BLStream Fingerprint

Table of Contents

1. Preface

BLStream Finger print is a set of practices applied in the company.

2. Project management

2.1. Definition of Done

2.2. Daily Standup

2.3. Demo

2.4. Scrum/Kanban Board

2.5. Planning Meeting

2.6. Retrospectives

2.7. Retrospectives shared with the customer

2.8. Project Practices Charter shared with the customer

2.9. Collocated Team (all team members PM included)

2.10. 3rd party libraries licences listed, approved

2.11. Clean Backlog

2.12. Responsive Product Owner

2.13. Grooming

2.14. BurnUp / BurnDown chart

3. Development

3.1. Unit Tests

Introduction

Unit testing is at the core of engineering practices in BLStream. It's not just a practice, it is a foundation on which many other, more sophisticated practices are built. Without unit tests it would be impossible to apply techniques like Continuous Delivery or Test Driven Development and it would make code refactorings and generally code maintenance much harder and error prone.

What do we understand by unit tests?

There are many definitions of "unit test" in the industry, see e.g. Martin Fowler - UnitTest, which makes the term rather confusing. What we understand by unit test is simply:

- they are written by programmers themselves
- they run fast, in seconds rather than minutes
- they are fine-grained, each test verifies single "thing" (in other words - there is one reason to fail).

Implementation details of how we write unit tests vary, depending on the tools and technologies the project is using. Nevertheless, common goal is to end up with a system that has complete suite of tests which "cover" all functionalities of the system. Thanks to that we are able to clean the code and to improve it anytime, not worrying that we accidentally break something. After each change we can run our tests and make sure that all works as expected. That's really powerful. It leads to higher quality of the code, fewer bugs, faster development, happier programmers and - at the end - more satisfied customers.

It also means that tests are not something additional, optional. They are required and are very important part of the system.

Testability

Writing tests might be a challenge, especially for old, legacy code where no tests were so far developed. Therefore when we start to develop a system, we pay a lot of attention from the very beginning to write testable code. Rules for writing testable code are described in many places, e.g. Guide to testable code by Misko Hevery, but details depend on project technologies, programming language paradigms, etc. As a general rule, we try to think about tests while writing production code or even to write tests before we implement a new feature. Strictly following that practice is called TDD and is described in chapter Test Driven Development. That would automatically ensure testability.

Even in legacy code without tests, we try to write tests firstly, before we make any changes. Thanks to that we can be sure that we didn't break anything while doing a bug fix or new feature development.

How we are doing unit testing?

The approach we take for writing tests is to treat tests as executable specification. It simply means that tests describe desired functionality. Each test verifies one, single behavior of the system. Since tests focus on behavior, they are not bound to implementation details. They verify public API of a "unit", ignoring private (hidden) methods and internal implementation structure.

Let's see an example. If we need to write a function that returns filtered and sorted list of items, in a test we try to verify desired behavior only by public API, ignoring the fact that filtering and sorting internally is implemented in separate functions.

Unit tests

```
@Test
public void shouldFilterActiveOnly() {
MyRepo repo = new MyRepo();

List<Item> items = repo.findItems();

assertThat(items).are(activeOnly());
}

@Test
public void shouldSortList() {
MyRepo repo = new MyRepo();

List<Item> items = repo.findItems();

assertThat(items).isSorted();
}
```

Example of implementation

```
public class MyRepo {

public List<Item> findItems() {
List<Item> items = newArrayList(new Item(), new Item());
List<Item> filteredItems = filter(items);
return sort(filteredItems);
}

private List<Item> sort(List<Item> items) {
List<Item> itemsSorted = newArrayList(items);
itemsSorted.sort(null);
return itemsSorted;
}

private List<Item> filter(List<Item> items) {

List<Item> filtered = new ArrayList<>();

for (Item item : items) {
if (item.isActive()) {
filtered.add(item);
}
}
return filtered;
}
}
```

As we see, tests use only public method `findItems` to verify both filtering and sorting and are independent of implementation details, they ignore two private functions: `filter` and `sort`. What advantages does it have? Well, firstly, all functionalities are "covered" by the tests. Secondly, we are not bound to implementation details and these details can evolve without breaking tests. For instance, in Java 8 you do filtering and sorting using new Stream API is a single chain of function calls:

Implementation in Java 8

```
public List<Item> findItems() {
List<Item> items = newArrayList(new Item(), new Item());
return items.stream()
.filter(Item::isActive)
.sorted()
.collect(Collectors.toList());
}
```

We made quite a big change, we removed 2 private methods but these are purely implementation

details. No tests require re-implementation. We are free to refactor, to improve technical details, yet the tests remain clean and stable because they are focused on **behavior**.

The tests that we presented show also basic structure of the test, which consists of 3 main steps: *Given, When, Then*, as on the example below:

Structure of a test

```
@Test
public void shouldSortList() {
MyRepo repo = new MyRepo(); # <1>

List<Item> items = repo.findItems(); # <2>

assertThat(items).isSorted(); # <3>
}
```

① Section *Given* - should set up all required objects, pre-conditions to the test

② Section *When* - behavior, action you are testing

③ Section *Then* - verifies expected state

Ideally, tests have only these 3 lines. Even if there is more to do in a test, it is always possible to refactor to these 3 lines.

We also do, from time to time, other kinds of testing which also employ unit test tools (like JUnit). These could be for instance learning tests (that verify how external library works), low level detailed tests which drive design of implementation details. Nevertheless they are not obligatory, they might be deleted when no longer needed, so not all rules mentioned above apply here. We also sometimes employ unit testing tools for integration tests which is described in a separate article Integration tests.

References

Online resources:

1. http://martinfowler.com/bliki/UnitTest.html
2. http://blog.8thlight.com/uncle-bob/2013/09/23/Test-first.html
3. http://blog.8thlight.com/uncle-bob/2014/01/27/TheChickenOrTheRoad.html
4. http://blog.arkency.com/2014/09/unit-tests-vs-class-tests/

Books:

1. http://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882
2. http://www.amazon.com/Test-Driven-Development-Kent-Beck/dp/0321146530

3. http://www.amazon.com/xUnit-Test-Patterns-Refactoring-Code/dp/0131495054

3.2. Test Driven Development

Introduction

TDD is a way of developing software in which you write tests first, before implementing a functionality. What kind of tests? It's about unit tests as defined in article Unit Tests. However, TDD additionally requires to apply a set of rules (details are below). Thanks to TDD you get source code that is fully testable, fully covered by tests, code that you can trust with your life. It means you know exactly what your code is doing (tests are executable spec), you won't be afraid to clean code, improve it, introduce any change. Your team is able to consistently go fast. Sounds great, doesn't it?
It is not trivial to write code in TDD manner though, but let's firstly see what exactly we mean by TDD.

How to do TDD?

Apart from the general rule that you write tests before any production code, there are slight differences in details, of how you do TDD. We like Uncle's Bob approach most. He defines TDD as an activity performed in 4 cycles, see TheCyclesOfTDD.

Firstly, you are not allowed to write a single line of production code without failing test. This is the lowest level rule expressed by so-called Three Laws of TDD:

IMPORTANT	1. You must write a failing test before you write any production code. 2. You must not write more of a test than is sufficient to fail, or fail to compile. 3. You must not write more production code than is sufficient to make the currently failing test pass.

Secondly, since *Getting software to work is only half of the job* (Kent Beck), we need to clean (via refactoring) the code. Therefore once you completed a unit test (or a set of small tests), you need to have a look at code you developed from a little distance and think what to improve having long term maintainability in mind. This way of looking at development cycle is called *Red-Green-Refactor*.

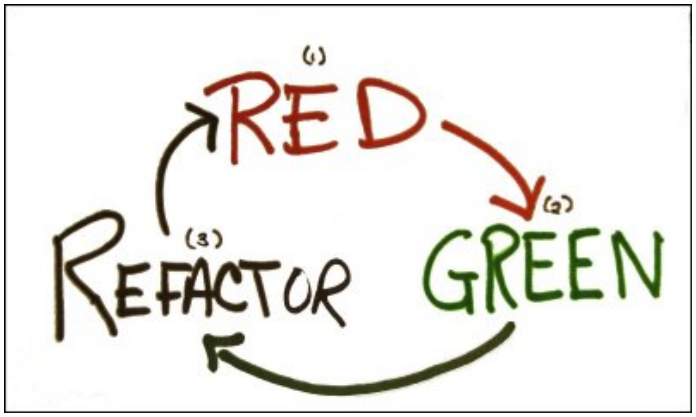

Figure 1. Red-Green-Refactor cycle

Please note that it means we don't have any separate phase in the project called *refactoring* or *code*

clean up. Refactorings and care about code quality are continuous, inherent in development process.

Thirdly, even applying previous rules, it's easy to fall into troubles. You may come up with a test that forces you to write tons of production code or even to throw away your whole current implementation. Instead, you should work in an incremental, step-by-step manner. How we can achieve that? By making our implementation code more and more generic with every subsequent test. Our code should not only fulfill requirements imposed by tests, it should naturally flow into direction of generic solution for all possible tests. Also tests themselves could break this natural process. If a test requires *revolution* in code maybe it's worth considering a smaller step. This cycle is called Specific/Generic cycle and should be considered every couple of tests. It definitely requires some practice and skills to get it right.

Fourthly, while writing small test cases and small portions of code, it might be easy to loose the big picture. Therefore from time to time (say every couple of hours), we need to consider also if our code is in line with general architecture outline, e.g. if we don't call DB directly from a GUI component. Generally, our architectures tend to follow Clean Architecture principles and as such lead to universal solutions which are highly testable, provide separation of concerns, clearly define boundaries of the system and its internal components.

In TDD tests are equally important as production code, therefore the same amount of care and attention is paid to them.

When to use TDD?

In BLStream we are not dogmatic about TDD. It is regarded as a good practice and if a team does not follow it, it must have good reason for it. Having said that, our experience is that some areas of software development are very well suited for TDD (algorithmics, finding new solutions), for others it is not very pragmatic (legacy systems, standardized issues - where the implementation is already known) and sometimes even not really possible (front-end stuff like CSS). Nevertheless we **always** write our code having testability in mind and advise to consider TDD in projects.

References

Online resources:

1. http://blog.cleancoder.com/uncle-bob/2014/12/17/TheCyclesOfTDD.html
2. http://blog.8thlight.com/uncle-bob/2013/09/23/Test-first.html
3. http://martinfowler.com/bliki/TestDrivenDevelopment.html
4. http://www.jamesshore.com/Agile-Book/test_driven_development.html

Books:

1. http://www.amazon.com/gp/product/0321146530
2. http://www.amazon.com/Growing-Object-Oriented-Software-Guided-Tests/dp/0321503627

3.3. Pair Programming

Agile software development requires, among other things, frequent feedback on all possible levels. There is no better way to get early feedback during software development than pair programming, see figure below.

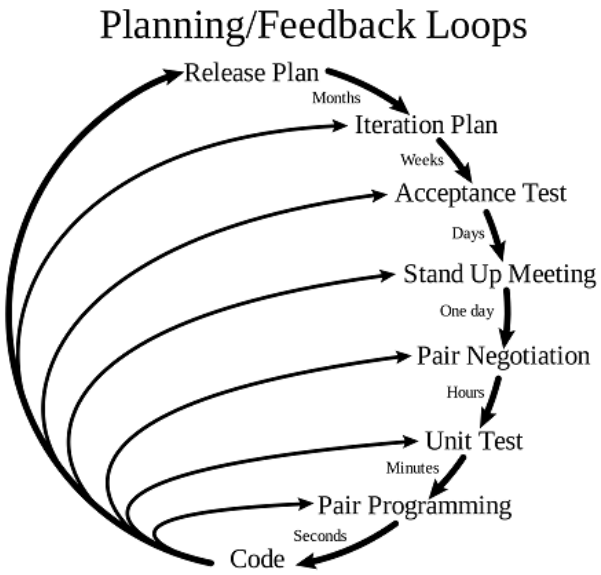

Figure 2. Feedback loops in Agile on XP example

As we see from this picture, pair programming is the lowest level feedback loop. It provides instant feedback from a developer to a developer while working on code. Sitting in pairs while programming has proved to lead to higher quality code, better knowledge sharing and mutual encouragement (see Pair Programming in Wikipedia). In BLStream we do pair programming and we found that technique very useful, especially under following circumstances:

- Development of an unclear/challening requirement
- Two people with different set of skills
- New person in a team
- New technologies and/or unknown tools

At the same time we found that technique might be inefficient for standard and well understood tasks, when it was obvious what needs to be done or when two people were equal in skills and experience. Also remote work - which is becoming more and more popular these days - makes working in pairs a little harder. Therefore we usually do pair programming for short periods of time, when it makes most sense. Rest of the time we find Code Reviews sufficient.

3.4. Code Reviews

In BLStream it is obligatory to do either Pair Programming or code reviews. Both techniques lead to higher quality, fewer bugs, sharing of knowledge and good practices among team members. Therefore, even though they require some investment of time, they lead in fact to higher productivity and lower total cost of the project, see Wikipedia for references to empirical studies.

Some time ago the dominating type of code reviews was "over the shoulder". It meant that reviewer had to come to developer's machine as the developer walked through the code. Nowadays the standard way to do code reviews is to use Git SCM together with some sort of assist tools. Firstly, code developed is usually using Git feature branches, which makes very easy for the reviewer to get the changed source code and check it in an isolated environment. Secondly, many tools emerged which mimic GitHub pull requests. These tools, e.g. Atlassian Stash, make use of Git feature branches and provide convenient web interface for both defining and reviewing pull requests. With these tools you can not only see changes to review in a commit-by-commit view, but also you can share comments, ask developer additional questions or even fire a build on CI environment (e.g. Jenkins) with tests, static code analysis, etc. They also allow to define mini-workflows, e.g. "two positive reviews are required before the change can be merged to the main branch". The following picture shows a basic screen for Atlassian Stash.

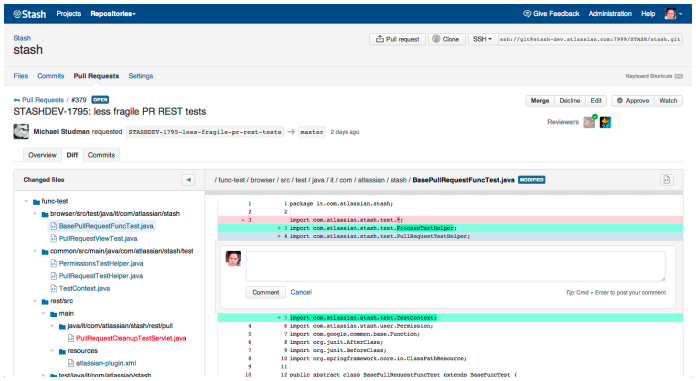

Figure 3. Stash screenshot

Of course tools do not make code review good, they just facilitate it. Teams need to define what exactly need to be checked on code reviews, especially which standards, conventions must be followed (code styles, architecture, GUI guidelines, performance impact, etc.). More on this in subsequent chapters.

3.5. Integration tests

3.6. Easy infrastructure setup

from nothing to running in <1h

3.7. Easy application setup

from nothing to running in <1h

3.8. Concurrency in application code accounted for

3.9. GUI Style Guide defined

3.10. Application Monitoring

3.11. Scalability requirements known and accounted for

3.12. Performance requirements known and accounted for

3.13. Static code analysis (backend)

3.14. Application events logging

3.15. OWASP Top 10 in Definition of Done

3.16. Authorization model defined

3.17. Continuous Integration

3.18. Continuous Delivery

3.19. Continuous Deployment

3.20. Documentation tracked in VCS

3.21. Documentation generated during CI

3.22. Parts of the documentation generated automatically

3.23. Automatic documentation of the executed tests

3.24. Documentation scope agreed

3.25. JS application framework

3.26. JS Build process

3.27. JS modules dependency management

3.28. JS Unit test

3.29. CSS builder

3.30. Static code analysis (JavaScript)

3.31. Truly RESTful interfaces

3.32. HTML validation

3.33. Database schema versioning

3.34. Database data versioning

3.35. Concurrency for DB writes

3.36. Version Control System

3.37. Branching strategy

4. Quality assurance

4.1. Radiator

4.2. Defect Tracking System

4.3. Defined bug lifecycle

4.4. At least 1 QA for every 4 developers

4.5. Bug report template

4.6. Bug triage meeting

4.7. Smoke

4.8. Integration

4.9. Functional / Acceptance

4.10. Spelling

4.11. Security

4.12. Performance

4.13. Exploratory

4.14. Usability

4.15. Versioned repository of the test scenarios

4.16. Pair testing