TDD – TEST-DRIVEN DEVELOPMENT

Gusztáv Adamis
adamis@tmit.bme.hu

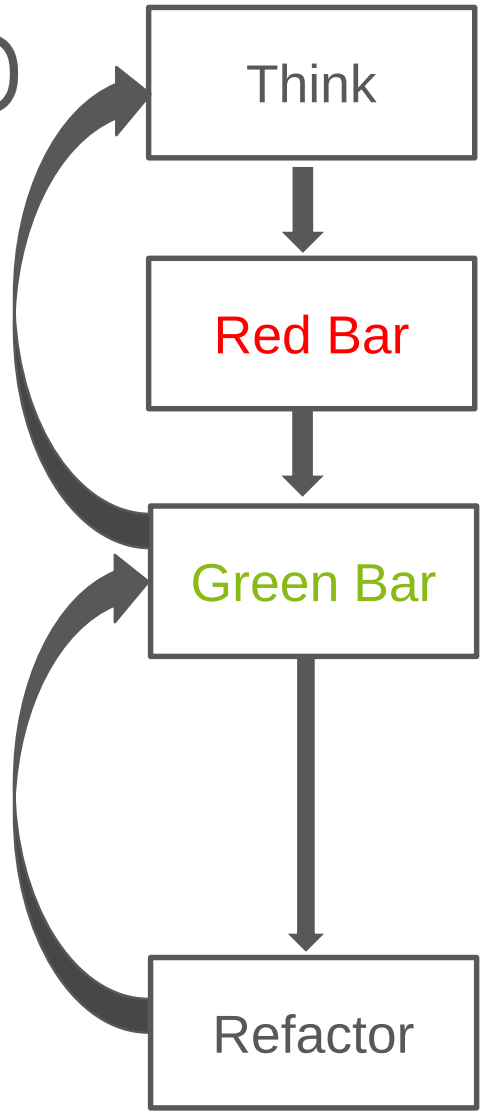
INTRODUCTION

- “What programming languages really need is a ‘DWIM’ instruction, Do what I mean, not what I say.”
- Software requires perfection
  - People not perfect – buggy code
- A tool needed to alert you immediately when a mistake is made
  - And even eliminate the need of debugging
- Test-Driven Development (Test First Development)
  - Rapid cycle of testing – coding – refactoring

INTRODUCTION

- Punchcards – intensive testing before compilation
- Language sensitive editors – can detect syntactical errors on-the-fly
- TDD – can find semantic errors (almost) on-the-fly
  - Every few minutes verifies the code
  - If error: only few lines to check
  - Therefore bugs are easy to find and fix

TDD



- New task – several lines only
- Write a test first
  - Shall fail
- Write the code
  - The simplest that passes the test
  - No future features
  - Refactor or new feature
    - Though not refactor on every cycle, always stop and seriously consider if design needs refactoring
- Refactor
  - No new feature just structure change
  - All test shall pass as before

TDD EXAMPLE: PARSE HTTP QUERY STRING

- Task:
  Program a Java class to parse an HTTP query string

- HTTP Query string:
  - http://www.example.com/mypage.html?crcat=test&crsource=test&crkw=buy-a-lot
  - Key=value pairs
  - Concatenated by &

THINK

- The first step is to imagine the features you want the code to have
- “I need my class to separate name/value pairs into a HashMap”
  - Unfortunately, this more than five lines to code -> think of a smaller increment
  - Often, the best way to make the increments smaller is to start with seemingly trivial cases
- “I need my class to put *one* name/value pair into a HashMap”

EXAMPLE – TEST WRITING

```
public void testOneNameValuePair() {
      QueryString query = new QueryString("name=value");
      assertEquals(1, query.count());
}
```

› To compile must be added:

```
public class QueryString {
      public QueryString(String queryString) {}
      public int count() { return 0; }
}
```

Test fails – Red Bar

EXAMPLE – CODING

- Write the code that passes the test
- Do not count with possible further requirements – use the simplest solution
  - Code remains shortest, fastest that satisfies the needs of the *current* specification
  - Specification may change at any time, at such case no unused code left in the program
  - Open for several possible continuations

```
public int count() { return 1; }
```

Test passes – Green Bar

NEXT STEP?

- Test an empty string as an argument
- Not to deal with multiple query strings yet

EXAMPLE

```
public void testNoNameValuePairs() {
        QueryString query = new QueryString("");
        assertEquals(0, query.count());
}
public class QueryString {
        private String _query;
        public QueryString(string queryString) {
                _query = queryString;
        }
        public int count() {
                if ("".equals(_query)) return 0;
                else return 1;
        }
}
```

testNull()

Duplication from tests remove

EXAMPLE – testNull()

```
public void testNull() {
      try {
            QueryString query = new QueryString(null)
            fail("Should throw exception");
      }
      catch (NullPointerException e) {
            // expected
      }
}
public QueryString(String queryString) {
      if (queryString == null) throw new
                  NullPointerException();
      _query = queryString;
}
```

~~testNull()~~

Duplication from tests remove

NEXT STEP?

- Write valueFor() that returns the value for a name
- Just for one entry

EXAMPLE – valueFor()

```
public void testOneNameValuePair() {
      QueryString query = new QueryString("name=value");
      assertEquals(1, query.count());
      assertEquals("value", query.valueFor("name"));
}

public String valueFor(String name) {
      String[] nameAndValue = _query.split("=");
      return nameAndValue[1];
}
```

Duplication from tests remove

Test not existing name

Test if 0 or more than 1 =

EXAMPLE MULTIPLE NAME/VALUE PAIRS

```
public void testMultipleNameValuePairs() {
      QueryString query = new
          QueryString("name1=val1&name2=val2&name3=val3");
      assertEquals("val1", query.valueFor("name1"));
      assertEquals("val2", query.valueFor("name2"));
      assertEquals("val3", query.valueFor("name3"));
}
```

EXAMPLE MULTIPLE NAME/VALUE PAIRS

```
public String valueFor(String name) {
      String[] pairs = _query.split("&");
      for (String pair : pairs) {
            String[] nameAndValue = pair.split("=");
      if (nameAndValue[0].equals(name))
            return nameAndValue[1];
      }
      throw new RuntimeException(name+" not found");
}
```

NEXT STEP?

- Write count( ) returning by the number of query strings for multiple query strings

EXAMPLE MULTIPLE count()

```
public void testMultipleNameValuePairs() {
   QueryString query = new
      QueryString("name1=val1&name2=val2&name3=val3");
   assertEquals(3, query.count());
   assertEquals("val1", query.valueFor("name1"));
   assertEquals("val2", query.valueFor("name2"));
   assertEquals("val3", query.valueFor("name3"));
}
```

EXAMPLE MULTIPLE count()

```
public int count() {
String[] pairs = _query.split("&");
return pairs.length;
}
```

But test didn’t pass…

(split returns by the orig. str if no separator found – problem if "" is the argument)

```
public int count() {
if ("".equals(_query)) return 0;
String[] pairs = _query.split("&");
return pairs.length;
}
```

TESTS

```
public class QueryStringTest extends TestCase {
        public void testOneNameValuePair() {
                QueryString query = new QueryString("name=value");
                assertEquals(1, query.count());
                assertEquals("value", query.valueFor("name"));
        }
        public void testMultipleNameValuePairs() {
                QueryString query =
                        new QueryString("name1=val1&name2=val2&name3=val3");
                assertEquals(3, query.count());
                assertEquals("val1", query.valueFor("name1"));
                assertEquals("val2", query.valueFor("name2"));
                assertEquals("val3", query.valueFor("name3"));
        }
        public void testNoNameValuePairs() {
                QueryString query = new QueryString("");
                assertEquals(0, query.count());
        }
        public void testNull() {
                try {
                        QueryString query = new QueryString(null);
                        fail("Should throw exception");
                }
                catch (NullPointerException e) { // expected
                }
        }
}
```

CODE

```
public class QueryString {
        private String _query;
        public QueryString(String queryString) {
        if (queryString == null) throw new NullPointerException();
                _query = queryString;
        }
        public int count() {
                if ("".equals(_query)) return 0;
                String[] pairs = _query.split("&");
                return pairs.length;
        }
        public String valueFor(String name) {
                String[] pairs = _query.split("&");
                for (String pair : pairs) {
                        String[] nameAndValue = pair.split("=");
                        if (nameAndValue[0].equals(name))
                                                return nameAndValue[1];
                }
                throw new RuntimeException(name + " not found");
        }
}
```

Duplication
Refactor needed

REFACTORING

- Refactoring is the process of code improvement where code is re-organised and rewritten to make it more efficient, easier to understand, etc.
- Refactoring is required because frequent releases mean that code is developed incrementally and therefore tends to become messy
- Refactoring should not change the functionality of the code
  - Same tests shall pass/fail as before (!!!!!!!!!!)
- Automated testing simplifies refactoring as you can see if the changed code still runs the tests successfully

CODE SMELLS

- Code smells can indicate possible errors in the code structure
- They are “strong warnings”, not for sure errors
  - In SOME cases they are acceptable, but in MOST cases they indicate problems
  - E.g. smoke smell at home: it may indicate that the house is on fire but also that someone burnt into the fireplace
  - Investigation needed -> typically code refactoring required
- Some of the typical code smell types on next slides

CODE SMELLS

- Divergent Change/Shotgun surgery
  - Unrelated changes affect the same class/Have to modify multiple classes to support changes to a single idea
- Primitive Obsession/Data Clumps
  - High-level design concepts represented with primitive types (instead of a class)/ Several primitives represent a concept as a group
- Data Class/Wannabee Static Class
  - In a class only data with getters and setters/ In a class methods without meaningful state (quasi static members)
  - Combine them

CODE SMELLS

- Coddling NULLs
  - If NULL received as parameter returning by NULL
  - Typically indicates a problem that is not properly handled
  - Instead of ‘forwarding’ NULL, throw an exception when NULL received as parameter
    - Unless NULL has explicitly defined semantics
- Time Dependency
  - Class’ methods must be called in a specific order/
  - Half-Baked Objects
    - Special case of Time Dependency: first be constructed, then initialized with a method call, then used
  - Typically indicates encapsulation problems

HOW TO REFACTOR

- Proceed small sequence of small transformation instead of one large
  - Not rewriting
  - Code transformation in several small, controllable steps
  - Run tests after each small step

TESTS

```
public class QueryStringTest extends TestCase {
        public void testOneNameValuePair() {
                QueryString query = new QueryString("name=value");
                assertEquals(1, query.count());
                assertEquals("value", query.valueFor("name"));
        }
        public void testMultipleNameValuePairs() {
                QueryString query =
                        new ueryString("name1=val1&name2=val2&name3=val3");
                assertEquals(3, query.count());
                assertEquals("val1", query.valueFor("name1"));
                assertEquals("val2", query.valueFor("name2"));
                assertEquals("val3", query.valueFor("name3"));
        }
        public void testNoNameValuePairs() {
                QueryString query = new QueryString("");
                assertEquals(0, query.count());
        }
        public void testNull() {
                try {
                        QueryString query = new QueryString(null);
                        fail("Should throw exception");
                }
                catch (NullPointerException e) { // expected
                }
        }
}
```

CODE

```
public class QueryString {
        private String _query;
        public QueryString(String queryString) {
        if (queryString == null) throw new NullPointerException();
                _query = queryString;
        }
        public int count() {
                if ("".equals(_query)) return 0;
                String[] pairs = _query.split("&");
                return pairs.length;
        }
        public String valueFor(String name) {
                String[] pairs = _query.split("&");
                for (String pair : pairs) {
                        String[] nameAndValue = pair.split("=");
                        if (nameAndValue[0].equals(name))
                                                return nameAndValue[1];
                }
                throw new RuntimeException(name + " not found");
        }
}
```

NEXT STEP?

- Eliminate duplication
- Single method that does the parsing
- The other methods call this rather parsing themselves
- This parser shall be called from constructor and parses the query string into a HashMap

- But this would be too large step
- Do it step-by-step
- First introduce HashMap to valueFor()

REFACTORING EXAMPLE

```
public String valueFor(String name) {
      HashMap<String, String> map = new
                  HashMap<String, String>();
      String[] pairs = _query.split("&");
      for (String pair : pairs) {
            String[] nameAndValue = pair.split("=");
            map.put(nameAndValue[0], nameAndValue[1]);
      }
      return map.get(name);
}
```

After making this refactoring the tests pass

NEXT STEP?

- Extract the parsing logic into its own method
  - parseQueryString()
- Extract Method refactoring technique

REFACTORING EXAMPLE

```
private HashMap<String, String> parseQueryString() {
      HashMap<String, String> map = new HashMap<String,
                                          String>();
      String[] pairs = _query.split("&");
      for (String pair : pairs) {
            String[] nameAndValue = pair.split("=");
            map.put(nameAndValue[0], nameAndValue[1]);
      }
      return map;
}
public String valueFor(String name) {
      HashMap<String, String> map = parseQueryString();
      return map.get(name);
}
```

REFACTORING EXAMPLE

- The tests passed again
  - Small steps -> be surprised if they didn’t
- Key point in refactoring:
  - By taking small steps, you remain in complete control of changes, which reduces surprises
  - Or if test fails: you know exactly where the problem is

NEXT STEP?

- Make parseQueryString() available to every method
  - Introduce a _map instance variable to class – that stores the hash table that can be accessed by every method

REFACTORING EXAMPLE

```
public class QueryString {
        private String _query;
        private HashMap<String, String> _map = new
                                        HashMap<String, String>();
...
        public String valueFor(String name) {
        HashMap<String, String> map = parseQueryString();
        return map.get(name);
        }
        private HashMap<String, String> parseQueryString() {
                String[] pairs = _query.split("&");
                for (String pair : pairs) {
                        String[] nameAndValue = pair.split("=");
                        _map.put(nameAndValue[0], nameAndValue[1]);
                }
        return _map;
        }
}
```

NEXT STEP?

- When instance variable introduced, no need for the return value in parseQueryString()

REFACTORING EXAMPLE

```
public class QueryString {
        private String _query;
        private HashMap<String, String> _map = new
                                HashMap<String, String>();
...
        public String valueFor(String name) {
                HashMap<String, String> map = parseQueryString();
                return _map.get(name);
        }
        private void HashMap<String, String> parseQueryString() {
                String[] pairs = _query.split("&");
                for (String pair : pairs) {
                        String[] nameAndValue = pair.split("=");
                        _map.put(nameAndValue[0], nameAndValue[1]);
                }
        return _map;
        }
}
```

NEXT STEP?

› parseQueryString() now can be called from constructor

REFACTORING EXAMPLE

```
public class QueryString {
        private String _query;
        private HashMap<String, String> _map = new
                    HashMap<String, String>();
        public QueryString(String queryString) {
              if (queryString == null) throw new
                                 NullPointerException();
                    _query = queryString;
                    parseQueryString();
              }
...
        public String valueFor(String name) {
        parseQueryString();
        return _map.get(name);
        }
...
}
```

REFACTORING EXAMPLE

- Seems like a simple refactoring
  - Moved only *one line* of code
- Yet tests fail
  - Parse method didn’t work with an empty string
- This shows why taking small steps is such a good idea
  - Because only one line of code was changed, can be known exactly what had gone wrong

REFACTORING EXAMPLE

```
private void parseQueryString() {
      if ("".equals(_query)) return;

      String[] pairs = _query.split("&");
      for (String pair : pairs) {
            String[] nameAndValue = pair.split("=");
            _map.put(nameAndValue[0], nameAndValue[1]);
      }
}
```

NEXT STEP?

› Finally remove parsing from count()

REFACTORING EXAMPLE

› From:

```
public int count() {
if ("".equals(_query)) return 0;
String[] pairs = _query.split("&");
return pairs.length;
}
```

› To:

```
public int count() {
return _map.size();
}
```

NEXT STEP?

- Remove _query instance variable that stored the unparsed query string
- Pass the query string as a parameter

REFACTORING EXAMPLE

```
public class QueryString {
        private HashMap<String, String> _map = new HashMap<String, String>();
        public QueryString(String queryString) {
                if (queryString == null) throw new NullPointerException();
                        parseQueryString(queryString);
                }
        public int count() {
                return _map.size();
        }
        public String valueFor(String name) {
                return _map.get(name);
        }
        private void parseQueryString(String query) {
                if ("".equals(query)) return;
                String[] pairs = query.split("&");
                for (String pair : pairs) {
                        String[] nameAndValue = pair.split("=");
                        _map.put(nameAndValue[0], nameAndValue[1]);
                }
        }
}
```

SIMPLE DESIGN

- Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away. (Antoine de Saint-Exupéry)

- Any intelligent fool can make things bigger, more complex and more violent. It takes a touch of genius and a lot of courage to move in the opposite direction. (Albert Einstein)

SIMPLE DESIGN

1. The system (code and tests together) must communicate everything you want to communicate

2. The system must contain no duplicate code

3. The system should have the fewest possible classes

4. The system should have the fewest possible methods

1. and 2. together: Once and Only Once rule

YAGNI – YOU AREN'T GONNA NEED IT

- Avoid speculative coding
  - No functionality shall be added early
    - Only those that are required by the current requirement (story)
  - Requirement may change
  - Unnecessary code can remain
    - Slow, harder to maintain
  - Since they are not needed, they are not well defined and implemented
  - Remove code that's no longer in use
    - It remains in version ctrl system if needed in future again
- One of the hardest things for developers not to do!
  - We are all tempted to add functionality now that we are just sure is going to be needed later
  - Extra functionality always slows us down and squanders resources

ONCE AND ONLY ONCE

- Remove code duplication
  - But don't just eliminate duplication; make sure that every important concept has an explicit representation in your design
- Rather than expressing concepts with a primitive data type, create a new type (class)

E.g. instead of representing a Dollars with simple decimal, create a class

```
public class Dollars {
        private decimal _dollars;
        public Dollars(decimal dollars) { _dollars = dollars; }
        public decimal AsDecimal() { return _dollars; }
        public boolean Equals(object o) {...}
}
```

ONCE AND ONLY ONCE

- Although basic data types may seem simpler (one less class), actually make your design more complex: no place for the concept
- As a result, when working with that concept, the code may need to re-implement basic behavior - widespread duplication
  - string parsing,
  - formatting,
  - simple operations
- Though duplication may be only fragments of code, but make your code hard to change
  - For example, if you want negative amounts to be red, all little fragment of formatting code must be found and fixed
  - By starting with a simple but explicit representation of the concept, you provide a location for future changes to congregate
  - Without it, leads to duplication and complex code.

SELF-DOCUMENTING CODE

- Simplicity is relative
  - If the rest of your team or future maintainers of your software find it too complicated, then it is
- Use naming conventions that are common for your language and team
- Use names that clearly reflect the intent of variables, methods, classes, etc.
- Before using a comment, ask your pair how to make the code express the idea without needing a comment
- Comments aren’t bad, but they are a sign that your code is more complex than it needs to be
  - Try to eliminate the need for comments when you can
- Later in details

ISOLATE LEGACY CODE

- When calling legacy functions widespread
  - Hard to modify or replace
- Hide behind an interface that you control
  - Use adapter classes instead of instantiating legacy classes directly
  - Create your own base class that extend the legacy classes instead of extending them directly
- Isolating legacy components also allows to extend the features of the component and gives a convenient interface to write tests against
  - Implement adapter class incrementally – support only those features that are actually needed – not everything
  - Write adapter's interface to match your needs not the component
- Makes code *a bit* more complex
  - But easier if legacy code shall be replaced

LIMIT PUBLISHED INTERFACES

- *Published interfaces* reduce your ability to make changes
  - Once an interface is published it shall not be modified because it is used in several programs
- Some teams treats *internal* interfaces as published
  - It limits the ability of refactoring
  - Non-published interfaces can be changed – with their callers
- Each published interface is a design decision commitment
  - But may be changed in future
  - Limit the number of interfaces
- The smaller the interface, the better
  - Much easier to add new elements to your API than to remove or change incorrect elements

INCREMENTAL DESIGN AND ARCHITECTURE

- No time for creating a well-designed plan
  - Incremental or evolutionary design
- Similar concepts as in TDD on all levels of design
- When first create an element (method, class, architecture)
  - Be as concrete and specific as can be
  - Regardless of how simple it is and how to solve future problems
- The *second* time work with that element, modify the design to make it more general
  - But only general enough to solve that problems it needs to solve, etc.
- Breakthroughs
  - When larger refactor needed

RISK-DRIVEN ARCHITECTURE

- Risk-Driven Architecture
  - Although designing for the present, it’s OK to *think* about future problems. Just don’t *implement* any solutions to stories that you haven’t yet scheduled
  - Although it would be inappropriate to implement features your customers haven’t asked for, you *can* direct your refactoring efforts toward reducing risk

INCREMENTAL VS UP-FRONT DESIGN

- *Isn’t incremental design more expensive than up-front design?*
- Just the opposite
  - Incremental design implements just enough code to support the current requirements, you start delivering features much more quickly with incremental design
  - When a predicted requirement changes, you haven’t coded any parts of the design to support it, so you haven’t wasted any effort

PERFORMANCE OPTIMIZATION

- Nowadays computers are complex
  - Several internal units, parallelism, pipelines, caches
- Hard to calculate the performance
  - Only with measurement
  - Performance tests are end-to-end tests
    - They have to measure the performance of the whole service
  - If performance test fails
    - Modify system
    - If better - keep, otherwise - throw
    - Once performance test passes, stop: increase the performance more only if needed!
    - If refactoring – run performance tests again

PERFORMANCE OPTIMIZATION

- Major drawbacks of (performance) optimization
  - Leads to complex, hard-to-understand and maintain code
  - Takes time away from delivering new feature (choice to optimize is a choice *not* to do something else)
  - Neither is in the customer's interest
  - Optimize only if serves a real, measureable need
- Potential performance problem
  - Explain to customer in terms of business tradeoffs and risks
  - Shall be the (business) decision of the customer

ESTIMATION OF A PERFORMANCE STORY

- Similarly to bug-fixing, the duration mostly depends on how long it takes to find the cause of the problem
  - Can be hard to estimate
  - Time-box the estimation
    - If not enough, write a new story