Méthodes de Test
Cours 2 — Kick-starting and Maintaining TDD

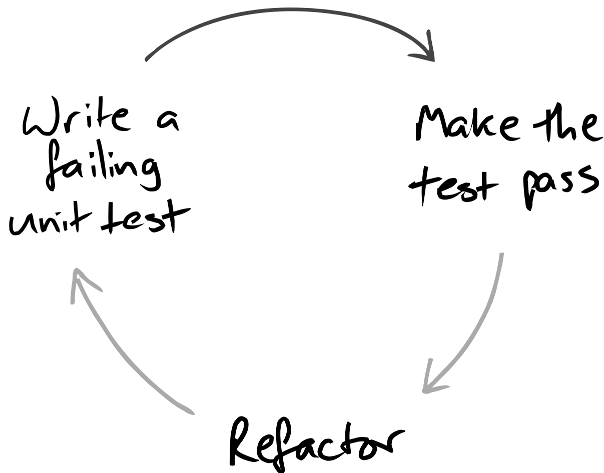
Stefano Zacchiroli
zack@pps.univ-paris-diderot.fr

Laboratoire PPS, Université Paris Diderot

2013-2014

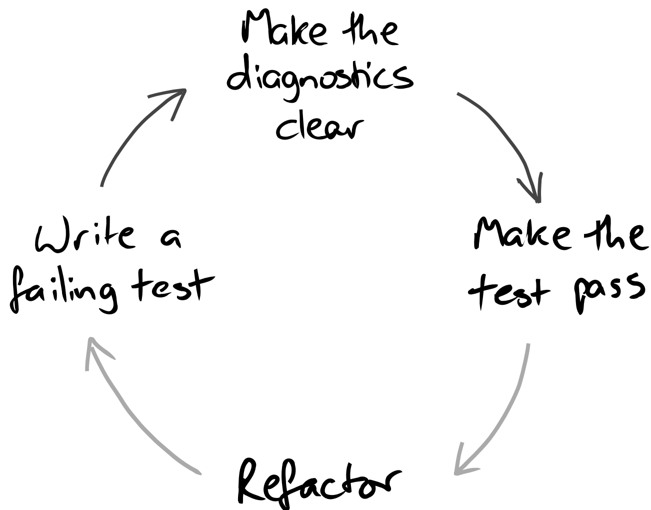
URL http://upsilon.cc/zack/teaching/1314/methtest/
Copyright © 2013 Stefano Zacchiroli
License Creative Commons Attribution-ShareAlike 3.0 Unported License
http://creativecommons.org/licenses/by-sa/3.0/

Reminder



GOOS, Figure 1.1

Reminder (cont.)



GOOS, Figure 5.2

Outline

1. TDD unit testing — case study
2. Kick-starting TDD
3. Maintaining TDD
   - Test smells
   - Test readability
   - Constructing complex test data
   - Test diagnostics

Outline

1. TDD unit testing — case study
2. Kick-starting TDD
3. Maintaining TDD
   - Test smells
   - Test readability
   - Constructing complex test data
   - Test diagnostics

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	25000
GE	400	100	40000
		Total	65000

Instrument	Shares	Price	Total
IBM	1000	25 USD	25000 USD
GE	400	150 CHF	40000 CHF
		Total	65000 USD

From	To	Rate
CHF	USD	1.5

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

```java
@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

```java
@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!

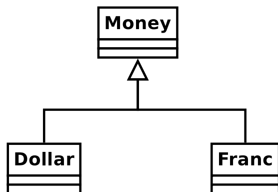
```
class Franc {
    private int amount;
    public Franc(int amount) { this.amount = amount; }
    public Franc times(int multiplier) {
        return new Franc(amount * multiplier);
    }
    public boolean equals(Object object) {
        Franc franc = (Franc) object;
        return this.amount =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

ToDo

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

CHF (cont.)

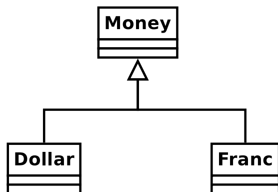
Refactoring to eliminate duplication:




```
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

1 TDD unit testing — case study

2 Kick-starting TDD

3 Maintaining TDD
- Test smells
- Test readability
- Constructing complex test data
- Test diagnostics

TDD — the bigger picture

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

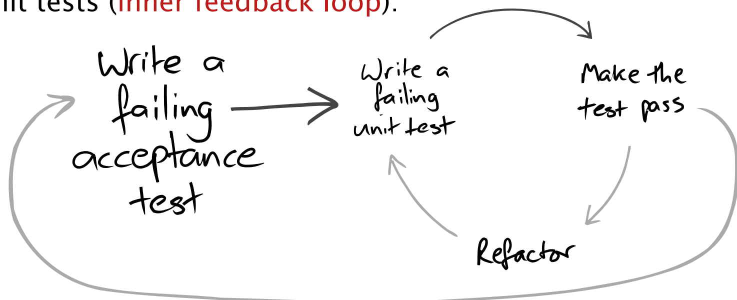
- you might end up having a lot of unused well-tested units
- you don't know where to start, nor when to stop

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

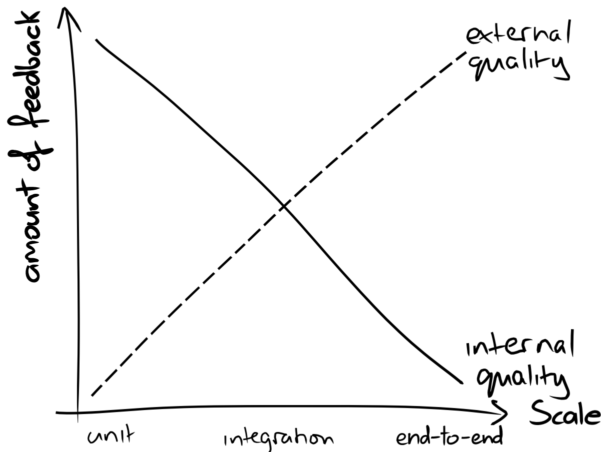
- black-box testing at system level
  - no instantiation/invocation of internal objects
  - use the system via its interfaces (user interface, external API, parsing its output and producing its inputs, etc.)
- test both the system and its processes
  - build
  - deployment in a realistic environment
    - 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 quality 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.

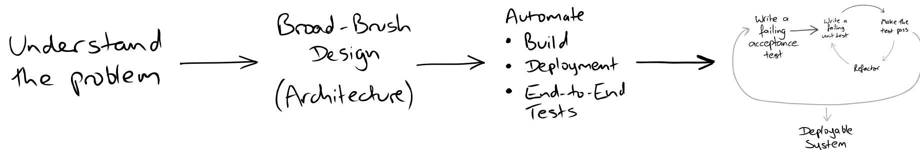
Walking skeleton

(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. Ant, Maven)
  - note: “just click a button in Eclipse” ≠ 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” ≠ automation
- ...
- iteration 0: implement, deploy, test first feature

Yes, it’s a lot of work!

Kick-starting TDD



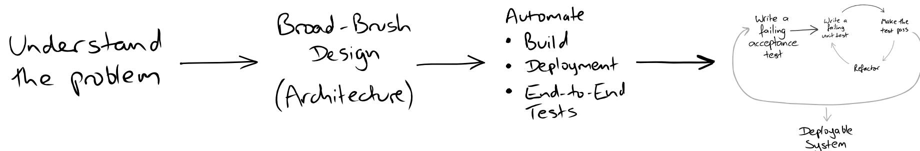
GOOS, Figure 4.2

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

TDD as a whole

Periodically reassess both your understanding of the problem and the toolchain

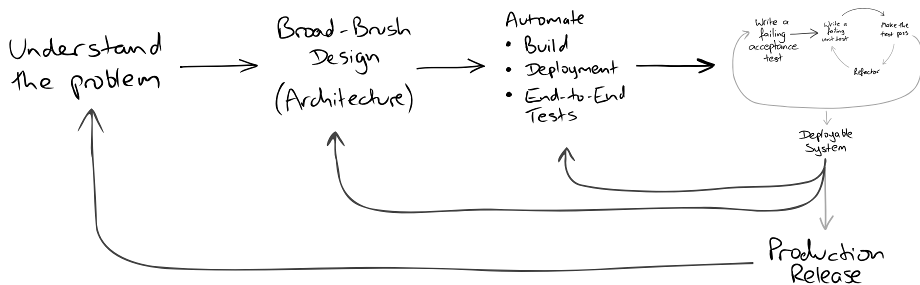
GOOS, Figure 4.2



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

Outline

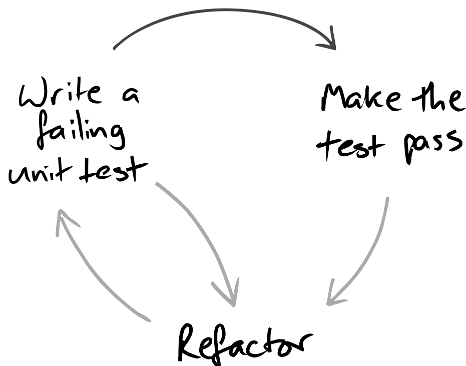
1 TDD unit testing — case study

2 Kick-starting TDD

3 Maintaining TDD
- Test smells
- Test readability
- Constructing complex test data
- Test diagnostics

Maintaining TDD — listen to the tests

- you encounter a feature which is difficult to test
- don't work around that, investigate *why* it is the case
- often, that leads to missed refactoring from previous iterations of the inner loop
  - potential future maintenance problem
- take the chance and do the refactoring



GOOS, Figure 5.3

More generally: notice test smells and let them guide your design (if the feature is yet to be implemented) or your refactoring (otherwise)

note: in the following, most of the code examples are from the GOOS book

Outline

1. TDD unit testing — case study
2. Kick-starting TDD
3. Maintaining TDD
   - Test smells
   - Test readability
   - Constructing complex test data
   - Test diagnostics

Tell, don't ask

"Train wreck" code: series of getters chained together like the carriages in a train:

```
((EditSaveCustomizer) master.getModelisable()
  .getDockablePanel()
    .getCustomizer())
      .getSaveItem().setEnabled(Boolean.FALSE.booleanValue());
```

what it actually means:

```
master.allowSavingOfCustomisations();
```

"Tell, don't ask" principle

Don't ask (recursively) access to objects internals that allow you to perform a specific operation. Rather, tell the (outermost) object to do something on your behalf; let it do the same, recursively, as needed.

This makes your tests (and code in general) more resistant to changes in object organization.

Tell, don't ask (cont.)

When you really have to ask, do so explicitly via well-defined query methods (i.e. queries that have clear names and well-defined semantics):

```
public void reserveSeats(ReservationRequest request) {
  for (Carriage carriage : carriages) {
    if (carriage.getSeats().getPercentReserved() < percentReservedBa
      request.reserveSeatsIn(carriage);
      return;
  } }
  request.cannotFindSeats(); }
```

```
public void reserveSeats(ReservationRequest request) {
  for (Carriage carriage : carriages) {
    if (carriage.hasSeatsAvailableWithin(percentReservedBarrier)) {
      request.reserveSeatsIn(carriage);
      return;
  } }
  request.cannotFindSeats(); }
```

Object relationships

When writing tests and designing OO systems, the following classification of object relationships come in handy [GOOS]:

Dependencies: services that the object requires from its peers so it can perform its responsibilities. Object cannot function without these services. It should not be possible to create the object without them.

Notifications: peers that need to be kept up to date with the object's activity. Notifications are fire-and-forget (i.e. can be lost) and the notifier doesn't care about which peers are listening.

Adjustments: peers that adjust the object's behaviour to the wider needs of the system. E.g. policy objects (cfr. the strategy design pattern).

Singletons are dependencies

```
public boolean acceptRequest(Request request) {
    final Date now = new Date();
    if (dateOfFirstRequest == null) {
        dateOfFirstRequest = now;
    } else if (firstDateIsDifferentFrom(now)) {
        return false;
    }
    // process the request
    return true;
}

@Test public void rejectsRequestsNotWithinTheSameDay() {
    receiver.acceptRequest(FIRST_REQUEST);
    // the next day                    ← how do we implement this?
    assertFalse("too late now",
                receiver.acceptRequest(SECOND_REQUEST));
}
```

Singletons are dependencies (cont.)

Problem: singletons hide dependencies — not surprising at all, as singletons are global variables.
Solution: make the dependency explicit.

Singletons are dependencies (cont.)

Problem: singletons hide dependencies — not surprising at all, as singletons are global variables.
Solution: make the dependency explicit.

```
public boolean acceptRequest(Request request) {
    final Date now = clock.now();
    if (dateOfFirstRequest == null) {
        dateOfFirstRequest = now;
    } else if (firstDateIsDifferentFrom(now)) {
        return false;
    }
    // process the request
    return true;
}
@Test public void rejectsRequestsNotWithinTheSameDay() {
    Receiver receiver = new Receiver(stubClock);
    stubClock.setNextDate(TODAY);
    receiver.acceptRequest(FIRST_REQUEST);
    stubClock.setNextDate(TOMORROW);
    assertFalse("too late now", receiver.acceptRequest(SECOND_REQUES
}
```

Singletons are dependencies (cont.)

The general problem highlighted by singletons is that hiding dependencies (e.g. via global variables) does not make them disappear. They are still there, ready to strike back—e.g. when, to test the code, you need to change/fake them.

Dependencies — golden rule

An object should only deal with values that are either local (i.e. created and managed within its scope) or passed in explicitly (e.g. at construction or method invocation-time).

This is the key rule to achieve context independent objects.

Bloated constructor

By adding one dependency at a time we might end up with:

```
public class MessageProcessor {
    public MessageProcessor(MessageUnpacker unpacker,
                            AuditTrail auditor,
                            CounterPartyFinder counterpartyFinder,
                            LocationFinder locationFinder,
                            DomesticNotifier domesticNotifier,
                            ImportedNotifier importedNotifier)
    { /* set the fields here */ }

    public void onMessage(Message rawMessage) {
        UnpackedMessage unpacked =
            unpacker.unpack(rawMessage, counterpartyFinder);
        auditor.recordReceiptOf(unpacked);
        // some other activity here
        if (locationFinder.isDomestic(unpacked)) {
            domesticNotifier.notify(unpacked.asDomesticMessage());
        } else {
            importedNotifier.notify(unpacked.asImportedMessage())
        }
}
```

Bloated constructor (cont.)

Solution #1: look for objects that are always used together ⇒ they probably denote a single concept that should be factorized out.

- `counterpartyFinder` always used with unpacker? then

```
UnpackedMessage unpacked = unpacker.unpack(rawMessage);
```

- `locationFinder` + 2 notifiers always used together? then

```
public MessageProcessor(MessageUnpacker unpacker,
                        AuditTrail auditor,
                        MessageDispatcher dispatcher)
                        // much better!
{ /* set the fields here */ }

public void onMessage(Message rawMessage) {
    UnpackedMessage unpacked =
        unpacker.unpack(rawMessage);
    auditor.recordReceiptOf(unpacked);
    // some other activity here
    dispatcher.dispatch(unpacked);
```

Confused object

```
public class Handset {
    public Handset(Network network, Camera camera, Display display,
                   DataNetwork dataNetwork, AddressBook addressBook,
                   Storage storage, Tuner tuner, ...)
    { /* set the fields here */ }
    public void placeCallTo(DirectoryNumber number) {
        network.openVoiceCallTo(number);
    }
    public void takePicture() {
        Frame frame = storage.allocateNewFrame();
        camera.takePictureInto(frame);
        display.showPicture(frame);
    }
    public void showWebPage(URL url) {
        display.renderHtml(dataNetwork.retrievePage(url));
    }
    public void showAddress(SearchTerm searchTerm) {
        display.showAddress(addressBook.findAddress(searchTerm));
    }
    public void playRadio(Frequency frequency) {
        tuner.tuneTo(frequency);
        tuner.play();
    }
    // ...
```

Confused object (cont.)

Another instance of bloated constructor.

Different problem: the object has too many responsibilities.

Solution #2: split distinct objects, each one with a single responsibility, possibly assemblying objects back using containers where needed. The resulting objects can now be tested separately and more easily — in particular test setup is much easier.

Alleged dependencies

```
public class RacingCar {
    private final Track track;
    private Tyres tyres;
    private Suspension suspension;
    private Wing frontWing;
    private Wing backWing;
    private double fuelLoad;
    private CarListener listener;
    private DrivingStrategy driver;
    public RacingCar(Track track, DrivingStrategy driver, Tyres tyres,
                     Suspension suspension, Wing frontWing, Wing backWing,
                     double fuelLoad, CarListener listener)
    {
        this.track = track;
        this.driver = driver;
        this.tyres = tyres;
        this.suspension = suspension;
        this.frontWing = frontWing;
        this.backWing = backWing;
        this.fuelLoad = fuelLoad;
        this.listener = listener;
    }
}
```

Alleged dependencies (cont.)

Another instance of bloated constructor.

Different problem: the object declares unneeded dependencies, i.e. some of them are not real dependencies. Objects can in fact be instantiated without providing them.

Solution #3: make arguments optional and rely on sane default values and getters/setters.

```
private DrivingStrategy driver = DriverTypes.borderlineAggressiveDriving();
private Tyres tyres = TyreTypes.mediumSlicks();
private Suspension suspension = SuspensionTypes.mediumStiffness();
private Wing frontWing = WingTypes.mediumDownforce();
private Wing backWing = WingTypes.mediumDownforce();
private double fuelLoad = 0.5;
private CarListener listener = CarListener.NONE;

public RacingCar(Track track) { this.track = track; }

public void setSuspension(Suspension suspension) { /* ... */
public void setTyres(Tyres tyres) { /* ... */
public void setEngine(Engine engine) { /* ... */
public void setListener(CarListener listener) { /* ... */
```

Outline

1. TDD unit testing — case study
2. Kick-starting TDD
3. Maintaining TDD
   - Test smells
   - Test readability
   - Constructing complex test data
   - Test diagnostics

Test readability

In general, we strive for readable (production) code.

- developers spend more time reading code than writing it
- readable code ⇒ code that is easy to change/debug/fix

For the same reasons, we want readable tests.

- tests break for valid reasons (refactoring, regressions, etc.)
- when that happens we want to be able to fix them easily
- otherwise TDD becomes a burden; we will be tempted to:
  - `@Ignore` specific tests
  - drop TDD as a whole

Test readability (cont.)

On the other hand: test readability has different goals than (production) code readability

Production code readability

Production code should be abstract on the values it manipulates, but concrete about *how* it gets the job done

Test readability

Tests describe *what* the production code does; test should read as declarative descriptions of what is being tested

Test names describe feature

```
public class TargetObjectTest // very bad convention
                              // no information conveyed
    @Test public void test1() {
    @Test public void test2() {
    @Test public void test3() {


public class TargetObjectTest { // bad convention too
            // you should test features, not methods
    @Test public void isReady() { [...]
    @Test public void choose() { [...]
    @Test public void choose1() { [...]

public class TargetObject {
    public void isReady() { [...]
    public void choose(Picker picker) { [...]
```

TestDox

A good naming convention for test is known as TestDox.

- each test name reads like a sentence
- each test has the target class as implicit subject

E.g.:

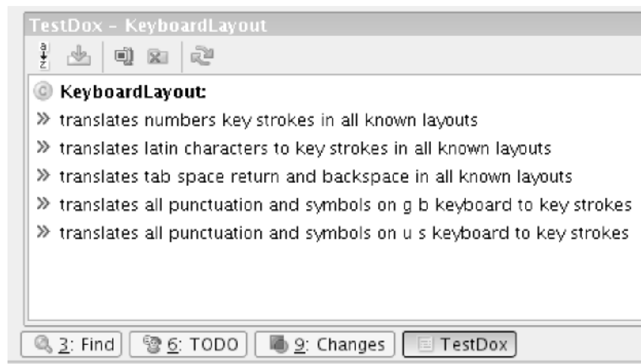
- a `List` holds items in the order they were added
- a `List` can hold multiple references to the same item
- a `List` throws an exception when removing a missing item

```
public class ListTests {
    @Test public void
    holdsItemsInTheOrderTheyWereAdded() { //

    @Test public void
    canHoldMultipleReferencesToTheSameItem() { //

    @Test public void
    throwsAnExceptionWhenRemovingAnItemItDoesntHold() { //
```

Note: we don't care about name length. Test names are discovered via reflection, we never have to type them.

TestDox — IDE support

Once a convention like TestDox is adopted, IDE support can be added to it, e.g.:



GOOS, Figure 21.1

Figure : TestDox IntelliJ plug-in

Canonical test structure

1. Setup: prepare the test context (environment, input, etc.)
2. Execute: call target code
3. Verify: check visible effects of target code
4. Teardown: (optional) cleanup leftover state that could break test isolation

Leaving teardown implicit, an equivalent mnemonic is Arrange, Act, Assert.

```
public class StringTemplateTest {
    @Test public void expandsMacrosSurroundedWithBraces() {
        StringTemplate template =
            new StringTemplate("{a}{b}"); // 1. setup
        HashMap<String,Object> macros = new HashMap<String,Object>()
        macros.put("a", "A");
        macros.put("b", "B");

        String expanded = template.expand(macros); // 2. execute
        assertEquals(expanded, "AB"); // 3. assert
    } // 4. (no) teardown
}
```

Canonical test structure (cont.)

Recommended test writing order:

1. start by writing the test name (helps to clarify the goal)
2. write target code invocation
3. write assertions
4. fill in the gaps, i.e. setup and teardown

Literals and variables

Test code tends to be more concrete than production code
⇒ test code tends to have more literal values.

```
Book book = catalog.lookup(666);
assertNull(book);
```

Problem: a literal value does not describe its role
(unless it's really really obvious, which is almost never the case).
Solution: use constants with self-describing names instead of literals

```
public static final int INVALID_BOOK_ID = 666;

Book book = catalog.lookup(INVALID_BOOK_ID);
assertNull(book);
```

Outline

1 TDD unit testing — case study

2 Kick-starting TDD

3 Maintaining TDD
- Test smells
- Test readability
- Constructing complex test data
- Test diagnostics

Complex test data

```
@Test public void
chargesCustomerForTotalCostOfAllOrderedItems() {
    Order order = new Order(
      new Customer("Sherlock Holmes",
        new Address("221b Baker Street",
                    "London",
                    new PostCode("NW1", "3RX"))));
    order.addLine(new OrderLine("Deerstalker Hat", 1));
    order.addLine(new OrderLine("Tweed Cape", 1));
    // ...
}
```

Effects:

- tests with complex setup logic ⇒ hard to read
- tests which will break often, not for good reasons
  - changes in constructor arguments, object structure, etc.

How can we be more resilient to this kind of changes?

Object mother pattern

```
Order order = ExampleOrders.newDeerstalkerAndCapeOrder();
```

Good for readability!

. . . but doesn't scale well to increase in variability:

```
Order order1 =
    ExampleOrders.newDeerstalkerAndCapeAndSwordstickOrder();
Order order2 = ExampleOrders.newDeerstalkerAndBootsOrder();
```

Test data builders

A better solution is to apply the builder pattern to test data.

We create a test data builder that has:

- a field for each constructor parameter, set to sane defaults
- chainable public methods for overwriting its default values
- a `build()` factory method, to be called last

Note: this is essentially a functional version of the builder pattern.

Test data builders — example

```
public class OrderBuilder {
    private Customer customer = new CustomerBuilder().build();
    private List<OrderLine> lines = new ArrayList<OrderLine>();
    private BigDecimal discountRate = BigDecimal.ZERO;
    public static OrderBuilder anOrder() {
        return new OrderBuilder(); }
    public OrderBuilder withCustomer(Customer customer) {
        this.customer = customer;
        return this; }
    public OrderBuilder withOrderLines(OrderLines lines) {
        this.lines = lines;
        return this; }
    public OrderBuilder withDiscount(BigDecimal discountRate) {
        this.discountRate = discountRate;
        return this; }
    public Order build() {
        Order order = new Order(customer);
        for (OrderLine line : lines) {
            order.addLine(line);
            order.setDiscountRate(discountRate); }
        return order;
} }
```

Test data builders — example (cont.)

```
Order order1 = new OrderBuilder().build();

Order order2 = newOrderBuilder().withDiscount(6.7).build();
```

- tests are more readable, due to reduced syntactic noise
- the default case is simple, the non-default cases stand-off
- tests are protected against changes in object structure
- errors are easier to spot, e.g.:

```
TestAddresses.newAddress("221b Baker Street", "London", "NW1 6XE");
```

vs

```
new AddressBuilder()
  .withStreet("221b Baker Street")
  .withStreet2("London")
  .withPostCode("NW1 6XE")
  .build();
```

Creating similar objects

We can use test data builders to factorize the creation of similar objects:

```
Order orderWithSmallDiscount = new OrderBuilder()
  .withLine("Deerstalker Hat", 1)
  .withLine("Tweed Cape", 1)
  .withDiscount(0.10)
  .build();
Order orderWithLargeDiscount = new OrderBuilder()
  .withLine("Deerstalker Hat", 1)
  .withLine("Tweed Cape", 1)
  .withDiscount(0.25)
  .build();
```

```
OrderBuilder hatAndCape = new OrderBuilder()
  .withLine("Deerstalker Hat", 1)
  .withLine("Tweed Cape", 1);
```

I.e.: builders as higher-order functions (take #1)

Creating similar objects — side-effects FAIL!

```
Order orderWithDiscount = hatAndCape.withDiscount(0.10).build();
Order orderWithGiftVoucher =
  hatAndCape.withGiftVoucher("abc").build();
```

Will you get a discount for `orderWithGiftVoucher`?

Creating similar objects — side-effects FAIL!

```
Order orderWithDiscount = hatAndCape.withDiscount(0.10).build();
Order orderWithGiftVoucher =
  hatAndCape.withGiftVoucher("abc").build();
```

Will you get a discount for `orderWithGiftVoucher`?

Yes :-(

Solution #1: copy constructor

```
Order orderWithDiscount = new OrderBuilder(hatAndCape)
  .withDiscount(0.10).build();
Order orderWithGiftVoucher = new OrderBuilder(hatAndCape)
  .withGiftVoucher("abc").build();
```

Solution #2: factory method that returns the builder

```
Order orderWithDiscount =
  hatAndCape.but().withDiscount(0.10).build();
Order orderWithGiftVoucher =
  hatAndCape.but().withGiftVoucher("abc").build();
```

I.e. builders as higher-order functions (take #2)

Combining builders

Nested `.build()` invocations are hard to read (syntactic noise, again). We can avoid them by passing builders around, instead of the objects built by them.

```
Order orderWithNoPostcode = new OrderBuilder()
   .withCustomer(
     new CustomerBuilder()
        .withAddress(new AddressBuilder().withNoPostcode().build())
        .build())
     .build();
```

```
Order order = new OrderBuilder()
   .withCustomer(
     new CustomerBuilder()
        .withAddress(new AddressBuilder().withNoPostcode()))
     .build();
```

I.e. builders as higher-order functions (take #3)

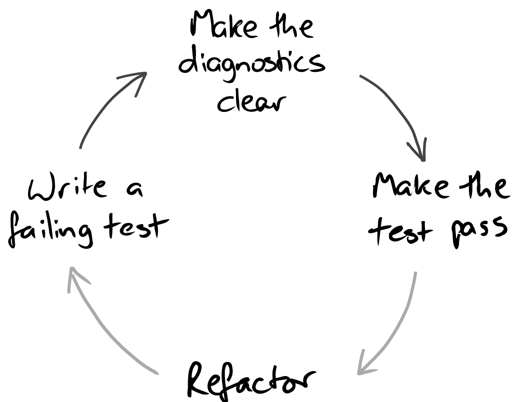
Outline

1. TDD unit testing — case study

2. Kick-starting TDD

3. Maintaining TDD
   - Test smells
   - Test readability
   - Constructing complex test data
   - Test diagnostics

Diagnostic — reminder



GOOS, Figure 5.2

Test readability was concerned with a static quality of tests as documentation.

Test diagnostics is concerned with a dynamic quality: how informative their runtime results are.

Explanatory assertions

```
Customer customer = order.getCustomer();
assertEquals("573242", customer.getAccountId());
assertEquals(16301, customer.getOutstandingBalance());

// result in test failures like:
ComparisonFailure: expected:<[16301]> but was:<[16103]>
```

```
assertEquals("account id", "573242", customer.getAccountId());
assertEquals("outstanding balance", 16301,
             customer.getOustandingBalance());

// result in:
ComparisonFailure:
outstanding balance expected:<[16301]> but was:<[16103]>
```

which one do you prefer?

Self-describing values

As an alternative, we can instrument values to be self-describing

```
Date startDate = new Date(1000);
Date endDate = new Date(2000);

// result in:
ComparisonFailure: expected: <Thu Jan 01 01:00:01 GMT 1970>
                    but was: <Thu Jan 01 01:00:02 GMT 1970>
```

it is correct, but doesn't tell us the meaning of these dates for the purposes of the test

⇓

```
Date startDate = namedDate(1000, "startDate");
Date endDate = namedDate(2000, "endDate");

Date namedDate(long timeValue, final String name) {
    return new Date(timeValue) {
      public String toString() { return name; } };
}

// result in:
ComparisonFailure: expected: <startDate> but was: <endDate>
```

Obviously canned value

Sometimes you cannot instrument values to become self-describing

- e.g.: primitive data types in Java

In those cases, try to use obviously canned values, i.e. values that when read will trigger reactions like “this cannot possibly be a correct value”, e.g.:

- negative values (where everyone would expect a positive one)
- `Integer.MAX_VALUE`
- ...