ASPROTUNITY
www.asprotunity.com

TDD For Investment Banking

v 1.1

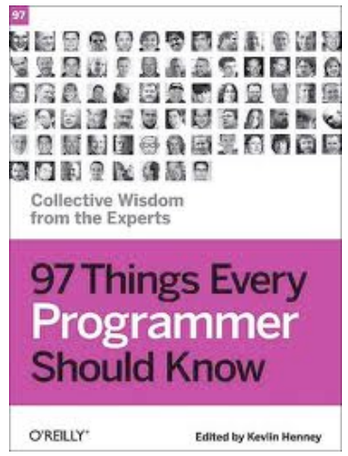
Giovanni Asproni
email: gasproni@asprotunity.com
twitter: @gasproni
linkedin: http://www.linkedin.com/in/gasproni

ASPROTUNITY
www.asprotunity.com

Giovanni Asproni

- I help software companies and teams to become more effective at producing and delivering high quality software. And I write code as well.
- Former Chair of the ACCU conference http://accu.org/index.php/conferences
- Former Chair of the London XPDay conference http://www.xpday.org

97 Things Every Programmer Should Know
Henney, 2010



Ground Rules

- Switch Off Mobile Phones
- Chatham House Rule: participants are free to use the information received, but neither the identity nor the affiliation of the speaker(s), nor that of any other participant, may be revealed

Introduction

- Who are you?
- What is your experience with (automated) testing?
  - Score yourself from 0 (no experience) to 5 (expert)
- What is your objective for the course?

Agenda

- Day 1
  - Introduction on testing
    - Cost of software
    - Testing vs quality and speed
    - Importance of test automation
    - TDD and refactoring
    - Testing time related behaviour
    - Some common test issues
    - Exercises
- Day 2
  - TDD of database code
  - TDD of asynchronous systems
  - TDD at the system level and relationship with requirements and design
  - Big exercise
  - Wrap up

What Is Testing?

- Testing is about finding answers to some questions
  - Does the software do what we want it to do?
  - Does the software not do what we don't want it to do?
  - Does the software do what our customers want?
  - What are the likelihood and consequences of failure?

From: "Perfect Software and Other Illusions about Testing", Jerry Weinberg

What Is Testing?

- Testing provides some information necessary to decide if the quality is good enough
- There are other parameters to keep into account
  - Cost
  - Time to market
  - Return on investment
  - Risk
  - Etc.

Cost Of Software

- $cost_{total} = cost_{develop} + cost_{maintain}$

- $cost_{maintain} = cost_{understand} + cost_{change} + cost_{test} + cost_{deploy}$

Cost Of Maintenance

- Usually $cost_{maintain}/cost_{total}$ ~ 60%
- It is important to write maintainable software

"Frequently Forgotten Fundamental Facts about Software Engineering" Robert Glass, IEEE Software, May/June 2001, http://www.eng.auburn.edu/~hendrix/comp6710/readings/Forgotten_Fundamentals_IEEE_Software_May_2001.pdf

Cost Of Software

- $cost_{maintain} = cost_{understand} + cost_{change} + cost_{test} + cost_{deploy}$

- The above costs are related with each other

- TDD aims at reducing $cost_{understand}$, $cost_{change}$, and $cost_{test}$

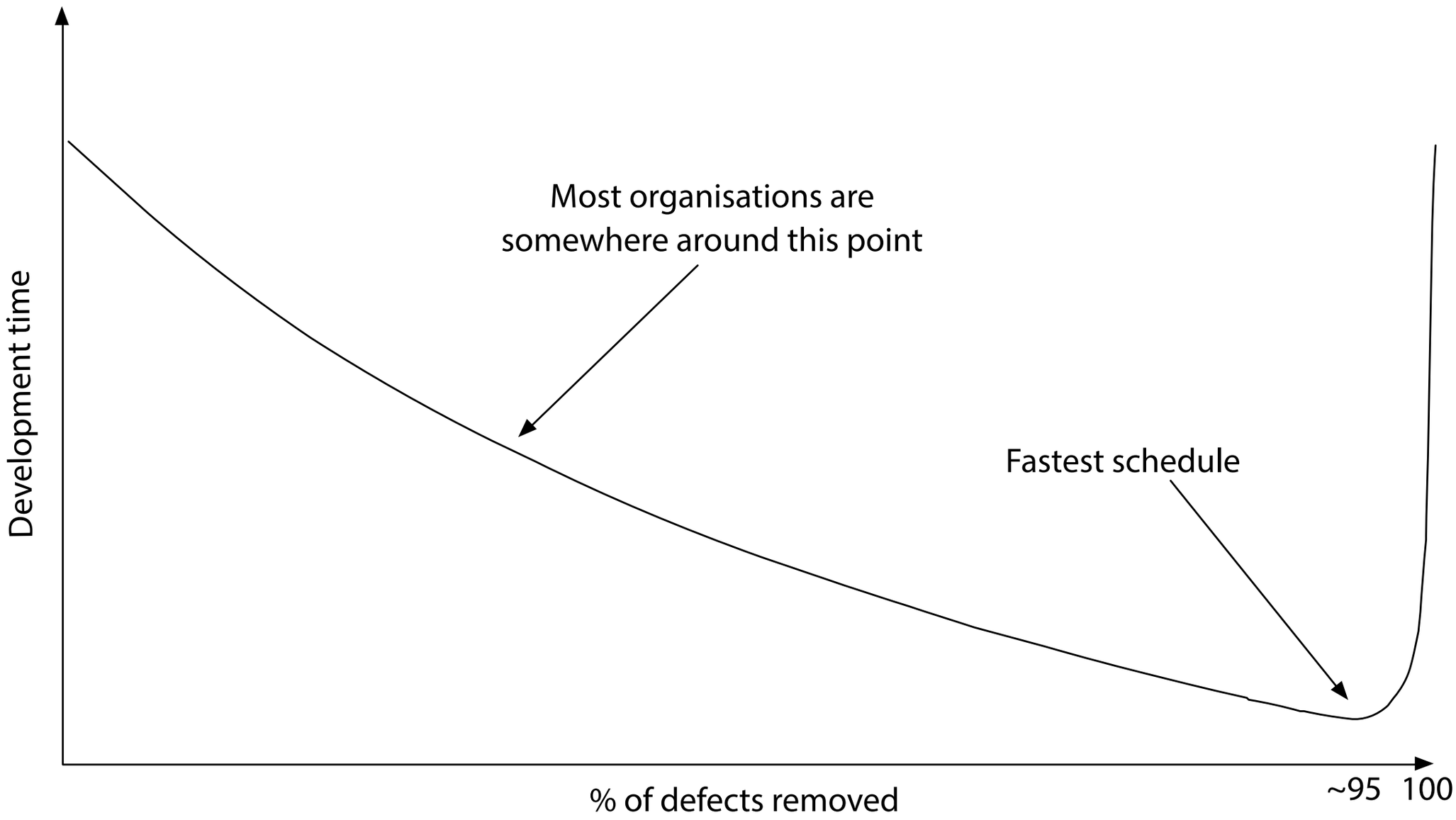
How Quality Affects Costs

As of 2011, the **average cost per function** point in the United States is about **$1,000 to build** software applications and another **$1,000 to maintain** and support them for five years: **$2,000 per function point in total**. For projects that **use effective combinations of defect prevention and defect removal** activities and achieve high quality levels, average development costs are only about **$700 per function point** and maintenance, and support costs drop to about **$500 per function point**: **$1,200 per function point in total**.

From “The Economics of Software Quality”,
Capers Jones and Olivier Bonsignour

ASPROTUNITY
www.asprotunity.com

How Quality Affects Delivery Time



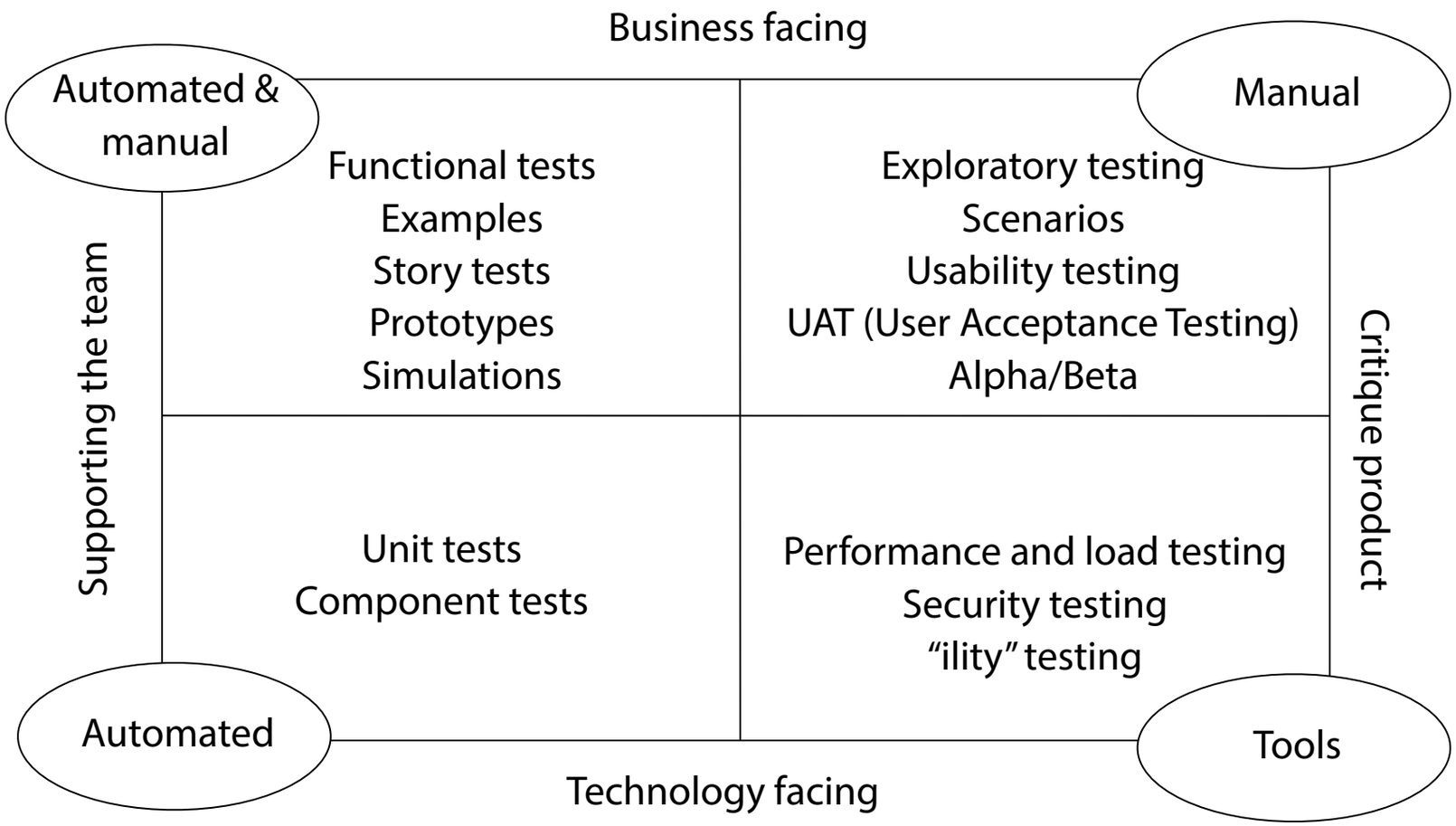
Source Steve McConnell: http://www.stevemcconnell.com/articles/art04.htm

"Quality is an accelerator"

John Clifford, Construx Software

ASPROTUNITY
www.asprotunity.com

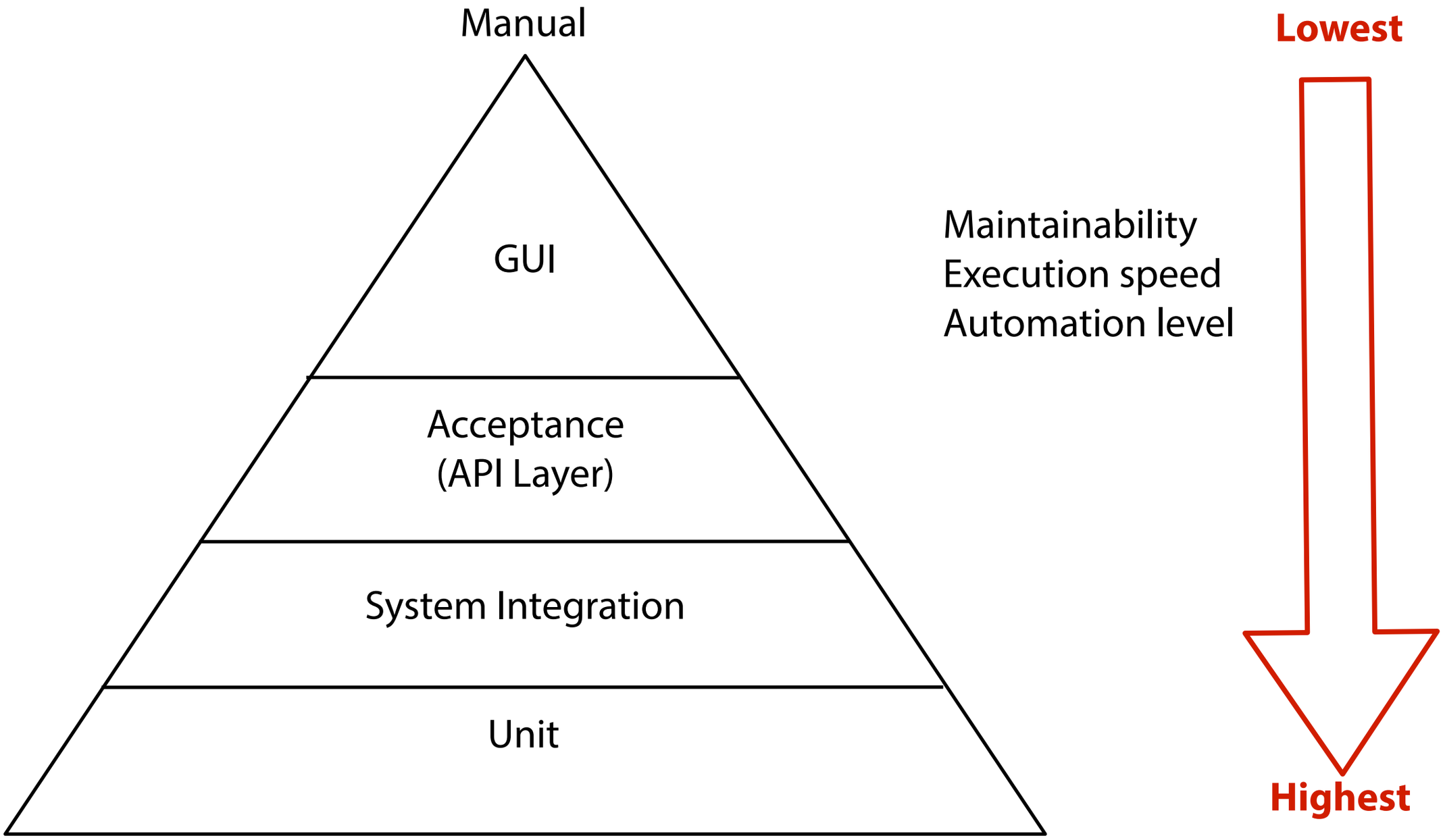
Agile Testing Quadrants



Adapted from "Agile Testing", by
Lisa Crispin and Janet Gregory

ASPROTUNITY
www.asprotunity.com

Test Automation Pyramid



Adapted from "Agile Testing"

The Need For Automation

- Automating repetitive tasks will free people up to do more value added work
  - Testers will have more time for more interesting testing activities (e.g., exploratory testing)
  - Developers will be able to refactor code and add functionality more easily
  - Users will have a better grasp of progress made

Manual Testing Is Still Important

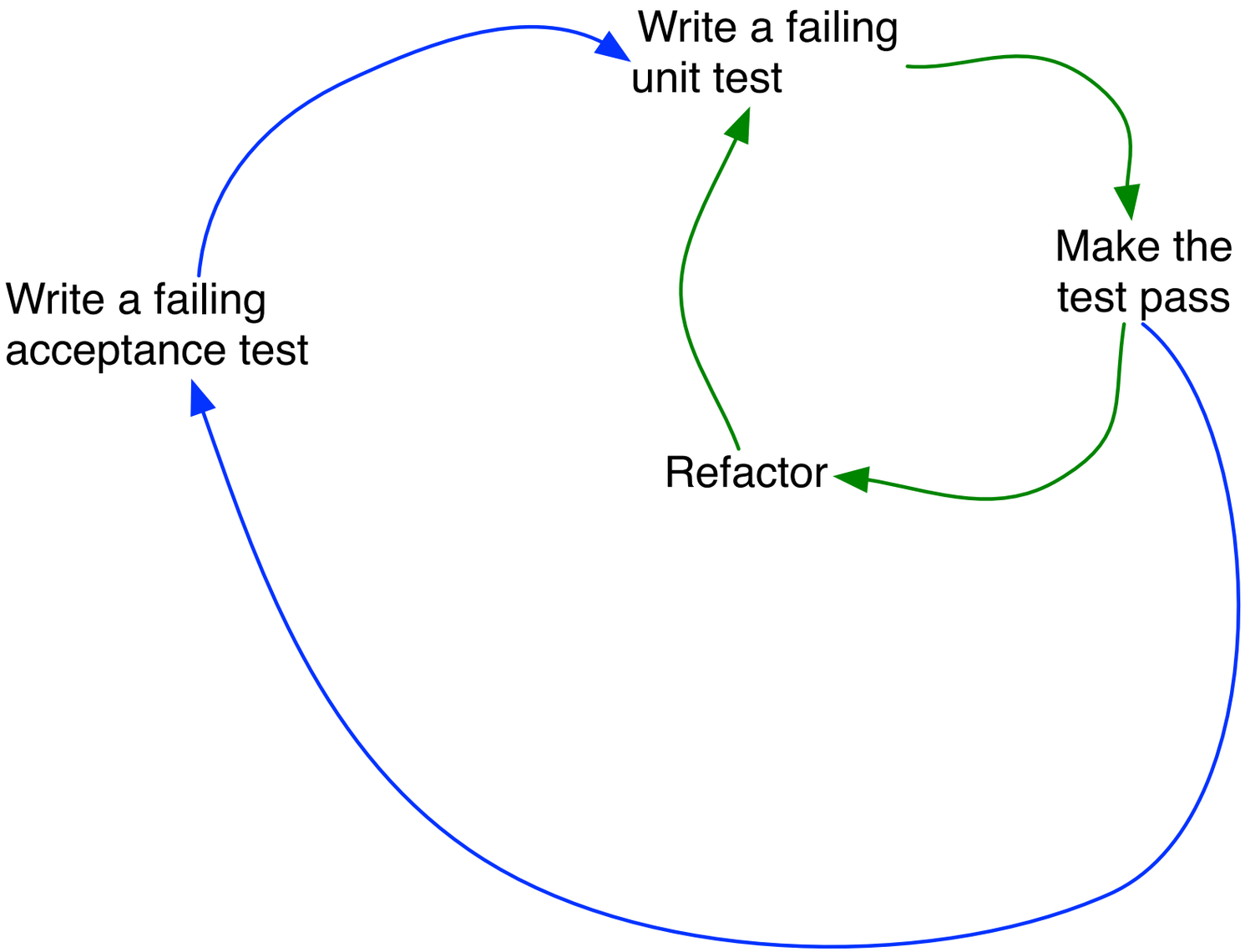
- Some tests cannot be automated, but they are still very important
- Exploratory testing
- Usability testing
- Etc.

Some Software Defects Origins

- Requirements: hardest to prevent and repair
- Design: most severe and pervasive
- Code: most numerous; easiest to fix
- Documentation: can be serious if ignored
- Bad Fixes: very difficult to find
- Bad test cases: common and troublesome

Data from Capers Jones

TDD Cycles



A Few Acronyms

- Test Driven Development (TDD)
- Acceptance Test Driven Development(ATDD)
- Behaviour Driven Development (BDD)

TDD Is…

- A method for designing software
- A way of reducing “analysis paralysis”
- A way to improve test coverage
- A documentation tool
- A way of defining “done”
- A method for making sure the requirements are understood and met (BDD, ATDD)

TDD Is...

- A method for designing software that is...
  - Testable
  - Maintainable
  - Usable
  - High quality

TDD Is…

- A way of reducing "analysis paralysis"
- Sometimes it is difficult to know were to start
- Writing a test first can help to clarify what we are trying to achieve

TDD And Design

- ~~BUFD - Big Up Front Design~~
- ~~NUFD - No Up Front Design~~
- RUFD - Rough Up Front Design
  - As detailed as necessary but no more
- Refactoring
  - Improve the design as you go

TDD Is...

- A way to improve test coverage
- This should be obvious...
- ...but remember to add the tests for the error conditions as well

TDD Is…

- A documentation tool
  - Well written tests document what the software does…
- ...are written in the language of the domain...
- ...and they are always up-to-date (provided you update and run them as you go)

Example

```
public class StackTest
{
    @Test
    public void isEmptyWhenNew()…

    @Test
    public void increasesSizeByOneAfterPushingAnElement()…

    @Test
    public void decreasesSizeByOneAfterPoppingAnElement()…

    @Test(expected = InvalidOperationException.class)
    public void cannotPopAnElementIfItsEmpty()…
     …
}
```

TDD Is…

- A way of defining "done"
  - When all the tests pass the implementation is complete
- Of course the set of tests has to be "sensible"
  - Test every important aspect of the code

Refactoring

The process of changing a software system by improving its internal structure **without** altering its external behaviour

Refactoring

- Is a "micro-technique" that is driven by finding small-scale improvements
  - Applied rigorously and consistently can lead to significant structural improvements
- It is not redesign, which is a large scale change
  - But refactoring techniques can be used to move from a design to another

Adapted from "Growing Object Oriented Software Driven by Tests", by Steve Freeman and Nat Pryce

Refactoring Does Not...

- Fix bugs
- Add or remove functionality

ASPROTUNITY
www.asprotunity.com

Refactoring (contrived) Example

Original

```
public class StackOfInts
    {
        private ArrayList<Integer> impl =
                                    new ArrayList<Integer>();
        …

        public int pop()
        {
            int lastIndex = impl.size() - 1;

            // This one may throw IndexOutOfBoundsException
            int result = impl.get(lastIndex);

            impl.remove(lastIndex - 1);

            return result;
        }
}
```

Refactored

```
public class StackOfInts
    {
        private LinkedList<Integer> impl =
                                new LinkedList<Integer>();
        …

        public int pop()
        {
            try {

                // This one may throw NoSuchElementException
                int result = impl.getLast();

                impl.remove(impl.size() - 2);

                return result;
            }
            catch(NoSuchElementException e) {
                throw new IndexOutOfBoundsException();
            }
        }
}
```

Same bug!

Anatomy Of A Test

- A test is always composed of

  1. A preparation phase where the necessary data and context are prepared

  2. The execution phase

  3. The verification phase

  4. Optional. A cleanup phase where the environment is put back in its original (before the test) state

     - This can, sometimes, be done at the beginning during the preparation phase

Example: Test Phases

```
@Test
public void decreasesSizeByOneAfterPoppingAnElement()
{
    StackOfInts stack = new StackOfInts();
    stack.push(1);
    int size = stack.Size();

    stack.pop();

    assertThat(stack.size(), equalTo(size - 1));
}
```

Preparation

Execution

Verification

No cleanup necessary in this case

Good Automated Tests Are

- A TRIP
  - Automatic
  - Thorough
  - Repeatable
  - Independent
  - Professional

Adapted from "Pragmatic Unit Testing In Java with JUnit", Andy Hunt and Dave Thomas

Automatic

- Automatic run
- Automatic verification

Example: Automatic Verification

```
@Test
public void PopDecreasesSizeByOneIfStackNotEmpty()
{

    StackOfInts stack = new StackOfInts();
    stack.push(1);
    int size = stack.Size();

    stack.pop();

    assertThat(stack.size(), equalTo(size - 1));

}
```

Verification ⟶

Thorough (1/3)

- Test everything that is likely to break
- Tricky code
- Limit conditions
- Etc.

Thorough (2/3)

- Code coverage
  - Like any measure, take it with a grain of salt
    - If it is very low (e.g., 10%) means there aren't enough tests
    - If it is very high (e.g., 99%) it may mean that several tests are not that relevant

Thorough (3/3)

- Defect distribution follows a power law
  - 80% of defects are found in 20% of the code
  - Test that 20% more thoroughly

Repeatable

- Tests must be able to run over and over, and in any order, giving always the same results
  - Everything the tests rely on must be under direct control of the tests
- Developers have their own private environments where they can build the code and run the tests
  - No data or code sharing

Independent

- From each other
- Tests should test one thing at a time
- For unit tests: also independent from the external environment

Professional

- Test code must be treated just like production code
- If the tests are messy they may hide defects
- If the tests have duplication, production code changes may cause cascade changes on tests

Unit Tests

- **Unit tests** verify the behaviour of a single class or method (or function or procedure)
- Written by developers for their own use
- Help developers describe what "done looks like"

Unit Tests Are Special

- They should **not**
  - Talk to the database
  - Communicate across the network
  - Touch the file system
  - Need special environment configuration (edit config files, etc.) to run
  - The reason is that they must be quick to run as they are run very often
    - This has an impact on the design of the application

From "Working Effectively With Legacy Code", Michael Feathers

What Will Your Code Look Like?

- More interfaces
- Smaller classes
- Smaller methods
- More cohesive
- More loosely coupled

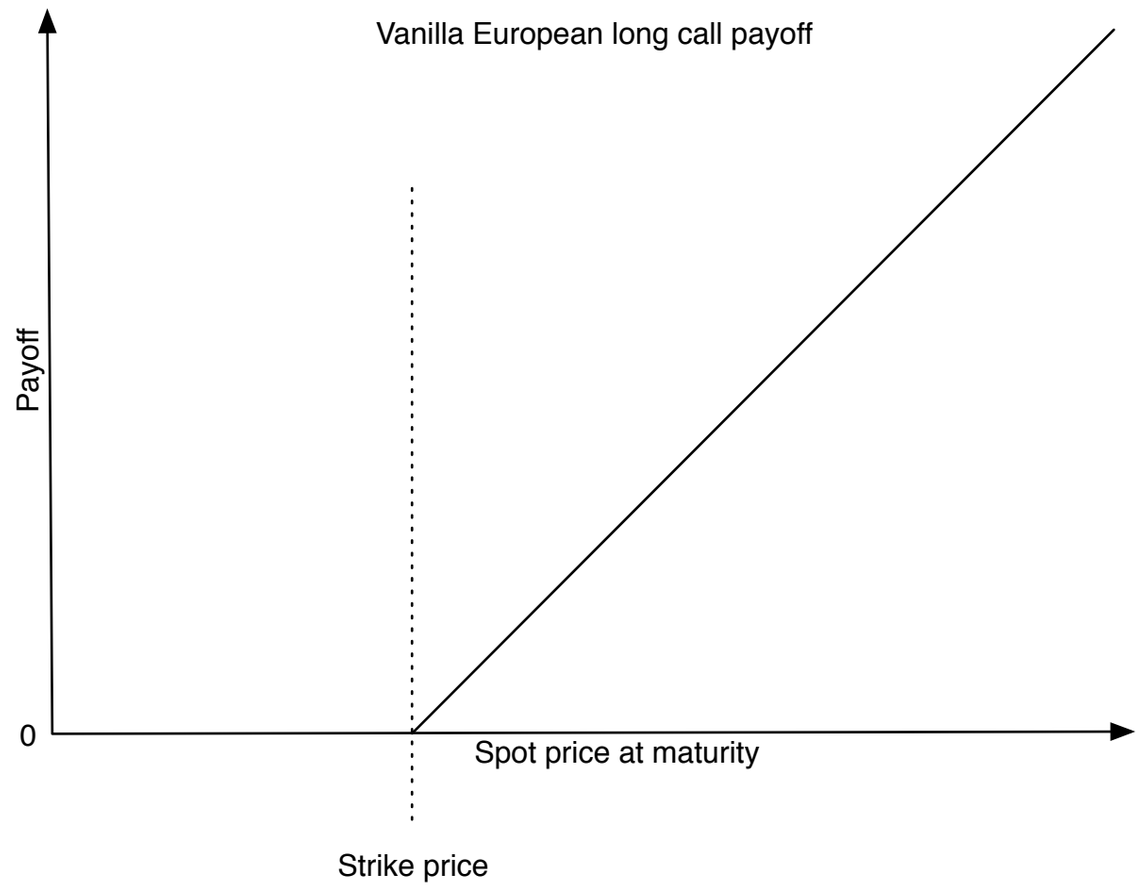
Exercise

Implement a function to calculate the payoff of an European vanilla option

ASPROTUNITY
www.asprotunity.com

European Long Call

- Can only be exercised at maturity
- Requirements
  - Use only the following operators (no pre-existing max, min, or other mathematical functions):
    - +, -, *, /, <, <=, >, >=
- Implement it using TDD



Test Doubles

- Sometimes we need to test classes that interact with other objects which are difficult to control
  - The real object has nondeterministic behaviour (e.g., random number generator)
  - The real object is difficult to set up (e.g., requiring a certain file system, database, or network environment)
  - The real object has behaviour that is hard to trigger (e.g., a network error)
  - The real object is slow
  - The real object has (or is) a user interface
  - The test needs to ask the real object about how it was used (e.g., confirm that a callback function was actually called)
  - The real object does not yet exist (e.g., interfacing with other teams or new hardware systems)

Test Doubles

- Dummies
- Stubs
- Mocks
- Fakes

Dummies

- Dummy objects are passed around but never actually used

- E.g., a mandatory argument in a constructor never used during a specific test

Stubs

- Stubs provide canned answers to calls made during the test. Stubs may also record information about calls (e.g., the number of times a method has been called)

Example: Stub

See code in com.asprotunity.exchange.server.EventServerWithStubTest

Mocks

- Mocks are objects pre-programmed with expectations which form a specification of the calls they are expected to receive

Example: Mock

See code in com.asprotunity.exchange.server.EventServerWithMockTest

Fakes

- Fake objects actually have working implementations, but take some shortcuts which make them not suitable for production (e.g., an in memory database)

Example: Fake

See code in com.asprotunity.exchange.server.FakeEventProducer

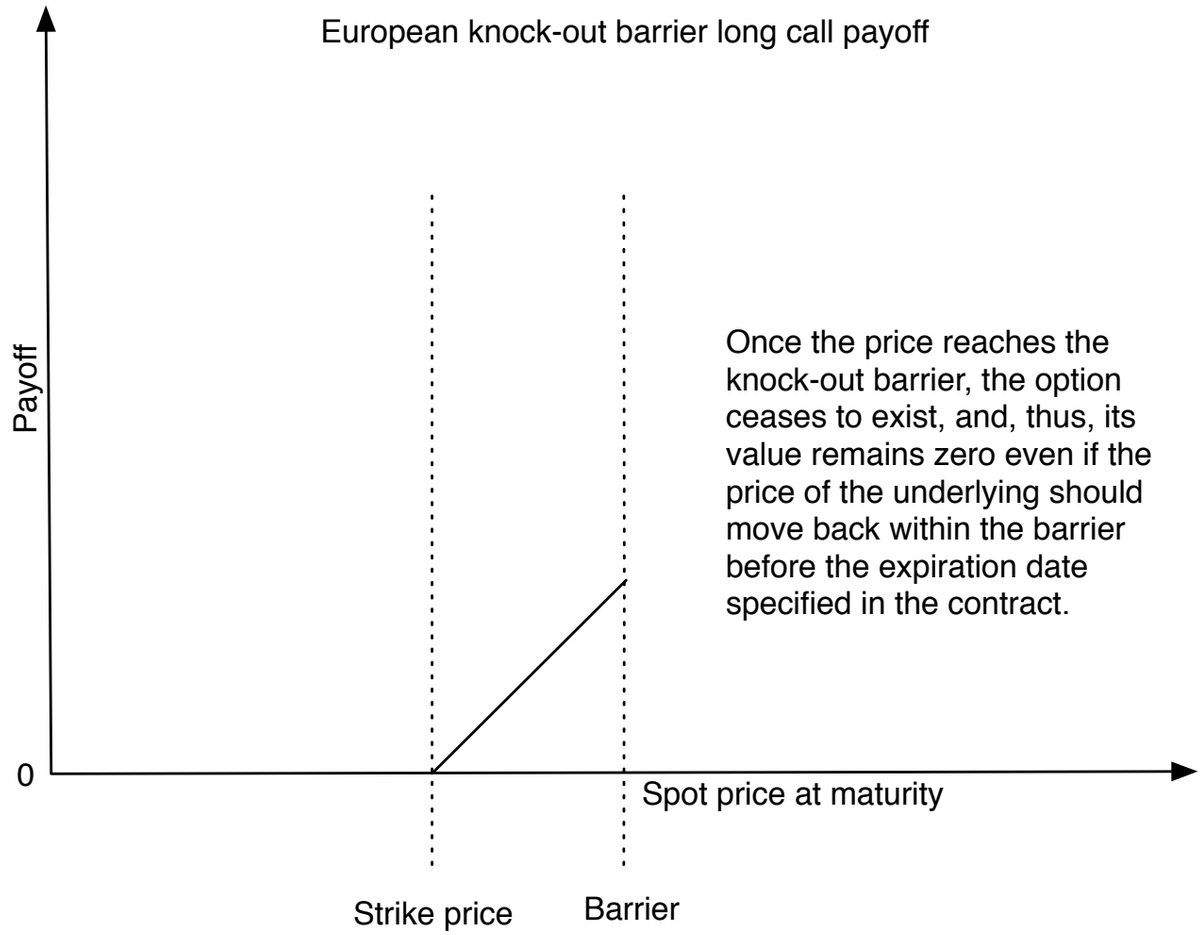
Exercise

Implement a function to calculate the payoff of an European knock-out barrier option

ASPROTUNITY
www.asprotunity.com

Knock-out Barrier Long Call

- Can only be exercised at maturity
- Requirements
  - Refactor the existing code to include the knock-out barrier
  - Use only the following operators (no pre-existing max, min, or other mathematical functions):
    - +, -, *, /, <, <=, >, >=
- Implement it using TDD



Classic And London School Of TDD

- Classic
  - Algorithmic in its emphasis
  - Triangulate through a sequence of test examples and generalise the solution as we go until we've covered just enough test cases to produce the general solution
- London
  - Focuses on roles responsibilities and interactions as opposed to algorithms
  - Identify the roles, responsibilities and key interactions/collaborations between roles in an end-to-end implementation of the solution to satisfy a system-level scenario or acceptance test. Implement the code needed in each collaborator, one at a time, faking it's direct collaborators and then work your way down through the "call stack" of interactions until you have a working end-to-end implementation that passes the front-end test

Adapted from: http://codemanship.co.uk/parlezuml/blog/?postid=987

Testing Times

This Is About...

- The use of dates and times in software applications
  - Typical mistakes and some solutions

Some Typical Mistakes

- Using different date representations inside the application
- Creating date / time instances directly
- Ignoring timezones, hours and minutes
- Multiple independent ways for providing reference dates and times

Date And Time Representations

```
public LocalDate calculateNewDate(long refDateExcel){
  …
  if (refDateExcel < 0) {
    throw InvalidDateException(…);
  }
  LocalDate refDate = … //do some translation from refDateExcel;
  // Do something to calculate the new date
  …
  LocalDate result = …
  …
  return result;
}
…
public long calculateNewDate(LocalDate date){
  …
}
```

Effects On Code

- Test and production code messy and difficult to write and follow
- Redundant data transformations
- It makes some of the problems listed before worse
  - Creating date / time instances directly
  - Multiple independent ways for providing reference dates and times

The (Obvious) Solution

- Choose a single representation in your domain and refactor your code accordingly

- Translate the different formats at the borders of your domain

- TDD helps. By driving the design with tests we are more likely to put only one date representation in our code

Date And Time Representations

```
public LocalDate calculateNewDate(LocalDate refDate){
  …
  // Do something to calculate the new date
  …
  LocalDate result = …
  …
  return result;
}
```

ASPROTUNITY
www.asprotunity.com

Creating Date And Time Instances Directly

```
public double calculateVolatility(String ticker,
                                  LocalDate refDate,
                                  DbConnection connection){
  …
  LocalDate today = new LocalDate();
  double result = 0.0;
  if (today.equals(refDate)) {
    …
    result = <some value>;
  }
  else {
    …
    result = <some other value>;
  }

  return result;
}
```

Effects On Code

- Tests difficult to write
  - How do you test for “today” in the previous example?

A Solution: Time Provider

- It is a factory that implements an interface
- The interface has a “production” implementation and one or more for testing
- The implementation for testing may allow for time travel

A Solution: Time Provider

```
public interface TimeProvider {
    LocalDate now();
}
```

A Solution: Time Provider

```
public class ProdTimeProvider implements TimeProvider {
    // Some fields
    …

    public ProdTimeProvider(…) {
      …
    }

    @Override
    LocalDate now() {
      …
    }

    // Some other methods
    …
}
```

A Solution: Time Provider

```
public class TestTimeProvider implements TimeProvider {
    // Some fields
    …

    public TestTimeProvider(LocalDate now) {
      …
    }

    @Override
    LocalDate now() {
      …
    }

    // Some other methods
    public void incrementByDays(int days) {
      …
    }
    …
}
```

ASPROTUNITY
www.asprotunity.com

Creating Date And Time Instances Revamped

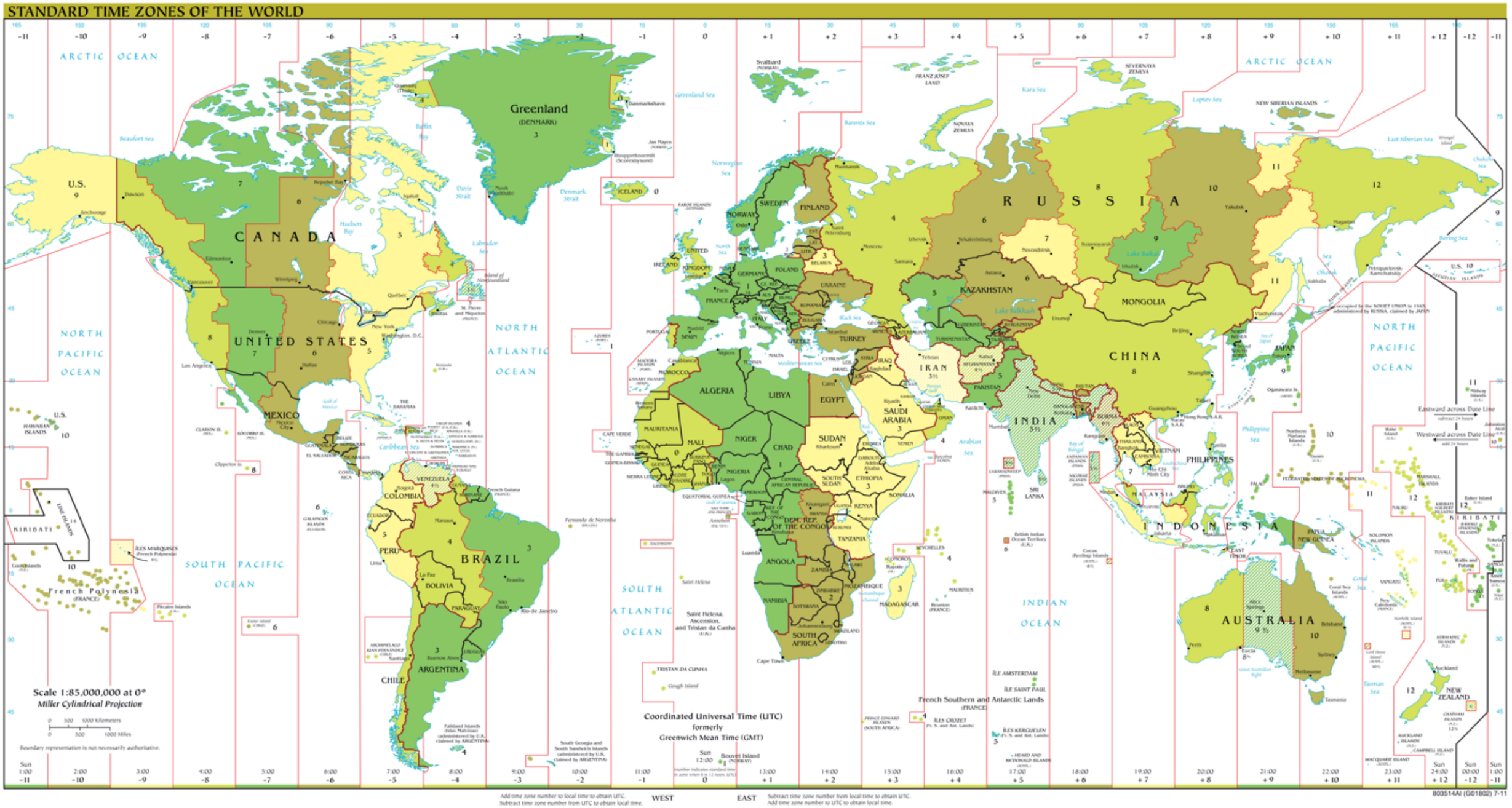
```
public double calculateVolatility(String ticker,
                                  LocalDate refDate,
                                  DbConnection connection){
  …
  LocalDate today = timeProvider.now()
  double result = 0.0;
  if (today.equals(refDate)) {
    …
    result = <some value>;
  }
  else {
    …
    result = <some other value>;
  }

  return result;
}
```

Avoid Using A Singleton For The Time Provider

- ...Your tests will thank you for that...

What Date Is Today?

It depends



from https://en.wikipedia.org/wiki/File:Standard_time_zones_of_the_world.png

Local dates and times are often not good enough

Dates And Timezones Are Not Good Enough Either

- Dates overlap in different timezones
- Today here can be tomorrow or yesterday somewhere else

Hours And Minutes To The Rescue

- The only sensible way is to use hours and minutes as well

Time Provider Slightly Modified

Before

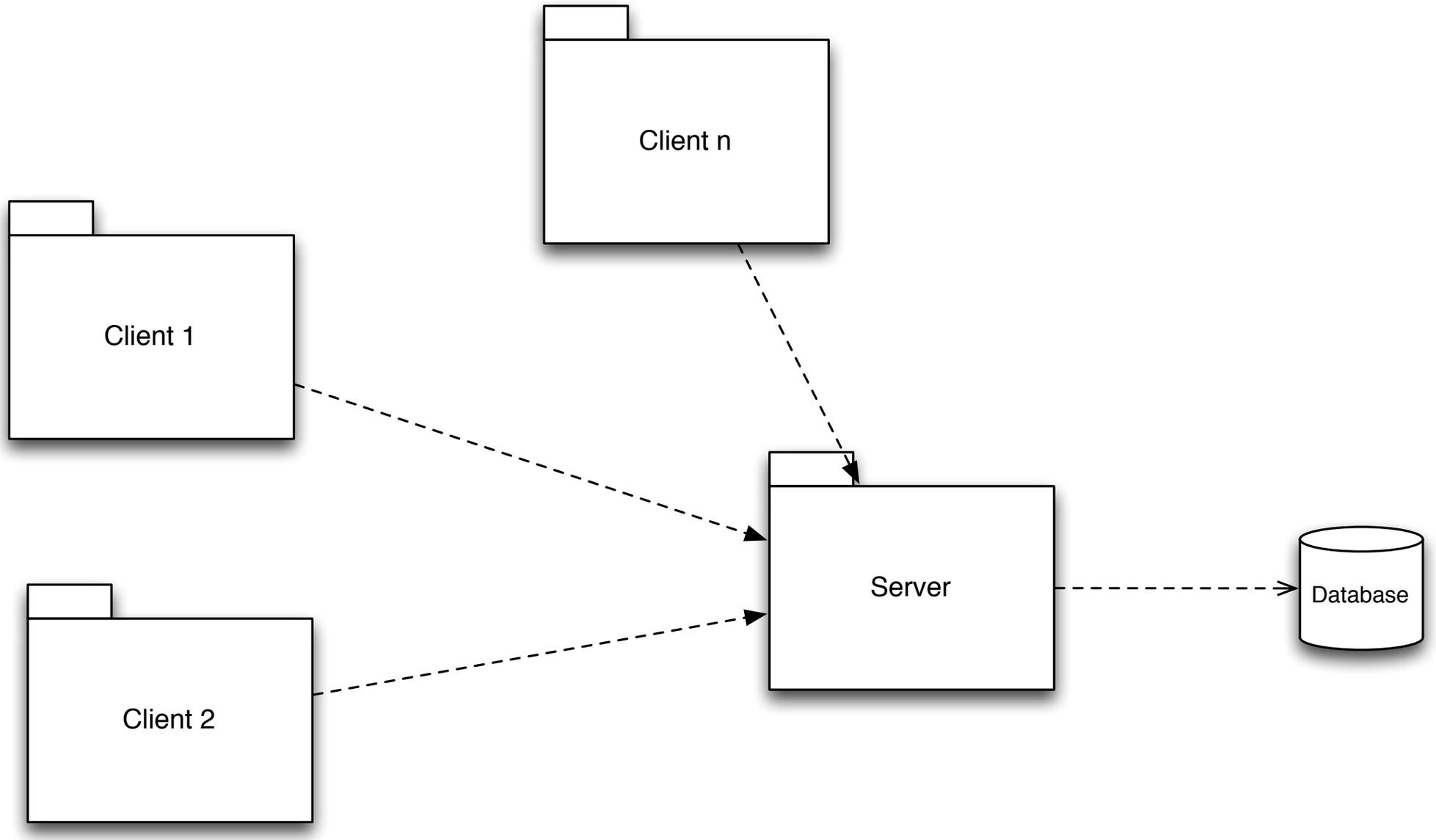
```
public interface TimeProvider {
    LocalDate now();
}
```

After

```
public interface TimeProvider {
    DateTime now();
}
```

Independent Ways For Providing Reference Dates

- Different date formats often come from different sources
- A more interesting one: Client Server systems



Effects On The Application

- If each client sent its own time to the server
  - A wrong setup on the client can cause the server to store the wrong time information
  - It could be difficult to reliably compare times on data sent by different clients
  - ...Especially if they are sent without the necessary timezone information

A Solution

- The time provider lives in the server
  - Use an absolute reference like UTC
- Clients send times to the server for informational purposes only
- Clients can use their timezones to translate the time information from the server into local time if necessary
- TDD on the server side can help in asking the right questions about what to do with the time information

Summing Up

- Refactor to use one date / time API only
- Introduce a factory for dates and times
- Use TDD to help you avoiding the mistakes in the first place
- Avoid the singleton pattern for the time provider
- ...That would make the code difficult to test, or difficult to manage, or both
- ...But if you use TDD you will discover that anyway...

Another look at the exercises

Technical Debt

- If you don't keep the production and test code clean, you'll accumulate **technical debt**
  - The code will be more difficult to extend and maintain
  - It will be easier to introduce defects
  - Just like financial debt, the later you pay it, the higher the interest charge will be

Technical Debt

- Just like financial debt, it may be necessary, but it has to be a **conscious choice**
  - An urgent bug fix may have a quick and dirty solution, or a better but more difficult to implement one
- If you don’t repay the debt as soon as possible, you’ll be in serious trouble

Boy Scout Rule

- Leave the campground cleaner than you found it

- To be applied when modifying existing code

- Small cleanups can go a long way

From “97 Things Every Programmer Should Know”, Bob Martin contribution

Talking Tests: Some Common Test Mistakes

The Rigorous

```
public void someTestMethod {
    // initialise only relevant data

    result = object.testingThis(args);

    // only relevant assertions here
}
```

The Logorrheic

```
public void someTestMethod {
    // initialise some context
    // initialise something else
    // plenty of mocking here
    // something irrelevant here
    // blah blah...

    result = object.testingThis(args);

    // some assertions here
}
```

The Confused (1/3)

```
public void someTestMethod {
    // optionally initialise some context

    result = object.testingThis(args);

    // ...gosh I didn't assert anything!
}
```

The Confused (2/3)

```
public void someTestMethod {
    // initialise some context
    // initialize some testing doubles
    //(mocks, fakes and stubs)

    result = aTestingDouble.testingThis(args);

    // assert something here
}
```

The Confused (3/3)

```
public void someTestMethod {
    // initialise some context

    result = object.testingThis(args);

    assertTrue(thisIsATautology);
}
```

The Know-it-all

```
public void someTestMethod {
    // initialise data
    result = object.testingThis(args);
    // some assertions here
    // initialise more data
    result = object.testingThat(other_args)
    // some more assertions here
    // initialise even more data
    result = object.testingAlsoThis(other_args)
    // even more assertions here
    // and so on and so forth...
}
```

The Hush Up

```
public void someTestMethod {
    try {
        // initialise context

        result = object.testingThis(args);

        // some assertions here
    }
    catch (Throwable t) {
    }

}
```

The Laconic

```
public void someTestMethod {
    makeAllAssumptionsInvisibleHere();

    result = object.testingThis(args);

    // some assertions here
}
```

The Laconic (Variation)

```
public void someTestMethod {
    // Rely on some defaults of the
    // application for the context

    result = object.testingThis(args);

    // some assertions here
}
```

The Very Laconic

```
public void someTestMethod {
    // This test has been left
    // intentionally blank
}
```

Day 2

Any questions from yesterday?

TDD Of Database Code

- Different approaches
  - Use SQL testing frameworks
  - Write the tests in Java
    - This is the approach I suggest

TDD Of Asynchronous Code

- Succeed Fast: Make asynchronous tests detect success as quickly as possible so that they provide rapid feedback (taken from "GOOS" book)
  - Use polling or notifications to detect success
- It may require some trial and error

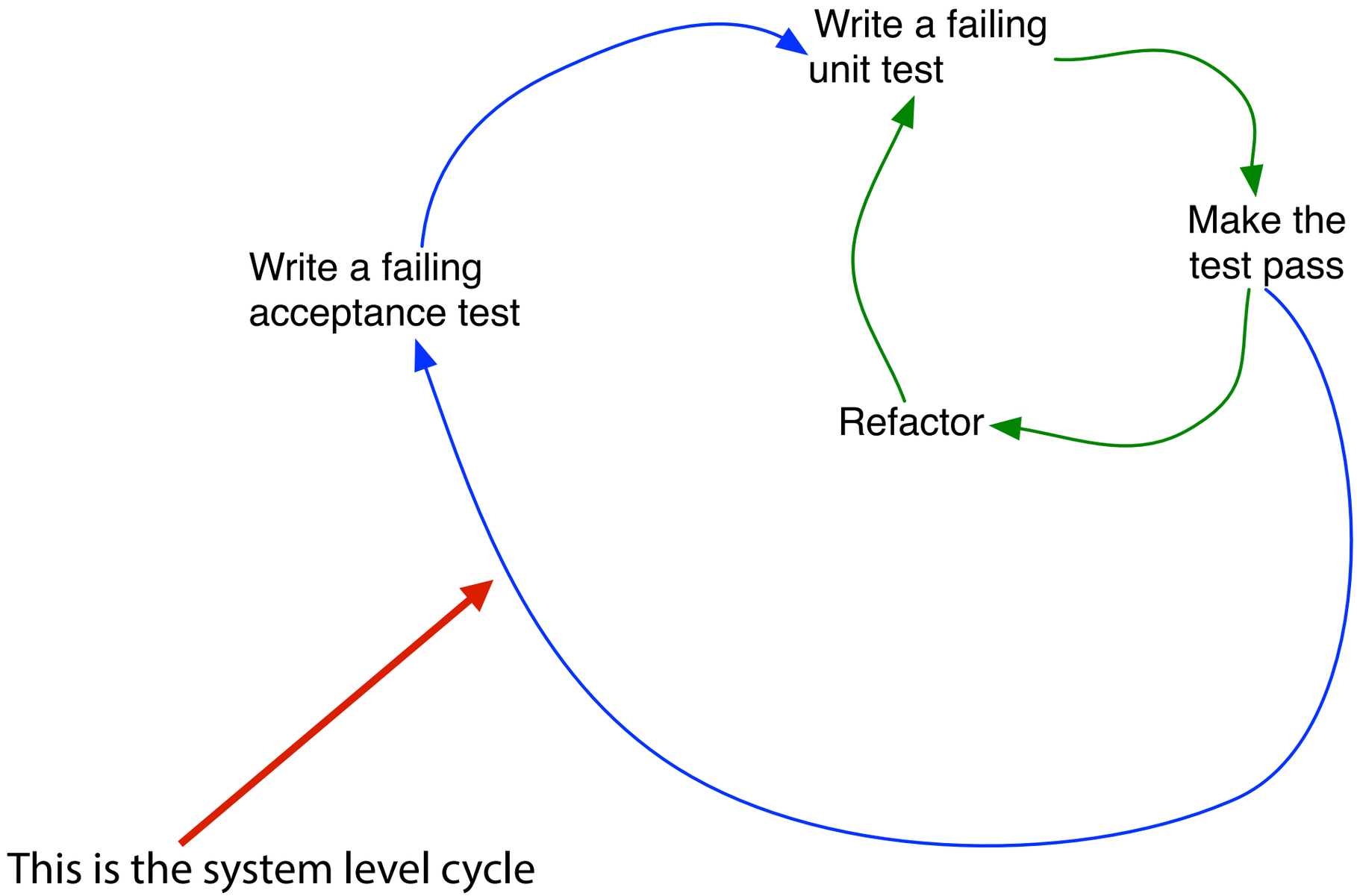
TDD At The System Level

- What it is
- Outside-in development and User Stories
- Impact on design, architecture and implementation

Different Levels Of System Testing

- Automated acceptance testing
  - ATDD, BDD
  - Integration tests
  - End to end tests
  - System tests
  - etc.

What It Is



BDD, ATDD...

BDD (Behaviour Driven Development) and ATDD (Acceptance Testing Driven Development) are two acronyms for the same thing: to create a better understanding of the requirements for all stakeholders, and (usually) to produce executable specifications for the application--which then will become the acceptance criteria.

Outside-in Development

Outside–in development focuses on satisfying the needs of stakeholders. The underlying theory behind outside–in software is that to create successful software, you must have a clear understanding of the goals and motivations of your stakeholders.

From: http://en.wikipedia.org/wiki/Outside–in_software_development

User Story

- As a Trader I want to be able to price an option based on the latest market data of the underlying asset

Some Tools For BDD And ATDD

- JBehave, http://jbehave.org
- Cucumber / Gherkin, http://cukes.info
- FIT (not used that much anymore), http://fit.c2.com
- FitNesse, http://fitnesse.org
- JUnit, http://www.junit.org

Gherkin

```
Feature: Some terse yet descriptive text of what is desired
  In order to realize a named business value
  As an explicit system actor
  I want to gain some beneficial outcome which furthers the goal

  Scenario: Some determinable business situation
    Given some precondition
      And some other precondition
     When some action by the actor
      And some other action
      And yet another action
     Then some testable outcome is achieved
      And something else we can check happens too

Scenario: A different situation
```

```
@Given("^some precondition$")
public void some_precondition() throws Throwable {
    // Express the Regexp above with the code you wish you had
}

@Given("^some other precondition$")
public void some_other_precondition() throws Throwable {
    // Express the Regexp above with the code you wish you had
}

@When("^some action by the actor$")
public void some_action_by_the_actor() throws Throwable {
    // Express the Regexp above with the code you wish you had
}

@When("^some other action$")
public void some_other_action() throws Throwable {
    // Express the Regexp above with the code you wish you had
}

@When("^yet another action$")
public void yet_another_action() throws Throwable {
    // Express the Regexp above with the code you wish you had
}

@Then("^some testable outcome is achieved$")
public void some_testable_outcome_is_achieved() throws
Throwable {
    // Express the Regexp above with the code you wish you had
}
```

JUnit

```
public class UserStoryNameGoesHereTest {

    @Test
    public void someDeterminableBusinessSituation() {
        ensureSomePrecondition(...);
        ensureSomeOtherPrecondition(...);

        actor.someAction();
        actor.someOtherAction();
        actor.yetAnotherAction();

        assertThatSomeTestableOutcomeIsAchieved();
    }
}
```

TDD At The System Level And Design

- RUFD - Rough Up Front Design
  - As detailed as necessary but no more
    - The “ilities” need to be taken care of early
- System developed in thin vertical slices
  - Walking skeleton

Ilities

- They are also known as **non-functional requirements** or **qualities**
  - Usability
  - Scalability
  - Security
  - Performance
  - Etc.

All quality requirements should be quantified.

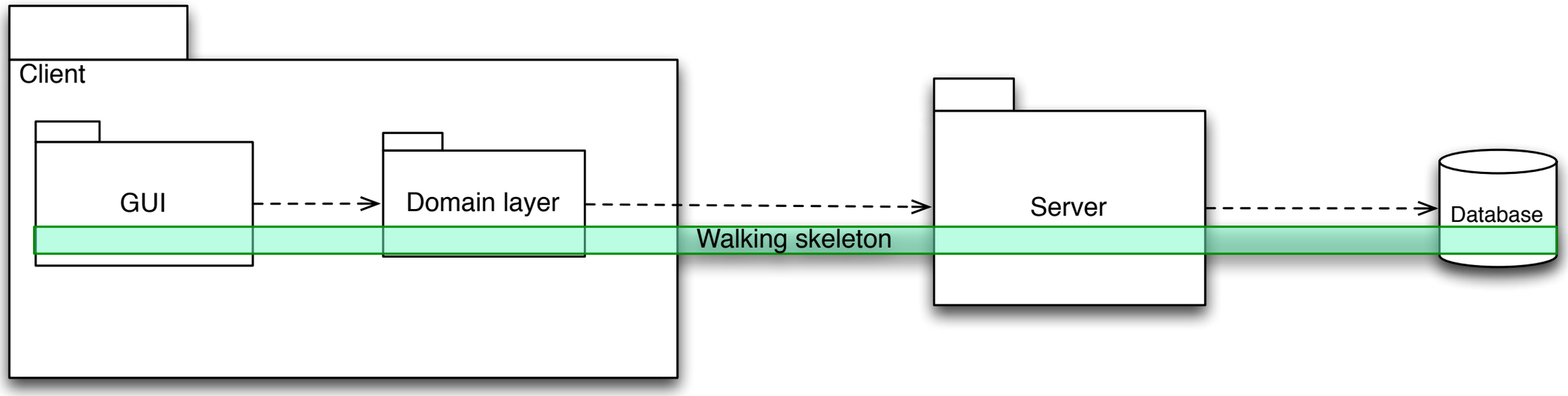
Tom Gilb, ACCU 2010 conference

Walking Skeleton

A Walking Skeleton is a tiny implementation of the system that performs a small end-to-end function. It need not use the final architecture, but it should link together the main architectural components. The architecture and the functionality can then evolve in parallel.

From: http://alistair.cockburn.us/Walking+skeleton

Walking Skeleton



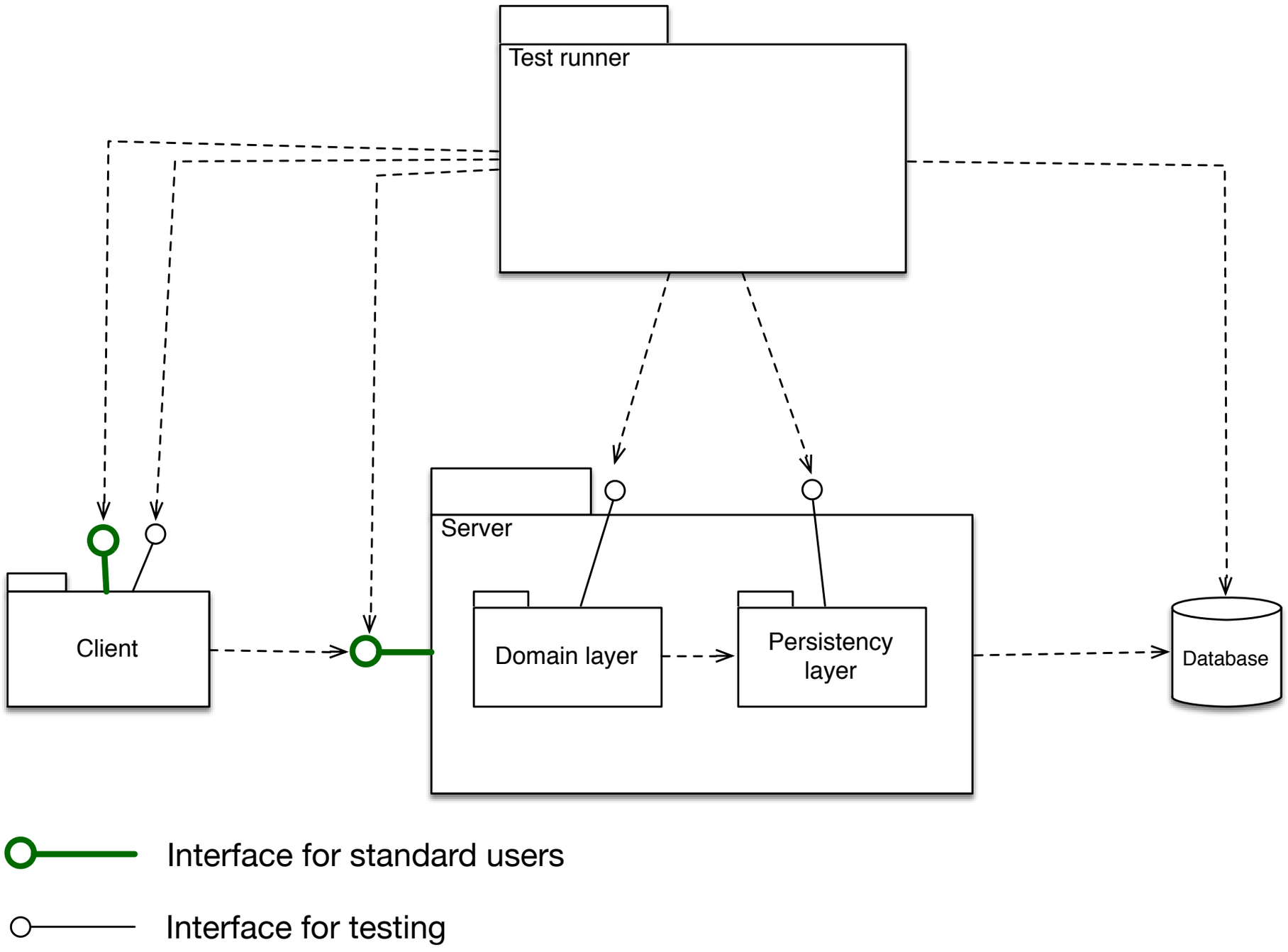
User Stories And Walking Skeletons

- Walking skeletons work well with “thin” user stories
  - The ability to slice stories that are “too big” is crucial
  - Sufficient domain knowledge in the development team is fundamental

Let's Look At The Previous Story

- As a Trader I want to be able to price an option based on the latest market data of the underlying asset

Design For Testability



TDD at all levels makes it easier to design a system for testability

Exercise

Write an option pricing platform

The Initial Story

- As a Trader I want to be able to price an option based on the latest spot price of the underlying asset

Exercise debrief

Choose Your Tests With Care

- Least testing
  - Low likelihood of failure, low consequences of failure
- Moderate testing
  - Low likelihood of failure, high consequences of failure
  - High likelihood of failure, low consequences of failure
- Most testing
  - High likelihood of failure, High consequences of failure

From: "Perfect Software and Other Illusions about Testing", Jerry Weinberg

Warning!

- TDD is just a tool in your toolset
  - It's usage doesn't make the code automatically good
  - However, the tests can highlight some design issues
  - Sometimes other techniques are more appropriate
    - GUI prototyping

Books