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. Easy infrastructure setup

from nothing to running in <1h

3.2. Easy application setup

from nothing to running in <1h

3.3. Concurrency in application code accounted for

3.4. GUI Style Guide defined

3.5. Application Monitoring

3.6. 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 a 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 accidentally.

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 it has? 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());
}
```

A change that is purely implementational, does not require tests to update. Tests remain stable, clean and are focused on behavior verification.

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 all rules mentioned above do not apply here. We also sometimes employ unit testing tools for integration tests which is described in a separate article Integration tests.

References

List of online resources which go deeper into unit testing:

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/

List of 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

3.7. Integration tests

3.8. Scalability requirements known and accounted for

3.9. Performance requirements known and accounted for

3.10. Static code analysis (backend)

3.11. Application events logging

3.12. OWASP Top 10 in Definition of Done

3.13. Authorization model defined

3.14. Continuous Integration

3.15. Continuous Delivery

3.16. Continuous Deployment

3.17. Documentation tracked in VCS

3.18. Documentation generated during CI

3.19. Parts of the documentation generated automatically

3.20. Automatic documentation of the executed tests

3.21. Documentation scope agreed

3.22. JS application framework

3.23. JS Build process

3.24. JS modules dependency management

3.25. JS Unit test

3.26. CSS builder

3.27. Static code analysis (JavaScript)

3.28. Truly RESTful interfaces

3.29. HTML validation

3.30. Code Reviews

3.31. Pair Programming

3.32. Test Driven Development

Introduction

When to apply

In practice

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