UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Refactoring of a Stovepipe System

Structuring 101worker

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science
im Studiengang Informatik

vorgelegt von

Carsten Hartenfels

Erstgutachter:	Prof. Dr. Ralf Lämmel Institut für Softwaretechnik
Zweitgutachter:	Dr. Martin Leinberger Institut für Softwaretechnik

Koblenz, im Juni 2015

Erklärung

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

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

.....

(Ort, Datum) (Unterschrift)

Zusammenfassung

101worker ist die modulare Wissensverarbeitungskomponente des 101companies-Projektes. Durch organisches Wachstum des Systems, statt Beachtung von bewährten Software-Design-Prinzipien, haben sich Wartungs- und Leistungsprobleme entwickelt. Diese Arbeit beschreibt diese Probleme, entwirft Anforderungen für das Refactoring des Systems und beschreibt und analysiert schließlich die resultierende Implementierung. Die Lösung involviert die Zusammenfassung von verstreuten und redundanten Informationen, aufsetzen von Unit- und funktionalen Test-Suiten und Inkrementalisierung der Busarchitektur von 101worker.

Abstract

101worker is the modular knowledge engineering component of the 101companies project. It has developed maintainability and performance problems due to growing organically, rather than following best software design practices. This thesis lays out these problems, drafts a set of requirements for refactoring the system and then describes and analyzes the resulting implementation. The solution involves collation of scattered and redundant information, setup of unit and functional test suites and incrementalization of the bus architecture of 101worker.

Contents

1 Introduction **1**

1.1 Thesis Structure 2

2 Background **3**

2.1 101companies 3

2.1.1 101repo 3

2.1.2 101worker 4

2.2 Problems 5

2.2.1 Brittleness 5

2.2.2 Rigidness 6

2.2.3 Performance Issues 6

2.2.4 Garbage Accumulation 6

2.2.5 Imperviousness 7

3 Requirements **8**

3.1 Documentation 8

3.2 Deployment 9

3.3 Unit Tests 10

3.4 Functional Tests 10

3.5 Incrementality 11

3.5.1 Laziness 11

3.5.2 Communication 12

3.5.3 Bookkeeping 13

3.5.4 Recovery 13

3.5.5 Escape Hatch 14

4 Solution **15**
4.1 Restructuring 15
4.1.1 Environment 15
4.1.2 Modules 16
4.2 Functional Test Architecture 17
4.2.1 Branches 17
4.2.2 Test Definitions 17
4.2.3 Execution 18
4.3 Incrementality 19
4.3.1 Workflow 20
4.3.2 Grammar 21
4.3.3 Inter-Process Communication 22
4.3.4 Error Recovery 22
4.3.5 Library Support 23
4.3.6 Performance Analysis 25

5 Related Work **26**

6 Concluding Remarks **27**
6.1 Summary 27
6.2 Limitations 27
6.3 Future Work 28

List of Figures

4.1 101test workflow 19
4.2 module execution workflow 20
4.3 layers of abstraction 24

List of Listings

4.1 environment definition, compiled excerpt 15
4.2 test definition excerpt 18
4.3 diff generated by `git diff --name-status` in 101repo 19
4.4 101diff grammar 21

List of Tables

4.1 rules for merging diffs 23
4.2 performance of 101worker (approximate) 25

Chapter 1

Introduction

101companies[1] is a computer science community project, providing an open knowledge resource about software languages, technologies and related concepts [FLSV12]. Artifacts over which information is gathered is stored in a virtual sofware repository called *101repo* [FLL+].

Its computational component and focus of this thesis is *101worker*, a modular bus system. Each of its modules focuses on a different task of knowledge engineering over the 101companies repository, such as extraction of structured information, computation of metrics or validation tasks [FLL+]. Its bus architecture allows modules to utilize results of modules executed previously in the chain, through which data is derived and accumulated.

However, 101worker's responsibilities have grown over time, and due to a lack of a common interface, its modules have developed redundancies and inconsistencies with each other, making it difficult to understand and extend the system. Due to the continuous growth of 101repo, it has also developed performance problems. These issues are characteristic for a stovepipe or legacy system [BMIM98, p. 159].

To solve these issues, 101worker must undergo refactoring: redundancies should be removed by congregating common information and inconsistencies need to be resolved by providing a common interface. It is also necessary to increase performance and scalability, so that further growth of the project is not impeded.

[1] http://101companies.org/

1.1 Thesis Structure

Chapter 2 elaborates on the necessary background knowledge of the 101companies project and will detail the issues that 101worker faced before its refactoring.

The requirements to solve those problems are iterated in Chapter 3. Challenges expected to be faced during implementation are discussed as well.

How these challenges were overcome is detailed in Chapter 4, which illustrates the solution design and execution.

Chapter 5 will relate 101worker to similar bus systems, for which the solutions developed in this thesis may be applicable as well.

Finally, Chapter 6 concludes with a summary of the results, their limitations and which future work may build upon them.

Chapter 2

Background

2.1 101companies

The 101companies wiki describes the project as follows:

"The 101companies Project (or '101project' or even just '101' for short) is an open knowledge resource covering software technologies, technological spaces, software languages, and software concepts. [...] Contributions are maintained in the 101repo and documented on the 101wiki and organized in themes. All available knowledge is processed by the 101worker; derived resources are made available as 101data; all relevant resources are made available as Linked Data explorable through 101explorer." [10114a]

This thesis concerns itself with the *101repo* and *101worker* parts of the 101companies project. The necessary background will be laid out in the coming Sections.

2.1.1 101repo

101repo is a confederated virtual software repository [10114b], made up of several physical Git repositories [LM12] hosted on GitHub[1].

At the core, there is a root repository of the same name[2], which contains the structure and internal data, such as information about languages, technologies and a few contributions. The external repositories hold only contributions. On a local machine, they are connected to the base repository using symbolic links.

[1] https://github.com/
[2] https://github.com/101companies/101repo

All files in 101repo are called *primary resources*, from which all information is ultimately derived from (see Section 2.1.2).

2.1.2 101worker

The 101worker[3] is the focus of this thesis. This is a server-side application that performs knowledge engineering tasks for the 101companies project.

Its job is to gather structured information[4] from all the heterogeneous contents of 101repo. This metadata is used in linked data experiments involving 101companies [LLSV14] [Lei13] [KLL+13], such as the 101explorer[5].

The important parts of 101worker will be summarized as follows.

Derivation

The process of generating metadata from resources is known as *deriving*. As of the time of writing, all derived information is represented as files in the JSON format [jso13].

In the simplest case, resources are derived directly from the primary resources in 101repo, such as lines of code metrics [LLSV14]. However, these already derived resources may then be used to derive further resources, possibly even in conjunction with the primary resource or other derived resources. This allows the system to iteratively collect more and more detailed information, or summarize several derived resources over folders or contributions.

The resources created this way are called *derived resources*. Derived resources that only require the primary resources for their derivation are also known as *secondary resources*.

101meta, a library that is part of 101worker, provides an application programming interface for derivation of resources[6].

[3] https://github.com/101companies/101worker

[4] See http://worker.101companies.org/ for the raw metadata.

[5] http://101companies.org/resources?format=html

[6] https://github.com/101companies/101worker/tree/master/libraries/101meta

Modules

Modules are largely independent pieces of software, each of them representing a certain task to be performed by 101worker. For example, the *pull101repo* module[7] is responsible for retrieving 101repo from various places on Github and the *matches101meta* module[8] derives basic metadata about files, such as the language they are written in or which known technologies they use.

Runner

Since modules are independent programs, they need to be executed in the correct order to properly perform their work. This task is accomplished by the *runner*, which receives a list of modules as its input and runs them in the given order. It also handles logging-related tasks, such as the elapsed time or the exit code from each module run.

A list of modules is called a *configuration*. Running all modules from a configuration is called a *cycle* of 101worker.

2.2 Problems

101worker is suffering from the symptoms of a stovepipe system [BMIM98, p. 159]: it has grown organically to fit certain needs, with only a very loose architectural design. As such, it has developed several problems common to these kinds of systems, which will be discussed in the following.

2.2.1 Brittleness

Aside from two test configurations for the runner, the entirety of 101worker – be it modules, tools or libraries – lacks any kind of tests. This leads to uncertainty [Bec00, p. 46] over correctness and functionality.

The consequences are visible in production: about one quarter of modules fail outright, several others produce an empty or otherwise incorrect output.

[7] https://github.com/101companies/101worker/tree/master/modules/pull101repo

[8] https://github.com/101companies/101worker/tree/master/modules/matches101meta

2.2.2 Rigidness

Paths, URLs and other configuration data is scattered across 101worker. Some of them are stored in libraries, others are in Makefiles [SMS06] and yet others are simply static strings inside modules themselves.

Not only is this inconsistent and violates the principle of the single point of truth [Ray03, ch. 4], but the hard-coded nature of these values makes it impossible to set up alternative environments for testing.

2.2.3 Performance Issues

Modules in 101worker walk the 101repo directory tree on each run, reading every file and deriving new resources when applicable. This indiscriminate approach coupled with the growth of 101repo[9] has lead to massive performance problems. Even when there were no changes whatsoever since the last cycle, and therefore nothing to be done, executing the production configuration takes approximately 30 minutes.

Combined with the lack of unit tests or a small-scale test environment, development becomes a chore. Even minor changes mean waiting for an entire production cycle to complete.

There have been attempts to mitigate this by using parallel processing and file-modification-time-based approaches, but they have proved ineffectual due to the sheer amount of files being accessed.

2.2.4 Garbage Accumulation

Due to the simplistic module behavior of walking the file system described above, primary resources deleted from a repository will not be detected. Instead of the derived resources being deleted too, they will be left dangling. This data is garbage, as it does not represent any actual information over 101repo.

Currently, the only way to guarantee that no superfluous resources are left over is to completely delete all derived files and rebuild all metadata. However, due to uncertainty about modules still working as expected (see Section 2.2.1),

[9] 101repo contained 50 contributions in April 2011 and grew to 226 contributions, 53 concepts, 3 features, 42 languages and 98 technologies in April 2015.

this is not done as to not lose any resources that cannot be restored due to broken modules.

2.2.5 Imperviousness

It is very difficult and time-consuming to set up 101worker on a new machine. This causes a large entry barrier, especially for developers new to the project.

On the one hand, this is an issue of comprehension. Module documentation is scarce and scattered, while interface documentation is completely absent, even in libraries. Much of the knowledge about what a module or library does is only in the head of the original developers, if at all. On top of that, many developers do not actively work in the 101companies project anymore and may not be easily reachable for questions.

On the other hand, this is about deployment. There is no clear way to set up a functioning version of 101worker on a developer's system. The way it is currently handled is via trial and error, installing missing dependencies as they appear from module failures. Compounded with the speed issues described in Section 2.2.3, this leads to developers having to sink several hours into even a basic setup.

Chapter 3

Requirements

Each requirement in this chapter is structured by a short block summarizing the requirement, followed by a detailed description and a discussion of challenges involved. The summary block itself is structured as follows:

Requirement:	A name for the requirement, used to refer to it later.
Addresses:	Problems from Section 2.2, which the requirement aims to solve.
Summary:	A short description of what is to be done to fulfill the requirement.

3.1 Documentation

Requirement:	Documentation
Addresses:	Imperviousness
Summary:	Engineer a structure for documents and add missing documentation.

To improve comprehension of the system (Section 2.2.5), a standardized way to document elements of 101worker is needed.

This includes in-code technical documentation, usage information (READMEs) and high-level documentation. For modules, there is already a structured description format in use [Lei13, p. 24-25], which may be expanded upon.

It will be a challenge to document existing modules, as in many cases the knowledge of their inner workings only exists in the minds of their authors. However, once a documentation standard is in place, a plan to build comprehensive documentation can be worked out.

3.2 Deployment

Requirement:	Deployment
Addresses:	Imperviousness
Summary:	Enable developers to set up a local 101worker, installing all its dependencies.

To allow developers easier access to a local system and enable easier setup of a new instance of 101worker (Section 2.2.5), there needs to be a way to automatically install all dependencies necessary for a production run.

To achieve this, there must be a way to let programs in 101worker declare their dependencies on external programs and libraries. Every program relevant to production should include these dependency requirements.

Package management is inconsistent across various platforms and technologies [Wik15] [MBDC+06]. Which of these must be supported needs to be explored so that a focused solution can be developed.

3.3 Unit Tests

Requirement:	Unit Tests
Addresses:	Brittleness, Performance Issues, Imperviousness
Summary:	Add unit tests to elements of 101worker and find a way to run them all automatically.

Modules, tools and libraries should have unit tests that verify that they work as specified [Bec00, p. 45]. When changes are made, these tests will help identify breakage without having to run 101worker at all (Sections 2.2.1 and 2.2.3).

There should also be a standardized way of running these tests, to which all programs must conform. This should also enable running all tests in the entire system automatically, aiding developers in validating a new or modified installation (Section 2.2.5) and enabling them to easily be run frequently, even during development work [Bec00, p. 116].

Without any unit tests in the existing system, the only challenge here is to actually write them and standardize a way to execute them.

3.4 Functional Tests

Requirement:	Functional Tests
Addresses:	Brittleness, Performance Issues, Rigidness
Summary:	Engineer a system to run pre-configured runs of 101worker as functional tests.

There should be a way to test the 101worker system as a whole, without having to make a full production run or rely on potentially changing data in 101repo (Section 2.2.3). To realize this, a functional test [Bec00, p. 178] system needs to be implemented.

However, currently all modules in 101worker output their data into a fixed location relative to the 101worker directory[1]. Test output should not always go into the same location, as it would overwrite production data and data from other tests. Some way of redirecting module output on demand must be engineered instead (Section 2.2.2).

3.5 Incrementality

Requirement:	Incrementality
Addresses:	Performance Issues, Garbage Accumulation
Summary:	Refactor 101worker to operate incrementally on the set of changes since its last run, rather than iterating over the entire repository on every run.

The problems of performance (Section 2.2.3) and accumulation of garbage data (Section 2.2.4) are rooted in the system's behavior of simply iterating the directory tree on each cycle. The solution to this is for 101worker to behave *incrementally* instead.

This requirement is made up of several aspects, which will each be described in the following subsections.

3.5.1 Laziness

Requirement:	Incrementality – Laziness
Addresses:	Performance Issues
Summary:	Only work on resources that changed since the last run.

[1] https://github.com/101companies/101worker/blob/3c7db0/README.md#module-contracts

Modules should only ever do as much work as necessary. That is, they should only ever derive resources that have been added or modified, and not bother with the ones that have stayed the same (Section 2.2.3).

Ideally, this should not require modules to check every file on the disk, as this may cause scalability issues if 101repo further grows in size. Instead, the changes should be available directly, so that only relevant files are accessed in the first place.

3.5.2 Communication

Requirement:	Incrementality – Communication
Addresses:	Garbage Accumulation
Summary:	Find a way to communicate changes occurring in each module to the following modules.

Modules should not only consume which changes occurred, they must also communicate which changes they themselves have caused for modules following after them, which may derive resources from other already derived resources.

A protocol for handling this communication must be defined and there should be a library for modules to programmatically deal with this protocol.

A problem that needs to be solved is what kind of inter-process communication should be used to realize this protocol. There are a wide-array of options available in Unix-like systems [Ste99], from which the ideal one must be chosen.

The simplest solution would be to use pipes [Ste99, p. 44] [Ray03, ch. 7], simply sending the changes to a module's standard input and reading the changes it produces from its standard output. However, modules already use their standard output for logging information[2], which makes implementing a protocol on top of it problematic. These bidirectional pipes are also prone to causing deadlock [Ste99, p. 56], which would have to be worked around.

Other solutions to inter-process communication include temporary files, sockets, shared memory or message queues [Ray03, ch. 7]. However, none of them are

[2] https://github.com/101companies/101worker/blob/15cc6dd/modules/predicates101meta/program.py#L11-L12 for example

as common and simple as pipes, nor do they model the type of communication quite as well.

3.5.3 Bookkeeping

Requirement:	Incrementality – Bookkeeping
Addresses:	Garbage Accumulation
Summary:	Handle deleted resources by also deleting or updating resources derived from them.

When resources are deleted, resources that are derived from them must be updated correctly (Section 2.2.4). Specifically:

- If all resources that a derived resource depends on have been deleted, the derived resource must be deleted too.
- Otherwise, if the resource only depends partially on the deleted one, it must be updated instead.

3.5.4 Recovery

Requirement:	Incrementality – Recovery
Addresses:	Garbage Accumulation
Summary:	On module failure, keep list of changes and recover them on its next run.

If a module fails to run, the changes it was supposed to handle must not be lost (Section 2.2.4). Instead, the same list of changes should be stored and recovered on the next 101worker run.

Additionally, the recovered changes must sensibly be merged with the changes the module is supposed to handle in the subsequent worker run.

3.5.5 Escape Hatch

Requirement:	Incrementality – Escape Hatch
Addresses:	Garbage Accumulation
Summary:	Disable incrementality features if 101worker codebase changes, so that modified modules can re-derive their data.

To keep derived data up to date in the face of changes (Section 2.2.4), it is in some cases necessary to forego incrementality in favor of re-deriving all files. For example, when a module's codebase is modified to output its data differently, all data the module previously derived needs to be updated to the new format. Modules should trigger this escape behavior automatically when they detect such a breaking change.

However, this must not conflict with the Bookkeeping requirement (Section 3.5.3) – deletions must be handled in all cases to prevent collection of resources derived from a source that does not exist anymore.

Chapter 4

Solution

4.1 Restructuring

As a prerequisite to several requirements, parts of 101worker have been restructured. An overview of the important changes will be given in the following.

4.1.1 Environment

```
# Directories
web101dir    : $output101dir/101web/
dumps101dir  : $web101dir/data/dumps/

# Files
config101    : $worker101dir/configs/production.json
rules101dump : $dumps101dir/rules.json

# URLs
repo101url   : https://github.com/101companies/101repo
wiki101url   : http://101companies.org/wiki/
```

Listing 4.1: environment definition, compiled excerpt

To mitigate the issue of repeated and hard-coded information (see Section 2.2.2), path and URL data has been made available as configuration files. All

existing data scattered over 101worker has been consolidated into a single configuration file used in production, see listing 4.1 for an example. Additional configuration files are used for tests, and may also be used for other alternate environments (see Section 4.2).

To make the configuration information available to programs, environment variables are used. The runner loads these variables at startup and all modules, which are child processes of the runner, receive access to them [Ray03, ch. 10].

Hard-coded paths in libraries and modules have been replaced with references to these environment variables. Existing configuration systems specific to Python[1] and Make[2] have been deprecated and modified to dispatch to the current environment configuration. Once all modules have been refactored to use the environment directly, these redundant systems can be removed.

4.1.2 Modules

Modules have received restructuring and refactoring in several different aspects. Module contracts for developers have been updated as well[3].

Descriptions

Module descriptions [Lei13, p. 24-25] have been extended and made mandatory. They now not only detail which resources a module outputs through derivation, but are also used to declare dependencies on other modules and environment variables (see Section 4.1.1).

These declarations are also validated by the runner, allowing discovery of missing dependencies before a cycle is started.

Documentation

Technical, usage and high-level documentation of modules has been standardized in the aforementioned module contracts. Samples and templates have been

[1] https://github.com/101companies/101worker/blob/master/libraries/101meta/const101.py

[2] https://github.com/101companies/101worker/blob/master/modules/Makefile.vars

[3] https://github.com/101companies/101worker#module-contracts

provided. This fulfills Requirement 3.1

At the time of writing, original developers of modules are being mobilized to document their modules appropriately.

Incrementalization

Modules that were suffering from accumulating garbage (Section 2.2.4) or performance issues (Section 2.2.3) have been refactored to be incremental, which addresses Requirement 3.5. See Section 4.3 for the details of the incrementality system and Section 4.3.6 for the results regarding performance.

4.2 Functional Test Architecture

The functional test architecture developed for 101worker has been named 101test[4], which fulfills Requirement 3.4. These functional tests test are meant to ensure that the system is functioning as a whole and produces the correct output, rather than verifying correctness of small code fragments as unit tests do [Bec00, p. 118].

4.2.1 Branches

It is necessary to simulate changes in 101repo so that new incrementality data can be generated. However, it would be very impractical to actually modify a Git repository for every test case. Additionally, there would need to be some way to define the changes in the repository so that they could be replicated when a test is run.

So instead, Git branches [LM12, ch. 7] are used. Testers create branches that represents revisions of 101repo and instead of just pulling the repository on every 101worker run, the branch specific to the test case is checked out instead.

4.2.2 Test Definitions

Tests are defined in a declarative manner in the YAML format [yam11] (see listing 4.2 for an excerpt). Each test is broken up into one or more test cases,

[4] https://github.com/101companies/101test

```
name    : test
tests   : 3
command : make -s test-test.debug

1 :
  diff  :
    101repo/test  : A
  files :
    101repo/test  :
      exists  : 1
      content : "This is a test file.\n"
    101repo/test2 :
      exists  : 0

# more test cases follow
```

Listing 4.2: test definition excerpt

which in turn contains information about which branch is to be pulled, which command is to be run and what constraints are to be checked afterwards.

These constraints include validating the final diff's contents (see Section 4.3), ensuring a file is present or absent and comparing a file's content with an expected result. Since virtually all of 101worker's derived resources are in JSON format, validating JSON output is supported too.

4.2.3 Execution

When a test is run, it iterates over the test cases defined for it. For each test case, a specified branch of the 101test repository is checked out to simulate pulling a new version of 101repo.

Subsequently a 101worker cycle is initiated using a subset of modules defined the test case. The output from this cycle is redirected into a directory specific for the test, so that production and other test data is not affected (see Section 4.1.1).

Once the cycle completes, the output is checked for validity, using the test's definition as described in Section 4.2.2. Following that, a new test cycle begins. See also figure 4.1 for a visual representation of this workflow.

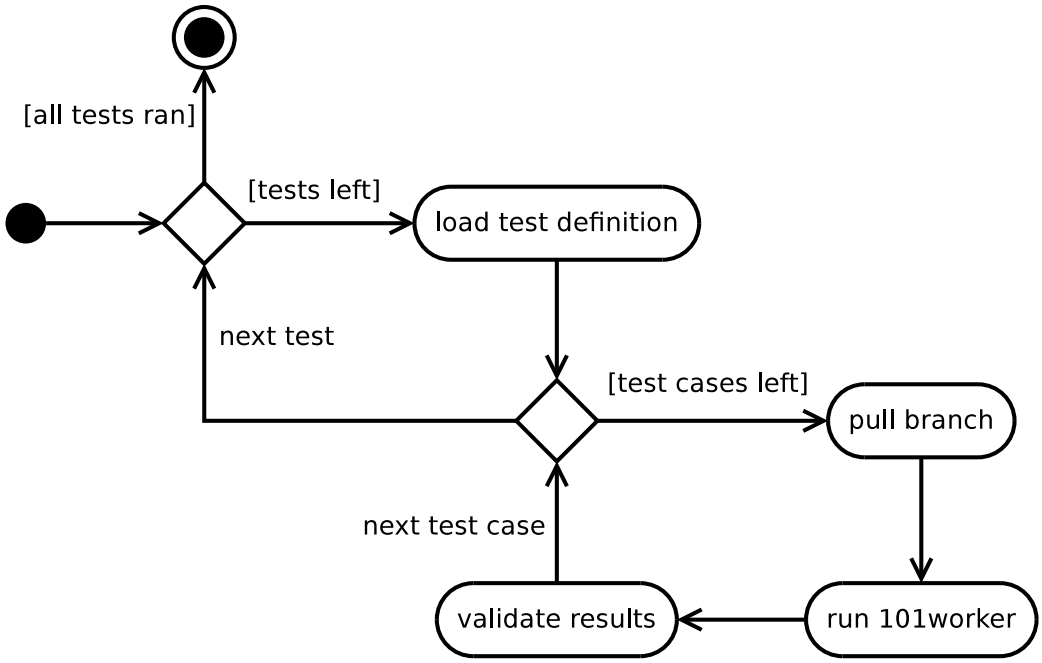

Figure 4.1: 101test workflow

4.3 Incrementality

```
1 D     concepts/Record_type/Point.hs
2 M     contributions/hibernate/scripts/CreateTables.sql
3 M     languages/Java/HelloWorld.java
4 D     languages/Java/sample/helloWorld.java
5 A     technologies/CSharpFragmentLocator/Company.cs
6 A     technologies/CSharpFragmentLocator/Makefile
```

Listing 4.3: diff generated by `git diff --name-status` in 101repo

To handle communication of changes for the requirement of incrementality (Section 3.5), a protocol named *101diff* has been developed. It is based on Git's name-status diff format (see listing 4.3). A list of changed files associated with the change operation (added, modified, deleted) will be called *diff*.

The different aspects of the incrementality solution will be discussed in the subsequent Sections, ending with an analysis of the resulting performance gains.

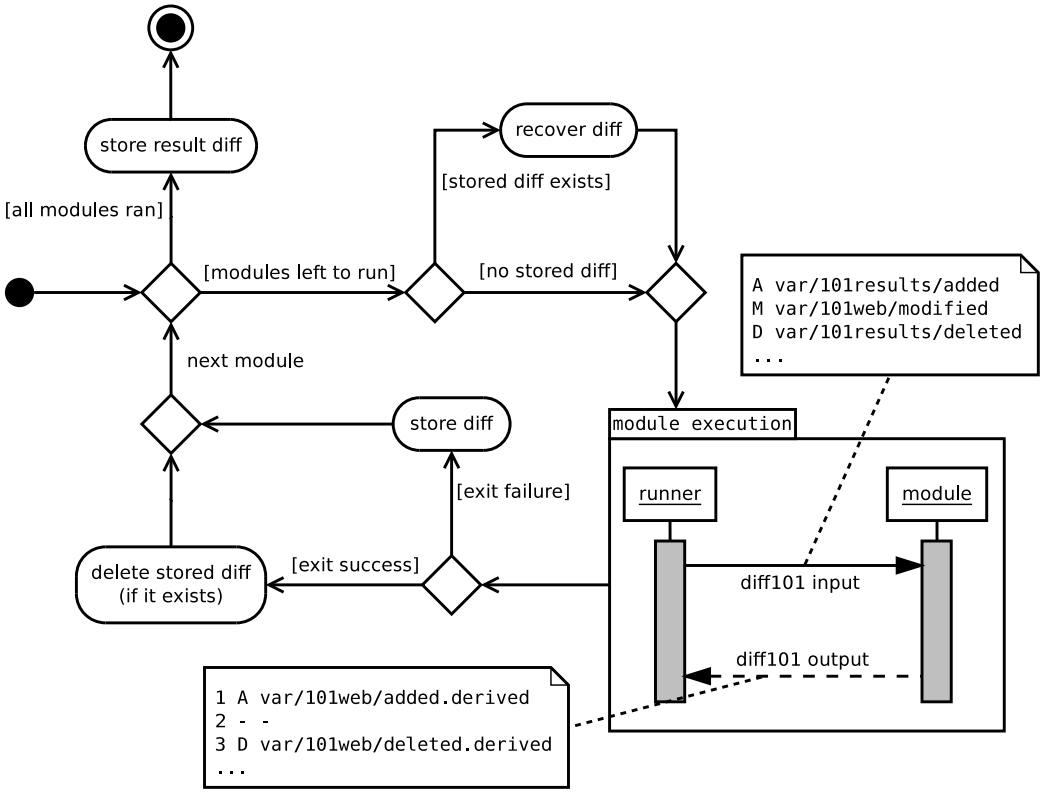

Figure 4.2: module execution workflow

4.3.1 Workflow

Whenever a 101worker cycle begins, the runner will start out with an empty diff. Each module receives the current (or recovered, see Section 4.3.4) diff as an input and, as they create, modify and delete files, produce their own diff output, which in turn becomes part of the current diff.

In practice, the pull101repo module[5], which uses Git to retrieve 101repo, generates the initial diff of primary resources simply from what Git provides it. Following modules discover changes via this diff input, derive secondary resources and in turn communicate a diff for each of them. Further following modules discover the change in the secondary resources, which prompts them to derive tertiary resources from them and again communicate the resulting diff. The process

[5] https://github.com/101companies/101worker/tree/master/modules/pull101repo

continues for resources of higher order.

See figure 4.2 and the following Sections for a detailed look at the 101diff protocol.

4.3.2 Grammar

```
input      = { inputline, newline};
output     = {outputline, newline};

inputline  =            op, filename;
outputline = linesread, op, filename
           | linesread, "-", "-"
           | garbage;

linesread  = digit, {digit};
op         = "A" | "M" | "D";
filename   = any;
garbage    = any;

any        = ?.+?;     (* for the sake of clarity, *)
digit      = ?\d?;     (* regular expressions are  *)
newline    = ?\n?;     (* used in this section     *)
```

Listing 4.4: 101diff grammar

Modules receive input in the form of a diff operation (*A* for added, *M* for modified and *D* for deleted) and an absolute file path, separated by whitespace, line-by-line.

Output being produced by modules contains an additional number at the beginning, also separated by whitespace, that represents the lines read from its input so far. A module may also just communicate how many lines of input it has read by using two hyphens in place of the diff operation and file path. This feature may be used for recovery purposes in the future (see Section 6.2).

Any output that cannot be parsed is simply treated as garbage and ignored – see the next Section 4.3.3 for details.

See listing 4.4 for an EBNF [ebn96] description of the protocol's grammar.

4.3.3 Inter-Process Communication

Communication between modules and the runner is handled via pipes: modules receive the change information via standard input and deliver their results via standard output. The concerns with this solution raised in Section 3.5.2 have been alleviated.

The issue with modules writing noisy logging information to their standard output has been solved by simply ignoring lines that cannot be parsed. This behavior has been inspired by how the Test Anything Protocol solves the same problem [SL06]. While there is a theoretical risk of module logs clashing with the 101diff grammar, none of the module logs in the past have shown such behavior, ruling out the issue occurring in production.

The deadlock problem has been solved by calculating all input data in advance and collecting all output data in a buffer before processing it. While this requires additional memory to store the data, it allows use of existing technology[6] to securely communicate all data without waiting for certain output.

As a last resort measure against modules locking up for any reason, a timeout has been implemented. If the execution time of any single module exceeds one hour, its process is terminated by force. As of the time of writing however, there has been no instance of such behavior outside of test cases.

4.3.4 Error Recovery

If a module exits unsuccessfully, the current diff is stored on disk by the runner. Whenever the runner finishes its run, it will also store the resulting diff on disk.

On the next run, the stored diff will be discovered by the runner. The recovery process involves merging the stored diff, the last result diff and the current diff together, so that the module input accurately reflects the actual changes that happened since it last ran successfully (see table 4.1).

If the module fails again, the merged diff is stored once more, overwriting the previously stored one, and the cycle begins anew. If the module succeeds, the stored diff is deleted and the runner will not attempt to recover anything on the next run.

See also the storing and recovering actions in figure 4.2.

[6] https://metacpan.org/pod/IPC::Run

older diff	newer diff	merge result
added	modified	added
added	deleted	no entry
modified	modified	modified
modified	deleted	deleted
deleted	added	modified
added	added	added
modified	added	modified
deleted	modified	modified
deleted	deleted	deleted

Table 4.1: rules for merging diffs

4.3.5 Library Support

Usage of incrementality features has been separated into a collection of libraries (see figure 4.3) [BMIM98, p. 162]. Modules use these libraries to operate only on the abstract concept of resources and are agnostic to the details of 101diff or the file system. Instead, they only focus on declaring and deriving resources.

incremental101

The incremental101 library presents a low-level access to the 101diff protocol. At its core, it handles parsing input, writing output and keeping track of the number of input lines read. The interface of the library provides functionality for iterating over the input, as well as writing and deleting resources.

Issues, such as creating the necessary folders and ensuring that files being deleted actually exist, are abstracted away for the user.

Additionally, if a file to be written already exists, it is only modified if the data to be written is actually different to the existing data on disk. This prevents creation of superfluous diff output, which prevents unnecessary work from following modules that derive from the created resource.

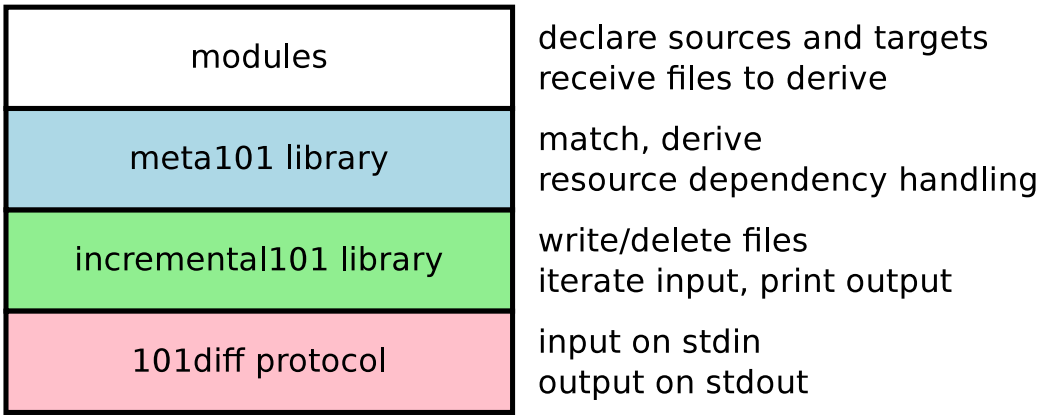

Figure 4.3: layers of abstraction

meta101

This represents a re-write of the original 101meta library built on the incremental101 library (see figure 4.3). It provides a high-level interface for modules to access derivation of resources: modules declare which resources they derive from, which resources they produce and provide callback functions that perform the actual derivation. The library automatically handles input and output from this data, as well as resolving changes in all the required resources to a single call into the module.

As per the escape requirement described in Section 3.5.5, meta101 can forego incrementality in favor of just walking the entire 101repo and re-matching or re-deriving all files. This is necessary if changes in module internals or 101meta rules invalidate the already derived resources. In line with the same requirements, deletions in the 101diff input are *always* processed, to properly remove resources that were derived from now nonexistent sources.

The escape is either triggered by the library itself, when it detects a 101meta rules change, or explicitly by modules. For example, the predicates101meta[7] will re-match the entire repository if it detects that its own executable or any predicate executable has been modified since the last run of 101worker.

This behavior is probably overly cautious, as even minute changes like changing the formatting of the source code will trigger the walk over the entire reposi-

[7] https://github.com/101companies/101worker/tree/master/modules/predicates101meta

	initial run	**subsequent run**
no incrementality	30 minutes	30 minutes
with incrementality	30 minutes	*1 minute*

Table 4.2: performance of 101worker (approximate)

tory. However, this is mitigated through incremental101, which does not re-write identical resources, as described in Section 4.3.5. This prevents a ripple-effect of all following modules re-handling unchanged resources and the unnecessary work is limited to the single, changed module.

4.3.6 Performance Analysis

An initial production run of 101worker after being deployed, without any already-derived data being present on the system, takes approximately 30 minutes. This time has, as expected, not improved through the changes described above.

However, any subsequent runs are able to make use of the data derived in the previous runs. While before, runs in which no changes occurred took the same time as the initial runs, with the newly implemented incrementality features they only take approximately *one minute*.

These thirty-fold performance improvements scale with the amount of changes since the last run, rather than with the size of 101repo. They also enable faster testing of module changes in production and quicker analysis of newly added contributions.

Chapter 5

Related Work

The bus system of 101worker is similar to classical build systems of compiled languages. For example, for building a program written in the C language, a compiler derives object files from source code, from which in turn a linker derives executables or library files [SMS06].

As such, similar issues regarding performance (see Section 2.2.3) and accumulation of garbage (see Section 2.2.4) apply: the compiler should only re-compile code that has changed since the last run, and object files derived from source files that have been removed should be removed as well.

While build systems such as Make [SMS06] or CMake [MH10] are able to detect necessary re-compilation, they require an explicit "clean" step to remove the derived files. Omitting this step can lead to strange errors caused by those left-over, garbage object files. The incrementality features of the refactored 101worker (see Section 4.3) would be applicable to handle automatic cleaning of these kinds of files.

Chapter 6

Concluding Remarks

6.1 Summary

In the process of this thesis, 101worker was successfully refactored into a more scalable and maintainable system. It has also become more cohesive and comprehensive, aiding future developers in gaining an understanding of the system more easily.

Issues with performance and old data accumulating has been solved by making the worker aware of changes and act on them incrementally. Documentation, installation and testing guidelines have been established, duplication of information has been removed in favor of centralized environment variables and the most important modules have been reworked to make use of the new incrementality features and libraries.

Libraries and modules that do not yet conform to the worker's incremental behavior have been removed, deprecated or are set to be refactored in the near future.

6.2 Limitations

Recovery on module failure is relatively limited due to modules having to maintain data dumps [Lei13, p. 24]. These dumps contain a summary of the module's results and need to be kept synchronized as files are added, modified and deleted. The dump files are only written when a module finishes its run.

If a module fails somewhere during its run, the dump is not written. When recovering from failure on the next run, the module will have to re-derive files that it already successfully derived in the last, failed run so that information from those files is not missing from the dump. If this were not the case, the module could re-start from the file it could not process instead.

To solve this, the dump could be written after each line has been processed. However, this would create a disproportionately large overhead from encoding and writing a large JSON file[1] every single time a resource is touched.

Alternatively, the dumps could be gathered from separate files after a module has finished its run, although it is questionable if this would actually result in a notable performance gain.

At the time of writing, however, no modules have exhibited spurious failure after processing a part of their input. Due to this lack of practical relevance, the solution has been put off until the issue actually manifests itself.

6.3 Future Work

Much of the refactoring done in this thesis paves the way for further improvement of 101worker. While the modules that were deemed most important to the knowledge engineering task of the system or had the largest performance impact have been reworked to act incrementally, some others could still benefit from such a refactoring[2].

Most modules also still lack comprehensive documentation, installation requirements, unit tests and functional tests. If possible, the original authors will be contacted and asked to provide at least parts of these.

[1] up to 1.5 Megabytes for a single dump at the time of writing

[2] Status of modules: https://github.com/101companies/101docs/blob/master/worker/TODO.md#refactor-modules

Bibliography

[10114a] 101companies project description. http://101companies.org/wiki/@project, 2014. Accessed on 2015-04-19.

[10114b] 101repo description. http://101companies.org/wiki/@repo, 2014. Accessed on 2015-04-19.

[Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[BMIM98] William J. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley Computer Publishing, 1998.

[ebn96] *ISO/IEC 14977:1996 Extended BNF*. International Organization for Standardization, 1996.

[FLL+] Jean-Marie Favre, Ralf Lämmel, Martin Leinberger, Thomas Schmorleiz, and Andrei Varanovich. Linking Documentation and Source Code in a Software Chrestomathy. In *Proceedings of WCRE 2012*. IEEE. 10 pages.

[FLSV12] Jean-Marie Favre, Ralf Lämmel, Thomas Schmorleiz, and Andrei Varanovich. *101companies*: a community project on software technologies and software languages. In *Proceedings of TOOLS 2012*, LNCS. Springer, 2012. 16 pages. To appear.

[jso13] *ECMA 404: The JSON Data Interchange Format*. ECMA International, 2013.

[KLL+13] Kevin Klein, Ralf Lämmel, Martin Leinberger, Thomas Schmorleiz, and Andrei Varanovich. A Linked Data approach to surfacing a software chrestomathy. 20 pages. Submitted for publication. Available online since 21 June 2013., 2013.

[Lei13] Martin Leinberger. Enhancement of a software chrestomathy for open linked data. Master's thesis, July 2013.

[LLSV14] Ralf Lämmel, Martin Leinberger, Thomas Schmorleiz, and Andrei Varanovich. Comparison of Feature Implementations across Languages, Technologies, and Styles. In *Proc. of IEEE CSMR-WCRE 2014*. IEEE, 2014. 5 pages.

[LM12] Jon Loeliger and Matthew McCullough. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development, 2nd Edition*. O'Reilly Media, Inc., 2012.

[MBDC+06] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jerome Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the complexity of large free and open source package-based software distributions. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 199–208. IEEE, 2006.

[MH10] Ken Martin and Bill Hoffman. *Mastering CMake*. Kitware, 2010.

[Ray03] Eric S. Raymond. *The Art of UNIX Programming*. Pearson Education, 2003.

[SL06] Michael G. Schwern and Andy Lester. Test Anything Protocol Specification. https://testanything.org/tap-specification.html, 2006. Accessed on 2015-04-21.

[SMS06] Richard M Stallman, Roland McGrath, and Paul D Smith. Gnu make manual. *Free Software Foundation*, 3, 2006.

[Ste99] W. Richard Stevens. *UNIX Network Programming Volume 2 Second Edition: Interprocess Communications*. Prentice Hall, Upper Saddle River, NJ, USA, 1999.

[Wik15] Wikipedia, the free encyclopedia. List of software package management systems. http://en.wikipedia.org/wiki/List_of_software_package_management_systems, 2015. Accessed on 2015-04-21.

[yam11] YAML: YAML ain't markup language. http://yaml.org/, 2011. Accessed on 2015-04-19.