Testing on a Large Scale

CONTACT@ADAMFURMANEK.PL

HTTP://BLOG.ADAMFURMANEK.PL

FURMANEKADAM

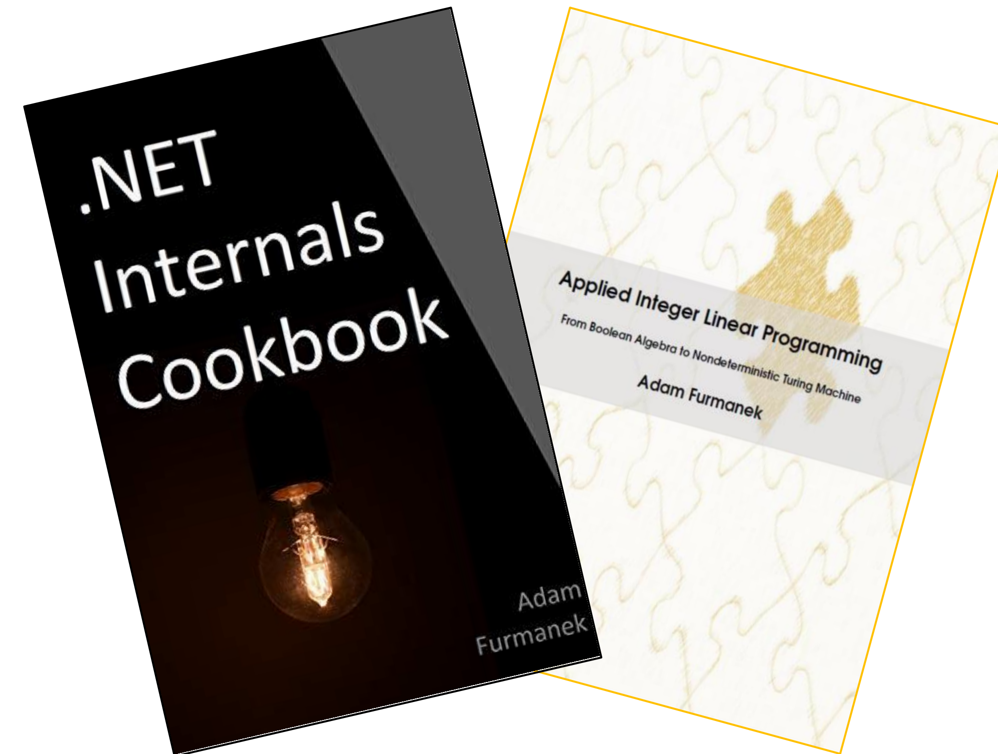
About me

Software Engineer, Blogger, Book Writer, Public Speaker.
Author of ***Applied Integer Linear Programming*** and ***.NET Internals Cookbook***.

http://blog.adamfurmanek.pl

contact@adamfurmanek.pl

furmanekadam



Agenda

Do we need tests?

How to do software to make testing easier.

How to keep tests maintainable.

How to test different domains.

Do we need tests?

Sure we do!

But we don’t test standard libraries. Why?

But we don’t have 100% code coverage. Why?

But we often decide to drop UI tests. Why?

But we mock external components, and run test suites that test everything but the production code. Why?

But we don’t test edge cases like faulty memory, broken network packets, or cosmic ray. Why?

But we don’t test „obvious” things like getters or setters. Why?

Reality

50%-80% of product's functionality is used rarely or never

- According to The Standish Group
- https://www.mountaingoatsoftware.com/blog/are-64-of-features-really-rarely-or-never-used

Without a proper process, cost and time can double over the lifecycle of a project

- https://www.cin.ufpe.br/~gmp/docs/papers/extreme_chaos2001.pdf

Knight Capital Group

- Repurposed bit flag caused a loss of $440M in 45 minutes
- https://en.wikipedia.org/wiki/Knight_Capital_Group

Mariner 1

- Due to incorrect equation caused by missing bar, the spacecraft had to be destroyed
- https://en.wikipedia.org/wiki/Mariner_1#Cause_of_the_malfunction

Belgium elections

- Cosmic ray increased numer of votes by 4096
- https://en.wikipedia.org/wiki/Electronic_voting_in_Belgium

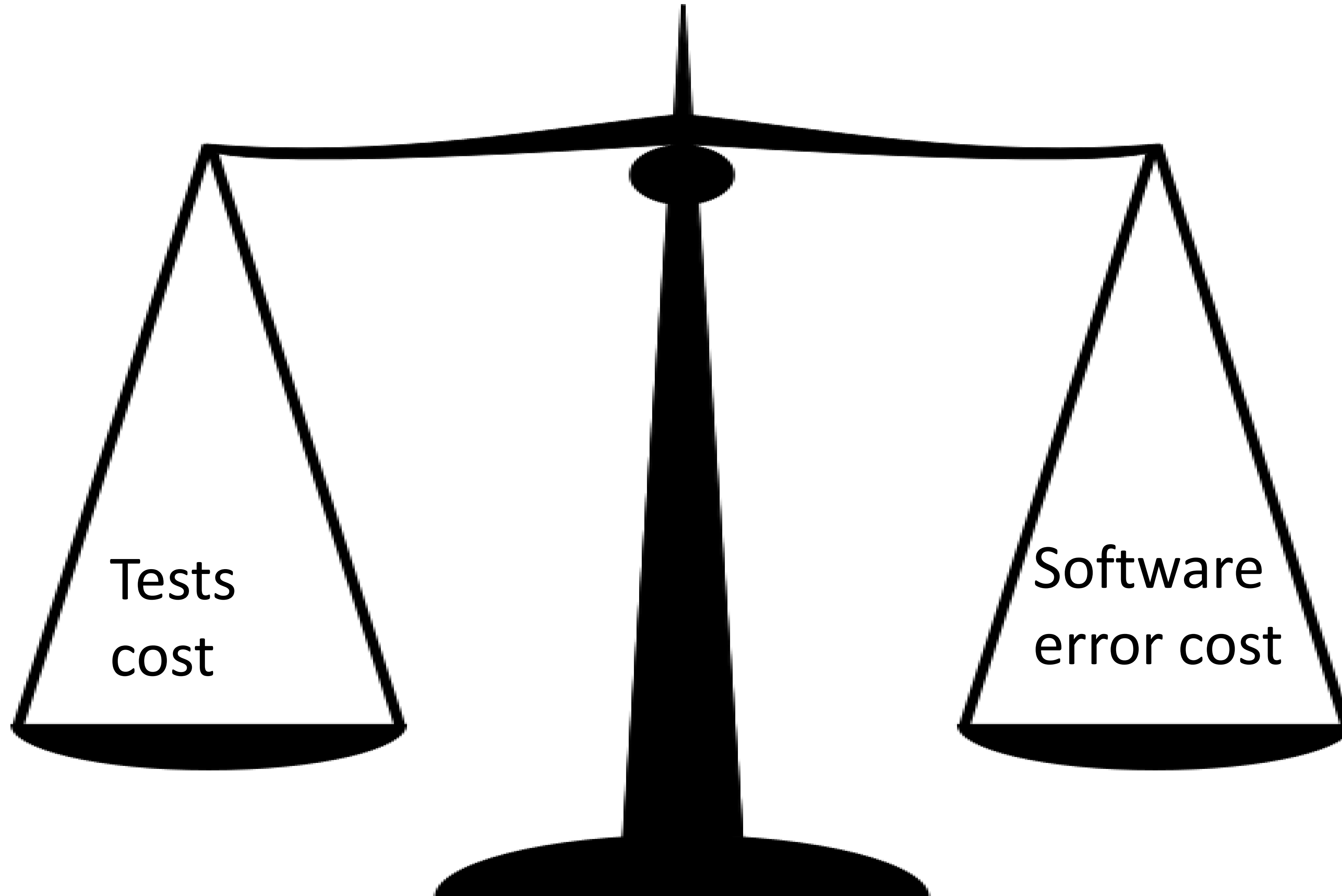
Therac-25

- At least 6 patients were given massive overdoses of radiation because of a race condition
- https://en.wikipedia.org/wiki/Therac-25#Problem_description

It’s all about the costs!

WE DON’T CREATE SOFTWARE JUST FOR THE SAKE OF CREATING IT!

PROBABILITY



A tale of two perfect programmers

PROGRAMMER NO 1

✓ Never makes mistakes.

✓ Writes code meeting all the business requirements.

? Runs tests after writing a single line of code.

PROGRAMMER NO 2

✓ Never makes mistakes.

✓ Writes code meeting all the business requirements.

? Never runs tests.

Who is „better”?

Real world

We are not perfect, but it's about **making trivial change as big as possible**

- It's a matter of experience, skills, and tools

We often don't need to run tests after writing small amount of code.

We're often not working on a critical piece of software.

We need to find a balance between „making sure we don't deliver a faulty software" and „taking the toll of delivering a faulty software".

We work in fast-paced environments. People come and go. They have different backgrounds, skills, priorities.

Environment changes. Libraries get security updates, hosts are deprecated and not supported, bugs are introduced. We need to test integration points between everything, even standard libraries.

Making things testable

3 aspects of developer experience

First aspect of DX
Be able to run locally and while offline

I want to **run all software locally** and with no Internet connectivity

- Databases, queues, service buses, authorization systems, external dependencies

I want to **run all the tests locally**

- Unit tests, integration tests, comparison tests, performance tests, hardware tests

I don't want to be forced to do it all locally

- Dev environments, pre-prod environments
- Data correctness? Data anonymity?
- Ongoing deployments?

I want to be able to **debug everything**

- My code, code of teammates, code of other teams in the company, external code, platform code

First aspect of DX
Be able to run locally and while offline

Use **docker containers**, **nix**, and **virtual machines**.

Configure your servers, so they can connect to your developer box. You can use **reverse tunnels** or **site-to-site VPN**.

Use **official emulators** where possible.

Have a load-balanced team environment that gets deployed with all production changes.

Synchronize data artifacts (databases, logs, clickstreams, etc) from production to development environment often. Make sure you don’t violate GDPR.

Deploy symbols for debugging. **Configure remote debuggers**.

Have a full IDE support.

Second aspect of DX
Automated feature branch testing

I don’t want to block my team’s environments to run tests.

I want to push changes to the feature branch and have it tested automatically.

I want to change in production-like environment. **No mocks**.

I want it to be fast.

Second aspect of DX
Automated feature branch testing

Have a build system capable of testing feature branches.

Each feature branch should **create a new environment** with **Infrastructure as Code**, **Test Containers**, **docker-compose**, or something similar.

All configuration should be applied automatically. Never hardcode things. Follow the Twelve-Factor Apps.

Feature branches should be tested independently.

Third aspect of DX
Automated deployment

I want to click „merge”, and then everything should be deployed to production.

All tests should be executed.

No manual steps should be needed.

Rollback should happen automatically if needed.

Features may be deployed gradually and on a schedule.

Third aspect of DX
Automated deployment

Get a proper CI/CD.

Have metrics in place. When metric spikes during the deployment, roll things back.

Use Infrastructure as Code.

Deploy features behind **feature toggles**.

All tests should be executed before and after the deployment.

Consider atomic updates, gradual deployments with load balancers, A/B tests.

Keep changes compatible. Don’t break your data. Be able to roll back database migrations.

Use **canary deployments** or **blue-green deployments**.

What beside tests?

Compilers

- They can verify correctness of our code
- They can prove our code works
- They are fast and automated
- The cost of meeting their requirements may be very high

Metrics

- They can quickly show all edge cases where our software malfunctions
- But they require the software deployment and the right traffic

Observability

- We need to integrate tools that can identify the root cause of issues
- We can integrate heuristics and AI approach for finding bugs

Customers

- They'll quickly identify most important bugs
- **They'll indicate which parts of the software are actually used**

Keeping tests maintainable

Why do we use mocks?

Imagine that it takes you one millisecond to run all tests with a new database.

The database matches the production one exactly.

The database is completely independent from other databases.

All tests run in parallel.

Databases is created on the fly, and is terminated after all tests are done.

All of that in one millisecond.

Would you ever mock such a database?

So why do we mock?

A production piece is slow.

We don’t have a license to run the production piece locally.

The production piece is expensive.

Tests are slow.

It’s technically impossible to run the production piece locally.

The production piece is a physical hardware, and we don’t have it with us.

Use Mockito-like mocks.

Implement your own mocks.

Use 3rd party mocks.

Use docker containers or virtual machines.

If you don’t run the production code, then you’re still runing a mock. No matter how clever the mock is.

How to write maintainable mocks

Don’t repeat a series of *Mockito.when* (or any *Arrange*) in an every single unit test.

- Start from a proper state, and replace just one part of it
- Builders are nice

Name test objects represeting proper business samples.

Don’t create invalid objects for tests.

Don’t generate data manually.

- Use things like *AutoFixture* instead
- Use property-based testing

Don’t repeat too much.

Don't repeat a series of *Mockito.when* (or any *Arrange*) in an every single unit test.

```java
@Test
public void test1() {
    // Arrange
    String id = "some-id";
    SomeState state = SomeState.builder()
            .id(id)
            .someInteger(1)
            .someString("Some string")
            .someDate(Instant.now())
            .status(Status.STATUS1)
            .build();
    // Act & Assert ...
}

@Test
public void test2() {
    // Arrange
    String id = "some-id";
    SomeState state = SomeState.builder()
            .id(id)
            .someInteger(1)
            .someString("Some string")
            .someDate(Instant.now())
            .status(Status.STATUS2)
            .build();
    // Act & Assert ...
}
```

```java
SomeState.SomeStateBuilder stateBuilder;

@BeforeEach
public void init() {
    String id = "some-id";
    stateBuilder = SomeState.builder()
            .id(id)
            .someInteger(1)
            .someString("Some string")
            .someDate(Instant.now())
            .status(Status.STATUS1);
}

@Test
public void test1() {
    // Arrange
    SomeState state = stateBuilder
            .status(Status.STATUS1)
            .build();
    // Act & Assert ...
}

@Test
public void test2() {
    // Arrange
    SomeState state = stateBuilder
            .status(Status.STATUS2)
            .build();
    // Act & Assert ...
}
```

Name test objects represeting proper business samples

```java
SomeState.SomeStateBuilder stateBuilder;

@BeforeEach
public void init() {
    String id = "some-id";
    stateBuilder = SomeState.builder()
            .id(id)
            .someInteger(1)
            .someString("Some string")
            .someDate(Instant.now())
            .status(Status.STATUS1);
}

@Test
public void test1() {
    // Arrange
    SomeState state = stateBuilder
            .status(Status.STATUS1)
            .build();
    // Act & Assert ...
}

@Test
public void test2() {
    // Arrange
    SomeState state = stateBuilder
            .status(Status.STATUS2)
            .build();
    // Act & Assert ...
}
```

```java
@Test
public void test1() {
    // Arrange
    SomeState state = StateRepository.stateInStatus1();
    // Act & Assert ...
}

@Test
public void test2() {
    // Arrange
    SomeState state = StateRepository.stateInStatus2();
    // Act & Assert ...
}
```

```java
@Test
public void test_readyToLaunch(){
  // Arrange
  SomeState state = StateRepository.readyToLaunch();
  // Act
}
```

Don’t generate data manually.

```
[Theory, CookbookAutoData]
public void Should_only_return_recipes_with_a_specific_ingredient(
    Cookbook sut,
    Ingredient ingredient)
{
    // When
    var recipes = sut.FindRecipies(ingredient);
    // Then
    Assert.True(recipes.All(r => r.Ingredients.Contains(ingredient)));
}
```

Don’t repeat too much.

```java
@Test
public void operation1_situation1_outcome1() {}

@Test
public void operation1_situation2_outcome2() {}

@Test
public void operation2_situation1_outcome1() {}

@Test
public void operation2_situation2_outcome2() {}
```

```java
@Nested
public class Operation1 {

    @Test
    public void situation1_outcome1() {}

    @Test
    public void situation2_outcome2() {}
}

@Nested
public class Operation2 {
    @Test
    public void situation1_outcome1() {}

    @Test
    public void situation2_outcome2() {}
}
```

Testing different domains

Mocks based on contracts

We need to mock production components (for whatever reason).

Mocks do not represent the actual components – whitebox and London (Mockist) TDD.

Don't use mocks if possible
- Follow Detroit TDD
- Run production code as much as possible
- Design by contract

Any non-production component, no matter how smart, is a mock.

Solution – **Liskov Substitution Principle**.

It's not (only) about the inheritance.

It's about the contract
- Preconditions
- Invariants
- Postconditions
- History principle

Contracts cannot be verified by the callee! **Only the caller can verify them**.

Contracts don't need to be written down in the code.

It’s easy to break contract with mocks

```
public <T, Client> T doWithOptionalBackupConnection(Client mainClient, Client backupClient, Function<Client, T> action){
    if(rand()%2 == 0){
        // First main, then backup
        return doInternal(mainClient, backupClient, action);
    }else {
        // First backup, then main
        return doInternal(backupClient, mainClient, action);
    }
}

private <T, Client> T doInternal(Client mainClient, Client backupClient, Function<Client, T> action){
    try {
        return action.apply(mainClient);
    } catch(Exception e){
        logger.log(e);
    }

    return action.apply(backupClient)
}
```

```
interface Client {
    String getData(String parameter);
}

public String doWork(Client mainClient, Client backupClient){
    String first = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("First"));
    String second = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("Second"));
    String third = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("Third"));
    return first + second + third;
}
```

It’s easy to break contract with mocks

```
interface Client {
    String getData(String parameter);
}

public String doWork(Client mainClient, Client backupClient){
    String first = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("First"));
    String second = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("Second"));
    String third = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("Third"));
    return first + second + third;
}
```

```
@Test
public void test(){
    assert(doWork(throwingClient, workingClient) == "Expected");
}
```

It’s easy to break contract with mocks

```
public <T, Client> T doWithOptionalBackupConnection(Client mainClient, Client backupClient, Function<Client, T> action){
    if(rand()%2 == 0){
      // First main, then backup
      return doInternal(mainClient, backupClient, action);
    }else {
      // First backup, then main
      return doInternal(backupClient, mainClient, action);
    }
}

private <T, Client> T doInternal(Client mainClient, Client backupClient, Function<Client, T> action){
    try {
        return action.apply(mainClient);
    } catch(Exception e){
        logger.log(e);
    }

    return action.apply(backupClient)
}
```

```
interface Client {
    CompletableFuture<String> getData(String parameter);
}

public String doWork(Client mainClient, Client backupClient){
    CompletableFuture<String> first = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("First"));
    CompletableFuture<String> second = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("Second"));
    CompletableFuture<String> third = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("Third"));
    return first.WaitForResult() + second.WaitForResult() + third.WaitForResult();
}
```

It’s easy to break contract with mocks

```
interface Client {
    CompletableFuture<String> getData(String parameter);
}

public String doWork(Client mainClient, Client backupClient){
    CompletableFuture<String> first = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("First"));
    CompletableFuture<String> second = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("Second"));
    CompletableFuture<String> third = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("Third"));
    return first.WaitForResult() + second.WaitForResult() + third.WaitForResult();
}
```

```
@Test
public void test(){
    assert(doWork(throwingClient, workingClient) == "Expected");
}
```

It’s easy to break contract with mocks

```
interface Client {
    String getData(String parameter);
}

public String doWork(Client mainClient, Client backupClient){
    String first = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("First"));
    String second = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("Second"));
    String third = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("Third"));
    return first + second + third;
}
```

```
@Test
public void test(){
    // Whatever set up needed for client
    prepareFailingConditions();

    assertThrows((client -> client.getData("First"))(clientUnderTest));
}
```

Infrastructure as Code (IaC)

We can scale environments easily.

We decrease the recovery time in case of a security breach.

We can test things independently.

We can test feature branches without blocking others.

Permissions? Allowlisting?
- This may require manual actions

Dynamic IP addresses? Domain names?
- IP may not be available. Some discovery is needed for DNS
- All configuration should be delivered via environment variables

Self-signed certificates?
- They may not work with domain names

Cost? License?
- Creating additional resources is expensive

Time?
- Deployment is slow!

Missing hardware?
- Cloud provider may not be able to provide enough resources

Rollbacks?
- It may be impossible to roll back in case of a filure

Testing distributed Extract-Transform-Load (ETL)

Unit tests work but they don't find the actual issues. **They don't test integration points**.

Pay attention to things easy to miss:

- DD.MM.YYYY vs MM.DD.YYYY
- Decimal separator
- Character encoding
- Null vs empty string vs any other empty value, including business markers like „N/A"
- Character escaping
- Differences in regular expressions

Test networking and marshalling in your heterogenous cluster.

Unit tests:

- To verify „it kinda works"
- To cover all impedance mismatch errors

End-to-end tests on a minimal amount of data:

- To verify caching and marshalling
- To see that it doesn't hang

Understand if you can delay your processing. If it breaks – can you catch up later on?

Testing Machine Learning (ML)

Testing ML is expensive. Consider reserving the hardware or going on-premise.

Unit testing is hard because it doesn't test anything.

Property-based testing is hard because of an inherent randomness.

Training is very sensitive to the **quality of the input data**.

You need to **run the pipeline end-to-end**, including all ETL transformations on the input data.

Use metrics to assess the quality of the trained model.

Make sure it actually works – run your inference.

Test with golden set to verify it works without segfaults:

- Have a well-prepared input data
- Train with a fixed seed
- Keep it short
- Maintaining the golden set is hard

Test with small production data to see that it actually works:

- Run all ETL transformations as a part of the test
- Verify metrics like AUROC
- Include inference as well
- This is slow

Is it bad if it breaks? How often do you need to retrain the model? Can you accept some outage?

User Interface (UI) and Hardware tests

Retry tests!

Use emulators, but remember that they are not perfect.

Record tests, so it’s easier to debug them. However, keep in mind that the recording slows things down, and may cause tests to fail.

Make them slow but stable – psychological aspect.

Test all possible variations.

Overprovision for stability.

Focus on correctness tests:

- Make them slow, automatically retried, but **stable**!
- They need to verify that the feature works
- Don’t ignore them if they fail

Have a seperation set of variation tests:

- For non-standard devices, resolutions, UI settings
- **They may fail much more often**, and it may be acceptable to ignore them
- Add an alarm when a test case fails for a prolonged period of time
- Review test cases, and look for patterns

Code hygiene

We can never prove the code will work. But we can be pretty sure.

Use compilers, linters, static analysis, type systems.

Think in terms of property testing. **Check „features", not „behawior"**.

Verify concurrency and distributed features with math models.

Check contracts.

Run mutation tests to figure out if you cover crucial parts.

Use strongly typed languages!

Use ever stronger strongly typed languages like Idris. Use Coq.

Use TLA+ for distributed and concurrent solutions.

Use mutation tests.

Learn SAT, ILP, CSP, LogP.

Use **high-level architectural patterns for organizing your code**

- Bulk Synchronous Parallel, Map-Reduce
- Actors, agents, structured concurrency

Load tests and Comparison tests

Take logs from production and replay them against your test environment.

- Redact secrets
- Anonymize Personally Identifiable Information

Sample logs from a wide time range – you want to get all your peak times.

Encode the state in your logs.

If you don't have enough logs – just play them over and over again.

Watch out for:

- Randomness
- Scheduled jobs changing state
- Non-significant differences (order of fields, timestamps etc)

Make sure you can recognize test requests from production ones.

Make sure you can recognize load test requests from production ones.

Log **source of requests** – do you know who sends them?

Think about **caches** – do you need to prefill them before running a load test?

Be able to stop your load tests at any time. Especially if you cause an actual issue in production!

Push to the left

Load tests are **slow and too late** in the pipeline.

We often don’t have enough data to run load tests locally.

Humans are often too slow to find issues with performance (like N+1 queries or table scans).

Developer databases are often different from the production ones.

Keep schemas and configuration in sync between developer environment and production.

Extract database execution plans with OpenTelemetry.

Verify if execution plans will scale well. Table scans are too slow for production.

Check time complexity.

Configuration

The concept of refactoring is very deceptive!

SQL queries will still work if we remove indexes.

ORMs will still work if we change eager loading to lazy loading.

Application will still work if we decrease memory just a little.

Database will still work if we downscale it.

We need to be able to verify if things are configured properly.

Test non-functional requirements.

Verify how the data was obtained.

Check if running configuration matches the desired one.

Compare logs and traces. Look for differences.

Have anomaly detection in place.

Documentation as Code

Keeping documentation and diagrams up to date is hard.

There is no way to verify if our code works as it's explained in the docs.

We don't need to know how it works. We need to see when it changes the way it works!

We can generate documentation automatically from code.

Have automated markdown files regenerated for each test when they run:

- Document inputs and outputs
- Put these files in your source control
- Commit them to the repository

Source control will indicate when there was a change in the behavior.

Use PlantUML, structurizr, or other tools that can visualize computer-generated documentation easily.

Synchronize your repository with Confluence and wikis.

A/B tests and feature toggles

Gather all data:

- Clickstreams, views, identifiers, input parameters, output parameters

Don't drop raw data!

Keep your ETL pipeline smooth.

Ideally: use one source of truth and one tech stack.

Remember that statistics is hard! You can't just look at numbers:

- p-value, z-score
- confidence interval

Do you need a testing environment at all?

- You need to keep everything backward compatible – APIs, data formats, table schemas, processing logic
- You need to keep everything forward compatible – don't drop unknown properties
- Rollback takes time and is not atomic
- Launching big features might be harder
- Consider using canary deployments and feature toggles instead

Rerun your A/B analysis periodically. Is it possible that the world has changed?

Metrics

Infrastructural:

- CPU, memory, GPU memory, number of threads, processes, disk usage

Runtime dependent:

- Heap usage, statistics of well-known GCs, number of promotions/demotions

Application dependent:

- Transactions, requests, business objects
- Use dimensions – country, day of week, type of customer

Aggregation is hard! Understand your percentiles and trimmed means.

Generate your dashboards automatically!

Configure alarms based on metrics. Emit 1 for failure and 0 for success to see errors on dashboards easily.

Have rollback in place when metrics change.

Run anomaly detection.

Review crucial business metrics manually with stakeholders on a regular basis.

Always emit metrics! Do not have metrics with missing datapoints.

Store metrics in logs in some way.

Backup tests

How do you test your backup procedures?

Who maintains the backup? Is it your cloud provider?

How fast can you restore the backup?

How fast can you recreate everything in case of a security breach?

Can you migrate to a different provider at all?

Make sure you test the backup and restore procedure.

Have some numbers showing how fast you can recover.

Keep backups locally, and in external storage provider.

Avoid vendor lock-in.

Rollback tests

Can you roll things back?

What detects whether you need to roll back or not?

What about API compatiblity?

What about data changes?

Do you have audits?

Can you roll back your IaC? Do you have enough machines?

What about caches? How do you roll them back?

Have metrics and alarms causing an automated rollback.

Maintain API compatiblity.

Deliver software in a way that it supports both old algorithm and a new one.

Use temporal tables or implement auditing manually.

Perform atomic updates where possible.

Summary

Code is a cost. So are tests. Don’t do them just for the sake of doing.

Don’t mock. No matter how clever the mock is, it’s still a mock!

Do not break contracts.

Maintain backward and forward compatibility.

Test your backups and rollback procedures.

Think about 3 aspects of DX.

Use IaC. Generate your documentation automatically.

Q&A

References

https://www.youtube.com/watch?v=f7i2wxQVffk – C4 models as Code

https://plantuml.com/ - PlantUML

https://www.idris-lang.org/ - Idris

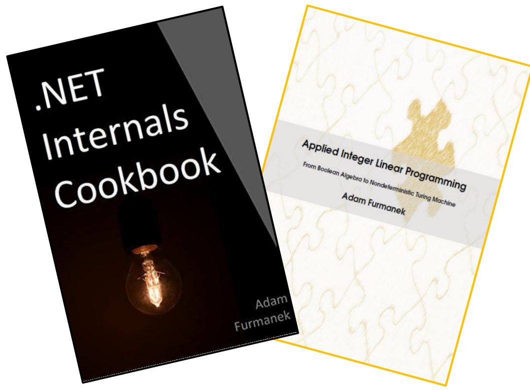
https://semaphoreci.com/blog/what-is-canary-deployment - Deployments

https://www.dynatrace.com/news/blog/what-is-observability-2/ - Observability

http://blog.adamfurmanek.pl/2022/05/07/types-and-programming-languages-part-10/ - TDD

http://blog.adamfurmanek.pl/2022/02/12/types-and-programming-languages-part-8/ - Testing

http://blog.adamfurmanek.pl/2021/10/23/types-and-programming-languages-part-7/ - DX



Random IT Utensils

IT, operating systems, maths, and more.

Thanks!

CONTACT@ADAMFURMANEK.PL

HTTP://BLOG.ADAMFURMANEK.PL

FURMANEKADAM