Karelia University of Applied Sciences
BBA, Information Technology

The Effects of Testing in Software Development

Juha Airaksinen

Thesis, December 2023

www.karelia.fi

Karelia AMMATTIKORKEAKOULU	**OPINNÄYTETYÖ** **Joulukuu 2023** **Tietojenkäsittelyn koulutus** Tikkarinne 9 80200 JOENSUU +358 13 260 600

Tekijä
Juha Airaksinen

Nimeke
Testauksen vaikutukset ohjelmistokehityksessä

Tiivistelmä

Opinnäytteen tarkoituksena oli selvittää, vaikuttaako testaus kehitysprosessiin ja ohjelmistovirheiden määrään, miten tiimit voivat parantaa testaamisprosessia ja miten sovelluksia voidaan luoda testivetoisella kehityksellä.

Opinnäytetyössä analysoitiin lähdekirjallisuutta ja etsittiin vastauksia seuraaviin kysymyksiin: onko olemassa esimerkkejä testauksen vaikutuksesta tuotekehitykseen, mitä menetelmiä kehittäjät käyttävät testien suunnittelussa ja miten kehittäjät voivat määrittää testien tehokkuuden. Testivetoista menetelmää hyödynnettiin opinnäytteessä sovelluksen kehittämisessä ja menetelmän tehokkuutta arvioitiin projektista tehtyjen havaintojen avulla.

Testauksella oli myönteinen vaikutus ohjelmiston laatuun ja se vähensi virheitä, vaikkakin se hidasti kehitystä alussa. Pitkällä aikavälillä testauksella voi olla suotuisa vaikutus kehitysnopeuteen. Ohjelmistoprojektista saadut tulokset osoittivat, että testivetoisella kehityksellä oli myönteinen vaikutus kehitysprosessiin. Projekti ei ollut riittävän pitkäkestoinen, jotta testauksen pitkäaikaisista vaikutuksista olisi voinut tehdä havaintoja.

Kieli englanti	Sivuja 34 Liitteet Liitesivumäärä

Asiasanat
ohjelmiston testaus, ohjelmiston kehitys, testivetoinen kehitys

Karelia
UNIVERSITY OF APPLIED SCIENCES

THESIS
December 2023
Degree Programme in Business Information Technology

Tikkarinne 9
80200 JOENSUU
FINLAND
+ 358 13 260 600

Author
Juha Airaksinen

Title
The Effects of Testing in Software Development

Abstract

The purpose of this thesis was to examine the effects of testing on software development. What effects does testing have on the process of development and defect rates? How can teams improve the process of testing, and how they can utilise test driven development in application development?

The thesis analysed existing literature to answer a few questions - namely, have there been any examples of the impact that testing has had on product development, what methods do developers use to design tests, and how can they determine their effectiveness? A personal finance application was built using test-driven development (TDD). Test driven methodology was evaluated in terms of its efficacy, based on observations from the project.

Testing had a positive effect on software quality and reduced defects, albeit at a small cost to the initial development time. Over long-term testing may have a beneficial impact on development speed. Findings from the software project were that TDD had a positive impact on the development process. However, the project was not long lived enough to draw conclusive statements on the effects of testing over the long term. However, testing showed some promise by preventing a few regressions from occurring during the project.

Language	Pages 34
English	Appendices
	Pages of Appendices

Keywords
software testing, software development, test driven development

CONTENTS

1 Introduction

The aim of this thesis is to examine the subject of software testing. Why is testing important in software development? What problems does testing help to solve and how does it affect the process of building and maintaining software.

The thesis is also a way for the author to advance their knowledge in the subject of software testing. This information could be useful for the author's professional development, as the ability to test and create tests is commonly required in the industry.

The paper will establish information on how testing or the lack of testing can affect organizations, and users. Where does testing have positive or negative impacts and how to test software most effectively?

Having established how testing affects software development the next focus will be on how to maximize any benefits while minimizing the downsides? What measures can be used to track the quality of tests and how to design systems for easy testing? Why testing should be automated and how automation can be achieved?

As a conclusion to this paper, the gathered literature will be used to build an application. The application is a simple personal finance tracker with basic functionality to evaluate the effectiveness of testing in application development.

2 Testing

When deciding upon whether to do testing or not it is important to evaluate if testing is value positive to an organization. What problems does testing solve, and does testing provide any other benefits aside from simple validation of requirements?

2.1 Reducing defects

Software defects can be financially expensive to organizations. Nissan recalled nearly a million vehicles in 2014 due to a software defect affecting the passenger side airbag which might have not deployed in the case of a collision (Charette 2014). A few software defects affected a radiation therapy machine Therac-25, which caused the loss of life of 6 individuals, the incidents were partially caused by lack of proper software testing (Apgar & Prentice 2022).

Organizations have limited resources to effectively allocate resources for testing. Organizations need to evaluate where testing has the highest value potential. Using risk assessment based on the potential impact and the likelihood of an event happening resources can be allocated appropriately. (Felderer, Haisjackl, Pekar & Breu 2014.)

Test driven development (TDD) has been shown to significantly decrease the defect density of products between 40-90% when compared to similar projects that did not practice TDD. Defect density measures how many defects are found compared to lines of code. While defect density saw a significant decrease in the teams using TDD they also saw a slight increase in development time of 15-35%. (Nagappan, Maximilien, Bhat & Williams 2008.) Test driven development is also connected with increased code quality (Causevic, Sundmark & Punnekkat 2011, 337-346).

2.2 Effects on productivity

Martin Fowler (2018b) advocates for regular refactoring to maintain the internal quality of a project. Poor internal quality can slow the development process as it makes it harder to understand and modify existing code. Refactoring should be done with the support of testing to prevent accidental regressions. (Fowler 2018b, 4, 46.)

Continuous Integration (CI) can reduce the amount of time spent on merging changes to a shared mainline branch. CI can also help developers find issues in their code quicker as it enforces testing often. This can have a positive impact on software quality as it encourages behaviours that can reduce the number of bugs. (AWS 2023.)

A systemic review on the industrial adoption of TDD investigated 48 studies concerning test driven development. From the review findings increased development time was one of the largest factors in stopping adoption. Out of the studies nine had a negative experience with TDD, but adversely five reported a positive experience. (Causevic etc. 2011, 337-346.)

2.3 Testing approaches

Testing can be used to validate business requirements; these types of tests can be categorized as functional. Tests can also be used to verify that software meets some non-functional requirements like stability, performance, and security. Unit and integration tests can be categorized as functional, while stress and security tests can be categorized as non-functional. (Tricentis 2023b.)

Unit testing should form the foundation of an organizations testing strategy. When compared to end to end or integration tests, unit tests better isolate failures which means less code to search for the underlying issue. (Wacker 2015.)

The definition of unit is not explicitly defined and causes occasional discussion on semantics. Martin Fowler (2014) distinguishes unit tests into sociable and solitary tests. A solitary test only tests one unit that does not depend on other units, while a sociable test can rely on other units. In sociable tests the relied upon units are assumed to work correctly and only the selected unit is tested. (Fowler 2014.)

Unit tests are good at ensuring small parts of a codebase function well in isolation, but usually make no assurances that they work together. To test the

interoperation of units we can use integration tests. Integration tests refer to tests that use multiple code modules, but they can also use mocks in place of real modules. (Fowler 2018a.) Integration tests should form the second most substantial portion of testing (Wacker 2015).

End-to-end testing emulates user actions. This type of testing is good at giving information on what failures will look like for the end user. End-to-end tests are appropriate for some use cases, but should not be singularly relied upon, as they provide inaccurate information. These tests should be used in combination with other testing methodologies. (Wacker 2015)

2.4 Non-functional testing

Testing can be used to optimize an application for wanted user behaviour. A/B testing is used to compare two alternative versions of an application or a part of an application to measure whether a change improves key metrics. These metrics could be screentime, likelihood do some action on the application like pressing a button. (Gallo 2017.) A/B testing can also be used to optimize the content shown to the user, for instance for video thumbnails (Porter 2023; Urban, Sreenivasan & Kannan 2016).

Testing can be used to ensure that software works even in conditions where it is under heavier load. Performance tests are important to make sure that the software experience does not degrade in cases where load is higher than or at expected levels. Performance tests can also help developers find bottlenecks or other un-optimal areas in software. (Tricentis 2023a.)

3 Effective Testing

3.1 Test metrics

According to DORA Google's Research and Assessment team the five main metrics for development team performance are deployment frequency, lead time for changes, time to restore service, reliability and change failure rate. Change failure rate is a measurement of how often a change causes a degradation of a service which requires action from the team in the form of rollbacks, fixes, or patches. Lead time is a measure of how long before newly committed code in a production environment can be used. Deployment frequency is a measurement how often changes are released and time to restore service measures time taken to react to a failure in production. (Portman 2020.)

Testing can reduce the number of bugs that reach production. Although testing is not a guarantee that bugs will not happen. It is still a valuable tool in reducing the failure rate of an application. (Nassri 2019.)

During development developers should be able to get feedback quickly in less than five seconds. When unit testing is done right, it can allow for quick feedback loops. Short feedback loops allow for experimentation and iteration. (Shore 2021, 337.) Shore advocates for being able to run at least 100 tests per second (Shore 2021, 373).

Code coverage is a metric that is sometimes misused. Achieving full coverage is not difficult, because the coverage metric does not evaluate on how areas of code were tested only that they were tested. This can allow for substandard quality tests that increase coverage while not testing the code. Coverage reports can be helpful in discovering untested areas of code, but they should not be used as a tool for quality measurement. (Fowler 2012.)

3.2 Software design

Requiring testing pressures code bases to become decoupled to allow for easier testing. (Reese 2022). Decoupling improves overall code quality as it reduces the work required to make changes to existing systems, instead of needing to change multiple places a single or few changes are sufficient (Shore 2021).

Dependency inversion is important in decoupling software components. This improves testability of the application. Inversion allows dependencies to be more easily changed during testing. Dependency injection is a common application of dependency inversion principle. (Smith 2023, 21.)

For demonstration a frequent problem dependency injection can solve is database connection management. During testing full resources might not be available and the state should be reset between tests and runs. This makes tests more resilient to mistakes where a previous test affects the next. It allows for tests to be run in isolation.

The example is created using SQLite in Rust, but this could be any database or other external dependency. If using databases like PostgreSQL or MySQL a library like Testcontainers can be used to create a temporary database. Instead rusqlite allows databases to be run in memory which makes it well suited for this demonstration. A connection will create an SQLite file in the given directory, and a struct can also be created to represent a row from the database (picture 1).

```rust
// src/shared.rs
use rusqlite::Connection;
use std::env::current_dir;

pub struct ExampleRow {
    pub id: i32,
    pub data: String,
}

pub fn init_connection(db_name: &str) -> Connection {
    let path = current_dir().map(|x| x.join(db_name)).unwrap();
    Connection::open(path).unwrap()
}
```

Picture 1. Rust code containing shared functionality between implementations.

The ‘init_connection’ function only constructs a database connection for a given filename in the current runtime directory. For demonstration purposes error handling is ignored for now. (picture 1.)

An example of a naïve solution for database connection management is to create a connection within each function. Creating connections this way means that they are tightly couple to the filesystem and a specific filename. When the application is run, or tested the same database is being used which might pollute either with unexpected data. (picture 2.)

```rust
// src/local_init.rs
use crate::shared;
const DB_NAME: &str = "local.sqlite";

pub fn create_table() {
    let conn = shared::init_connection(DB_NAME);
    conn.execute(
        "CREATE TABLE IF NOT EXISTS example (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            data TEXT NOT NULL
        )",
        [],
    )
    .unwrap();
}

pub fn insert_data(data: &str) {
    let conn = crate::shared::init_connection(DB_NAME);
    conn.execute("INSERT INTO example (data) VALUES (?)", [data])
        .unwrap();
}

pub fn get_data() -> Vec<shared::ExampleRow> {
    let conn = shared::init_connection(DB_NAME);
    let mut stmt = conn.prepare("SELECT id, data FROM example").unwrap();
    let rows = stmt
        .query_map([], |row| {
            Ok(shared::ExampleRow {
                id: row.get(0)?,
                data: row.get(1)?,
            })
        })
        .unwrap();
    rows.map(|x| x.unwrap()).collect()
}
```

Picture 2. Naïve solution where connections are created inside the function.

A better approach for this solution would be to construct a single or multiple connections that can be reused in each function. In this case just reusing a single connection already allows for more testable code, although might not be effective use in real world applications that have to deal with concurrent requests. The solution is like the naïve solution. Only difference is that the connection is

passed through the function parameters instead of constructed by the function. (picture 3.)

```rust
// src/injection_init.rs
use rusqlite::Connection;
use crate::shared;

pub fn create_table(conn: &Connection) {
    conn.execute(
        "CREATE TABLE IF NOT EXISTS example (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            data TEXT NOT NULL
        )",
        [],
    )
    .unwrap();
}

pub fn insert_data(conn: &Connection, data: &str) {
    conn.execute(
        "INSERT INTO example (data) VALUES (?)",
        [data],
    )
    .unwrap();
}

pub fn get_data(conn: &Connection) -> Vec<shared::ExampleRow> {
    let mut stmt = conn.prepare("SELECT id, data FROM example").unwrap();
    let rows = stmt
        .query_map([], |row| {
            Ok(shared::ExampleRow {
                id: row.get(0)?,
                data: row.get(1)?,
            })
        })
        .unwrap();
    rows.map(|x| x.unwrap()).collect()
}
```

Picture 3. Dependency injection solution where connection is passed through the parameters

Both solutions are easy to use and there is no significant difference between them from the developer perspective. Although, as demonstrated earlier database connection creation inside each function leads to code duplication, and if the database needs to be change to something else developers would need to remember to change each location. The replication increases the cost of changes and is an avenue for bugs.

The application can be run with ‘cargo run’. After each run the number of entries in each database increases by three. On the initial run the expected number of entries should be three and on a second run six and so on. (picture 4 & 5)

```rust
// src/main.rs
mod injection_init;
mod local_init;
mod shared;

fn main() {
    local_use_sample();
    println!();
    injection_use_sample();
}

const DATA: [&str; 3] = ["foo", "bar", "baz"];

fn injection_use_sample() {
    let conn = shared::init_connection("injection.sqlite");
    injection_init::create_table(&conn);
    for data in DATA {
        injection_init::insert_data(&conn, data);
    }

    let rows = injection_init::get_data(&conn);

    println!("injection_use_sample");
    for row in rows {
        println!("id: {}, data: {}", row.id, row.data);
    }
}

fn local_use_sample() {
    local_init::create_table();
    for data in DATA {
        local_init::insert_data(data);
    }

    let rows = local_init::get_data();

    println!("local_use_sample");
    for row in rows {
        println!("id: {}, data: {}", row.id, row.data);
    }
}
```

Picture 4. Using implementations in the main file

```
local_use_sample
id: 1, data: foo
id: 2, data: bar
id: 3, data: baz

injection_use_sample
id: 1, data: foo
id: 2, data: ba
id: 3, data: baz
```

Picture 5. Results of ‘cargo run’ on the first execution.

When testing both the injection solution and naïve solution, both produce correct results on the first test run of the application. After the first test run the test will fail for the naïve solution as it uses the existing file on the system. The broken test could be fixed with added logic to the test. Counting the number of rows in the database initially and the subtracting the new count after insertions. Although this might work, over time it would create more and more entries to the database. Dependency injection solves this problem with less complexity. (picture 6.)

```rust
#[cfg(test)]
use rusqlite::Connection;

#[test]
fn injection_testing() {
    let conn = Connection::open_in_memory().unwrap();
    injection_init::create_table(&conn);
    for data in DATA {
        injection_init::insert_data(&conn, data);
    }

    let rows = injection_init::get_data(&conn);

    println!("injection_testing");
    // provides reliable results
    assert_eq!(rows.len(), 3);
}

#[test]
fn local_testing() {
    local_init::create_table();
    for data in DATA {
        local_init::insert_data(data);
    }

    let rows = local_init::get_data();

    println!("local_testing");

    // this fails after the first run
    // assuming that the database was not used previously
    assert_eq!(rows.len(), 3);
}
```

Picture 6. Simple test for both solutions

3.3 Test automation

Testing is an important part of continuous integration. Continuous integration ensures that the code in the mainline branch of the repository is ready to be built and delivered if needed. Changes should be made incrementally, and each

change is analysed and tested before it is integrated to the mainline. Automating the integration process helps to free up time of developers increasing productivity. (GitLab 2023.)

Developers can create automated steps that can be run before they commit changes with pre-commit hooks. Pre-commit hooks will execute before a commit and can do any kind of validation necessary. Common validation steps are running tests, ensuring code style is consistent with linters and validating that documentation is up to date. (Chacon & Straub 2014, chapter 8.3.)

Although git pre-commit hooks can run all the necessary validation steps, a continuous integration pipeline should still exist. Pre-commit hooks are easy to skip if needed. Also, pre-commit hooks need to be installed to the contributors' local copy of the repository. Pre-commit hooks are an effective way to prevent developers from creating accidental bad commits but are not a reliable in ensuring the mainline branch remains build ready.

A useful strategy in software development is to automatically test or run the application after each file change, in some cases the application can be reloaded in place retaining the previous state (Shore 2021). Being able to test applications after each file changes allows for a fast feedback cycle which is good for test driven development.

In test driven development tests are created before implementation, the tests help to drive the design. Initially this test should fail as the implementation is missing. New code is written to satisfy the created test condition until the test is no longer failing. After the test is working new tests can be added to drive the requirements forward or the existing code refactored to improve the implementation. (Codecademy 2023.)

4 Application development

To experiment and evaluate software testing concepts a personal finance tracker was developed. The application was meant to store data only locally with no external service connections. Also, the application should offer a graphical user interface and should be a standalone application that does not run in the browser.

For features the application needed to be able to list and search added transactions. It needed a way to create categories based on user input. The created categories could be used to tag a transaction with and later be used for filtering. A transaction could be categorized under no category, or any number of categories. The data could be also used to create simple aggregations like transactions in a month and transactions grouped by categories.

Outside of these functional and non-functional requirements it was important to also be able to demonstrate testing within the application development. Security and future expandability were not considered during the project but should be considerations in real world applications.

The final application can be found in GitHub in the following URL https://github.com/Zerkath/finance-app.

4.1 Prototyping

4.1.1 Technology selection

To avoid committing to poorly suited tools for the application different frameworks and languages were tried and experimented with. During experimentation test driven development was not used instead development speed was prioritized. Quality was a non-issue for the prototype as the application would be

thrown away after the prototyping phase. The prototype application can be found in GitHub in the following URL https://github.com/Zerkath/finance-app-prototype.

The approach used during the prototyping phase was like the spike solution methodology described by Shore (2021). Spike solution can be used when thinking ahead can be too difficult. The spike is used to figure out approaches to the problem. A spike should be short taking less than a day. During a spike TDD is not used and the built solution is discarded. Later it is rebuilt using TDD. (Shore 2021 355, 384.)

Initially Fyne a native application toolkit in Go was tried to develop the application but was dropped due to unfamiliarity with the ecosystem. Other options were investigated namely Electron and Tauri. Both Electron and Tauri allow the use of web technologies to develop standalone desktop applications. Tauri was chosen as Tauri can build smaller and more efficient applications when compared to Electron. (Tauri 2023). Tauri uses the Rust language for the backend, a performance-oriented programming language.

For local data storage SQLite was chosen because the data for the application was going to be highly structured. SQLite is highly suited for the requirements of the application as SQLite is an embedded SQL database engine. SQLite reads and writes data directly to the disc without requiring a separate process.

4.1.2 Implementation

Prototyping was important to verify that everything could be implemented with the selected technologies. Initial worries of the technologies were the capability of testing and limitations of SQLite. SQLite does not have as many advanced features as Postgres or MySQL and could potentially limit the design.

At the beginning the database design schema was implemented as storing and querying data was a core requirement. Once the database was verified to be capable of meeting the design goals other problem areas could be examined. Testing backend functionality was simple as the SQLite adaptor rusqlite in Rust allowed databases to be created in-memory. These in-memory databases would allow for easy experimentation of queries within tests. Creating coverage reports could be achieved with a dependency to tarpaulin a code coverage report tool.

The frontend portion of the application required more changes to configuration and additional dependencies when compared to the backend. The required dependencies were Vitest a testing framework and testing-library a tool to test web pages. Both dependencies were easy to install with the help of their documentation.

The prototype showed that everything could be achieved with the chosen technologies. Prototyping also helped to clarify the design, for instance the initial plan would have not been able to track income alongside expenses.

4.2 Development tooling

For the project setting up good tooling was important in enabling effective development. The tooling would need to do a few basic actions like re-testing the code after each file change and updating the graphical user interface on changes if running the application in development mode. Continuous integration pipeline was not setup for the project. Due to the lack of a CI, a pre-commit hook was an important addition to the project.

Tauri already had support for reloading the UI after file changes. Tauri achieved this with the use of Vite and something else for backend code. After changes are made to the backend code, the project is rebuilt, and the application is restarted which takes longer than changes on the frontend.

To minimize the friction when developing the backend, it was also important to be able to only test and run the backend portion of the application. With the addition of cargo-watch to the Rust dependencies, any Cargo command could be rerun on file changes. Cargo is the package manager for Rust, it is also the build tool and test runner.

To allow for the development of the application without starting the application fully. A way to run both cargo and npm commands in parallel was needed. The npm package ‘concurrently’ allowed for parallel execution of cargo and npm if the processes were started from npm. Any cargo command could be run from npm with the simple addition of ‘”cargo”: “cd src-tauri; cargo”’. With the command added cargo could be run with just ‘npm run cargo’. The scripts in the npm package.json could be combined together to achieve the wanted result. (picture 7.)

```
"scripts": {
  "test:frontend:watch": "vitest",
  "test:backend:watch": "cd src-tauri; cargo watch -x test",
  "test:watch": "concurrently --kill-others \"npm run test:frontend:watch\" \"npm run test:backend:watch\"",
  "test": "concurrently --kill-others-on-fail \"npm run test:backend\" \"npm run test:frontend\"",
  "lint": "prettier --check . && eslint ."
}
```

Picture 7. Custom npm scripts for running tests

Finally to run tests and compilation before each commit a simple git pre-commit hook could be created and included in '.git/hooks' folder. Pre-commit hooks are run before each commit, and the commit will fail if the script returns a non-zero status code. Checking formatting is faster than running tests and should be performed before to avoid wasting time.

```
#!/bin/bash
# Verify the formatting and linting are followed
npm run lint
# Runs both test:frontend & test:backend
npm run test
exit 0
```

Picture 8. Pre-commit hook written in bash running npm commands

These tools were enough to start the development of the application. Shore (2021) recommends against relying on an integrated development environment as it is not as flexible and migration to a script is inevitable (Shore 2021, 337).

4.3 Test driven development

The backend of the application was developed with test driven development. In the following section TDD will be demonstrated step by step the process of implementing categories for the application.

When developing the previously setup command 'npm run cargo test:backend:watch' could be run and it would run all tests after each change.

At the beginning no tests were implemented for the application. To start out with a test was created to verify that tests could fail, and everything was setup correctly. The test would be run and throw and error. (picture 9.)

```rust
#[cfg(test)]
mod tests {

    #[test]
    fn should_fail() -> Result<(), ()> {
        Err(())
    }
}
```

Picture 9. First test for backend

Once the test was failing as expected, real tests could be made. Although first some setting up was still required to be able to test database queries. In the previous section the topic of dependency injection was already covered, and it could be utilized in a comparable way for the application. For now, also updating the test to use a newly created function that should handle the action of inserting a row to the database in the future. The created function can be set to throw for until the next step. (picture 10.)

```rust
pub fn upsert_category(db: &Connection, label: &str) -> Result<(), rusqlite::Error> {

    // We want to fail for now as we don't have everything setup
    Err(rusqlite::Error::InvalidQuery)
}

#[cfg(test)]
mod tests {

    use super::*;

    fn init_db_in_memory() -> Result<Connection, rusqlite::Error> {
        let mut db = Connection::open_in_memory()?;
        Ok(db)
    }

    #[test]
    fn category_insert_should_succeed() -> Result<(), rusqlite::Error>{
        let conn = init_db_in_memory()?;
        upsert_category(&conn, "test")?;
        Ok(())
    }
}
```

Picture 10. Database creation in memory and insertion test

As a final setup step a database table is needed before inserting can be tested. The table will receive some constraints and rules. For example, the id and label are unique. The database is responsible for id creation. (picture 11.)

```rust
pub fn init_tables(db: &Connection) -> Result<(), rusqlite::Error> {
    db.execute_batch(
        "CREATE TABLE IF NOT EXISTS categories (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            label TEXT NOT NULL UNIQUE
        )"
    )?;
    Ok(())
}


fn init_db_in_memory() -> Result<Connection, rusqlite::Error> {
    let mut db = Connection::open_in_memory()?;
    init_tables(&mut db)?;
    Ok(db)
}
```

Picture 11. Table creation statement

The previously created insertion method could be updated to insert a value into the database. After the change tests are passing, but the implementation is missing some validation and restrictions. The validation requirements can be added to the test later, but there are not enough methods to create assertions yet. (picture 12.)

```rust
pub fn upsert_category(db: &Connection, label: &str) -> Result<(), rusqlite::Error> {
    db.execute(
        "INSERT OR IGNORE INTO categories (label) VALUES (:label)",
        named_params! {
            ":label": label
        }
    )?;
    Ok(())
}
```

Picture 12. Inserting a label into categories table

As part of the requirements for the applications API a method to query all labels was needed. The method would also be helpful in testing the implementations for categories. Allowing for simple assertions after insertions for instance. (picture 13.)

```rust
pub fn get_categories(db: &Connection) -> Result<Vec<String>, rusqlite::Error> {
    let mut stmt = db.prepare("SELECT label FROM categories")?;
    let mut rows = stmt.query([])?;
    let mut categories = Vec::new();
    while let Some(row) = rows.next()? {
        categories.push(row.get(0)?);
    }
    Ok(categories)
}

#[test]
fn category_query_should_return_nil_when_new() -> Result<(), rusqlite::Error>{
    let conn = init_db_in_memory()?;
    let list = get_categories(&conn)?;

    if list.len() > 0 {
        panic!("Expected empty list, got {:?}", list);
    } else {
        Ok(())
    }
}

#[test]
fn category_after_insert_should_be_readable() -> Result<(), rusqlite::Error>{
    let conn = init_db_in_memory()?;
    upsert_category(&conn, "test")?;
    let list = get_categories(&conn)?;

    if list.len() != 1 {
        panic!("Expected list with one item, got {:?}", list);
    } else {
        Ok(())
    }
}
```

Picture 13. Reading categories and testing empty and single category listing

The category insertions should be idempotent, as duplicate entries are not desired. Idempotency for insertions in this instance makes the method also resilient to concurrent requests. Although the application won't be handling concurrency and won't benefit from the resiliency. The application does benefit from preventing potential confusion when using the application. Duplicate entries might cause accidental actions on incorrect categories. For example, user removing a duplicate category, but it turned out to be the original. (picture 14.)

The idempotency test should fail as the insertion method does not normalize the input. Capitalization in the label string is not supported as it could lead to inconsistent views on the frontend. The string should be formatted by the frontend if needed. (picture 14.)

```rust
#[test]
fn category_insert_should_be_idempotent() -> Result<(), rusqlite::Error>{
    let conn = init_db_in_memory()?;
    upsert_category(&conn, "test")?;
    upsert_category(&conn, "test")?;
    let list = get_categories(&conn)?;

    if list.len() != 1 {
        panic!("Expected list with one item, got {:?}", list);
    } else {
        Ok(())
    }
}

#[test]
fn category_insert_should_ignore_casing() -> Result<(), rusqlite::Error>{
    let conn = init_db_in_memory()?;
    upsert_category(&conn, "test")?;
    upsert_category(&conn, "TEST")?;
    let list = get_categories(&conn)?;

    if list.len() != 1 {
        panic!("Expected list with one item, got {:?}", list);
    } else {
        Ok(())
    }
}
```

Picture 14. Idempotency test

The lower case idempotency test passed, but the varying casing test did not. Both tests should return the same result as casing should not be stored in the database for uniformity. The insertion statement can be updated to convert the given string value to lowercase. This means that the database will be only storing lowercase values and the idempotency issue will be resolved. After converting the value to lowercase the tests are passing. (picture 15.)

```rust
pub fn upsert_category(db: &Connection, label: &str) -> Result<(), rusqlite::Error> {
    db.execute(
        "INSERT OR IGNORE INTO categories (label) VALUES (:label)",
        named_params! {
            ":label": label.to_lowercase()
        }
    )?;
    Ok(())
}
```

Picture 15. Updating insert method to normalize it before storage

Another normalization that should be done is the removal of unnecessary whitespace. In this case surrounding whitespace characters are not desired as they can have the same effect as a simple duplicate string. The UI might not be capable of displaying the difference between a string with whitespace and could lead to similar confusion as regular duplicate entries. Also, the normalization should not affect whitespace between words. (picture 16.)

```rust
#[test]
fn category_insert_should_ignore_surrounding_whitespace() -> Result<(), rusqlite::Error>{
    let conn = init_db_in_memory()?;
    upsert_category(&conn, "foobar")?;
    upsert_category(&conn, " foobar")?;
    upsert_category(&conn, "foobar")?;
    upsert_category(&conn, " foobar")?;
    upsert_category(&conn, "foo bar")?;
    let list = get_categories(&conn)?;

    if list.len() != 2 {
        panic!("Expected list with two items, got {:?}", list);
    } else {
        Ok(())
    }
}
```

Picture 16. Testing varying surrounding whitespace cases

To remove surrounding whitespace the operation trim could be used. It only affects surrounding whitespace and does not remove the whitespace within the string. (picture 16.)

```rust
pub fn upsert_category(db: &Connection, label: &str) -> Result<(), rusqlite::Error> {
    db.execute(
        "INSERT OR IGNORE INTO categories (label) VALUES (:label)",
        named_params! {
            ":label": label.to_lowercase().trim()
        }
    )?;
    Ok(())
}
```

Picture 16. Updating the insertion method to trim whitespace surrounding the input

This concludes the step-by-step example of TDD. The methodology was used when constructing the rest of the backend functionality. This established a code base where each method was tested, and breaking changes were fast to discover. In some instances, creating tests felt unnecessary, but it is not

immediately apparent if the test helps to prevent some bad changes in the future. Most things should be tested for this reason. As estimating if a test is necessary or not is feasible to do reliably and it is better to go on the side of caution.

4.4 Preventing regressions

During application development the listing view only had support for basic pagination with no filtering capability. The functionality to search and filter by category was added later. Previously created tests helped to prevent a few regressions from happening.

At the beginning the function signature only the following parameters ‘page_size’ and ‘current_page’. The function also needed the addition of the following parameters ‘search’ and ‘selected_categories’. The search was a string, and the category was a list of category ids. When both were left empty, the behaviour should not change. Tests required the new function parameters to be added manually as empty.

Adding the parameters to the function did not break existing functionality. Adding search was simple. Search needed to find either name or description in a row to return it. This could be done with the SQL keyword LIKE. The named parameter search was trimmed of surrounding whitespace and wrapped in % to match within a string value. So given a search of ‘t’ if the database contains a row where name or description contains ‘t’ anywhere the value would be returned. For example, ‘rent’, ‘therapy’ and ‘taxi’ are valid returns. In SQLite the LIKE operator is case-insensitive and will return correctly regardless of if the value stored is in upper- or lowercase. (picture 17.)

```
1 SELECT COUNT(*)
2 FROM transactions as t
3 WHERE (name LIKE (:search) OR description LIKE (:search)
```

Picture 17. Selection statement for searches

The selection statement for the category was more complex. Testing helped to prevent a regression in this instance. As previously stated, the category list should return everything when left empty. The initial SQLite query selection would use an id IN statement in a subquery to select the correct elements. The subquery would select from ‘transaction_category’ table, to get all transaction ids where category_id was within the named parameter ids. (picture 18.)

```
SELECT COUNT(*)
FROM transactions as t
WHERE t.id IN (
  SELECT transaction_id
  FROM transaction_categories
  WHERE category_id IN (:ids)
)
AND (name LIKE (:search) OR description LIKE (:search))
```

Picture 18. Initial selection statement

This selection statement worked correctly in cases where a list of ids was given, but in cases where the list was empty a regression happened. The selection would not return entries. Due to the extensive testing in the area with empty arrays for ids parameter, the issue was easy identify and resolve with an adjusted selection statement. If the length of the given list was empty the subquery would be skipped. (picture 19.)

```
SELECT COUNT(*)
FROM transactions as t
WHERE (
  LENGTH(:ids) = 0 OR
  t.id IN (
    SELECT transaction_id
    FROM transaction_categories
    WHERE category_id IN (:ids)
  )
)
AND (name LIKE (:search) OR description LIKE (:search))
```

Picture 19. Improved selection with no regressions

4.5 Testing frontend components

In the application frontend components did not have a lot of logic to test. Also, instead of using TDD when developing the frontend, the development server was used. The development server provided a fast feedback loop when making changes.

The most complex component got tested with Vitest and testing-library. The component was responsible for allowing the editing and removal of a category. Initial state of the component would display two buttons and a disabled text field. One of the buttons would enable editing, the other would delete the entry. (picture 20.)

```
1  const getScreen = () => {
2    const screen = render(CategoryComponent, { label: 'Test', categoryId: 10 });
3    const editOrCancelButton = screen.getByTestId('category-modify-action');
4    const saveOrDeleteButton = screen.getByTestId('category-apply-action');
5    const textField = screen.getByTestId('category-input');
6
7    return { editOrCancelButton, saveOrDeleteButton, textField };
8  };
9
10
11 test(
12   'Two buttons & textfield should be available, edit, delete & textfield should be disabled',
13   async () => {
14     const { editOrCancelButton, saveOrDeleteButton, textField } = getScreen();
15     expect(editOrCancelButton.textContent).toBe('Edit');
16     expect(saveOrDeleteButton.textContent).toBe('Delete');
17     expect(textField.disabled).toBe(true);
18     expect(textField.value).toBe('Test');
19 });
```

Picture 20. Asserting initial state of the component

If editing was chosen the text field would become editable, edit button would become cancel and delete would become save. When editing an entry in the text field on cancel should revert to the initial value. Saving would store the current value and change the state back to non-editing. (picture 21.)

```
test(
  'After clicking edit, should have cancel and save available',
  async () => {
    const { editOrCancelButton, saveOrDeleteButton, textField } = getScreen();

    await fireEvent.click(editOrCancelButton);

    expect(editOrCancelButton.textContent).toBe('Cancel');
    expect(saveOrDeleteButton.textContent).toBe('Save');
    expect(textField.disabled).toBe(false);
});

test(
  'Editing then cancelling should revert back to original value',
  async () => {
    const { editOrCancelButton, saveOrDeleteButton, textField } = getScreen();

    expect(textField.disabled).toBe(true);
    expect(textField.value).toBe('Test');

    await fireEvent.click(editOrCancelButton);
    await fireEvent.input(textField, { target: { value: 'Changed' } });

    expect(textField.disabled).toBe(false);

    expect(
      textField.value,
      'Inputting new value should affect the value stored by this field'
    ).toBe('Changed');

    await fireEvent.click(editOrCancelButton);
    expect(editOrCancelButton.textContent).toBe('Edit');
    expect(saveOrDeleteButton.textContent).toBe('Delete');
    expect(textField.disabled).toBe(true);
    expect(textField.value).toBe('Test');
});
```

Picture 21. Asserting behaviour of edit and cancel

Testing components in the previously described way is heavily coupled to implementation details. The tests are fragile and could break with minor changes to the components, even though the functionality would remain the same. For this reason, end-to-end tests would potentially be a better fit for frontend testing. Also, components could be tested for accessibility as assertions for those are more resilient to changes.

5 Conclusions

5.1 Results

The goal of this thesis was to evaluate the effectiveness of testing in software development. What are some of the main benefits of testing, how can testing be done more effectively, and how to apply testing in a software project?

According to the literature reviewed, software testing has an overall positive impact in software development. It is useful in preventing defects, although testing is not a guarantee that errors will not happen. Application development is initially slowed by testing, but testing might have a stabilizing effect on development speed over the long term. Reducing the time, it takes to make changes to software later in a project.

Before adopting testing into a project, teams should evaluate whether their priorities overlap with testing benefits. If a team is working on a software solution that might not have a market, the priority on developing the application quickly might be more important than its maintainability and quality. If a product fails, it doesn’t need to be maintained. For proven products testing is more important.

Testing solutions where cost of failure is high is important. Software that must work with high-risk industries like medical, financial and robotics industries could benefit from testing more as a risk mitigation strategy.

Test driven development is widely advocated for its benefits. Based on the reviewed literature, TDD can have a substantial impact on software development in a positive way. TDD was used in this thesis to build a standalone application and many positive benefits could already be observed. TDD helped to prevent regressions during the project, and it also made it easier to develop features, but it did increase the workload due the addition of needing to write tests.

5.2 Limitations

The software project was not long lived enough to make conclusive statements on the effectiveness of TDD over the long-term. A longer software project would be required to evaluate the effectiveness of testing more thoroughly. Although regression prevention and ease the of refactoring with tests showed that long term benefits are likely.

The project also had less focus on testing the frontend portion of the application. Strategies for effective UI testing were not covered enough, and the paper might have limited usefulness to frontend developers.

5.3 Personal development of the author

The paper advanced the knowledge of the author in many ways. It helped to introduce the author to topics outside of their current work role. The author did not have previous experience in using test driven development to develop applications, mostly creating tests after implementation was in place.

The application development project also exposed the author to technologies they had not used previously but could be helpful in their career. Namely the Rust language which could potentially be helpful in their current role. Also, during project, the author got to work with more fundamental concepts like planning and creating SQL queries without the help of object relational mappings.

The author has already started to adapt their behaviour in developing software. Creating a suit of tests when implementing new features to applications. Also using testing to benchmark solutions to verify a change improves application speed instead of working off assumptions.

5.4 Future research

In this paper some topics were covered only in brief detail. Some of the topics could be covered in further detail, namely test-driven development effects in longer projects and frontend testing.

Accessibility testing is an important topic that was not covered by this paper and could be researched in the future. The topic of accessibility is broad and could be helpful in both web development and game development. Accessibility allows more people to engage with built software solutions.

Benchmarking is an important topic as more efficient software can help to can reduce the amount of energy needed to provide services, but more importantly efficient software could allow older devices to remain relevant longer, in turn helping to reduce e-waste.

References

Apgar, C. Prentice, R. 2022. Therac-25. https://ethicsunwrapped.utexas.edu/wp-content/uploads/2022/10/Therac-25-1.pdf. 8.12.2023

AWS. 2023. What is Continuous Integration?. https://aws.amazon.com/devops/continuous-integration/. 21.11.2023

Causevic, A. Sundmark, D. Punnekkat, S. 2011. Factors Limiting Industrial Adoption of Test Driven Development: A Systematic Review. Germany. Fourth IEEE International Conference on Software Testing, Verification and Validation.

Codecademy. 2023. Red, Green, Refactor. https://www.codecademy.com/article/tdd-red-green-refactor. 22.11.2023

Chacon, S & Straub, B. 2014. Pro Git 2nd edition. https://git-scm.com/book/en/v2. 22.11.2023

Charette, R. 2014. Nissan Recalls Nearly 1 million Cars for Air Bag Software Fix. https://spectrum.ieee.org/nissan-recalls-nearly-1-million-cars-for-airbag-software-fix. 22.11.2023

Felderer, M. Haisjackl, C. Pekar, V & Breu, R. 2014. A Risk Assessment Framework for Software Testing. https://www.researchgate.net/publication/282182753_A_Risk_Assessment_Framework_for_Software_Testing. 23.11.2023

Fowler, M. 2018a. IntegrationTest. https://martinfowler.com/bliki/IntegrationTest.html. 15.11.2023

Fowler, M. 2018b. Refactoring: Improving the Design of Existing Code (2nd Edition). USA. Addison-Wesley. 8.5.2023.

Fowler, M. 2012. TestCoverage. https://martinfowler.com/bliki/TestCoverage.html. 15.11.2023

Fowler, M. 2014. UnitTest. https://martinfowler.com/bliki/UnitTest.html. 15.11.2023

Gallo, A. 2017. A Refresher on A/B testing. https://hbr.org/2017/06/a-refresher-on-ab-testing. 15.11.2023

GitLab. 2023. What is CI/CD?. https://about.gitlab.com/topics/ci-cd/. 22.11.2023

Nagappan, N. Bhat, T. Maximilien, M. Williams, L. 2008. Realizing the quality of improvement through test driven development: results and experiences of four industrial teams. https://urly.fi/3jXb. 14.11.2023

Nassri, A. 2019. Release with confidence: How testing and CI/CD can keep bugs out of production. https://cloud.google.com/blog/products/application-development/release-with-confidence-how-testing-and-cicd-can-keep-bugs-out-of-production. 15.11.2023

Porter, J. 2023. Youtube is making it easier for creators to choose that perfect thumbnail. https://www.theverge.com/2023/6/23/23771045/youtube-test-and-compare-a-b-testing-thumbnails-feature. 15.11.2023

Portman, D. 2020. Are you an Elite DevOps performer? Find out with the Four Keys Project. https://cloud.google.com/blog/products/devops-sre/using-the-four-keys-to-measure-your-devops-performance. 15.11.2023

Reese, J. 2022. Unit testing best practices with .NET Core and .NET Standard. https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices. 16.11.2023

Shore, J. 2021. The Art of Agile Development Second Edition. USA. O'Reilly Media

Smith, A. 2023. Architecting Modern Web Applications with ASP.NET Core and Microsoft Azure. https://urly.fi/3jKV. 16.11.2023

Tauri. 2023. Benchmarks. https://tauri.app/v1/references/benchmarks. 26.11.2023

Tricentis. 2023a. Performance testing, best practices, metrics & more. https://www.tricentis.com/learn/performance-testing. 14.11.2023

Tricentis. 2023b. Software testing. https://www.tricentis.com/learn/software-testing. 15.11.2023

Urban, S. Sreenivasan, R. Kannan, V. 2016. It's All A/Bout Testing: The Netflix Experimentation Platform. https://netflixtechblog.com/its-all-a-bout-testing-the-netflix-experimentation-platform-4e1ca458c15. 15.11.2023

Wacker, M. 2015. Just Say no More End-to-End Tests. https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html. 15.11.2023