GDAŃSK UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRONICS, TELECOMMUNICATIONS AND INFORMATICS

Student’s name and surname: Łukasz Myśliński
ID: 143297
Postgraduate studies
Mode of study: Full-time studies
Field of study: Informatics
Specialization/profile: -

MASTER'S THESIS

Title of thesis: Context-aware build automation system for Scala

Title of thesis (in Polish): Kontekstowo zorientowany system automatyzacji budowy projektów dla języka Scala

Supervisor signature	Head of Department signature
dr inż. Waldemar Korłub	

Date of thesis submission to faculty office:

OŚWIADCZENIE

Imię i nazwisko: Łukasz Myśliński
Data i miejsce urodzenia: 28.12.1993, Gdańsk
Nr albumu: 143297
Wydział: Wydział Elektroniki, Telekomunikacji i Informatyki
Kierunek: informatyka
Poziom studiów: drugi
Forma studiów: stacjonarne

Ja, niżej podpisany(a), wyrażam zgodę/nie wyrażam zgody* na korzystanie z mojej pracy dyplomowej zatytułowanej: Kontekstowo zorientowany system automatyzacji budowy projektów dla języka Scala
do celów naukowych lub dydaktycznych.[1]

Gdańsk, dnia ....
podpis studenta

Świadomy(a) odpowiedzialności karnej z tytułu naruszenia przepisów ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz. U. z 2016 r., poz. 666 z późn. zm.) i konsekwencji dyscyplinarnych określonych w ustawie Prawo o szkolnictwie wyższym (Dz. U. z 2012 r., poz. 572 z późn. zm.),[2] a także odpowiedzialności cywilno-prawnej oświadczam, że przedkładana praca dyplomowa została opracowana przeze mnie samodzielnie.

Niniejsza(y) praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadaniem tytułu zawodowego.

Wszystkie informacje umieszczone w ww. pracy dyplomowej, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami zgodnie z art. 34 ustawy o prawie autorskim i prawach pokrewnych.

Potwierdzam zgodność niniejszej wersji pracy dyplomowej z załączoną wersją elektroniczną.

Gdańsk, dnia ....
podpis studenta

Upoważniam Politechnikę Gdańską do umieszczenia ww. pracy dyplomowej w wersji elektronicznej w otwartym, cyfrowym repozytorium instytucjonalnym Politechniki Gdańskiej oraz poddawania jej procesom weryfikacji i ochrony przed przywłaszczaniem jej autorstwa.

Gdańsk, dnia ....
podpis studenta

*) niepotrzebne skreślić

[1] Zarządzenie Rektora Politechniki Gdańskiej nr 34/2009 z 9 listopada 2009 r., załącznik nr 8 do instrukcji archiwalnej PG.
[2] Ustawa z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym:
Art. 214 ustęp 4. W razie podejrzenia popełnienia przez studenta czynu podlegającego na przypisaniu sobie autorstwa istotnego fragmentu lub innych elementów cudzego utworu rektor niezwłocznie poleca przeprowadzenie postępowania wyjaśniającego.
Art. 214 ustęp 6. Jeżeli w wyniku postępowania wyjaśniającego zebrany materiał potwierdza popełnienie czynu, o którym mowa w ust. 4, rektor wstrzymuje postępowanie o nadanie tytułu zawodowego do czasu wydania orzeczenia przez komisję dyscyplinarną oraz składa zawiadomienie o popełnieniu przestępstwa.

Abstract

Build tools are an integral part of software development. In this thesis the author examines what is the function of a build tool, lays down the theory for creating a build tool for the Scala language, and successfully creates one called fsbt. Concepts such as incremental compilation, dependency management, custom DSL creation and context discovery are discussed in depth. The created build tool is examined and compared with established ones like Ant, Gradle, Maven or sbt. Experiments are carried out to demonstrate the advantage in speed and simplicity over competitors. A real world use case is also included. In the summary the author focuses on examining the results of the experiments and challenges for successful build tool popularization.

Keywords: build tool, compilation, scala, java, sbt, maven, gradle, build process, development, continuous integration, devops

Streszczenie

Narzędzia do budowy projektów są nierozłączną częścią wytwarzania oprogramowania. W tej pracy Autor bada jaka jest ich rola, czym powinno charakteryzować się narzędzie przeznaczone dla języka Scala i z powodzeniem opracowuje własne narzędzie pod nazwą fsbt. Rozdział pierwszy pokrótce przedstawia zagadnienia omawiane w pracy.

W rodziale drugim przedstawiony zostaje język Scala, a w szczególności charaterystyka tej platformy i dostępne kompilatory. Omawiany jest proces przyrostowej kompilacji. Wprowadzane jest pojęcie języków domenowych, z rozróżnieniem języków wewnętrzych i zewnętrznych. Kolejno przedstawiane są istniejące narzędzia do budowy projektów w języku Scala, takie jak Ant, Maven, Gradle oraz sbt. Poruszany jest także temat zarządzania zależnościami.

Rozdział trzeci zawiera szczegółowy opis opracowanego narzędzia fsbt. Przedstawiane są przyjęte założenia oraz architektura programu. Wprowadzony zostaje własny język domenowy do konfiguracji projektów. Autor omawia metody użyte do rozwiązania problemów implementacyjnych, takich jak wykrywanie konteksu aplikacji, zarządzanie zależnościami, wsparcie dla wielomodułowych projektów czy rozwiązywanie zależności cyklicznych. Prezentowane są scenariusze działania w przypadku zrównoleglenia procesu budowy aplikacji.

W rozdziale czwartym przestawione są metryki wybrane przez Autora do porównania stworzenego narzędzia fsbt z innymi, istniejącymi rozwiązaniami, w szczególności metryki złożoności Halsteada. Na podstawie wyników analizy z użyciem danych metryk Autor przedstawia wnioski na temat wygody i łatwości użytkowania porównywanych narzędzi. Opisywane są wyniki analizy porównawczej dla takich kryteriów jak prędkość zwykłej i przyrostowej kompilacji, a także kompilacji równoległej wraz z uruchomieniem testów dla wielu modułów. Wyniki eksperymentów są komentowane i tłumaczone. Przedstawiony jest także przykład użycia przygotowanego narzędzia fsbt w rzeczywistej aplikacji.

W rozdziale piątym Autor analizuje wyniki eksperymentów oraz przyszłe wyzwania na drodze do spopularyzowania opracowanego narzędzia. Po podsumowaniu rezultatów oraz omówieniu planów dalszego rozwoju przedstawiono spis rysunków, spis tabel oraz spis literatury obejmujący 36 pozycji, które zostały zacytowane w pracy.

Table of Contents

List of Important Symbols and Abbreviations

ASM – Assembly Language

AST – Abstract Syntax Tree

CLI – Command Line Interface

IDE – Integrated Development Environment

DAG – Directed Acyclic Graph

DSL – Domain Specific Language

FP – Functional Programming

JDK – Java Development Kit

JRE – Java Runtime Environment

OOP – Object Oriented Programming

REPL – Read Eval Print Loop

VM – Virtual Machine

Chapter 1

Introduction, Objectives and Aims

Managing modern software development has become a domain in itself. Most projects have dependencies and configuration that must be managed. Automated testing has become a vital component of a software product lifecycle. Integration with IDE's and external services such as code review platforms or code analysis tools is often desirable, if not necessary. Cloud environments are becoming increasingly common, further increasing the complexity of development[1]. In case of professional software development compilation is merely one of many important steps to delivering a product. That is why we rarely speak of compilation or building in terms of producing the final software product – rather about the *build process*.

A build process should be fast, reliable and highly configurable while being easy to maintain and extend [2] [3] [4] [5]. That is no easy feat to achieve [6], which is why existing build systems are continuing to evolve and new ones are being developed [7] [8] [9] [10]. Just as the Java and its ecosystem have come a long way from its initial release, build tools used to develop on the JVM have evolved as well [11].

In this work the author focuses on build systems for one of the languages present on the JVM platform, namely Scala. Being a relatively young language, the Scala ecosystem has only one Scala-first key player in the build tool ecosystem – sbt, the Scala Build Tool. However, being a member of the JVM platform, existing Java tools such as Maven and Gradle are completely interoperable with Scala and actively compete with (and in many cases outperform) SBT. The author believes that there is still room for growth in the build tool world, thus he aims to create his own build tool – fsbt, fast scala build tool.

In chapter 2 the necessary theory for this paper will be established. Scala itself will be introduced. Important concepts like dependency management, incremental compilation, JVM internal mechanisms and others will be described in detail. All of this will be used to grasp the complexity of defining a project build lifecycle. A brief description of existing build tools is presented.

Chapter 3 introduces fsbt, the build tool created by the author of this paper. Assumptions and goals are presented. Problems such as identification of the execution context, modularity, execution schemes and creating a custom DSL are discussed. A solution to these problems is laid out, along with examples of usage.

Chapter 4 focuses on experiments carried out on a range of build tools, including fsbt. The author describes problems with comparing build tools and proposes a feasible approach. Halstead metrics and benchmark methodology is presented. Experiments involving testing multi-module support and incremental compilation speed are carried out and summarized. A real world use case is presented.

Chapter 5 discusses the results of this paper. The success of the project is explained and reasoned about. Build tools not mentioned in this paper are acknowledged. Plans for future work are presented, along with requirements for the project to succeed in popularity.

Chapter 2

Problem Analysis

2.1 The Scala Programming Language

Scala is a general purpose programming language [12]. The design was first established in 2001 by Martin Odersky and the first official release was published in 2004. Scala was designed with functional programming in mind from the beginning, despite the fact that Java remained a purely object-oriented language until the release of Java 8 in 2014.

Scala is a fully object-oriented language as well. Conceptually, every value is an object and every operation is a method-call. The language supports advanced component architectures through classes and traits. Many popular OOP concepts, such as the Singleton pattern are built-in with the use of object definitions [13].

Previously mentioned functional programming plays a key role in Scala [14]. Many of its concepts originate from Haskell, the go-to purely functional programming language. In Scala, functions are first-class citizens, which means that functions are treated as objects. Specifically, this means that the language supports constructing new functions during the execution of a program, storing them in data structures, passing them as arguments to other functions, and returning them as the values of other functions. Implicity is also one of core Scala features – classes, methods and parameters can be implicit which means they can be used in a non-explicit way – automtic conversion of one type to another can occur, parameters can be passed without passing them in the function call [15].

2.1.1 Scala Compilation

These high-level concepts implemented in Scala unfortunately greatly increase the difficulty of the build process. Extensive type system and multiple abstraction mechanism result in long compilation times. The basic compiler in the latest Scala version (2.12.2) includes 24 distinct phases:

Listing 2.1: Scala compiler phases

```
~ > scalac -Xshow-phases
    phase name  id  description
    ----------  --  -----------
        parser   1  parse source into ASTs, perform simple desugaring
         namer   2  resolve names, attach symbols to named trees
packageobjects   3  load package objects
         typer   4  the meat and potatoes: type the trees
        patmat   5  translate match expressions
superaccessors   6  add super accessors in traits and nested classes
    extmethods   7  add extension methods for inline classes
      pickler    8  serialize symbol tables
     refchecks   9  reference/override checking, translate nested
                     objects
       uncurry  10  uncurry, translate function values to anonymous
                     classes
        fields  11  synthesize accessors and fields, add bitmaps for
                     lazy vals
     tailcalls  12  replace tail calls by jumps
    specialize  13  @specialized-driven class and method specialization
 explicitouter  14  this refs to outer pointers
       erasure  15  erase types, add interfaces for traits
   posterasure  16  clean up erased inline classes
    lambdalift  17  move nested functions to top level
  constructors  18  move field definitions into constructors
       flatten  19  eliminate inner classes
         mixin  20  mixin composition
       cleanup  21  platform-specific cleanups, generate reflective
                     calls
    delambdafy  22  remove lambdas
           jvm  23  generate JVM bytecode
      terminal  24  the last phase during a compilation run
```

Most of these phases are responsible for optimization and unfolding the abstractions of the language. The most notable impact on the compilation time is introduced by the 'typer' phase. It can amount to as much as half of the entire compilation time. Long compilation is a significant problem during development, thus considerable effort has been put into the improvement of this process.

There are multiple Scala compilers available:

Scalac is the default Scala compiler, included with the distribution. It has been around since the inception of Scala and does only what is described in the phases in listing 2.1.

Fast Scala Compiler or fsc is the first advancement over the original compiler. Instead of starting up the JVM each time to compile the source files, this compiler uses a daemon approach. By running a daemon in the background, it is working in a client-server fashion to allow faster subsequent compilation of the same project. Currently it is hardly used, as more efficient solutions were created.

Zinc originally started as a part of SBT. The daemon approach taken by fsc was a welcome improvement, but it was still missing the key component to vastly speeding up the compilation: incremental compilation. Incremental compilation means recompiling only these source files, that were affected by the changes made from previous build. This process will be described in detail in chapter 2.1.2. Lightbend, one of key companies behind Scala, during the development of SBT decided to implement their own compiler, that would be focused on performing this task. This concept was originally included in Scalac, however it was only detecting changes made to source files without regard of their impact on other files. File-base approach was fundamentally broken, so there was a need to develop a compiler capable of building a graph of dependecies between source files, in order to properly detect which files should be recompiled. Zinc was originally an internal part of SBT, however given the popularity of Maven as a build tool for Scala it was later extracted and elevated as a standalone project. Currently Zinc is the go-to compiler for Scala, yielding vast time advantage over Scalac and FSC.

Dotty – Martin Odersky, the founder of Scala began working on Dotty in late 2013. In his most recent talk at Voxxon Days [16] in 2016, Martin has presented Dotty to the general public. It is a completely new compiler for Scala, written from scratch and based on Dot, a graph description language [17]. It is still under active development, however it has reached a point of maturity allowing it to compile itself. It incorporates features of Zinc such as incremental compilation, and adds an extensive list of features and optimizations. While Dotty is not yet finished, it is widely considered to be the future of Scala.

2.1.2 Incremental compilation

As described in [18], incremental compilation for Scala is a difficult process. Its goal is to recompile as little source files as possible, while making sure that all bytecode dependencies have been updated. These conditions work against each other – effectively establishing the files that need to be recompiled. There are several approaches that can tackle this problem. The most simple one is simply recompiling all the files that have been changed. Without showing any examples, one can tell that this approach is not going to work for any non-trivial program. Source files can import definitions and functions from other source files, thus creating a dependency from one to another. In order to solve this, the incremental compiler needs to perform two tasks:

- index source files, to detect changes made to method signatures and other components that may affect other files,
- keep a graph of dependencies between source files. If a change in the API has been detected, the dependent source files need to be marked for recompilation.

In case of Zinc, both of this tasks are performed by extracting information from the Scala compiler. This is performed by adding three additional phases to the compiler:

- **API extraction phase** – this phase focuses on extracting information from the Scala compiler, transforming it and serializing it for future reference on the disk. It consist of two operations:
  - **mapping Types and Symbols**. In order to simplify the analysis process, the entire signature of a class or trait with their parent signatures,
  - **hashing an API representation**. Because the only information required to determine a diffrence in the API is the signature, a hash of the signature is a sufficient representation for analysis purposes. This approach greatly reduces the memory usage of the incremental compiler,
- **dependency phase** In this phase all symbols are extracted from the source files. The symbols are then mapped to their corresponding source files. Mapping the symbols to the compiled source files is more complicated, because no such information is present in the Scala compiler. A simple heuristic is used to map compiled classes by their qualified class name to their classpath entries. Finally, a tree walk is responsible for marking files which can inflict a dependency on other files,
- **analyzer phase** This phase uses information about the location of compiled source files retrieved from the Scala compiler to validate timestamps and locate missing class files. In both cases, a recompilation is required.

There are two patterns that hashing approach must withstand: the **enrichment pattern** and the **inheritence pattern**. Enrichment pattern assumes that if any member of a class is added/-modified/removed then compilation results of other classes won't be affected unless they have a reference to that particular member. An example of the enrichement pattern is an addition of a new parameter to a method. Such a change affects all usages of the method, therefore the hash comparison will always result in marking all the necessary source files.

Inheritance which applies concepts like overriding and mix-in composition that introduces new fields to classes inheriting from traits can not be solved in this fashion. Instead, the dependencies which are introduced by inheritance are not subject to the name-hashing analysis and are tracked separatly. Scala features don't make the situation any easier:

- named parameters require inclusion of parameter names in the interface definition of a method/class,

- adding a method to a trait requires recompiling all of descendant classes. This is no diffrent from Java with its interfaces. However, Scala traits can contain implementation, which makes calls such as super.method() a special case,
- pattern matching must check whether all options of possible matches have been exhausted. It is one of the reasons why dependency tracking must depend on the entire class hierarchy.

Because the problems outlined above have not yet been resolved completly, a solution is not presented here. Decreasing the complexity of the language and sacrificing configurability in favour of simplicity would go a long way, but as long as features are being added to the language we can expect incremental compilation to become ever harder to solve.

2.2 Domain Specific Languages

2.2.1 Oveview

It is safe to assume that the goal of software development is solving problems within a certain area or a domain. General-purpose Programming Languages (GPPL) require complex structures and syntax to model and manipulate a given problem. This can often prove highly inneffective and error-prone, as many studies have found [19] [20]. A domain specific language at its core is a computer language which is designed towards a particular kind of problem, rather than a general purpose language that's aimed at any kind of software problem[21]. Domain specific languages have been talked about, and used for almost as long as computing has been done. Examples of DSL include all query languages (like SQL), all template languages, shell scripts as well as data storage and exchange languages (XML, YAML, JSON), and document languages like LaTex, HTML or CSS.

2.2.2 Internal and external DSL

As distinguished in [22], there are two types of DSLs: Internal DSLs and external DSLs. An internal DSL is built upon the host language and is expanding on the syntax of the language to create a new way of information defining. All parts of the new DSL must be understandable to the compiler (or interpreter) of the host language. To give an example, listing 2.2 presents a Lunar Lander game written in a BASIC-like DSL in Scala [23].

Listing 2.2: Lunar Lander game in a BASIC-like DSL

```
object Lunar extends Baysick {
  def main(args:Array[String]) = {
    10 PRINT "Welcome to Baysick Lunar Lander v0.9"
    20 LET ('dist := 100)
    30 LET ('v := 1)
    40 LET ('fuel := 1000)
    50 LET ('mass := 1000)

    60 PRINT "You are drifting towards the moon."

    70 PRINT "You must decide how much fuel to burn."
    80 PRINT "To accelerate enter a positive number"
    90 PRINT "To decelerate a negative"

    100 PRINT "Distance " % 'dist % "km, " % "Velocity " % 'v % "km/s,
        " % "Fuel " % 'fuel
    110 INPUT 'burn
    120 IF ABS('burn) <= 'fuel THEN 150
    130 PRINT "You don't have that much fuel"

    140 GOTO 100
    150 LET ('v := 'v + 'burn * 10 / ('fuel + 'mass))
    160 LET ('fuel := 'fuel - ABS('burn))
    170 LET ('dist := 'dist - 'v)
    180 IF 'dist > 0 THEN 100
    190 PRINT "You have hit the surface"
    200 IF 'v < 3 THEN 240
    210 PRINT "Hit surface too fast (" % 'v % ")km/s"
    220 PRINT "You Crashed!"

    230 GOTO 250
    240 PRINT "Well done"

    250 END

    RUN
  }
}
```

Scala is a DSL-friendly language, which means it has a large amount of flexibility in its syntax. This particular DSL has been built to imitate a BASIC syntax, however it is still valid Scala code. All literals presented inside the main function are interpreted into corresponding Scala classes and objects, allowing for composing them in a syntax not originally supported by Scala. Internal DSLs are very useful for writing code that is domain-focused and easily understandable. Without them one would often have to resort to building complex object and method calls, that are often way more demanding in terms of difficulty.

An external DSL does not have to comply to the rules of the host language. They are parsed independently, and allow unlimited flexibility. This feature comes at the cost of having to parse the syntax inside the program – instead of letting the compiler take care of it for the developer. It involves lexical analysis which is a process of matching sequences of characters into tokens:

- a *token* is a pair consisting of an abstract symbol representing a lexical unit,
- a *pattern* is a description of the form that different lexemes take,
- a *lexeme* is a sequence of characters in a source program matching a pattern.

A unit performing lexical analysis is called a lexer. Each programming language must include a lexer in the compilation process to validate the language vocabulary i.e. the C language has Lex and Flex, while Java uses JFlex or JavaCC.

The next step is parsing. A parser's job is to verify the grammar of a language. While lexers verify individual language units, parsers group them into multi-unit language constructs such as definitions, assignments, calls etc. Parsing is a field in itself, with many different approaches like backtracking or predictive parsing. The main distinction between parsers is based on how the input can be derived from the start symbol of the grammar.

In a top-down approach, which is attempting to find left-most derivations of an input-stream by trying to match grammar expressions recursively, tokens are consumed left to right. On the other hand, in a bottom-up approach, a parser starts with the input and tries to rewrite it to the start symbol. It works by locating the most simple elements of the grammar, and grouping them into more complex ones. Examples of such parsers are LR parsers.

Parsing along with lexing is an essential part of programming language design. The same principles which apply to general programming languages are applied when constructing DSL's. In order to create a DSL, one must define a vocabulary, a grammar, as well as write a lexer and a parser for those. Fortunately, Scala has a library specifically for that purpose. Scala Parser Combinators supplies the programmer with a set of basic tokens such as a *number* or *identifier* that can be used to build a grammar with easy. Another features is automatic mapping of discovered grammatic constructs. This matter will be further discussed in chapter 3.2.2.

2.3 Comparison of existing build tools

Judging by adoption rates and impact on the industry, there are only a few important build tools, namely Ant (with Ivy), Maven, Gradle and SBT. Of course there are other interesting, innovative ones such as Pants, Bazel or CBT, however none of those can be considered widely adopted or mature. For this reason, only the established ones will be referenced and compared throughout this paper.

2.3.1 Ant

Apache Ant is one of the most established tools in the JVM world. It is a software tool for automating software build processes, which originated from the Apache Tomcat project in early 2000 [24]. It was designed as a JVM-specific Makefile alternative.

To build a project with Ant a project organization is required, as well as the Ant buildfile. The buildfile is the most important aspect of Ant. It is a document written in XML, which is not accidental. Usually software projects are described in terms of dependencies – Java projects are best described as a combination of packages and components, which follow the package and object model of the language. At the time Ant was being developed, no other build tool took such an approach. Also, the buildfile had to fulfill the following conditions:

- simple, easy to learn syntax – a DSL that would be easy to learn and familiar to Java developers,
- an existing library for parsing the language,
- the ability to describe the build process in terms of packages and operations.

XML was the only language fulfilling these conditions. It was widely used in many areas of Java development, such as Enterprise Java Beans(EJB), Java Server Pages(JSP's) as well as commonly used for configuration files. XML was the best choice for Ant.

The root element of the buildfile is always the <project> tag, containing the rest of our configuration. Inside a project properties are defined, which describe the directory structure of our project, as well as targets, which contains a series of operations to be performed on our project. Multiple attributes are available for each tag. Listing 2.3 contains a simple Ant build.xml example.

Listing 2.3: A sample build.xml file

```
<project name="MyProject" default="dist" basedir=".">
  <description>
    simple example build file
  </description>
  <!-- set global properties for this build -->
  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist" location="dist"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init"
        description="compile␣the␣source">
    <!-- Compile the java code from ${src} into ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile"
        description="generate␣the␣distribution">
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}/lib"/>

    <!-- Put everything in ${build} into the MyProject-${DSTAMP}.jar
        file -->
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build
        }"/>
  </target>

  <target name="clean"
        description="clean␣up">
    <!-- Delete the ${build} and ${dist} directory trees -->
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>
</project>
```

Listing 2.3 contains a description as well as a set of properties and targets. Properties allow us to define variables inside the build.xml. A target is a set of tasks that should be performed by Ant. Each target should specify which operations should be performed on what data and when. For example, the compile target is dependent on the init target, which means it cannot be run before the init target is completed. Inside the compile target one can notice a single javac operation, which compiles code contained in the ${src} directory, and outputs the results in to the ${build} directory.

Running this Ant file is as simple as typing ant in the console. Ant runtime automatically resolves the build.xml file and tries executing the main target. Given that Ant is written in Java, it is a multi-platform tool. Running it on Windows should give the same results as using a UNIX shell.

This particular build.xml is good enough for a small project – it defines a minimal configuration for creating a fast, efficient workflow. A simple command is enough to create a runnable jar, and a complete distribution package with another one.

When Ant was introduced, it a was an obvious improvement over Make – the syntax was easy to read and understand, the tool was designed specifically for Java and allowed for much less troublesome development. But unfortunately, it shares a plenty of Make's downsides. While the sample build.xml might seem simple – it is a lot of configuration just to make a simple project build without having to use the compiler manually. The example above works perfectly fine for small projects, but as projects get bigger, managing build scripts becomes almost as problematic as in case of Makefiles.

In a medium-sized project, consisting of several teams of developers working on the same product, many things can go the wrong way. If each team is responsible for one or more modules, exclusive to their team, they will create tasks and apply styling they see fit for each particular module. If each team defines its own rules and tasks for building code, assembling the final product as well as changing anything build-related in a module not belonging to your team becomes a very tedious task.

Although Ant is a very precise and powerful tool, which allows to carefully tailor solutions adequate to the given software project, it places significant overhead on the developers. There is no standardized directory structure, compilation or deployment strategy shipped with Ant. Maintaining a build process containing dozens of modules means a requirement to manage each module lifecycle individually. However, this must not always be the case. The flexibility of Ant makes it a great tool for experienced developers, as it allows for complete control of the project – each part of the project's workflow can be defined.

There is one aspect of Ant that wasn't discussed yet – namely dependency management. The reason for that is simple – Ant has none. Rarely a project exists on its own – code reuse is one of the core aspects of software development.

There are a few strategies to be adopted to manage the dependencies. A version control system could be used to include pre-built libraries into our repository – but the size of dependencies can become a problem very quickly. A project library directory could also by used. This solution however becomes problematic once multiple projects depend on the same libraries, forcing the developer to keep these files in multiple locations. Other problems include nested dependencies – libraries depending upon different versions of other libraries. In the end it is possible to end up attaching the same libraries multiple times, only in different versions.

As a solution to these problems, Apache Ivy was created. It is a tool for managing project dependencies. It is characterized by flexibility and configurability – it is not tied to any methodology or structure, and can be integrated into any build process. It's closely integrated with Apache Ant, although it is a standalone tool as well.

It features a mechanism called transitive dependency management – it is responsible for managing the dependencies of project's direct dependencies. By only having to specify the top-level library to be used, the dependency tree will be managed by Ivy. That reduces the end user complexity by a significant margin. It also features a conflict resolution engine, which can be configured according to preference. Several strategies can be set, from always selecting all revisions to throwing exceptions upon any conflicts.

Ant and Ivy work best when used together. They were designed to complement each other, and from the technical point of view they contain all of the core functionality to be considered a modern build tool. Yet, they leave much to be desired in terms of usability. The configuration overhead and lack of standardization has lead to the creation of far more advanced tools, such as Maven and Gradle.

2.3.2 Maven

Apache Maven has been built upon the concepts established by Ant and Ivy [25]. Maven is a project management tool which encompasses a project object model, a set of standards, a project lifecycle, a dependency management system, and logic for executing plugin goals at defined phases in a lifecycle [26]. Maven usage comes down to defining the project in a POM – project object model.

At its core, Maven uses a concept called convention over configuration. All projects built with Maven should assume reasonable defaults. Without much configuration, systems should ”just work”. For example, source code is assumed to be in ${basedir}/src/main/java and resources are assumed to be in ${basedir}/src/main/resources directories. This goes beyond just simple directory locations. It’s core plugins apply a common set of conventions for compiling source code, packaging distributions, generating documentation, and many other processes. Being opinionated, having a defined life-cycle and a set of plugins that know how to build and assemble software are just a few of its key strengths. In contrast to Apache Ant, where everything needs to be configured manually, this makes a big difference in terms of effort required for managing a project. If one follows these conventions, using Maven requires almost zero effort. By simply putting our source code where necessary and defining the POM, Maven will take care of the rest.

In times before Maven, developers had to split their attention between working on the actual code, and learning about the caveats of the build configuration established by the team. The introduction of a common interface for approaching different lifecycle aspects has made developer’s work easier.

Maven at its core isn’t all that powerful – it is merely capable of parsing the POM file. That is because most of its key functionality has been pushed outside of the core package – and put into plugins. The plugin system is one of the key aspects of Maven. The plugins operate in the lifecycle goals like compiling source code, packaging bytecode, running tests, and any other task which need to happen in a build. Plugins are available for the entire range of lifecycle operations, and are maintained centrally and shared universally.

The Project Object Model, or POM as it called, is the one and only information source about the project, defined in XML. It is used to define dependencies, submodules, plugins and properties. It also contains details such as what is the project’s license and who is contributing to it, among other useful information. Each project must be identified by a set of at least three tags:

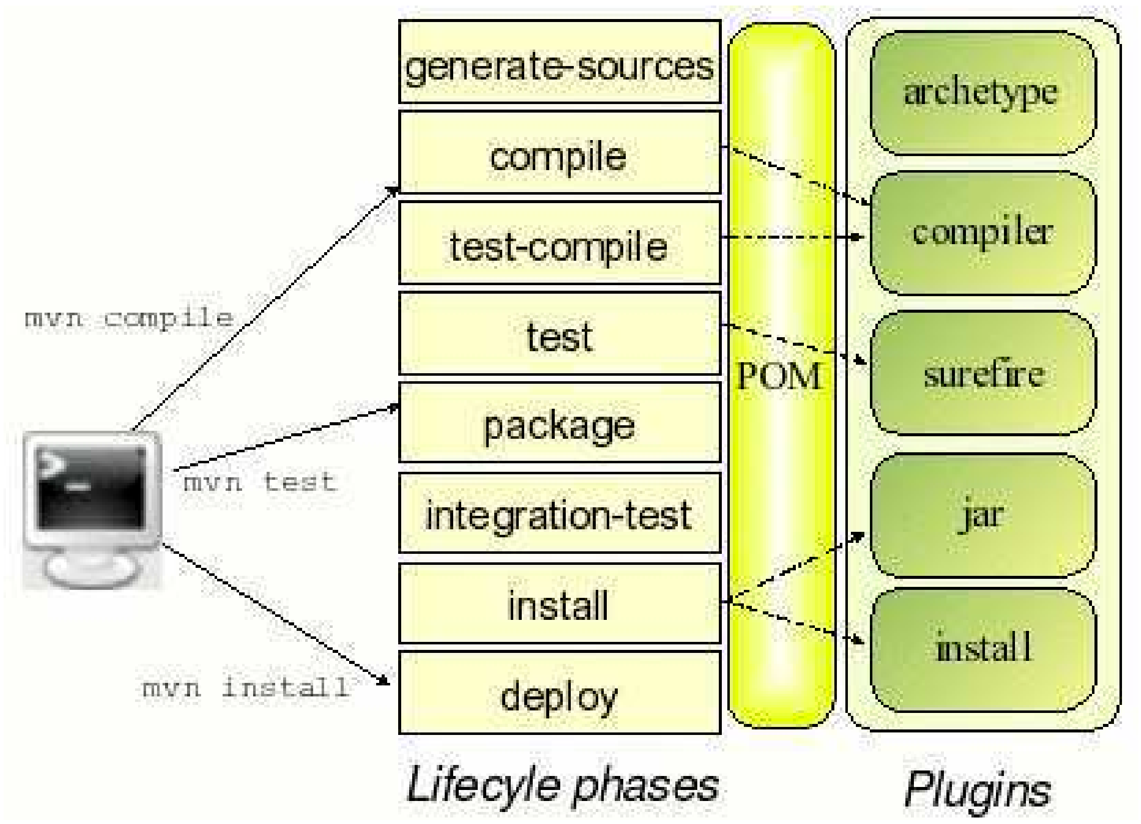


Figure 2.1: Maven task diagram [27]

- *groupId* – a high-level domain descriptor grouping similar projects, ex. *org.hiberanate*,
- *artifactId* – a unique identifier among a shared domain, ex. *hibernate-annotations*,
- *version* – Maven allows following versioning strategies:
  - A release version – it is ”tagged” and should be considered a stable release, that is should not be changed once published. For example: *1.0.0.Final*,
  - A Snapshot version – Snapshots are artifacts still in the process of active development. For example, *1.0.0-SNAPSHOT* is a working copy of a project, expected to be released with the *1.0.0* version.

Scope tag on the other hand allows to define the domain of the dependencies. It is common for most programming environments, that different sets of libraries are required for building, testing and running the application. The local development runtime environment might also be different from the production runtime, which is why Maven allows to set following scopes for each dependency:

- *compile* – This is the default scope, which means it will be used in case no other scope was specified. Dependencies marked as compile are available in all classpaths of a project. Furthermore, those dependencies are propagated to dependent projects.

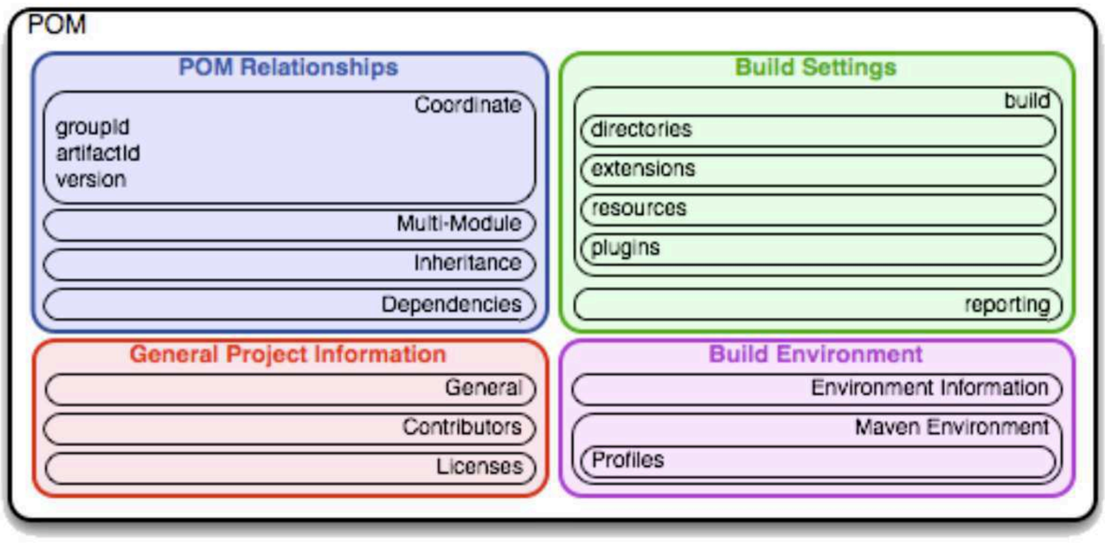


Figure 2.2: POM structure [28]

- *provided* – This is much like compile, but indicates that the JRE or an execution container should provide the dependency at runtime. For example, when building a web application for the Java Enterprise Edition, you would set the dependency on the Servlet API and related Java EE APIs to scope provided because the web container provides those classes. This scope is only available on the compilation and test classpath, and is not transitive.
- *runtime* – This scope indicates that the dependency is not required for compilation, but is for execution. It is in the runtime and test classpaths, but not the compile classpath.
- *test* – This scope indicates that the dependency is not required for normal use of the application, and is only available for the test compilation and execution phases. This scope is not transitive.
- *system* – This scope is similar to provided except that you have to provide the JAR which contains it explicitly. The artifact is always available and is not looked up in a repository.
- *import* – (only available in Maven 2.0.9 or later) This scope is only supported on a dependency of type pom in the <dependencyManagement> section. It indicates the dependency to be replaced with the effective list of dependencies in the specified POM's <dependencyManagement> section. Since they are replaced, dependencies with a scope of import do not actually participate in limiting the transitivity of a dependency.

Dependency propagation is an important aspect of building a multi-module project. Instead of manually copying the dependency definitions between different components, a parent POM can be used. The parent POM contains all necessary definitions, which will be automatically included in all POM's that reference it as its parent.

Other tags such as *classifier* or *type* allow for more complex project-related tasks, such as configuring the output type of the build or packing type for the build product.

2.3.3 Gradle

While Maven did substantially improve over its predecessors, there was still room for improvement. XML turned out to be less than perfect for writing long, complex build configurations. Maven also has faced issues with resolving conflicts between versions of the same library.

In the year 2012, the landscape was altered by a new competitor – Gradle. Gradle has learned from both Ant and Maven, and has incorporated the best parts from each of these tools in its design principles [29].

Declarative build and build-by-convention – Gradle uses a custom DSL built upon the Groovy language. Build-by-convention means reasonable defaults are assumed for certain tasks, i.e. java source files must be in *src/main/java*, tests must be in *src/main/test*, resources in *src/-main/resources*, supplied dependencies in *build/libs* etc. However, Gradle does not oblige you to use these conventions and you can change them if you want.

Multi-project builds – Gradle supports multi-project builds and also partial builds. If you build a subproject, Gradle takes care of building all the subprojects that it depends on.

Dependency Management – There are multiple dependency resolution mechanisms available

Ant and Maven compatibility – Gradle completely supports Ant tasks, Maven and Ivy repository infrastructure for publishing and retrieving dependencies. It also provides a converter for turning a Maven pom.xml to Gradle script.

Gradle wrapper – Gradle can be run inside a Gradle Wrapper script, which is native to a Unix or Windows environment. This comes very useful in continuous integration servers.

Complex API – Gradle has a rich and well established API, allowing for fine-grained tuning and monitoring of the build process.

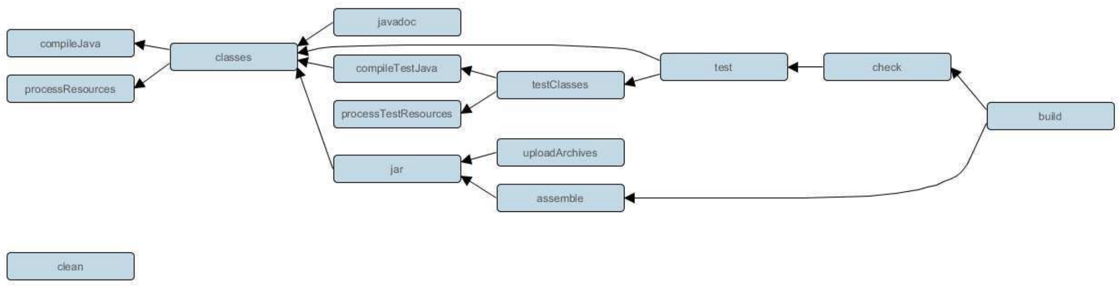


Figure 2.3: Gradle task graph [30]

A custom DSL is one of the major features offered by Gradle. Opening the API by using a programming language to customize a build was a new approach in the Java world. The additional flexiblity has led Google to choosing Gradle as the primary build tool for the Android platform in 2014. In spite of its popularity amongst developers, the battle between Gradle and Maven for the crown in Java build tool world is still in favor of Maven.

Java build tools comparison

While the community has widely agreed that Ant belongs to the past, the battle between Maven and Gradle is not to be settled shortly.

- Maven is the current standard and it is safe to assume most developers are familiar with XML. Gradle however uses Groovy, which is not a very popular language – developers often need to learn it only for the purpose of managing the build pipeline.
- Creating a simple project with Gradle doesn't require almost any configuration at all. However, as the project grows other time, the flexibility works as a double-edged sword – often the configuration requires lots of code and effort to maintain.
- Because configuration is code, it can contain bugs. Creating bugs in a declarative language such as XML is far more unlikely, thus Maven is often chosen despite the awkwardness and unintuitiveness of XML.
- Performance on the other hand is definitely in favor of Gradle, with all of its caching, incremental compilation and test detection mechanisms.

  The topics mentioned here are merely the tip of the iceberg, as issues such as dependency handling, multi-module support, plugins and external tooling support also play a big role in build tool adoption.

2.3.4 The sbt tool

The sbt tool was originally created in 2008. By 2010, when sbt 0.7 came out, many open-source Scala projects were using sbt as their build tool. By being designed from the ground-up with Scala in mind, it quickly came on track to become the default build tool for Scala. The sbt tool treats Scala as a first class citizen, which means every command and configuration option is in fact a Scala DSL. The user is also able to interact with a powerful CLI that can understand complex Scala syntax. This power allows for a a very high level of customization compared to other build tools. Unfortunately, the design principles incorporated in sbt are widely considered too complex and hard to grasp [31].

An sbt project is configured in a build.sbt file in a declarative fashion. The code resembles Scala, but is in fact a Scala DSL which has to be compiled before it is used. The process of building the project consist of the following steps:

- *build project* compilation,
- *build project* execution, which creates a *build definition*,
- interpretation of the *build definition*, which creates a *task* graph for the build,
- *task graph* execution, which means actually running the project build itself.

This hierarchy might seem confusing and for good reason. The author has found no sign of any other build tool that uses two layers of abstraction on top of a build definition. What is more, there is a *special project* (called project) inside an sbt project which can also contain some Scala code. Figure 2.4 illustrates the build process of a build definition on a sample project.

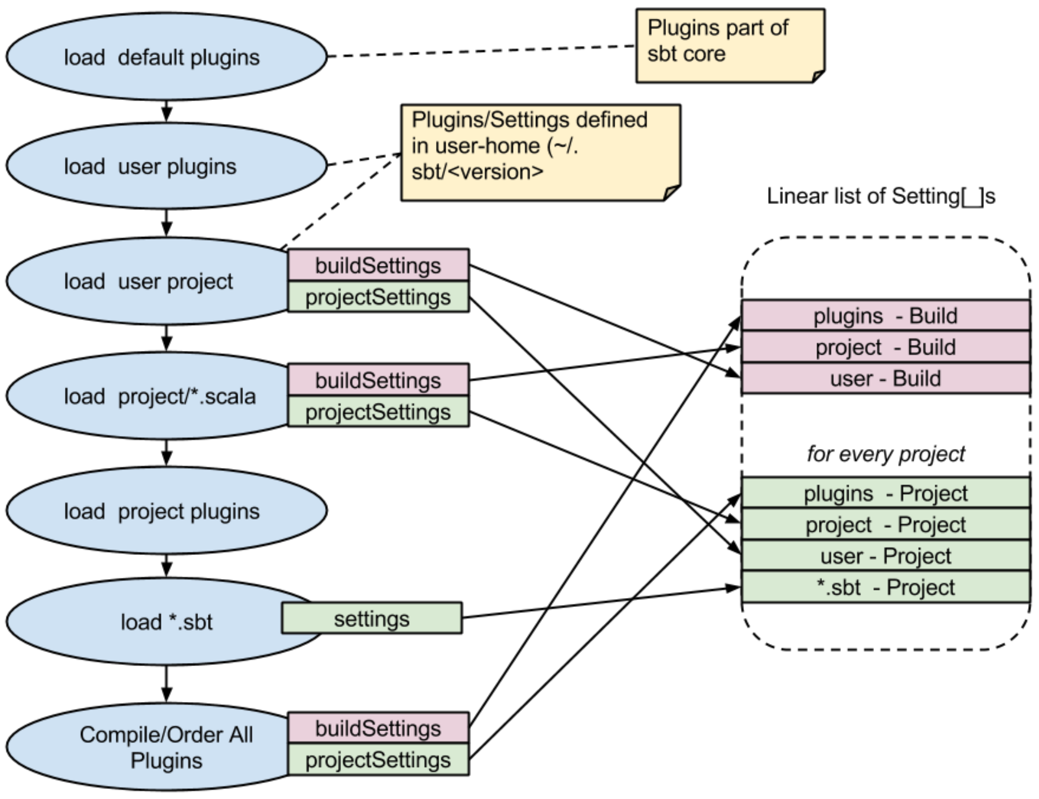


Figure 2.4: sbt task graph [32]

While the design does have its merits after deeper analysis, the entry treshold to sbt is very high compared to other build tools. Because it is currently established as the go-to build tool for Scala, it is unfortunate that potential Scala users might be discouraged by sbt. Even the Scala website currently recommends installing Scala as part of sbt, or with and IDE. But complexity is not the only characteristic of this build tool. It also offers certain advantages:

- Incremental compilation – one of the reasons for its success was the Scala-first approach not only in the build definition, but also execution. An efficient incremental compiler for Scala was one of sbt's key features. This compiler would later advance to a standalone solution, Zinc,
- parallel task execution,
- Scala-based build definition that can use the full flexibility of Scala code,
- supports mixed Scala/Java projects,

- modularization with sub-projects and external projects,
- extensive dependency management with inline declarations, external Ivy or Maven configuration files, or manual management,
- continuous compilation and testing with triggered execution.

The list does contain some unique features, yet there is one more downside to sbt, namely speed. Not the speed of tasks themselves – these are often quite efficient, or limited by Scala's performance. The speed of interaction however leaves much to be desired. Any invocation of sbt (even as simple as the clean command) takes upwards of 1 second to complete. That is due to necessary JVM startup and loading all the complex logic that allows beforementioned functionality. That argument does lose its power upon the discovery that sbt is supposed to be used as a REPL instead of running one-off commands. In case of REPL usage it does perform much better, but once again it is very different from the standard most developers are used to.

The sbt tool is the direct reason behind this thesis. The author, in his dissatisfaction with the tool, decided to attempt creating a build tool that would address the two biggest concerns about sbt – speed and ease of use. Other build tools mentioned in this chapter are treating Scala as a second-class citizen, but that doesn't make them any less competitive in the long run. It is in the intention of the author to create a tool that improves on sbt's downsides by applying concepts well established in the industry by the likes of Maven and Gradle.

2.4 Dependency Managment

As explained in chapter 2.3.2., it is considered a standard for Java modules and libraries to be identified by a set of attributes such as groupId, artifactId and version, as well as some optional ones like scope. The most popular artifact repositories are based on Maven in the sense that they use pom.xml to include information about themselves and their relevant dependencies. By convention, an aritfact is available at an url constructed from its descriptors, ie. `http://central.maven.org/maven2/org/easymock/easymock/3.5/` is an address in the central maven repository for an artifact *easymock* with a group *org.easymock* and version *3.5*. The URL contains a directory with a pom, jar, and checksums for both of them. These 3 files are necessary to obtain the artifact itself, information about it, as well as whether it was downloaded properly.

If no explicit scope value is provided in the pom, a compile scope is assumed for a depenency. This means that all libraries referenced in the pom must be present at compilation time to make sure no compilation errors occur. The dependencies are unfolded in a recursive manner, ie. dependencies of the dependecies must also be satisfied etc. It is necessary to have the entire dependency tree present at runtime for a given scope.

In many cases the dependency tree will reference the same artifact multiple times – often in different versions. A simple heuristic to choose the latest version specified is often a good enough approach, although many other heuristics are possible. A common term for describing issues with conflicting versions is called *dependency hell*. Maven uses a *nearest wins* strategy, which means that a version encounted earliest in the tree (descending from the root in BFS fashion) will be chosen.

2.5 Conclusions and remarks

In this chapter the author has introduced the Scala programing language. Incremental compilation was briefly presented and explained. The differences between internal and external DSLs were demonstrated. The evolution of Java build tools was reasoned upon by the analysis of tools such as Ant, Maven or Gradle. The sbt tool was referenced as the only important Scala-focused build tool. Dependency management overview was given.

All of this will be used to understand the choices the author has made many while designing and implementing his own build tool, presented in chapter 3. Becoming familiar with different approaches for managing a build lifecycle is essential to drawing correct conclusions from their successes and failures.

Chapter 3

FSBT, the Fast Scala Build Tool

3.1 Assumptions and goals

The goal of the author of this paper is to create a working build tool. As explained in previous chapters, build tools more often than not are highly complex programs, developed continually over many years by hundreds of programmers, often with support from large companies (such as Lightbend behind SBT or Google behind Dart). In order to complete this research in a reasonable time, the author has had to highly limit the features of the solution presented here. Therefore, the author has deciced to limit the scope of work to the following features:

- *Configuration file* – In order to control project specific settings, the tool must support a configuration file. The configuration file shall be loosely complaint with the SBT build.sbt file syntax, in order to be familiar to the majority of Scala developers. The configuration file should allow configuration of the following options:

  - setting the project name,
  - setting the project version,
  - configuring the project's dependencies and their scope.

  The configuration file should be mandatory for the build tool to work.

- *Managing dependencies* – The tool must be able to collect the dependency information defined in the configuration file. That information may contain the scope of the dependency, which should be treated accordingly. All dependencies should be resolved from remote repositories and cached locally for speed of use. All relations between dependencies should be determined and valid classpaths should be established for all supported operations.

- *Compilation* – The tool must be able to compile Scala and Java code. In order to achieve acceptable compilation speed, incremental compilation is required. Therefore, Zinc has been chosen as the compiler used for both Scala and Java.

- *Cleaning* – The tool must be able to clean the target directory of all compilation artifacts.
- *Running* – The tool must be able to run a compiled program with all valid settings. The run configuration should in part be deducted from the context.
- *Testing* – The tool must be able to run tests. The testing ecosystem on the JVM platform is immense, and each testing library or framework requires a diffrent approach from the author. For the scope of this implementation, JUnit and ScalaTest have been chosen to be supported as the most popular options for Java and Scala.
- *A basic CLI interface* – The tool must incorporate a command line interface to perform beforementioned operations. The interface should allow for swift interaction with the build tool, and minimize startup overhead.

According to the author, the set of features listed above is what one could consider a minimal build tool allowing for modern software development.

3.2 Architecture

The core element of the design is the Nailgun project [33]. Nailgun consist of a client, a server and a protocol for starting JVM programs from the command line without the overhead of starting a JVM. Typically upon starting a Java program, all necessary classes must be loaded onto the heap. This tends to take make startup time slow in case of most programs which are longer than a few lines of code. A startup time for a command-line tool of over 1-2 seconds seems unacceptable to the author – it is part of the reason why working with sbt is frustrating. Nailgun solves this problem by dividing a program into two components:

- a Java server with the application logic, which listens for incoming connections,
- a client written in C, responsible for making calls to the server. A native client ensures swift execution.

When this design is applied, one no longer executes program directly. Instead, the actual program is called by the C client, which passes the parameters such as where did the call originate from, or what parameters were supplied. While it does allow for swift usage, it does not come without a price:

- The overhead still exists on the first call – The server must be started before it can receive any call. This usually happens with the first execution of the client, which checks whether the server is up and running.
- Increased complexity – The program now consist of two independent parts, which means problems with communication between the two might occur. Adding to that, the end user of the program which uses this architecture should in most cases by aware of the existence of a client and a server

• Concurrent connections – it is possible to make multiple concurrent client requests to the server, which must assure that no conflicts occur.

The last problem on the list was addressed by the author with a coding architecture decision: the server should be as stateless as possible. This is achieved by implementing every component on the server in a purely functional manner. Each command sent from the FSBT clients is treated completely in separation from other ongoing tasks and no state is preserved between runs. With each invocation, the configuration file as well as all options and dependencies are parsed. No shared state means no potential conflicts and race conditions. Unfortunately, there are some forced exceptions to this rule, such as a the caching module, which is stateful by design. Figure 3.1 illustrates the coupling of main components of proposed build tool.

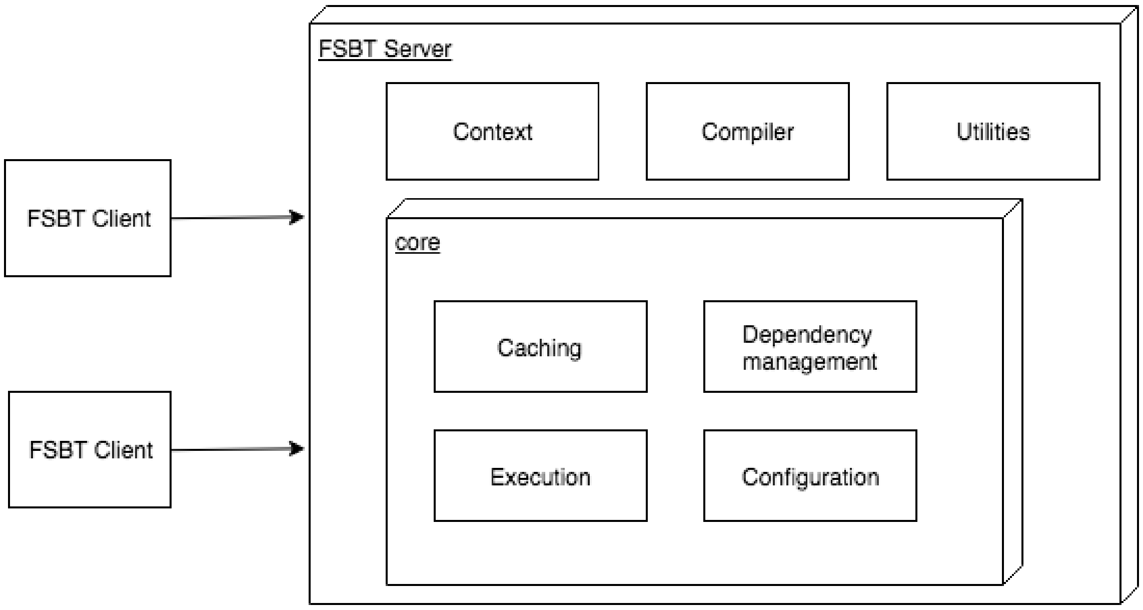


Figure 3.1: High-level architecture overview

3.3 Discovering the context

Running a program on the JVM platform involves passing a class containing a static main method to the Java runtime. In case of framworks, such as Spring or Play, the proper main method often resides inside the library classes instead of the source code of program. Some tools, such as CBT [34], detect all available options and leave it up to the developer to device which one to use. While it is a very good approach, a reasonable default option is also a good choice, one that should resonate better with less experienced developers.

For the simplest case, where the main method resides inside the source code, the author has chosen to use a simple heuristic based on compiled bytecode analysis. It is known that a valid main method is represented in the following form in the bytecode: *([Ljava/lang/String;)V*

With the use of Scala ASM library, FSBT is able to visit each compiled class (irrelevant to whether the source is written in Java or Scala). The ClassVisitor class presented in listing 3.1 is responsible for asserting whether the visited class contains a main method, in which case it saves its name in the ContextResolver.

Listing 3.1: Main method detection

```
class MyClassVisitor(contextResolver: ContextResolver) extends
    ClassVisitor(Opcodes.ASM4){

  override def visit(version: Int, access: Int, name: String, signature
      : String, superName: String, interfaces: Array[String]): Unit = {
    contextResolver.className=Some(name)
    contextResolver.superClass=Some(superName)
    super.visit(version,access, name, signature, superName, interfaces)
  }

  override def visitMethod(access: Int, name: String, desc: String,
      signature: String, exceptions: Array[String]): MethodVisitor ={
    if(name == ContextResolver.mainMethodName && desc ==
        ContextResolver.mainMethodDesc){
      contextResolver.mainMethodFound = true
    }

    super.visitMethod(access, name, desc, signature, exceptions)
  }

  override def visitSource(source: String, debug: String): Unit = {
    contextResolver.fileName = Some(source)
    super.visitSource(source, debug)
  }
}
```

In case of libraries which are expected to be the startup point of an application, there is no need for bytecode parsing. Instead, one can read the manifest of all libraries present on the classpath at run invocation on FSBT to detect which libraries support this configuration. This functionality has been skipped for the time being.

3.4 Scala parser combinators

As mentioned previously, a project should be configured in a single file only. The syntax of this file should be defined in an external Scala DSL. Thankfully, Scala has a core library for exactly this purpose. Scala Parser Combinators is designed for easy development of custom lexers and parsers. The author has leveraged this tool to design a configuration file format similar to the one used by sbt. Listing 3.2 contains a ConfigDSL class extending a JavaTokenParsers class. This extension supplies the class with a set of predefined Parsers for detecting simple expressions like stringLiteral , decimal or ident.

Listing 3.2: the ConfigDSL class

```
class ConfigDSL extends JavaTokenParsers {}
```

The class consists of multiple function definitions for parsing a text file to explicit objects. The first definition simply states that a configuration file consist of one or more expressions:

```
def configuration = rep(expression)
```

Further functions define what an expression can consist of, ie. an assingment of a value or one or more dependency definitions:

```
def expression = assignment | singleDep | multiDep | unmanagedJar
```

These functions are only used for aggregation of previous analysis. Let’s examine how a dependency is detected:

```
def singleDep = "libraryDependencies" ~ "+=" ~ rep(dependency) ^^ {
    case (a ~ b ~ c) => MapEntry(a, DependencyList(c)) }
```

This function contains syntax originating from the Scala Parser Combinators library. The following syntax ”libraryDependencies” ~ ”+=” ~ rep(dependency) is tailored to detect a string in the following form:

”libraryDependencies += ”junit” % ”junit” % ”4.12”

The interesting part here is the remaining part of the expression:

^^ { case (a ~ b ~ c) => MapEntry(a, DependencyList(c))\}

It means that each identifier detected in the string is assigned to a temporary variable, which can be used to construct final objects from this matching. Here was can see that we’re using the detected literals to create an instance of MapEntry with a nested DependencyList. All other config file values are imported to the program using this mechanism. The companion ConfigDSL object is responsible only for calling the parser mechanism and with the use of pattern matching returns the result, if it is successful. The matching mechism also works as a validation mechanism – if there is an expression that does not match any of the parser function, the parsing fails automatically. This takes care of the lexical validity of the configuration file. The logical validity is a more complex issue – for instance, dependencies may or may not have Scala version as a part of their artifact identifier. This can not however be deducted beforehand and to the knowledge of the author no other build tool is able to determine whether a dependency definition is valid.

Initially, the author had the intention to support a syntax as similar as possible to sbt. After consideration, it was determined that this approach might not be the best for following reasons:

- False promise – sbt syntax is a very complex DSL which is supposed to feel like Scala. However, it is merely a DSL and does not support all of the features of Scala ie. pattern matching.
- Simplicity – by getting rid of the false promise of Scala syntax the user is not tempted to use Scala in configuration constructs.

Thus, an as-simple-as-possible approach was taken. A sample fsbt configuration file is presented in listing 3.3.

Listing 3.3: Sample fsbt configuration

```
name = "fsbt-test-project"
version = "0.0.1"
scalatest_version = "3.0.1"

dependencies = {
    "junit" % "junit" % "4.12",
    "org.scalatest" %% "scalatest" % ${scalatest_version} % "test"
}
```

The only syntax element common with sbt is the dependency syntax, which was borrowed for its simplicity. A configuration file is ment to be no more than a set of variables, with no business logic. If that were to be supported in the future, it would be best to externalize it for a clear separation of concerns. For comparison, the author presents other build tool's configurations on listings 3.4-3.6:

Listing 3.4: Equivalent sbt configuration

```
name := "single-module"
organization := "com.example"
version := "1.0.0-SNAPSHOT"

libraryDependencies ++= Seq(
   "org.scalatest" %% "scalatest" % "3.0.0" % Test,
   "com.novocode" % "junit-interface" % "0.11" % Test
)
```

The sbt configuration is nearly identical, as explained above. The only difference is a more complex operator syntax.

Listing 3.5: Equivalent gradle configuration

```
plugins {
    id "com.github.maiflai.scalatest" version "0.18"
}

apply plugin: 'scala'

dependencies {
    compile 'org.scala-lang:scala-library:2.12.4'
    testCompile 'org.scalatest:scalatest_2.12:3.0.0'
    testCompile 'junit:junit:4.12'
    testCompile 'org.scalatest:scalatest_2.12:3.0.1'
    testRuntime 'org.pegdown:pegdown:1.4.2'
}

repositories {
  jcenter()
  mavenCentral()
}
```

We can see that gradle requires a bit more configuration due to not supporting Scala natively, also custom definitions for remote repositories and required. Nevertheless, the build file is quite readable.

Listing 3.6: Equivalent maven configuration

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0">
  <modelVersion>4.0.0</modelVersion>
  <artifactId>subproject0</artifactId>
  <packaging>jar</packaging>
  <groupId>com.example</groupId>
  <version>1.0.0-SNAPSHOT</version>
  <name>subproject0</name>
  <dependencies>
    <dependency>
      <groupId>org.scalatest</groupId>
      <artifactId>scalatest_2.12</artifactId>
      <version>3.0.0</version>
  </dependency>
  <dependency>
    <groupId>org.scala-lang</groupId>
    <artifactId>scala-library</artifactId>
    <version>2.12.4</version>
  </dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.7</version>
      <configuration>
        <skipTests>true</skipTests>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.scalatest</groupId>
      <artifactId>scalatest-maven-plugin</artifactId>
      <version>1.0</version>
      <executions>
        <execution>
          <id>test</id>
          <goals>
            <goal>test</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <groupId>net.alchim31.maven</groupId>
      <artifactId>scala-maven-plugin</artifactId>
      <executions>
        <execution>
          <id>add-scala-source</id>
          <phase>initialize</phase>
          <goals>
            <goal>add-source</goal>
            </goals>
            <configuration>
              <sourceDir>src/main/scala</sourceDir>
              <testSourceDir>src/test/scala</testSourceDir>
            </configuration>
          </execution>
          <execution>
            <id>compile-scala-source</id>
            <phase>compile</phase>
            <goals>
              <goal>compile</goal>
            </goals>
          </execution>
          <execution>
            <id>compile-scala-test</id>
            <phase>test-compile</phase>
            <goals>
              <goal>testCompile</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>net.alchim31.maven</groupId>
        <artifactId>scala-maven-plugin</artifactId>
        <version>3.3.1</version>
      </plugin>
    </plugins>
  </build>
</project>
```

As one could expect, Maven configuration does prove to be huge at 84 lines. Not only requires it a multiple of plugins, but also one must put much effort into understanding what exactly is configured.

3.5 Startup and command line interface

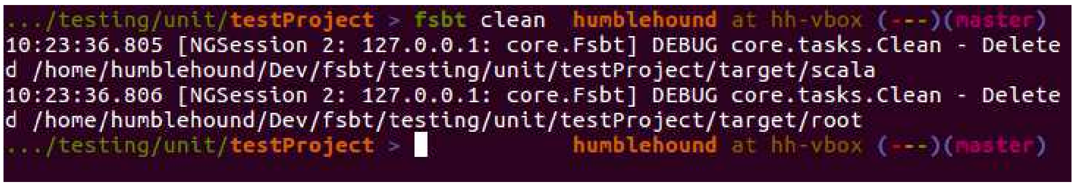
The fsbt tool is a client-server application. The server uses the Nailgun library handling requests and a bash script as a client. If the server isn't running, the client launches it at a specific port and waits for the startup process to complete. The end user is not supposed to know he is actually using a server. This is achieved by implementing the server as a completely immutable application – no state is preserved in the application memory. This ensures, that concurrent client calls are completely deterministic and no race conditions can occur.

An invocation of fsbt results in the following process:

- the fsbt bash script checks whether the server is already running. If it is not detected, the bash script finds the executable on the path, executes it and waits for the server to start listening for connections. This process currently adds less than 1 second to the execution time.
- Additional arguments, in this case compile options are passed on to the server in a request executed with the use of Nailgun binary client.
- The server receives the requests and creates a session object for it. Using the session data, it checks whether an fsbt configuration file is present at the caller location and verifies its integrity. If that succeeds, commands are mapped into corresponding tasks and executed sequentially.

As specified in chapter 3.1, the build tool is able to perform the following operations:

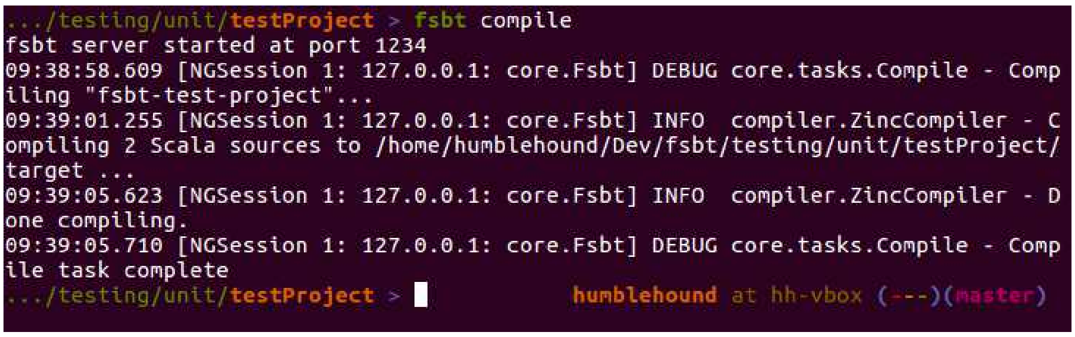
- clean compilation output (figure 3.2),



```
.../testing/unit/testProject > fsbt clean  humblehound at hh-vbox (---)(master)
10:23:36.805 [NGSession 2: 127.0.0.1: core.Fsbt] DEBUG core.tasks.Clean - Delete
d /home/humblehound/Dev/fsbt/testing/unit/testProject/target/scala
10:23:36.806 [NGSession 2: 127.0.0.1: core.Fsbt] DEBUG core.tasks.Clean - Delete
d /home/humblehound/Dev/fsbt/testing/unit/testProject/target/root
.../testing/unit/testProject >              humblehound at hh-vbox (---)(master)
```

Figure 3.2: fsbt clean example

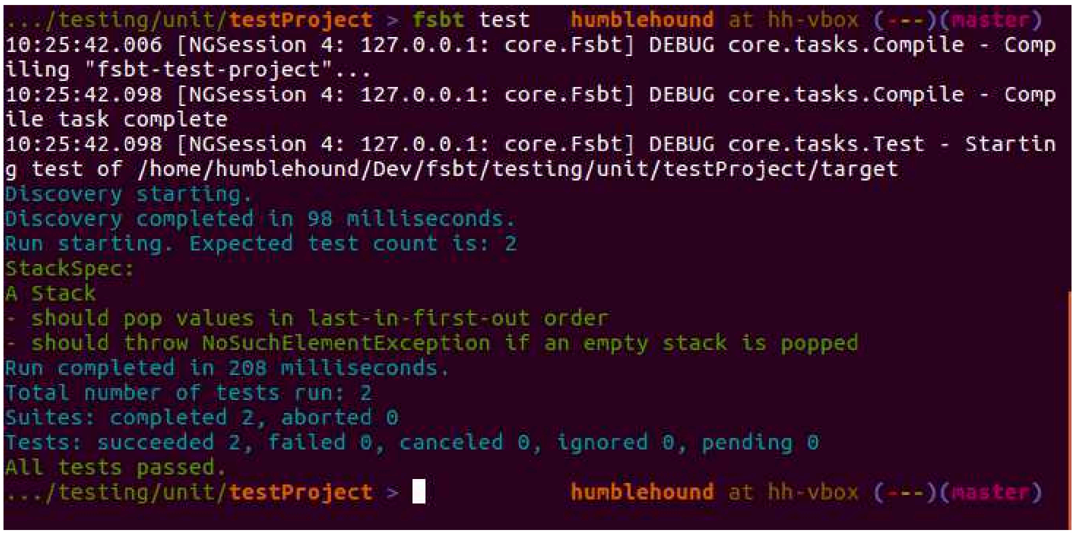
- compile source files (figure 3.3),



```
.../testing/unit/testProject > fsbt compile
fsbt server started at port 1234
09:38:58.609 [NGSession 1: 127.0.0.1: core.Fsbt] DEBUG core.tasks.Compile - Comp
iling "fsbt-test-project"...
09:39:01.255 [NGSession 1: 127.0.0.1: core.Fsbt] INFO  compiler.ZincCompiler - C
ompiling 2 Scala sources to /home/humblehound/Dev/fsbt/testing/unit/testProject/
target ...
09:39:05.623 [NGSession 1: 127.0.0.1: core.Fsbt] INFO  compiler.ZincCompiler - D
one compiling.
09:39:05.710 [NGSession 1: 127.0.0.1: core.Fsbt] DEBUG core.tasks.Compile - Comp
ile task complete
.../testing/unit/testProject >              humblehound at hh-vbox (---)(master)
```

Figure 3.3: fsbt compile example

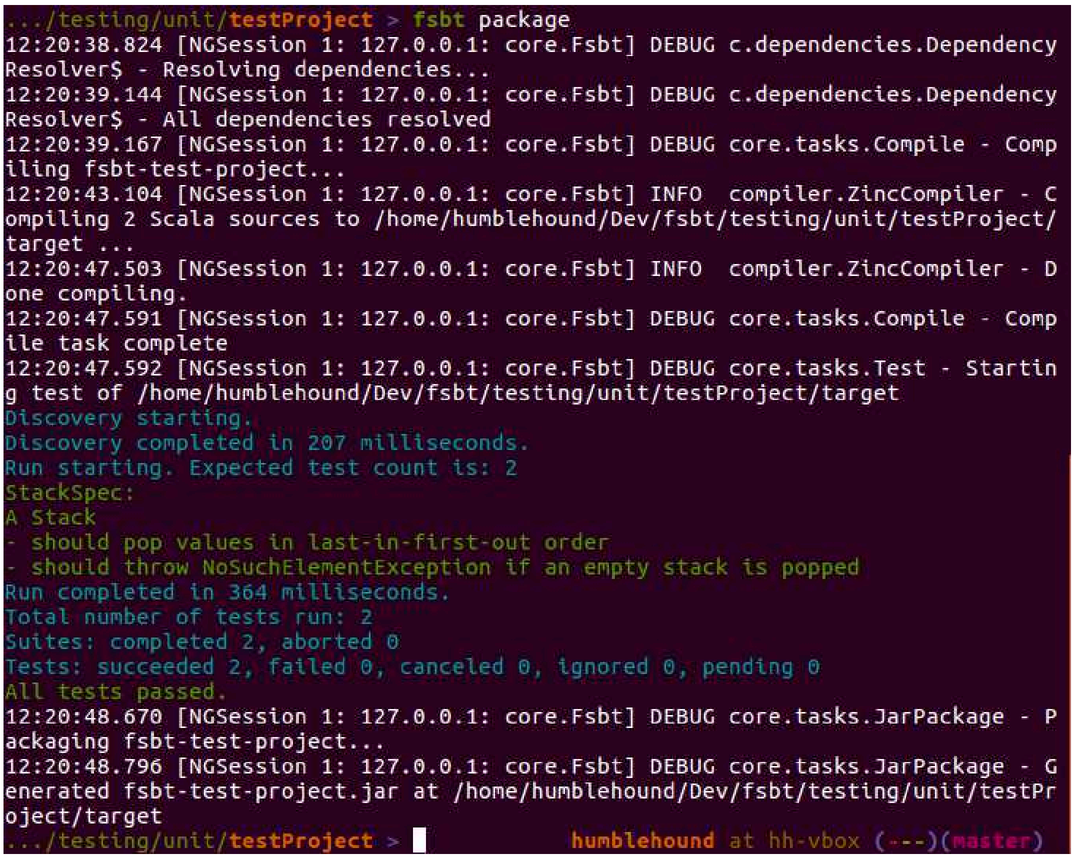
• run tests (figure 3.4),



```
.../testing/unit/testProject > fsbt test    humblehound at hh-vbox (---)(master)
10:25:42.006 [NGSession 4: 127.0.0.1: core.Fsbt] DEBUG core.tasks.Compile - Comp
iling "fsbt-test-project"...
10:25:42.098 [NGSession 4: 127.0.0.1: core.Fsbt] DEBUG core.tasks.Compile - Comp
ile task complete
10:25:42.098 [NGSession 4: 127.0.0.1: core.Fsbt] DEBUG core.tasks.Test - Startin
g test of /home/humblehound/Dev/fsbt/testing/unit/testProject/target
Discovery starting.
Discovery completed in 98 milliseconds.
Run starting. Expected test count is: 2
StackSpec:
A Stack
- should pop values in last-in-first-out order
- should throw NoSuchElementException if an empty stack is popped
Run completed in 208 milliseconds.
Total number of tests run: 2
Suites: completed 2, aborted 0
Tests: succeeded 2, failed 0, canceled 0, ignored 0, pending 0
All tests passed.
.../testing/unit/testProject >              humblehound at hh-vbox (---)(master)
```

Figure 3.4: fsbt test example

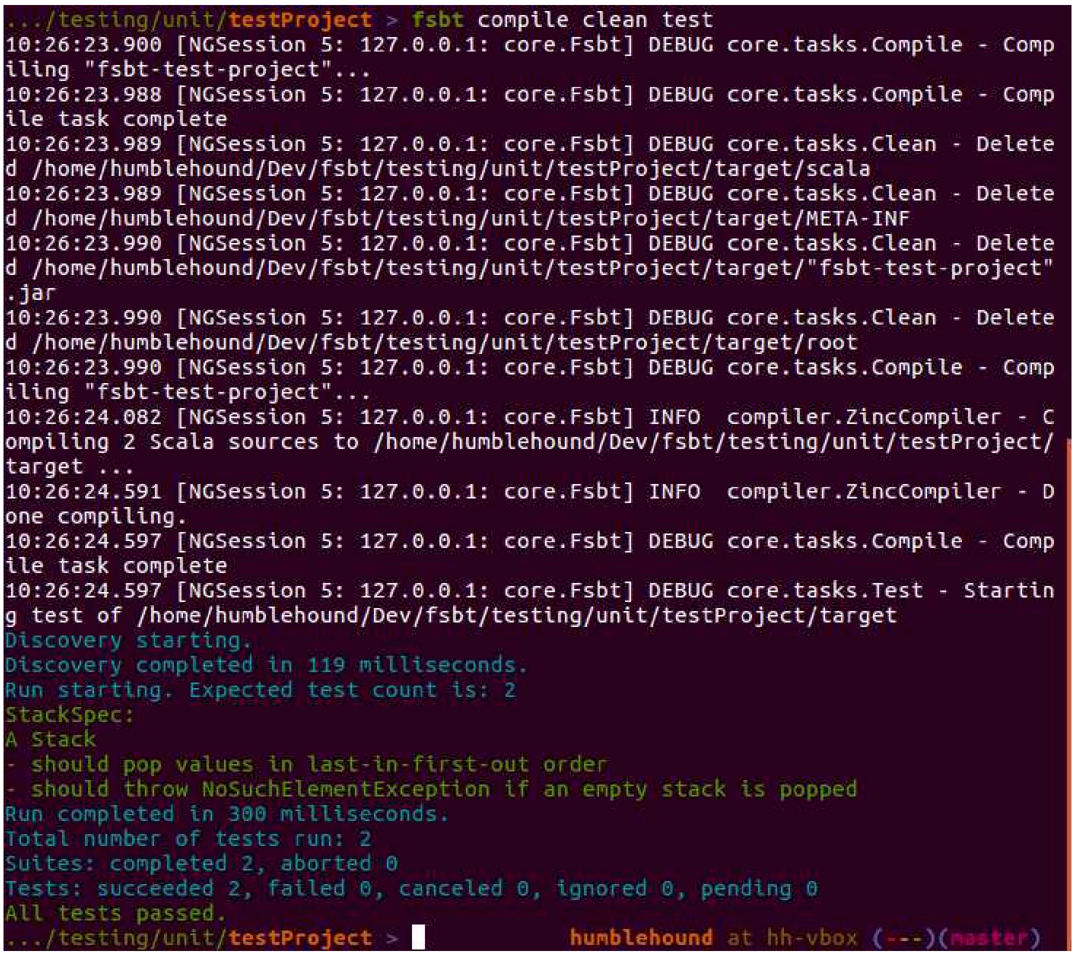
• package the compilation product into a jar (figure 3.5),



```
.../testing/unit/testProject > fsbt package
12:20:38.824 [NGSession 1: 127.0.0.1: core.Fsbt] DEBUG c.dependencies.Dependency
Resolver$ - Resolving dependencies...
12:20:39.144 [NGSession 1: 127.0.0.1: core.Fsbt] DEBUG c.dependencies.Dependency
Resolver$ - All dependencies resolved
12:20:39.167 [NGSession 1: 127.0.0.1: core.Fsbt] DEBUG core.tasks.Compile - Comp
iling fsbt-test-project...
12:20:43.104 [NGSession 1: 127.0.0.1: core.Fsbt] INFO  compiler.ZincCompiler - C
ompiling 2 Scala sources to /home/humblehound/Dev/fsbt/testing/unit/testProject/
target ...
12:20:47.503 [NGSession 1: 127.0.0.1: core.Fsbt] INFO  compiler.ZincCompiler - D
one compiling.
12:20:47.591 [NGSession 1: 127.0.0.1: core.Fsbt] DEBUG core.tasks.Compile - Comp
ile task complete
12:20:47.592 [NGSession 1: 127.0.0.1: core.Fsbt] DEBUG core.tasks.Test - Startin
g test of /home/humblehound/Dev/fsbt/testing/unit/testProject/target
Discovery starting.
Discovery completed in 207 milliseconds.
Run starting. Expected test count is: 2
StackSpec:
A Stack
- should pop values in last-in-first-out order
- should throw NoSuchElementException if an empty stack is popped
Run completed in 364 milliseconds.
Total number of tests run: 2
Suites: completed 2, aborted 0
Tests: succeeded 2, failed 0, canceled 0, ignored 0, pending 0
All tests passed.
12:20:48.670 [NGSession 1: 127.0.0.1: core.Fsbt] DEBUG core.tasks.JarPackage - P
ackaging fsbt-test-project...
12:20:48.796 [NGSession 1: 127.0.0.1: core.Fsbt] DEBUG core.tasks.JarPackage - G
enerated fsbt-test-project.jar at /home/humblehound/Dev/fsbt/testing/unit/testPr
oject/target
.../testing/unit/testProject >              humblehound at hh-vbox (---)(master)
```

Figure 3.5: fsbt package example

• run multiple commands in a sequence (figure 3.6).



```
.../testing/unit/testProject > fsbt compile clean test
10:26:23.900 [NGSession 5: 127.0.0.1: core.Fsbt] DEBUG core.tasks.Compile - Comp
iling "fsbt-test-project"...
10:26:23.988 [NGSession 5: 127.0.0.1: core.Fsbt] DEBUG core.tasks.Compile - Comp
ile task complete
10:26:23.989 [NGSession 5: 127.0.0.1: core.Fsbt] DEBUG core.tasks.Clean - Delete
d /home/humblehound/Dev/fsbt/testing/unit/testProject/target/scala
10:26:23.989 [NGSession 5: 127.0.0.1: core.Fsbt] DEBUG core.tasks.Clean - Delete
d /home/humblehound/Dev/fsbt/testing/unit/testProject/target/META-INF
10:26:23.990 [NGSession 5: 127.0.0.1: core.Fsbt] DEBUG core.tasks.Clean - Delete
d /home/humblehound/Dev/fsbt/testing/unit/testProject/target/"fsbt-test-project"
.jar
10:26:23.990 [NGSession 5: 127.0.0.1: core.Fsbt] DEBUG core.tasks.Clean - Delete
d /home/humblehound/Dev/fsbt/testing/unit/testProject/target/root
10:26:23.990 [NGSession 5: 127.0.0.1: core.Fsbt] DEBUG core.tasks.Compile - Comp
iling "fsbt-test-project"...
10:26:24.082 [NGSession 5: 127.0.0.1: core.Fsbt] INFO  compiler.ZincCompiler - C
ompiling 2 Scala sources to /home/humblehound/Dev/fsbt/testing/unit/testProject/
target ...
10:26:24.591 [NGSession 5: 127.0.0.1: core.Fsbt] INFO  compiler.ZincCompiler - D
one compiling.
10:26:24.597 [NGSession 5: 127.0.0.1: core.Fsbt] DEBUG core.tasks.Compile - Comp
ile task complete
10:26:24.597 [NGSession 5: 127.0.0.1: core.Fsbt] DEBUG core.tasks.Test - Startin
g test of /home/humblehound/Dev/fsbt/testing/unit/testProject/target
Discovery starting.
Discovery completed in 119 milliseconds.
Run starting. Expected test count is: 2
StackSpec:
A Stack
- should pop values in last-in-first-out order
- should throw NoSuchElementException if an empty stack is popped
Run completed in 300 milliseconds.
Total number of tests run: 2
Suites: completed 2, aborted 0
Tests: succeeded 2, failed 0, canceled 0, ignored 0, pending 0
All tests passed.
.../testing/unit/testProject >            humblehound at hh-vbox (---)(master)
```

Figure 3.6: fsbt complex use case example

3.6 Dependency mechanism

Dependency management is divided into two parts: scope resolution and dependency resolution. Scope resolution takes care of establishing valid identifiers for modules and querying remote repositories for the POM files describing their dependencies. In case of no such files, an error is returned.

Dependency mechanism has been based around the POM model. Each dependency may contain some dependencies, which can also have their dependencies etc. This recursive structure is handled by logic presented below.

Table 3.1 illustrates the relationship between dependencies – if a dependency is set to the scope in the left column, transitive dependencies of that dependency with the scope across the top row will result in a dependency in the main project with the scope listed at the intersection. If no scope is listed, it means the dependency will be omitted.

Table 3.1: Nested dependencies relationship

	compile	provided	runtime	test
compile	compile	-	runtime	-
provided	provided	-	provided	-
runtime	runtime	-	runtime	-
test	test	-	test	-

To give an example, if a project P has a compile dependency A, which has compile dependencies on dependencies B, C and a test dependency D, then A, B, C must be present on the classpath for compilation to succeed, and D for test to succeed. Once the relationship has been established, dependency resolution takes place. The corresponding jar files are downloaded and placed in the fsbt cache folder inside the user home directory.

3.7 Multi-module support

The implementation for this problem has been based on the Maven solution, but with slight modifications. Project A may have a dependency on Projects B and C, however B and C must not have a dependency on A to work. In other words, a top-bottom approach only is allowed. This enforces the developer to decouple the application properly. As a result, circular dependencies are also forbidden.

In case of a multi-module project, tasks are executed on submodules prior to the parent module. Figure 3.7 illustrates a simple, successful scenario of this flow. Tasks such as compile and test are executed in parallel, given that there are no dependencies between submodules.

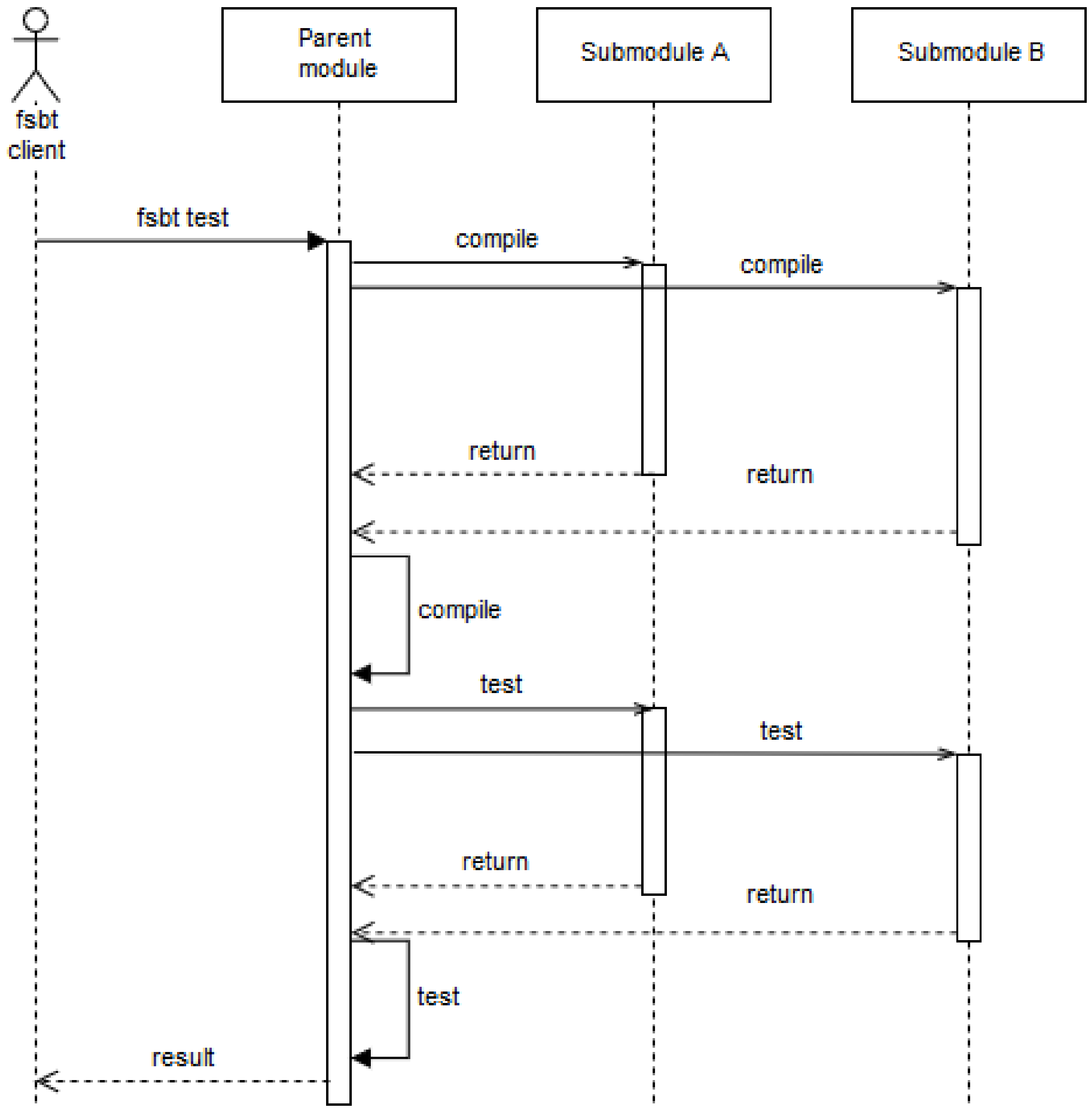


Figure 3.7: multi-module test execution diagram

3.8 Self dependencies

More often than not, submodules depend on each other i.e. one must be built before the other. That case has been illustrated in the figure 3.8. There are two types of relations between modules: submodule and depends_on. The submodule relation is used to describe a module that is a part of this module. The depends_on relation on the other hand is used to reference modules in other modules in the same project. This allows for building complex execution graphs. FSBT will try to optimize each task execution for maximum performance.

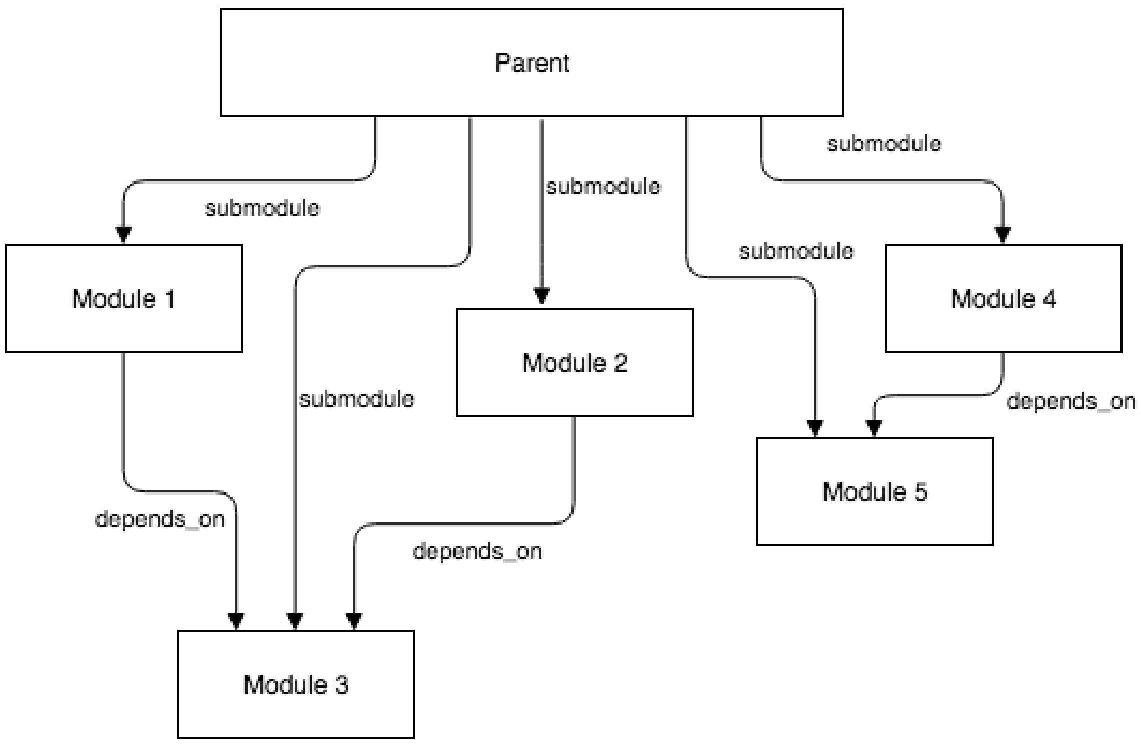


Figure 3.8: complex relations between modules

Figure 3.8 depicts a project with 5 submodules, where modules 1 and 2 are indirectly dependent on module 3, while module 4 depends on module 5. All modules are specified as submodules of the parent project, otherwise they would not be recognized by FSBT.

3.9 Execution schemes

Based on the information extracted from the project configuration FSBT will choose one of the following execution types, depending on the task specified:

- One-off execution – An execution scheme for tasks not directly related to the module configuration i.e. Running a compiled project or stopping the FSBT daemon
- For-all execution – For tasks which do not need to take into consideration any relations between modules it is viable to run the same task in parallel on all submodules i.e Running tests or cleaning the target directory
- Synchronized parallel execution – A scheme which analyses the relations between modules and starts tasks as soon possible, used for compiling. Figure 3.9 contains an example of such an execution.

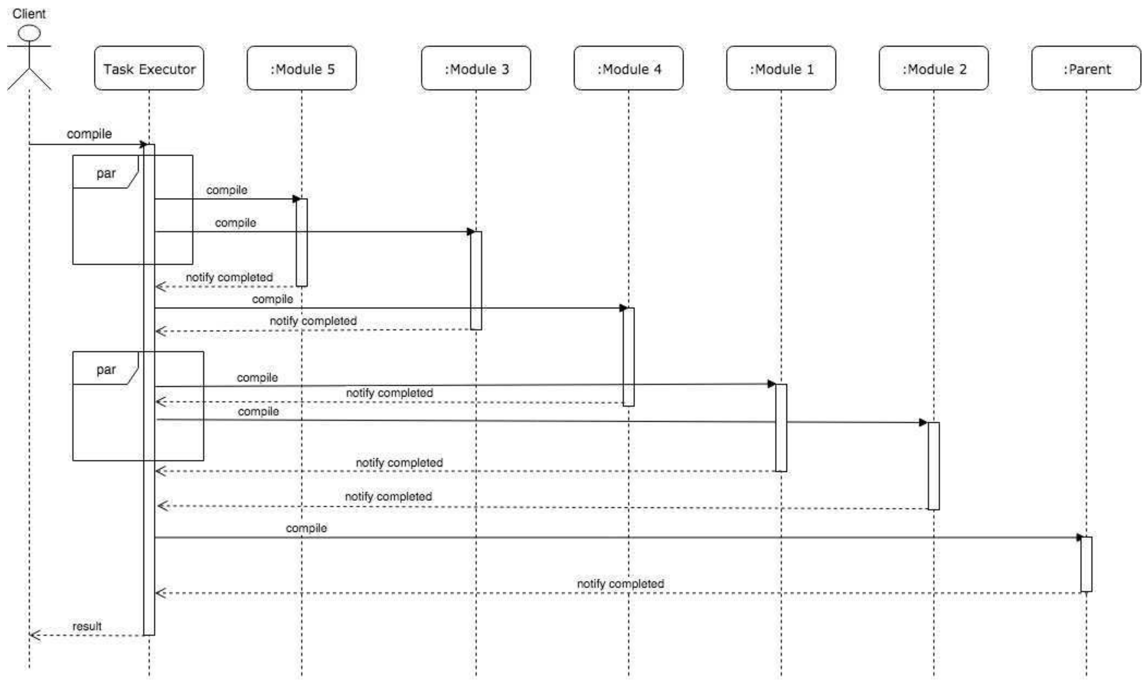


Figure 3.9: Synchronized parallel execution example

Figure 3.9 contains a representation of an execution plan for the project illustrated in the figure 3.8. Once a compilation task has been submitted from the client to the FSBT server, the Task Executor finds modules which are not dependent on by any other module, in this case Modules 3 and 5. Each of the modules notifies the executor of the completion. Because the executor keeps track of which modules need to complete for others to start, upon completion of Module 3 Modules 1 and 2 are ready to start. Module 5 behaves in the same fashion upon the completion of 4. Once all 5 modules have completed, the parent module is ready to start.

3.10 Incremental compilation

Incremental compilation is implemented by the Zinc compiler. The compiler defines an API to be consumed by the build tool. The application must resolve following concerns:

- Scala location – A complete set of Scala libraries must be present for the compiler to work. It is up to the build tool author to make sure it is available. In case of fsbt, Scala is bundled with fsbt so all libraries come from the fsbt classpath.
- Analysis cache – Zinc analyzes all source files before compilation and computes the relationships between class files. This allows for efficient recompilation. The results need to be stored on disk for easy reuse between launches. Fsbt caches the results in the user home directory.
- Classpath and source files – All dependencies for compilation must be resolved before invoking the compiler and put on the classpath. Detecting source files is handled with the use of regular expressions.

3.11 Testing

Most build tools use a dedicated plugin for each testing library. Since a plugin system is not in the implementation scope, a simple ScalaTest implementation is used. If ScalaTest is specified as a dependency on target project classpath, a dedicated ScalaTest runner is invoked in an external JVM. An internal call would be a much better approach, although no API is exposed to develop such functionality. ScalaTest allows for running ScalaTest as well as Junit tests.

3.12 Packaging

The packaging process on the JVM is relatively simple, as depicted in the Figure 3.10.

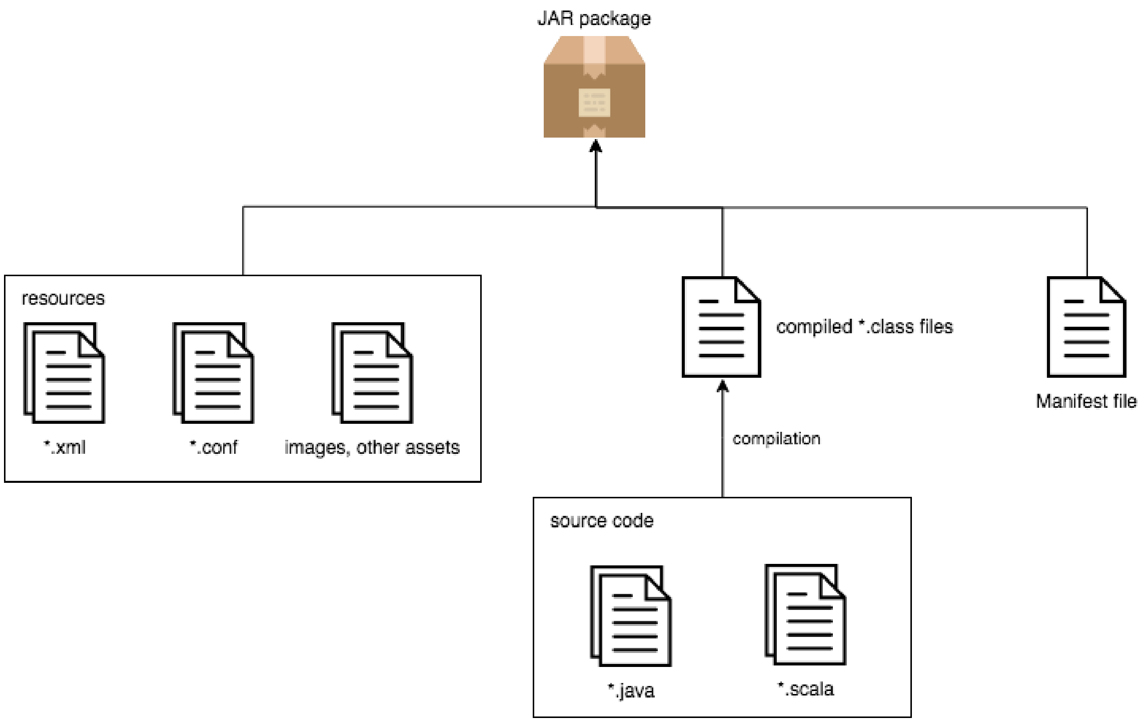


Figure 3.10: Building a jar file

A jar file is a distribution package for programs running on the JVM. It is based on the zip file format. It may contain the following elements:

- compiled class files,
- configuration files and other assets,
- a manifest file.

The manifest file

The manifest file should be called MANIFEST.MF to be recognized. It may contain the following information:

- Manifest-Version – the version of the Manifest specification, typically 1.0,
- Created-By – information about the author of the jar,
- Main-Class – defines the entry point of the packaged application,
- Sealed – used to limit the visibility of packages.

A manifest file might be divided into sections – the main section and individual sections, separated by blank lines, as shown in the listing 3.7.

Listing 3.7: A sample manifest file

```
Manifest-Version: 1.0
Created-By: 1.6.0 (Sun Microsystems Inc.)
Main-Class: com.java2s.Main
Sealed: true

Name: book/data/
Sealed: false

Name: images/logo.bmp
Content-Type: image/bmp
```

Individual sections always come after the main section and must start with a Name directive. They can be used to override setting from the main section and selectively expose content.

Because all JVM languages compile to class files, a jar file may contain code from all supported languages. The compiled class files are zipped along with all resource files, such as configuration files or images. The structure of the jar file reflects the structure of the program. Because Java runtime comes with a dedicated utility application called jar for creating jars, it is very simple task from the build tool perspective.

3.13 Conclusions and remarks

In this chapter the author has demonstrated the reasoning behind the fsbt tool. A scope of implementation was chosen. Architecture, as well as the impact of using a client-server architecture with the use of the Nailgun project was described. Core problems, such as creating a custom DSL, discovering the main method or resolving dependencies were described and solutions were presented. Execution schemes and examples of real world usage have been laid out. In the following chapter, a set of metrics will be introduced and a series of tests will be carried out in order to compare the fsbt tool with other established build tools.

Chapter 4

Comparison and experiments

In this chapter the author describes metrics used for measing the quality of build tools and uses them to perform experiments, results of which are presented and commented. Real world usage is also tested.

To asses the quality of fsbt, a set of metrics is required. This proves to be a challenge for the following reasons:

- No dedicated metrics – Measuring software complexity is a well known academic field, with a plenty of published papers [35] [36]. Unfortunately, they mostly focus on assesing executable source code complexity, not custom-tailored DSL. For this reason, metrics such as Cyclomatic Complexity or Information Flow Metrics are not well suited for build tools.
- Measuring user experience – User experience is one of the most important aspects of build tools. However, measuring it is greatly based on the domain and aspects of the product. Since the command-line interface is the most important interface to the client it would be difficult to apply any established measures [35].
- Test environment and configuration – Build tools are different by nature, and an option which is customizable in one tool might not be in another one. Furthermore, they might handle the same tasks differently by design. Design choices are debatable, but they do impact the possiblities of configuring a neutral enrivonment. To illustrate this, Maven requires explicit parameters to use multiple threads, while gradle and sbt automatically use all available cores.

4.1 Metrics

The author has decided on the following set of metrics:

1. **Halstead's Complexity metrics** While typically used to asses software (source code) complexity, Halstead metrics analyzes code as a sequence of operators and their associated operands. This approach seems perfectly suited for DSLs as well. Table 4.1 contains the

inputs for the Halstead measures, while table 4.2 contains the actual Halstead measures, along with the meaning and formula of each metric.

Table 4.1: Halstead measures input

Parameter	Meaning
n_1	Number of unique operators
n_2	Number of unique operands
N_1	Number of total occurrence of operators
n_2	Number of total occurrence of operands

Table 4.2: Halstead measures

Parameter	Meaning	Formula
n	Vocabulary	$n_1 + n_2$
N	Length	$N_1 + N_2$
V	Volume	$N * Log_2 n$
D	Difficulty	$\frac{n_1}{2} * \frac{N_2}{n_2}$
E	Efforts	$D * V$
B	Errors	$\frac{V}{3000}$
T	Testing time	$\frac{E}{18s}$

2. **Benchmarks**

   While the static metrics mentioned above do describe the information expressed by configuration, they don't test actual usage of the build tools. For this reason, a series of tests will be executed. Two test scenarios have been prepared:

   - compilation and recompilation speed of a single module,
   - compilation and testing speed of a complex module with submodules.

4.2 Experiments

4.2.1 Halstead's Complexity metrics

The metrics were calculated on a simple single-module project with two dependencies and zero non-necessary configuration. Due to diffrent approach and need for plugins in certian build tools, there were big differences in configuration, but they were made as minimal as possible. A sample sbt configuration used to compute its score, as shown in the listing 4.1:

Listing 4.1: sbt test configuration

```
name := "single-module"
organization := "com.example"
version := "1.0.0-SNAPSHOT"

libraryDependencies ++= Seq(
  "org.scalatest" %% "scalatest" % "3.0.0" % Test,
  "com.novocode" % "junit-interface" % "0.11" % Test
)
```

Halstead has only two input units: an operator and an operand. That might be not totally clear for build tools, because unlike complete programming languages, they're DSLs and undergo many stages of processing, thus it is occasionally hard to tell whether an indentifier is actually a new key (an operand) or a predefined value (an operator). Such conflicts were resolved by prefering the operator. Tables 4.3 and 4.4 contain calculated measures for fsbt, sbt, gradle and maven.

Table 4.3: Halstead measures part 1

Build tool	Vocabulary	Length	Calculated length
fsbt	20	25	68.81
sbt	19	28	61.75
gradle	26	32	101.95
maven	68	531	345.99

Table 4.4: Halstead measures part 2

Build tool	Volume	Difficulty	Efforts	Errors	Testing time
fsbt	108.05	3.00	324.14	0.04	18.01
sbt	118.94	4.95	588.76	0.04	32.71
gradle	150.41	3.30	496.37	0.05	27.58
maven	3232.44	21.21	68566.97	1.08	3809.28

The results are not surprising – all build tools come relatively close, with the exception of Maven. This stems from the fact the it is XML-based, which is why it involves a lot of boilerplate operators. The difference is so substantial, that even with the use of logarithmic scale it is hard to illustrate the differences on a plot without dwarfing other build tools. Figures 4.1, 4.2 and 4.3 contain difficulty, effort and volume plots respectively. In all cases Maven results were drastically off the scale.

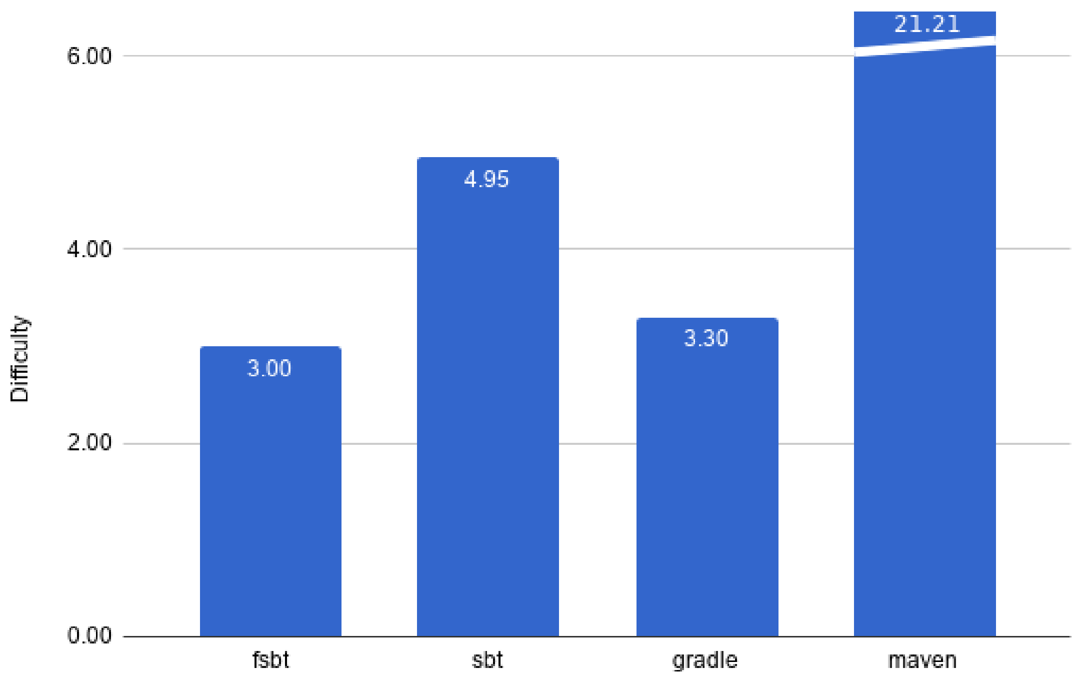


Figure 4.1: Difficulty comparison

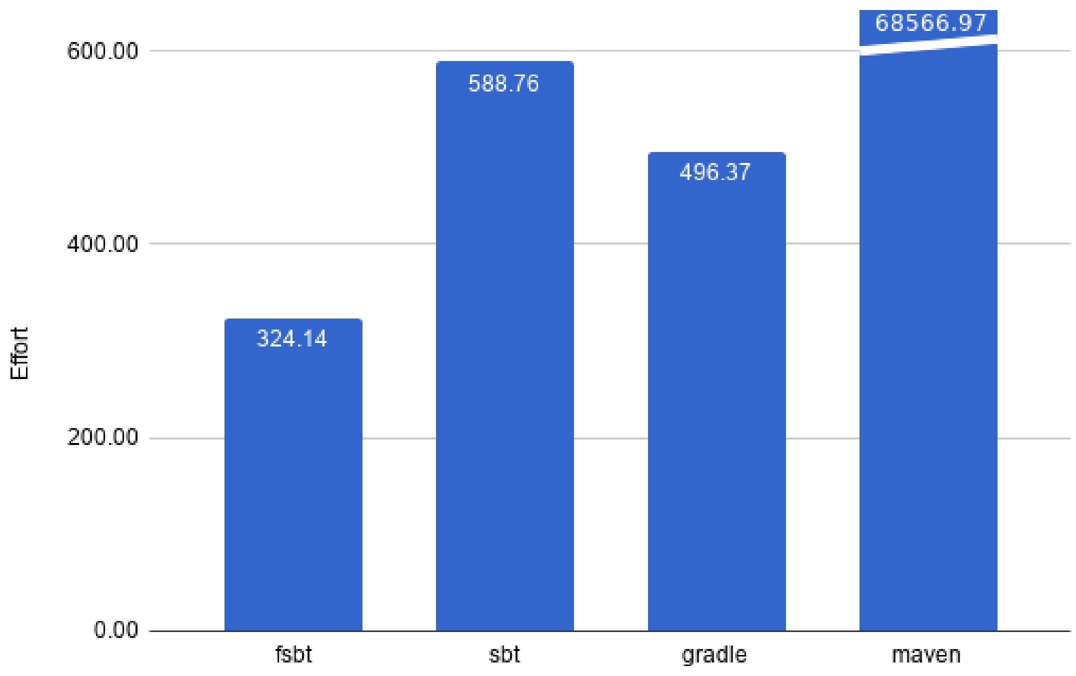


Figure 4.2: Effort comparison

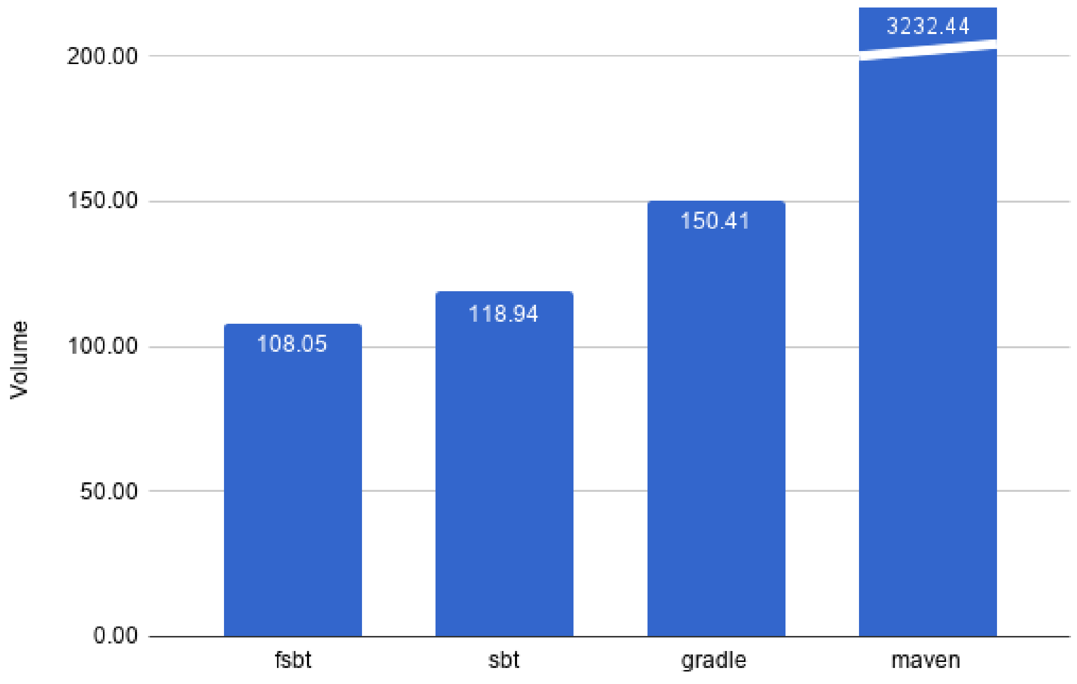


Figure 4.3: Volume comparison

All experiments have been carried out on an Intel Core i7-4760HQ @ 2.70Ghz with 4 cores and 16 GB of RAM. The test environment consisted of a fresh Ubuntu 16.04 installation inside a Virtualbox VM. The following build tools were compared with each other:

- Maven 3.3.9,
- Gradle 4.4.0,
- sbt 1.0.4,
- fsbt 0.0.1.

4.2.2 Compilation and recompilation speed of a single module

Figure 4.4 depicts average compilation times of a simple project consisting of a 100 Scala source files. Two scenarios were compared: initial compilation and recompilation after a change in a single source file.

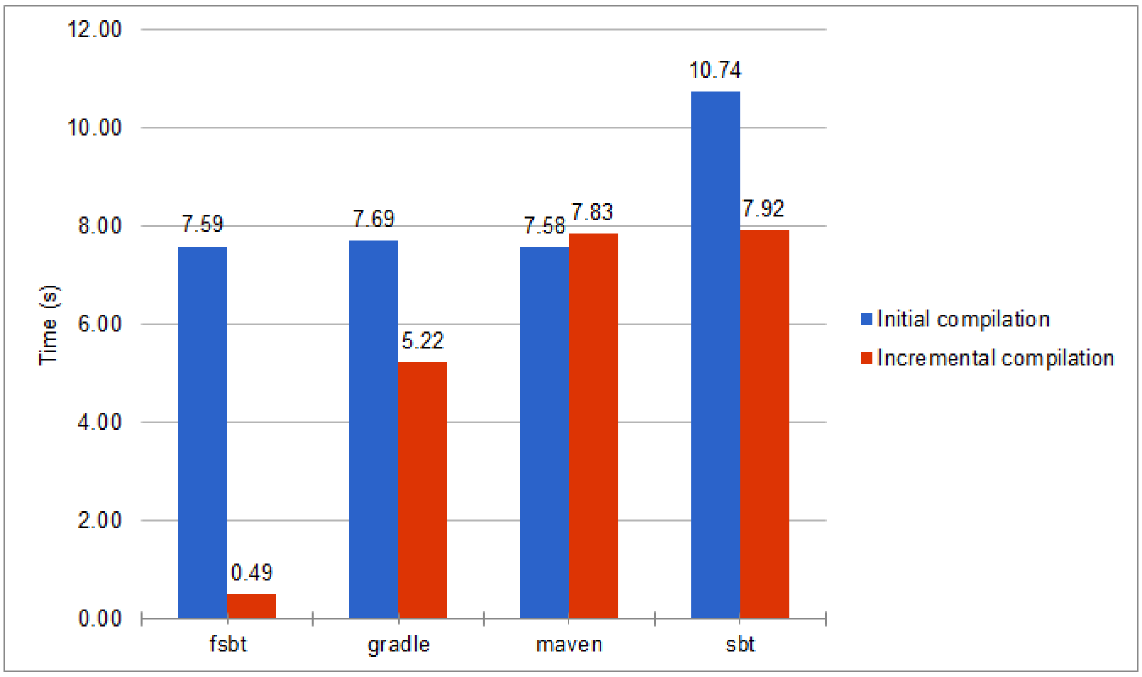


Figure 4.4: Comparison of Scala compilation times

The initial compilation times do not differ substantially – the average for each tool is comparable, with Maven being slightly faster than the rest. What does stand out, is sbt's slow compilation time – which is most likely the effect of sbt's long startup overhead (up to 7 seconds). Sadly this factor cannot be ommited from the test, for there is no way to measure sbt's performance in REPL mode.

The results get compilacated when examining recompilation. The huge diffrence between fsbt and its competitors comes from instant startup time and compilation result caching. If a comparison of only the time spent actually compiling by each tool was made, the graph would be much more balanced. However, it would not reflect the actual usage in real-world scenarios.

4.2.3 Compilation and testing speed of a complex module with submodules

Figure 4.5 depicts a comparison of compilation and testing of a project with 8 submodules. For the sequential run, all parallel options have been disabled in each build tool. For the parallel run, the thread count has been set to the number of available processors (4).

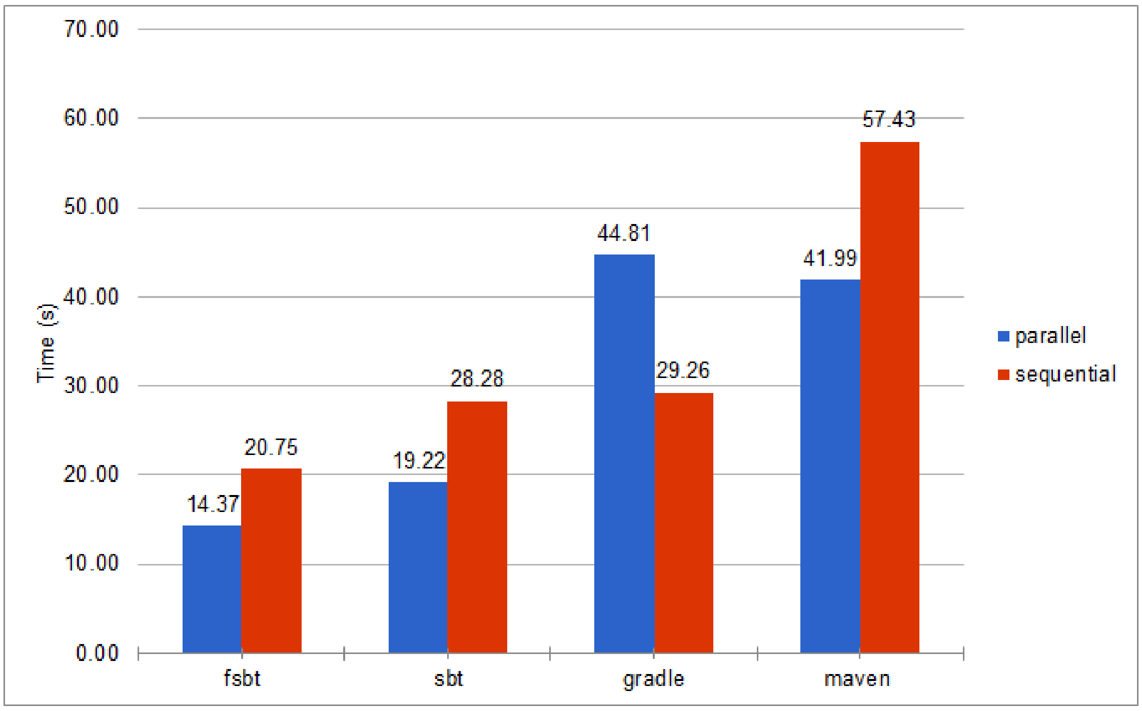


Figure 4.5: Compilation & subsequent testing benchmark

Fsbt is the fastest in both scenarios, with sbt slightly behind it. Interestingly gradle has taken much longer to process the submodules in parallel then sequentially by a large margin. This could be caused by inefficient thread distribution or large overhead of splitting the work.

4.2.4 Observations

To the satisfaction of the author, fsbt has outpaced or scored almost identically as all of its competitors in the benchmarks. Halstead metrics do yield some information, but the same conclusions could be made with a non-discrete assessment of the build files – the subtle diffrences between Gradle, sbt and fsbt are negligible, while Maven's enormous configuration size was to be expected.

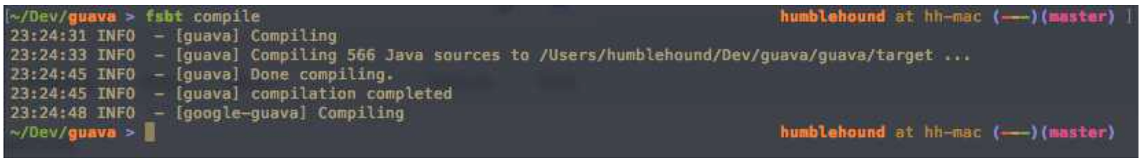
These metrics have little to do with real-world usage. In a development world full of linters, parsers, IDE support and intelligent autocompletion, it might often turn out that Maven is easier to use and understand than Gradle – all due to third-party support and simplicity (not in configuration expressions, but in problem representation).

4.2.5 Real world usage

Build tools must be highly practical, because they're a tool. Unlike many academic programs, they must be easy to use and be appealing to a broader population. The best metric for quality of fsbt would be a migration of a large open-source program or library to use it. Unfortunately, this has proven to be a big problem. Most successful open-source projects are quite large in size – this results in a complicated build process, which often relies on multiple of different plugins and approaches. It is fair to assume, that the larger the project, the more complicated the build process. To put things into perspective:

- Akka, a leading Scala library uses a near 500 line sbt build file,
- Play framework, one of the most popular web frameworks for Scala uses a 400 line sbt build file,
- Apache Spark, one of the most popular Scala tools uses a 3000 line pom.xml file.

The numbers used above are just for the build files aggregating all the modules. Nonetheless, the author was able to migrate one notable library to fsbt – Google Guava. After migrating the pom.xml file to build.fsbt, it was possible to resolve all dependencies and compile the core of Google Guava, as presented on Figure 4.6.



```
[~/Dev/guava > fsbt compile                                                   humblehound at hh-mac (——)(master) ]
23:24:31 INFO  - [guava] Compiling
23:24:33 INFO  - [guava] Compiling 566 Java sources to /Users/humblehound/Dev/guava/guava/target ...
23:24:45 INFO  - [guava] Done compiling.
23:24:45 INFO  - [guava] compilation completed
23:24:48 INFO  - [google-guava] Compiling
~/Dev/guava >                                                                 humblehound at hh-mac (——)(master)
```

Figure 4.6: Google Guava core compilation

The fsbt tool has resolved over 70 dependencies and compiled the program in about 12 seconds, as opposed to 17 seconds with the use of Maven. It is a significant result as it proves that the designed tool can be successfully applied to real-world projects.

Chapter 5

Summary

5.1 Conclusions

As stated in chapter 1, the goal of the author was to create a competitive build tool for Scala. While the road to wide adoption for any new open-source project is long and troublesome, the foundation was successfully laid for future work. A custom grammatic for configuration files with the use of DSLs was created. The created build tool, fsbt, was created in a client-server architecture and is capable of effectively parallelizing tasks which are dependent on each other. It has proven to be faster in many cases than its competitors. The context-oriented scope discovery and stateless approach have proven to be a good choice by the results of the experiments. All of these features were applied to a real-world, open source project created by Google, that is Google Guava, which was successfully built in a shorter time than with a build tool it is currently using. The success is largely in part due to current implementation scope, but with careful watch over the project's life on Github it should be possible to maintain this advantage. While competitors like Gradle or sbt offer far greater functionality, their development effort greatly exceed the one devoted by the author in terms of time and number of involved developers..

The results of the experiments show that fsbt is advantageous in terms of raw execution time. The Halstead metrics do show usability score on par with the rivals. The author must however note that if one was to do a real-world comparison, following features would have to be tested as well:

- IDE support,
- plugin count,
- community and corporate support,
- customizability.

These features have been omitted from this work due to the fact that fsbt is not able to compete with others in these regards. The effort of writing plugins for unusual use cases and IDE integration is not essential for a build tool to work, which is why it has been skipped. There are many other build tools that were not referenced, such as Pants, Bazel or CBT, however these are still in early phase of development and not documented well enough to be included in this paper.

5.2 Future work

With the foundation laid, the author does plan on releasing the first official version of fsbt shortly after the completion of this paper. If the project does attract interest from the developer community, development will be continued. The most important features to be implemented include:

- plugin system,
- deployment mechanism,
- additional configuration options,
- lifecycle hooks and tasks,
- IDE integration.

The source code of fsbt and the testing benchmark is available at `https://github.com/Humblehound/fsbt`. The author does hope that the project yields enough promise to be developed to a point of being a full-featured tool, ready for use in a production environment.

List of Figures

List of Tables

Bibliography

[1] Mini Shridhar, Bram Adams, and Foutse Khomh. A qualitative analysis of software build system changes and build ownership styles. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 29:1–29:10, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2774-9. doi: 10.1145/2652524.2652547. URL `http://doi.acm.org/10.1145/2652524.2652547`.

[2] Shane McIntosh, Bram Adams, Thanh H.D. Nguyen, Yasutaka Kamei, and Ahmed E. Hassan. An empirical study of build maintenance effort. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 141–150, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0. doi: 10.1145/1985793.1985813. URL `http://doi.acm.org/10.1145/1985793.1985813`.

[3] Shane McIntosh. Build system maintenance. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1167–1169, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0. doi: 10.1145/1985793.1986031. URL `http://doi.acm.org/10.1145/1985793.1986031`.

[4] Andrey Mokhov, Neil Mitchell, Simon Peyton Jones, and Simon Marlow. Non-recursive make considered harmful: Build systems at scale. In *Proceedings of the 9th International Symposium on Haskell*, Haskell 2016, pages 170–181, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4434-0. doi: 10.1145/2976002.2976011. URL `http://doi.acm.org/10.1145/2976002.2976011`.

[5] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1981. ISBN 0138221227.

[6] Matúš Sulír and Jaroslav Porubän. A quantitative study of java software buildability. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU 2016, pages 17–25, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4638-2. doi: 10.1145/3001878.3001882. URL `http://doi.acm.org/10.1145/3001878.3001882`.

[7] B. Adams, H. Tromp, K. de Schutter, and W. de Meuter. Design recovery and maintenance of build systems. In *2007 IEEE International Conference on Software Maintenance*, pages

114–123, Oct 2007. doi: 10.1109/ICSM.2007.4362624.

[8] Maria Consuelo Franky, Jaime A. Pavlich-Mariscal, Leonardo Giral, Andrea Barraza-Urbina, Luisa Barrera, and Angee Zambrano. Achieving software reuse and integration in a large-scale software development company: Practical experience of the lion project. *SIGSOFT Softw. Eng. Notes*, 40(4):1–9, July 2015. ISSN 0163-5948. doi: 10.1145/2788630.2788643. URL `http://doi.acm.org/10.1145/2788630.2788643`.

[9] Neil Mitchell. Shake before building: Replacing make with haskell. *SIGPLAN Not.*, 47(9): 55–66, September 2012. ISSN 0362-1340. doi: 10.1145/2398856.2364538. URL `http://doi.acm.org/10.1145/2398856.2364538`.

[10] Ahmet Celik, Alex Knaust, Aleksandar Milicevic, and Milos Gligoric. Build system with lazy retrieval for java projects. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 643–654, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4218-6. doi: 10.1145/2950290.2950358. URL `http://doi.acm.org/10.1145/2950290.2950358`.

[11] Shane Mcintosh, Bram Adams, and Ahmed E. Hassan. The evolution of java build systems. *Empirical Softw. Engg.*, 17(4-5):578–608, August 2012. ISSN 1382-3256. doi: 10.1007/s10664-011-9169-5. URL `http://dx.doi.org/10.1007/s10664-011-9169-5`.

[12] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[13] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide, 2Nd Edition*. Artima Incorporation, USA, 2nd edition, 2011. ISBN 0981531644, 9780981531649.

[14] Chris Birchall. ”a deep dive into scalac” - lecture, February 2017. URL `https://www.slideshare.net/Odersky/from-dot-to-dotty`.

[15] C.S. Horstmann. *Scala for the Impatient*. Pearson Education, 2016. ISBN 9780134540658. URL `https://books.google.pl/books?id=t57LDQAAQBAJ`.

[16] Martin Odersky. ”from dot to dotty” - lecture, February 2017. URL `https://www.slideshare.net/Odersky/from-dot-to-dotty`.

[17] Tiark Rompf and Nada Amin. Type soundness for dependent object types (dot). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 624–641, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4444-9. doi: 10.1145/2983990.2984008. URL `http://doi.acm.org/10.1145/2983990.2984008`.

[18] *sbt Reference Manual 1.0*, August 2018. URL `www.scala-sbt.org/1.0/docs/sbt-reference.pdf`.

[19] Chi Tau Robert Lai David M. Weiss. *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[20] Miklós Maróti Péter Völgyesi Greg Nordstrom Jonathan Sprinkle Gábor Karsai Ákos Lédeczi, Árpád Bakay. Composing visual syntax for domain specific languages. *Computer*, 2001.

[21] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010. ISBN 0321712943, 9780321712943.

[22] Martin Fowler with Rebecca Simmons. *Domain Specific Languages*. Pearson Education, 2010.

[23] Michael Fogus. ”baysick: A scala dsl implementing basic, March 2009. URL `http://blog.fogus.me/2009/03/26/baysick-a-scala-dsl-implementing-basic/`.

[24] Eric Burke Jesse Tilly. *”Ant: The Definitive Guide”*, 2002. URL `https://doc.lagout.org/Others/O%27Reilly%20Ant%20The%20Definitive%20Guide.pdf`.

[25] Brian Fox Jason Van Zyl Eric Redmond Larry Shatzer Tim O’Brien, John Casey. *”Maven: The Complete Reference”*. URL `http://books.sonatype.com/mvnref-book/reference/`.

[26] Jonathan Lalou. *Apache Maven Dependency Management*. Packt Publishing, 2013. ISBN 1783283017, 9781783283019.

[27] Maven lifecycle phases. `https://premaseem.wordpress.com/2012/04/25/maven-2-lifecycle-phases/`, .

[28] Maven pom image. `http://books.sonatype.com/mvnref-book/reference/pom-relationships-sect-pom.html`, .

[29] Adam Murdoch Hans Dockter. *Gradle User Guide*, 2007. URL `https://docs.gradle.org/current/userguide/userguide_single.html`.

[30] gradle tasks graph. `https://docs.gradle.org/current/userguide/img/javaPluginTasks.png`.

[31] So, what’s wrong with sbt? `http://www.lihaoyi.com/post/SowhatswrongwithSBT.html`. Accessed: 2017-11-12.

[32] sbt task graph. `https://www.scala-sbt.org/1.0/docs/Setting-Initialization.html`.

[33] Nailgun website. `http://martiansoftware.com/nailgun/`. Accessed: 2018-01-11.

[34] Chris’s build tool. `https://github.com/cvogt/cbt`. Accessed: 2018-02-12.

[35] D. Kafura and G. R. Reddy. The use of software complexity metrics in software maintenance. *IEEE Trans. Softw. Eng.*, 13(3):335–343, March 1987. ISSN 0098-5589. doi: 10.1109/TSE.1987.233164. URL `http://dx.doi.org/10.1109/TSE.1987.233164`.

[36] Akond Rahman, Asif Partho, David Meder, and Laurie Williams. Which factors influence

practitioners' usage of build automation tools? In *Proceedings of the 3rd International Workshop on Rapid Continuous Software Engineering*, RCoSE '17, pages 20–26, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-0428-1. doi: 10.1109/RCoSE.2017..8. URL `https://doi.org/10.1109/RCoSE.2017..8`.