Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis Nr. 45

Regression Test Suite Selection and Minimization Based on Feature Modeling

Patrick Alt

Course of Study:	Computer Science
Examiner:	Prof. Dr. Stefan Wagner
Supervisor:	Asim Abdulkhaleq, M.Sc., Tariq King, Ph.D
Commenced:	July 13, 2015
Completed:	January 12, 2016
CR-Classification:	D.2.4, D.2.5

Abstract

Regression testing of enterprise software is expensive as the whole system needs to be retested after code changes. Most of regression testing can be automated, but it still requires a lot of resources and time. Previous work at Ultimate Software designed and implemented a taxonomy manager to allow domain experts to categorize and organize tests by modeling the application domain as a set of hierarchical features, and any cross-dependencies among them. One of the long term goals of this type of modeling is to provide cost effective testing during regression as modifications to the software are validated over time. To enable this type of testing, this work presents the design and implementation of an approach for reestablishing traceability links between features and source code. With that, code metrics can be collected on the feature level and used for assessing risks and the current state of the application on the system level. It is shown that categorizing tests can greatly reduce the amount of time needed to select and run regression tests using an industrial case study. By applying knowledge gained from the traceability links between features and source code to risk-based testing it is also possible to minimize test suites by simplifying or removing tests of low risk features.

Kurzfassung

Das Verwalten und Ausführen von Regressionstests ist sehr teuer, da nach Änderungen im Programmcode das gesamte System erneut getestet werden muss. Ein Großteil von Regressionstests kann automatisiert werden, benötigt aber trotzdem viele Ressourcen und kostet Zeit. Vorangegangene Arbeit bei Ultimate Software konzipierte und erstellte einen Taxonomie-Manager, der es Domainexperten erlaubt, Tests zu kategorisieren und zu organisieren, indem die Programmdomain durch ein Set von hierarchischen Features dargestellt wird. Eines der Langzeitziele dieser Art von Modellierung ist kosteneffektives Regressionstesten. Um dies zu ermöglichen, präsentiert diese Masterarbeit den Entwurf und die Implementierung eines Ansatzes zur Wiederherstellung von Links zwischen Features und Quellcode. Damit können Code-Metriken auf dem Feature-Level gesammelt und für Risikobeurteilungen eingesetzt werden. Es wird durch ein Fallbeispiel aus der Industrie gezeigt, dass das Kategorisieren von Tests die benötigte Zeit zur Testauswahl und -ausführung signifikant verringern kann. Indem das von der Rückführbarkeit von Quellcode zu Features erhaltene Wissen auf risikobasiertes Testen angewendet wird, ist es auch möglich, Test-Suites durch Vereinfachen oder Entfernen von Tests für Features mit niedrigem Risiko zu verkleinern.

Contents

List of Figures

List of Tables

1 Introduction

This chapter begins with an overview of this Master's thesis and motivates the work. The problem statement, contributions and the context in which this thesis evolved are also detailed.

1.1 Overview

Regression testing is an important part of enterprise software development. It is a type of software testing that is used for making sure that implementing new features or fixing bugs in a system does not break existing functionality. This is usually done by re-executing tests that have been implemented previously. Whether this is done on every new commit to the source code repository or during a dedicated regression period, it always needs many resources which makes regression testing very expensive. Even if most of it is automated, it results in cost because the continuous integration needs to be maintained which requires expertise and skilled workers. Also, when changing one part of the code, the whole system has to be retested because there might be unknown dependencies.

When modeling the application domain as a set of hierarchical features and with cross-dependencies among them in a feature taxonomy, this knowledge can be manifested and made available to automated testing tools [WGAL13]. If tests are tagged with nodes of this feature hierarchy and tests are mapped to the actual code they are covering this can be used for multiple improvements in regression testing. For example, hot spots in the code, which are executed by high priority features, can be identified and considered for additional testing. Also, based on the feature a developer is working on, the selection of a sub-set of tests to be executed can be simplified.

The research contained in this thesis will explore the possibilities that using a feature taxonomy in regression testing will bring. In order to do this, we first show how different existing feature location techniques can be used to tag source code sequences or blocks with specific features. Then, we use this mapping for test selection improving the risk-based testing approach as well as regression testing. To validate the feasibility of the proposals we use examples from the industry.

This Master's thesis evolved in partnership with Ultimate Software [Ult16], a leading cloud services provider of human capital management solutions. Ultimate Software puts a great focus on testing to ensure best software quality and developed multiple internal tools to simplify and enhance black-box as well as white-box testing.

The remainder of this chapter contains the motivation of this work and the problem statement, as well as the contributions and context in which this thesis evolved. Chapter 2 explains the background this work is based on and the tools used while chapter 3 discusses related work. Chapter 4 outlays the design and chapter 5 details the implementation of the prototype created. After that, the evaluation is done in chapter 6 and chapter 7 concludes the thesis and gives an outlook to future work.

1.2 Motivation and Problem Statement

During regression testing, tests are typically executed every time a change is introduced into the system, no matter whether it is a complete new feature or only a small bug fix.

Wang et al. showed that feature modeling can reduce test selection time and effort while increasing coverage compared to a manual test selection process [WGAL13]. This leads to the idea of applying a similar approach to pure software applications. For that, it becomes necessary to be able to connect high level features with their implementation in the source code.

In the industry, risk-based testing is considered very important as it focuses testing activities on parts which could result in the most losses in case of bugs or failures, thus using costly resources more efficiently. Risk-based testing is mostly specification based, but it becomes necessary to also take code metrics into account since, for example, even high priority features could be very simple in the implementation, thus needing less testing.

Traceability among artifacts plays a key role in regression testing but typically these links between source code and requirements are lost during development. Although there is existing research on re-establishing traceability links between requirements and source code, the work presents a few challenges: It is unclear which approaches are the most practically applicable. Furthermore, there is not much work done on how the information leveraged from these approaches can be used to optimize software testing (e.g., test selection and test execution). This Master's thesis investigates how reestablishing this mapping can help us to optimize regression testing.

Several research questions surface: (1) What is the most practical way of re-establishing traceability links between features/requirements and source code? For this, existing feature location approaches need to be evaluated and prototypes implemented. (2) Can using a feature taxonomy to represent the problem domain improve regression testing? If so, how? (3) Can the information gathered in (1) be leveraged to improve findings/outcome of (2)? If so, how?

This Master's thesis will try to answer these questions by researching and evaluating how feature modeling and requirements-to-source-code traceability links can optimize regression testing.

1.3 Contributions

This work makes the following contributions:

1. A new approach for reestablishing the traceability links between requirements and source code and leveraging them for testing in a practical way. For example, it is possible to analyze code coverage on system level while program code and testing code are not interacting directly with each other, e.g. this approach can be used if the testing code opens a browser and interacts with a web application that is analyzed over the network.

2. The prototypes built and the results they have produced in the context of a real-world case study at Ultimate Software, including enhancements to the build pipeline to improve performance and scalability of acceptance test execution in a continuous integration system. Reestablishing the traceability links between requirements and source code is automated and can be executed in the build pipeline without any effort necessary from the developers or test engineers.

3. Evaluation of those results and determination of the feasibility of the approach by applying the result data to risk-based testing. This enables the possibility of taking code metrics into account when assessing risks.

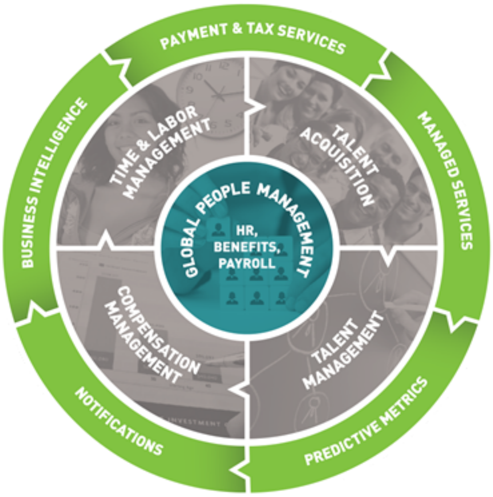


Figure 1.1: Product Wheel [Ult16] of UltiPro, Ultimate Software's core product

1.4 Context

Research and development work for this Master's thesis was done at Ultimate Software [Ult16] which is a cloud services provider of human capital management solutions based in Florida. Its main product is UltiPro, an integrated human capital management system which deals with all aspects of employee management from recruiting to retirement (see Ultimate Software's product wheel in figure 1.1).

The core of UltiPro consists of global human resources (HR) management, benefits management and the payroll engine calculating taxes and paychecks. Other subproducts like Recruiting and Onboarding for talent acquisition or applications for talent management, compensation management and time and labor management are add-ons that tie into the core. This means the application domains are interconnected. Therefore, building a unified solution to human capital management is very complex and requires extensive domain knowledge.

To ensure best software quality, the company has a team designing and developing internal tools to simplify and enhance black-box as well as white-box testing. This team is led by test architects who put a great focus also on researching the latest work about software testing or exploring forms of software testing that are not generally used. For example, recently a tool implementing and automating mutation testing [JH11] was created.

This Master's thesis evolved in conjunction of this internal team and a production team which is developing the UltiPro Recruiting product. This way the research and evaluation could be done with an actual enterprise software application.

2 Background

The following section details the fundamentals this work is based on and the context in which it was developed.

2.1 Regression Testing

Regression testing consists of different kinds of testing, mainly unit testing, integration testing and system testing which are all focused on ensuring existing functionality does not break when new features are added to the code base or when hot fixes are applied [LW89]. These tests are usually written during the development process of the corresponding feature or hot fix and checked in along with the code to be able to rerun them for regression.

Regression testing of enterprise software is an expensive process, requiring thousands of tests to be executed at multiple levels.

Using the latest technologies for build and testing agents, unit and integration testing are fairly cheap compared to system level testing because the tests have to be executed at a much higher level (end-to-end) and thus they are more time-consuming. While unit tests and integration tests are almost always automated, system testing is also often a manual process during regression periods which needs additional resources in the form of engineers. For automating system level testing, Ultimate Software uses a tool called Echo (detailed in section 2.6.2).

Regression tests usually consist of a combination of white-box and black-box tests. White-box tests are testing a program's internal structure as opposed to its required functionality. Tests are set up to follow different execution paths through the program by manipulating the input values and validating the output values or checking whether specific other methods have been called. By making sure that every path is exercised the tester tries to maximize the statement coverage [Ost02].

Black-box testing in comparison is validating the functionality of a system or component in terms of the requirements. A tester is specifying inputs and validating the outputs or results against the specification and is not interested in the internal structure or how the system is creating the results [Bei95].

2.2 Risk-Based Testing

Risk-Based testing is a technique for prioritizing testing activities so that one can get the most out of testing efforts [Aml00].

A bug in a system may result in undesirable consequences for the user of the system as well as for the company or team that develops the system. Such consequences are uncertain, but if they become reality they lead to loss. Risk is the probable frequency and probable magnitude of such a loss [Gro11].

A testing approach that takes this risk into account is called risk-based testing [Leh11]. When identifying and analyzing risks of a system, testing efforts can be focused on the most critical parts, thus resources can be spent more wisely. Testers, without being introduced to risk-based testing, usually use a form of risk-based testing, especially during exploratory testing because for that they make decisions on e.g. what is the most important new feature to test that one more thoroughly [Aml00]. However, risk-based testing, as opposed to traditional testing, is done in a systematic way and uses formal risk assessment methods.

Assessing risks can be done using risk heuristics [Bac99] and risk is calculated as the product of cost and probability (Risk = Cost * Probability) [Aml00], where cost and probability are weighted sums of different factors:

$$\begin{aligned}\text{Cost} = &(\text{weight for impact factor 1} * \text{value for this factor})\\ &+ (\text{weight for impact factor 2} * \text{value for this factor})\\ &+ \ldots\\ &+ (\text{weight for impact factor n} * \text{value for this factor})\end{aligned}$$

$$\begin{aligned}\text{Probability} = &(\text{weight for probability factor 1} * \text{value for this factor})\\ &+ (\text{weight for probability factor 2} * \text{value for this factor})\\ &+ \ldots\\ &+ (\text{weight for probability factor n} * \text{value for this factor})\end{aligned}$$

Risk Area	Criticality (Weight: 10)	Frequency (Weight: 3)	Complexity (Weight: 3)	Schedule (Weight: 10)	Risk
New Hire Integration	4	4	4	4	$(4*10+4*3)$ $*(4*3+4*10)$ $= 2704$
Logo Config	3	1	2	5	$(3*10+1*3)$ $*(2*3+5*10)$ $= 1848$
Sort Opportunity	3	3	2	1	$(3*10+3*3)$ $*(2*3+1*10)$ $= 624$

Table 2.1: Example for calculating risk

Possible heuristics include:

- Criticality – How important is the feature to the customer?
  Example: 1 - Unimportant, 5 - Business Critical
- Complexity – Feature may contain complicated input and processing steps
  Example: 1 - Simple, 5 - Highly Complex
- Popularity/Frequency – Feature is heavily used by customers
  Example: 1 - Rarely used, 5 - Always used
- Schedule – Moving fast can lead to poor quality
  Example: 1 – No time pressure, 5 - Aggressive Schedule

Weights to these factors are usually assigned with the values 1 (Low), 3 (Medium) and 10 (High). Table 2.1 shows an example risk calculation partly based on the UltiPro Recruiting product. In there, criticality and frequency are used as impact factors while complexity and schedule are used as probability factors.

Based on these calculations testing efforts can be prioritized: High risk items need thorough testing, medium risk items are tested ordinarily and low risk items need only light testing or no testing at all.

Up to now risk is mostly assessed from a business perspective (popularity, criticality or complexity from a domain point of view), but it lacks taking into account information from the code (e.g. how complex is the implementation of a specific feature). This Master's thesis opens up possibilities to take these aspects into account as well.

2.3 Model-Based Testing

Model-based testing is an approach to system testing by creating a model for the system under test and using this model to interact with the system [EFW01]. Since these tests are executed on the system level and the test cases are derived from the model, not from the source code, model-based-testing falls into the category of black-box testing. Ultimate Software uses a tool called Echo for creating models of its products and simplifying the creation of functional tests.

2.4 Feature Modeling

In an effort to improve traceability between requirements and test cases or test fixtures, feature modeling is used to capture the domain in a systematic way [KCH+90]. In a feature taxonomy, the application domain can be modeled as a set of hierarchical features, and with any cross-dependencies among them.

There exist several different notations for feature models [BSRC10]. For example, parent-child relationships can be marked as "mandatory", "optional", "alternative" (exclusive "or") or "or". For the industrial case study of this work, we use and extend a previously created, custom taxonomy manager that is used for feature modeling at Ultimate Software (see section 2.6.1).

2.5 Feature Location Approaches

Functional tests don't interact with the source code directly, the system under test and the test execution framework are often even executed on different servers. In case of cloud services like Ultimate Software is providing these are usually web pages running on cloud environments. These web applications are tested by functional tests executed from build agents in the pipeline. Thus, detecting the connection between tests and the respecting code that they are covering cannot be done by simple static analysis. Thus, we will be using feature location approaches in order to map source code to features and the feature taxonomy in order to map features to tests.

Many researchers are already active in the field of feature location [DRGP13]. These approaches are mainly categorized in dynamic, static and textual feature location approaches.

With dynamic feature location you use execution traces created when running specific sets of features and compare them against each other. For example, parts of the code that are executed only for one test or a subset of test cases must be specific to the tests associated with a feature while parts of the code that are executed for most of the tests cannot be associated with a feature. This approach is called Software Reconnaissance [EV05] [WGHT99] [EK01].

For static feature location the developer usually needs to seed the feature location tool with some entry points. The tool is then analyzing the source code statically and follows call graphs and static dependencies to find code specific to a feature [RM02]. Executing the code is not necessary.

Textual feature location approaches use language processing and information retrieval in order to find features in the code based on names of classes, methods or variables and comments [MSR04].

Combining different approaches proved to provide better results [EKS03] [EAAG08].

Another way of locating features is using meta data of source code repositories [Yao01].

2.6 Tools

This section details all the tools and applications that will be used or extended in this work for the industrial case study.

2.6.1 Taxonomy Manager

Ultimate Software developed an internal tool for managing feature taxonomies (see figure 2.1) that can be changed and extended by business analysts and developers, and managed by domain testers. In addition to the parent-child relationship (curved lines) in the tree, cross-dependencies between features can be modeled (arrows) by simple click and drag'n'drop actions.

Furthermore, synonyms can be assigned to every feature tag so that, for example, somebody who is looking for a feature does not have to know the actual name of the feature like it is used in the taxonomy.

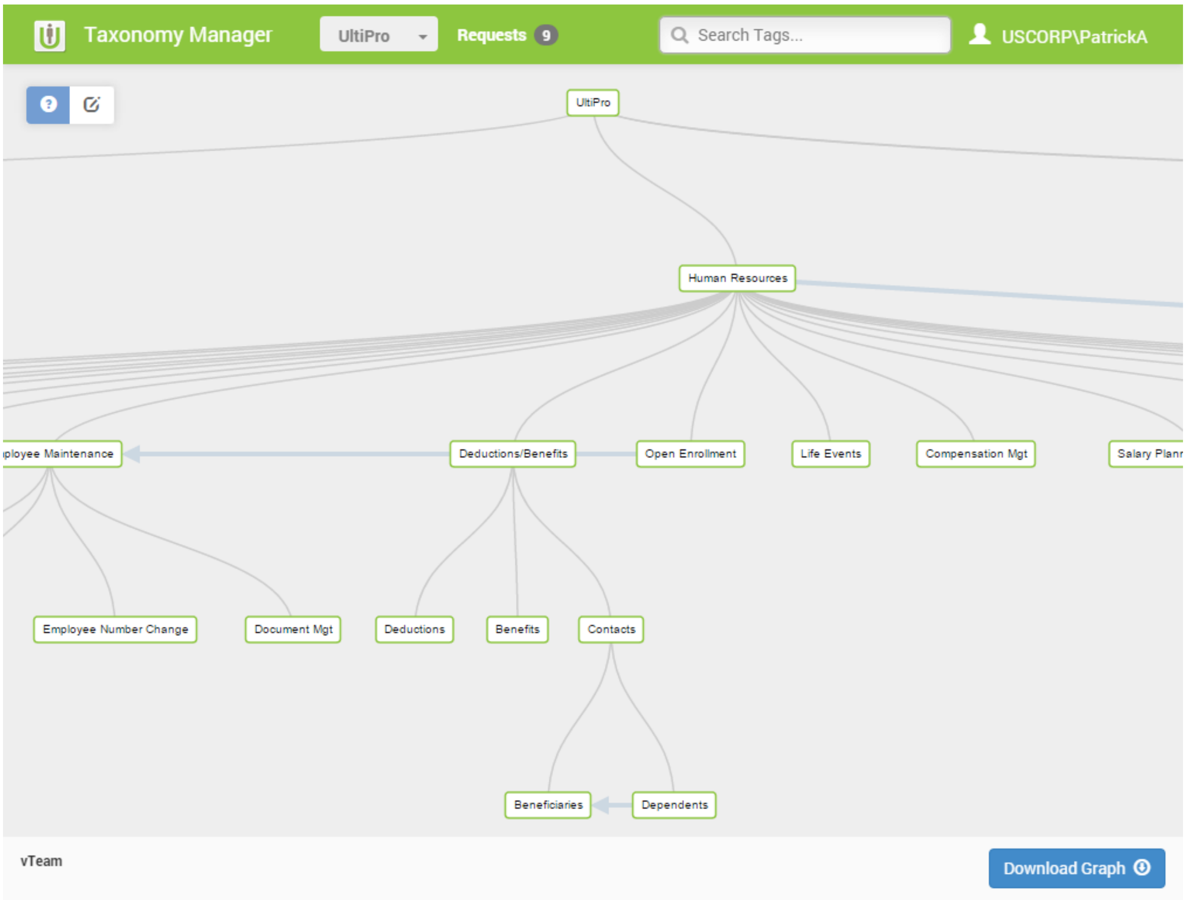
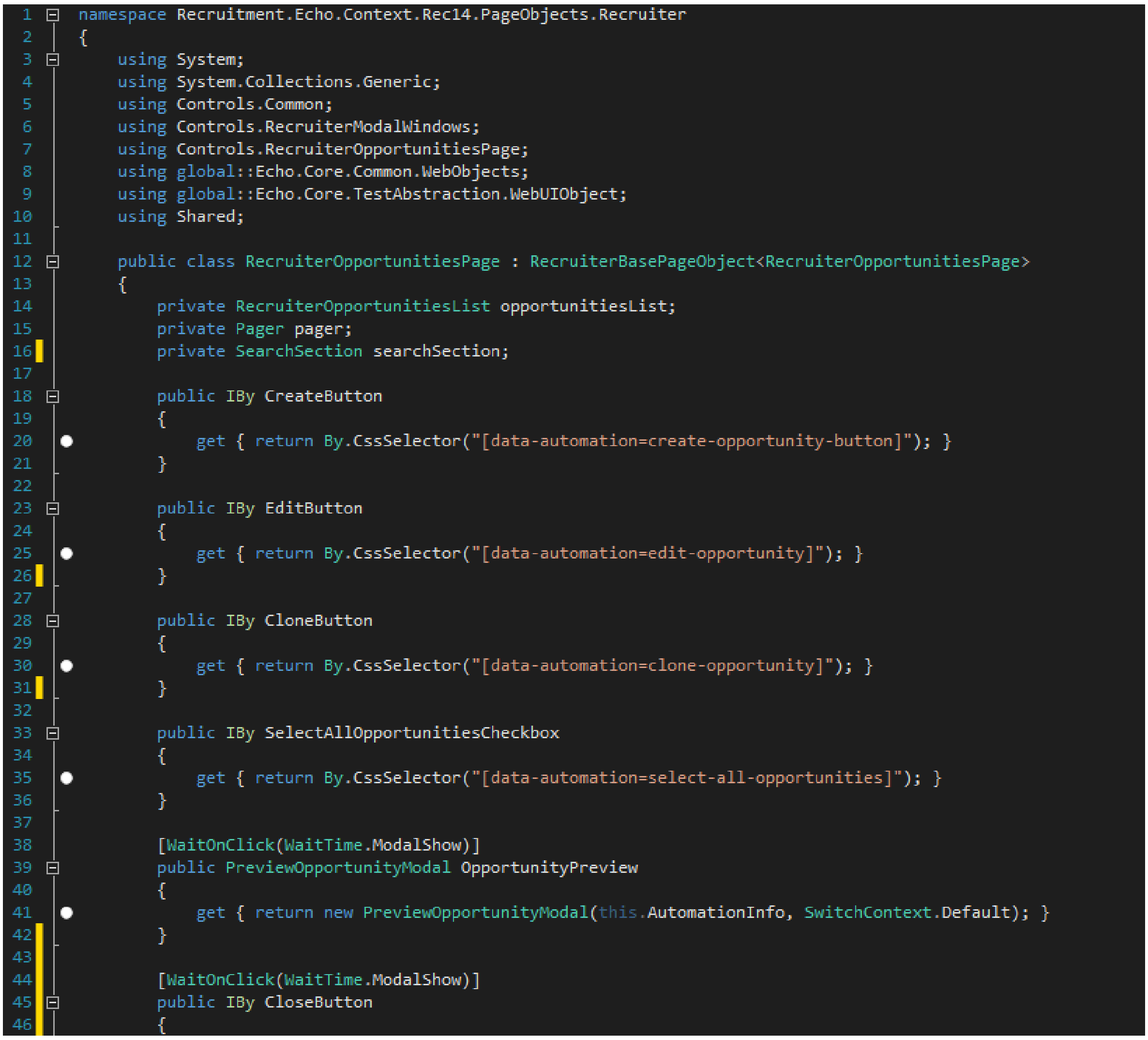


Figure 2.1: Part of the feature taxonomy of UltiPro

2.6.2 Echo

Ultimate Software developed a tool called Echo in order to streamline and standardize the way functional automated system tests are authored and in order to abstract out different system level automation frameworks. Currently Echo is supporting Selenium, Coded UI and Sikuli in that regard as well as Appium for mobile applications.

Echo also provides the ability to define application specific models by creating page object classes with properties defining the controls that can be found on those pages and how the respective automation framework can find these controls on the pages. Please see figure 2.2 for a small example of such a page object class. This page of the Recruiting web application is called `RecruiterOpportunitiesPage` and it has several controls including a `CreateButton`, an `EditButton`, or a sub-page in a modal called `OpportunityPreview`. For all these properties it is also defined how the automation framework (Selenium in this case) can find those controls on the page by specifying CSS selectors.

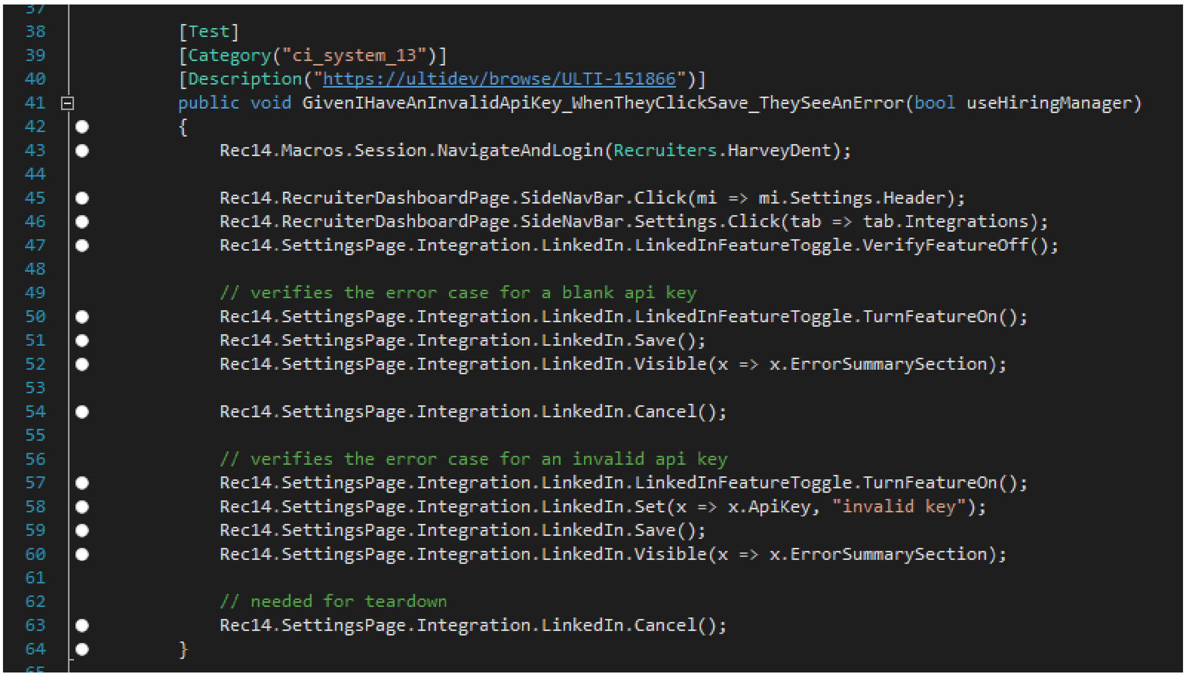


```
namespace Recruitment.Echo.Context.Rec14.PageObjects.Recruiter
{
    using System;
    using System.Collections.Generic;
    using Controls.Common;
    using Controls.RecruiterModalWindows;
    using Controls.RecruiterOpportunitiesPage;
    using global::Echo.Core.Common.WebObjects;
    using global::Echo.Core.TestAbstraction.WebUIObject;
    using Shared;

    public class RecruiterOpportunitiesPage : RecruiterBasePageObject<RecruiterOpportunitiesPage>
    {
        private RecruiterOpportunitiesList opportunitiesList;
        private Pager pager;
        private SearchSection searchSection;

        public IBy CreateButton
        {
            get { return By.CssSelector("[data-automation=create-opportunity-button]"); }
        }

        public IBy EditButton
        {
            get { return By.CssSelector("[data-automation=edit-opportunity]"); }
        }

        public IBy CloneButton
        {
            get { return By.CssSelector("[data-automation=clone-opportunity]"); }
        }

        public IBy SelectAllOpportunitiesCheckbox
        {
            get { return By.CssSelector("[data-automation=select-all-opportunities]"); }
        }

        [WaitOnClick(WaitTime.ModalShow)]
        public PreviewOpportunityModal OpportunityPreview
        {
            get { return new PreviewOpportunityModal(this.AutomationInfo, SwitchContext.Default); }
        }

        [WaitOnClick(WaitTime.ModalShow)]
        public IBy CloseButton
        {
```

Figure 2.2: Part of a page object of the Recruiting product

An example of how this model is used in a test can be seen in figure 2.3. It is one test that first logs in and then performs some actions on the dashboard page and the settings page. When tests like this one are executed Echo will open an actual browser (supported are Firefox, Chrome and Internet Explorer) and perform the specified automation commands on the web application like a human user.

Please also note the `Category` attribute at the top of the test which is currently only used for slicing tests into different build pipelines. This attribute will later be used to tag tests and test fixtures with features of the feature taxonomy.



```
        [Test]
        [Category("ci_system_13")]
        [Description("https://ultidev/browse/ULTI-151866")]
        public void GivenIHaveAnInvalidApiKey_WhenTheyClickSave_TheySeeAnError(bool useHiringManager)
        {
            Rec14.Macros.Session.NavigateAndLogin(Recruiters.HarveyDent);

            Rec14.RecruiterDashboardPage.SideNavBar.Click(mi => mi.Settings.Header);
            Rec14.RecruiterDashboardPage.SideNavBar.Settings.Click(tab => tab.Integrations);
            Rec14.SettingsPage.Integration.LinkedIn.LinkedInFeatureToggle.VerifyFeatureOff();

            // verifies the error case for a blank api key
            Rec14.SettingsPage.Integration.LinkedIn.LinkedInFeatureToggle.TurnFeatureOn();
            Rec14.SettingsPage.Integration.LinkedIn.Save();
            Rec14.SettingsPage.Integration.LinkedIn.Visible(x => x.ErrorSummarySection);

            Rec14.SettingsPage.Integration.LinkedIn.Cancel();

            // verifies the error case for an invalid api key
            Rec14.SettingsPage.Integration.LinkedIn.LinkedInFeatureToggle.TurnFeatureOn();
            Rec14.SettingsPage.Integration.LinkedIn.Set(x => x.ApiKey, "invalid key");
            Rec14.SettingsPage.Integration.LinkedIn.Save();
            Rec14.SettingsPage.Integration.LinkedIn.Visible(x => x.ErrorSummarySection);

            // needed for teardown
            Rec14.SettingsPage.Integration.LinkedIn.Cancel();
        }
```

Figure 2.3: Example test of the Recruiting product

2.6.3 TeamCity and Continuous Integration

TeamCity is a build management and continuous integration server developed by JetBrains [Jet16b]. In TeamCity you can set up build pipelines for each of your products which are connected to the corresponding source code repositories.

On each commit (usually including commits to branches other than the main branch) the build pipeline is triggered to go through a series of automated tasks to confirm the validity of the checked-in changes. In most cases, first some syntax checks (using linters) are done to be able to reject obvious mistakes as early as possible. Then the product is compiled and unit tests are executed. If these are successful integration tests can be executed and the product deployed to a test environment.

The next steps usually involves the acceptance tests which are often led by so-called smoke tests. Smoke tests usually quickly touch every main part of the product to detect major defects before the time-consuming functional or non-functional tests are kicked-off [DRP99]. The name "smoke test" refers to the smoke that an electric device may produce after just turning it on when not functioning correctly.

If all acceptance tests pass the product can be considered for a release. The necessary binaries for that are usually provided by TeamCity as artifacts or you can trigger a production deployment directly through TeamCity if it is part of the build pipeline like it is the case for the UltiPro Recruiting product.

2.6.4 UltiPro Recruiting

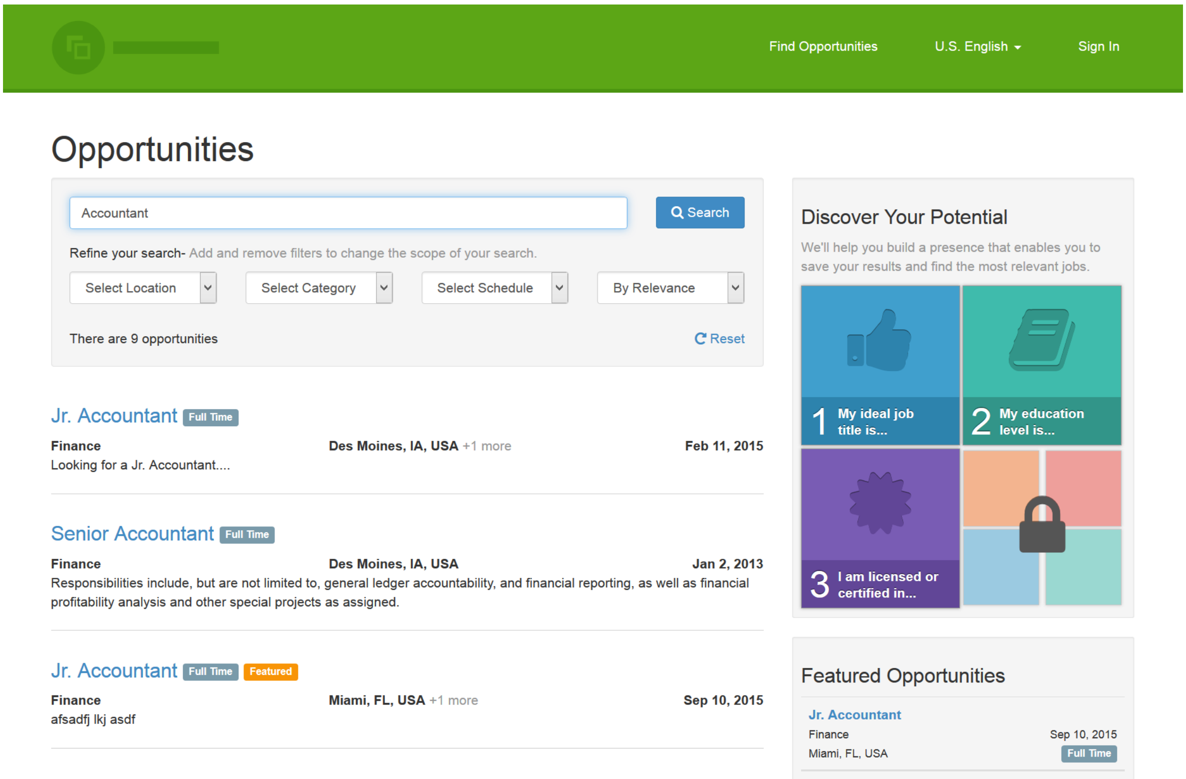
Although the results of this Master's thesis will be used throughout the company in almost all the development teams sometime in the future, the UltiPro Recruiting product [Rec16] served as a prototype for the initial design and evaluation.

Recruiting is a modern web application that simplifies talent acquisition for companies. Candidates seeking jobs can browse through opportunities published by the correspondent employer and apply to them directly online (see figure 2.4 for a demo job board).

UltiPro Recruiting is a multi-tenant system meaning it is deployed to only a few server centers (two in the USA and one in Canada), but each instance serves multiple tenants (customers). To the outside visitors these web pages appear completely separate and customers are not aware of each other. This fact becomes important during the design and implementation of this work.

For regression testing, the Recruiting solution currently contains 14628 unit tests, 1907 integration tests and 1624 functional tests (using Echo), all of which are currently executed for the acceptance process of every new feature and bug fix. A complete run of the pipeline currently takes one hour and ten minutes on average, if enough build agents are idle (details will be provided in chapter 6).

Recruiting is a ASP.Net application written in `C#` and using MongoDB as its database. The client side uses Knockout and Twitter's Bootstrap for an appealing and responsive web appearance.



Find Opportunities U.S. English Sign In

Opportunities

Accountant

Search

Refine your search- Add and remove filters to change the scope of your search.

Select Location | Select Category | Select Schedule | By Relevance

There are 9 opportunities

Reset

Jr. Accountant Full Time

Finance Des Moines, IA, USA +1 more Feb 11, 2015

Looking for a Jr. Accountant....

Senior Accountant Full Time

Finance Des Moines, IA, USA Jan 2, 2013

Responsibilities include, but are not limited to, general ledger accountability, and financial reporting, as well as financial profitability analysis and other special projects as assigned.

Jr. Accountant Full Time Featured

Finance Miami, FL, USA +1 more Sep 10, 2015

afsadfj lkj asdf

Discover Your Potential

We'll help you build a presence that enables you to save your results and find the most relevant jobs.

1 My ideal job title is...

2 My education level is...

3 I am licensed or certified in...

Featured Opportunities

Jr. Accountant

Finance Sep 10, 2015

Miami, FL, USA Full Time

Figure 2.4: Demo page of UltiPro Recruiting

3 Related Work

The following paragraphs summarize work related to this thesis and points out significant differences.

The industrial case study of Wang et al. [WGAL13] about automated test case selection using feature modeling presented a systematic and automated approach for test case selection in the context of Cisco's Video Conferencing Systems. They introduced a feature model for the Saturn product line of Cisco which was created using the expertise of their test engineers and based on the domain knowledge and system information. They also specify cross-dependencies between features. It is worth noting that building this feature model was a one-time effort because according to their test engineers, the functionalities of Saturn do not change much. By contrast, the feature model of an evolving software application needs to be maintained as the system grows.

In addition to the feature model, the authors also modeled the test structure using what they call a Component Family Model. This Component Family Model represents how products are assembled by modeling relations between components and parts. Thus, tests could be associated with specific parts of the product depending on which part they test. Furthermore, the Component Family Model is linked with the feature model (e.g. a part of the product can be linked with one or more features from the feature model).

The feature model of the Saturn product line can then be used when a new product needs to be tested: A test engineer simply selects the relevant features from the feature model based on his domain knowledge and the system can automatically find the relevant tests by traversing the Component Family Model for every specified feature and dependent features. The authors have shown that using this approach significantly reduced test selection time and effort while coverage was increased compared to the previous manual approach.

For a pure software application, the concept of a Component Family Model is not applicable because new products are usually not built based on parts of a product line. A software application usually evolves over time while new features are added.

Thus, this Master's thesis presents a new way of linking tests with features of a feature model.

Automatic test case selection for regression testing is already actively researched [GHK$^+$01] [RH96]. Since executing all regression tests when changes are introduced is very expensive or time-consuming, the alternative idea is to rerun only a subset of the tests which has two implications: First, running algorithms or other test selection methods can carry costs as well and second, the test effectiveness may be reduced because tests that would detect bugs may be discarded.

For example, the linear equation technique [HR90] uses systems of linear equations to represent relations between tests and program code that are obtained by tracking executions. Then, an integer programming algorithm is used to identify necessary tests.

Another approach is the graph walk technique [Rot96] that builds control flow graphs for the original program and the changed code. Edges in these graphs are potential affected entities that are associated with test cases using execution traces. By doing depth-first searches in the two graphs in parallel it detects which tests need to be executed in order to cover the changed parts when the targets of two edges with the same label differ.

Benedusi et al. present an approach that uses the set of program paths through a program represented as algebraic expressions. It is transforming these expressions to acyclic paths from entry to exit point of the program and analyzing tests to detect which of those paths these tests traverse. By comparing the paths to the ones of a modified program it selects those tests for execution that traverse the modified paths [BCC88].

Most of the approaches have one requirement in common: Code needs to be instrumented in order to collect execution traces or data flow information. Schumm [Sch09] presents a practicable and feasible solution for real-world applications using the white-box testing tool CodeCover [Cod16]. It is developed as a plugin for the integrated development environment Eclipse and supports the programming language Java.

By analyzing which tests are covering which parts of the code it can associate tests with code statements or methods, thus supporting unit tests and integration tests. By calculating the differences between two program versions by comparing the code it can automatically select all test cases that are affected by the changes. This way only the necessary tests are executed.

According to the author the analysis of big industry software systems is supported, however the test selection technique works only for unit and integration tests because the code coverage analysis depends on the direct connection of program code and testing code in the same environment (e.g. a unit test directly calls a method of the program code). In order to support functional tests, this Master's thesis proposes a new way of code coverage analysis with which it is not necessary for program code and testing code to be directly connected.

Chen et al. present a specification-based approach to regression test selection [CPS02]. By using an activity diagram to represent the desired system behavior the traceability between requirements and test cases is maintained much like the feature taxonomy of this Master thesis. However, maintaining an activity diagram whenever the specification or requirements change requires more than maintaining a feature taxonomy since it is more detailed. In order to establish traceability between source code and the activity diagram, they use a code change history which needs to be documented manually by the developer whenever a code change is introduced. As opposed to that, this Master's thesis presents an automated approach of establishing traceability between source code and requirements.

This Master's thesis will use feature location for associating program code with features of the domain. Since we will have test that are tagged with features, the feature location approach commonly called software reconnaissance [EV05] is the most feasible method that can be applied here. It is based on dynamic feature location (see section 2.5) and uses execution traces of tests that are associated to specific features to discover which parts of the code are executed for which features. According to Wilde and Scully [WS95] this requires surprisingly only a small number of tests.

However, it is not detailed how execution traces are gathered exactly other than mentioning that a test coverage tool is used. In order to apply software reconnaissance to functional tests in an enterprise software environment, this thesis proposes an approach that makes dynamic feature location possible on system level even if tests are interacting with the application through a browser (and thus are not sharing the same environment).

This thesis is combining software reconnaissance with the findings of Wang et al. [WGAL13] that feature modeling can significantly improve test selection in order to reduce costs of regression testing and enhance risk-based testing.

4 Design

This chapter details the creation of the feature model for Recruiting and the design thoughts for accomplishing the goals of this Master's thesis. First, the steps involved in coming up with the initial feature taxonomy are outlined, then changes to the taxonomy manager are suggested and after that the necessary changes and ideas for implementing a feature location technique for functional tests are explained.

4.1 Feature Taxonomy and Taxonomy Manager

The first step for this work involved creating an initial feature taxonomy of an actual industry software application in order to be able to base the evaluation on real-world data. The UltiPro Recruiting product was selected for this purpose because it is one of the newer products of Ultimate Software and is a stand-alone application that is only loosely coupled to core UltiPro via API requests, as opposed to most of the other products which are tightly integrated into UltiPro. This makes a separated analysis and evaluation much easier.

The creation of the Recruiting feature taxonomy was kicked off in a one-hour meeting with two business analysts of the product who are familiar with the complete domain knowledge of the application. Based on a root tag called `Recruiting` all entities of the domain have been arranged in a hierarchy as it was seen fit using the taxonomy manager (see section 2.6.1). It seemed feasible to model the domain around the three main roles in the application:

- Candidates, who search for and apply to opportunities as well as maintain a presence with their profile in the application.
- Recruiters, who browse candidates and their applications, move candidates through the recruiting process and publish new opportunities.
- Recruitment Administrators, who administer and manage the application for a company.

Thus, the tags `Candidates`, `Recruiters` and `Recruitment Administrators` were created as children of `Recruiting` and all other domain aspects where added as sub-hierarchies pretty quickly within this one-hour meeting because everybody was very familiar with the domain.

A question arose whether the taxonomy should be built strictly only based on the domain or whether features that are not necessarily part of the domain should also be represented in the taxonomy. For example, the "Discover Your Potential" section of the main screen for candidates (see figure 2.4) is not considered part of the domain, but solely as an implemented feature which is simplifying the search for opportunities by using visually appealing elements.

A further meeting with two test architects of the company resulted in the decision that big separate features like the "Discover Your Potential" section that is seen distinctively from a functional point of view should indeed be represented in the taxonomy, but other than that the feature model should be based on the domain.

In total three one-hour meetings with these two test architects were necessary to go through the hierarchy in the taxonomy manager and rearrange tags based in their knowledge about feature modeling and testing.

During this whole process it became clear very quickly that the existing taxonomy manager was never actually used before as a lot of bugs were detected and many functionalities for creation of the taxonomy were either missing or not intuitively to use. For example, moving a sub-graph to a new parent tag caused the web page to freeze and the graph frequently collapsed and expanded without reason.

In addition, new functionalities were to be added to support risk-based testing and the integration into the build pipeline of a product:

- A priority (low, medium, high) should be selectable for every feature tag.
- A simple overview window for every tag showing data like priority, synonyms and custom dependencies (specified by adding cross-dependency links in the tree).
- An API interface for the automated validation of feature tags.

The last item is necessary for assuring that only valid features are used when tagging tests. This can be accomplished by adding an automated test to the product solution which verifies every feature tag found in the code base by sending it to the taxonomy manager. In a similar way an automated test needs to be added which ensures that every test is actually tagged with at least one feature. This may seem unnecessary because software engineers and especially the test engineers should make annotations like this as an important part of their work but it turned out that they are often sloppy with things that don't directly contribute to fulfilling the requirements of their current task.

4.2 Simplifying and Enhancing Test Execution

In preparation for being able to execute only a subset of tests in the TeamCity pipeline, the structure and setup of the functional tests needed to change. Currently the 1624 functional tests are executed in 21 different build configurations in order to be able to run them concurrently. In figure 4.1 you can see the acceptance test project in TeamCity for the Recruiting solution. First, a test environment gets deployed with the current code of a branch, then the integration tests and the system tests get executed in parallel. Due to the amount of time that the system tests need (almost 11 hours if executed sequentially) they are currently split manually into 21 system test groups, each of which takes between 30 and 40 minutes to complete.

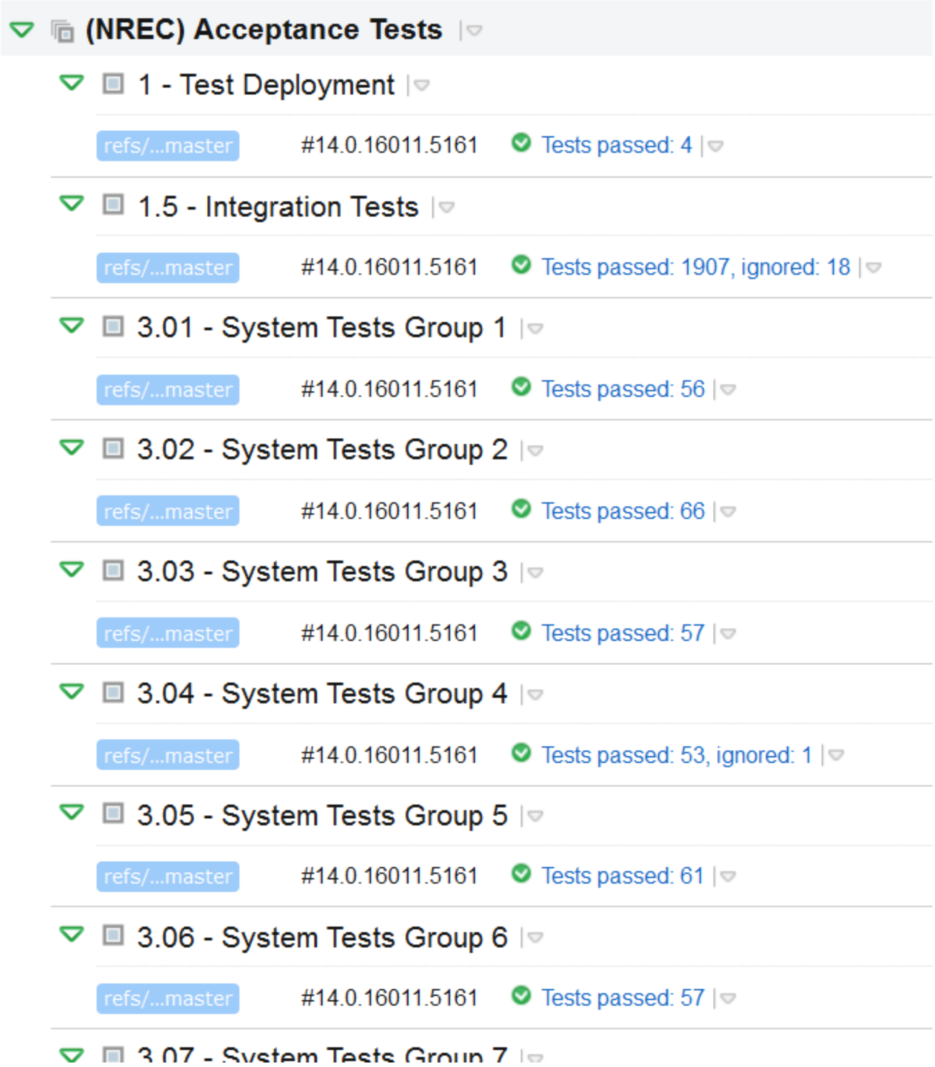
The splitting of the functional tests is accomplished using the `Category` attribute of the testing framework NUnit [PNV$^+$16] on which all tests in the solution are based. By specifying a string `ci_system_X` with $X = 1, 2, \ldots, 21$ which can be used for filtering in the NUnit test runner the test fixtures are manually balanced between the 21 system test configurations in TeamCity.

In order to be able to execute all tests related to a feature while still running them as efficiently as possible this design needs to change. For example, in case the tests of a specific feature and their dependent tests all happen to be in the same testing group, we would lose the advantage of concurrency.

It is possible to solve this problem by updating the testing framework to NUnit 3.0 [PPBC16] which natively supports concurrent test execution. However, a single build agent would not be able to handle the load of dozens of browser instances running concurrently. Thus, we try to transfer the load to a so-called Selenium Grid. The next two sections cover this in more detail.

4.2.1 NUnit 3

NUnit 3 has been in development for several years now and just recently a stable version has been released. By upgrading to NUnit 3 it is possible to parallelize tests on the fixture level without having to spawn threads as a user. It is handled internally by



(NREC) Acceptance Tests

1 - Test Deployment
refs/...master #14.0.16011.5161 Tests passed: 4

1.5 - Integration Tests
refs/...master #14.0.16011.5161 Tests passed: 1907, ignored: 18

3.01 - System Tests Group 1
refs/...master #14.0.16011.5161 Tests passed: 56

3.02 - System Tests Group 2
refs/...master #14.0.16011.5161 Tests passed: 66

3.03 - System Tests Group 3
refs/...master #14.0.16011.5161 Tests passed: 57

3.04 - System Tests Group 4
refs/...master #14.0.16011.5161 Tests passed: 53, ignored: 1

3.05 - System Tests Group 5
refs/...master #14.0.16011.5161 Tests passed: 61

3.06 - System Tests Group 6
refs/...master #14.0.16011.5161 Tests passed: 57

3.07 - System Tests Group 7

Figure 4.1: Part of the system test groups of Recruiting in TeamCity

NUnit and the number of worker threads can be specified as an option of the NUnit test runner.

In order to avoid tests interfering with each other it comes in handy that Recruiting is a multi-tenant system which allows us to protect test fixtures against each other by simply running them against their individual tenant in the system.

This has previously been done by setting up one tenant for each of the 21 system test groups. Now we need to activate one tenant for every test fixture instead.

4.2.2 Selenium Grid

The Echo tests in Recruiting use Selenium [Sel16a] as the underlying execution framework. By deploying a so-called Selenium Grid [Sel16b] it is possible to outsource the browser execution of tests to other nodes in the network. The execution of the tests is still managed by the NUnit test runner on the build agent, but it communicates Selenium commands to the grid which is then automatically balancing the execution. Thus, the main load is taken off of the build agent.

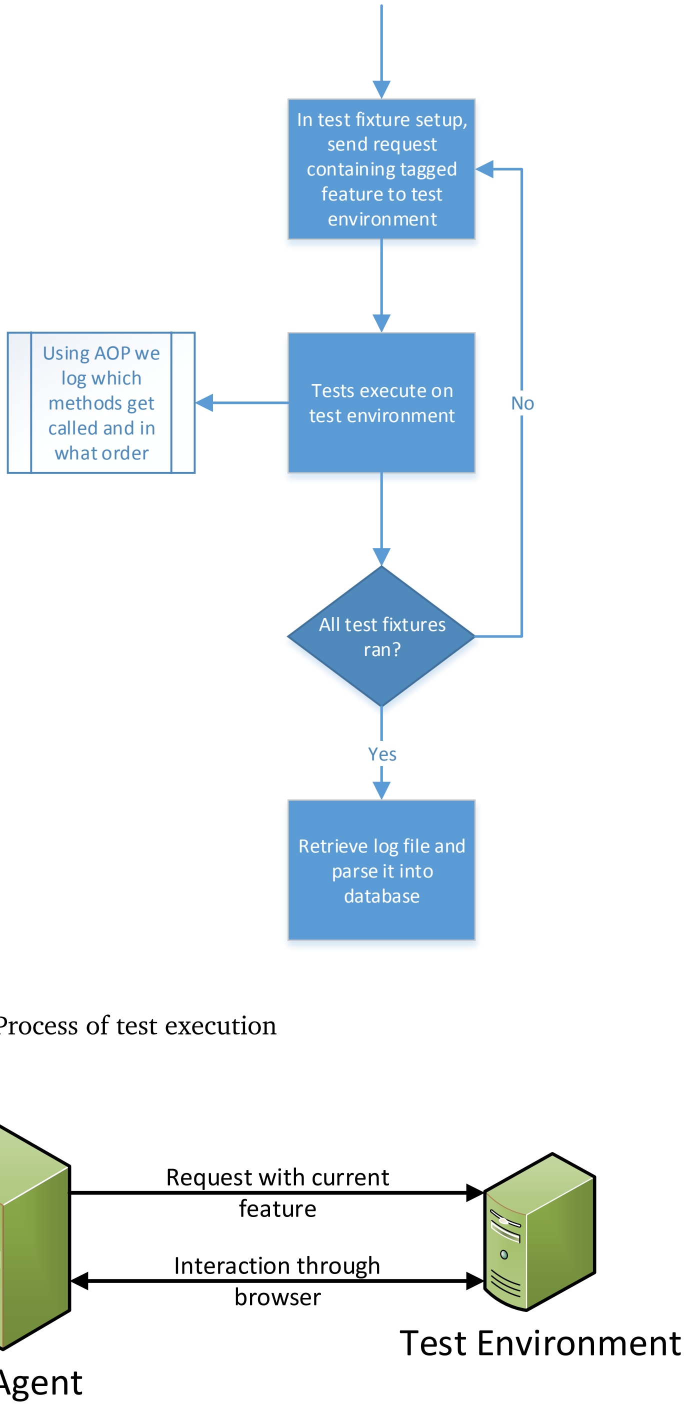
4.3 Execution Tracing

In order to associate features with source code, thus reestablishing traceability links between features and source code, we use a form of software reconnaissance, a dynamic feature location technique. Assuming tests are tagged with features, we need to run those tests and monitor which tests are reaching which methods or statements.

If we were to trace unit or integration tests, this can easily be done with existing code coverage tools like NCover [NCo16], dotCover [Jet16a] or OpenCover [Ope16]. However, since we are trying to trace functional tests in the build pipeline there are two major problems:

1. Testing code (Echo tests) and application code (the test environment) are completely separated, i.e. the Echo tests are running in a build agent and the application code is deployed in a test environment, thus somewhere else on the network. Communication between these happens only through the browser or through API calls.
2. Several test fixtures are running concurrently and we need to uniquely identify which feature is currently executed at any given point in time and at any executed line of code.

Figure 4.2a outlines the designed process which is able to solve these problems. NUnit 3 is configured to parallelize tests on fixture level. Thus, we need to do additional work only during each test fixture setup, as opposed to for each test. Since every test fixture uses its own tenant in the test environment (see section 4.2.1), the fixture starts off with transmitting the currently tested feature for this fixture and tenant to the test environment by sending an API request. With that, the application maintains a mapping from tenant to feature.


In test fixture setup, send request containing tagged feature to test environment
Using AOP we log which methods get called and in what order
Tests execute on test environment
No
All test fixtures ran?
Yes
Retrieve log file and parse it into database
Request with current feature
Interaction through browser
Build Agent
Test Environment

(a) Process of test execution

(b) Communication between build agent and test environment

Figure 4.2: Design for gathering execution traces

After that the tests of the fixture can be executed. All of them are running sequentially on the same tenant. In order to gather execution traces the application is modified using aspect oriented programming (AOP) [KLM+97]. Using this approach it is possible for us to execute custom code at any given point in the application.

In additional to the initial API request containing the current feature, the testing code and the test environment interact only through the browser (or API requests that test the API endpoints) with each other, thus with HTTP requests. This way, in the application there is always a current HTTP context object available which allows us to retrieve the current tenant as well. With that, the custom code added via AOP can query the current feature for the active tenant at any interesting point in the code and write this information to a log file.

After all test fixtures have been executed this log file can be retrieved from the test environment and analyzed. Parsing the information into a database seems reasonable as this way the data can be easily retrieved in different forms.

5 Implementation

This chapter provides insight in the implementation of the different parts of this work, starting with the taxonomy manager, the test management, going to the Selenium Grid and the execution tracing and ending with the database collection setup used for storing the tracing results.

5.1 Feature Taxonomy and Taxonomy Manager

Here the improvements to the taxonomy manager and the procedure for tagging and managing tagged tests are presented.

5.1.1 Improved Taxonomy Manager

In figure 5.1 you can see the new layout of the taxonomy manager (compare to figure 2.1) with an excerpt of the created feature model for the Recruiting product. Green nodes can be expanded by double clicking on them to reveal their children.

The Recruiting taxonomy currently has 149 nodes in total, but is continuously extended as new features are added. As you can see, during the meetings with the test architects more tags have been added to the first level in additional to the initial three (Candidates, Recruiters and Recruitment Administrators), for example, Internationalization/Translations, product integration as well as tenant management do not fit underneath one of those.

When you click on a tag in the tree it will be highlighted together with the cross-dependencies and a window appears displaying detailed information about the tag (see figure 5.2). Amongst others you can see the priority, tags that are dependent on the selected tag (also transitively dependent ones) and an NUnit 3 selector which can be used with the NUnit console runner to execute all tests related to this feature. For example, if a developer or test engineer works on a hot fix for a feature, he/she can

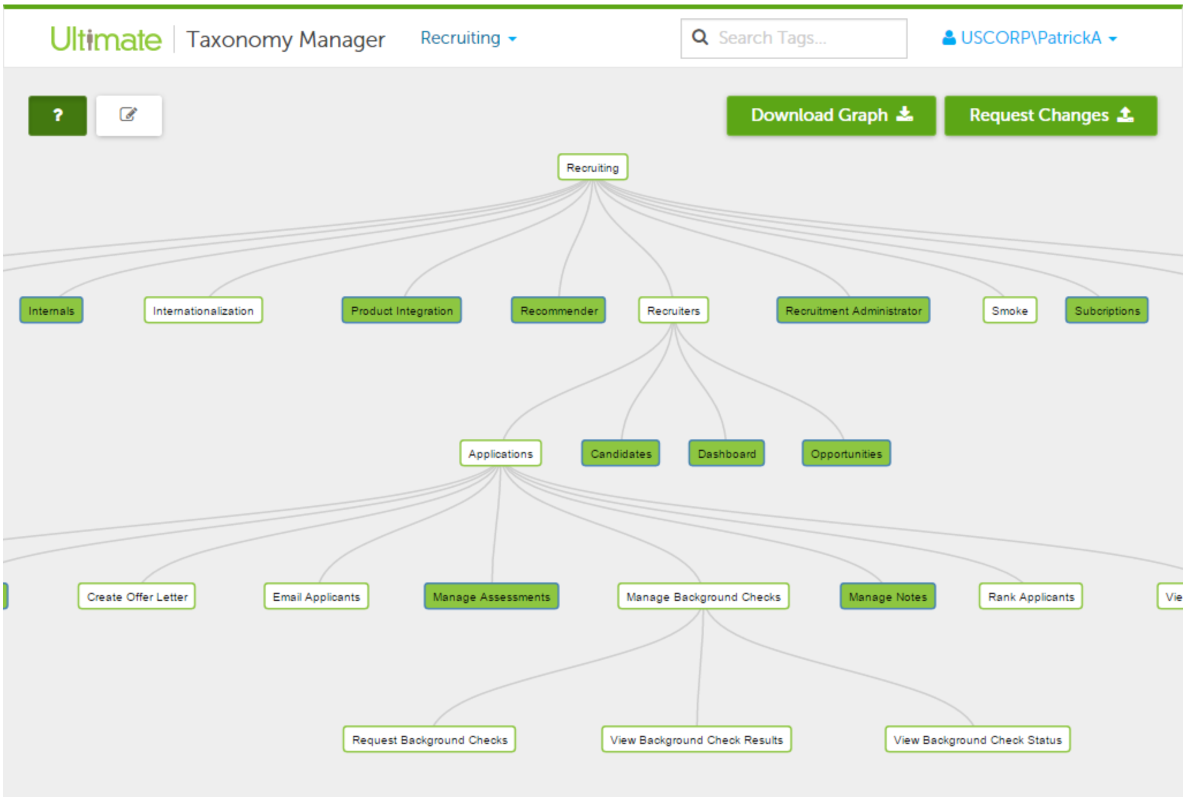
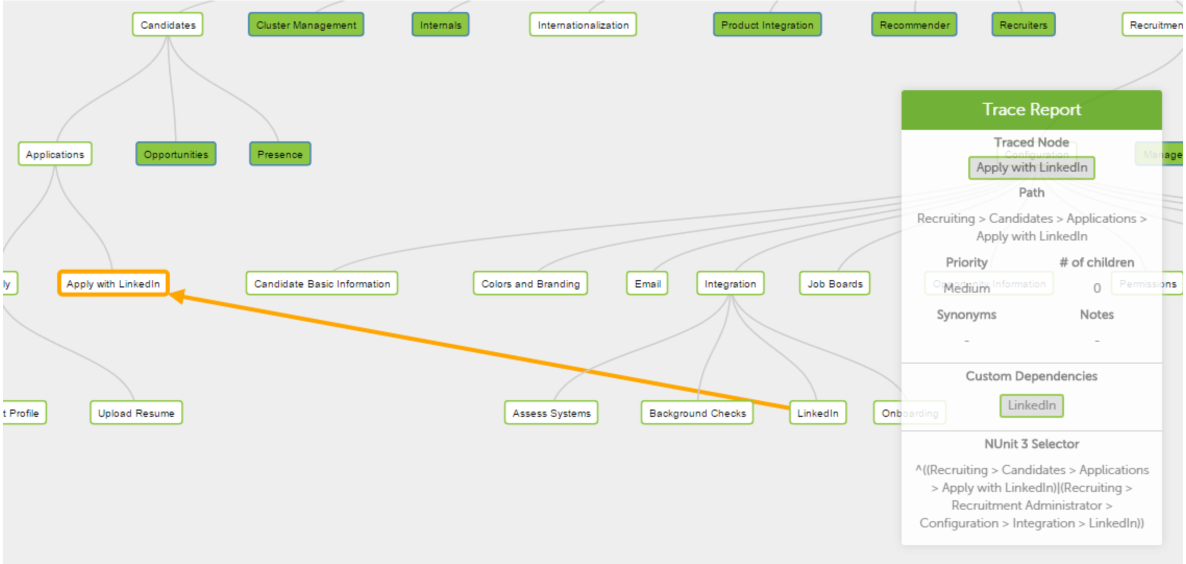


Figure 5.1: New layout of the taxonomy manager with excerpt of the Recruiting feature model



Trace Report

Traced Node

Apply with LinkedIn

Path

Recruiting > Candidates > Applications > Apply with LinkedIn

Priority	# of children
Medium	0

Synonyms	Notes
-	-

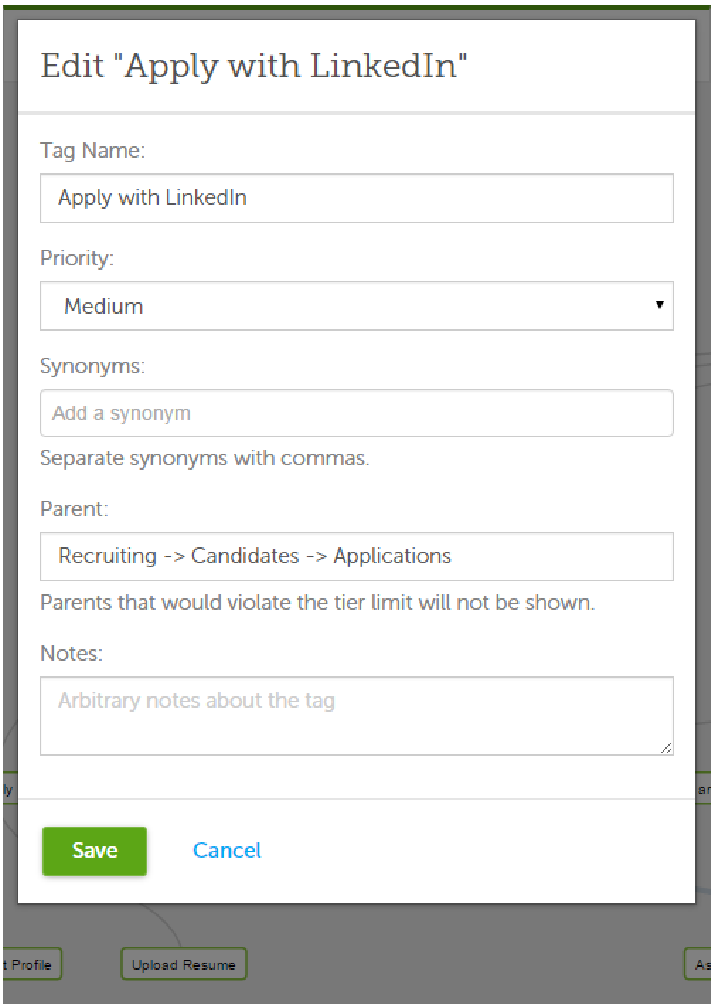
Custom Dependencies

LinkedIn

NUnit 3 Selector

^((Recruiting > Candidates > Applications > Apply with LinkedIn)|(Recruiting > Recruitment Administrator > Configuration > Integration > LinkedIn))

Figure 5.2: Displaying detailed information about a feature tag



Edit "Apply with LinkedIn"

Tag Name: Apply with LinkedIn

Priority: Medium

Synonyms: Add a synonym

Separate synonyms with commas.

Parent: Recruiting -> Candidates -> Applications

Parents that would violate the tier limit will not be shown.

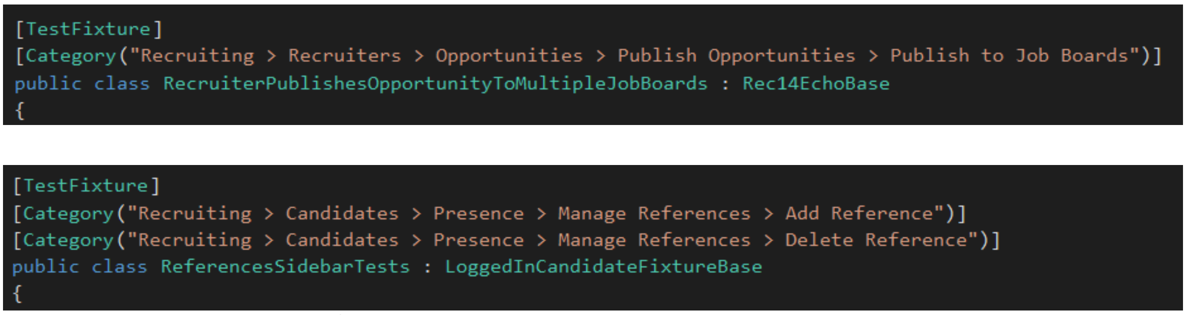
Notes: Arbitrary notes about the tag

Figure 5.3: Edit view of a tag

just copy and paste this selector which is basically a regular expression to the "filter" command line option of the test runner.

As you can see in figure 5.3 it is now possible to select a priority for each tag when editing it. A text field for optional notes was also added as this seen necessary by the test architects for documenting additional domain knowledge.

It is also worth noting that developers and test engineers can only submit requests for changes to the feature model by working with the tree and hitting the "Request Changes" button on the top right. The person responsible for the feature model can approve or deny these requests or make further modifications before approving. These persons are likely to be business analysts or domain testers. This was restricted so that the feature model does not grow without managing it.



```
[TestFixture]
[Category("Recruiting > Recruiters > Opportunities > Publish Opportunities > Publish to Job Boards")]
public class RecruiterPublishesOpportunityToMultipleJobBoards : Rec14EchoBase
{
```

```
[TestFixture]
[Category("Recruiting > Candidates > Presence > Manage References > Add Reference")]
[Category("Recruiting > Candidates > Presence > Manage References > Delete Reference")]
public class ReferencesSidebarTests : LoggedInCandidateFixtureBase
{
```

Figure 5.4: Examples of tagged tests

5.1.2 Tagging Test Fixtures and Managing Consistency

In figure 5.4 you can see the syntax for tagging test fixtures with features. The `Category` attribute which was previously used for sorting the tests into groups is now used for specifying one or more associated features. In order to assign uniquely identifiable features the whole path starting from the root node needs to be specified. Every level is separated by the "greater than" sign as it indicates a little arrow. The root node is included as well since other products might share the same domain at some point.

To ensure that every system test fixture has at least one feature assigned, a test has been added that uses reflection to scan the system test assembly for classes with the `TestFixture` attribute and validating that these classes also have at least one `Category` attribute which constructor argument starts with "Recruiting". If not, the name space and name of the classes without a feature will be displayed as part of an error message so that the developer can add it.

Another test in the same fashion is collecting all feature tags in the `Category` attributes and sending them to the taxonomy manager via an API request. The manager is validating them and returning all strings that could not be identified as valid features.

These tests are executed together with the unit tests since they take less than a second and this way missing or wrong feature tags are detected very early in the build pipeline.

5.2 Selenium Grid

The selenium grid is a `Java` application that can be distributed to multiple nodes in a network. It consists of a hub, which is managing and forwarding the incoming traffic to available worker nodes, and an arbitrary number of worker nodes that are hosting the browsers which are actually used for executing the Selenium tests.

The setup is very easy because the worker nodes are automatically connecting to the hub once they are given the corresponding IP address and port number. To get maximum performance, the hub and each worker node was setup on its own node in the companies cloud. Since the available quota of nodes is restricted for each team, we were working with a grid of ten nodes and the hub to avoid hindering development work of the team by reserving too many nodes.

Ten nodes with five worker threads each provided 50 browsers that could run tests concurrently. This setup was used by one build configuration in the TeamCity pipeline to execute all the system tests that have been previously executed by 21 build agents. By limiting the number of workers of the NUnit test runner we could specify how many of the 50 browser instances are actually used for a test run since the grid might be shared with multiple concurrent runs of the whole build pipeline. Details of timing analyses are given in chapter 6.

5.3 Code Instrumentation

As mentioned in section 4.3 aspect oriented programming is a good way for adding custom instructions without modifying the original program. There are different ways for doing that, for example by using PostSharp [Pos16] which is a compiler extension or by using Mono.Cecil [Mon16] with which it is possible to edit an already compiled assembly.

5.3.1 Implementation with PostSharp

With PostSharp you can change or add additional behavior to existing code by specifying custom attributes. For common tasks like logging or code contracts, PostSharp already offers fully implemented attributes that just need to be added to the required classes, methods or parameters.

For more customization, existing attributes can be extended in order to implement custom behavior. For example, the `OnMethodBoundaryAspect` attribute can be extended while overriding the methods `OnEntry(MethodExecutionArgs)` and `OnExit(MethodExecutionArgs)` in order to add custom code that will be executed before or after a method. The argument of type `MethodExecutionArgs` carries all necessary information about the method being called. This way, we could add logging code which writes out the currently tested feature and active tenant.

By adding this attribute `MyMethodTracer` which extends from `OnMethodBoundaryAspect` to a method in the code it will enable logging for that specific method, but these attributes can also be applied on assembly level for a whole `C#` project by adding

```
[assembly: MyMethodTracer]
```

into the `AssemblyInfo.cs` file making it very easy to enable and disable this custom behavior for every method.

However, it became clear very soon that logging on method level is not enough in order to get meaningful code coverage data. For this, it was necessary to get logging on statement level which is not possible easily with PostSharp. Thus, Mono.Cecil was selected for solving this issue.

5.3.2 Implementation with Mono.Cecil

Mono.Cecil allows for real instrumentation of a program by changing an assembly on the CIL level. CIL is the Common Intermediate Language used by the .NET framework.

As opposed to the PostSharp implementation, the solution with Mono.Cecil is a separate command line program that needs to be executed once the actual application, that is to be instrumented, was compiled in order to rewrite the desired assembly. Because of that, this solution is also very easy to switch on and off since the execution of the command line program can simply be skipped. This is important for adding this functionality to the build pipeline because we don’t want to collect execution traces every time the pipeline is triggered.

After parsing an assembly, Mono.Cecil provides a list of types/classes and a list of methods for each type found in the assembly. Given this information we can start instrumenting the code by searching the assembly instructions for the beginning of each statement in a method which is done using sequence points provided by debugging symbols in the code.

Then, before each statement we insert IL code that calls a logging function with the current sequence point and full method name. This function is then looking up the currently tested features for the active tenant by using the current HTTP context, which includes the tenant information in the URL route, and by using the tenant-to-feature map explained in section 5.3.3. Once the currently tested features have been retrieved from the map they are also stored in the current HTTP context to improve performance. This is possible because one HTTP context cannot belong to two different sets of features.

All the gathered information for each executed statement is written to a log file in the assembly. The command line program itself creates a log file about each statement instrumented. This log is later used for calculating the relative amount of statements being covered.

5.3.3 Managing the Active Features

As explained in section 4.3 every test fixture sends an API request with the currently tested features and the tenant name to the test environment with the application after the tenant was created specifically for this fixture. This API request can also easily be disabled when it is not needed because the tenant creation and the API request are happening in a common parent class of all system tests called `Rec14EchoBase` (cp. figure 5.4).

The mapping of tenant to currently tested features is stored in a separate collection in the MongoDB of the application in order to be persistent across multiple HTTP requests. This does not change or pollute the database when running the tests without having tracing enabled, either, because collections in MongoDB are only created as soon as something gets stored in them.

5.4 Database Collections and Tools for Analysis

For logging the platform NLog [NLo16] was used with a rolling archive numbering scheme so that log files could be limited in size. The build agents did not handle one big log file (several GB in size) very well while multiple smaller log files did not create any problems.

In order to analyze the collected data, the log files are first parsed into a set of three different MongoDB collections:

- A feature coverage collection which stores a data set for each method. Amongst information about how to uniquely identify the method using name space name, type name, method name and argument types it holds a list of features that were responsible for calling the correspondent method.

- A method coverage collection which carries a list of methods for each tested feature.
- A statement coverage collection which stores a data set for each statement that was executed during the test run including the correspondent method name, line number and a list of features that hit this statement.

The method coverage collection provides data about how broad a feature is using code in the application while the statement coverage collection will be able to provide total code coverage analysis. The feature coverage collection allows us to talk about how specific certain parts of the code are. For example, if there is only one feature in the list of a method that means this method was specifically written for that feature. On the other hand, if there are very many different features in this list, that method will belong to a very general component of the system.

NDepend [NDe16] is used in order to take code metrics into account. It is a powerful tool that analyzes C# projects and gives insight into the code base. Amongst others it is able to calculate the Cyclomatic Complexity [McC76] of methods and types. The Cyclomatic Complexity is one generally accepted method for determining the complexity of source code. Especially for software testing it is a useful metric for determining the number of necessary tests for a good coverage on unit level [WMW96].

NDepend provides a tool set which they call NDepend Power Tools that can be used as an API interface for programmatically retrieving e.g. code metrics like the Cyclomatic Complexity. For this Master's thesis a command line tool was written that is able to convert the unique method signatures provided by Mono.Cecil and consisting of the name space name, the type name, the method name and the arguments to the syntax used by NDepend in order to retrieve the Cyclomatic Complexity for every method in the analyzed solution. These complexities are then stored in the feature coverage collection as well.

6 Evaluation

In the following sections the improvements done to the build pipeline of Recruiting are evaluated and the code base is analyzed in regards to risk-based testing.

6.1 Improvements with Using NUnit 3 and Selenium Grid

This section evaluates the changes to the testing framework and the build pipeline for the unit, integration and especially the system tests.

6.1.1 Unit and Integration Tests

NUnit 3 is parallelizing the test execution automatically on the assembly level. For example, unit and integration tests have separate testing projects for each application project that they are testing. Thus, these tests don't interfere with each other without having to take care of that. So these test executions are parallelized automatically just by updating to NUnit 3. This way, the execution time for unit tests in the pipeline could be reduced from 5 minutes 16 seconds on average to 3 minutes and 20 seconds. The execution time of all integration tests was reduced from approximately 28 minutes to 16 minutes.

6.1.2 System Tests

Previously the system tests were executed on 21 build agents against one test environment. If all necessary build agents were idle at the beginning of the execution so that all 21 system test groups could start off at the same time, they needed around 40 minutes to complete. However, usually the builds are sitting in the queue for a while until the agents become available. Since 21 build agents were necessary, the total execution time was usually much more than 40 minutes because the agents became only gradually available.

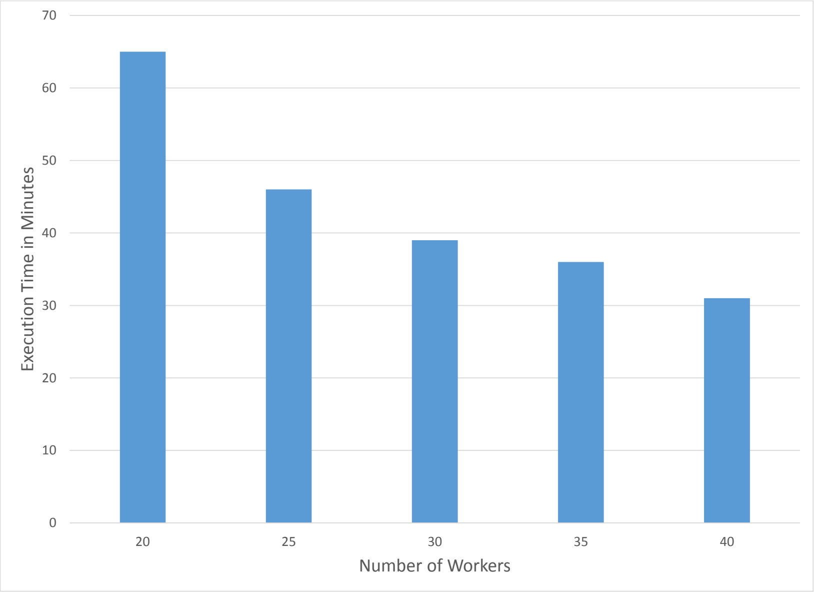


Figure 6.1: Total test execution time when changing the number of Selenium workers

Outsourcing the browser interaction greatly reduced the load on the build agent. The CPU usage of agents running the tests on the Selenium Grid was only about 30%, even if all tests are kicked off on one agent. So only a single build agent is necessary, thus the tests are kicked off as soon as one agent becomes available. The Selenium Grid is statically available and the test environment against which all the tests run is deployed in the same way as before.

Using a Selenium Grid with 10 nodes and 50 worker threads in total we can vary the number of used workers by limiting the amount of worker threads of the test runner on the build agent. Figure 6.1 displays the total execution time of all 1624 system tests over the number of used worker threads in the Selenium Grid.

When using more than 40 worker threads for the same test run, tests started failing sporadically because the load on the test environment was too high. This amount of concurrent requests on the same test environment (which usually runs in a small virtual machine) delayed responses until some tests timed out.

Using 20 worker threads would imply we should get around the same timing results as with the old system of using 21 concurrent test groups, instead it took slightly more than an hour. However, the difference can be explained because we are using a smaller number of physical machines now (10 Selenium nodes compared to 21 build agents) and also because the scheduling done by the Selenium Hub does not seem to be very good. Around the end of the execution only a very small amount of workers are still active finishing up long running test fixtures.

Using 25 worker nodes takes only slightly longer than with the previous system and only half of the grid is utilized allowing two concurrent executions of the acceptance tests. Thus, 25 is the setting that is currently in use. Increasing the number of Selenium nodes to allow more worker threads per run is currently not possible due to the quota limit. However, once the Selenium Grid was sufficiently tested by the Recruiting product team, a much larger grid will be deployed that can be used by multiple teams.

6.2 Advantage of Categorizing Tests for Developers

With tagged tests, it is very easy for developers and test engineers to select only a sub-set of all system tests when working on a feature or a hot fix. For example, if a developer fixed a reported bug of a feature and he wants to make sure that his changes did not break anything he can use the feature taxonomy to execute only those tests related to the feature before the whole pipeline has to be kicked off.

The following analyzes the amount of time saved by executing tests for a specific feature compared to executing all system tests. Figure 6.2 shows the number of tests written for specific features. Only the features with more than 30 tests are displayed to keep it clearly arranged. For these features with the most tests, the execution time is displayed in figure 6.3.

As you can see running the regression tests for the feature with the most tests needs 16 minutes which is a huge difference to the usual 40 minutes when running the whole pipeline. In addition, these tests are done so fast they could be executed locally as well in order to save the time that the pipeline needs to go through the unit tests, setting up the test environment and similar tasks (this time is not included in the 40 minutes). When working with other feature even more time is saved.

6.3 Identifying Common and Feature-Specific Methods

Using the feature coverage collection in the database after the data has been gathered, we can now categorize methods based on their specificity. Figure 6.4 shows an overview over this categorization by putting each method into one of 3 buckets: The first bucket

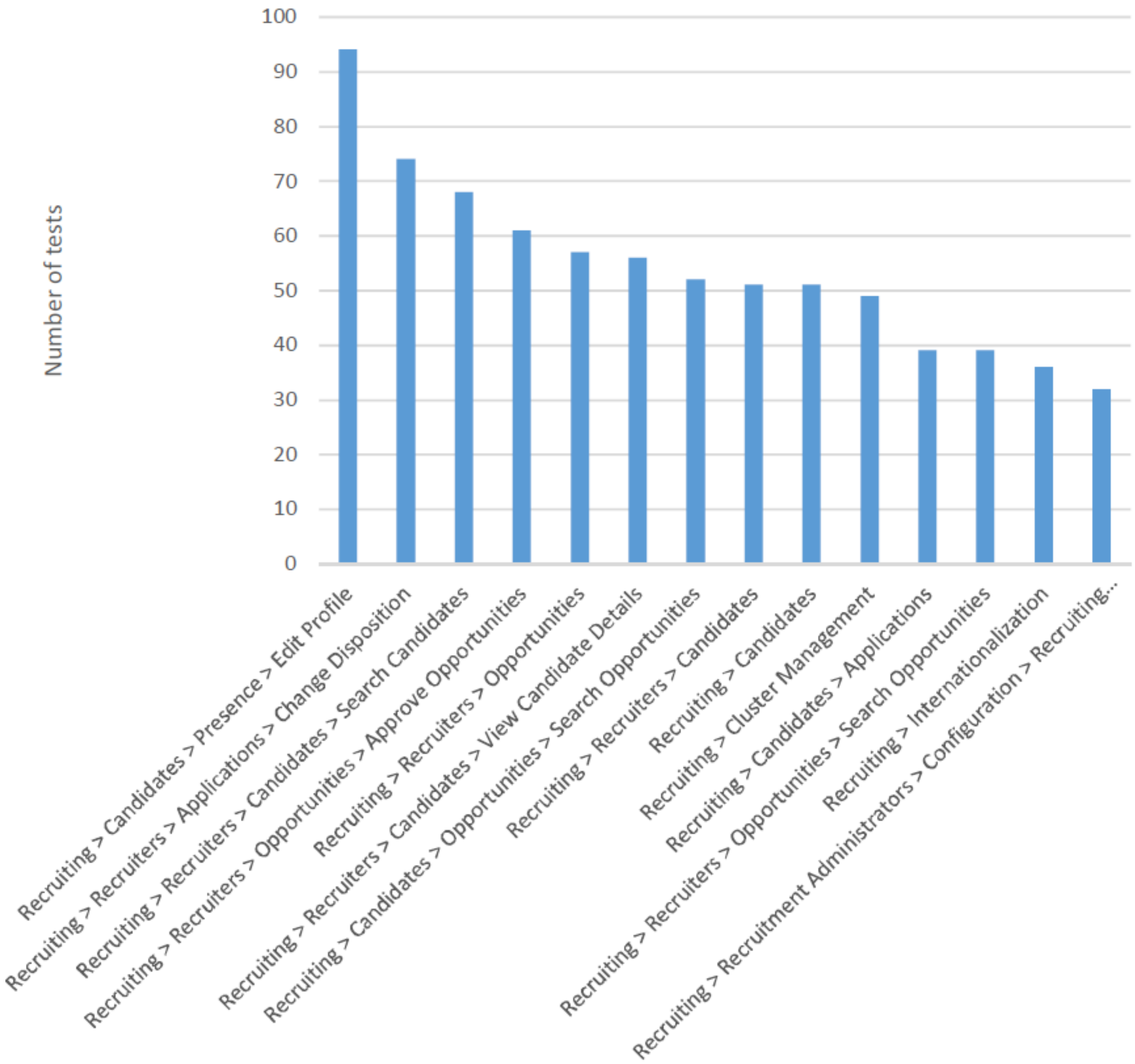


Figure 6.2: Number of tests tagged for features

contains methods that are only called for at most three different features, the second bucket contains methods that are called by 4 to 50 different features and every method that is called by more features is put into the third bucket. As you can see there are close to 400 methods that are very specific to features (mostly a sub-tree in the hierarchy), around 150 are more generic and 105 methods are called for almost all features, thus belonging to very common components.

Test engineers can use specificity of certain methods when creating tests. Methods that are commonly used are usually already well tested just because they are exercised very often. More specific methods might need more attention by the test engineers.

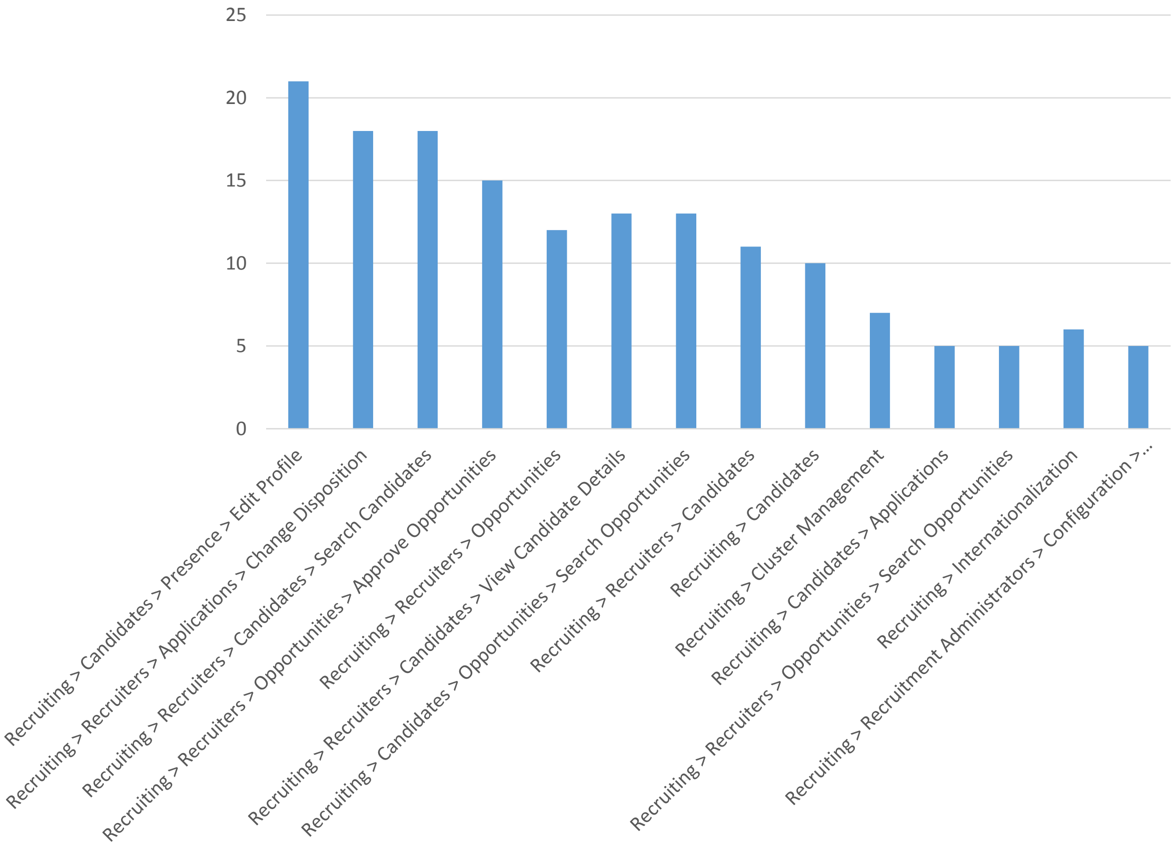


Figure 6.3: Execution time of tests for features

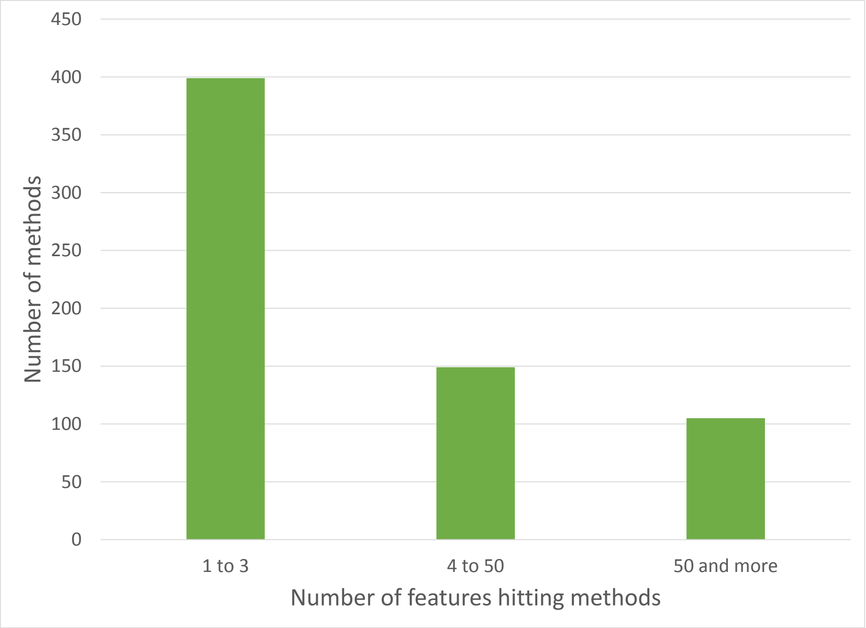


Figure 6.4: Categorizing methods based on how many features they are serving

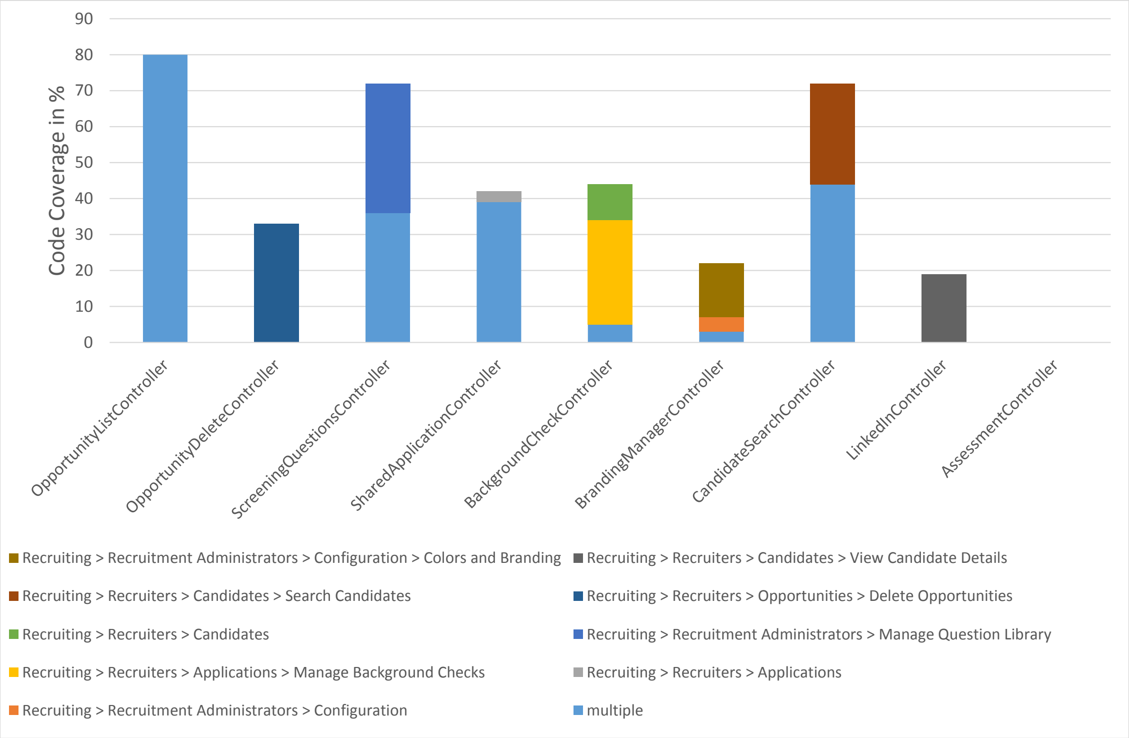


Figure 6.5: Excerpt of code coverage for individual features

6.4 Code Coverage for Individual Features

By analyzing the data of the statement coverage collection we can calculate code coverage in general and for specific features. The statement coverage collection contains an entry for every statement in the assembly that got hit by one or more features. When instrumenting the code we also collected information of all existing statements (see section 5.3.2) which we can now use to calculate the code coverage by comparing the existing statements with the ones that got actually executed using a small script.

Figure 6.5 shows only an excerpt of the data since the whole data set is too big and aggregating the data is of no use in this case. In the diagram, you can see the amount of code coverage of some of the controller classes and how much of it individual features are contributing or if the code is covered by multiple features. For example, the `OpportunityListController` is covered by 80% and all of the covered statements are hit by multiple features. This is due to the fact that this controller is providing searchable and filterable lists of opportunities that are used on different pages. In contrast, the `OpportunityDeleteController` is only hit by the "Delete Opportunities" feature. However, it is only covered by about 30%.

The `BackgroundCheckController` is covered by approximately 30% by the "Manage Background Check" feature and by 10% by "Recruiting > Recruiters > Candidates" probably because tests of the latter feature are looking at the candidates list page for recruiters which includes the results of potential background checks.

An interesting fact is that the `LinkedInController` is exclusively covered by the "View Candidate Details" feature although the feature "Recruiting > Candidates > Applications > Apply with LinkedIn" exists (see figure 5.2). An investigation showed that the corresponding tests are part of the small number of tests in the regression test suite that are marked with a special attribute which prevents execution in the build pipeline. These tests are only executed when running the system tests locally because they test the integration with LinkedIn sending actual data to the LinkedIn API using a test account. To prevent flooding LinkedIn with test data, these tests are reserved for local execution. The small percentage of coverage with the other feature is caused by the fact that this controller is also handling the creation of a link to LinkedIn on a candidate's detail view.

The `AssessmentController` is an example for a class that is not yet covered at all by automated system tests.

6.5 Risk-Based Testing and Minimizing Test Sets

As explained in section 2.2 risk-based testing tries to focus testing efforts on parts of the system with the highest risk. Now that we successfully reestablished traceability links between features and source code, we can take this information into account when determining risk of features.

For that we look at some high priority features and some low priority features of Recruiting. While creating the feature taxonomy of Recruiting the following features have been identified as crucial for the product:

- Recruiting > Candidates > Opportunities > Search Opportunities
- Recruiting > Candidates > Applications > Apply
- Recruiting > Recruiters > Applications > Change Disposition

since searching for job opportunities and applying for them is the core functionality of Recruiting. The same holds for the possibility for recruiters to move potential candidates through the recruiting process which is called "changing dispositions" in the application and the domain.

In contrast, the following features have been tagged with a low priority:

- Recruiting > Candidates > Presence > Manage References
- Recruiting > Recommender > Provide Recommendation
- Recruiting > Recruiters > Dashboard

Managing professional references and the possibility for external people to submit recommendations for candidates are considered as low priority because they are not used frequently. The only feature currently available on the dashboard of recruiters is the presentation of very basic statistics. This is seen with a low priority because the analysis of data is possible with the much more powerful reporting capabilities.

Most features of the Recruiting application are currently categorized with medium priority.

In the following the low and high priority features will be abbreviated by just using their leaf node name. In table 6.1 you can see these features together with the number of test cases tagged with them, their complexity and the code coverage. The values are taken from the analysis of the code project containing the presentation layer of the application. For the complexity we use the sum of the Cyclomatic Complexity values of all called methods of the feature as a simple metric (depending on the use case you could use other metrics for complexity here as well). The code coverage value depicts the code coverage of methods that were executed solely by the corresponding feature. Common methods that are executed for multiple features are not taken into account for this value.

The "Search Opportunities" feature has 52 test cases and a complexity value of 433, the "Change Disposition" has a much higher complexity value and also more tests. However, the "Apply" feature which is also a high priority feature has the second highest complexity value in the table, but only 6 test cases. Thus, this feature is considered a high risk item and it requires much more testing.

On the other hand, the low priority feature "Manage References" has a very high number of tests (it does not appear in figure 6.2 because these tests have been associated with children of the tag), but a lower complexity value. Thus, the number of tests could probably be reduced without increasing the risk.

Feature	Priority	# of tests	Complexity	Coverage
Search Opportunities	High	52	433	71%
Apply	High	6	519	71%
Change Disposition	High	74	1166	68%
Manage References	Low	72	424	68%
Provide Recommendation	Low	11	379	71%
Dashboard	Low	20	354	64%

Table 6.1: Risk heuristics for Recruiting taking the code base into account (metrics from the presentation assembly)

Feature	Priority	# of tests	Complexity	Coverage
Search Opportunities	High	52	377	76%
Apply	High	6	409	76%
Change Disposition	High	74	664	67%
Manage References	Low	72	363	78%
Provide Recommendation	Low	11	299	76%
Dashboard	Low	20	274	70%

Table 6.2: Risk heuristics for Recruiting taking the code base into account (metrics from the domain assembly)

In table 6.2 you can see the same analysis for values from the domain assembly. The domain assembly contains the code project that is implementing the business domain logic. The code coverage seems to be slightly higher in general, only code responsible for changing dispositions has considerably less coverage which should be improved by the test engineers.

6.6 Other Results

The total code coverage from the system tests is approximately 64% for both assemblies. A code coverage as high as with unit tests (86% for Recruiting) cannot be achieved with system tests without exceptionally more effort because the system cannot be controlled as easily from the system level and certain error cases that are checked on unit level are highly unlikely to occur and they are thus not tested on system level. Especially in Recruiting there are very many checks for write results from the database or results from mappers that transform data transfer objects into `C#` objects that are mostly not tested on system level which explains this low coverage value.

Calling logging methods for every statement in an application adds a substantial amount of load onto the test environment. When instrumenting several assemblies of the product, the application is slowed down to a degree where not all of the requests could be answered in time anymore which resulted in test failures due to time outs. Thus, for the evaluation above only the assembly containing the presentation layer was instrumented (unless otherwise noted) which contains, amongst other things, the controllers, data transfer object mappers, validators and input filters. The implementation with PostSharp was able to handle a few assemblies, but provides less data.

Automated tests for ensuring and validating feature tags turned out to be substantial because developers tended to be very sloppy with these things. However, one problem could not be solved: When developers or test engineers added new system tests they sometimes based them on existing tests by copying/pasting the general structure of an existing test into a new file. This way, they copied the existing feature tags as well without paying attention on whether they need to be replaced for the new test. No automated approach could detect this since a valid feature tag is already set.

7 Conclusion and Future Work

The following concludes this Master's thesis by summarizing the work and the results and gives an outlook to future work that can still be done based on the thesis at hand.

7.1 Conclusion

This work demonstrated how using a feature model and reestablishing traceability links between features and source code can help the testing process for enterprise software applications.

By using a newly developed system for tracing code execution even if testing code and application code are running on different machines on the network, source code can be connected to features in the domain. As it was shown we can use this information in order to take code metrics into account when assessing risks in a software application. Thus, high risk items that need more testing or more regression tests can be identified. On the other hand, low risk items can be revisited and some tests may be removed in order to reduce costs or time of regression testing.

Enabling developers and test engineers to select sub-sets of tests when working on a specific feature greatly improves time needed for running regression tests. This was accomplished using the tagged tests and changing the functional tests to use a Selenium Grid. Other teams will also benefit from the creation of the Selenium Grid setup and the development of best practices for working with it. Thus, the costs for regression testing of multiple teams can be reduced.

It became clear that improving testing is hard work and needs cooperation from everybody in the development team. Enforcement in form of automated tests can help with reminding developers of necessary tasks, but unfortunately not everything can be caught with tests.

It is also important to note that creating the feature taxonomy for Recruiting, a medium-size industry software application, could be done in only four one-hour meetings. Thus, the expenditure of time is very well invested compared to the gain that results from it. However, it is essential for this that everybody involved in creating the feature model has extensive domain knowledge. In any way, this should convince other software development teams to invest in managing a feature model for their product. At Ultimate Software, the taxonomy manager will soon be used by other teams to create and manage their feature models.

7.2 Future Work

As shown selecting sub-sets of tests fixtures for running regression tests reduces costs and time. The next step would be to automate the test selection by comparing different versions of the program like in [Sch09] to find changed parts of the code. This differential can then be used together with the traceability links between features and source code to automatically find features that are affected by the changes. After that, tests based on these features can be executed. Due to the abstraction to features, this would execute even those tests that have been added to the test suites after the traceability links were created.

For the risk analysis, further metrics can be taken into account now that the connection between features and source code is established. For example, the code coverage on unit and integration level can be considered additionally when assessing risk items. Furthermore, the testing pyramid [Coh09] can efficiently be improved by pushing tests from the system level down to the unit level.

The performance of the execution tracing needs to be improved in order to be able to instrument several assemblies at once without affecting the test execution with failing tests. In addition, so far the execution tracing has only been implemented for C# code, but the general approach is portable to any programming language that can be instrumented. Especially, execution tracing for JavaScript would be important for the Recruiting product as the client side is based on the JavaScript library Knockout.

As detailed in section 2.5, Software Reconnaissance is not the only feature location technique. By augmenting the current dynamic approach with static or textual feature location approaches more insight into the connection between features and source code could be gained.

The Selenium Grid which was set up for the Recruiting product needs to be increased in size and made available for other teams. A bigger grid greatly improves performance of the acceptance tests. Also, more build agents would be available for running lower level tests faster because the builds don't spend as much time in the waiting queue anymore.

Finally, another important aspect is visualizing the code coverage results from the system tests directly in the development environment like it is usually available for unit and integration tests. This way, test engineers can more easily see which parts of the code may require additional testing.

Bibliography

[Aml00] S. Amland. Risk-based testing :: Risk analysis fundamentals and metrics for software testing including a financial application case study. *Journal of Systems and Software*, 53(3):287 – 295, 2000. (Cited on page 16)

[Bac99] J. Bach. Heuristic risk-based testing. *Software Testing and Quality Engineering Magazine*, 11(9), 1999. (Cited on page 16)

[BCC88] P. Benedusi, A. Cmitile, U. D. Carlini. Post-maintenance testing based on path change analysis. In *Software Maintenance, 1988., Proceedings of the Conference on*, pp. 352–361. IEEE, 1988. (Cited on page 26)

[Bei95] B. Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., 1995. (Cited on page 16)

[BSRC10] D. Benavides, S. Segura, A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010. (Cited on page 18)

[Cod16] CodeCover – an open-source glass-box testing tool. `https://codecover.org`, 2016. (Cited on page 26)

[Coh09] M. Cohn. The Forgotten Layer of the Test Automation Pyramid. `http://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid`, 2009. (Cited on page 56)

[CPS02] Y. Chen, R. L. Probert, D. P. Sims. Specification-based regression test selection with risk analysis. In *Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research*, p. 1. IBM Press, 2002. (Cited on page 27)

[DRGP13] B. Dit, M. Revelle, M. Gethers, D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013. (Cited on page 18)

[DRP99] E. Dustin, J. Rashka, J. Paul. *Automated software testing: introduction, management, and performance*. Addison-Wesley Professional, 1999. (Cited on page 22)

[EAAG08] M. Eaddy, A. V. Aho, G. Antoniol, Y. Guéhéneuc. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. pp. 53–62, 2008. (Cited on page 19)

[EFW01] I. K. El-Far, J. A. Whittaker. Model-Based Software Testing. *Encyclopedia of Software Engineering*, 2001. (Cited on page 18)

[EK01] T. Eisenbarth, R. Koschke. Derivation of feature component maps by means of concept analysis. 2001. (Cited on page 19)

[EKS03] T. Eisenbarth, R. Koschke, D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 2003. (Cited on page 19)

[EV05] A. Eisenberg, K. Volder. Dynamic feature traces: Finding features in unfamiliar code. pp. 337–346, 2005. doi:10.1109/ICSM.2005.42. (Cited on pages 19 and 27)

[GHK$^+$01] T. L. Graves, M. Harrold, J. Kim, A. Porter, G. Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(2):184–208, 2001. (Cited on page 26)

[Gro11] T. O. Group. *Risk Management - The Open Group Guide*. 2011. (Cited on page 16)

[HR90] J. Hartmann, D. Robson. RETEST-development of a selective revalidation prototype environment for use in software maintenance. In *System Sciences, 1990., Proceedings of the Twenty-Third Annual Hawaii International Conference on*, volume 2, pp. 92–101. IEEE, 1990. (Cited on page 26)

[Jet16a] JetBrains. dotCover – .NET Unit Test Runner and Code Coverage Tool. `https://www.jetbrains.com/dotcover/`, 2016. (Cited on page 33)

[Jet16b] JetBrains. TeamCity – Powerful continuous integration out of the box. `https://www.jetbrains.com/teamcity/`, 2016. (Cited on page 22)

[JH11] Y. Jia, M. Harman. An Analysis and Survey of the Development of Mutation Testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011. (Cited on page 12)

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, DTIC Document, 1990. (Cited on page 18)

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin. *Aspect-oriented programming*. Springer, 1997. (Cited on page 35)

[Leh11] M. Lehto. The concept of risk-based testing and its advantages and disadvantages. `https://www.ictstandard.org/article/2011-10-25/concept-risk-based-testing-and-its-advantages-and-disadvantages`, 2011. (Cited on page 16)

[LW89] H. Leung, L. White. Insights into regression testing [software testing]. In *Software Maintenance, 1989., Proceedings., Conference on*, pp. 60–69. 1989. (Cited on page 15)

[McC76] T. J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976. (Cited on page 44)

[Mon16] Mono.Cecil – a library to generate and inspect programs and libraries in the ECMA CIL form. `https://github.com/jbevain/cecil`, 2016. (Cited on page 41)

[MSR04] A. Marcus, A. Sergeyev, V. Rajlich. An information retrieval approach to concept location in source code. 2004. doi:10.1109/WCRE.2004.10. (Cited on page 19)

[NCo16] NCover – .NET Code Coverage for .NET Developers. `https://www.ncover.com/`, 2016. (Cited on page 33)

[NDe16] NDepend – Improve your .NET code quality with NDepend. `http://www.ndepend.com/`, 2016. (Cited on page 44)

[NLo16] NLog – flexible & free open-source logging for .NET. `http://nlog-project.org/`, 2016. (Cited on page 43)

[Ope16] OpenCover – A open-source code coverage tool for .NET 2 and above. `https://github.com/OpenCover/opencover`, 2016. (Cited on page 33)

[Ost02] T. Ostrand. White-Box Testing. *Encyclopedia of Software Engineering*, 2002. (Cited on page 15)

[PNV+16] C. Poole, J. W. Newkirk, A. A. Vorontsov, M. C. Two, P. A. Craig. NUnit – The most popular and widely used unit testing framework for .NET. `http://nunit.org`, 2016. (Cited on page 31)

[Pos16] PostSharp – the #1 pattern-aware extension to C# and VB. https://www.postsharp.net/, 2016. (Cited on page 41)

[PPBC16] C. Poole, R. Prouse, S. Busoli, N. Colvin. NUnit 3.0 – a unit-testing framework for all .Net languages. https://github.com/nunit, 2016. (Cited on page 31)

[Rec16] Solution Feature – Recruitment: Acquire Top Talent. http://www.ultimatesoftware.com/UltiPro-Solution-Features-Recruitment, 2016. (Cited on page 23)

[RH96] G. Rothermel, M. Harrold. Analyzing regression test selection techniques. *Software Engineering, IEEE Transactions on*, 22(8):529–551, 1996. (Cited on page 26)

[RM02] M. Robillard, G. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. 2002. (Cited on page 19)

[Rot96] G. Rothermel. *Efficient, effective regression testing using safe test selection techniques*. Ph.D. thesis, Clemson University, 1996. (Cited on page 26)

[Sch09] S. Schumm. *Praxistaugliche Unterstützung beim selektiven Regressionstest*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2009. (Cited on pages 26 and 56)

[Sel16a] Selenium – Web Browser Automation. http://www.seleniumhq.org/, 2016. (Cited on page 33)

[Sel16b] Selenium Grid – Outsourcing Browser Execution. https://github.com/SeleniumHQ/selenium/wiki/Grid2, 2016. (Cited on page 33)

[Ult16] Ultimate Software – HR Software & HR Payroll Solutions for Human Capital Management. http://www.ultimatesoftware.com/, 2016. (Cited on pages 7, 10 and 12)

[WGAL13] S. Wang, A. Gotlieb, S. Ali, M. Liaaen. Automated Test Case Selection Using Feature Model: An Industrial Case Study. In *Model-Driven Engineering Languages and Systems*, volume 8107 of *Lecture Notes in Computer Science*, pp. 237–253. Springer Berlin Heidelberg, 2013. (Cited on pages 9, 10, 25 and 27)

[WGHT99] E. W. Wong, S. S. Gokhale, J. R. Horgan, K. S. Trivedi. Locating program features using execution slices. pp. 194–203, 1999. (Cited on page 19)

[WMW96] A. H. Watson, T. J. McCabe, D. R. Wallace. Structured testing: A testing methodology using the cyclomatic complexity metric. *NIST special Publication*, 500(235):1–114, 1996. (Cited on page 44)

[WS95] N. Wilde, M. C. Scully. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995. (Cited on page 27)

[Yao01] A. Yao. CVSSearch: Searching through source code using CVS comments. 2001. (Cited on page 19)

All links were last followed on January 11, 2016.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

____

place, date, signature