Software Project

Lecture 6

Wouter Swierstra

Last time

Automatically building software

Scripting with Bash

Today

Quality Assurance

The pitfalls of iterative software development

Constant deadlines lead to poor design and *technical debt*:

- We need this feature ready for the demo...
- Last day of the iteration, time to clear the task board...
- We'll refactor this code later...

If your development velocity is dropping, something might be wrong.

What can you do to prevent technical debt?

Software quality assurance techniques

- Establish a 'definition of done'
- Plan conservatively
- Software testing
- Continuous integration & test-driven development
- Code reviews & pair programming

What is the definition of done?

By now you should have completed your first iterations.

Every iteration, you tackle the stories from your sprint backlog.

But when is a story *done*?

Define what *done* means

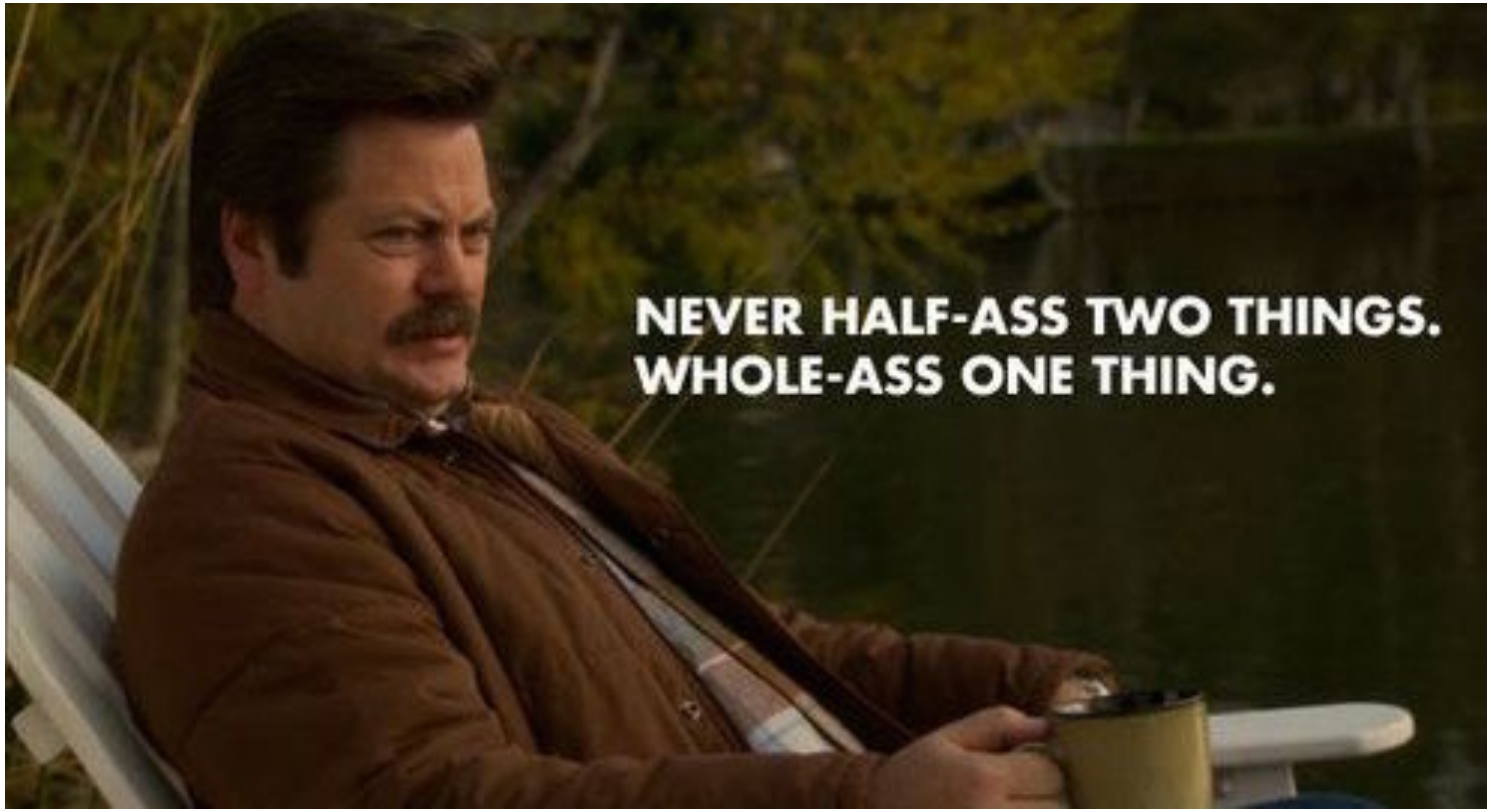
- The code has unit tests
- The code has been reviewed by another developer
- Automated tests have been created for the entire story
- Exploratory testing has been done by another team member
- Thorough user documentation and code comments
- Performance has been tested and is acceptable.

Homework

What is your definition of done?

Commit to this definition and stick to it.

Plan accordingly



Software testing

Introduction

Testing has its limitations:

Testing can only show the presence of errors, never their absence.
(Dijkstra's law – 1970)

But it is the most widely used techniques to monitor code quality.

Testing: definition

Software testing consists of the *dynamic* verification of the behaviour of a program on a *finite* set of test cases, suitably *selected* from the usually infinite executions domain, against the specified *expected* behaviour.

Testing is dynamic, not static

Testing means *running* a program, i.e., evaluating its *dynamic* behaviour.

- We need to run tests in order to make sure that all environment factors that may influence the execution are taken into account
- Results may depend on the system state
- They may be affected by memory unavailability, installed libraries, etc.

Testing: finite

Even most trivial programs have a very large ('semi-infinite') set of possible inputs and executions.

- Ideally, we would do *exhaustive testing* (test all possible executions; this is essentially a brute force correctness proof),
- ...but this is clearly infeasible except for isolated functions.

Testing: selecting tests

Since exhaustive testing is infeasible, we must make a *selection* of test cases such that we get as much confidence about the correctness of the system as possible in the given amount of time.

- The tests should *cover* as many classes of executions as possible.
- Finding a good selection criterion is very hard.

Testing: expected

In order to evaluate a test result we must know what the expected, i.e., 'correct', result is.

This is not as easy as it sounds, since the requirements are often underspecified, so we don’t know what the expected results are.

Sometimes a prototype is available to test against;

'Not crashing' is also a spec!

Pro-tip: Testing – part of the definition of done

During the planning session, specify when every story in the upcoming sprint backlog is done.

Translate these into (automated) tests.

At the end of the iteration, show that the automated test passes.

This provides evidence that the story is finished – but also grows your testsuite along with your codebase.

Target of the tests

We can test software at different levels:

- Unit testing
- Integration testing
- System/release testing

Unit testing

Unit testing is a method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine if they are fit for use.

Unit testing is an important part of Agile software practices.

Unit testing rules

- Before pushing to the central repository, you should run all unit tests. If anything breaks, fix it.
- Whenever you find a bug, add a test that tries to trigger the bug.
- Whenever you add a new feature, add a test to monitor it works.
- Once you have thousands of such tests, you can modify your system with confidence.

Unit Testing: xUnit

- Testing framework for programming language X.
- The original was SUnit for testing Smalltalk code.
- Then JUnit (Java), CppUnit (C++), PyUnit (Python), VBUnit (Visual Basic), and many other clones.

Unit testing

The general idea

- You write a bit of code that performs some test which either fails or succeeds.
- The testing framework runs all tests in succession.
- Any failures are reported back immediately.

Writing good unit tests

- Test one unit of code at a time – this way a failing test gives you precise information.
- Mock out external services and state;
- Don't test code that will not break, like getters and setters – focus on the difficult parts.

Tests are code too!

- Choose meaningful names (`testSuccesfulTransfer` rather than `test3`);
- Avoid repetition;
- Organize tests, just as you would organize other code.

Integration and system testing

Integration testing

After unit testing each individual component is 'correct'.

But do components work when composed with each other?

Integration testing tries to verify that the *entire* system works, rather than a single unit of code.

Often requires 'scripts' to run code implemented in different languages, start servers, etc.

System/release testing

System testing looks at the entire system, the software, hardware, network, etc.

'It works on my machine' – is really not good enough...

System testing demonstrates that the code is ready for release.

What to test?

Pareto-Zipf law – *Approximately 80 percent of defects come from 20 percent of modules.*

This law applies along all dimensions.

In other words, errors are *not* uniformly distributed over: modules, programmers in a team, statements in a program, etc.

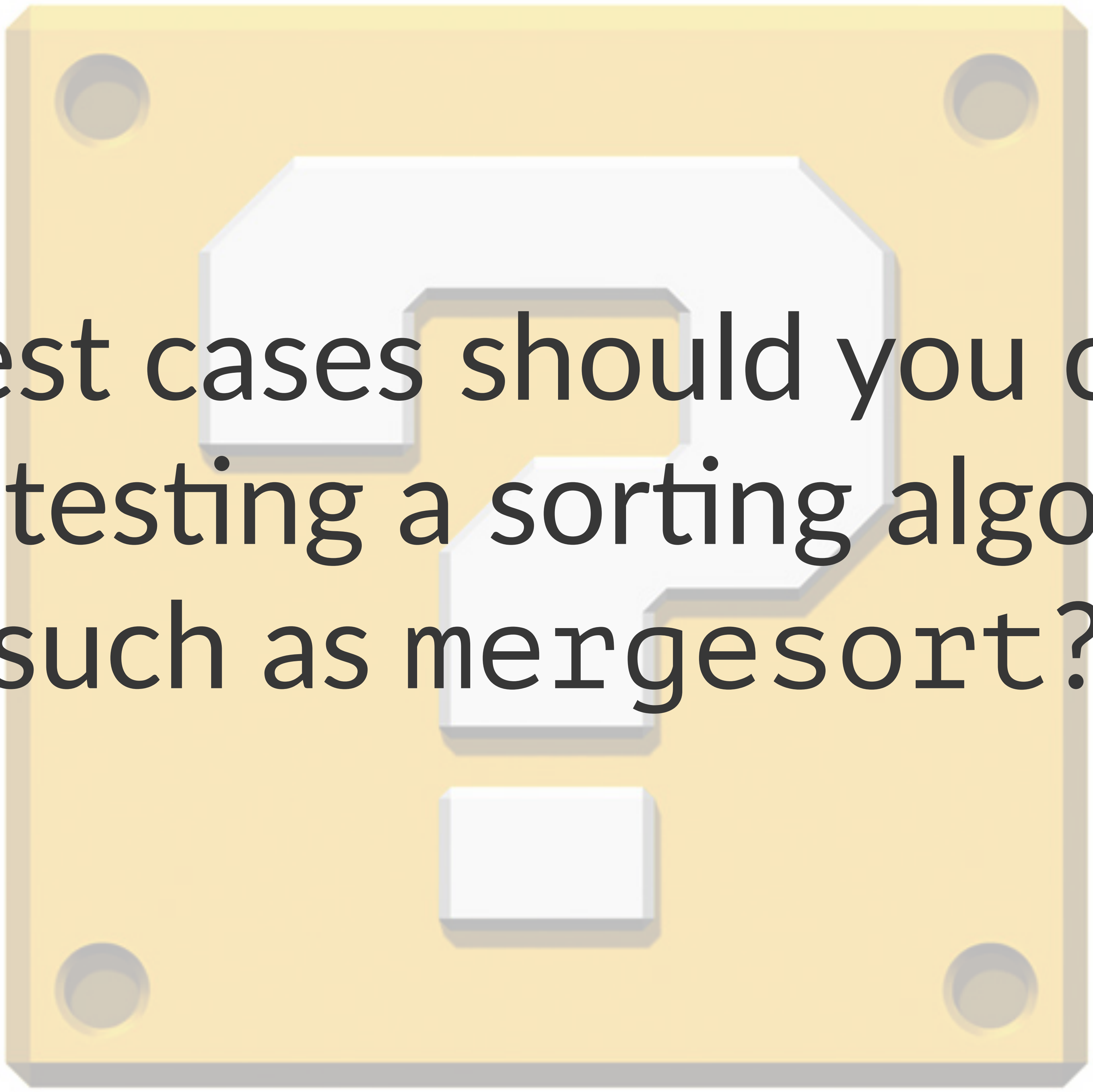
Focus your testing on the 20%.

What to test – partition based testing

We cannot test every possible input.

Identify classes of cases for which behaviour will be similar.

Try to cover as many of these classes as possible.



What test cases should you consider when testing a sorting algorithm, such as `mergesort`?

Example test cases – I

- empty sequence
- already sorted sequence
- reversely sorted sequence
- random sequence

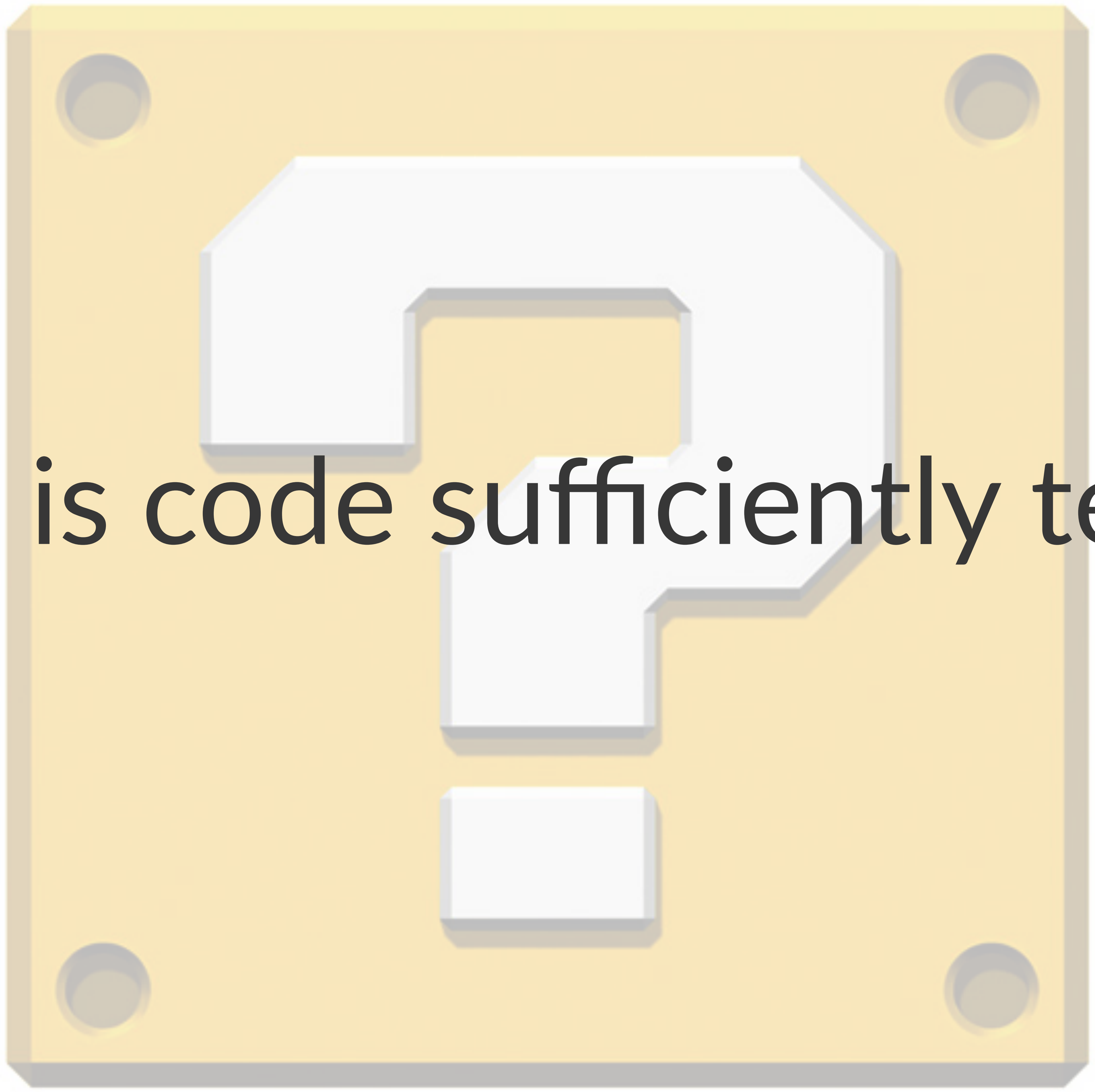
Example test cases – II

- very large sequence (indices over `65535` or even `2^32 - 1`)
- length of the input array is a power of two;
- length of the input array is a power of two + 1;
- length of the input array is a power of two - 1;
- combinations of the above

Don't just check the array is sorted: also that it contains the same values as the input. Or check it against a (slow) specification.

Effectiveness: Code Coverage

Code coverage measures how much of your code is actually tested.



When is code sufficiently tested?

Coverage Criteria

- Statement - all statements visited
- Branch - all branches tested
- Path - all possible combinations of branches are tested

Coverage

- A statement is *covered* if it is executed.
- Similar for sequences of statements.
- A test set covers a statement if at least one test executes that statement.

Statement coverage

- Statement coverage is weakest form of coverage.
- Still not easy to get 100% statement coverage.
- Error handling code, rarely occurring events, ...

```
if (param > 20) {
  error("This should never happen!");
}
```

Remember: focus on the important parts!

Branch coverage

- Conditions must evaluate to both true and false.

```
if (x < 10)
  print("a");
else
  print("b");
```

- What about dead code?

```
if (systemSupportsPrintf())
    printf("Hello World!)
else
    abort()
```

Branch coverage implies statement coverage, but not vice versa.

How to use code coverage

- Plan your new stories;
- Establish acceptance tests.
- Implement these stories and add tests;
- Run the tests and check the coverage.
- Add tests or refine the design and code until coverage is satisfactory.

What coverage percentages to aim for?

- Goals of your project.
- Cost of failure.
- Code coverage tool and metrics that you use.
- The part of code being tested.
- …

Bug and issue tracking

- Use your product backlog to track the status of known bugs, feature requests, etc.
- Bugs can *block* other bugs – take this into account when planning.
- Plan bugfixes along with other stories!

Continuous integration – best practices

- Automate the build
- Make the build self-testing
- Merge regularly
- Test in a production environment
- Share the results

Test driven development

After agreeing on the stories to tackle this iteration, agree on the acceptance tests.

Now start writing these tests.

Then make sure they pass.

Finally, refactor your code before the iteration is over.

Advantages of TDD

No code gets comitted without tests.

Starting with the tests, encourages *loose coupling*

Forces you to think about specifications first, rather than implementations.

Code review

If you're using GitHub, opening a pull request is a great way to do a *code review* with your fellow developers.

- Is the code clearly structured?
- Are there tests?
- Is the code documented?
- Does it adhere to your coding guidelines?

There are many code review checklists available online – use them!

Pair programming

Often, it can help to develop code together with a second team member:

- One developer writes the individual lines;
- The other monitors the design and tries to spot bugs.

Pair programming ensures code is shared between different developers from the start.

And may reduce the risk of integration...

Remaining presentation

Next up, you will present your quality assurance principles.