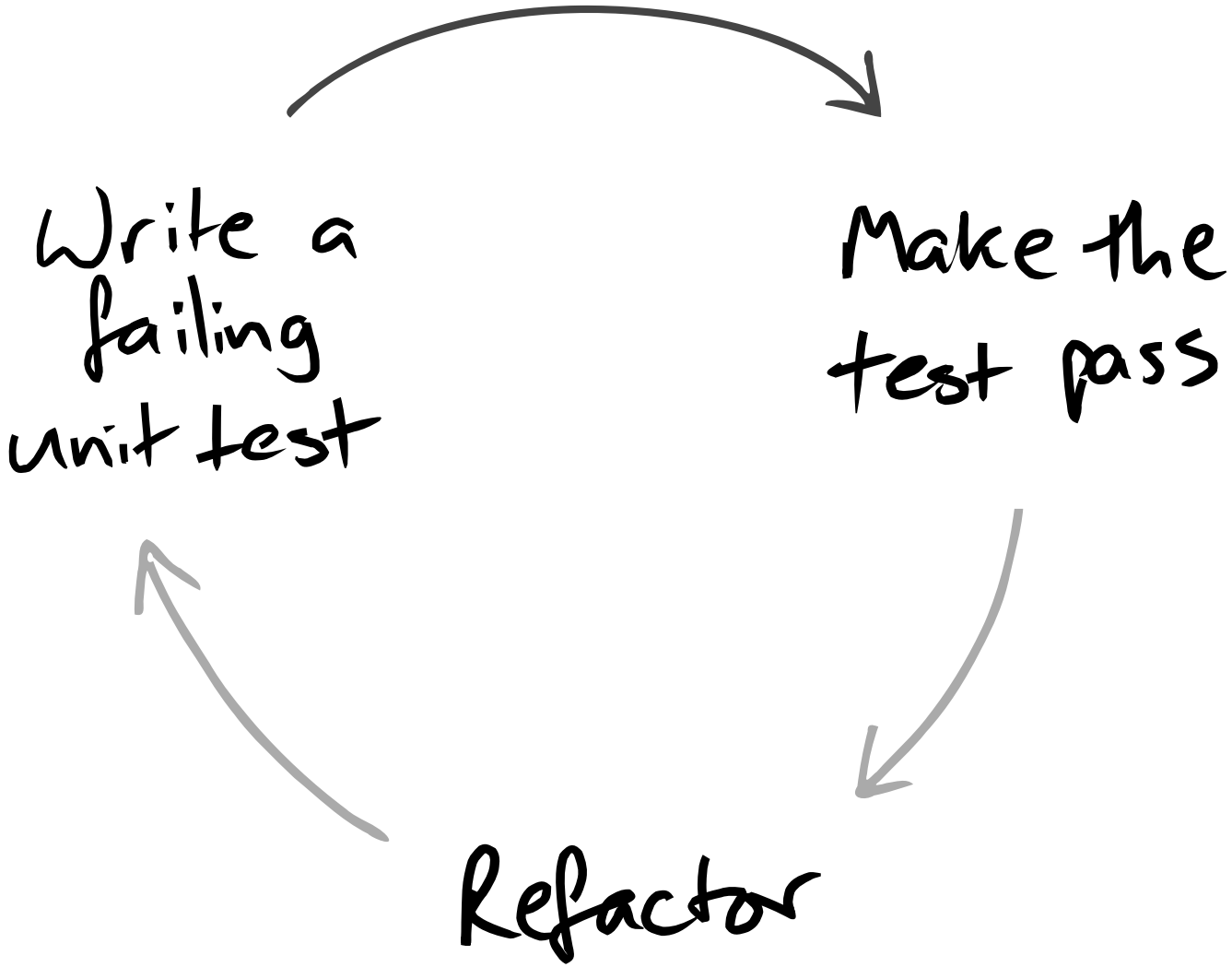
write tests *before* code

- don't write tests only to verify code *after* it's done
  leverage testing as a design activity
- write tests to clarify our ideas about what the code should do

  "I was finally able to separate logical from physical design. I'd always been told to do that but no one ever explained how."

  — *Kent Beck*
- write tests to get rapid feedback about design ideas
  - if *a test* is difficult to write, design is often wrong

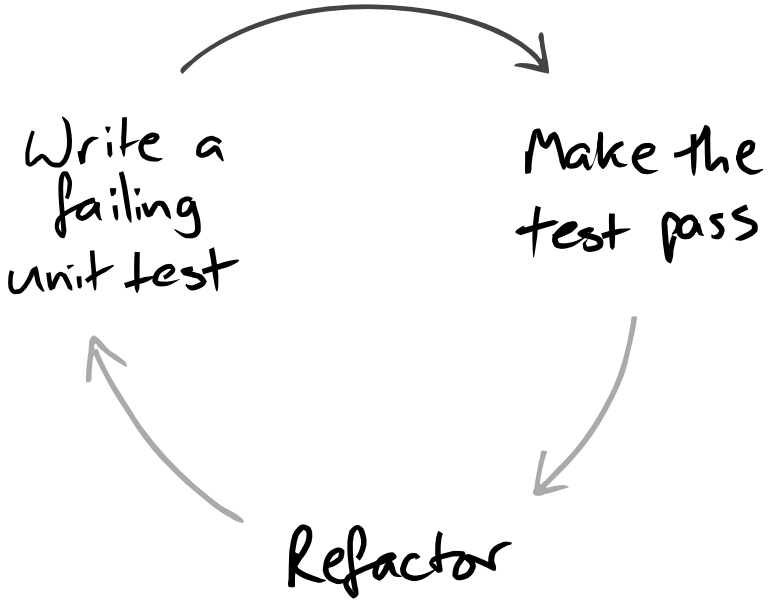
TDD in a nutshell



GOOS, Figure 1.1

The TDD development cycle

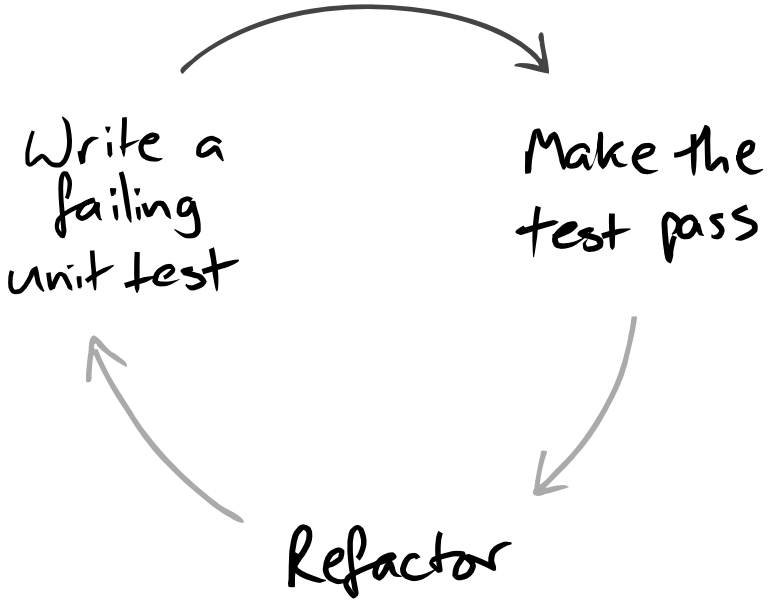
1. red: write a failing test
2. green: write code that makes it pass
3. refactor the code to be as simple as possible



GOOS, Figure 1.1

The TDD development cycle (cont.)

1. red: write a failing test
2. green: write code that makes it pass
3. refactor the code to be as simple as possible



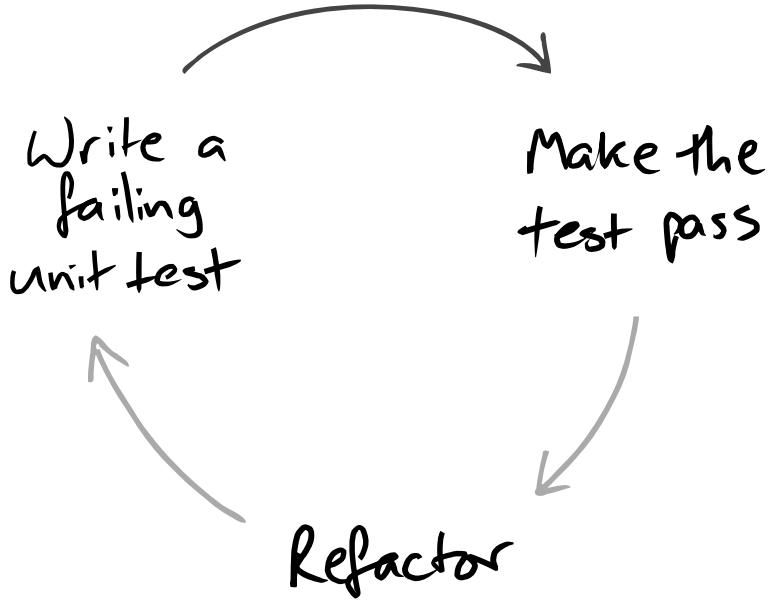
GOOS, Figure 1.1

By writing the test we:

- clarify acceptance criteria
- are pushed to design loosely coupled components
  - otherwise they are difficult to test
- document the code, via an executable description of it
- *incrementally* build a regression test suite

The TDD development cycle (cont.)

1. red: write a failing test
2. green: write code that makes it pass
3. refactor the code to be as simple as possible



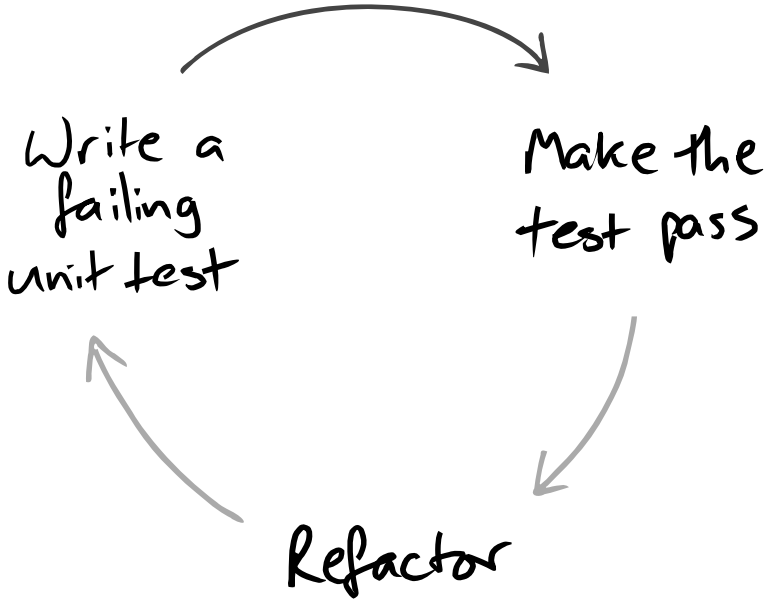
GOOS, Figure 1.1

By running the test we:

- detect errors when the context is fresh in our mind
- have a measure of progress, know when to stop (i.e., when we are "done enough")
  - avoid over-coding, "gold plating"

The TDD development cycle (cont.)

1. red: write a failing test
2. green: write code that makes it pass
3. refactor the code to be as simple as possible



GOOS, Figure 1.1

TDD golden rule

Never write a new functionality without a failing test.

Step 1 — write a failing unit test

This looks easy. But requires some care:

- write the test you want to read, ideally in 3 steps
  1. prepare test environment (input data and/or context)
  2. invoke the logic under testing
  3. verify that the results are correct

  If it cannot be done in a few lines (ideally: 3), write helper code.

  Remember: tests are documentation too.

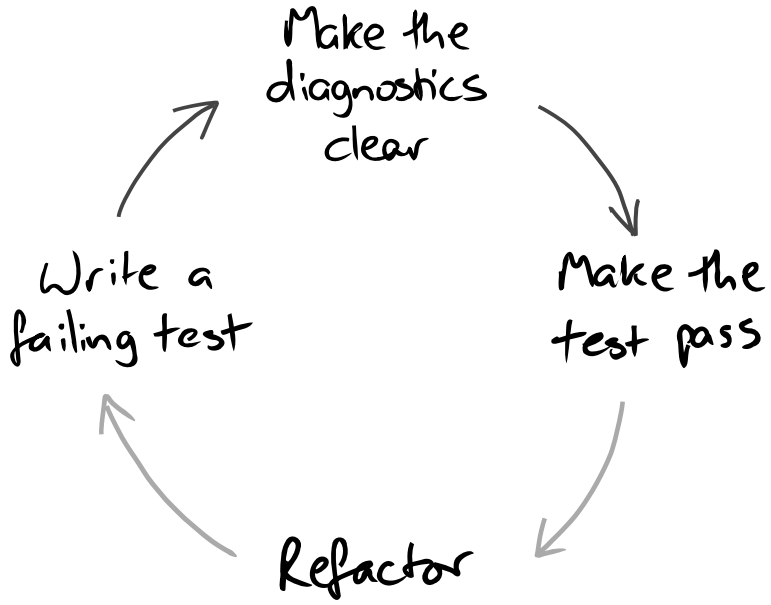
Step 1 — write a failing unit test

This looks easy. But requires some care:

- write the test you want to read, ideally in 3 steps
  1. prepare test environment (input data and/or context)
  2. invoke the logic under testing
  3. verify that the results are correct

  If it cannot be done in a few lines (ideally: 3), write helper code.

  Remember: tests are documentation too.
- watch the test fail *before* making it pass
  - otherwise you're not sure about your assumptions
  - if the test fails in unexpected ways, fix it (= the test)
  - if the diagnostic isn't clear, fix it



GOOS, Figure 5.2

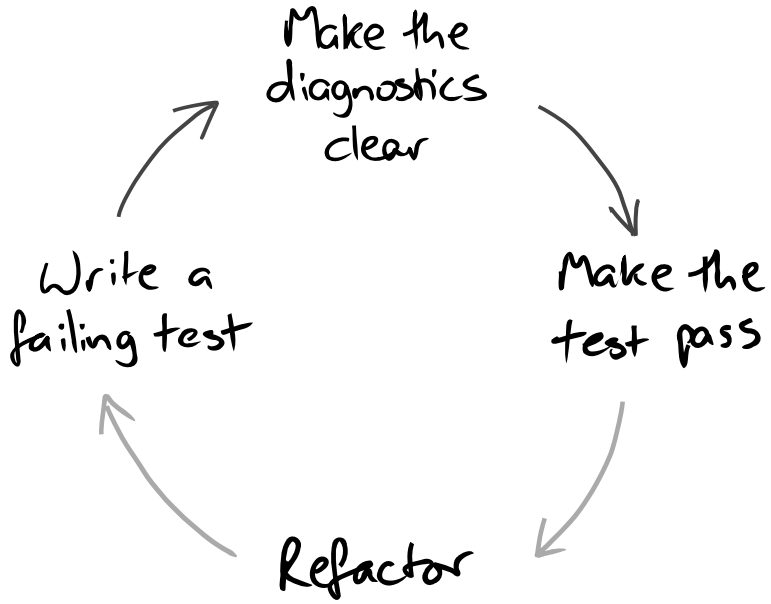
Step 1 — write a failing unit test

This looks easy. But requires some care:

- write the test you want to read, ideally in 3 steps
  1. prepare test environment (input data and/or context)
  2. invoke the logic under testing
  3. verify that the results are correct

  If it cannot be done in a few lines (ideally: 3), write helper code.

  Remember: tests are documentation too.
- watch the test fail *before* making it pass
  - otherwise you're not sure about your assumptions
  - if the test fails in unexpected ways, fix it (= the test)
  - if the diagnostic isn't clear, fix it
- unit test behavior, not functions
  - for TDD test coverage is less important than readable tests



GOOS, Figure 5.2

Step 2 — make the test pass (you won't like this)

To make the test pass we allow ourselves to take shortcuts.

Common strategies to make the test pass:

- fake it — all sorts of dirty tricks
  - e.g., return the constant value the test expects

Step 2 — make the test pass (you won't like this)

To make the test pass we allow ourselves to take shortcuts.

Common strategies to make the test pass:

- fake it — all sorts of dirty tricks
  - e.g., return the constant value the test expects
- obvious implementation — just type in the "obviously right" implementation
  - it takes experience to tune your confidence
  - too confident: you will have bad surprises
  - too prudent: you'll fake it too often
  - tip: use confidence increasingly, fall back when you get an unexpected "red bar" (i.e., test failure)

Step 2 — make the test pass (you won't like this)

To make the test pass we allow ourselves to take shortcuts.

Common strategies to make the test pass:

- fake it — all sorts of dirty tricks
  - e.g., return the constant value the test expects
- obvious implementation — just type in the "obviously right" implementation
  - it takes experience to tune your confidence
  - too confident: you will have bad surprises
  - too prudent: you'll fake it too often
  - tip: use confidence increasingly, fall back when you get an unexpected "red bar" (i.e., test failure)
- triangulation — when you have more than 2–3 tests that use the same implementation, factor out a common one
  - corollary: triangulation is commonly applied after several applications of the previous techniques

Step 3 — refactor

At this point: we have a test, some new code, and we are reasonably convinced that it is that code that makes the test pass.
We can now improve the code design, using tests as a safety net.

The goal of refactoring is to improve the design of existing code, without altering its external behavior (see Fowler 1999). We only give some of its intuitions here:

Code smells

- duplicate code
- long methods / large class
- too many parameters
- inappropriate intimacy
- Liskov principle violation
- complex conditionals
- ...

Techniques

- encapsulate field
- generalize type
- conditionals → polymorphism
- extract class / method
- rename method / field
- pull up / push down
- ...

TDD cycle — example

Goal: get a list of the elements contained in a binary tree

1. write a unit test that (pseudo-code)
   1. creates a binary tree containing the elements 2, 3, 1
   2. invokes a `toList` method on it
   3. asserts that `toList`'s return value = `[2; 3; 1]`

   Run all tests and ensures the new one fails *as we expect*
   - e.g., with a compilation error due to the lack of `toList`

TDD cycle — example

Goal: get a list of the elements contained in a binary tree

1. write a unit test that (pseudo-code)
   1. creates a binary tree containing the elements 2, 3, 1
   2. invokes a `toList` method on it
   3. asserts that `toList`'s return value = `[2; 3; 1]`

   Run all tests and ensures the new one fails *as we expect*
   - e.g., with a compilation error due to the lack of `toList`
2. implement `toList`
   - either by faking it: `return([2; 3; 1])`
   - or by writing the implementation you consider obvious

   Run all tests and ensures the new one succeeds

TDD cycle — example

Goal: get a list of the elements contained in a binary tree

1. write a unit test that (pseudo-code)
   1. creates a binary tree containing the elements 2, 3, 1
   2. invokes a `toList` method on it
   3. asserts that `toList`'s return value = `[2; 3; 1]`

   Run all tests and ensures the new one fails *as we expect*
   - e.g., with a compilation error due to the lack of `toList`
2. implement `toList`
   - either by faking it: `return([2; 3; 1])`
   - or by writing the implementation you consider obvious

   Run all tests and ensures the new one succeeds
3. refactor, e.g.:
   - write a proper implementation if you faked it
   - clean up, clean up, clean up

   Run all tests and ensures the new one still succeeds

TDD cycle — exercise

Exercise (bug fixing work-flow)

You have adopted TDD as the development process for your project. Describe the work-flow you would use to fix a bug.

?

TDD cycle — exercise

Exercise (bug fixing work-flow)

You have adopted TDD as the development process for your project. Describe the work-flow you would use to fix a bug.

Possible work-flow:

0. receive bug report
1. run all tests to ensure clean slate
2. create a new test that
   - recreates the context of the (alleged) bug
   - would succeed if the bug didn't exist
3. run all tests
   - new test fails → reproducible bug
   - new test passes → unreproducible bug → investigate with submitter

TDD cycle — exercise (cont.)

Exercise (bug fixing work-flow)

You have adopted TDD as the development process for your project. Describe the work-flow you would use to fix a bug.

Possible work-flow:

5. fix the bug
   - even with dirty workarounds, to ensure you've cornered it
6. run all tests
   - all test passes → bingo!
   - new test fails → try again (= go back to 5)
   - old tests fail → regression, try again
7. refactor as needed
   - from workaround, to proper fix
8. release fix (including the new test!)

Outline

The money example

There are a several “great classics” among case studies to learn TDD and in particular its “rhythm.” Some of the most famous are:

- The money example — included in Kent Beck’s milestone book on TDD
- The bowling game — `http://www.objectmentor.com/resources/articles/xpepisode.htm`

In the remainder we are going to discuss (some parts of) the money example. Disclaimers:

- the rhythm might seem slow at first, you will be tempted to use obvious implementation more often than in the example (which uses fake it very often)
- that’s fine, you will find your own rhythm; in the meantime starting slow will help understand the philosophy
- we will take shortcuts, check out the full example in the book

Goal: multi-currency money

Instrument	Shares	Price	Total
IBM	1000	25	25 000
GE	400	100	40 000
		Total	65 000

Instrument	Shares	Price	Total
IBM	1000	25 USD	25 000 USD
Novartis	400	150 CHF	60 000 CHF
		Total	65 000 USD

From	To	Rate
CHF	USD	1.5

Notation — To-do list

When applying TDD you will often stumble upon items you want to work on (e.g., design improvements) which you have to postpone to the appropriate phase (e.g., refactoring). To keep track of them we will use to-do lists like this one:

ToDo

- oh yes, we should really do this
- **but we are currently working on this**
- ~~this is done~~
- ~~this too~~

Initial to-do list for the money example:

ToDo

- 5 USD + 10 CHF = 10 USD if rate is 2:1
- **5 USD * 2 = 10 USD**

Multiplication

Write a failing unit test:

```
@Test
public void dollarMultiplication() {
    Dollar five= new Dollar(5);
    five.times(2);
    assertEquals(10, five.amount);
}
```

Multiplication

Write a failing unit test:

```
@Test
public void dollarMultiplication() {
    Dollar five= new Dollar(5);
    five.times(2);
    assertEquals(10, five.amount);
}
```

Which doesn't compile!

- no class `Dollar`
- no constructor
- no method `times(int)`
- no field `amount`

Multiplication

Write a failing unit test:

```
@Test
public void dollarMultiplication() {
    Dollar five= new Dollar(5);
    five.times(2);
    assertEquals(10, five.amount);
}
```

Which doesn't compile!

- no class `Dollar`
- no constructor
- no method `times(int)`
- no field `amount`

That's fine! We progressed: we now have a more immediate goal to achieve (make the test compile).

Multiplication (cont.)

Let's address one compilation error at a time...

Multiplication (cont.)

Let's address one compilation error at a time...

1. **public class** Dollar { };
2. **public** Dollar(**int** amount) { /*empty */};
3. **public void** times(**int** multiplier) { /*empty */};
4. **public int** amount;

Multiplication (cont.)

Let's address one compilation error at a time...

1. **public class** Dollar { };
2. **public** Dollar(**int** amount) { */*empty */*};
3. **public void** times(**int** multiplier) { */*empty */*};
4. **public int** amount;

YAY! Now the test compiles... and fails with a red bar.

Progress: we now have a measure of how far we are from success—1 test is failing, we are just 1 test away from success.

Multiplication (cont.)

Let's make the bar green (you won't like this)

public int amount = 10; *//fake it*

The test now passes!
But that obviously *not* the right solution so. . . refactor.

Multiplication (cont.)

We currently have duplication in our code, even if it's hidden:

- the test contains a $5 * 2$ multiplication
- the code contains $10 (= 5 * 2)$

we want to factor out the duplication.

```
public Dollar(int amount) {
    this.amount = amount;
}
public void times(int multiplier) {
    this.amount *= multiplier;
}
```

Multiplication (cont.)

We currently have duplication in our code, even if it's hidden:

- the test contains a $5 * 2$ multiplication
- the code contains $10 (= 5 * 2)$

we want to factor out the duplication.

```
public Dollar(int amount) {
    this.amount = amount;
}
public void times(int multiplier) {
    this.amount *= multiplier;
}
```

ToDo

- 5 USD + 10 CHF = 10 USD if rate is 2:1
- ~~5 USD * 2 = 10 USD~~
- make amount private
- **avoid Dollar side effects**
- allow to have cents

Functional objects

```
@Test
public void dollarMultiplication() {
    Dollar five= new Dollar(5);
    five.times(2);
    assertEquals(10, five.amount);
    five.times(3);
    assertEquals(15, five.amount); // mmmmmhhhh...
}
```

Functional objects

```
@Test
public void dollarMultiplication() {
    Dollar five= new Dollar(5);
    five.times(2);
    assertEquals(10, five.amount);
    five.times(3);
    assertEquals(15, five.amount); // mmmmmhhhh...
}
```

⇓

```
@Test
public void dollarMultiplication() {
    Dollar five= new Dollar(5);
    Dollar product = five.times(2);
    assertEquals(10, product.amount);
    product = five.times(3);
    assertEquals(15, product.amount); // better design!
}
```

Red bar again!

Functional objects (cont.)

Let's make it compile:

```
Dollar times(int multiplier) {
    amount *= multiplier;
    return null; // fake it
}
```

test now compiles but doesn't pass

Functional objects (cont.)

Let's make it pass:

```
Dollar times(int multiplier) {
    return new Dollar(amount * multiplier);
}
```

Functional objects (cont.)

Let's make it pass:

```
Dollar times(int multiplier) {
    return new Dollar(amount * multiplier);
}
```

Green bar: YAY!

ToDo

- 5 USD + 10 CHF = 10 USD if rate is 2:1
- ~~5 USD * 2 = 10 USD~~
- make amount private
- ~~avoid Dollar side effects~~
- allow to have cents
- **equality**

Equality

```
@Test
public void dollarsEquality() {
    assertEquals(new Dollar(5), new Dollar(5));

    // same as, but clearer than
    // assertTrue(new Dollar(5).equals(new Dollar(5)));
}
```

- the test compiles (why?)
- but fails (why?)

Equality (cont.)

```
public boolean equals(Object object) {
    return true;
}
```

Test passes!

Equality (cont.)

```
public boolean equals(Object object) {
    return true;
}
```

Test passes!

But we want a proper implementation, so let's prove we have a bug in the current implementation. . . with a test!

```
@Test
public void dollarsInequality() {
    assertNotEquals(new Dollar(5), new Dollar(6));
}
```

(indeed we have a bug)

Equality (cont.)

We can now triangulate to a more general (and correct) solution:

```
public boolean equals(Object object) {
    Dollar dollar = (Dollar) object;
    return this.amount == dollar.amount;
}
```

Green bar!!

ToDo

- 5 USD + 10 CHF = 10 USD if rate is 2:1
- make `amount` private
- allow to have cents
- ~~equality~~
- equality against `null`
- equality against `Object`
- **5 CHF * 2 = 10 CHF**

CHF

```
@Test
public void francMultiplication() {
    Franc five = new Franc(5);
    assertEquals(new Franc(10), five.times(2));
    assertEquals(new Franc(15), five.times(3));
}
```

What's the shortest step which will bring us to green bar?

CHF (cont.)

Copy/paste/adapt from Dollar!

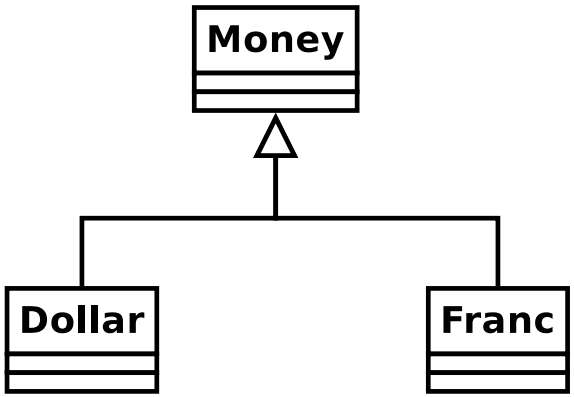
```
class Franc {
    private int amount;
    public Franc(int amount) { this.amount = amount; }
    public Franc times(int multiplier) {
        return new Franc(amount * multiplier);
    }
    public boolean equals(Object object) {
        Franc franc = (Franc) object;
        return this.amount = franc.amount;
    }
}
```

ToDo

- 5 USD + 10 CHF = 10 USD if rate is 2:1
- **Dollar/Franc duplication**
- common `equals`
- common `times`

CHF (cont.)

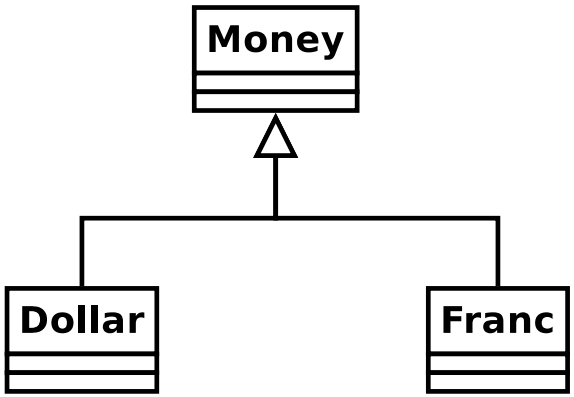
Refactoring to eliminate duplication:




```
class Money { protected int amount; }
class Dollar extends Money { /* ... */ }
class Franc extends Money { /* ... */ }
```

CHF (cont.)

Refactoring to eliminate duplication:




```
class Money { protected int amount; }
class Dollar extends Money { /* ... */ }
class Franc extends Money { /* ... */ }
```

```
public boolean equals(Object object) { // how about equals?
    Money dollar = (Dollar) object;    // ???
    return this.amount = dollar.amount;
}
```

ToDo

- 5 USD + 10 CHF = 10 USD if rate is 2:1
- Dollar/Franc duplication
- **common equals**
- common times

Equality redux

Bug!

```
@Test
public void equalityFrancDollar() { // this passes
    assertEquals(new Dollar(5), new Dollar(5));
    assertEquals(new Franc(7), new Franc(7));
}

@Test
public void inequalityFrancDollar() { // exception!
    assertNotEquals(new Dollar(5), new Franc(6));
    assertNotEquals(new Franc(7), new Dollar(8));
}
```

Equality redux (cont.)

```
// pull-up in class Money
public boolean equals(Object object) {
    Money money = (Money) object;
    return amount == money.amount
      && getClass().equals(money.getClass());
}
```

ToDo

- 5 USD + 10 CHF = 10 USD if rate is 2:1
- Dollar/Franc duplication
- ~~common equals~~
- common times

Etc.

. . . . . .

Outline

TDD — the bigger picture

It is tempting to use only *unit* tests to implement TDD, but:

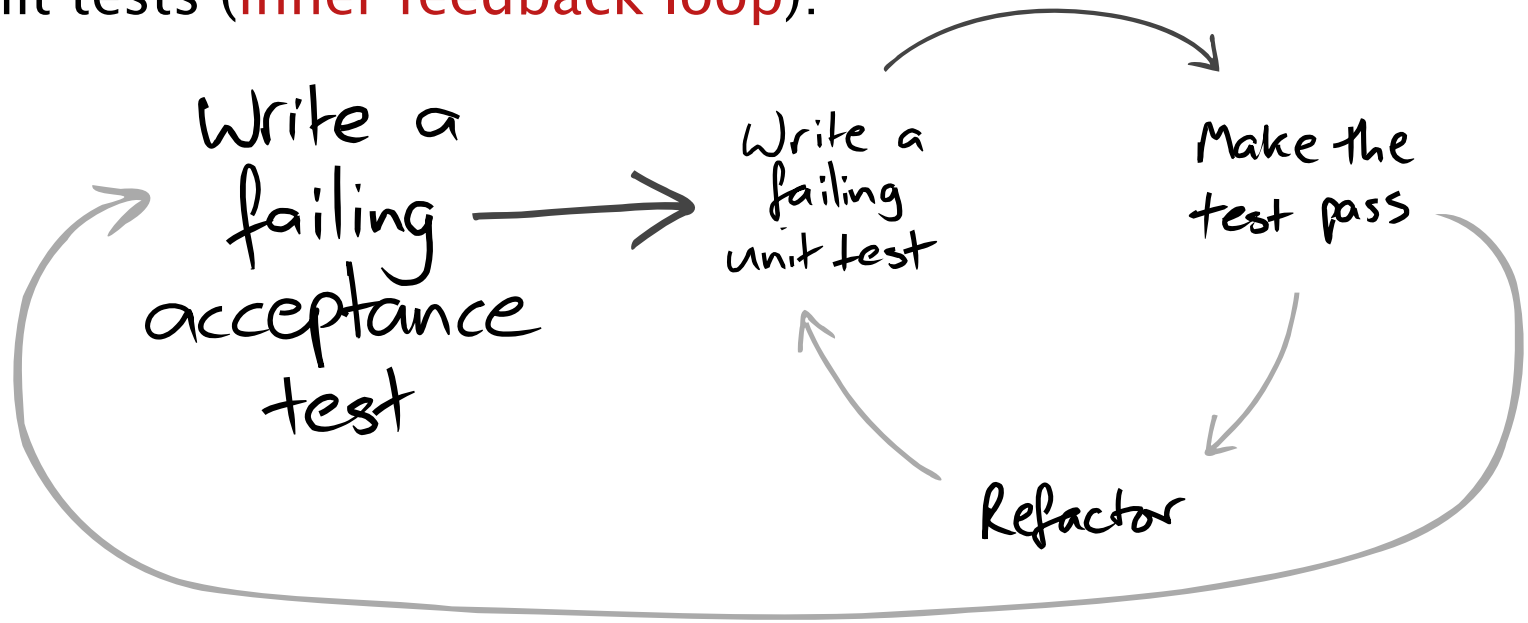
- you might end up having a lot of unused well-tested units
- you don't know where to start, nor when to stop

TDD — the bigger picture

It is tempting to use only *unit* tests to implement TDD, but:

- you might end up having a lot of unused well-tested units
- you don't know where to start, nor when to stop

That's why TDD leverages both acceptance (outer feedback loop) and unit tests (inner feedback loop):



GOOS, Figure 1.2

End-to-end testing

Acceptance tests should exercise the system end-to-end

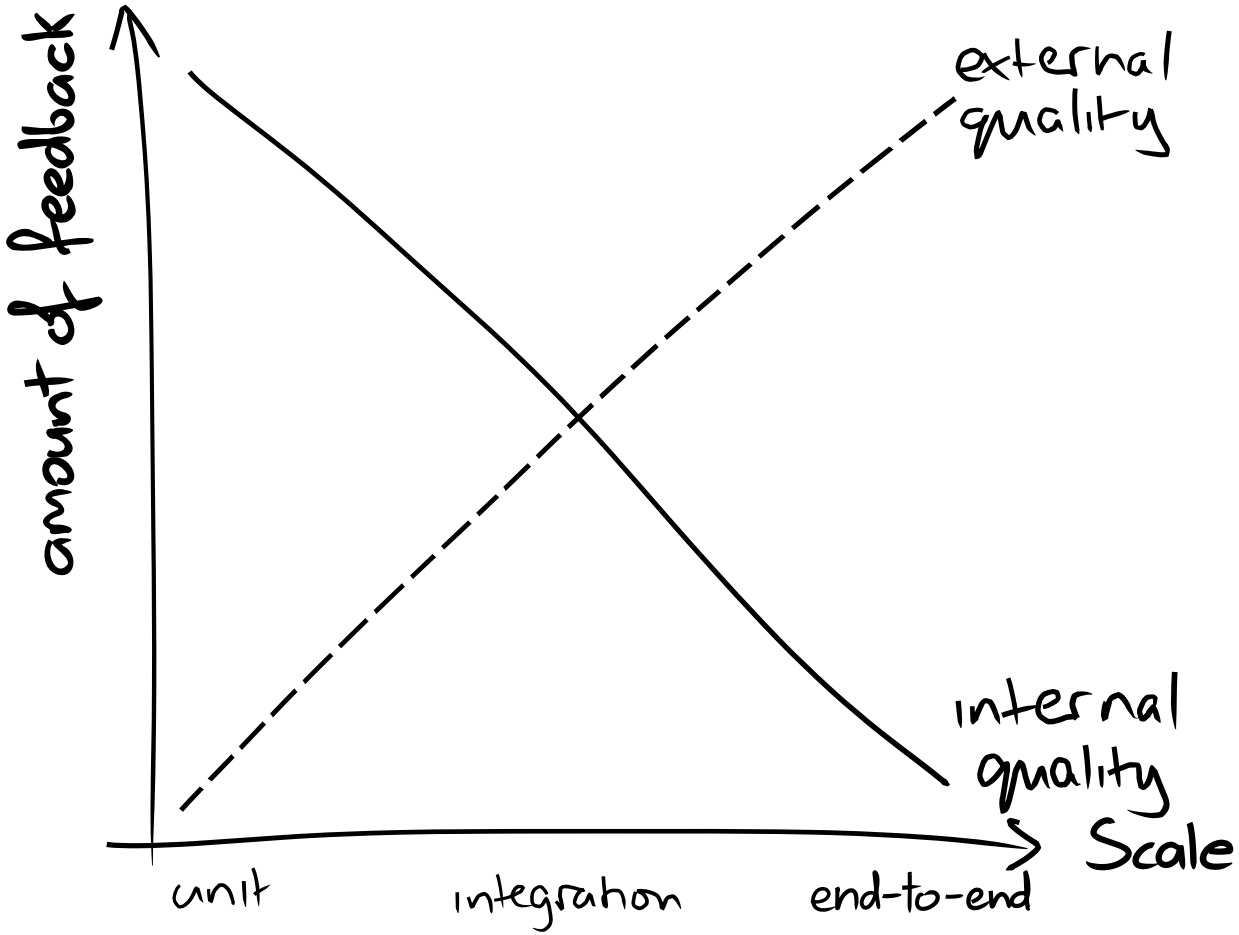
- black-box testing at system level
  - no instantiation/invocation of internal objects
  - use the system via its interfaces (user interface, external API, parsing its output and producing its inputs, etc.)
- test both the system and its processes
  - build
  - deployment in a realistic environment
    - don't trust the results of acceptance tests run in development environments
  - any other qualification mechanism
    - e.g., static analyses, stress testing, benchmark, etc.

Testing, quality, feedback

External quality: how well the system meets the needs of its users

Internal quality: how well the system meets the needs of its developers

- e.g., good design: low coupling & high cohesion
- it is often harder to push for internal than external quality, but we need to do so to cope with changes



GOOS, Figure 1.3

First feature paradox

Writing the first acceptance test at the beginning of a project is problematic:

- we want to test end-to-end the system and its processes
- but we don't have yet the tooling to make the test fail

First feature paradox

Writing the first acceptance test at the beginning of a project is problematic:

- we want to test end-to-end the system and its processes
- but we don't have yet the tooling to make the test fail

To get out of the paradox we compromise a bit, implementing a walking skeleton to kick start TDD.

Definition (walking skeleton)

An implementation of the smallest possible part of real functionality that we can automatically build, deploy, and test end-to-end.

To implement the walking skeleton we need to automate a lot of processes. That will force us to understand them better.

Example

The walking skeleton of a DBMS-backed web application will just show a static "Hello, World" web page.

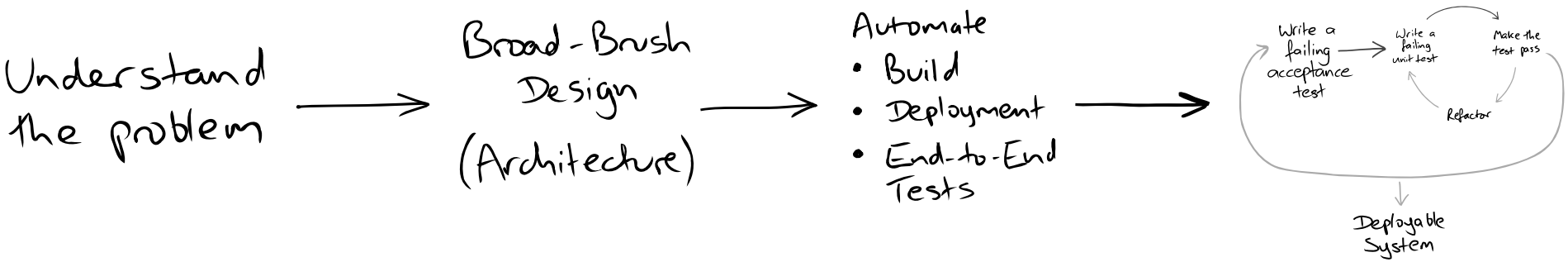
Walking skeleton

(Some of the) tasks to be completed as part of a walking skeleton:

- create a VCS repository, check in the code
  - requirements: choose Version Control System, choose hosting
- automate the build process
  - requirement: choose build tool (e.g., Make, Maven, etc.)
  - note: "just click a button in Eclipse" $\neq$ automation
- automate deployment in a realistic environment
  - requirement: choose packaging/deployment mechanisms
- automate test execution
  - requirement: choose test framework
  - again: "just click a button in Eclipse" $\neq$ automation
- ...
- iteration 0: implement, deploy, test first feature

Yes, it's a lot of work!

Kick-starting TDD



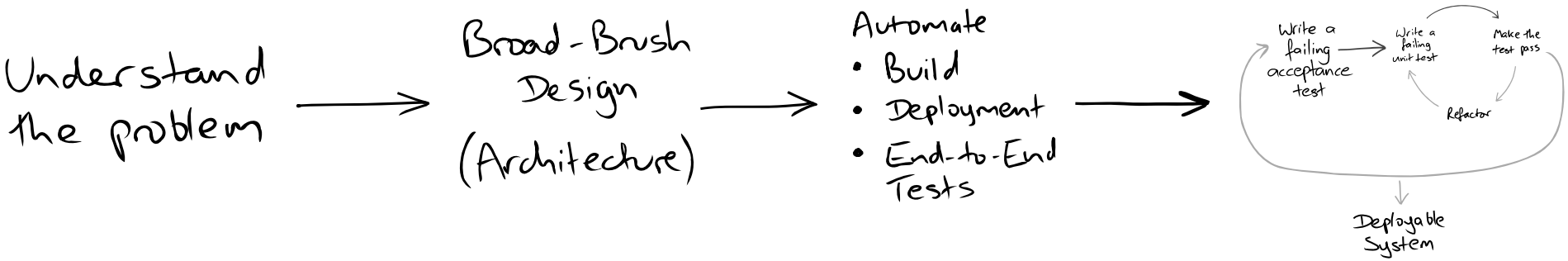
GOOS, Figure 4.2

Note: “Broad-Brush Design” ≠ “Big Design Up Front (BDUF)”

TDD as a whole

Periodically reassess both your understanding of the problem and the toolchain

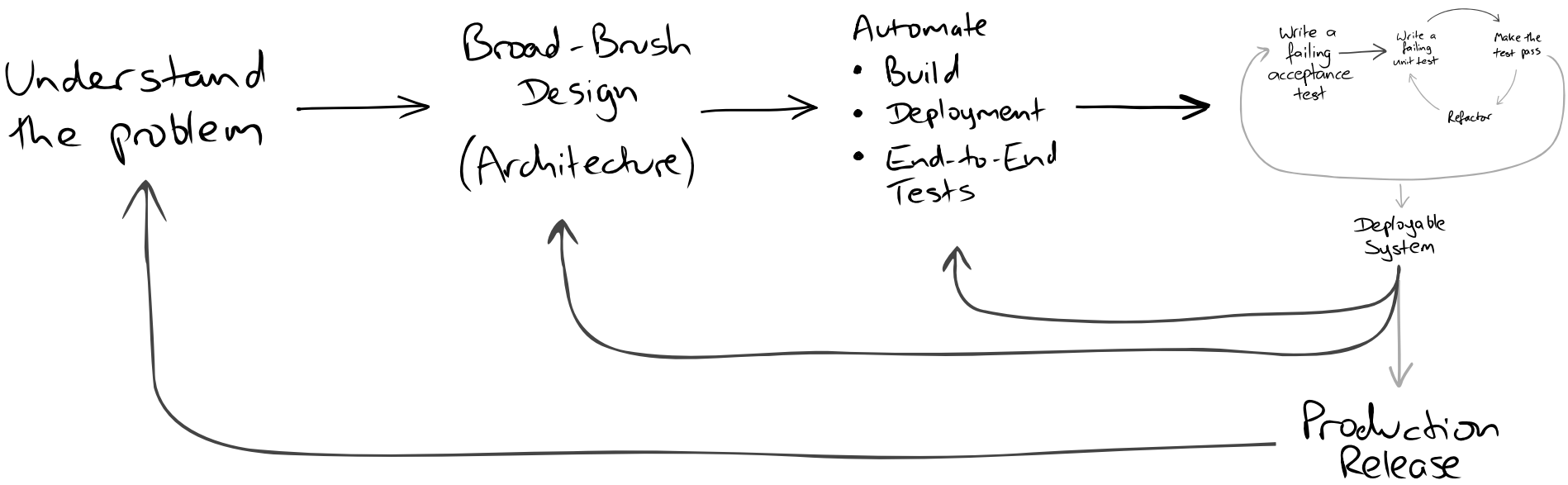
GOOS, Figure 4.2



TDD as a whole

Periodically reassess both your understanding of the problem and the toolchain

GOOS, Figure 4.3



Test suites organization

unit and integration test suites

- should always pass
- should run fast

acceptance test suite

- catch regressions
- should always pass
- might take longer to run

new acceptance test suite

- corresponds to work in progress
- will keep on failing during inner loop iterations

Bibliography


Steve Freeman and Nat Pryce
Growing Object-Oriented Software, Guided by Tests[2]
Addison-Wesley, 2009.


Kent Beck
Test Driven Development: By Example
Addison-Wesley, 2002.


Martin Fowler
Refactoring: Improving the Design of Existing Code
Addison-Wesley Professional, 1999.


Kent Beck
Simple smalltalk testing: With patterns
The Smalltalk Report 4.2 (1994): 16-18.
available at `http://www.xprogramming.com/testfram.htm`

2. referred to as the "GOOS" book in this lecture slides