UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Build management for a software chrestomathy

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science
im Studiengang Informatik

vorgelegt von
Thomas Bernau

Erstgutachter: Ralf Lämmel
Institut für Informatik

Zweitgutachter: Andrei Varanovich
Institut für Informatik

Koblenz, im Juli 2013

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

....

(Ort, Datum) (Unterschrift)

Abstract

In the context of this thesis a build management effort for the 101companies chrestomathy is bootstrapped. An analysis of the initial situation, its valuation and as a result the definition of requirements for build management from a requirements engineering perspective lay the foundation for a study of related work. A systematic discussion of improvements to build related qualities of chrestomathies and build management tools in the context of build systems and continuous integration enables the choice of tools and the design of a solution. The solution is then implemented and briefly discussed.

Zusammenfassung

Im Rahmen dieser Arbeit wird ein Ansatz für Build Management für die Chrestomathie 101companies erarbeitet. Eine Analyse der ausgehenden Situation, ihre Bewertung und als Resultat die Definition von Anforderungen zum Build Management aus der Sicht des Anforderungsmanagement schaffen die Grundlage für eine Untersuchung themenbezogener Arbeiten. Eine systematische Diskussion zur Verbesserung von Build-bezogenen Eigenschaften von Chrestomathien und Build Management Tools im Zusammenhang mit Build Systemen und kontinuierlicher Integration ermöglicht die Wahl von Werkzeugen und den Entwurf einer Lösung. Diese Lösung wird anschließend umgesetzt und kurz diskutiert.

Contents

1 Introduction **8**
1.1 Motivation . . . 8
1.2 Build Management . . . 8
1.2.1 Build Process . . . 8
1.2.2 Managing Builds . . . 8
1.2.3 Revision Control . . . 9
1.2.4 Build Systems . . . 9
1.2.5 Continuous Integration . . . 9
1.3 Chrestomathies . . . 10
1.3.1 Definition . . . 10
1.3.2 Properties . . . 10
1.3.3 101companies . . . 10
1.4 Goal . . . 10

2 Requirement Engineering **11**
2.1 Scope . . . 11
2.2 Stakeholders . . . 12
2.2.1 Users . . . 12
Definition . . . 12
Concern . . . 12
Relevance . . . 12
2.2.2 Contributors . . . 13
Definition . . . 13
Concern . . . 13
Relevance . . . 13
2.2.3 Researchers . . . 13
Definition . . . 13
Concern . . . 13
Relevance . . . 13
2.2.4 Technologists . . . 14
Definition . . . 14
Concern . . . 14
Relevance . . . 14
2.3 Requirements . . . 14
2.3.1 Build Automation . . . 14
Definition . . . 14
Explanation . . . 14
Discussion . . . 14

2.3.2 Test Automation . . . 15
Definition . . . 15
Explanation . . . 15
Discussion . . . 15
2.3.3 Neutral Build . . . 15
Definition . . . 15
Explanation . . . 15
Discussion . . . 16
2.3.4 Test Coverage . . . 16
Definition . . . 16
Explanation . . . 16
Discussion . . . 16
2.3.5 Notification Automation . . . 16
Definition . . . 16
Explanation . . . 16
Discussion . . . 16
2.3.6 Dependency Management . . . 16
Definition . . . 16
Explanation . . . 17
Discussion . . . 17
2.3.7 Limited Heterogeneity . . . 17
Definition . . . 17
Explanation . . . 17
Discussion . . . 17
2.3.8 Comprehensibility . . . 17
Definition . . . 17
Explanation . . . 17
Discussion . . . 18
2.3.9 Comparability . . . 18
Definition . . . 18
Explanation . . . 18
Discussion . . . 18
2.3.10 Discoverability . . . 18
Definition . . . 18
Explanation . . . 18
Discussion . . . 19
2.3.11 Usability . . . 19
Definition . . . 19
Explanation . . . 19
Discussion . . . 19
2.3.12 Maintainability . . . 19
Definition . . . 19
Explanation . . . 19
Discussion . . . 20
2.3.13 System Independence . . . 20
Definition . . . 20
Explanation . . . 20
Discussion . . . 20
2.3.14 Portability . . . 20

Definition . . . 20
Explanation . . . 20
Discussion . . . 20
2.3.15 Scalability . . . 21
Definition . . . 21
Explanation . . . 21
Discussion . . . 21
2.3.16 Extensibility . . . 21
Definition . . . 21
Explanation . . . 21
Discussion . . . 21
2.4 Synopsis . . . 22

3 Related Work **23**
3.1 101companies . . . 23
3.2 Homogenization . . . 23
3.2.1 General . . . 23
3.2.2 Java . . . 24
3.3 Build systems . . . 24
3.3.1 Scope and Relevance . . . 24
3.3.2 General . . . 24
Build Maintenance Effort . . . 24
Build Code Analysis . . . 25
Scalability . . . 25
3.3.3 Make . . . 25
3.3.4 Ant . . . 25
3.3.5 Maven . . . 26
3.3.6 Gradle . . . 26
3.3.7 Synopsis . . . 26
3.4 Continuous integration . . . 26
3.4.1 Scope and Relevance . . . 26
3.4.2 General . . . 27
Integration . . . 27
Scalability . . . 27
3.4.3 CruiseControl . . . 27
3.4.4 Hudson . . . 27
3.4.5 Jenkins . . . 27
3.4.6 Travis . . . 28
3.4.7 Synopsis . . . 28

4 Design **29**
4.1 Scope . . . 29
4.2 Homogenization . . . 29
4.2.1 Structural Conventions . . . 29
4.2.2 Naming Conventions . . . 29
4.2.3 Result . . . 30
4.3 Build systems . . . 31
4.3.1 Structural Conventions . . . 31
4.3.2 Architecture . . . 31
4.3.3 Result . . . 31

4.4 Continuous integration 32
4.4.1 Properties 32
4.4.2 Architecture 32

5 Implementation 33
5.1 Homogenization 33
5.1.1 antlrObjects 33
5.1.2 Documentation 36
5.1.3 Problems 36
5.2 Build systems 36
5.2.1 Setup 36
5.2.2 Usage 37
5.3 Continuous integration 38
5.3.1 Setup 38
5.3.2 Usage 38

6 Discussion 39
6.1 Homogenization 39
6.1.1 Beyond Simple Java 39
6.1.2 Consequences 39
6.2 Beyond Gradle 39
6.3 Beyond Jenkins 40

7 Summary 41

8 Future Work 42

Chapter 1

Introduction

1.1 Motivation

The 101companies chrestomathy [1] currently covers a broad variety of software languages - from description languages such as XML or HTML over scripts (SQL) all the way to programming languages like Java, C++, Haskell and many [2] more - and their technologies. With the increasing complexity of the project the need for standardization and build related improvements arises. Requirements need to be identified and applied to search for relevant related work and to bootstrap an appropriate solution. The challenge of this thesis lies in the importance of the term chrestomathy as it has significant differences in requirements to usual build efforts in software projects. At first the terminology needs to be determined.

1.2 Build Management

1.2.1 Build Process

The build process consists of several simple steps such as compilation of code that together transform a set of given source files into an executable program [4] and possibly execute tests. The specific build process of a project depends on its purpose and use cases. A build process for a trivial Java project would only consist of the command line operation *javac Class.java* to derive the executable Class.class file. The more complex such projects become a corresponding build process could consist of arbitrarily many sequential operations involving all kinds of tools.

1.2.2 Managing Builds

Build management is the activity of organizing and implementing build processes. Its purpose is usually to provide a means of automation for build related issues and therefore to reduce build efforts during development and to ensure software quality by enforcing conventions and regular test execution. Different layers can be identified: management of source code (revision control), automation of the build process (build systems) and management of build automation as to when and what is executed (continuous integration). Each layer offers different tools with different capabilities.

1.2.3 Revision Control

Revision control [5] manages changes to source code (or in general files) of repositories and is an essential process in development with more than one developer. With features like change revocation it also offers a high value for developers working by themselves and is furthermore useful for central storage for cross-device usage. Most tools work similar and mainly differ in the approach of how to treat simultaneous changes (e.g. don't care or lock file).

1.2.4 Build Systems

Build systems [6] are tools for build automation. They are used to provide a means for representation of build processes (in the form of build scripts) and their execution. A build process in its representation is expressed by tasks and dependencies between tasks. Tasks are atomic units of execution such as a simple compile command or an invocation of another tool with parameters. Dependencies between tasks express the ordering in which the individual steps of a build process have to take place.

Modern build systems include by default a life cycle. This defines a standard build process and is mostly connected to the language the build tool was primarily intended for. It is merely a dependency chain that invokes pre-defined commonly used tasks and therefore frees the build script of unimportant or trivial tasks (see for example the Maven Build Lifecycle [7]).

An important difference between build systems is the language used in build scripts. For example Makefiles use their own language which is reminiscent of Unix commands, Ant and Maven use XML and Gradle uses a mixture of domain specific language to express tasks and Groovy to write your own tasks. This has an influence on expressiveness as for example [8] shows: for a single project the build scripts in the form of Ant XML, Maven XML and Gradle DSL ranged from 138(Ant) over 73(Maven) to 38(Gradle) lines of code.

A best practice modern build systems rely on is the principle of convention over configuration [9] which means that standards are defined by the tool and used by default. This means that targeting certain directories for certain tasks are obsolete and dependencies only need to be declared once. With the creation of the Maven Central repository [10] dependency management is handled implicitly by the build system in the sense that only the relevant (as in actively used) dependency needs to be expressed and all transitively required dependencies are resolved automatically.

The choice for a build system is usually expensive to revoke as transitioning from one build system to another requires a lot of work (as in reengineering). A careful study of requirements and tool specifics is essential for the choice.

1.2.5 Continuous Integration

Continuous integration has its origin in extreme programming [11] where developers merge their workspaces several times per day. It is a practice used to quickly find and resolve issues resulting from changes and to keep a common workspace for everyone in a working state. The most extreme variation of continuous integration is real-time integration [12] where builds are executed as soon as changes to the repository are observed.

A continuous integration tool can usually be seen as a bot. It idles until a trigger (such as the change to the observed repository or a point in time) occurs and then executes a relevant job. These jobs can usually exceed the invocation of build systems and - depending

on the tool - offer a variety of services like e-mail notification on errors or the release of a freshly built version of the project for use to a public repository [13].

The choice of a build or revision control system in the context of continuous integration is a relatively cheap decision since a job can be configured to use any kind of (supported) tool. The configuration required for such changes is - in modern continuous integration servers - usually limited to a few clicks or for natively unsupported tools to a single additional installation for use.

1.3 Chrestomathies

1.3.1 Definition

"[...] a program chrestomathy is a collection of similar programs written in various programming languages, for the purpose of demonstrating differences in syntax, semantics and idioms for each language." [14] With the rise of many new technologies the use for chrestomathies has expended beyond software language comparison towards bridging the gap between technological spaces [15]. Their purpose as an educational system for learning, comprehension and comparison remains untouched but the realization, maintenance and extension is more challenging than ever.

1.3.2 Properties

Chrestomathies use a theme to preserve comparability. Such a theme defines the common structure and/or architecture of all implementations. The definition and realization of such a collection is challenging in the sense that strong conventions are required for comprehensibility and comparability which contradict with the problem of technological spaces and their very own themes. The principle of separation of concerns on the one hand is more important than ever yet its applicability is not well-understood in science or industry (most likely as a result of insufficient use cases). Research on technological spaces and their relations has just begun and so the challenges that modern chrestomathies face are very young, mostly undefined and not well-understood.

1.3.3 101companies

101companies is a relatively new chrestomathy with the goal of better understanding (and teaching) differences and commonalities of technologies from different technological spaces. It uses the theme of companies that consist of departments and their employees as a data model and uses features like totaling or halving the salaries of all employees to express functionalities [3]. It maintains a wiki [17] and a repository [16].

1.4 Goal

The goal of this thesis is to bootstrap a solution for problems discussed in the next chapter from a requirements engineering view using relevant related work. Potential future improvements are to be discovered and prepared if possible. The scope of the thesis is - due to the complexity - limited to Java implementations of the 101companies chrestomathy. The layer of revision control is omitted in the discussion ahead since github is used consciously and represents a cheap decision in the sense of continuous integration in section 1.2.5.

Chapter 2

Requirement Engineering

2.1 Scope

The build process for chrestomathies relates to a specific set of requirements different from those found in build systems for traditional software [18]. 101companies is a community project that provides an infrastructure to develop and access knowledge about software languages and their technologies. It is not a closed software project with the goal to deliver a closed product with well defined use cases. It is clearly improbable to be able to foresee all values that the project will or might offer. With its openness one has to realize that the build process might not only be a simple utility but rather a product of its own.
In a standard software project non-professional users are supposed to use the final, closed product (as in executable software) as opposed to many parts of the build process (including compilation, preparation, generation, setting up databases, integration, test execution, ...). The use of a build script is - under this assumption - not only restricted to usage within the development process but becomes part of the shipped product. At this point any stakeholder needs to be considered. A potential solution for build management needs to take these conditions into account.
Another property of chrestomathies is the desirable simplicity of implementations. The biggest issue for build systems in larger software projects (see for example glassfish [19], an open source application server) is to handle the complexity of dependencies between modules and subsystems that results in complex hierarchical dependency trees. Chrestomathies use a rather flat architecture with a lot of independent modules (implementations) with limited complexity as exemplified in figure 2.1 on page 12. Though complexity is limited a chrestomathy faces a different challenge namely the broad variety of distinct dependencies. A system for build management should be able to cover most dependencies by default in some way and should be extensible for new and unsupported technologies. The size of its community also has an impact on this aspect of build systems.

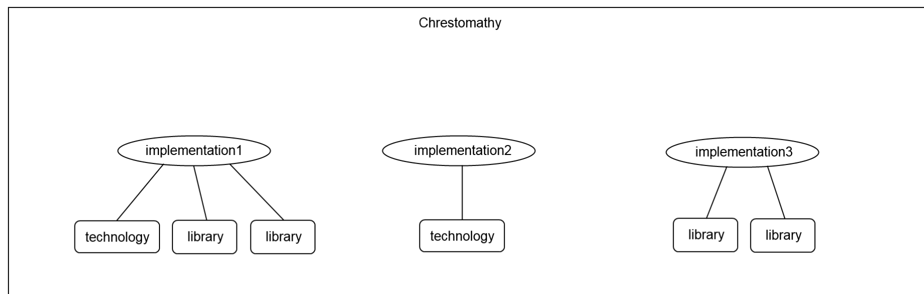


Figure 2.1: Flat architecture with independent modules

2.2 Stakeholders

For a detailed analysis of requirements we first have to examine the most important stakeholders. As of J.-M. Favre *et al.* [3] there are four different groups:

2.2.1 Users

Definition

Users are those stakeholders that use any of the components of the 101companies system (wiki [17], repository [16] or other services like the discovery service [20]) for educational purposes. We can differentiate between learners and teachers where the latter inherits requirements from the former.

Concern

Focus of this group is the provided service of the 101companies system: discovering software languages and their technologies in an accessible and structured manner, exploring alternatives to and advantages of different approaches for technological issues using a consistent model with uniform implementations, comparing language specific technologies in a consistent environment and learning applications of technologies in software languages in an easy-to-use fashion with high accessibility and simplicity.

Relevance

In the context of this thesis users define the lower bound of the 101companies build management effort as to preserving and improving basic build related issues such as resolving dependencies, test execution, preparing and compiling implementations, making the code discoverable and readable and making the execution of implementations simple. It is obvious that not all individual needs of users (such as supporting large amounts of IDEs) can be satisfied and therefore a simple solution with strong conventions is needed to make it easy for users to adjust implementations to formats usable with their tool of choice.

2.2.2 Contributors

Definition

As opposed to the users of the 101companies system contributors add to the chrestomathy in various ways. J.-M. Favre *et al.* [3] differentiate between 7 groups of contributors. One can classify these groups into internal and external roles - as to whether they contribute to the 101companies system directly (in terms of architecture and services) or to its content. The latter should be accessible to anyone and should therefore be seen as external.

Concern

The emphasis for these stakeholders has a diverse set of use cases. Engineers will be interested in any aspect of the 101companies project - as for example this thesis covers internal and external aspects - while developers of implementations might only be working with the 101companies repository [16] and wiki [17].

Relevance

Even though building and testing are part of the development process [21] we must assume that developers are not particularly interested in the build process itself (as far as build implementation goes). Compilation and test execution nowadays are widely integrated in IDEs [22]. This results in a lower bound similar to the one of users. Build efforts should not harm convenience for contributors but rather play a helpful role by introducing simple standards and notational and structural conventions. Additionally other requirements can be identified such as maintainability, scalability and extensibility.

2.2.3 Researchers

Definition

The 101companies project has a lot of potential for researchers in computer science. J.-M. Favre *et al.* [3] specifically mention software linguists and ontologists but there might be an even broader audience of researchers e.g. seeking statistical data.

Concern

Potential interests of this group could vary and span over all aspects of 101companies. This might even go beyond the theme of technologies and into architecture and modeling of the system itself.

Relevance

There are three potential use cases for building concerning researchers: Either building is irrelevant (as in statistical studies), it is a tool for usage as it is for users or it is a subject of the research. Once again building efforts should not interfere with researcher's interests and possibly support accessibility and comparability of data.

2.2.4 Technologists

Definition

Technologists develop new technologies such as tools, libraries, APIs or frameworks.

Concern

Chrestomathies add an opportunity for creators of software technologies or even languages to host, present, exemplify and document their work to address a wide audience. 101companies is used as a platform for advertising and publishing and puts a new technology into context of other given technologies with similar goals.

Relevance

This group could be bringing in new implementations or ideas for the 101companies system. Their use cases relate to those of contributors since they either contribute directly via an implementation of their technology or indirectly through ideas, hints or pointers. Building efforts should not pollute implementations and keep technology-representations as pure as possible. Requirements for this group usually are a subset of requirements for contributors.

2.3 Requirements

We will now define and discuss requirements for build management for 101companies.

2.3.1 Build Automation

Definition

The build process should be automated.

Explanation

The automation should work in a way such that a build system can be invoked to fully build the project. This has to cover all steps of the build process (preparation and compilation, packaging, test execution, deployment, creating documentation).

Discussion

The initial situation in the chrestomathy is inconsistent as figure 2.2 on page 15 shows. Most implementations either have no build automation at all (45) or are implemented as eclipse [23] projects (42). Less than a third uses make [24] (38) and a small fraction uses ant [25] (4), maven [26] (2) or sbt [27] (2). This results in a coverage of around 45% automation.

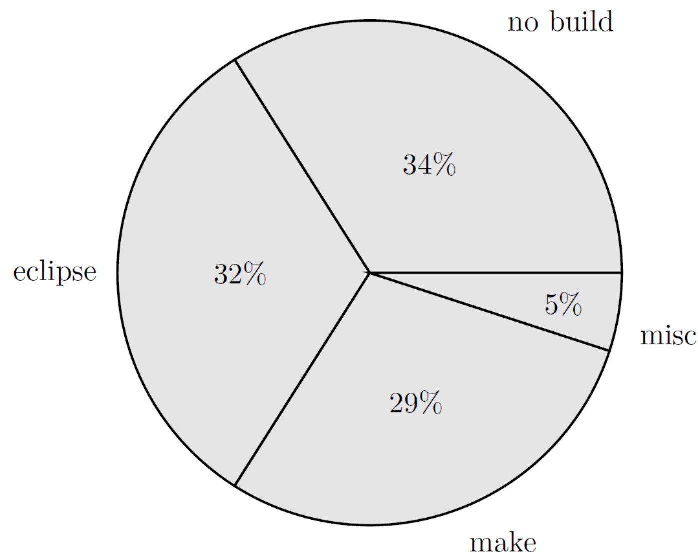


Figure 2.2: Build systems initially used in 101companies

2.3.2 Test Automation

Definition

Tests should be executed automatically. Test results should be discoverable.

Explanation

Test execution is part of the build process. Yet results should be persistent such that they can be viewed. They should also have an established representation (e.g. using the XML format).

Discussion

Only implementations using a build system with a given build life cycle that includes testing using a testing framework (such as maven and sbt) offer test automation which results in approximately 2% test automation over all implementations.

2.3.3 Neutral Build

Definition

Builds and tests should be done in a neutral environment.

Explanation

A system for verification of build and test automation should be neutral [28] in the sense of not being used for any other purpose and therefore being free of hazards [28].

Discussion

There is initially no system in place for this requirement to apply.

2.3.4 Test Coverage

Definition

Tests should cover and therefore prove the functionality of features of the 101companies model [29].

Explanation

Features of the 101companies system represent a sequence of instructions necessary to achieve a certain goal using a specific technology or scheme. Tests should prove that these sequences actually work as intended. As opposed to regular software it is sufficient to only test features as opposed to every method using all possible use cases. 101companies does not and should not claim to provide flawless programs since this would imply a prove for functionality of the technology/technologies used.

Discussion

Test coverage of Java implementations of the 101companies system is initially incomplete. Most implementations cover all of their features. Yet - also corresponding to the problem of heterogeneity in the model of implementations - neither could all features be discovered easily nor could their tests.

2.3.5 Notification Automation

Definition

Results of a build including tests should be propagated automatically.

Explanation

A chrestomathy and its technological space(s) [15] continuously move. As implementations, their technologies and even their languages undergo changes proper functioning can't be guaranteed by default. Test execution can show malfunctions but these may not be discovered in time. In case of failures there should be a system in place that notifies persons in authority automatically via appropriate communication channels (e.g. email).

Discussion

As a consequence of low automation coverage for building and testing there is currently no opportunity for automated notification.

2.3.6 Dependency Management

Definition

Internal and external dependencies should be described effectively.

Explanation

Internal dependencies are derived artefacts such as generated files whereas external dependencies are retrieved artefacts like libraries. Both internal and external dependencies should be described as part of the build. Resolution as well as application should happen at build time and as part of the build process as opposed to storage of said artefacts. The repository of the 101companies system should contain the implementation of a usages of technologies and only that. Obsolescence of ”snapshotted” artefacts should be removed. This requirement also correlates with maintainability.

Discussion

As of the beginning of this thesis all implementations with external dependencies contain those within their directory - either in the root or a subdirectory. This includes implementations that are technologically capable of fulfilling this requirement e.g. using maven.

2.3.7 Limited Heterogeneity

Definition

Technologically similar implementations should be similar in terms of build system, dependency management, testing, file and directory organization.

Explanation

All implementations of the 101companies system pursue the same goal of defining models and features using technologies. They all follow the same scheme of companies, departments and employees and features therefore. As a consequence all implementations should follow this scheme also in terms of naming. A user of the 101companies system should only need to understand structuring once.

Discussion

Researches using the Java implementations have shown 7 different names of packages containing specific features where sometimes more than one was present and more than 10 different names for feature and test classes with none following (Java) naming conventions. Test class names either had nothing in common with feature class names or were named the same. Additionally all technologically different implementations seemed to be close to their technological space assuming a given interpretation of names.

2.3.8 Comprehensibility

Definition

Comprehensibility should not be affected in a negative way.

Explanation

Understanding how an implementation works should be an easy task. This includes both the source code itself as well as the procedure necessary to execute it.

Discussion

With the given heterogeneity in 101companies comprehending a new technology also implies comprehending its specific technological space. Even though this might be intended as a small representation of the naming conventions of its referring technological space the fact that naming of features is inconsistent throughout implementations that functionally do the same makes comparability and comprehensibility unnecessarily difficult and prevents users from accessing new areas of interest.
Additionally most of the implementations don't specify their execution and require the user to follow instructions on the 101wiki, about a third is tedious to comprehend as users have to descend directories to hunt down Makefiles for information.

2.3.9 Comparability

Definition

Implementations should be comparable assuming technological and linguistic similarity.

Explanation

If two implementations approach the same problem (for example serialization using XML) using different technologies comparing differences in structure or code should be easy. Functionally equivalent parts of the code should be in equivalent places of directories or even language-specific namespaces (such as packages in Java).

Discussion

The usage of different names and package names for features and the inconsistent and incomplete build automation make it impossible to directly compare implementations (if they are not developed by the same person within a short period of time to keep it set-wise consistent). Without comparability between (similar) technologies 101companies will be a lose collection of implementations as opposed to a "[...] structured [...] knowledge resource" [3].

2.3.10 Discoverability

Definition

Implementations and their artefacts should be easily discoverable.

Explanation

The structure of implementations should be intuitive and logically close to the idea [17] described by the 101companies system. Users should find what they expect in directories simply by looking at the name and - in reverse - should know where to look for certain aspects of the implementation.

Discussion

The current heterogeneity and pollution (with internal and external dependencies) of the 101companies system's implementations make them difficult to navigate. A lot of effort is required to see what exactly is unique to an implementation and where to find it. Functionally equivalent parts of different implementations are implemented differently in the sense of modularization intensity and method. Fully understanding the structure of one implementation only helps for comprehension of some others where several contributions can not be discovered easily and require adaptation to new architectures.

2.3.11 Usability

Definition

Implementations should be easy to use.

Explanation

Users should not need to worry about build specific actions if they don't intend to and there should be as few and as simple steps as possible to successfully build and test. Additionally implementations should be automatically prepared for popular IDEs such that users can import and use them easily.

Discussion

Some implementations are already easy to use though that is assuming the use of eclipse or make (which is non-trivial for users outside a Linux distribution). Implementations that are prepared by the use of eclipse achieve so by implicit platform (in the sense of IDE) dependence and by shipping internal and external dependencies alongside the actual contribution. As both are undesired properties for open and community driven projects for many reasons beside accessibility and maintainability the implicit use of eclipse is rather a compromise than a solution. Using make offers build automation but since it is a rather old build tool (initial release 1977 [30]) with its roots in the UNIX environment it comes with a loss of accessibility and a few peculiarities [31].

2.3.12 Maintainability

Definition

Builds should be easy to maintain.

Explanation

Changes to builds should be easy to perform. This includes changes to dependencies (like upgrading to higher versions of technologies), changes to the structure of implementations, changes to the feature model of 101companies or even adoption of new build technologies. The principle of convention over configuration [9] is useful for understanding this requirement.

Discussion

Since 101companies does not initially use any conventions regarding directory layout or modularization implementations are tedious to extend or change also in relation to comprehensibility. External dependencies are held within contributions which makes updating a non-trivial task. Automated testing is effectively non-existent which makes finding problems as also related to system independence hard to detect. Additionally time cost for keeping the system fully functional is high since potentially every maintenance effort consists of manual work.

2.3.13 System Independence

Definition

The building process should work on different operating systems.

Explanation

All steps of the build process should be automated as of 2.3.1. This should not only be true for a specific machine but for all popular operating systems. Necessary preparations should be equally simple to make for every system.

Discussion

As make has its roots in UNIX parts of the build script might be incompatible with different members of the operating system families (Mac, Linux, Windows). Additionally it can be non-trivial to establish support for make on different platforms since other tools like Cygwin [32] may be required. The use of eclipse makes contributions operating system independent but also binds them to the IDE.

2.3.14 Portability

Definition

Implementations themselves should be compatible with different environments.

Explanation

Any implementation should be usable in the sense of the build process on any operating system. This might not always be applicable for certain technologies since they are intendedly system dependant.

Discussion

For some technologies that are intendedly system dependant there exist tools to enable them on different operating systems. This is for example true for the contribution *hadoop* [33] for windows. Required preparations however are disproportionally complex and documentation is scarce. Such preparations would either need to be automated as of 2.3.1 or properly documented as part of the 101companies system.

2.3.15 Scalability

Definition

Building should be efficient.

Explanation

As the chrestomathy grows more and more contributions need to be build. A system that executes builds and tests (as part of a 101companies service) should be able to do so in a reasonable amount of time. In order to guarantee reasonability the system needs to include a means for concurrent execution.

Discussion

The build system make and build mechanisms of eclipse - as most build systems - offer concurrent execution for independent tasks. The current situation however does not yet offer automatic build and test execution for integration purposes. Therefore scalability is not relevant in the current status of 101companies but becomes more relevant as build management efforts go on.

2.3.16 Extensibility

Definition

Build tools should be extensible.

Explanation

As more and more new technologies and languages emerge build tools should adept to them by adding new functionalities and support in some way. This could be in the sense of updates or extensions.

Discussion

With currently no automated testing this requirement is only relevant in the sense of build systems like make supporting new dependencies. By the nature of these tools invoking and including such dependencies can always be achieved with basic means like shell commands. With the desired automation of regular tests this requirement also becomes applicable to relevant tools.

2.4 Synopsis

At first the requirements above can be seperated into functional and non-functional requirements:

- Functional Requirements
  - Build Automation
  - Test Automation
  - Notification Automation
  - Dependency Management
- Non-functional Requirements
  - Neutral Build
  - Test Coverage
  - Limited Heterogeneity
  - Comprehensibility
  - Discoverability
  - Usability
  - Maintainability
  - System Independence
  - Portability
  - Scalability
  - Extensibility

The requirements above may target different parts of a system for build management. For example dependency management is commonly integrated in build systems whereas notification automation is realized by continuous integration tools. Also reengineering on contributions is required to meet several non-functional requirements.

Some requirements can be of opposing objective as for example comprehensibility and maintainability or usability and maintainability can not be both fully achieved. In fact tradeoffs are required. Maintainability in the context of build scripts necessitates abstraction for limitation of redundancy which for usability means that individual implementations requiring a common resource can not be examined for themselves. The same situation is harmful for comprehensibility when trying to understand what is executed and when it is executed in an implementation where such information is scattered over several build scripts. Allowing for such redundancy in favor of usability and comprehensibility hence reduces maintainability.

It is important to realize that even in the absence of such opposing goals it can be undesirable to fully satisfy a requirement. Take for example test coverage: in usual software products test coverage is used to preclude unwanted behaviour and prove correctness with many use cases and broad testing (e.g. many test cases per function or class). The actual usage of such software usually then happens through interfaces (e.g. APIs or GUIs). In the context of a software chrestomathy like 101companies this broad testing becomes of less importance as the concrete usage of the software at hand is limited to defined tests instead of potentially abusive exploitation and test cases only reflect usage patterns (as e.g. used in development by other programs). Implementations don't lay claim to complete correctness.

Chapter 3

Related Work

3.1 101companies

The 101companies project [3] is - by the time of this thesis - undergoing heavy reworking. A changing feature model [29] gives the opportunity to better understand required capabilities of reengineering efforts. Contributions should in some way provide a facility to feasibly be adapted to changes of the data [34] and feature [35] model. At this point modularization becomes a necessity. Depending on the technology/technologies and objective at hand the degree of useful modularization might be limited though as for example POJO [36] Java classes (say data structures) include behaviour (see for example contribution javaComposition) and therefore implementations of the 101companies feature model. This is explicitly intended. A common structure is needed nevertheless.

Furthermore the 101worker [37] is getting improved and extended which might offer opportunities for build related improvements in the future like for example including test results or adding specific fragments of files automatically to a wiki page. This trivial example would increase conformity and automation and therefore make the contribution process even simpler and more error proof.

The 101wiki [17] itself is also changing and new functionalities could be useful for documentation and propagation of implicit standards. A good example is the effort on a new contribution process web page for adding new implementations of technologies to 101companies. A build could be triggered (if build automation is supported) and the implementation verified.

3.2 Homogenization

3.2.1 General

Making software comprehensible is a common need in software development and is often undervalued. Maintaining and changing poorly designed and documented legacy systems is a challenging and costly reengineering effort. This issue becomes in fact much more relevant in the context of chrestomathies as comprehension of a specific implementation is a frequently repeated activity. Additionally users of the 101companies system are largely no specialists in reengineering. A need for a homogenous architecture and a common namespace for implementations is obvious.

As software languages become commonly used experts seek for optimization of development and improvement of software quality. Hence best practices [38] [39] and patterns emerge that improve qualities like comprehensibility, maintainability and changeability of code. Because of significant differences in software languages best practices mostly only apply to a certain family of languages. As a result homogenization efforts for 101companies can not achieve the same degree of conformance for all implementations. This could be a disadvantage for inter-language comprehensibility but also an advantage for gaining an insight in technological spaces and language specific properties.

An important principle in the context of homogenization is the separation of concerns [40]. Since 101companies implicitly offers a meta model for the programming task and requirements for data structures and behaviours concerns are well defined.

3.2.2 Java

Since this thesis focuses on Java projects for build management Java best practices [41] apply. It is important however to regard that some best practices might violate requirements or could simply be undesirable as some implementations are explicitly violating them by design (see for example contributions javaExorcism).

3.3 Build systems

3.3.1 Scope and Relevance

Scientific work on build systems is scarce and focuses heavily on specialized topics. Fundamental work has been done many years ago (see for example [42]) and is already well implemented in build tools or too specialized to find application at all. This makes it hard to find information applicable on 101companies and chrestomathies in specific that are not yet implicitly covered by features of popular build systems.

Another problem with most build systems [6] is their origin in a technological space that defined their use cases and resulted in high specialization and low flexibility or support for technologies. A few build systems have prevailed for use across technological spaces which will be briefly discussed in this section.

The main difference that needs to be considered once again are the special requirements for chrestomathies that demand for simple usage, broad support for integration of new technologies, potential efficiency and maintainability as opposed to one project specific capability. An all round solution may be preferable to specialized tools though the system needs to be open for changes and integration of tools required by specific technologies at any time.

The most common build systems will be supplemented by a newer approach in the shape of Gradle. Make is present in this section since it is used by many 101companies implementations initially and therefore needs to be considered.

3.3.2 General

Build Maintenance Effort

A focus on scientific work is the evaluation of effort required for maintenance of build systems [44] which showed that coupling between production or test files and a build system in place is low to medium. With 101companies build management using a rather low

architecture this promises even lower coupling and therefore higher maintainability. With build systems enforcing conventions for the structure of projects tasks allow for flexible structures and therefore a changing architecture. This is useful for naming conventions and adaptability to a possibly changing and extending feature model of 101companies.

Another current focus in science is the question of ownership assessment as to whether build management should be performed by all developers concerned or by a specialized build management team [44] [45]. This might be interesting in the context of contribution and integration of new implementations in the future.

Build Code Analysis

A recent effort [46] has expressed the need for code analysis like it is widely used already for software languages in IDEs in the context of build scripts. It shows a solution for symbolic evaluation of Makefiles and hopefully initiates efforts on tool support for build script development in general. Experiences with this thesis have shown that the creation of correct build scripts and debugging of issues caused by a change in or first migration into a build system is a very time intensive task. This is especially appropriate if many technologies with many different properties need to be maintained.

As a result this increases the relevance of build maintenance effort for chrestomathies and implies cautiousness with estimation of required resources to maintain and operate a software chrestomathy with quality.

Scalability

Excessive builds for large software projects have created a need for efficiency. This need arose approaches for improving build time and therefore reducing idle time of development. The most common approach is persistence in builds [42] [43] in the sense of only taking changes into consideration for a successive build. This has lead to splitting a build into atomic build steps (tasks) that can be called independently and only need to be executed again if relevant resources have been affected by a change.

Such principle is nowadays common in build systems and useful for potential scalability issues with the 101companies chrestomathy. In combination with a revision control tool like Git this allows for efficient, fast and highly responsive builds.

3.3.3 Make

Make [24] is a family of build tools that exists since 1977 [24]. Make tools are widely used in the Unix environment and rely on command line expressions that are being executed under certain conditions. Therefore make tools have limited functionality on windows as certain unix commands simply do not exist. Make does not have a life cycle and hence requires sequential definition of build steps. It manages task dependencies but does not offer transitive or cascading dependencies. It is still very popular for C-related projects but is replaced elsewhere by Ant or Maven.

3.3.4 Ant

Ant [25] is a build system written in Java and therefore platform independent. It was originally created for building Java projects but ”can build anything that can be described in terms of targets and tasks” [25]. It uses XML for build scripts and allows for custom tasks being implemented in Java. It does not have a life cycle and also needs all tasks to be

defined manually. Ant allows for transitive dependency management but does not natively support the maven central repository. Ant is part of maven as it is used to model custom tasks.

3.3.5 Maven

Maven [26] is a software project management tool also written in Java and originally intended for building Java projects. It has all the functionalities of ant and can be seen as a successor. Build scripts are also implemented in XML and custom tasks can be expressed via ant. In addition to defining execution and dependencies of source files it also allows for an automatic management of external dependencies using the maven central repository. Maven implements a default life cycle extensible via plugins that models standard tasks that can be customized further. A port of the Gradle wrapper for maven is available (automating the installation of maven).

3.3.6 Gradle

Gradle [47] is a build automation tool that is written in Groovy. It merges ideas of both Maven and Ant to achieve flexibility as custom ant tasks and to enforce conventions via a default life cycle (also extensible via plugins). A build script is written in a Groovy domain-specific language which combines both a standardized way to call tasks and the implementation of custom tasks at build script level. Gradle supports the maven central repository. It comes with the possibility of automating the installation of Gradle [48] in order to standardize the version used for building.

3.3.7 Synopsis

As will be shown later in this chapter there is no need to rely on only one build system as continuous integration tools offer support for several of them via plugins and all of them via manual installation on the build server. A desirable solution would involve a consistent build system for all implementation but with regard to the diversity implied by the 101companies system and its technological spaces and technologies this is an unrealistic belief. However language or technology specific choices have to be made. With the requirements of chapter 2 Make with its system dependence is inappropriate. Ant misses out on dependency management features to clean the repository of external dependencies and with this the choice is between Maven and Gradle as a dominant build system for 101companies especially in respect of exemplified Java contributions. As Gradle offers a better way for build customization (integration of custom tasks in the build script) and the Groovy domain-specific language is syntactically more discoverable this thesis will make use of Gradle.

3.4 Continuous integration

3.4.1 Scope and Relevance

As with build systems scientific work on this topic is scarce and specialized or fundamental and mostly implemented in continuous integration tools. The decision for a specific tool weights much heavier as a continuous integration tool is the central tool of build management. It bridges lower level systems like build systems and revision control and deals

with organizational subjects. It should be extensible and offer functionalities to meet the requirements defined in chapter 2.

3.4.2 General

Integration

There are a few books dealing with continuous integration in general such as [49] that show advantages and applications. Scientific papers such as [50] approach build system integration anew and present a redundant implementation of functionalities of recent CI servers. With the notion of jobs and the ability to express dependencies between them the idea of build system integration is implicitly implemented. With the splitting of build processes into atomic tasks this offers the composition of any build system with any other at any point in time during the build.

Scalability

Build systems already offer scalability in the sense of executing tasks only if necessary. With continuous integration tools this property and requirement can be improved even further with parallel processing of jobs [51] [52]. This allows for highly efficient builds especially in the context of chrestomathies. Implementations are (or at least should be) independent from each other and can be build in a distributed manner. This allows for vast numbers of implementations to be feasibly managed assuming appropriate resources.

3.4.3 CruiseControl

CruiseControl [53] is a continuous integration tool. There are three variants available with two being specially implemented for .Net and Ruby. It offers in its general variant integration for most revision control tools, native integration for a few build system (Ant, Maven, OpenMake, NAnt), scalability features, notification automation via several channels (such as e.g. Email, IRC or RSS) and more features [54]. Existing jobs can be viewed and executed via a web interface but are created and configured via XML locally. CruiseControl offers a plugin architecture for the addition of more features.

3.4.4 Hudson

Hudson [55] is an extensible continuous integration server that like CruiseControl supports other build management tools and offers a web interface but has a higher number of features such as management of builds using a web interface and a bigger number of supported tools. It was a popular integration tool from 2008 until 2012 when it was moved from github to the Eclipse Foundation [56]. ”There were very few commits to the source code after May 2012.” [57]

3.4.5 Jenkins

Jenkins [58] is another continuous integration server that offers extensibility. It was created as a fork of Hudson in 2012 by its original developers [58] and has since then a larger community. Hence it offers the same functionality as Hudson but supplies a much bigger number of plugins and a more active development.

3.4.6 Travis

Travis CI [59] is a hosted continuous integration service for open source projects in GitHub. It offers the infrastructure to build and test implementations on a shared and distributed system of virtual machines that are reinitiated from an image before every execution [60]. Customization happens at shell layer since unsupported features need to be installed on the machine first via e.g. apt-get. Additionally shared means that intervals from build trigger to build execution may vary depending on traffic.

3.4.7 Synopsis

With respect to the requirements specified in chapter 2 and the overview of features [54] Jenkins and Travis need to be considered. Travis offers a complete service with powerful customization at build time but therefore loses speed and persistence. Build results are published on the web site of the provider and can only be discovered to the degree they are implemented. Also any integration in the 101companies system like for example the 101wiki can be problematic. At this point Jenkins offers the most broad support of tools and technologies with highest customization and lowest integration cost.

Chapter 4

Design

4.1 Scope

As this thesis focuses on Java implementations the next section will be most relevant to object oriented languages. Conventions in naming and structure can however be used to model other architectures specific to the language at hand and its properties.

4.2 Homogenization

4.2.1 Structural Conventions

Java offers classes and packages for modelling structures and define access privileges. As classes are in a part of relation to packages this allows for an implementation of the 101companies feature and data model. A company, department and employee are part of the data model whereas functionalities are part of the (functional) feature model. This requires two packages. These packages directly contain feature respectively data code as a representation of their capabilities in the 101companies system.

This approach however is limiting in the sense that technologies might need their own infrastructure within the implementation. This is already obvious for design patterns [61] such as the visitor or template pattern.

For this case subpackages should be allowed that are located within the package they naturally belong to (feature model vs data model). As a visitor is an arbitrary representation for a behaviour or function it should be located in a subpackage of the package containing functional features. This allows for any kind of customization without violating the simple convention essential for comprehensibility and comparability.

4.2.2 Naming Conventions

As Java projects should create their own namespace a main package is needed. This would be by convention *org._101companies* which is arguably undesirable. For historical reasons the name *org.softlang.company* will be chosen instead.

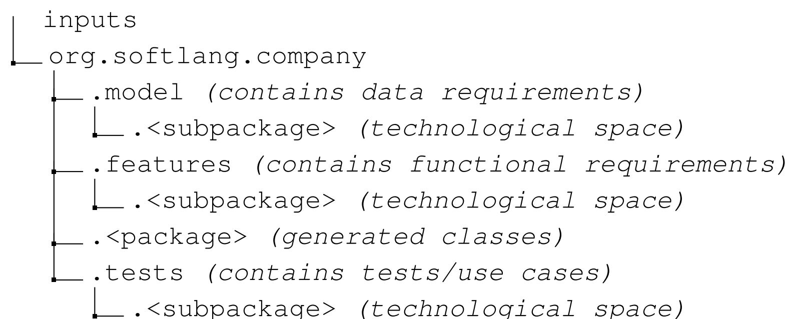
High level names such as package and class names should be consistent as long as they represent features of the 101companies feature namespace [29]. Implemented features should preferably be contained in a class with the feature name however this is not always possible as for example plain old Java objects integrate behaviour and data into the same

class. To keep the convention as a representation of the 101companies feature model, class names with implemented features should exist nevertheless. This could be realized via simple forwarding methods which also provide useful pointers for high accessibility. This would in turn also implicitly standardize usage for tests. Technology-specific names for classes, methods or attributes can not be prohibited without major abstraction from technological spaces which is still a mostly unknown and active field of research. These inconsistencies however must not pollute feature and data model packages and should be used consciously. Such technological space could be represented within subpackages without harming comprehensibility for chrestomathy related models while at the same time giving insight into a specific technological space. The name of the subpackage should be appropriate to the technology used. Since generated files may contain feature and data model (depending on the technology) such generated files should generally be located in their own package in *org.softlang.company*.

Files used for execution of the program should also have a standardized location. Since these can be cross language and are mostly used to execute tests they should not be put in a package but rather collected in an intuitively named folder (which usually is "inputs"). Generated outputs should - for consistency - end up in a folder named "outputs" next to the inputs folder. Test cases can be handled in different ways. Modern approaches mix production and test code within one package for higher discoverability and changeability but this is mainly done for the purpose of complete testing. Implementations of a chrestomathy are use cases for technologies and don't require deep testing (especially because of their simplicity and limited application field). Tests also don't need access to package restricted functionalities since once again application of the implementations is strongly limited. Test names however should use the conventions above and should by default be automatically executed for tests without notion (which results in *<feature>Test.java* since it is used for Maven, Ant and Gradle commonly) by most build systems.

4.2.3 Result

With the simple conventions above we get a standardized structure for potentially all possible implementations which is highly declarative:



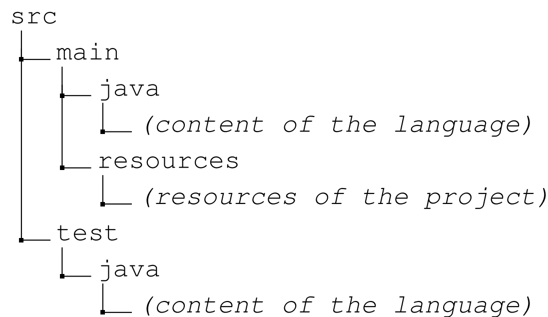
Naming is well defined for contents of standardized packages by the 101companies feature namespace [29] and free for representation of technological space in subpackages. This allows for applicability for any implementation relevant in the context of 101compa-

nies.

4.3 Build systems

4.3.1 Structural Conventions

Gradle as all other modern build systems use a structure and naming convention for software projects. It allows for undeclared implicit locating of source code for compilation and makes implementations and their build script more discoverable and predictable over all projects following these conventions. Such conventions usually don't need a notion as they are implied by the build system used. The conventions for java look like this:



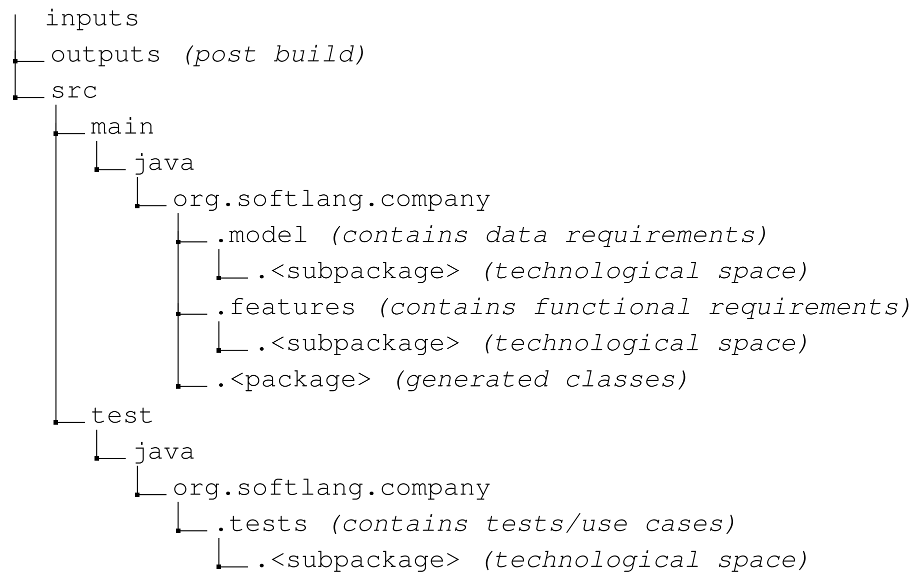
By default Gradle uses "src/main/resources" for inputs. Output files will be placed into "build/classes/main". This is unintuitive and harms discoverability. With the conventions in chapter 4.2 and with outputs not being part of the source (which folder src implies) inputs and outputs should stay in the root folder of the project. Build related information such as test results and compiled classes will be located in the folder "build" post build.

4.3.2 Architecture

Gradle build scripts offer nesting for inheritance of properties and therefore reduction of redundancy (all subprojects will inherit the same properties from the superproject). This is useful for maintenance. However it lowers discoverability, comprehensibility, usability and makes project builds depend on the higher level build script. Redundancy reduction of inheritance can also be achieved with the use of the 101worker to synchronize build scripts automatically (for example as in versions used). To improve usability the Gradle wrapper [48] will be used.

4.3.3 Result

Resulting from the additional conventions and build system specific conventions the structure will look like this: For further naming conventions see chapter 5.



4.4 Continuous integration

4.4.1 Properties

With the use of Jenkins as continuous integration server build management becomes significantly easier. Jenkins offers support for Gradle and Git via plugins. The same holds for other build systems [62] and revision control tools [63]. For unsupported tools Jenkins offers native support for shell scripts (and Windows batch commands). This allows for any build system to be used and therefore is highly extensible and changeable if the need arises. Jenkins also offers plugins for other purposes like notification or even UI improvements and new UI views (for e.g. publishing test results). For a complete list of plugins see [64].

4.4.2 Architecture

A build server is to be set up to run a Jenkins instance. This instance will - for demonstration - build projects affected by build efforts nightly and run on Ubuntu. Technical details to the setup will be explained in the next chapter.

Chapter 5

Implementation

5.1 Homogenization

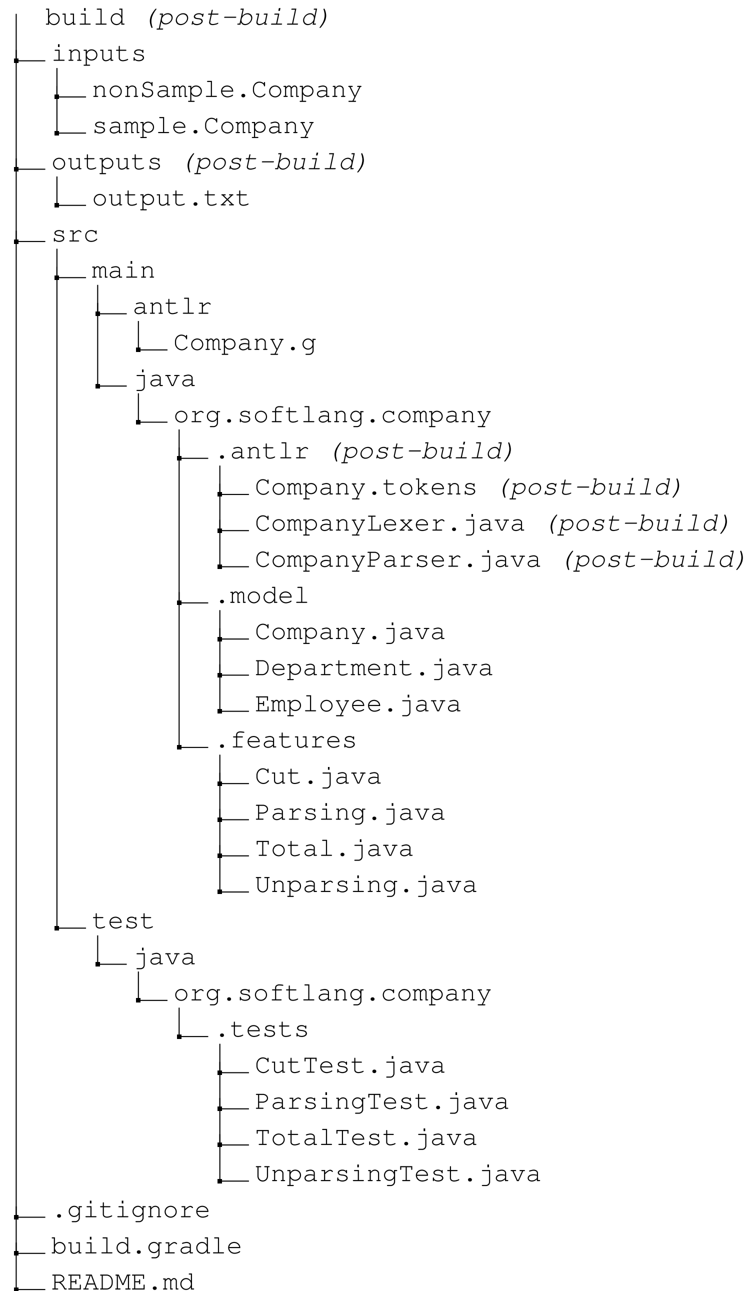
Affected by the homogenization were 27 implementations. They were chosen by a categorization of all Java implementations in terms of complications for build automation. Implementations with dependencies on execution environment (such as eclipse plugins) or additional software installed (such as web applications) were beyond the scope of the thesis. Affected implementations were reengineered to meet the requirements and conventions discussed before.

5.1.1 antlrObjects

As an example the contribution antlrObjects is presented. The structure of the contribution before:

```
│   .settings (eclipse-related)
│   └── .org.eclipse.jdt.core.prefs
├── inputs
│   ├── nonSample.Company
│   └── sample.Company
├── org.softlang
│   ├── company
│   │   ├── Company.java
│   │   ├── Department.java
│   │   └── Employee.java
│   ├── parser
│   │   ├── Company.g
│   │   ├── Company.tokens
│   │   ├── CompanyLexer.java
│   │   ├── CompanyParser.java
│   │   └── Makefile
│   ├── pp
│   │   └── PpCompany.java
│   └── tests
│       ├── Basics.java
│       ├── Parsing.java
│       └── PrettyPrinting.java
├── .101meta (eclipse-related)
├── .classpath (eclipse-related)
├── .gitignore
├── .project (eclipse-related)
├── antlr-3.2.jar
├── COPYRIGHT
├── Makefile
├── output.txt
└── README.md
```

And after enforcement of conventions:



It is - with the assumption that the viewer is familiar with the theme of 101companies - obvious that the new approach is more declarative, closer to the feature model and can be consistently used between implementations. Related classes are grouped and function-

alities contained in clearly named artefacts. Additionally changes to the feature model can be applied easily and new features implemented in a modular and independent way.

5.1.2 Documentation

Documentation has also been improved. The 101companies wiki now includes an illustration based on the 101companies feature model and follows a common structure for affected implementations. See antlrObjects for an example.

5.1.3 Problems

Some implementations might purposefully violate best practices and code conventions. This is for example true for contribution javaExorcism. Such contributions can not be fit into any convention and require individual and cautious structuring and naming. Another issue for chrestomathies was revealed when migrating antlr implementations from Make to Gradle. An inappropriate command line argument was the reason for system dependant directory errors during the build. The error was searched for on contribution level and couldn't be resolved until a coincidental research on antlr resolved the confusion. This shows the value of strong conventions that create confidence on the contribution level. Maintenance can be split into different layers (contribution, build, integration) to reduce potential work.

5.2 Build systems

5.2.1 Setup

Affected contributions were migrated to Gradle. This includes migration/creation of build scripts and usage of gradle default directory layout (with the exceptions from chapter 4.3). The Gradle wrapper [48] was used to enable installation automation and the resulting shell and batch files modified and the required jar file moved to tools to keep the contributions folder of the repository clean from non-contributions. Build scripts that were at first complicated were trivialized to the degree of high readability. The whole relevant code for building for contribution antlrObjects looks like this:

```
1  apply plugin: 'java'
2
3  repositories {
4      mavenCentral()
5  }
6
7  dependencies {
8      testCompile group: 'junit', name: 'junit', version: '
           4.11+'
9      compile group: 'org.antlr', name: 'antlr', version: '
           3.2'
10 }
11
12 task generateFromANTLR(type: JavaExec) {
13     inputs.dir file('src/main/antlr')
14     outputs.dir file('src/main/java/org/softlang/company/
           antlr')
15
16     classpath = compileJava.classpath
17
18     main = 'org.antlr.Tool'
19     args = ['-fo', 'src/main/java/org/softlang/company/
           antlr', 'src/main/antlr/Company.g']
20 }
21
22 compileJava {
23     dependsOn generateFromANTLR
24 }
```

A single task executed prior to the compilation to generate files. Dependencies JUnit and antlr are downloaded via the Maven Central repository.

5.2.2 Usage

Ease of use is an important requirement for users. This is ensured by the Gradle wrapper which only requires a Java JDK installed and the set JAVA_HOME environment variable to be executed. Building is then as simple as the command line *gradle build*. Eclipse is supported by the usage of *gradle eclipse* after build.

5.3 Continuous integration

5.3.1 Setup

The build server was set up using an Ubuntu server in a virtual machine. Java was installed as a prerequisite using apt-get. Jenkins was then installed similarly using the guide on the Jenkins wiki [65]. The standard port for http (80) needed to be redirected to Jenkins-native (8080) to avoid Ubuntu user and port right conflicts. Because of initial issues with the Git plugin for Jenkins Git was installed manually via apt-get and set up using the Jenkins web interface. Access management for Jenkins was enabled to ensure read-only access for unregistered users.

5.3.2 Usage

Usage for the Jenkins server is simple. The user interface is by the time of this thesis available under http://build.101companies.org and can be used to manage jobs and install and setup new plugins (assuming appropriate rights).

Chapter 6

Discussion

6.1 Homogenization

6.1.1 Beyond Simple Java

Naming and structural conventions for languages other than Java can be derived using those best practices specific to the language and the feature model of 101companies to ensure homogeneity. The highest level of abstraction (e.g. file name) should be derived from the 101companies feature model. The second highest level (e.g. function names) can then be used to reasonably model characteristics of the technology at hand. It is however desirable to stick to a common convention throughout technological spaces and software languages. The separation of concerns should always be considered and some degree of modularization can be essential to keep a consistent and comprehensible theme.

6.1.2 Consequences

The extension of build management efforts on more parts of the 101companies system requires a new discussion based on the software language. The efforts from this thesis can help with a general theme but language specific characteristics can make conventions not apply and require a new set of conventions (with respect to the feature model). For further improvements a large scale study on technological spaces would be necessary to better understand naming issues (e.g. inconsistent naming for Un-/Parsing and De-/Serialization ”in the wild”).

6.2 Beyond Gradle

No build system is a jack of all trades and depending on the software language a different build system might be more applicable. With the use of Jenkins this is not an issue as every tool can be invoked from a Jenkins job (even if it is just a shell script). Even though it may be desirable to stick to a single build system throughout the whole chrestomathy it is an unrealistic view and the use of an additional build system for another purpose should not be seen as a bad practice. One must always consider that chrestomathies face different challenges than most software projects.

6.3 Beyond Jenkins

Potential problems can occur when trying to automate the integration of web application or other projects that require software to be running and accessible during a build like databases. This is a big challenge for chrestomathies as a build server can not be simply customized to the point of supporting a single project. For certain tasks there exist tools in the shape of Jenkins plugins like the JBoss Management Plugin to manage a JBoss application server at build time but with the broad variety of potential contributions and the little support in general for small or unknown tools build management for such implementations will face research and engineering intense challenges.

Chapter 7

Summary

In this thesis an incremental solution for 101companies build management was bootstrapped. Several steps were performed:

- Requirements for build management of the 101companies chrestomathy were discovered and discussed given the previous situation. Differences between chrestomathies and usual software projects became apparent as the build process itself becomes part of the software and therefore plays a crucial role in several use cases.
- Related work was identified and evaluated to get an overview of available tools and relevant scientific knowledge (which turned out to be scarce). Most scientific work is either very specific and mostly non-applicable for the 101companies chrestomathy or a standard in current tools.
- Tools were compared and evaluated with respect to the previously discovered requirements. Gradle turned out to be the build system which applied to the most requirements and Jenkins showed to be an open and extensible tool for continuous integration.
- Best practices were examined in terms of applicability and feasibility. This revealed a limited applicability on chrestomathies due to a specific set of requirements.
- A solution was designed and implemented. 27 Java contributions were reengineered for homogenization, improved in terms of documentation, migrated to automated building with Gradle and used to demonstrate continuous integration with Jenkins.

Chapter 8

Future Work

Potential improvements are diversified. The most obvious ones include the integration of all (feasible) Java projects and all other contributions to use the conventions defined before and to be included in the test automation(e.g. Haskell implementations [68]). With the ongoing development of new services in the 101companies system documentation, automation and maintenance of contributions can be improved further. Conversely conventions on implementations open new opportunities for services to automate parts of the documentation completely (listing functional features by looking up e.g. methods ...). Additional build systems can be used in contributions to also cover build tools as technologies relevant to software language engineering and therefore the 101companies chrestomathy.

The build process could also be integrated into the contribution process for real-time testing of newly contributed implementations. Research on automated build script generation [69] would be useful in this context.

A testing framework for contributions could be developed to further standardize and automate testing and to improve tests beyond the functional requirements.

Build support for web applications or projects with dependency on other programs (such as data bases) could be established [70]. System-dependant testing in the sense of operating several build servers for different operating systems could improve quality assurance for 101companies. Peculiarities and issues between operating systems would be more discoverable and functionality of build automation and therefore compatibility for different environments verified.

Ownership assessment [45] for build management especially in regard to the potential growth of 101companies needs to be discussed.

Deeper study of technological spaces and their differences and similarities can help improving the 101companies feature model to - by abstraction - further specify conventions for comprehensibility and accessibility for inexperienced users. This issue is a more general topic and goes far beyond the scope of build management. Nevertheless it can help to drastically increase comprehensibility and decrease naming discrepancy.

Bibliography

[1] 101companies project web page.
http://101companies.org/

[2] 101companies: Software languages used.
https://github.com/101companies/101repo/tree/master/languages

[3] Jean-Marie Favre, Ralf Lämmel, Thomas Schmorleiz and Andrei Varanovich. *101companies: A Community Project on Software Technologies and Software Languages*. In *TOOLS Europe 2012*, pages 58-74, Berlin Heidelberg, 2012, Springer.

[4] Wikipedia: Software build.
http://en.wikipedia.org/wiki/Software_build

[5] Wikipedia: Revision control.
http://en.wikipedia.org/wiki/Revision_control

[6] Wikipedia: List of build automation software.
http://en.wikipedia.org/wiki/List_of_build_automation_software

[7] Maven - Introduction to the Build Lifecycle.
http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

[8] Ant, Maven and Gradle – A side by Side Comparison.
http://xpanxionsoftware.wordpress.com/2012/05/11/ant-maven-and-gradle-a-side-by-side-comparison/

[9] Wikipedia: Convention over configuration.
http://en.wikipedia.org/wiki/Convention_over_configuration

[10] The Maven Central Repository.
http://search.maven.org/

[11] Wikipedia: Extreme Programming.
http://en.wikipedia.org/wiki/Extreme_programming

[12] Mário Luís Guimaraes, António Rito-Silva. *Towards Real-Time Integration*. 2010, ACM.

[13] Continuous Integration in Zeiten agiler Programmierung.
http://www.heise.de/developer/artikel/Continuous-Integration-in-Zeiten-agiler-Programmierung-1427092.html

[14] Wikipedia: Chrestomathy.
http://en.wikipedia.org/wiki/Chrestomathy

[15] Ivan Kurtev, Jean Bézivin and Mehmet Aksit.
Technological Spaces: an Initial Appraisal. In *CoopIS, DOA 2002 Federated Conferences*, 2002, Industrial track.

[16] 101companies simpleJava implementations on GitHub.
https://github.com/101companies/101simplejava

[17] 101companies project wiki page.
http://101companies.org/wiki

[18] Wikipedia: Build automation
http://en.wikipedia.org/wiki/Build_automation

[19] A glassfish dependency tree.
https://weblogs.java.net/blog/kohsuke/archive/2008/01/glassfish_v3_de.html

[20] 101companies discovery service.
http://101companies.org/resources?format=html

[21] Wikipedia: Software development process.
http://en.wikipedia.org/wiki/Software_development_process

[22] Wikipedia: Integrated development environment.
http://en.wikipedia.org/wiki/Integrated_development_environment

[23] The Eclipse IDE.
http://www.eclipse.org/

[24] The tool make.
https://en.wikipedia.org/wiki/Make_(software)

[25] Apache Ant.
http://ant.apache.org/

[26] Apache Maven.
http://maven.apache.org/

[27] The SBT tool.
http://www.scala-sbt.org/

[28] Wikipedia: Neutral build http://en.wikipedia.org/wiki/Neutral_build

[29] 101companies feature model.
http://101companies.org/wiki/Namespace:Feature

[30] Wikipedia: make.
http://en.wikipedia.org/wiki/Make_(software)

[31] Ant Introduction.
http://ant.apache.org/manual/intro.html

[32] Cygwin for Windows.
`http://www.cygwin.com/`

[33] 101companies contribution hadoop.
`http://101companies.org/wiki/Contribution:hadoop`

[34] 101companies data requirements.
`http://101companies.org/wiki/Data_requirement`

[35] 101companies functional requirements.
`http://101companies.org/wiki/Functional_requirement`

[36] Wikipedia: Plain Old Java Object.
`http://en.wikipedia.org/wiki/Plain_Old_Java_Object`

[37] 101companies worker.
`https://github.com/101companies/101worker`

[38] Wikipedia: Best Coding Practices.
`http://en.wikipedia.org/wiki/Best_Coding_Practices`

[39] Microsoft Developer Network: Coding Techniques and Programming Practices.
`http://msdn.microsoft.com/en-us/library/aa260844(v=vs.60).aspx#cfr_bestprac`

[40] Wikipedia: Separation of concerns.
`http://en.wikipedia.org/wiki/Separation_of_concerns`

[41] Oracle: Code Conventions for the Java Programming Language.
`http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html`

[42] Dag I. K. Sjøberg, Ray Welland, Malcolm P. Atkinson, Paul Philbrow and Cathy Waite. *Exploiting Persistence in Build Management*. 1997, ACM. In *Software-Practice and Experience Volume 27 Issue 4*, pages 447-480, April 1997, John Wiley and Sons, Inc., NY, USA.

[43] Glenn Ammons. *Grexmk: Speeding Up Scripted Builds*. 2006, ACM.

[44] Shane McIntosh, Bram Adams, Thanh H. D. Nguyen, Yasutaka Kamei and Ahmed E. Hassan. *An Empirical Study of Build Maintenance Effort*. 2011, ACM.

[45] Shane McIntosh. *Build System Maintenance*. 2011, ACM.

[46] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen and Tien N. Nguyen. *Build Code Analysis with Symbolic Evaluation*. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 650-660, 2012, IEEE Press Piscataway.

[47] Gradle.
`http://www.gradle.org/`

[48] The Gradle Wrapper.
`http://www.gradle.org/docs/current/userguide/gradle_wrapper.html`

[49] Paul M. Duvall.
Continuous Integration: Improving Software Quality and Reducing Risk, ISBN-13 978-0321336385, 2007, Addison-Wesley Professional.

[50] Christoph Elsner, Daniel Lohmann and Wolfgang Schröder-Preikschat. *An Infrastructure for Composing Build Systems of Software Product Lines*. 2011, ACM.

[51] Stefan Dösinger, Richard Mordinyi and Stefan Biffl. *Communicating Continuous Integration Servers for Increasing Effectiveness of Automated Testing*. 2012, ACM.

[52] Even-André Karlsson, Lars-Göran Andersson and Per Leion. *Daily build and feature development in large distributed projects*. 2000, ACM.

[53] CruiseControl.
http://cruisecontrol.sourceforge.net/

[54] ThoughtWorks: CI Feature Matrix.
http://confluence.public.thoughtworks.org/display/CC/CI+Feature+Matrix

[55] Hudson: Extensible continuous integration server.
http://hudson-ci.org/

[56] Eclipse Foundation: Hudson.
http://www.eclipse.org/hudson/

[57] Wikipedia: Hudson.
http://en.wikipedia.org/wiki/Hudson_(software)

[58] Hudson: An extendable open source continuous integration server.
http://jenkins-ci.org/

[59] Travis continuous integration service.
http://about.travis-ci.org/

[60] Travis continuous integration service environment.
http://about.travis-ci.org/docs/user/ci-environment/

[61] Javacamp: About Design Pattern.
http://www.javacamp.org/designPattern/

[62] Jenkins build tool plugins.
https://wiki.jenkins-ci.org/display/JENKINS/Plugins#Plugins-Buildtools

[63] Jenkins revision control tool plugins.
https://wiki.jenkins-ci.org/display/JENKINS/Plugins#Plugins-Sourcecodemanagement

[64] Complete list of Jenkins plugins.
https://wiki.jenkins-ci.org/display/JENKINS/Plugins

[65] Complete list of Jenkins plugins.
https://wiki.jenkins-ci.org/display/JENKINS/Installing+Jenkins+on+Ubuntu

[66] Continuous Integration Best Practices with Rational Team Concert.
https://jazz.net/library/article/474

[67] Wikipedia: Continuous Integration.
http://en.wikipedia.org/wiki/Continuous_integration#Principles

[68] Haskell and Continuous Integration.
`http://www.yesodweb.com/blog/2012/10/haskell-and-ci`

[69] Charng-da Lu, Matthew D. Jones and Thomas R. Furlani.
Automatically Mining Program Build Information via Signature Matching. 2011, ACM.

[70] Eelco Dolstra, Martin Bravenboer and Eelco Visser.
Service Configuration Management. 2005, ACM.