UNIVERSITÄT MANNHEIM

This thesis's title

Subtitle

Term Paper

submitted: January 2004

by: Martin Mustermann
born March 28th 1984
in Eberbach

Student ID Number: 012345

University of Mannheim
Chair of Business Administration and Information Systems
D – 68131 Mannheim
Phone: +49 621-181-1691, Fax +49 621-181-1692
Internet: http://heinzl.bwl.uni-mannheim.de

Contents

List of Figures

List of Tables

1. How to use the template

Introduction

2. Software Reuse

Douglas McIlroy (McIlroy, 1968) envisaged in the 1960s a software system composed of already existent components. Where you basically put different components, that already exist, together in order to form the system is being built. Although the IT industry has tried for many years to improve the speed and reduce cots of software development by reusing components, the McIlroy's vision is still the exception rather than the rule.

Although the definition of software reuse seems trivial, it is possible to find several definitions in the literature. However, most of them are similar to the definition proposed by Krueger & W. (1992):

"Software reuse is the process of creating software systems from existing software rather than building software systems from scratch"

Since we already have a formal definition of software reuse, the next question would be: What can be reused?. To this regard, the work of Frakes & Terry (1996) defined a list 2 of potentially reusable software artifacts:

However, reuse traditionally means the reuse of code fragments and components Mili (2002). When we talk about components, we mean any cohesive and compact unit of software functionality with a well defined interface (Hummel et al., 2008). Therefore a component can be simple classes or even web-services or Enterprise Beans.

1. architecture	6. estimates (templates)
2. source code	7. human interfaces
3. data	8. plans
4. designs	9. requirements
5. documentation	10. test cases

Table 2.1.: Potentially reusable aspects of software projects according to Frakes & Terry (1996).

Software reuse emerges as a solution for the so called software development crisis (Kim & Stohr, 1992). Organizations face several problems in software development including increased costs, delayed schedules, unsatisfied requirements, and software professional shortage. Therefore, by reusing software components projects can reduce time-to-market, lower development costs, and increase software quality Frakes & Kyo Kang (2005).

2.1. Benefits of Software Reuse

Software reuse has not failed to attract the attention of industry due to its alleged benefits. In the industry the need of reduction of redundancies, as well as costs reduction and quality improvements is perceived. Moreover, the vision of fostering innovation and market penetration due to shorter production cycles promised obvious strategic business advantages (Bauer & Vetro', 2016). Research literature has reported benefits from successful adoption of software reuse:

- Lower cost and faster development:
- Higher quality:
- Standardized architecture:
- Risk reduction:

Unfortunately the adoption of suitable reuse strategy is pretty challenging as it takes place in a multifaceted environment and, thus, incorporates aspects ranging from technical to organizational at different level of abstractions (Bauer & Vetro', 2016).

2.2. Challenges of Software Reuse

Although software reuse brings many benefits, it has failed to take off. Find the right third-party software component to be reused based on a well-defined specification is one of the most challenging approaches because it requires a clear-cut matching of the potential reuse candidate and the given specification. Although there are several search tools available, most of them are still text-based and it neither reflect nor support the need to match the reuse candidates with the syntactic and semantic characteristics of a specification (Hummel & Janjic, 2013). Hummel & Janjic (2013), based on software retrieval literature, identified fours problems for implementing a sustainable reuse repository.

- Repository problem (Seacord, 1999): This reuse repository should create and maintain a big enough software collection to provide promise search.
- Representation problem (Pole & Frakes, 1994): The repository should represent and index its content in a way that makes it easily accessible.
- Usability problem (Garcia et al., 2006): The repository should permit characterizing a desired component with reasonable effort and precision.
- Retrieval problem (Prieto-Díaz & Freeman, 1987): The repository should execute the queries with high precision to retrieve the desired component.

Although the two first challenges have been address in the last years, the last two have not been addressed completely. By the rise of open-source movement and the improvements in the Internet connectivity, software developers have got access to vast swathes of free software, thus the problem of sources of components is not problem any more. Furthermore, with the advances in database and text search technologies code-search engines have made the creation of *Internet-scale* software repositories wherefore this repository problem can be regarded as solved (Hummel & Janjic, 2013). For the second problem, clever ranking approaches such as the ranking proposed by Inoue et al. (2005) which ranks those components higher in the result list of a search that are more often used than others amongst the indexed files. Moreover, the idea of parsing source code in order to extract objects and their method introduced by Koders.com, where techniques that addressed the representation problem. However, the last two problems remain still in the focus of interest in the research community.

Beside the technical problems described above, organizational challenges and human factors have been identified as potential inhibitors to a successful implementation of reuse practices (Morisio et al., 2002). Business strategy, management commitment, and company culture are organizational factors that might affect software reuse (Standish, 1984). Moreover, technical problems are strongly related to human factors, such as cognitive efforts, program understanding, and motivation. In the work of Bauer & Vetro' (2016) the following organizational obstacles were identified in the literature:

- Organization structure: Factors like competition, overlapping or unclear responsibilities, priority conflicts, and lack of coordination of reuse activities jeopardize the successful of reuse implementation.
- Inertia: Some organizations usually assess their managers and developers based on the success of their isolated projects, which incentives local optimization which impact reuse in a company-wide scale.
- Knowledge: If an organization plans to implement reuse practices, it needs clear position organization-wide and research into current methods and techniques for reuse.
- Measurement:
- Management: As implementing reuse practices require changes in governance strategies, it causes additional overhead that tends to be underestimated in the initial planning, which put at risk successful of reuse.
- Economic: Implementation of reusability requires investment and long-term support from management to resolve restrictive resource constraints.
- Disincentives: The quality of the reusable component candidates is one of the strongest disincentives. Moreover the criteria applied to measure developers and managers have an impact on their motivation to take part into reuse.

2.3. Code-search engines

Code-search engines are the heart of the new generations of reuse support tools. For example, Code Conjurer (Hummel et al., 2008) uses Merobase component search engine, CodeGenie relies on Sourcerer (Lemos et al., 2007) or ParseWeb that works over Google Code Search [1] (Thummalapenta & Xie, 2007). Next we will describe some code search engines that are still online the time of this work.

[1]Shutdown January, 15 2012 (Horowitz, 2011)

- searchcode[2] searchcode is a free source code search engine. Code snippets and open source (free software) repositories are indexed and searchable. Most information is presented in such a way that you shouldn't need to click through, but can if required.
- NerdyData[3] It is a search engine for source code. It supports HTML, Javascript and CSS. With this search engine you can retrieve the web pages which are using a specific file. For example if you look up for *font-awesome.min.css* it will retrieve the sites that are using this file.
- Spars-J[4] It is a keyword and name matching code search engine on open source XML, Java and JSPs based on component rank and keyword rank algorithm.
- ComponentSource[5] It is a keyword-matching of component descriptions in a marketplace. Here you can find components for different platforms and technologies.
- Merobase[6] Merobase is a component search engine that uses Lucene [7] to index programming language units from various open source repositories (such as Sourceforge, Google Code, or the Apache projects) and the open Web. Merobase, when crawling the code, its analysis software identifies the basic abstraction implemented by a module and stores it in a language agnostic description format. This description includes the abstraction's name, methods names, parameter signatures, among others.

Although Google or Yahoo! are not specialized code search engines, they are able to return more precise results than code search engines(Hummel et al., 2007).

2.3.1. Merobase

Merobase contains special parsers for each supported programming languages (currently it supports Java, C++, and C#, also it supports WSDL files, binary

[2]https://searchcode.com/ (accessed: 13.01.2017)
[3]https://nerdydata.com (accessed: 13.01.2017)
[4]http://sel.ist.osaka-u.ac.jp/SPARS/ (accessed: 13.01.2017)
[5]https://www.componentsource.com/ (accessed: 13.01.2017)
[6]http://www.merobase.com/ - Official project deprecated, we use a local implementation
[7]Apache LuceneTM is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform (Foundation, 2017).

Java classes from JARs and .NET binaries), which extracts syntactical information, stores it in the index and search for it later. Moreover, it contains a special parser for JUnit which is able to extract the interface of the class under test from test cases.

Every time a user make a request to Merobase, the parsers described above are invoked and try to extract as much syntactic information from the query as possible. If none of the parsers recognizes parsable code in the query, it executes a simple keyword search. Based on parsed syntactic information, Merobase supports retrieval by class and operation names, signature matching and by matching the full interface of classes described before.

When a JUnit test case is submitted, a test-driven search is triggered. In this case, Merobase automatically tries to compile, adapt and test the highest ranked candidates. If a candidate is relying on additional classes, the algorithm uses dependency information to locate them as well. It is important to point out that the actual compilation and testing are not carried out on the search server itself, but on dedicated virtual machines within sandboxes. This is due to the fact that this ensures that the executed code does not have the possibility to do anything harmful to the user's system or bring the whole testing-environment down (Hummel & Janjic, 2013).

The current implementation of Merobase, which will we used in our prototype, has been enhanced in several way since the implementation described above (Kessel & Atkinson, 2016). Many new metrics are measured on the software components, both statically and dynamically, when the test cases executed on them. Also, the semantic retrieval of components has been improved by adding a better parser for Java components which supports the creation of entire class hierarchies. With this, information about the components' class hierarchies can be retrieved and stored.

On the other hand, the current implementation is populated with software components extracted from Maven projects mainly extracted from the Maven Central Repository[8].

[8] https://repo1.maven.org/maven2/ (accessed: 15.01.2017)

2.4. Component retrieval techniques

As we stated in 2.2, several problems of software reuse have been already addressed. Now one of the big problems is how to choose the so-called best component. In this section we will talk about the different component retrieval techniques proposed in the literature. Then we will explain further what test-driven component retrieval means.

Mili et al. (1998) divided the component retrieval techniques in six independent groups:

- Information retrieval methods: It is basically the methods of information retrieval used for retrieving candidates by applying textual analysis to software assets. This group has high recall and medium precision.
- Descriptive methods: These methods add additional textual description of assets such as keyword or facet definition, to the textual analysis. These are high precision and high recall.
- Operational semantic methods: These methods use sampling of assets to retrieve candidates. They have very high precision and high recall.
- Denotational semantic methods: These methods use signature methods for retrieving candidates. The have very high precision and high recall.
- Structural methods: These methods deal with the structure of the components to retrieve candidates. This group has very high precision and recall.
- Topological methods: This is an approach to minimize the distance between query and reusable candidates. It is difficult to estimate or define recall and precision for these methods (Mili et al., 1998).

Although Mili et al. (1998) proposed six groups, the last one relies on a *measurable* retrieval techniques so it can be considered as an approach for ranking the candidate components of a query (Hummel et al., 2007).

In the work Hummel et al. (2007) several retrieval techniques were compared. They compared signature matching, text-based, name-based and interface-driven. These techniques were tested using Merobase search engine. The result they obtained was that the best technique is interface-driven in terms of precision.

Another technique that was presented in Hummel & Atkinson (2004), is test-driven search. This technique ruses test cases as a vehicle to retrieve the component candidates that fit better the requirements of the user. It promises to improve the precision of the searches in comparison with other techniques explained before. Below we will explain with further details how this techniques works.

2.4.1. Test-driven search

Simple techniques for component retrieval such as keyword-driven search or signature-driven search may lead to a tons of candidates from which just a small group might be interesting for the user, in spite of the results fulfill the search criteria. This is due to the fact that not all of them fulfill the functional properties of the desire artifact. Here is were test-driven search comes to light.

Test-driven reuse approach was first introduced in Hummel & Atkinson (2004). It is a technique that emerges as a solution to the problem of retrieving so many irrelevant component candidates due to large amount of component in a repository. Specifically this technique deals with the usability and retrieval problems we explained before. It relies on test cases written by user, extract the different interfaces defined in the test case, then it makes the search of reuse candidate, then it run the tests and *cleans* the result set of candidates that do not pass the test. Finally it shows the cleaned result set to the user. This is the test-driven reuse cycle which can be seen in the figure 2.1.

In the first step, the user provides a JUnit test case of the desired component. Then

2.5. Ranking components

2.5.1. SOCORA

The ranking component approach presented in Kessel & Atkinson (2016) works as follow. First, it establishes a partial ordering of candidates using non-dominated sets determined by non-dominated sorting without assuming any preferences of the user. Depending on the priorities assigned to each selected criterion, it then

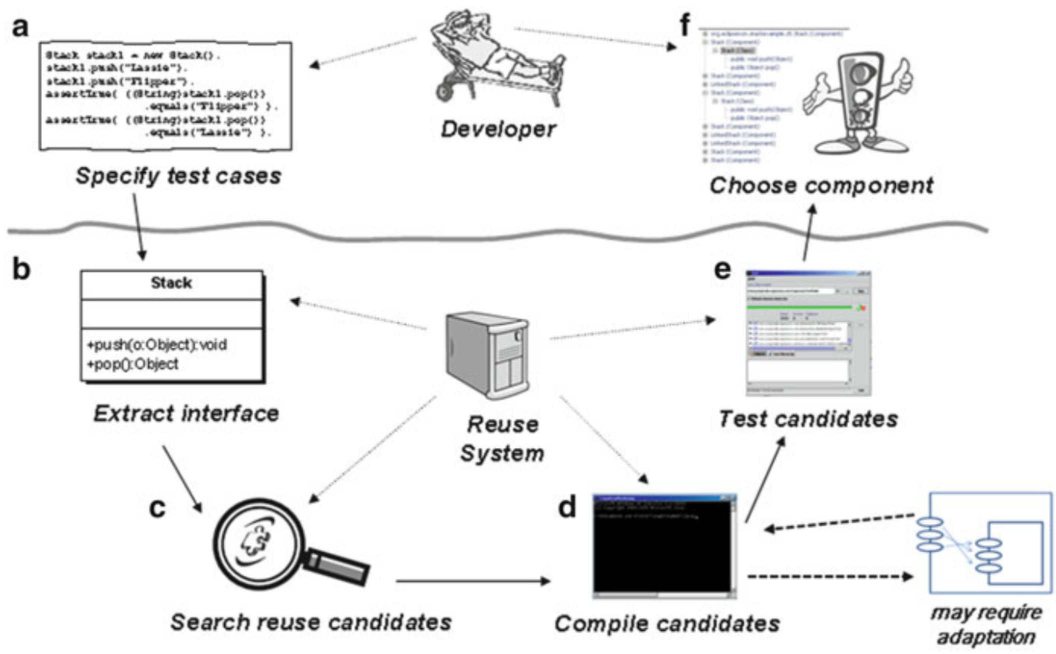


Figure 2.1.: The test-driven reuse *cycle* (Hummel & Janjic, 2013)

subrank each non-dominated set in a recursive way until all unique priorities and their corresponding subcriteria are applied to the current level of non-dominated sets, eventually resulting in nested subrankings. Note that in each step, when (new)non-dominated sets are determined by non-dominated sorting, the ranks of sets may change according to their partial ordering. In general, partial subranking are deliberately supported since the user is not required to assign specific priorities to all his/her selected criteria. The priorities are defined by simple integer values (highest value represent more priority than a lower value) with a default priority assignment of 1 for each selected criterion. If the user wants to assign a higher or lower priority to a selected criterion, he/she may increase or decrease the priority by 1 or even a larger number. Both unique and equal priority values may be assigned to selected criteria to establish either a partial or strict ordering. For each priority value, all corresponding criteria are selected to subrank each non-dominated set of non-distinguishable candidates. This process starts with the highest priority value and continues in descending order until the smallest priority value is reached (Kessel & Atkinson, 2016). The figure 2.2 is a schematic illustration of how the component ranking works.

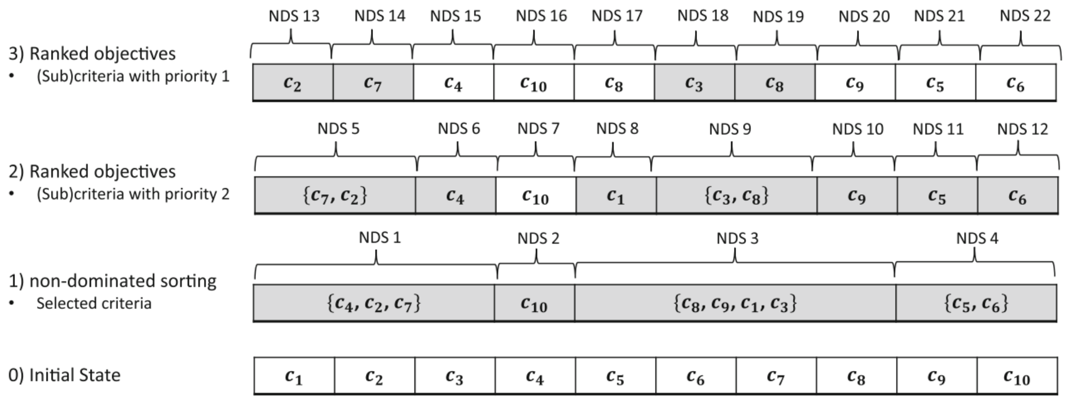


Figure 2.2.: Schematic illustration of SOCORA ranking approach (Kessel & Atkinson, 2016)

2.5.1.1. Merobase integration

The prototype of the ranking component approach is integrated with the component search engine Merobase to retrieve the candidates. The figure 2.3 shows how SOCORA[9] is integrated with Merobase.

In the first step the user's functional requirements are described as a JUnit test specification using a HTML5 web-based GUI. Here the user can specify a set of non-functional *quality* criteria that he/she thinks to be important for his/her needs, also he/she optionally can provide relative priorities to partially order them. In the second step, the interface signatures in the JUnit test are extracted and a text-based search is performed in order to find suitable components. In the step 3, the result set of component from the second step are filtered by applying the JUnit test in turn, and those that do not match the filtering criteria are removed. Also, during the compilation and execution of the test, the metric selected by the user are evaluated. In the step 4, the information from the last step is used by the ranking algorithm to order the remaining components in the result set. In the step 5 the remaining components in the result set are displayed to the user. In the last step, the user can further analyze the component candidates by exploring the effects of different partial ordering (step 6a), explore different criteria (step 6b), and can group them in different way to shrink the result set (step 6c).

[9]SOCORA prototype http://socora.merobase.com (accessed: 13.01.2017)

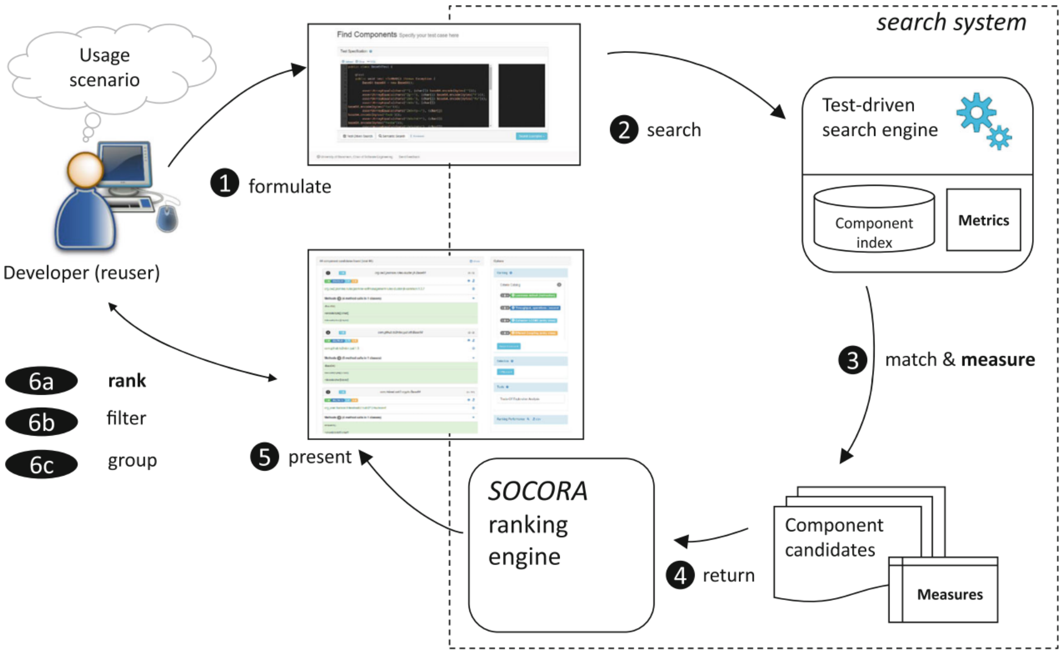


Figure 2.3.: SOCORA and Merobase integration (extracted from (Kessel & Atkinson, 2016))

3. Continuous Integration

In this chapter we will explain what Continuous Integration (CI) means, we will describe the process steps, its benefits and challenges. Finally we will explain how CI can be leveraged in order to improve Software Reuse.

3.1. What is Continuous Integration?

The term can be found in the context of micro-processes development in the work of Booch et al. (2007). Then it was adopted by Kent Beck in his definition of Extreme Programming (Beck, 1999). However, it was Martin Fowler who was credited with establishing the current definitions of the practices. Fowler defines CI as following:

CI is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily, thus leading to multiple integrations per day. Each integration is verified by an automates build (including test) to detect integration errors as quickly as possible. Many teams has found that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapid(Fowler, 2006).

Continuous Integration emerges as a solution for the painful moment of software integration. Although the process of integrating software is not a problem for a one-person project, when it increases in complexity it becomes more problematic. For example, in the old days, a software was divided in modules and each of them were developed independently, once they done, those modules were put together in one step at the end of the project. Doing that led to all sorts of software quality problems, which are costly and often lead to project delays (Duvall et al., 2007).

This step of module integration was a tense moment as errors and failures appeared and they were difficult to find and fix at that stage of the development. In this manner, instead of waiting till the end of modules development to integrate, CI proposes to integrate frequently, usually one person should integrate at least once a day.

Therefore we can break up the CI workflow in the following steps (Fowler, 2006):

CI Workflow:

- Developers check out code into their private workspaces
- When done, commit the changes to the repository
- The CI server monitors the repository and checks out changes when they occur.
- The CI server builds the system and runs unit and integration tests.
- The CI server releases deployable artifacts for testing.
- The CI server assigns a build label to the version of the code it just built.
- The CI server informs the team of the outcome of the build.
- In case the build or test failed, the team fixes the issue at the earliest opportunity.
- Continue to continually integrate and test throughout the project.

This uncomplicated workflow helps the development team to focus on the current code change till it is verified and validated. Moreover, it provides continuous feedback on the quality of the code.

3.1.1. CI Tools

Although the CI is tool agnostic, the selection of them depends on the project, framework in use, skill-set of the stakeholders and other factors. There are however two must-have tools of any CI system: (1) the version control system (VCS) and (2) the CI server.

The most popular VCS are Git, Mercurial and SVN. On top of them, we can find Version Control Platforms (VCP) such as Github, Gitlab or Bitbucket. In terms of CI servers, we can find Jenkins, Hudson, GoCD as open source projects; TravisCI, CircleCI, CodeShip and Team City as commercial tools.

Therefore choosing the appropriate tools is about finding the balance between price, setup, configuration efforts, ease-of-use, integration capabilities between the selected tools, framework suitability and maintainability in respect to current code base.

3.1.1.1. Git

Within the several different VCSs available, Git is one of the most popular[1]. Git emerges in 2005 as an alternative to BitKeeper[2] after the break of the relationship between the commercial company behind BitKeeper and the community that developed the Linux kernel (Chacon, 2009).

The main difference between Git and the other VCSs is the way it thinks about its data. Other VCSs such as Subversion, CVS, Perforce, and so on, think of the information they keep as a set of files and the changes made to each file over time. On the other hand, Git thinks its data like a set of snapshots of a miniature file system. Therefore, every time a user commits, or save the state of the project, Git basically takes a picture of what all the files look like at that moment and store a reference to that snapshot. In addition, Git has three main states where the files can reside in: committed, modified, and staged. Committed means that the data is stored in the local database. Modified means that the file was modified but it has not been committed yet. And staged means that the modified file has been marked to go into the next commit (Chacon, 2009). The figure 3.1 depicts these three states.

The git directory is the core of Git as it is where the metadata and object database for a project is stored. The working directory is a single checkout of one version of the project. Finally, the staging area is a file which is generally stored in the Git directory. This file contains information about what will go into the next commit.

[1]https://rhodecode.com/insights/version-control-systems-2016 (accessed: 14.01.2017)
[2]https://www.bitkeeper.com/ (accessed: 13.01.2017)

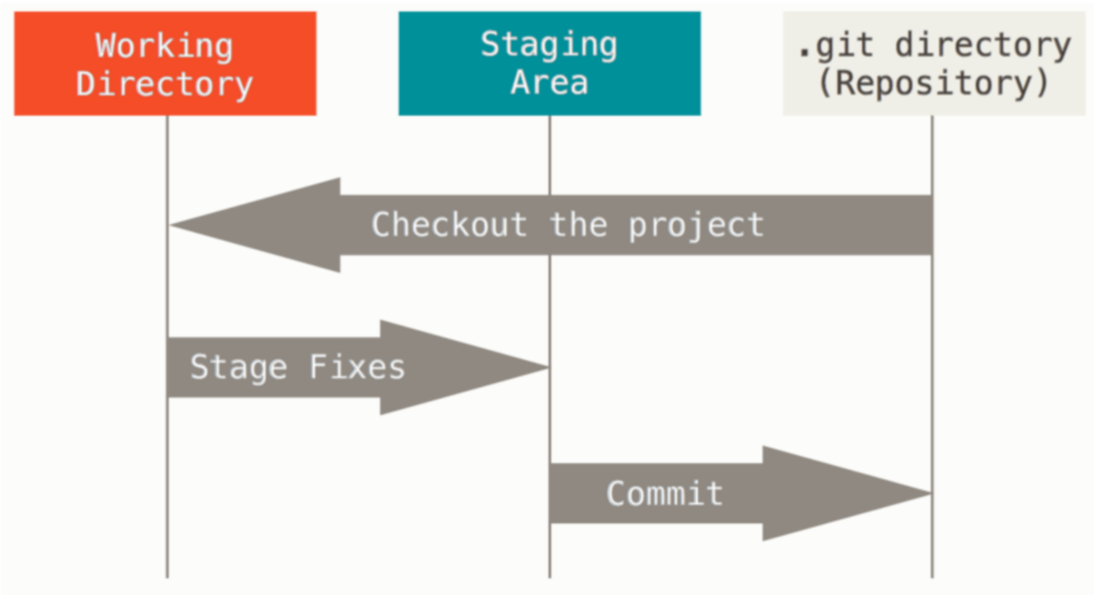


Figure 3.1.: The three main states of Git (Chacon, 2009)

3.1.1.2. Github

Github[3] is a web-based Git repository hosting service, which offers all the functionality of Git and its own features. It provides several collaboration tools such as wikis, bug tracking, feature requests, and task management as well as access control. Its development started in 2007 as a side project of P. J. Hyett and Chris Wanstrath (Weis, 2014) and it was officially lunch on the 10th of April in 2008. Since then, Github has gain popularity through the community reaching more than 50 million projects hosted nowadays.

3.1.1.3. Jenkins

Jenkins is an open source automation server developed in Java. It was originally founded in 2006 as *Hudson*, however a disputed with Oracle the community behind Hudson decided to change the project name to Jenkins[4].

Jenkins enables developers to reliable build, test, and deploy their software. Its extensible and plugin-based architecture has permitted to create a tons of plugins

[3] https://github.com/ (accessed: 13.01.2017)
[4] http://archive.is/fl45 (accessed: 13.01.2017)

to adapt the CI server to a multitude of build, test, and deployment automation workloads. In 2015, Jenkins surpassed 100.000 known installation making it the most widely deployed automation server[5].

3.2. Benefits in adopting CI/CD

Several are the benefits that come with the adoption of Continuous Integration in a software project. In this section we will describe five areas that are improved after CI implementation (Rejström, 2016). It is important to point out that these benefits come along with other practices such as agile transformation (Laanti et al., 2011) and lean software development (Poppendieck & Poppendieck, 2003).

- Shorter time-to-market: Many benefits come with the adoption of frequent releases. With fast and frequent releases is possible to get feedback quickly from customer and market, thus the organization gets better understanding of their needs expectations (Neely & Stolt, 2013). With that is possible to focus on the most relevant features. Another benefit of delivering frequently is waste reduction as features can be deployed as soon as they are done (Leppanen et al., 2015). Furthermore, with frequent releases is possible to experiment with new features easily and with low impact (Neely & Stolt, 2013).
- Rapid feedback: With frequent releases is possible to show progress to customers and by that get feedback quickly. With that the development team can focus on important features instead of waste time in features that are not relevant for the customers or the market.
- Improved software quality: Researches have reported a decreased in the number of open bugs and production incidents after adopting CI practices (Mäntylä et al., 2015) and the link between software quality improvement and the heavy reliance on automated test combined with smaller more manageable releases (Leppanen et al., 2015).
- Improved release reliability: In the work of Neely & Stolt (2013) was proved that a working deployment pipeline along with intensive automated testing and fast rollback mechanism possitively affects release reliability and

[5]https://jenkins.io/press/ (accessed: 13.01.2017)

quality. With small and frequent releases fewer things can go wrong in a release according to Fowler (2013). In fact, reduction in the stress of developers and other stakeholder have been found by adopting CI (Neely & Stolt, 2013) (Chen, 2015).

- Improved developer productivity: By automating deployment process, environment configuration and other non-value adding tasks, significant time saving for developers have been observed (Rodríguez et al., 2016). Also, in the work of Humble & Farley (2010) was observed that although the setup cost of a deployment pipeline can be high, after it is setup developers can focus on value adding software development works.

3.3. Challenges in adopting CI/CD

Adopting CI can be very beneficial for a software project as described above, however we can face some challenges that can jeopardize the successful implementation of it. According to the literature, there are seven common challenges in the implementation of a CI:

- Change resistance: Transforming the development of a project towards continuous integration practices requires investment and involvement from the entire organization (Rodríguez et al., 2016). Therefore any transformation in how an organization works, will receive a resistance on both a personal level and decision level within the organization.
- External constraint: As a software project is part of a context, external constraint may appear. Normally customer preferences and domain imposed restrictions are sources of constraints. For example in highly restricted domains, legal regulations may require extensive testing before new version can be allowed to enter production (Rodríguez et al., 2016).
- QA effort: The automated tests suit needs to be exhaustive enough in order to ensure the quality of what is being built. Thus it can lead to increase QA efforts due to difficulties in managing the test automation infrastructure (Rodríguez et al., 2016).
- Legacy code: Normally legacy software has not been design for being automatically tested and may cause integration failures which may inhibit the

continuous deployment process. The ownership of legacy code might belong to another company or team which shift the testing responsibility and this might delay the deployment process.

- Complex software: If the complex if the software project is high, then setting up the CI workflow is more challenging (Leppanen et al., 2015).
- Environment management: Keeping all the environments used in development, testing and production sync and similar can be challenging. This is due to the fact that differences in the environment configuration can lead to undetected issues appear in production. Therefore is essential to have a good configuration management that provisions environments automatically (Leppanen et al., 2015).
- Manual testing: Although automated tests are very beneficial, some aspects of software need to be manually tested such as security issues, performance and UX/UI. Therefore heavy manual testing can impact the overall speed and smoothness of the process (Leppanen et al., 2015).

3.4. Successful CI practices

In order to reduce the risks in adopting CI practices, eight principles, based on several literature reviews, were defined in (Rejström, 2016) which should guide the CI implementation within organizations. These principles are generic solutions which can be adapted in most cases.

- Automate the build: The task that triggers the whole pipeline is the commit. After each commit, the next step should be build the binaries. The executable that result of the build, should go through the pipeline until it gets validated, verified and deployment ready.
- Make your build self-testing:
- Every commit should build on an integration machine
- Keep the build fast
- Keep the build green
- Test in a clone of the production environment

- Everyone can see what is happening
- Automate deployment

These 8 principles should guide the CI/CD implementation within organizations. The principles have been stablished as best practices through the validation of these concepts through several literature reviews [Rodríguez et al. (2016); Mäntylä et al. (2015); Ståhl & Bosch (2014)]. They can be viewed almost as a design pattern for adopting CI/CD, offering a generic solution adaptable to most cases.

4. Simile

In this work we present Simile, which is a tool that leverages the Continuous Integration process in order to recommend similar components that are being developed in a software project. A common use case would look like the following:

First of all we will assume that the development team is using Git as VCS, Jenkins as CI server and Java as main language. Once a member of the team makes a commit and pushes to the remote Git server, simile will be trigger by Jenkins. This tool will receives the repository url where the project is. Then it will clone the repository locally and then it will go through the source code. Simile will extract the different class names and methods of the source code and it will make requests to SOCORA. SOCORA will response with all the components that are similar to the components extracted from the project. Simile will get this response and it will generate a report listing all the similar components to the components that are being developed by the team. This report will be sent to the email of all the members of team so they will know if there is already a component that can be reuse instead of developing it again from scratch.

The main objective of Simile is to help the team to find similar components so they would be able to reuse them. Thus they would reduce time development and costs through reuse components that are already tested and ready to work.

4.1. CI integration

Our prototype leverages the Continuous Integration process to improve the chances of component reuse. Whenever

4.2. SOCORA Integration

5. Ein Kapitel

Hier kommt eine kurze Zusammenfassung des Kapitels hin.

5.1. Bilder

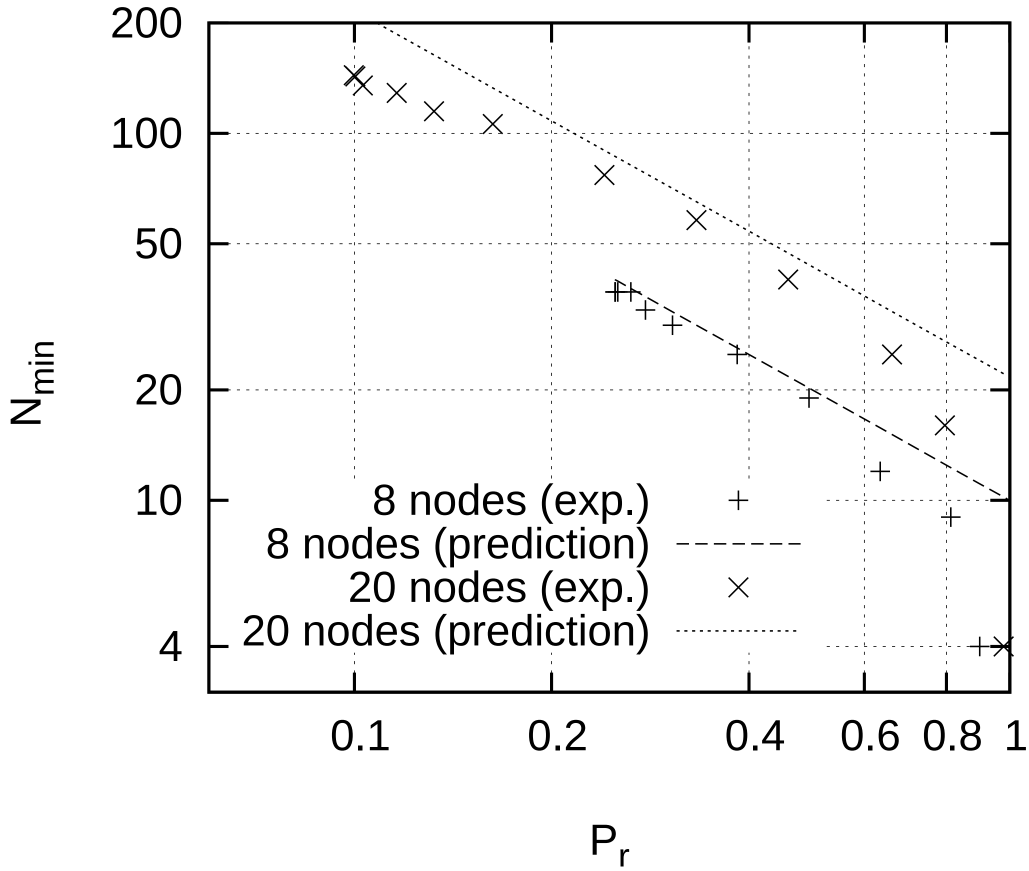
In Abbildung 5.1 findet man ein schönes Beispiel für das Paket subfigure. Subfigure ist sehr praktisch beim Setzen von mehreren Bildern in ein Bild. Man beachte, dass man mit dem Befehl ref auf Bilder verweisen kann, wenn diesen mit dem Befehl label eine Marke zugewiesen wurde. Nicht alle Dateiformate können so problemlos wie ".eps" mit Latex direkt eingebunden werden. Benötigt man zum Beispiel eine pdf Grafik, so kann man das Dokument mit pdflatex compilieren. Hat man alle Bilder sowohl als pdf als auch als eps, kann man wahlweise latex oder pdflatex nutzen. Ansonsten hilft auch zeitweiliges auskommentieren der Bilder.

Zum Zeichnen von plots ist gnuplot sehr zu empfehlen (vgl. Abbildung 5.1(a)). Ist perfekt für zwei oder dreidimensionale Plots und schaut um Welten besser aus als Excel.

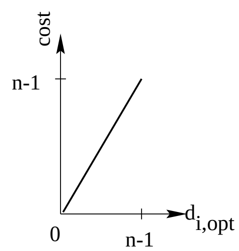
Zum Malen von Bildern ist das Programm xfig oder jfig sehr gut geeignet (vgl. Abbildung 5.1(b)). Schaut zwar am Anfang ein bißchen seltsam aus, ist aber sehr mächtig.

5.2. Editor

Als Editor für Fortgeschrittene ist xemacs gut geeignet. Bei Verwendung von xemacs ist der Gebrauch der reftex und auctex-pakete zu empfehlen. Als Rechtschreibprogramm sollte aspell verwendet werden und folgender Code in init.el eingefügt werden.



(a) Ein mit gnuplot erzeugtes Bild



(b) Ein mit xfig erzeugtes Bild

Figure 5.1.: Beispiele für schöne Bilder.

```
(require 'iso-cvt)
(add-hook 'LaTeX-mode-hook
(function (lambda ()
;; Setze Anfuehrungszeichen etc. fuer Style german
(TeX-run-style-hooks "german")
;;
;; Lade Buffer und wandle nach ISO Latin-1:
(format-encode-buffer 'plain)
)))

(setq ispell-silently-savep t) ;save new words in pdict without questioning
(setq ispell-help-in-bufferp 'electric) ;get a better help buffer
(setq ispell-program-name "aspell")
(setq ispell-extra-args '("-W" "2"))

(column-number-mode t)

(custom-set-variables
'(paren-mode 'sexp nil (paren)))

(setq reftex-plug-into-AUCTeX t)
(require 'reftex "reftex" t)
(turn-on-reftex) ; use reftex
```

Neulinge können auch einen sonstigen Editor oder Sachen wie Texnic Center, etc verwenden.

5.3. Mathematik

In latex kann man Formeln wunderschön setzen.

As the following stuff is in English, we must change the hyphenation style.

The OCST problem is defined as follows. Let $G = (V, E)$ be a connected, undirected graph with $n = |V|$ nodes and $m = |E|$ edges. There are communication, or transportation demands, between every pair of nodes. An $n \times n$ *demand matrix* $R = (r_{ij})$ specifies the demands, where r_{ij} is the amount of traffic required between location v_i and v_j. Similarly, an $n \times n$ *distance matrix* $W = w_{ij}$ determines the distance weights between each pair of sites. A tree $T = (V, F)$ where $F \subseteq E$ and $|F| = |V| - 1$ is called a *spanning tree* of G if it connects all the nodes. The weight $w(T)$ of the spanning tree is the weighted sum over all pairs of vertices of the cost of the path between the pair in T. The communication cost over the tree T is defined as

$$w(T) = \sum_{i,j \in V} w_{ij} b_{ij}, \tag{5.1}$$

where $B = b_{ij}$ denotes the traffic flowing directly and indirectly across the edge connecting nodes i and j. It is calculated according to the structure of T. T is the optimal communication spanning tree if $w(T) \leq w(T')$ for all other spanning trees T'. The OCST problem becomes the minimum spanning tree (MST) problem if $w(T) = \sum_{i,j \in V} w_{ij}$. Then, T is the simple minimum spanning tree if $w(T) \leq w(T')$ for all other spanning trees T'.

Cayley's formula identifies the number of spanning trees on n nodes as n^{n-2} . Furthermore, there are n different stars on a graph of n nodes. The similarity between two spanning trees T_i and T_j can be measured using the distance $d_{ij} \in \{0, 1, \ldots, n-1\}$ as

$$d_{ij} = \frac{1}{2} \sum_{u,v \in V, u<v} |l^i_{uv} - l^j_{uv}|,$$

where l^i_{uv} is 1 if an edge from u to v exists in T_i and 0 if it does not exist in T_i. The number of edges that two trees T_i and T_j have in common is $n - 1 - d_{ij}$.

Ab jetzt wieder deutsche Trennung.

In Abbildung 5.2 gibt es noch ein Beispiel für ein Bild und in Tabelle 5.1 gibt es ein einfaches Bild. Tabelle 5.2 stellt eine etwas kompliziertere Tabelle dar.

Beim Aufbau von Kommunikationsnetzen stehen Unternehmen vor dem Problem, eine Menge von unterschiedlichen Standorten so durch Leitungen zu verbinden, dass bestimmte Qualitätsanforderungen (z.B. Sicherheit, Bandbreite, Zuverlässigkeit, Skalierbarkeit, etc.) eingehalten werden und gleichzeitig die Netzstruktur so gewählt wird, dass die Kosten des Netzaufbaus minimiert werden. Unternehmen, für welche der Aufbau von Kommunikationsnetzwerken relevant ist, lassen sich

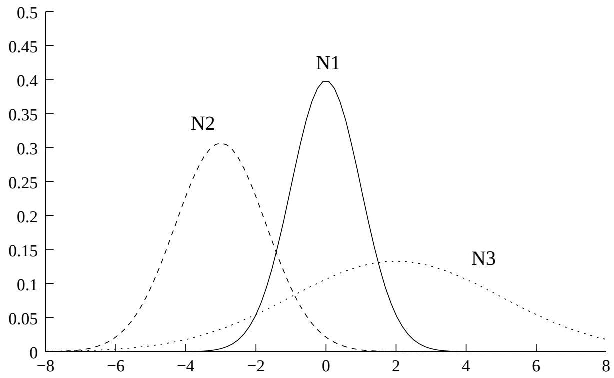


Abbildung 5.2.: Dichten univariater Normalverteilungen

Tabelle 5.1.: Eine sehr einfache Tabelle

linksbündig	zentriert	rechtsbündig
1	2	3,141

in zwei Gruppen unterteilen:

Auf der einen Seite sind Unternehmen zu finden, für die Kommunikationsnetzwerke ein wichtiger Teil ihrer Unternehmensinfrastruktur sind (z.B. Banken, Verlagshäuser, Bahn, Bundeswehr, Unternehmen mit mehreren Niederlassungen, etc.) und welche daher die Kommunikationsnetzwerke oft selbst betreiben. Ein Beispiel hierfür ist die DATEV eG, welche über ein eigenes „Genossenschaftsnetz" mehrere Dutzend Niederlassungen an die Zentrale in Nürnberg anbindet. Über dieses Kommunikationsnetz wird sowohl der gesamte Daten- als auch Telefonverkehr innerhalb des Unternehmens abgewickelt.

Auf der anderen Seite stehen Unternehmen, für die der Aufbau und der Betrieb von Kommunikationsnetzwerken Kernaufgaben darstellen (z.B. Telekommunikationsunternehmen (Deutsche Telekom oder Deutsches Forschungsnetz (DFN)), Mobilfunkbetreiber (vodafon, T-mobil) oder lokale Provider (NetCologne, Stadtwerke). Für derartige Unternehmen gehört der Aufbau und Betrieb von Kommunikationsnetzwerken zum Kerngeschäft. Netzwerke sind so aufzubauen und weiterzuentwickeln, dass Qualitätskriterien eingehalten werden und gleichzeitig die Betriebskosten minimiert werden.

Tabelle 5.2.: Performance of GAs using different types of representations for deceptive tree problems of different sizes and with different T_{opt} (arbitrary tree, MST, and star)

T_{opt}		oder 3				
		P_{succ}	fitness		t_{conv}	
			μ	σ	μ	σ
arbitrary tree	Prüfer number	0.54	0.49	(0.6)	26.0	(8.6
	NetKey	0.78	0.23	(0.4)	23.0	(6.0)
	LB (P_1=1)	0.09	1.69	(0.8)	19.5	(7.6)
	LB (P_1=20)	0.82	0.18	(0.4)	23.3	(6.1)
	P_1=P_2=1	0.12	1.24	(0.6)	27.4	(8.0)
	heur. xover	0	2.63	(0.5)	8.7	(2.4)
	h. ini & xover	0	3.88	(0.1)	0.4	(0.4)

Beim Aufbau von Kommunikationsnetzwerken kann unterschieden werden zwischen vermaschten und baumförmigen Netzwerken (vergleiche Abbildung 5.2). In vermaschten Netzen existieren mehrere unterschiedliche Wege für den Transport von Daten zwischen zwei Standorten. Daher muss bei vermaschten Netzwerken das Routing der Daten durch das Netzwerk gesteuert werden (wie z.B. im Internet). In der Regel existieren mehrere Wege zwischen zwei Standorten und es muss mit Hilfe von Routingmechanismen festgelegt werden, welchen Weg die Daten durch das Netz nehmen sollen (wie z.B. durch Routingtabellen für den IP-Verkehr im Internet). Diese Entscheidung wird unter anderem durch die Entfernung von Standorten als auch durch die Auslastung der unterschiedlichen Leitungen beeinflusst. Die Ermittlung von Routingtabellen ist in der Praxis recht aufwendig und es besteht die Gefahr, dass Nachrichten Umwege durch das Netz nehmen oder sogar im Kreis laufen. Vermaschte Netzwerke bieten allerdings eine erhöhte Ausfallsicherheit, da beim Vorhandensein von mehreren Wegen zwischen zwei Knoten Daten auch dann noch ausgetauscht werden können, wenn ein Teil des Netzwerkes ausfällt.

Baumförmige Netze stellen einen Sonderfall von vermaschten Kommunikationsnetzwerken dar. Hierbei existiert genau ein möglicher Weg zwischen den Knoten im Netz. Daher muss keine Entscheidung über das Routing getroffen werden und Routingmechanismen zur Steuerung des Datenflusses entfallen. Allerdings haben baumförmige Netzwerke das Problem, dass durch den Ausfall einer Leitung ein Teil der Knoten nicht mehr erreichbar sind. Baumförmige Kommunikationsnetzwerke werden oft bei einer überschaubaren Anzahl von Standorten wie z.B. in Local Area Networks (LAN) oder Unternehmensnetzwerken („Corporate Networks”) eingesetzt. Insbesondere Unternehmen, welche eigene Netzwerke betreiben (z.B. die DATEV eG), verwenden oft baumförmige Kommunikationsnetzwerkstrukturen wegen ihrer geringereren Komplexität.

Das Problem des kostenminimalen Aufbaus von vermaschten bzw. baumförmigen Kommunikationsnetzen wurde schon sehr frühzeitig in der Literatur formalisiert und beschrieben. Beim „network design problem” soll diejenige Netzwerkstruktur bestimmt werden, welche alle Standorte miteinander verbindet, alle Kommunikationsanforderungen zwischen den einzelnen Standorten erfüllt und gleichzeitig minimale Kosten besitzt. Aus dem allgemeinen „network design problem” lässt sich das „optimal communication spanning tree” (OCST)-Problem ableiten, bei welchem die Struktur des gewünschten Netzwerkes baumförmig ist. Im folgenden Abschnitt soll näher auf das OCST Problem eingegangen werden.

6. Summary, Conclusions, and Further Work

The purpose of this book is to understand the influence of representations on the performance of genetic and evolutionary algorithms. This chapter summarizes the work contained in this study and lists its major contributions.

6.1. Summary

This is the final section 6.1. We started in Chap. 1 by providing the necessary background for examining representations for GEAs. Researchers recognized early that representations have a large influence on the performance of GEAs. Consequently, after a brief introduction into representations and GEAs, we discussed how the influence of representations on problem difficulty can be measured. The chapter ended with prior guidelines for choosing high-quality representations. Most of them are mainly based on empirical observations and intuition and not on theoretical analysis.

Therefore, we presented in Chap. 5 three aspects of a theory of representations for GEAs. We investigated how the locality, scaling, and locality of an encoding influences GEA performance. The performance of GEAs is determined by the solution quality at the end of a run and the number of generations until the population is converged. Consequently, for redundant and exponentially scaled encodings, we presented population sizing models and described how the time to convergence is changed. Furthermore, we were able to demonstrate that high-locality encodings do not change the difficulty of a problem; in contrast, when using low-locality encodings, on average, the difficulty of problems changes. Therefore, easy problems become more difficult and difficult problems become easier by the use of low-locality encodings. For all three properties of encodings, the theoretical models were verified with empirical results.

6.2. Conclusions

We summarize the most important contributions of this work.

Framework for design and analysis of representations (and operators) for GEAs. The main purpose of this study was to present a framework which describes how genetic representations influence the performance of GEAs. The performance of GEAs is measured by the solution quality at the end of the run and the number of generations until the population is converged. The proposed framework allows us to analyze the influence of existing representations on GEA performance and to develop efficient new representations in a theory-guided way. Furthermore, we illustrated that the framework can also be used for the design and analysis of search operators, which are relevant for direct encodings. Based on the framework, the development of high-quality representations remains not only a matter of intuition and random search but becomes an engineering design task. Even though more work is needed, we believe that the results presented are sufficiently compelling to recommend increased use of the framework.

Redundancy, Scaling, and Locality. These are the three elements of the proposed framework of representations. We demonstrated that these three properties of representations influence GEA performance and presented theoretical models to predict how solution quality and time to convergence changes. By examining the redundancy, scaling, and locality of an encoding, we are able to predict the influence of representations on GEA performance.

The theoretical analysis shows that the redundancy of an encoding influences the supply of building blocks (BB) in the initial population. r denotes the number of genotypic BBs that represent the best phenotypic BB, and k_r denotes the order of redundancy. For synonymously redundant encodings, where all genotypes that represent the same phenotype are similar to each other, the probability of GEA failure goes either with $O(\exp(-r/2^{k_r}))$ (uniformly scaled representations) or with $O(\exp(-\sqrt{r/2^{k_r}}))$ (exponentially scaled representations). Therefore, GEA performance increases if the representation overrepresents high-quality BBs. If a representation is uniformly redundant, that means each phenotype is represented by the same number of genotypes, GEA performance remains unchanged in comparison to non-redundant encodings.

The analysis of the scaling of an encoding reveals that non-uniformly scaled representations modify the dynamics of genetic search. If exponentially scaled representations are used, the alleles are solved serially which increases the overall time until convergence and results in problems with genetic drift but allows rough approximations of the expected optimal solution after a few generations.

We know from previous work that the high locality of an encoding is a necessary condition for efficient mutation-based search. An encoding has high locality if neighboring phenotypes correspond to neighboring genotypes. Investigating the influence of locality shows that high-locality encodings do not change the difficulty of a problem. In contrast, low-locality encodings, where phenotypic neighbors do not correspond to genotypic neighbors, change problem difficulty and make, on average, easy problems more difficult and deceptive problems easier. Therefore, to assure that an easy problem remains easy, high-locality representations are necessary.

6.3. Further Work

What are the open questions? What should be done next?

Literaturverzeichnis

Bauer, V. & Vetro', A. (2016). Comparing reuse practices in two large software-producing companies. *Journal of Systems and Software*, 117, 545–582.

Beck, K. (1999). *Extreme Programming Explained: Embrace Change*. Number c.

Booch, G., Maksimchuk, R. a., Engle, M. W., Young, B. J., Conallen, J., & Houston, K. a. (2007). *Object-Oriented Analysis and Design with Applications*, volume 1.

Chacon, S. (2009). Pro Git. *Control*, (pp. 1–210).

Chen, L. (2015). Towards Architecting for Continuous Delivery. In *2015 12th Working IEEE/IFIP Conference on Software Architecture* (pp. 131–134).: IEEE.

Duvall, P., Matyas, S., & Glover, A. (2007). *Continuous Integration: Improving Software Quality and Reducing Risk*.

Foundation, T. A. S. (2017). Apache Lucene - Apache Lucene Core.

Fowler, M. (2006). Continuous Integration.

Fowler, M. (2013). Continuous Delivery.

Frakes, W. & Kyo Kang, K. (2005). Software reuse research: status and future. *IEEE Transactions on Software Engineering*, 31(7), 529–536.

Frakes, W. & Terry, C. (1996). Software reuse: metrics and models. *ACM Computing Surveys*, 28(2), 415–435.

Garcia, V. C., de Almeida, E. S., Lisboa, L. B., Martins, A. C., Meira, S. R. L., Lucredio, D., & Fortes, R. P. d. M. (2006). Toward a Code Search Engine Based on the State-of-Art and Practice. In *2006 13th Asia Pacific Software Engineering Conference (APSEC'06)* (pp. 61–70).: IEEE.

Horowitz, B. (2011). Official Google Blog: A fall sweep.

Humble, J. & Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.*

Hummel, O. & Atkinson, C. (2004). Extreme harvesting: Test driven discovery and reuse of software components. In *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, IRI-2004* (pp. 66–72).

Hummel, O. & Janjic, W. (2013). Test-Driven Reuse: Key to Improving Precision of Search Engines for Software Reuse. In *Finding Source Code on the Web for Remix and Reuse* (pp. 227–250). New York, NY: Springer New York.

Hummel, O., Janjic, W., & Atkinson, C. (2007). Evaluating the efficiency of retrieval methods for component repositories. In *19th International Conference on Software Engineering and Knowledge Engineering, SEKE 2007* (pp. 404–409).

Hummel, O., Janjic, W., & Atkinson, C. (2008). Code Conjurer: Pulling Reusable Software out of Thin Air. *IEEE Software*, 25(5), 45–52.

Inoue, K., Yokomori, R., Yamamoto, T., Matsushita, M., & Kusumoto, S. (2005). Ranking significance of software components based on use relations. *IEEE Transactions on Software Engineering*, 31(3), 213–225.

Kessel, M. & Atkinson, C. (2016). Ranking software components for reuse based on non-functional properties. *Information Systems Frontiers*, (pp. 1–29).

Kim, Y. & Stohr, E. (1992). Software reuse: issues and research directions. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences* (pp. 612–623 vol.4).: IEEE.

Krueger, C. W. & W., C. (1992). Software reuse. *ACM Computing Surveys*, 24(2), 131–183.

Laanti, M., Salo, O., & Abrahamsson, P. (2011). Agile methods rapidly replacing traditional methods at Nokia: A survey of opinions on agile transformation. *Information and Software Technology*, 53(3), 276–290.

Lemos, O. A. L., Bajracharya, S. K., & Ossher, J. (2007). CodeGenie: a Tool for Test-Driven Source Code Search. *OOPSLA 2007*, 07pp, 525–526.

Leppanen, M., Makinen, S., Pagels, M., Eloranta, V.-P., Itkonen, J., Mantyla, M. V., & Mannisto, T. (2015). The highways and country roads to continuous deployment. *IEEE Software*, 32(2), 64–72.

Mäntylä, M. V., Adams, B., Khomh, F., Engström, E., & Petersen, K. (2015). On rapid releases and software testing: a case study and a semi-systematic literature review. *Empirical Software Engineering*, 20(5), 1384–1425.

McIlroy, M. (1968). Mass Produced Software Components. *Software Engineering: Report on Conference Sponsored by NATO*, (pp. 138–155).

Mili, a., Mili, R., & Mittermeir, R. (1998). A survey of software reuse libraries. *Annals of Software Engineering*, 5(1), 349–414–414.

Mili, H. (2002). *Reuse based software engineering : techniques, organization and measurement*. Wiley.

Morisio, M., Ezran, M., & Tully, C. (2002). Success and failure factors in software reuse. *IEEE Transactions on Software Engineering*, 28(4), 340–357.

Neely, S. & Stolt, S. (2013). Continuous Delivery? Easy! Just Change Everything (Well, Maybe It Is Not That Easy). In *2013 Agile Conference* (pp. 121–128).: IEEE.

Pole, T. P. & Frakes, W. (1994). An Empirical Study of Representation Methods for Reusable Software Components. *IEEE Transactions on Software Engineering*, 20(8), 617–630.

Poppendieck, M. M. B. & Poppendieck, T. D. (2003). *Lean software development : an agile toolkit*. Addison-Wesley.

Prieto-Díaz, R. & Freeman, P. (1987). Classifying Software for Reusability. *IEEE Software*, 4(1), 6–16.

Rejström, K. (2016). Implementing Continuous Integration in a Small Company: A Case Study.

Rodríguez, P., Haghighatkhah, A., Lwakatare, L. E., Teppola, S., Suomalainen, T., Eskeli, J., Karvonen, T., Kuvaja, P., Verner, J. M., & Oivo, M. (2016). Continuous deployment of software intensive products and services: A systematic mapping study. *Journal of Systems and Software*, 123, 263–291.

Seacord, R. (1999). Software engineering component repositories. *Proceedings of the International Workshop on*.

Ståhl, D. & Bosch, J. (2014). Automated software integration flows in industry: a multiple-case study. In *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014* (pp. 54–63). New York, New York, USA: ACM Press.

Standish, T. A. (1984). An Essay on Software Reuse. *IEEE Transactions on Software Engineering*, SE-10(5), 494–497.

Thummalapenta, S. & Xie, T. (2007). Parseweb: a programmer assistant for reusing open source code on the web. *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, (pp. 204–213).

Weis, K. (2014). GitHub CEO and Co-Founder Chris Wanstrath Keynoting Esri's DevSummit! — ArcGIS Blog.

Anhang

A. First class of appendices

A.1. Some appendix

This is a sample appendix entry.

Eidesstattliche Erklärung

Hiermit versichere ich, dass diese Abschlussarbeit von mir persönlich verfasst ist und dass ich keinerlei fremde Hilfe in Anspruch genommen habe. Ebenso versichere ich, dass diese Arbeit oder Teile daraus weder von mir selbst noch von anderen als Leistungsnachweise andernorts eingereicht wurden. Wörtliche oder sinngemäße Übernahmen aus anderen Schriften und Veröffentlichungen in gedruckter oder elektronischer Form sind gekennzeichnet. Sämtliche Sekundärliteratur und sonstige Quellen sind nachgewiesen und in der Bibliographie aufgeführt. Das Gleiche gilt für graphische Darstellungen und Bilder sowie für alle Internet-Quellen.

Ich bin ferner damit einverstanden, dass meine Arbeit zum Zwecke eines Plagiatsabgleichs in elektronischer Form anonymisiert versendet und gespeichert werden kann. Mir ist bekannt, dass von der Korrektur der Arbeit abgesehen werden kann, wenn die Erklärung nicht erteilt wird.

Ort, den Datum

Martin Mustermann