Test Case Selection Based on Code Changes

Filip Normann

Institutionen för informationsteknologi
Department of Information Technology
Abstract

Test Case Selection Based on Code Changes

Filip Normann

As a software project grows, continuous integration (CI) requires more and more resources and the lag for the developer to get feedback from a code commit gets larger and larger. This thesis is about researching and implementing a way to reduce the end to end time of the CI flow for an Ericsson inhouse product by reducing the time to run regression tests. The inhouse product is a large java framework for testing radio equipment. Research was done to figure out how to implement test case selection for the project. The research was concluded with the decision to make a test case selection algorithm written in Groovy. Leveraging code dependencies and the delta between the committed code and its parent commit. The designed solution shows the potential to reduce time and resources spent testing committed code. Saving both computing power and reducing time for test result feedback to get back to developers, increasing productivity for the development team.
Table of Contents

Contents

Abstract............................................................................................................................................... 3
Table of Contents.................................................................................................................................. 5
Introduction........................................................................................................................................... 7
Problem description ............................................................................................................................... 9
Core concepts......................................................................................................................................... 11
    Continuous Integration (CI).................................................................................................................. 11
    Gerrit .................................................................................................................................................... 11
    Gerrit review ....................................................................................................................................... 11
    Jenkins .................................................................................................................................................. 11
    Surefire ................................................................................................................................................ 12
    Checksum ............................................................................................................................................ 12
    Test case optimization ....................................................................................................................... 13
    Problematization of test case selection ............................................................................................ 13
Research Phase ...................................................................................................................................... 16
    Comparing solutions .......................................................................................................................... 16
    Self-made tool..................................................................................................................................... 16
Evaluating Alternatives ......................................................................................................................... 17
    Starts and Ekstazi .............................................................................................................................. 17
    Alternate way forward ....................................................................................................................... 17
    Inspiration from STARTS .................................................................................................................. 17
    Crossroads ......................................................................................................................................... 18
    Decision .............................................................................................................................................. 19
Final design ............................................................................................................................................ 20
Results ................................................................................................................................................... 22
Discussion .............................................................................................................................................. 23
    Test impact analysis in visual studio ............................................................................................... 24
    Taming Google-Scale Continuous Testing ......................................................................................... 24
    Test Load Balancer (TLB) .................................................................................................................. 24
    Infinitest .............................................................................................................................................. 24
Future work ........................................................................................................................................... 24
Reflection on thesis work ...................................................................................................................... 26
    Had I started over with what I know today ...................................................................................... 26
    Final words ....................................................................................................................................... 26
Introduction

Continuous integration (CI) is a process in software development used to integrate changes often to detect problems early in the development process. In practice continuous integration is merging copies or branches of a software project to a main copy/branch several times a day as small, incremental changes.

Applying continuous integration means that commits to the code base are made frequently, which sets a new requirement for the software project applying it. Namely that the project needs to make it possible to merge often and conform to the fast speed at which developers integrate their code. To meet this requirement a new tool/concept called a continuous integration flow (CI flow) has emerged in the software industry. A CI flow is an automated tool which performs different types of checking. The checking is typically done on code changes, commits, by building the project, running different types of tests and statically analysing the code by a set of syntactic rules. A lot of tools have emerged to fill the needs of a CI flow, ranging from out of the box solutions to more sophisticated flows, allowing for complete configurability for the person or people responsible for creating and maintaining the flow.

The automation of verifying code changes with testing and code analysis enables this new type of workflow for developers which compares to previous solutions where changes were typically big updates with months in between, using manual testing in some cases. With a CI flow the developer is automatically fed confidence that he or she hasn’t changed or broken any legacy functionality (old functionality that needs to be kept) when committing code, assuming the testing is thorough enough since a CI flow is only ever as good as its tests. Continuous integration also means that the typical commit is small, meaning an increased amount of incremental updates to the code base compared to projects not implementing CI. Incremental updates completely remove the cases of very large and extensive updates, which are more likely to cause issues and can cause big impact for customers. Applying continuous integration generally reduces the time to market – the time for a change to come in to effect. All of which serves as huge incentives for software projects and companies to implement continuous integration.

There is extensive effort to improve today’s CI flows within companies and on the many open source platforms which specialize in CI flows. One of the biggest requirements of a CI flow is availability; a CI flow can’t afford much downtime as this will cause time loss for the end users of the flow – the developers. To have redundancy in the case of downtime can be difficult for certain software projects that require specific hardware for integration tests. In which case even more efforts might be expended on the availability of the flow’s components.

As software products and development teams get larger, the incentive to create better and more sophisticated CI flows is increased. When building more sophisticated CI flows that allow for better quality assurance there’s also the risk of increasing analysis time. Having a fast CI flow is almost as important as having high availability. Having a fast continuous integration flow with a fast feedback loop for the developer allows the developer to work faster and more efficiently. If the CI flow is efficient it has the potential to be faster than the developer’s machine in terms of building and testing their changes as it makes more economical sense to have the CI flow run on expensive and powerful hardware.

The confidence for the CI flow’s testing is an important point for the developers working with the flow, and it would require the developers of a project to write good tests. It is important how well the automated tests are written and the coverage they have for the legacy functionality that needs to be kept. The higher confidence the automated testing can give the more effective they are – as it directly impacts how much effort the developers needs to spend on checking and looking for code that might cause legacy features to break, also called legacy breakers. In other words, the developers for a project need to spend time to make sure that the automated tests can assert the requirements for the features they help implement, such that today’s features which are tomorrow’s legacy features are well tested.
Problem description

"Ericsson is one of the leading providers of Information and Communication Technology (ICT) to service providers, with about 40% of the world’s mobile traffic carried through our networks" [1]. Ericsson has a department that develops a test automation framework in Java, called MJE. The framework is used by testers for developing automated tests for the hardware supporting different radio standards (2G, 3G, 4G and 5G).

The MJE repository is “open access” within big parts of Ericsson, allowing the users (testers) to implement and add code they deem useful for their own products. The MJE framework is written in Java, and because it serves as an integral tool for testers, it is important for it to be flexible to change. It is also important for the framework to keep its legacy functionality towards the code wrapping around it, be it for new radio standards, bugfixes or new functionality.

For MJE to be easily customizable and still maintain its functionality it is rigorously tested. Updates to the repository requires the code to go through testing, which is done within a sophisticated CI flow.

The MJE framework is complex has a lot of developers contributing to the code, because it is complex a lot of time goes into writing tests for it such that the requirements set on new features are not lost over time. Because MJE has a lot of developers the framework has over time gathered more and more tests to ensure that the functionality it once promised is kept.

The tests are divided into two categories: pre-merge tests and post-merge tests. The pre-merge tests are unit tests written specifically for the framework by its own developers. The pre-merge tests are run for every code change committed to Ericsson’s version control system. The pre-merge tests are gating for code to be merged to master, meaning that they need to pass for a code change to be merged and subsequently released to the users. Whereas the post-merge tests are run post new code being merged to master. The regression tests consists of both unit tests and integration tests and are done in stages after which different “confidence levels” of the specific framework version are set.

MJE implements its CI flow as an automated system consisting of a repository management system, an automation server to setup the different stages of the flow, a build tool and a database that holds the software releases that passes the flow.

The CI flow is triggered using webhooks towards the version control system. Depending on the action made on the version control system (vcs) different parts of the flow are started: the pre-merge, when a change is pushed onto the vcs or the post-merge after a change is merged on the vcs. After a flow is triggered the different stages of the flow are performed on the available servers (CI flow slaves). The automation server micromanages the build of the project through the use of scripts, build tools and plugins. There are multiple types of tests ran in the integration flow, including unit tests, static analysis of the code and checks for code coverage. Some of these tools will be explained in more detail in the following chapter (Core concepts).
Figure 1 above is an illustration of the part of the CI flow wherein the focus of this thesis lies, namely the pre-merge flow. Whenever a change is pushed to the version control system, the pre-merge CI flow starts jobs on its slaves. Each stage, when completed, send its verdict (pass or fail) onto the next stage in line, ultimately leaving its verdict for the code change back to the version control system.

The problem of this thesis is to try to reduce the time to run tests, more specifically the unit tests that are run for every code commit. The unit tests are today the most time-consuming part of the pre-merge CI flow and all of the unit tests are run at every commit no matter how big, small or complex the code change is.

The initial idea is to try to reduce the test times by reducing the scope of tests run. The aim of this thesis is to research and finding out the viability of reducing the test scope, and if viable, which options there are to determine a reduction of the test scope. To analyse the presented options with time reduction and cost/reliability of the solution and implement a solution for the chosen option. It is not within the scope of this thesis to verify the final tool.
Core concepts

Continuous Integration (CI)

CI is a process in software development. It is the practice of merging copies or branches of a software project to a main copy/branch several times a day.

Gerrit

Gerrit is a collaboration tool for software developers, using the version control system Git. Gerrit is web-based with a big focus on providing a platform for code reviews. It is an open source project originally forked from another similar tool called Rietveld.

Gerrit review

After a developer has pushed code to Gerrit, Gerrit is configured to require a positive review of the code prior to it being merged into the master branch. A positive review means that a vote for the code to be merged has passed in favour of merging. The ones voting on the code change are usually developers responsible for the repository as a whole, or the areas affected by the code change. A vote needn’t be more than one person approving of the change. The concept of reviewing is used to get more eyes on changed/new code and thus hopefully increasing the quality of the repository.

Jenkins

Jenkins is an open source automation server supporting plugins, extensions and more to assist in building and deploying software. Jenkins is written in Java and was created by Kohsuke Kawaguchi. Jenkins was initially released in 2011 [2]. Jenkins is still supported today in 2019 under the MIT license, with claims of being the “lead open source automation server” [3].

The non-human part of the software development process is automated by Jenkins. More specifically Jenkins is a tool used to implement continuous integration and continuous delivery. Jenkins runs in servlet containers like Apache Tomcat which makes it a server based-system. Jenkins supports most version control tools such as: AccuRev, CVS, Git, Subversion, Mercurial, Perforce, TD/OMS, ClearCase and RTC. It can also execute Ant, Maven, sbt based projects, arbitrary shell scripts and windows batch commands both through plugins and through functions available through its extensive scripting functionality.

Projects containing programming languages other than Java can also use Jenkins with help of plugins. Plugins are software components that can help Jenkins to integrate with version control systems and bug databases. Plugins can also influence the way Jenkins looks or add new functionalities. Jenkins has a rich set of available plugins. Typical plugins used in Jenkins are those that support analyzing different types of reports created during builds. For example, test reports and code coverage reports. Other examples of plugins are email plugins used for sending mails to the users at the occurrences of certain events such as build failure, build success or if it detects instability in a specific build.

Maven

Maven is a project management tool for build automation supporting software projects written in Java. Maven solves the issue of building complex projects using a build configuration file. The file is defined in XML and is called a project object model file (POM). The POM is typically located at the root of the project and describes the project’s build configuration. The POM includes dependencies used in the project, the version of the dependencies and more specific configurations of the tools used for build automation. Examples of the tools used for build automation are the chosen tools for compiling, running tests and deployment.

Maven handles dependency gathering and downloading, verifying source files, compiling, analyzing, running tests, installing and packaging the project as a local JAR-SNAPSHOT which can then be imported by Maven itself or used as a JAR.
Maven has a defined lifecycle defining the build process. The lifecycle is divided into stages, each stage defines a different part in the build process for the project. Maven has default configurations for the stages but typically requires extensive configuration for larger Java projects. To define the lifecycle the user edits the project object model file (POM) which describes how the software is to be built using Maven plugins.

Maven solves dependencies in the build process by attempting to download dependencies from a central Maven repository (an open access database available on the World Wide Web for remote downloading). Maven will however initially try to resolve a dependency by checking if it already exists in the build machine’s local Maven repository. The dependencies are downloaded and stored as JAR files. Maven’s fetching of dependencies can be configured to a local or private database that follows Maven’s supported file structure. The private database can then be used as a proxy towards the World Wide Web to save developers in a software project from fetching malicious software which might have been uploaded on the central Maven repository.

Surefire

Surefire is a Maven plugin used to define the test phase of the Maven cycle, its only goal is to execute unit tests of an application. Surefire generates reports by default when running a test suite. The reports are generated in two different formats plain text and XML.

Checksum

A checksum is an integer generated from a checksum function running on a dataset and can be used to determine the integrity of a specific dataset. A checksum function is meant to return a completely different value for different dataset inputs. Meaning that if the integrity of the dataset is not held, the value returned from performing the checksum function on the dataset will not be equal to the value generated when performing the checksum function on the original dataset. The typical use is when downloading software to make sure that the file was not corrupted on the way to the end user who has access to both the checksum, the checksum function and the provided dataset. The end user retrieves the dataset, the checksum and the function used to generate the checksum and can then use the function on the retrieved dataset to confirm its consistency with the original dataset by comparing with the given checksum.

On its own a checksum is not enough to confirm data authenticity as any third party also would have access to the checksum function, meaning that if a third party gets access to alter the dataset and the checksum before they reach the end user, they can change the provided checksum to match the changed dataset.
Test case optimization

For a developer the goal of tests is to verify the behavior of code. Optimizing tests is an effort to minimize the time the developer or tester must wait to get a verdict on a code change. Some examples of test optimization are removing redundant tests from the test suite, running tests in parallel and test case selection. Test case selection is often manually done by developers in their own environment to verify their code prior to pushing it, instead of waiting for all tests to be ran in the CI flow.

Test case prioritization

The test case prioritization problem is whenever a program $P$ is modified to $P'$, to prioritize the tests in the suite $T$ so that the test cases most likely affected by the change are run first.

Running the “most likely affected” tests first would have the outcome of a quicker feedback to the developer in the cases where tests fail assuming that the test failures are reported as soon as they happen.

Test case prioritization would also effectively reduce the hardware resource expenditure when running a test suite in the case of a test failure, assuming the test case runtime is using some type of “fail fast” function, where it stops running as soon as a test fails.

Test case selection

![Figure 3 Inspired by [5]](image)

The test case selection problem is the problem of finding a particular subset ($T'$) of all tests ($T$) whenever a change in the original program ($P$) is made. The subset $T'$ should be as small as possible (precise) without losing its function (safe). "As small as possible" also called “safe”, is when the fault revealing tests for the modified program $P'$ are all part of the subset $T'$. The aim is thus to create a function $t(P, P', T)$ that returns $T'$ which is both Safe and Precise.

For a program $P$ and a set of tests $T$ it is not possible to know which tests in $T$ are affected by the modification $P'$ of the original program $P$ unless all branches of all tests in $T$ are exhausted.

The problem is that selecting the tests and running the selected test scope must take a shorter time than running the whole test scope.

$\text{TimeToRun}(\text{runTests}(t(P, P', T))) < \text{TimeToRun}(\text{runTests}(T))$

Problematization of test case selection

The point of a test case selection algorithm is to reduce the time to run the tests that can be used to verify a change in your program. It is not an impossible problem; you could run all tests and see
which lines are traversed in each test case and then deduce which tests to run (select). The test case selection algorithm, to be effective, cannot afford to run the tests to determine which tests to run as that would consume as much resources as running all the tests and effectively make the algorithm useless. The time consumption of the algorithm is important and refers to the time it takes to select which tests are to be run or not.

You can’t mathematically determine what a program will do ahead of time without running the program (halting problem). Therefore, the selection can’t be proven to be ‘correct’ – only selected the tests impacted by the change (safe & precise), without spending “too much time”. A ‘correct’ test case selection is referred to as completely safe.

The test case selection algorithm needs to be fast so that the time to run the selection and run the selected tests is less than simply running all the available tests. A type of heuristic algorithm needs to be used to select the tests; where the selected test suite might not be safe.

The algorithm needs to be safe enough – that it reduces the resources used in the test job in the continuous integration flow enough that it offsets any potential slip throughs. To measure the safety of the algorithm the hit rate of the algorithm and how often it succeeds to pick the faulty tests in N cases needs to be measured.

If one wants to be sure of safety all tests can be selected, this is where the second concept of selecting tests comes in – precision. Precision is how well the test scope is reduced. Safe and precise are not necessarily opposing concepts, where only one can be achieved, but rather two conditions that, if fulfilled, gives the desired test case selection.

Dependencies

![Dependency graph example](image)

In Java a class is typically represented by a file. Classes are used as abstraction tools for the data and functions they contain. Whenever a class is used by another class, it is dependent on by the class using it; also called a dependency. A large Java codebase like MJE generally contains a lot of classes whose dependencies can be represented by a directed graph (not necessarily connected).

Static/Dynamic dependency checking

Dependency checking is the act of solving for the dependencies of some code. Static checking refers to the act of parsing code to determine dependencies, whereas dynamic checking is real time monitoring of a program and extracting its dependencies [7].
jdeps

According to [8], "jdeps is a new command line tool added since JDK 8 for developers to use to understand the static dependencies of their applications and libraries". When ran jdeps will show the package or class level dependencies of the input depending on the configuration (configured using flags). jdeps can be run on a specific class file, a directory or a JAR file.

Groovy

According to [9], "Apache Groovy is a powerful, optionally typed and dynamic language, with static typing and static compilation capabilities, for the Java platform aimed at improving developer productivity thanks to a concise, familiar and easy to learn syntax".

Regex

A regular expression sometimes called regex is an expression that describes a search pattern, usually targeting strings to extract desired parts of the data.
**Method**

**Research Phase**

Research began with a general idea of what test case selection was, with no experience of how to implement it or if it was a fitting solution for the problems in the CI flow which were to be addressed.

The initial suggested solution, suggested by my supervisor, Irfan Mian, was to select tests to run by linking certain “code directories” to their corresponding “test directory”.

Said suggestion was prototyped and soon expanded upon after some related work was found. Larger companies such as Google [10] and Microsoft [11] implementing test case selection sparking interest in the idea of selecting tests in MJE’s CI flow. Two research articles [12] and [7] describing test case selection solutions in the form of two Maven plugins called STARTS and Ekstazi were found. The creators of STARTS and Ekstazi described their solutions in detail, which gave ideas and helped further this thesis.

**Comparing solutions**

After finding STARTS and Ekstazi the two plugins were tested and evaluated; comparing the runtime for running all MJE’s unit tests and using the plugins to run only the selected tests for a few code changes. The plugins were incompatible with MJE’s CI flow (requiring data to be stored between runs) and the time reductions were disappointing even without accounting for the incompatibility. The STARTS plugin used a JDK8 tool called jdeps to check the projects dependencies which started the idea of a self-made test case selection tool using jdeps to create a dependency graph. The graph could then be used to calculate which tests need to be run depending on which classes are changed.

**Self-made tool**

The decision to attempt to make a self-made test case selection tool was made as the available tools showed disappointing results. A proof of concept was made by testing Oracle’s own java dependency tool called jdeps, by running it against the MJE codebase and timing how long it might take to solve for the compile time dependencies. Because the time to run all tests were multiple minutes, the dependency tool could run for a while and still be a net positive timewise assuming it would cut the number of tests run by a noticeable margin. The dependency analyser required around 1/48th the time to run all the tests. An assumption was made that it would not take very long to analyse the dependency output and run it together with the modified files. Thus, the initial proof of concept was accepted to be build upon further.

It was decided that the tool should be written in the programming language Groovy, because much of the continuous integration flow was written in Groovy and because of Groovy’s natural ties and similarities to Java which the MJE developers had much experience in. Groovy also allowed for code to be written as a script, which allowed for easy implementation of the self-made tool’s algorithm, and since Groovy is similar to Java it could be ported to Java without too much difficulty.

After the implementation of the self-made tool (the Test Case Selector, TCS) it needed to be tested. It was easy to prove that it worked for X cases, but the safety and precision had to be proven: it had to be safe enough. It also had to be precise and depending on how precise the TCS proved to be, it could be less safe and still be safe enough. To prove that the TCS is safe enough is not within the scope of this thesis as it would have to be proven over time, with hundreds, possibly thousands of changes to prove that it is safe enough to run in a production environment. Safe enough is a loose concept, which is relative to the effort that is needed to correct any slippages of the reduced testing and how much reducing the test scope reduces the resources needed to run the tests (time and hardware).
Evaluating Alternatives

Starts and Ekstazi

STARTS and Ekstazi are Maven plugins developed by research teams at the University of Illinois, USA. The plugins inject into the testing stage of the Maven lifecycle when building a Maven project, meaning they are specifically tailored for Java projects using the building tool Maven. Ekstazi uses dynamic dependency checking on class level, by invoking all tests on its first run and, for every test, monitor which files in the project were read from and written to and saves the classes' dependencies into graphs. The graphs are stored in files in an /ekstazi/-directory in the projects root folder for use in future runs; meaning that on the first run in a clean workspace, Ekstazi has no saved dependencies and runs all tests. Prior to running tests Ekstazi calculates the checksums of compiled files and saves them for future use to determine which files have been changed since the last time the plugin was run. This also makes Ekstazi independent of version handling systems when computing which files have been modified since the last run, but it makes Ekstazi dependent on the data it stored in its previous run(s) to be most effective.

STARTS instead uses static dependency checking on class level to determine dependencies. To do this STARTS uses a Java (SDK 1.8 and above) tool called jdeps which provides static dependency checking. STARTS saves the dependencies gathered using jdeps in graphs written to files in a /STARTS/-directory like Ekstazi. STARTS also uses the same checksum function that Ekstazi does to determine which files have been modified since its last run.

STARTS and Ekstazi proved big and clunky and did not integrate well in MJE’s Jenkins environment. Due to Jenkins dividing up every new job on to a “clean workspace” saving the dependencies and checksums between runs becomes complicated and a solution is very likely to be unstable.

Both plugins required a long time to run even when run with no code changes since the last analysis and dependencies already collected from said analysis (See tables below). The reason for the large overhead is theorized to be due to parts of the MJE framework being configured to always compile, despite no code changes between the two compilations, reflection used in MJE and the way the plugins handle reflection.

<table>
<thead>
<tr>
<th>No changes made</th>
<th>Baseline all unit tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>All unit tests</td>
<td>1</td>
</tr>
<tr>
<td>STARTS with dependencies saved</td>
<td>0.32</td>
</tr>
</tbody>
</table>

Showing a comparison of running all tests and STARTS with dependencies collected from a previous run with no changes since. (Expected the STARTS runtime to be very short since no changes were made and no tests should have been run).

Alternate way forward

A script making use of the file structure of the project was created to have an alternate solution and possibly inspire new ideas. It started as a proof of concept which could be built upon and improved if proven promising.

The script made use of the way the MJE framework’s code is structured. When a file has been modified, the script used ‘git status’ to get the path of the modified file(s) from the root directory of the framework. The script then used regex and Groovy’s built in functionality for strings allowing for easy replacing of certain parts of a string. Everything in the corresponding test folder e.g. if module-name/src/main/java/path_to_file/class.java was changed all tests in module-name/src/test/java/path_to_file/* would be run.

The script was not thought of as the finished product because the script was precise but assumed to be unsafe. There is not always a corresponding folder, rendering the testing non-existent.

Inspiration from STARTS
STARTS had documentation which was used as inspiration to an alternate solution. STARTS implements a static dependency check which proved interesting and was chosen to investigate upon which led to spark an idea. The idea was that jdeps could be used to create a solution which would not require a rework or changes to the working CI flow and wouldn’t be dependent on the upkeep of an open source project consisting of ~20 000 lines of documentation and code.

Using jdeps to gather dependencies was tested and proved a lot faster than expected, comparing with the STARTS overhead. A prototype without the need to save anything between runs was created: a script solution using jdeps for static dependency checking, the Maven test plugin Surefire to run the tests, Git to determine modified files.

Running jdeps with the following command line argument in Linux with SDK 1.8:

```
jdeps -e shared\path\.+ -recursive -verbose:class rootDir
```

The command is called in bash with the environment variable $JAVA_HOME set to a JDK ≥1.8. The code shown above is the Linux command line argument used in the "final design" below. The command is called with three flags. The -e flag filters the class dependencies allowed in the output. The -recursive flag tells jdeps to recursively traverse through the specified directory. -verbose:class defines that the output is to be on class level as opposed to package or module level. And ‘rootDir’ defines the directory jdeps was chosen to run from.

Running jdeps requires the code to be compiled beforehand and for the MJE framework with thousands of classes, takes around 10 seconds to run. The output of the command is by default printed on the command line.

See [13] for more information on jdeps, including example outputs and flags.

**Crossroads**

After researching different solutions, a choice had to be made about which one to implement. All choices; Ekstazi, STARTS, project structure solution and the jdeps solution could have all been improved on. But there was no time to address all the different solutions to completion and thus; one of them had to be chosen as the solution to be worked and improved on. A pros and cons list of the different solutions was made to help decide upon which one to choose (Shown below).

The "checksum problem" is the incompatibility problem between the CI environment and STARTS/Ekstazi requiring files to be saved and stored in between runs. As well as parts of the MJE framework being configured to always compile, which likely causes the checksum of certain classes to never match previous runs and thus always registering as a modified class.

**Ekstazi**

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Check</td>
<td>Small developer</td>
</tr>
<tr>
<td>Safe</td>
<td>Not open source</td>
</tr>
<tr>
<td>Precise</td>
<td>Difficulty getting it to work</td>
</tr>
<tr>
<td></td>
<td>&quot;Checksum problem&quot;</td>
</tr>
</tbody>
</table>

**STARTS**

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Check</td>
<td>Small developer</td>
</tr>
</tbody>
</table>
Open source | Big codebase
Safe | Issues with reflection
Precise | "Checksum problem"

Test case selection using project structure

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>Triggers on non-code changes</td>
</tr>
<tr>
<td>Precise</td>
<td>Unsafe</td>
</tr>
<tr>
<td>Very fast</td>
<td></td>
</tr>
</tbody>
</table>

Test case selection using jdeps

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fairly simple, small</td>
<td>Triggers on non-code changes</td>
</tr>
<tr>
<td>Precise</td>
<td>Unsure about safety</td>
</tr>
<tr>
<td>Fast</td>
<td>Doesn’t inherently work with reflection</td>
</tr>
<tr>
<td>Can easily be improved upon</td>
<td>Risky, could prove difficult to implement</td>
</tr>
<tr>
<td></td>
<td>Dependent on jdeps tool</td>
</tr>
<tr>
<td></td>
<td>Requires compilation in beforehand</td>
</tr>
</tbody>
</table>

Decision

The decision to create a prototype of test case selection using jdeps was made. The name of the solution was proposed: "Test Case Selector", or TCS for short, and a prototype that works locally as a script runnable from the command line was to be implemented as a first step. The idea with the prototype was to use it as a proof of concept (POC) and in future work run it to select tests and then run said tests in the real production CI flow used in MJE and possibly other Java projects within Ericsson if proven viable.
Final design
The Test Case Selector (TCS) was written in Groovy because the MJE framework was written in Java which Groovy effectively works as a superset of and because the CI flow was written in Groovy. The Test Case Selector was in the first iteration, written as a script with the intention for it to be ran on the command line of a cloned Git repo. It used Git’s status command to get the list of modified files to base the test case selection on.

The `jdeps` command is run to solve the dependencies for the whole project every time the Test Case Selector is ran because no files are saved between two Jenkins runs (The workspace the Jenkins job uses is empty at the start of every run). Any attempts to save the dependency information, gathered from running `jdeps`, between runs was ignored as part of this thesis and added as potential future work.

The output from running `jdeps` is consequently saved into a large string. The `jdeps` output is structured as: one class followed by the directory `jdeps` was ran in `(rootDir)` and newline to define a "root class", followed by one or more lines with a class followed by the “rootDir” and newline to show the classes which the "root class" depend on (example below).

```
exampleRootClass (rootDir)
  -->dependentClassA           rootDir
  -->dependentClassB           rootDir
  -->dependentClassC           rootDir

exampleRootClass2 (rootDir)
```

... A list `jdepsOutput` where each element is a line from the `jdeps` output is made using Groovy’s `split()` method with regex looking for newline '\n' on the `jdeps` output.

Parsing of the `jdepsOutputList` is then done using a switch statement over every element in the list. The cases in the switch statement uses regex to find certain patterns; whether the current element is a root class, or not, and what the last iterated over root class was.

The data gathered from the iterating over the `jdepsOutput` is saved into two hash maps, one (dependentTests) using classes as keys and a list of the dependent tests for said class as the value. The other hash map (dependentFiles) using classes as keys and a list of classes dependent on the class as the value.

The reason why the whole `jdepsOutput` is iterated over is because the sorting into hash maps is done using two cheap operations: iterating over each element of the `jdepOutputList` and adding it to a hash map. The hash maps then provide cheap lookup operations which can be used when looking for dependencies. In contrast to devising an algorithm to iterate over the `jdepsOutputList` and look for every instance of a modified file, which for one or two modified files might be cheaper, but not in most cases as it would make the "depth function" slow. In other words, in the second case without saving into hash maps an increase in the number of modified files greatly increases the computation time because it trades iterating over and storing all the `jdepsOutput` in trade of iterating once for every modified file without storing into hash maps.

The `dependentTests` hash map is used to map a class to the tests which are directly dependent on it. Which means that calling `dependentTests.get('xClass')` returns a list of tests dependent on `xClass`, assuming there are any. This hash map can therefore be used, in tandem with the modified files to get the tests directly dependent on the changes made in a code change.

The `dependentFiles` is instead used to map a class to the classes which are directly dependent on it. Thus, calling `dependentFiles.get('xClass')` returns a list of all classes dependent on the `xClass`, assuming there are any. This hash map can therefore be used to do more thorough testing by applying the "depth function" to gather tests that are not directly dependent on a code change, but rather dependent on a class which has dependencies to a class modified by a code change.
The “depth function” makes the “final design” a greedy approach to the test case selection problem. It refers to a function in the TCS, traversing the dependency graph of the input project when selecting tests to run - to increase the selected test scope depending on the set depth. The higher depth that is set, the farther away from the originally changed classes the algorithm will go in the generated dependency graph. This is done because usually a project’s dependency graph is a connected graph, or close to it, meaning that selecting all “dependent tests” by exhausting all connected dependencies for a modified class is likely to choose most, if not all tests - leading to imprecise test case selection.

A change to a certain class could potentially affect a test in a very distant dependency (indirect dependency stretching over several edges in the “dependency graph”). Therefore, a guess of how far the algorithm should deviate from the originally modified class and its test dependencies must be made, called depth.
Results

All the test results below were gathered running the same "cut out", a subset of all the unit tests in the MJE framework. The times are gathered from a typical run of each case (except for "All tests"). All tests are run on the same shared machine and are normalized where 1 is the time to run all tests.

Note that the Test Case Selector acts very differently for different input (code changes) and the numbers therefore the results only serve to give an idea of the effect that the proposed solution has.

**Depth** is an integer that is referring to the “depth function” mentioned in the previous chapter. The value describes the number of edges away from the changed files/classes the TCS is to traverse when selecting tests to run.

The "No failure" cases have a change which does not result in any test failures.
The "Failure" case has an erroneous change made which is picked up by some unit tests.
The "No change" case is for no changes.

<table>
<thead>
<tr>
<th>No failure test (Change #0)</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>All tests</td>
<td>1</td>
</tr>
<tr>
<td>TCS Depth 0</td>
<td>0.077</td>
</tr>
<tr>
<td>TCS Depth 1</td>
<td>0.789</td>
</tr>
<tr>
<td>STARTS</td>
<td>1.369</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No failure test (Change #1)</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>All tests</td>
<td>1</td>
</tr>
<tr>
<td>TCS Depth 0</td>
<td>0.043 <strong>No tests found</strong></td>
</tr>
<tr>
<td>TCS Depth 1</td>
<td>0.063</td>
</tr>
<tr>
<td>STARTS</td>
<td>1.400</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Failure test</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>All tests</td>
<td>1</td>
</tr>
<tr>
<td>TCS Depth 0</td>
<td>0.053 <strong>DIDN’T FIND ERRONEOUS TESTS</strong></td>
</tr>
<tr>
<td>TCS Depth 1</td>
<td>0.081</td>
</tr>
<tr>
<td>STARTS</td>
<td>1.233</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No change</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>All tests</td>
<td>1</td>
</tr>
<tr>
<td>TCS Depth 0</td>
<td>0.059</td>
</tr>
<tr>
<td>TCS Depth 1</td>
<td>0.059</td>
</tr>
<tr>
<td>STARTS</td>
<td>0.320</td>
</tr>
</tbody>
</table>

The runtime compares the time of running the case compared to the “All test” case which serves as a baseline. The time includes the time for the test selectors to select tests followed by running the selected tests until there is a test failure or the list of selected tests is exhausted.

For the "no failure" and "failure" tests there is a change made to one of the files in the framework code to simulate a code change of one file.

The TCS tests were run with a depth level of 0 and 1 for the different changes.

The "STARTS" tests were runs with the STARTS plugin using 'mvn starts:starts'.
The "All tests" for the tests that didn’t crash unit tests were gathered by calculating the average runtime for five independent runs of all unit tests. `mvn surefire:test` was used for the closest comparison to the Test Case Selector, which uses a similar command line argument.

**Discussion**

The results were gathered from running the different solutions on small code changes in a large codebase with a lot of unit tests. The results are very few and the variance is high despite the test case selection algorithms being deterministic, because different input (which classes were modified) causes radical differences in the output. Therefore, the results only serve to give an idea of how the different solutions would fare in comparison to each other.

The results can be measured by comparing the runs for "all tests" to the runs for the TCS and STARTS solutions. The STARTS solution requires a previous run of all tests to solve dependencies which is not considered in these results.

The theory for why the STARTS solution falls far behind is that STARTS’ dependency checking is very expensive when the codes checksum differs between runs. Parts of the MJE codebase is generated in compile time and is therefore configured to always compile, thought to cause the checksum of certain generated classes to never match previous runs and thus always registering as modified classes. As shown when STARTS is run without any changes made (“No change” table) it still has a long runtime despite any actual changes being made.

The results from the first "No failure" test was not expected; the times shown in this case is most likely resulting from a core class with a lot of dependencies being changed, which is supported by comparing the difference in time between the two depth levels for the same test. A depth level of 0 runs the tests dependent on the changed file, while depth level 1 also adds the tests for the classes dependent on the originally changed file. Thus, if a class is very central in the dependency graph of a project: adding more depth will increase the number of tests ran by a significant amount.

The results above would suggest that a depth level of 0 is not enough. While implementing the TCS this appeared obvious as the Test Case Selector often failed to select the tests that proved the changed code to be erroneous when ran with a depth level of 0. Setting the depth level to 1 could be reduced by adding very rigorous test cases for all classes, rendering adding more depth useless and thus not doing it (setting depth to 0). I do not believe it achievable or worth the time to make tests rigorous enough for a depth level of 0 to suffice due to human error. The whole point of traversing the dependencies at a certain depth is to work around human errors such as test cases made for a specific class not catching every bug.

Because reflection is used in the code, the dependency analysis using jdeps does not suffice because it cannot solve reflection dependencies in compile time. Thus, the TCS solution cannot only rely on jdeps to gather all dependencies [14]. To deal with reflection and other exceptions (e.g. data files), a method (exception handler) with a switch case statement has been added to the TCS. The “exception handler” traverses the modified files as well as the tests selected at the end of the script's runtime, to add or remove tests based on manually configured conditions.

More rigorous testing of the TCS has begun in a "staging CI environment", a mirror of the real "production CI". The results from these tests are expected to reveal most, if not all, exceptions that need to be added to the "exception handler". The TCS will be run for a while on the staging server to gather more data and find more exceptions with the possibility of a future in the real CI flow if it proves safe enough.
Related Work

Test impact analysis in visual studio

Microsoft began working with test case selection, which they call test impact analysis (TIA), some time ago with articles dating from 2009 [11]. They have since kept the TIA feature in Visual Studios [15]. For a given commit entering the CI flow the feature will automatically select tests necessary to validate the commit. The feature follows previously failed tests, impacted tests and newly added tests. The feature also incorporates safe fall back - when it doesn’t know how to select tests it will run them all. They do however not support parallelization of tests nor code coverage calculations for changes when using TIA.

Taming Google-Scale Continuous Testing

In the paper [10] the authors discuss how Google has increased its reliance on continuous integration and explains how they spend enormous resources on testing code changes. Due to the resource requirement they are unable to test their code changes individually and have begun implementing test case selection by leveraging correlations between code, developers and test cases. They discuss their solution to the problem in a general sense, leveraging data and machine learning to their advantage in test case selection.

Test Load Balancer (TLB)

TLB [16] is written in Java and integrates with many popular build servers such as Jenkins. TLB is a tool that can be used to partition subsets of tests onto different machines both virtual and not to run the tests in parallel without any overlap of tests meaning: No tests are run more than once and no test is missed out.

Infinitest

Infinitest [17] is a plugin for IntelliJ and Eclipse written in and for Java and is used on the developer side for automatically testing the code whenever a change is made. It implements test case selection and runs tests whenever a file with dependencies is saved. Limited testing was made with this plugin, however in the testing it proved relatively slow and consumed a lot of computing power.

Future work

The TCS has been implemented and tested for unit testing. It shows great promise on the staging server and is likely to be hooked into the CI flow. There are some flaws that have not been addressed yet due to time restrictions. In this section I will discuss some of the flaws that can be improved upon in future work, the way forward for the TCS, and how it could hopefully be implemented in post-merge testing.

The TCS checks for modified files using Git and looking for modified files which is flawed since certain changes are not code changes, but rather changes in comments or even code structure with no impact on the ultimately compiled classes. STARTS/Ekstazi address this through the use of their "checksum solution" which isn't operable in MJE's CI flow. It is likely a valuable addition to add a solution for this to the Test Case Selector. The solution could perhaps be a simple static check of the changes showed using the diff command with Git.

The way TCS handles reflection by manually adding (hardcoding) exceptions to certain classes is problematic due to it being both error prone, not very precise, and very specific to the MJE codebase. A proposed solution to this problem is to have some type of dynamic dependency checking and saving dependencies between runs like Ekstazi would.

However, solving flaws comes with a cost and risks causing a larger overhead for the TCS which is ultimately overhead for running tests. Many of the solutions also risk the TCS code to get bloated and large which could cause the TCS to get too complex. There are two paths the TCS can take in the
future, the lightweight path with as few complexities as possible and the more complex path adding more and more functionality to the TCS with the downside of making the upkeep more complicated and time-consuming in the future.

There is also an idea inspired by Google's solution, gathering data and statistics and weighing changes depending on factors such as which developer is making the change and how fragile/unstable the class(es) that were changed have proven to be during historic runs.

Another idea to reduce the time spent in testing and overall health of the code is reducing the number of unit tests. This could be done by gathering data from test runs over a period, which could then be used to investigate whether certain tests are redundant and can be removed. Reducing not only test times but also the codebases’ size. Such research might also prove that some classes are not tested well enough in their unit tests and need more unit tests.

An alternate option is testing the Infinitest plugin. I am not confident that the plugin will prove useful because of reflection, unstable tests and exceptions that would need to be addressed. When tested briefly Infinitest consumed a lot of resources on the developer's machine which could cause unwanted lag and lower responsiveness for the developer which might cause the plugin to be nothing more than a nuisance.

Much of the future work is dependent on the TCS working well in the testing environment for the CI flow and thus making it worth improving on. Many of the improvements suggested would be minor. The TCS in its current state today would reduce the test times so much that the time required for compiling and other parts of the CI flow overtake unit testing in time consumption. If the TCS solution works for the pre-merge tests there is perhaps even greater potential for it to reduce overall test time by using it for the post-merge tests as well. Because the post-merge tests require a lot more time and the equipment used for them is expensive, it would be very valuable to implement test case selection for these tests.
Reflection on thesis work

I went into the thesis work lacking a lot of knowledge about the background, tools and processes that were used in the MJE framework. I came out on the other side with a much better understanding of Java development, tools and CI in a big software project.

I had only the one expectation of producing something that could be of value, be it a solution or just as an idea study. One of the main issues I faced was scope – when should I be happy with the research I have done and try to execute on it before I run out of time, rather than keep trying to find out if there are any better ways to solve the task.

The idea of testcase selection is very interesting and could prove immensely powerful to reduce hardware resources spent testing code and I am happy that I stumbled upon it as someone who is both interested in algorithms and efficiency in general.

Had I started over with what I know today

I would have made the tool purely in Java from the start because of the easy portability of said Java code. If the tool was written in Java it could also easily be deployed on a repository manager platform, with easy access from future implementations of continuous integration tools too. I also would have tried to discuss more about requirements and the drawbacks of a solution such as the TCS, as it is not proven safe, and made sure that it would get support within the organization to be implemented: if proven safe enough. I would also have made the solution a bit more generic, with easier use for different projects: although this was something I had in my mind when developing, it often came second to making anything usable, due to the lack of time.

Final words

It also might have been fun to try to make the tool open source. Open source is a big theme in the software industry today and it might have been an interesting/fun tool to open to the public. Despite of its limitations with reflection and the fact that it is indeed not a hundred percent safe, it might prove interesting and be something someone else might draw inspiration from; like I did with STARTS and Ekstazi.

Ultimately, I am quite content with how my work turned out, given the timeframe, the knowledge I began with and the initial understanding I had of the problem.
Bibliography


