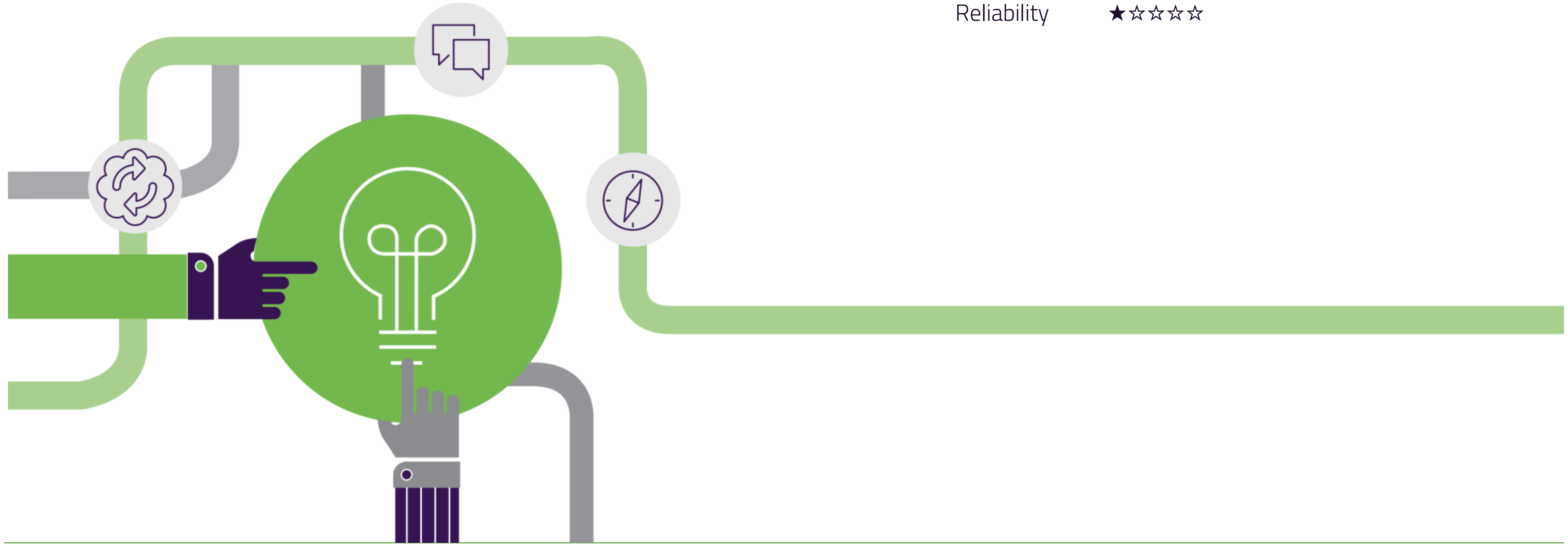
| | |
|---|---|
| Client: | <<client>> |
| Document no: | XYZ-001-D01-A |
| Author: | Byran Wills-Heath |

# [Sample] Quality Report: <<project>>

Illumination
Software Quality Assessment

## Overall Score

| | |
|---|---|
| Habitability | ★★★☆☆ |
| Scalability | ★★☆☆☆ |
| Reliability | ★☆☆☆☆ |



## Introduction

At Bluefruit, we define 'Quality' as the successful implementation of both Conceptual and Perceived Integrities, a concept developed in Mary Poppendeick's 'Lean Software Development: An Agile Toolkit' book.

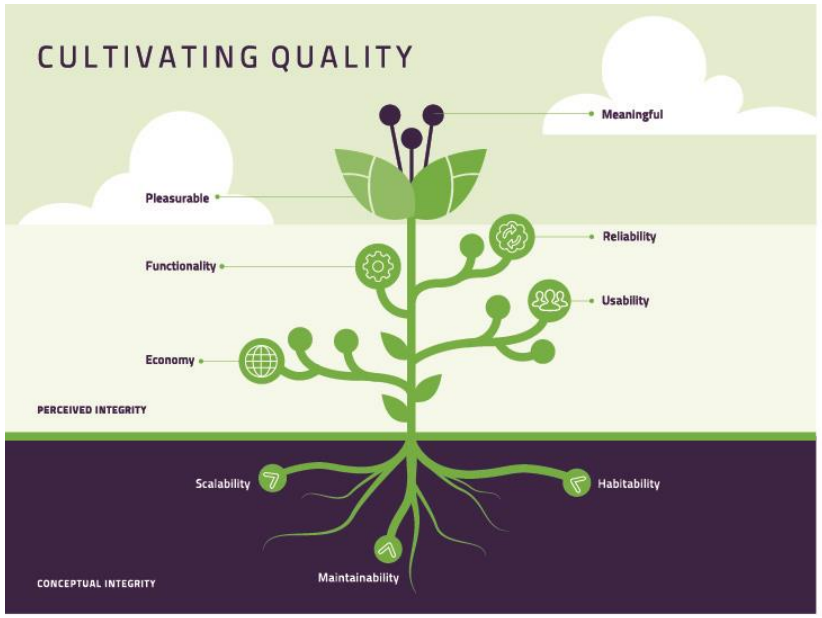
Conceptual Integrity describes the elements of the software that are not experienced by the end users. This includes:

- Habitability – how easy is it to live and work within the code
- Scalability – how easy is it to extend the functionality of the software
- Reliability – how easy is it to test and debug the code

Meanwhile, Perceived Integrity is what the end user experiences directly. It is what ultimately creates the 'intuitive feeling' for the user. In order to achieve this, the software must be:

- Functional - each feature should work as specified, and controls should perform as expected.
- Reliable - the software should not crash regularly or have errors that make it unreliable.
- Economical - the software should be efficient, and provide the simplest solutions to problems so that the user expends minimal effort for each task.
- Usable - each feature should be intuitive and easily understandable to the target client.
- Pleasurable - the software should be enjoyable to use, and give the user a sense of satisfaction and pleasure when using it.
- Meaningful - the software should be providing real value to the customer on every level.

These are key features of the User Experience Hierarchy of Needs, and we believe that true quality works its way up to successfully achieve both the Conceptual and Perceived Integrities. In fact, we like to visualise this as a plant, with the Conceptual Integrity (Habitability, Scalability and Reliability) being underneath the soil, and the Perceived Integrity being everything that we can see, including the leaves and flowers at the top.



To see our full discussion on this, please read our 'What is Quality?' blog post.

## Metrics

There are some "rules of thumb" that give us indications of how effectively source code achieves the above goals. Here is a list of some useful ones:

- Small files – small discrete components are easier to understand and easier to test. Keeping source code files less than 1000 lines long is an indicator that this has been achieved.
- Small functions – functions are much easier to comprehend if they can be viewed without scrolling, therefore we recommend that functions are less than 50 statements long.
- Simple functions – when functions have many different routes through them (too many branches) then it is very difficult to test all the possibilities and they are very difficult to comprehend, therefore we recommend an average cyclomatic complexity of less than 5 (maximum 20). Cyclomatic complexity refers to the number of routes through a function.
- Optimal commenting – commenting is a contentious subject. Comments can help, but code should be self-explanatory. Comments can also be a liability, i.e. they need maintaining. We recommend a target between 10%-30% of commenting. Commented-out code is also a poor practice; a source code management system should be used to archive redundant code.
- No "Magic Numbers" – raw numerals should rarely be used in source code because they lack meaning for a reader. Also, they are unmaintainable, because usually there are multiple instances of the number. Labels (enums, "#define"s or constants) should be used instead.
- Unit tests – automated testing vastly improves the reliability of code. Unit tests can be run on every build of an application to ensure defects haven't been injected. Ideally more than 80% of source code should be covered by unit tests.

# <<project>> Project Analysis

## Quantitative Results

We have analysed the <<project>> source code and it scores poorly on some of the above metrics. Please see the table below for details:

| Metric | Target | <<project>> Project |
|---|---|---|
| Maximum file size | <1000 lines | 610 |
| Average statements/function | 5-48 | 34 |
| Maximum statements/function | 100 | 85 |
| Average complexity | <5 | 3.54 |
| Maximum complexity | <20 | 38 |
| Average depth | <2.6 | 1.53 |
| Maximum depth | 4-8 | 7 |
| Commenting | 10-30 | 27.0 |
| Magic numbers | Low usage | High usage |
| Unit tests | >80% coverage | 0% |

## Qualitative Results

- There is a lack of documentation for the code. However the code is source controlled using Git which aids with working out how the code was developed.
- There is no sign of any unit tests. This questions the reliability of the code, but also adds to the lack of documentation.
- The code has not been structured with testability in mind.
- There is no sign of a test script, so it is difficult to evaluate the state of completion.

## Recommended Actions

- Restructure the code for testability and introduce unit tests.
- Create a full regression test plan and measure the current state of the project.
- Refactor the overly complex areas of the code.
- Remove magic numbers.
- Refactor functions with large complexity into smaller functions.

## Other Things We May Assess

### Code Smells – Addition/Alternative to Habitability, Scalability, Reliability

- Rigidity–The system is hard to change because every change forces many other changes to other parts of the system.
- Fragility–Changes cause the system to break in places that have no conceptual relationship to the part that was changed.
- Immobility–It is hard to disentangle the system into components that can be reused in other systems.
- Viscosity–Doing things right is harder that doing things wrong.
- Needless Complexity–The design contains infrastructure that adds no direct benefit.
- Needless Repetition–The design contains repeating structures that could be unified under a single abstraction.
- Opacity–It is hard to read and understand. It does not express its intent well.

## Other Things We May Assess

### Model

Technical documentation

Domain Model

Other UML Diagrams

Plain old white board diagrams

Specification document

Behaviour specification

| | What? | How? |
|---|---|---|
| Dynamic | Behavioural Spec | Code Documents |
| Static | Glossary/Domain Model | Technical Specification |

### Implementation

Unit tests

Vertical slicing

Coding guidelines & style

Single build script

### Test

System/Acceptance Tests

Regression testing

Testing is automated as much as possible

Bug tracking system

System documentation (system overview, user, support and operations documentation) and training materials have been validated

### Deployment

Release notes

Installation scripts/program

Product packaged for production

Able to de-install/recover system if you run into problems

De-install has been tested

Customer feedback system in place (e.g. Bug tracker or Change Request system)

In field updates – if required by market

Pre-production testing sandbox to validate system works

Go/no-go decision points in deployment process

### Configuration Management

Everything is stored using SCM/CM tool (including Source code, documentation, etc...)

### Environment

Tools required to build the system are documented

Required tools are stored in the SCM/CM

Old tools no longer supported by OS and may have bugs