Loughborough University

Computer Science

18COC251

B627091

Alembicue: A projectional editor & IDE for container application architecture

George Garside

Supervisor: Dr. Daniel Reidenbach

Loughborough University

Summer 2019

Abstract

The lack of a competent integrated development environment for Docker is a detriment to the field of containerisation and cloud computing. This project documents the development of **Alembicue**, a projectional and productive Docker IDE, implementing a more formally specified language definition, representative of what is necessary to create images for use with a containerisation platform. Alembicue guarantees syntactic correctness with a projectional editor mutating an abstract syntax tree, incorporating contextual intentions and quick fixes to suggest and automatically apply standards and best practices. Alembicue's launch has been a success, being received well by the community with tens of thousands of interested visitors to the project page, and elated testimonials from those having used the application and the innovative concepts portrayed within.

Acknowledgements

I would like to thank my supervisor, Dr. Daniel Reidenbach, for his consistent support and valuable feedback on my work. I would also like to thank my family and friends for their support in my studies.

alembicate, *v.* 1627. transitive. Chiefly figurative: to produce, refine, or transform (an idea, emotion, etc.) as if in an alembic. Cf. ALEMBICATED adj.

cue, *n.*[2] 1553. 1c. A stimulus or signal to perception, articulation, or other physiological response.

Oxford English Dictionary (1989).

alembicue, *n.* 2019. Chiefly emblematic: the juncture of the production of imagery and the container articulation process founded on the aforesaid.

With apologies to linguists.

Contents

Figures

Tables

Part I:

Foundation

This first part provides an overview to the project, beginning with its motivation, objectives and deliverables (chapter 1), followed by a review of the existing literature on this topic (chapter 2). Background information (chapter 3) is provided on the topics of containerisation, on which the language will be developed, and a comparison of language workbenches, which will be used for development of the language. The gap analysis (chapter 4) identifies issues with existing tooling with the same containerisation objectives, and discusses areas for innovation above these tools. Finally, a summary of tooling and languages chosen for use in the project provides an overview of the development environment (chapter 5).

1 Introduction

1.1 Overview

The aim of the project was to design, develop and release Alembicue, an integrated development environment for composing Docker images, centred around a projectional editor for Dockerfiles.

Gap Analysis (chapter 4)

The lack of a comprehensive and integrated development environment for Docker causes issues for both developer adoption of the containerisation platform and the implementation of advanced platform functionality for professionals, as identified in gap analysis. Appropriate in-depth research of new and old editor types and functionality was necessary to establish a foundation for design with an understanding of the context of the solution.

Literature Review (chapter 2)

Background (chapter 3)

Design (chapter 6)

Development was led by the choice of applicable tooling and languages including that of a language workbench forming the centre of defining and implementing a language specification. This language was derived to represent the image build language used by Docker's image build process, while benefiting from the advantages of projectional and structural editing identified in the lessons learned from literature review. Incorporating this editor alongside orchestration mechanics to run containers based on the built images integrates an end-to-end workflow for containerisation of services, streamlining the process for novices and experts alike.

Tools (chapter 5)

Background: Language workbenches (section 3.2)

Integrated development environment (Part III)

This report documents the processes involved in reaching this aim, through the completion of various objectives.

Objectives (section 1.4)

1.2 Motivation

Docker is used by more than 7 million developers worldwide (Evans Data Corp., 2018; Stack Overflow, 2019), in use by over half of all developers (Stack Overflow, 2019) and 92% of organisations (JetBrains s.r.o., 2018a). Despite this, the field is lacking a competent integrated development environment for Docker. It is left to the combined power of various tools augmenting developer knowledge to create and maintain Docker images and containers (section 3.1). These tools can be difficult to set up and use requiring a full containerisation platform installation, and may still only provide varying levels of accuracy in linting (code checking) of syntax, style and references.

These existing tools and their issues identified are covered in detail in the gap analysis (chapter 4).

A primary source of difficulty in building images for containerisation is the Dockerfile language. A Dockerfile is written by a containerisation developer and contains instructions used for the production of a Docker image, each of which creates a new build layer for the image. Docker does not provide a formal specification for the language, only officially providing a Dockerfile reference (Docker, 2018c) containing a brief description of layout, keywords and

instruction parameters. To be able to write a Dockerfile, it is necessary to understand this document, which can be difficult due to the informal nature of the specification. This leads to inaccuracies and misinterpretations both in developers using the specification to write their Dockerfiles and in implementations of linting in existing tools aiming to assist developers.

This difficulty presents issues for novices and experts alike. According to the Stack Overflow Developer Survey 2019, Docker is the platform with the most interest by far from developers who are not currently developing using it (Stack Overflow, 2019). For developers looking to begin using Docker, the Dockerfile language reference can be difficult to interpret due to its informal nature. Lacking a rigid and consistent structure to the parameters and syntax used to distinguish types, it can be very easy to make a syntax error, which is only presented when the lengthy build process takes place. Furthermore, Docker is not just for development novices, with Docker ranking second in most loved platforms for development in the aforementioned survey (Stack Overflow, 2019). For experts, the advanced functionality of the Dockerfile is under-represented by tooling, yet this is the area that needs the most assistance due to its complexity. For example, multi-stage builds require careful referencing in instructions crossing stages, yet issues in tooling support and ambiguity in usage contribute to 22% of developers dreading the Docker platform (Stack Overflow, 2019).

The build process mentioned here used by Docker for the creation of an image from a Dockerfile is discussed further in the background chapter of this report.

Defining an understandable and consistent specification for the Dockerfile language, alongside an innovative editor for the language ensuring syntax compliance and providing contextual assistance in development, as part of an integrated development environment capable of supporting an entire containerisation workflow, would help novices and experts alike make the best use of the capabilities of Docker for their containerisation needs.

1.3 Benefits

The projectional editor, the core of Alembicue, edits an abstract syntax tree (AST) which directly represents the structure of the document and its content, instead of the usual text document editing, but with an interface similar to that of a standard IDE for text-based programming languages. By defining a structure of the language and behaviours for editing instances of the language, the editor enforces a requirements for properties, references and children, assisting with code completion and providing instant compliance checking for type system rules.

Since the projection in the editor is a precise and faithful reflection of the AST enforcing constraints defined in the language, it is not possible to enter invalid syntax prior to ‘parsing’ as with regular text files/editors, as there is no parsing step. For example, in a projectional editor, typing with an insertion point at a location which cannot possibly produce a valid tree with the entered characters

will have no effect, as the characters are prevented from being entered by having no suitable destination.

In addition to enforcing structure and constraints based on the language, since this editor is not limited to text, the AST can be projected in alternative ways which may make understanding relationships easier. Multiple projections of nodes can be used to represent references, making parts of the file available in other locations and editable from either. Presenting the referenced node to the user ensures correct use of references, confirming that correct destinations of references are selected in the editor and by extension in the tree, and encouraging quick fixes to be performed on the secondary projection of the node, improving editing efficiency.

Containerisation is an area of active research and development, with implementations being iterated upon and new versions of the containerisation platform being released occasionally. Keywords and parameters which may have been valid for a previous version can be deprecated, and limited assistance provided in migrating to newer file versions. By clearly marking deprecated keys as such in Alembicue, the developer can see the problem immediately and fix the issue before further issues occur later on. Issues can be resolved easily by providing intentions for automatically migrating the deprecated code away from old concepts to their replacement with equivalent functionality.

1.4 Objectives

For the successful completion of this project to its aim, a number of objectives were delineated.

1. Establish prerequisites to the project, including its motivation, methodology and work plan to ensure the project stays on track.
2. Research and review existing literature in the field for a greater understanding of the work which has taken place prior, providing lessons learned to be built upon.
3. Research, discuss and present conclusions, for an overview of containerisation necessary for the development of the containerisation language, and language workbenches for the implementation of the language.
4. Establish weaknesses in existing tooling for containerisation, and areas for innovation in Alembicue, through a detailed gap analysis.
5. Choose and set up tooling to begin development, including version control and continuous integration.
6. Design and implement a base level of the language, with core concepts such as the abstract syntax tree, and extensibility for continued development.

7. Design, develop and implement instructions for the language, including projectional editor components for each instruction, type system constraints and behaviours.

Each of these terms are introduced in Design (chapter 6).

8. Develop and implement an integrated development environment, incorporating the projectional editor environment and Docker orchestration to run the language.
9. Perform suitable testing of each component of Alembicue and the overall application using established practices and methods.
10. Release Alembicue for download and use by developers to write their own containerisation files.
11. Evaluate the outcome of the project against each objective and the project's overall aim, including feedback received from Alembicue's release and possible future work to be carried out.

1.4.1 Deliverables

Following that which is established in project prerequisite analysis, a number of deliverables are decided upon.

1. A simple projectional editor is the first deliverable of the project, as this is the core of Alembicue. The ability to add instructions, representing lines of code which each perform a particular task (section 7.1), is made possible with the implementation of a type system, and various supporting concepts including key-value pairs, and ports and protocols.
2. The Alembicue language is defined and implemented, containing all instructions each with parameters, children and references applicable for the task the instruction needs to perform.
3. Intentions, type system messages and refactoring is delivered for instructions where appropriate to make the editing experience reflect expectations of familiar text editors, as well as providing advanced functionality only possible with a projectional editor.
4. Testing ensures functionality of Alembicue is correct, meeting the specification of the language and ensuring the usability of the software's interface.
5. The standalone Alembicue application is compiled for various platforms, including the Alembicue language.
6. A project webpage is delivered to provide a location for users to download Alembicue.

1.4.2 Deliverable constraints

Constraints apply to the project, its objectives and deliverables, providing additional boundaries and guidance to the project's development.

2. **Scope**: The language is comprised of instructions, each of which needs to be incorporated into Alembicue's editor. These instructions are `FROM`, `RUN`, `CMD`, `LABEL`, `MAINTAINER`, `EXPOSE`, `ENV`, `ADD`, `COPY`, `ENTRYPOINT`, `VOLUME`, `USER`, `WORKDIR`, `ARG`, `ONBUILD`, `STOPSIGNAL`, and `SHELL`.

4. **Scope**: Testing requires manual testing through test plans created during development of functionality, automated unit testing for individual components and integration testing for the composition of modules.

5. **Scope**: Alembicue as a standalone application must support macOS, Windows and Linux to provide the greatest coverage to the developer user base.

6. **Schedule**: The project, including its release, must be completed prior to the deadline for the project as a whole, to be able to discuss and evaluate the release as part of the project's report.

1.5 Methodology

Methodologies for development were decided upon early in the project which ensured all development and work product adhered to such standards.

1.5.1 Formal methodologies

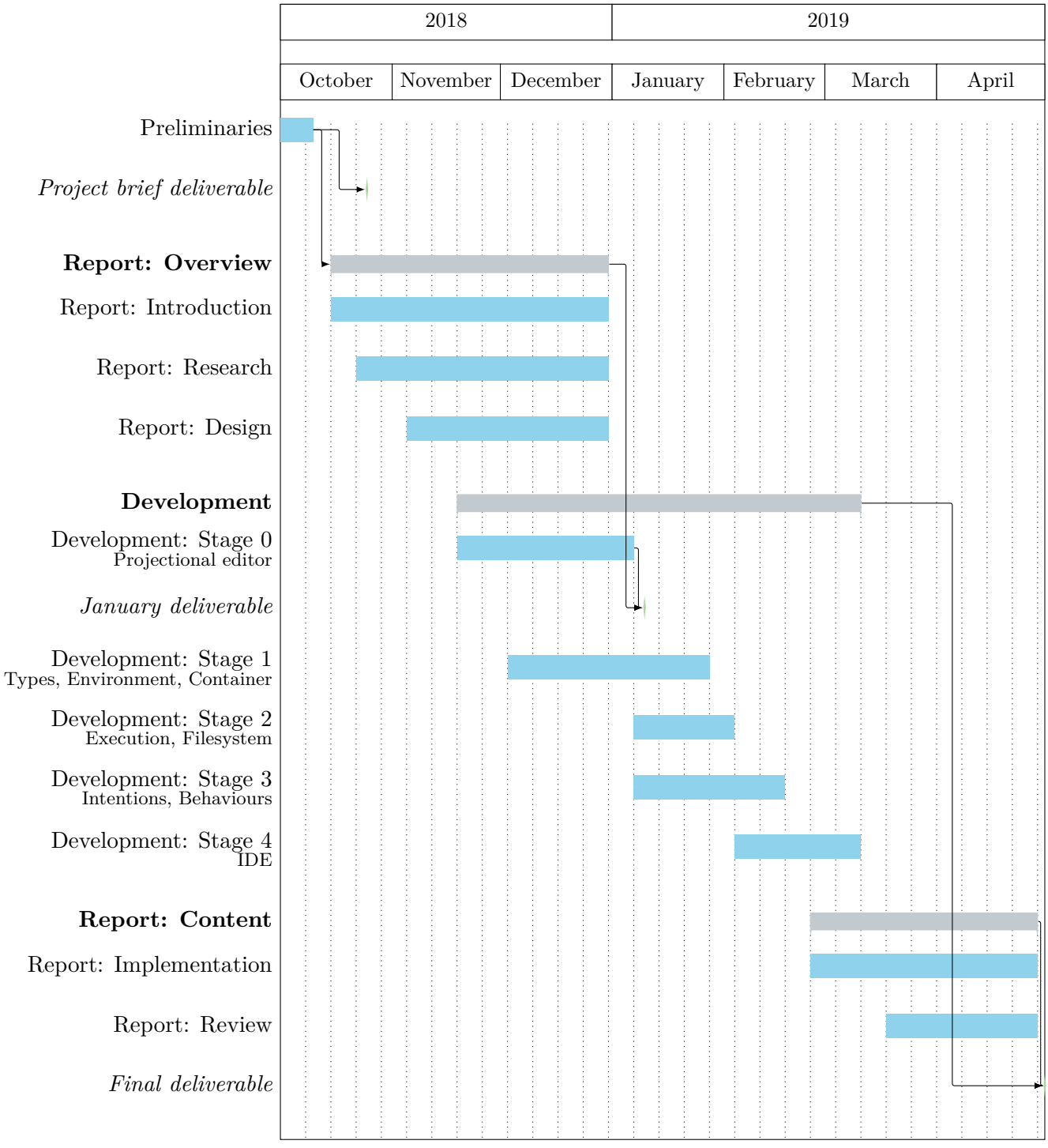
- Throw-away prototyping was used for the early stages of the user interface and basic functionality to quickly experiment and test different ideas and possibilities. This includes prototyping the structure of the underlying language components and of the user interface components used to display the language. While the learning that took place from the prototyping efforts was retained for main development, the work product was not.

- Incremental building was used for the main development of the project, including the language definitions and implementation of the editor. The language was broken down into sets of functionality which could be implemented in any order, providing each prerequisite is met for each subsequent definition. From a high level, these sets were rigidly defined with a strict set of capabilities which needed to be implemented to meet the step. Such implementation was refined and components moved between stages as development progressed due to gaining a better understanding of the structure of the development and of the prerequisites needed for implementing various language and editor components.

1.5.2 Work plan

A work plan was developed at the start of the project and improved over the course of the design and development stages, as well as the writing of the project report. This work plan was presented in a Gantt chart (Figure 1.1) to provide an easy to understand overview of timescales for various components of the project. Annotated in the chart are the three milestones of the project: project brief, January deliverable and final deliverable. These helped define deadlines for individual comments necessary to make up those deliverables, which were included in the Gantt chart and shown by the 'Finish to Start' dependency relationship connection lines.

Figure 1.1
Project Gantt chart



2 Literature Review

There exists extensive literature on the topic of cloud computing, but this project only relates to containerisation in terms of the language used for image construction. Therefore, material relating to cloud computing, containerisation, orchestration, or related areas are not covered as part of the literature review. Instead, focus is placed on source code editing paradigms and representations.

2.1 Plain text editing

In 1975, `ed` and `vi` was the beginning of text editors which provided simple functionality suitable for editing innumerable file types (Performance Computing, 1984). As discussed in the introduction to O'Reilly's 'Learning the `vi` Editor', such editors needed to be broadly applicable 'whether those files contain data, source code, or sentences' (Lamb and Robbins, 1998). These original editors are thought to be 'unintuitive and cumbersome' for beginners, in part due to distinct modes of operation: insert and command (Lamb and Robbins, 1998).

Discussion has also included other plain text editing paradigms which were developed in a similar era. This includes Sed, a non-interactive text editor based on the line-based editing of the interactive `ed`. Amongst the criticisms shown of Sed by McMahon (1978) is the lack of 'immediate verification' compared to an interactive editor. Conversely, 'complicated editing scripts [...] saves considerable typing [and] attendant errors'. In this project, combining both characteristics provides the optimal experience for the user.

Returning to interactive text editing, studies have shown that interactive line editors can also be difficult to use. The nature of line editors means that text manipulation is regularly performed on text not displayed on the computer screen. Gomez (1988) found that users must rely on spacial memory to understand the context to their commands in such an environment. Such studies performing tests by providing a copy of the text printed beside the user does somewhat diminish the ability to assess how difficult using a line editor is when needing to keep the output in mind. However, having to look down to the piece of paper also has its own negative effect on editing speed. A follow-up study by Gomez et al. (1983) showed that a screen editor can halve the time taken for the same tasks to be completed. In this project, the ability to immediately see the effect of operations on the document is essential in reducing cognitive load on the user.

Hunt and Thomas (1999, p.73) have discussed how plain text files can contain structure, with the structure being identified by characters in the file visible to the user. Plain text editors require that 'you must type in codes that are used by [another] program' (Lamb and Robbins, 1998) such as a compiler. Advances in plain text editors can incorporate aspects of structure parsing for functionality

aiming to assist the user with text editing. In 1998, `Vim` added syntax highlighting to its 1996 graphical user interface (Moolenaar, 1998) which helps distinguish 'different mnemonics [and] program structures' to the user (Yuen et al., 1997). This was discussed to be assistive in debugging and visualisation of program flow.

The core developments in plain text editing are widely implemented in editors and OSs alike, such as keyboard shortcut standards (Apple, 2018), and 'friendly editors' incorporating colour syntax highlighting into 2000's nano in 2008 (GNU, 2008). It is therefore wise for this project to follow such standards, while circumventing problems with rudimentary editors.

2.2 Structure editing

The term 'structure editor' refers to text or component editing software with syntax understanding, compositing the source code in the form of a syntax tree. In the presentation of Collberg (2009), the focus of structure editing is its benefit to the compiler, 'relieving [it] of lexing & parsing'. The benefit to the compiler requires a connection between compiler and editor, not expected between a text editor able to edit any text file and the compiler for the specific language being edited. Sufrin (1982) has explored the concept of formalising a text editor's design, but even admit that their 'choice of document model' caused 'an enormous number of special cases'. This project specifically aims to avoid such issues from the outset by ensuring the underlying structure remains expandable as the language develops, while also providing compilable files in their standard format to be parsed as normal by the existing compiler.

As well as internal storage considerations, structure editors need to provide an editing interface. Instantiating a tree from source code as part of the editing workflow, separate from the compiler, was discussed in a paper by Koorn (1992) regarding the development of 'GSE: a generic text and structure editor'. Where a compiler may create a tree from the source code with text parts, a structure editor must associate additional information with nodes on the tree for the tree to be useful in an opposite direction: from tree to source code. It is therefore necessary to 'store position information as an annotation in nodes on the tree' (Koorn, 1992, p.4). This is a crucial step in development towards projectional editing, but as noted by Koorn (1992) something which is not ideal for the long-term development of editors, as this is extraneous information if the original representation does not need to be referred to again.

Sufrin (1982) concluded the 'further work' section of their paper noting that an extension to the existing 'structure-oriented editors' would be for 'operations [to be] tree-oriented' as 'the "document" is now a tree'. Other work in this area includes a study by Ko, Aung and Myers (2005) in which it was found that most edits made by Java programmers preserved structure in the document. By

considering the document as a tree and doing away with unstructured text, 'more sophisticated support' can be provided alongside the lack of 'omissions of delimiters'. Following design strategies identified in the study such as a 'top-down interaction technique for every edit', traditional structure editors increase the number of steps required to perform tasks. This project aims to retain the 'interaction techniques that preserve structure', while exploring new techniques to provide flexibility not traditionally available.

It is widely known that structure edits to unstructured source code 'fail in the presence of syntax ambiguities' (Ko, Aung and Myers, 2005). Research into the benefits of structure editors notes that by completely removing the need for parsing, syntax errors are prevented. Such editing environments include the Cornell Program Synthesizer (Teitelbaum and Reps, 1981).

2.3 Projectional editing

Besides the mentioned benefits of structure editors, new visualisations of source code are possible based on the structure of the document. The term 'projectional editor' was coined by Fowler (2005), and popularised later also by Fowler (2008), defining it as an editor which 'manipulates the abstract representation and projects multiple editable representations for the programmer to change the definition of the system'. This briefly describes the characteristics of the first deliverable in this project, namely describing the behaviour of the software which will form the core of the editor.

abstract representation The depiction of components of the language defined and developed by its concepts and aspects rather than characters used in a plain text document. The text regions, tables and grids provide a visualisation of the document's data.

multiple representations The projection of references from the abstract syntax tree into the editor using editor components for the destination of the reference to be edited.

editable system definition The ability to define the behaviour of the image through steps contained in the document.

The earliest reference to an editor which edits a tree directly is a patent 'Hierarchical structure editor for web sites' granted to Rae Technology Inc. (Rae Technology Inc., 1996), a spin-off of Apple Computer in 1992 (Knibbe, 1994). This patent appears to be with regard to a very different editing interface, featuring graphical interactions with a tree similar in appearance to a flowchart. This flowchart interface for high level editing can provide a useful user experience, and may be useful for a container orchestration interface, but does not provide the granularity needed for Alembicue's modification of the steps within an image definition.

3 Background

As mentioned in the context of literature covered in the literature review, neither cloud computing nor containerisation are foci of this project. The focus of this project is the language used by Docker, a popular implementation of a containerisation platform. Therefore this report heavily covers the Dockerfile language as necessary for its implementation in Alembicue. Some background on Docker's containerisation process can provide context to the language defined in this report and incorporated into Alembicue, therefore content relevant to Alembicue on the topic of containerisation is covered in this section.

3.1 Containerisation

The notion of containerisation is to package an application with all its dependencies, to be run standalone on a machine like a virtual machine instance, but in user space sharing the host kernel. Containers have 'their own network and storage stacks, as well as resource management capabilities' (Turnbull, 2014), which makes them powerful for running isolated or controlled environments for development, testing, and production code alike.

3.1.1 Images & containers

An **image** is an immutable record of filesystem contents and execution parameters surrounded by a collection of metadata. The filesystem contents is recorded as a list of changes to the filesystem from instructions given as part of the image build process (subsection 3.1.2). Images are created from the build process which uses a file of instructions to save an image to disk, along with the **context** in which the build took place which containing files and resources used in the build.

A **container** can be thought of as an instance of image. Created from running an image, it is a standalone package containing a runtime, filesystem with system libraries and tools, and executable code (Docker, 2018b).

Alembicue focuses on the development of images with its editor for creating files used by the image build process. Alembicue also integrates with Docker to be able to create containers from such images, providing end-to-end support for a container-based development workflow.

3.1.2 Image build process

Docker images are created by parsing a Dockerfile containing instructions on how to build an image from a context, with instructions defining each layer of the build process. These layers can configure the environment, run commands, execute binaries, and modify the filesystem, creating an immutable history of the changes made to the image, which then becomes the start point of the subsequent creation of a container (subsection 3.1.1). The parsing is performed by Moby's

Dockerfile builder, which combines reading and parsing of the Dockerfile, as well as construction of an image, into one process (Nephin et al., 2019).

The essence of the domain-specific language is that each non-empty line is an instruction, beginning with a keyword (including comments where the keyword is #) to define the operation for that step of the build and continuing with required or optional parameters to modify behaviour.

Each instruction begins a new layer in the underlying image.

In the example shown (Figure 3.1), the first line begins with a keyword **FROM**, which per the informal specification for the language, is to be followed with a sans-whitespace string identifying the image with name (scratch) and optional namespace. This is followed with an optional parameter, denoted by the keyword **AS** and continuing with a string (orig) to refer to the container later in the file, a reference to the container used for running commands between this and a subsequent **FROM** instruction.

A full description of the **FROM** instruction is given in Environment configuration (section 8.1).

```
FROM scratch AS orig
COPY hello /
CMD ["/hello"]
```

Figure 3.1
Example 'binary run' instructions

3.2 Language workbenches

A language workbench is software assisting with language engineering by providing the ability to develop new languages in a standardised manner with tooling support. A wide array of language workbench tools exist, focusing on and providing enhancement to various stages of the language creation process. Wildly different methodologies and implementations of both language development and consumption can enhance or limit flexibility in crucial ways.

There were a number of early language workbench implementations in the late 1980s and early 1990s. These were rudimentary for creating languages powerful enough for the time.

- SEM (Teichroew et al., 1980)
- Metaview (Sorenson, et al., 1988)
- MetaPlex (Chen and Nunamaker, 1989)
- MetaEdit (Smolander et al., 1991)

By the late 1990s, graphical workbenches were developed, but the languages themselves were still focused on the necessities of the time — the workbenches focused on this, providing key functionality in support of simple customisation.

- MetaEdit+ (Kelly, Lyytinen and Rossi, 1996)
- GME (Ledeczi et al., 2001)

For general-purpose programming languages, support of textual notations needed to be included in such language workbench software.

- Centaur (Borras et al., 1988)
- ASF+SDF Meta-Environment (Klint, 1993)
- LRC (Kuiper and Saraiva, 1998)
- GemMex (Anlauff, Kutter and Pierantonio, 1999)
- LISA (Mernik et al., 2002)

While general-purpose languages required development as much as domain-specific languages, the domain-specific nature of the latter necessitated user-friendly tools for their creation, such that they could be created with less experience of language engineering and at a faster pace than their general-purpose counterparts. This lead to an introduction of domain-specific language workbenches in the late 2000s and early 2010s.

- Rascal (Klint, Van Der Storm and Vinju, 2009)
- Spoofax (Kats and Visser, 2010)
- Eclipse Xtext (Eysholdt and Behrens, 2010)
- JastAdd (Söderberg and Hedin, 2011)

While general-purpose programming languages advance with increasingly user-friendly features, it becomes less necessary for domain-specific languages to be so different from the norm. Projectional language workbenches provide the different experience of a projectional editing environment (section 2.3) to domain-specific language creation and their subsequent use in production.

- JetBrains MPS (Voelter and Pech, 2012)
- Whole Platform (Solmi, 2017)

For this project, I focused on Eclipse Xtext (subsection 3.2.1), Whole Platform (subsection 3.2.2) and JetBrains MPS (subsection 3.2.3). The following sections cover these language workbenches, comparing their benefits and weaknesses relevant for this project. In comparing these software, the symbols ✔, ━, and ✘ are used to represent positives, concerns and considerations, and negatives respectively.

3.2.1 Eclipse Xtext

Based on the popular Eclipse IDE (White, 2014), Eclipse Xtext is a framework for developing a text-based language by providing a grammar to use for the language, and the workbench provides a parser with type checking (Efftinge and Koehnlein, 1972).

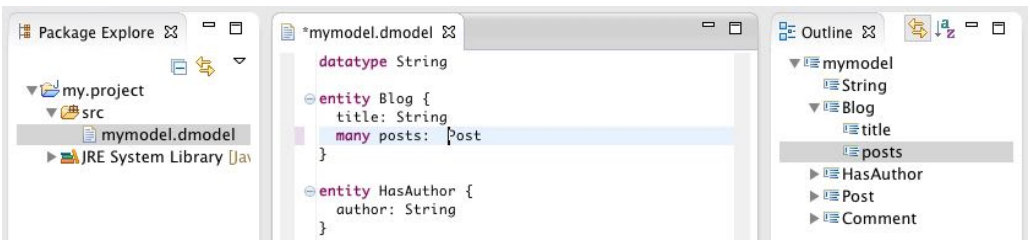


Figure 3.2
Xtext example in Eclipse IDE screenshot (Efftinge, 2015)

Advantages & Disadvantages

- ✔ Xtext is intended to be easy to learn and use, being a text-based language both to create the language and for the language to be created. One can use a grammar to directly create language and parser for that language, which can parse text files containing that language, all within the Eclipse environment.
- ✔ Languages created with Xtext are generally human readable, with the use of text files and writing keywords directly to file. This improves the ability for files to be accessible outside of the ecosystem of the Eclipse platform used to create the language.
- ━ Eclipse is no longer the most favoured development environment; according to Stack Overflow, other editors take preference far above Eclipse (Stack Overflow, 2018; 2019). Basing future development on an editor which is losing popularity may mean that such development is not favoured by the majority, where the benefits of using such a tool based on the platform must outweigh the negativities of using such a platform.
- ✖ While performance is one of the key desires for Xtext, the necessity for a parser to read the files slows down the workflow. This is extenuated by writing out human readable files potentially containing lengthy keywords for the purpose of making the file presentable in the editor.

3.2.2 Whole Platform

Another language workbench based on the Eclipse IDE framework is Whole Platform. While based on Eclipse, this language workbench is projectional (section 2.3), with the ability to for 'multiple kinds of notations [...] including grammar layouts [and] tree and graph layouts for diagram oriented languages' (Solmi, 2017).

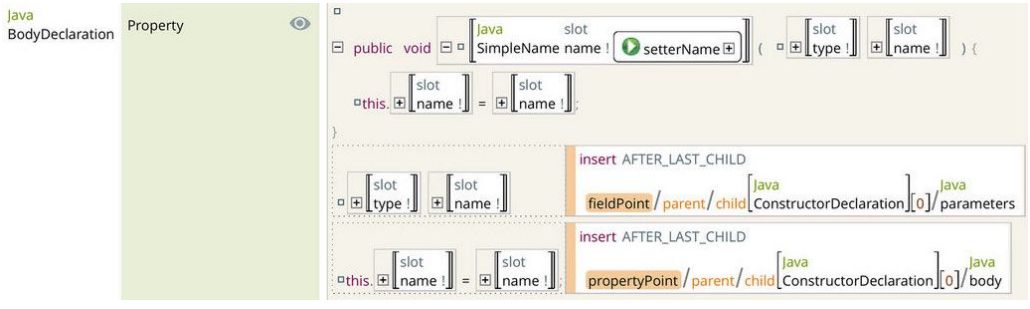


Figure 3.3
Whole Platform example projectional editor for construction of Java (Solmi, 2017)

Advantages & Disadvantages

- ✔ The projectional nature of Whole Platform provides the ability for 'multiple kinds of notations [...] including grammar layouts [and] tree and graph layouts for diagram oriented languages' (Solmi, 2017). These provide new visualisations to existing grammars, or the ability for more creative grammars which can utilise such representations of the underlying structure.

- Whole Platform provides a vast array of additional behaviour and functionality on top of the defined language, including 'stream based persistence and Java/XML serialisers (Solmi, 2011). This functionality, while useful for many languages which could be developed with this platform, shows that the platform has a focus which perhaps does not coincide with the aims of the Dockerfile language implemented in Alembicue. The Dockerfile language is intended to be rudimentary, with very simple syntax and basic instructions — such advanced functionality would perhaps complicate both development and use of the language beyond what would be necessary to provide an easy to use and modify grammar.

- Whole Platform focuses on executable code and languages which are designed for the purpose of generating executables. Defining a model in terms of its 'translation to another language which [have] executable semantics' (Solmi, 2011) is insufficient for a language such as which is implemented in Alembicue. This is due to the nature of the underlying Dockerfile language, which does not have roots in that of executable definitions.

3.2.3 JetBrains MPS

Also providing a projectional editor is JetBrains MPS, the Meta Programming System language workbench based on the IntelliJ platform. MPS focuses on converting 'domain processes and knowledge [to] a language that directly uses [those] concepts and logic' (JetBrains s.r.o., 2017).

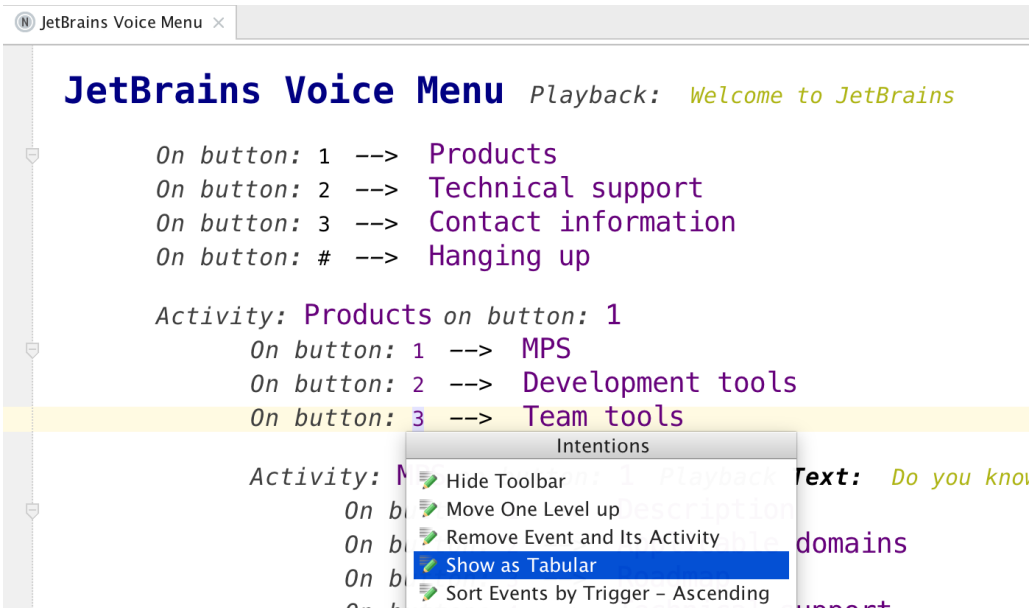


Figure 3.4
JetBrains MPS example projectional editor for voice menu (JetBrains s.r.o., 2018b)

Advantages & Disadvantages

- MPS idealises language-oriented programming, with an ability to provide direct relationships between the language created and domain-specific processes. MPS provides the greatest flexibility between desired concepts and output language, with options to extend `BaseLanguage` or start from scratch.

Flexibility can bring complexity — perusal of the documentation given by JetBrains and third parties over the MPS tool demonstrates the complexity of development. As MPS is a tool for developing languages, it is without surprise that numerous languages are used with MPS to create more languages. One requires a good understanding of various languages: as well as Java and the projectional version of Java[1] used in MPS, one needs to learn BaseLanguage and the SModel language, and various languages for the editor and other concepts. These can be thought of as two entirely new languages, alongside a projectional editor and tooling, and abstractions to existing languages, a good understanding of which is required for the success of this project and the development of Alembicue.

This additional learning would need to be factored into the time of development with MPS.

Choice Based on this investigation into these three language workbench tools, MPS was chosen as the tool to use for the creation of Alembicue. While concerns over the complexity of learning the tool should be mitigated by an appropriate project plan with sufficient time to learn such languages, the benefits of the flexibility of the language to be created provided by the tool produces a powerful projectional editor for the language atop a familiar and liked IDE[2].

[1] For example, Java would use `new ArrayList<String>` whereas MPS would use `new arraylist<string>`. This is because some Java classes are built in to the SModel language as domain-specific abstractions to be transformed to Java code.

[2] Unlike Eclipse, the IntelliJ platform is regarded as one of the most popular development environments with increasing market share (Stack Overflow, 2018).

4 Gap Analysis

A detailed gap analysis helps identify a gap in the market for future development. In this case, the development of Alembicue focuses on assisting with the development of Dockerfiles. Therefore identifying issues with existing tooling which also aims to provide Dockerfile assistance is useful in directing the development of Alembicue to ensuring such issues are not repeated, alongside identifying areas for innovation that are not captured by the existing tooling and which Alembicue could potentially support. This chapter details such findings to shape the design of Alembicue.

There exists various tools attempting to provide some partial assistance in certain aspects of writing Dockerfiles, such as linting (identifying errors) and syntax highlighting. The three most popular tools according to Google are dockerfile-editor.com (section 4.1), Visual Studio Code (section 4.2) and Docker's build tool (section 4.3). Alongside identifying what these tools achieve well, identifying issues with these tools helps suggest areas for improvement in future tools such as in the development of Alembicue.

4.1 dockerfile-editor.com

One of the highest results for 'docker editor' on Google at the time of writing is dockerfile-editor.com. This attempts to provide an accessible way to create Dockerfiles, and aims to perform syntax checking on the Dockerfile.

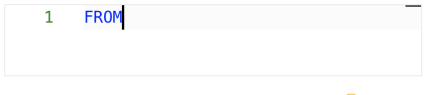
While the source code of the software does not appear to be available, reversing the packed JavaScript allows the functionality of the editor to be analysed in more detail. Built on top of `react-mono-editor` which provides an API for registering language features (contributing to most of the application's source code), the editor itself defines the following functionality using 200 lines of code:

1. A set of Dockerfile commands as possible instructions for completion anywhere.
2. Functions to retrieve Docker image names and versions from the public Docker Hub repository.

However, this functionality is woefully lacking for consumption by developers looking to use the software for composing a Dockerfile. Firstly, there are a number of language definition issues.

- ✖ The extent of syntax checking is to check each line begins with one of the 18 valid keywords beginning an instruction. Beyond this point, the validity of each line is not checked. This means a line consisting solely of an instruction without any parameters is always marked as valid by the application, despite not being a valid file when the build is attempted (Figure 4.1).

- File requirements are incorrectly defined. An empty file is regarded as a syntactically correct file, but attempting to use the file with `docker build` throws an error that the Dockerfile 'cannot be empty'.
- A lack of context for each instruction within the file permits instructions otherwise unavailable due to existence or position of other instructions in the file. For example, instructions which cannot be outside of a build stage are permitted in the syntax checker, yet immediately rejected by the Docker daemon (Figure 4.2).

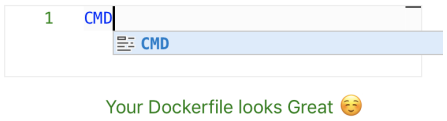


(a) Application reporting syntactically correct file

```
$ docker build .
Sending build context to Docker
daemon 2.048kB
Error response from daemon:
Dockerfile parse error line 1: FROM
requires either one or three
arguments
```

(b) Docker daemon reporting syntax error

Figure 4.1
Dockerfile with instruction missing required parameter



(a) Application reporting syntactically correct file

```
$ docker build .
Sending build context to Docker
daemon 2.048kB
Error response from daemon: No build
stage in current context
```

(b) Docker daemon reporting syntax error

Figure 4.2
Dockerfile with instruction incorrectly outside of build context

There are also a number of problems with the user interface of the editor and user experience of the application.

- Functionality to retrieve Docker image names and versions is dysfunctional. The application makes invalid requests to `hub.docker.com` which are blocked by Docker's access control checks, reporting an invalid image name. Even if this behaviour correctly queried Docker Hub, it would be incorrect to say that an image which does not appear publicly on Docker Hub is invalid, since images can be located anywhere, including locally or on a custom repository. Showing a syntax error for an invalid image name for such a scenario could lead to incorrect conclusions made about files, whether manually or autonomously by continuous integration systems.

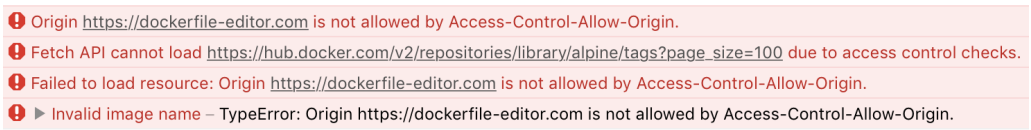


Figure 4.3
Docker Hub request failed access control checks

- 70,000 lines of JavaScript is an extremely large quantity for such an application, and the interface is extremely slow with regular UI thread freezing. This appears to be (in part) an implementation issue of the editor in this case and not with the use of the editor itself, with console warnings present regarding the incorrect setup of a web worker to avoid using the UI thread for background tasks. The usability of such a tool is considerably impacted, demonstrating that a future tool needs to be performant such as in its parsing.

- The language promotes the insertion of keywords anywhere in a file, encouraging code completion for instructions in all insertion point positions, not just the start of a line. This means code completion attempts to complete keywords where keywords are not expected, such as for the parameters of all instructions (Figure 4.4). This does not appear to be due to a lack of language structure defining where instruction keywords can exist in a file (as this is defined by the editor base that this application is built upon), but rather that all words in the file are used as code completion for any string in the file. Overriding the ⏎ enter key to provide these completions unfortunately can provide an unexpected completion action where one would not normally expect a completion, slowing down users proficient with other software.
- Syntax highlighting in the editor is inconsistent in its identification of components of the language. For example, in cases where the instruction of the previous line is solely the keyword without whitespace after it, the instruction keyword on the subsequent line is not highlighted as a keyword. This kind of inconsistency can reduce comprehension of the file, as one may expect the subsequent line is a continuation of the previous line, such as with the use of a line continuation marker (`\`), which in this editor also makes the keyword lose its blue highlighting, but this time correctly so.
- Complementing the previous remarks of syntax highlighting and the lack of appropriate syntax checking for instruction contents, there are also cases where syntax highlighting demonstrates an issue with the file, but which is not reflected in the syntax checking conclusion of the file. For example, mismatched quotes and brackets do not display an error, but syntax highlighting displays incorrect highlighting following the characters entered. This lack of continuity between syntax highlighting and syntax checking provides many opportunities for misinterpretation of the file's contents and the syntax of instruction parameters and options.

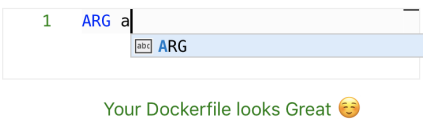


Figure 4.4
Incorrect code completion suggestion for instruction keyword within instruction

4.2 Visual Studio Code

Microsoft's Visual Studio Code (VSC) provides built-in support for the Dockerfile language with the Docker Language Basics extension. In addition, further Docker support is provided through a marketplace extension now maintained by Microsoft. This extension is extremely popular, with over 10 million installations, providing highlighting, code completion and warnings for Dockerfile syntax (Dias and Weatherford, 2018).

- Warnings and errors are presented in the 'Problems' list at the bottom of the editor, and a red underline is displayed at the point of the error, which can be helpful in identifying issues with the file. However, it is often the case that there is no fix available for the error, leaving the user to work out how to resolve the problem. For example, an empty file demonstrates the problem of not having a `FROM` instruction, however no code action is available to automatically resolve the issue, requiring that the user manually insert the correct instruction to fix the issue.

The term *code action* is Visual Studio Code terminology for an intention, a semi-automatic or fully autonomous technique to resolve issues in code introduced in this report's editor design chapter (section 6.4).

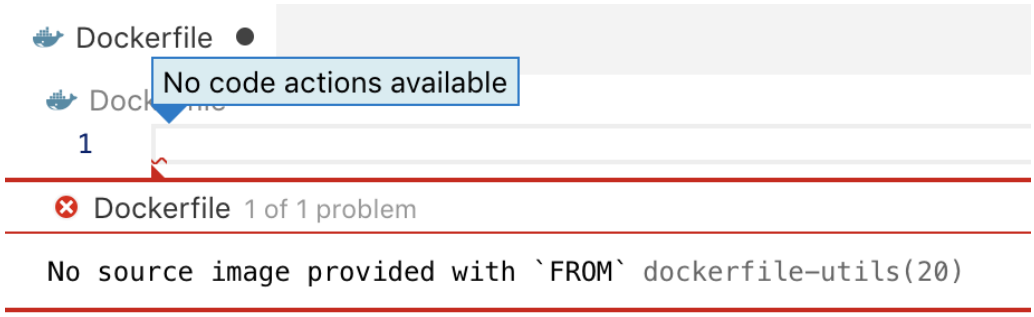


Figure 4.5 Visual Studio Code Dockerfile error lacking intention to resolve issue

- The instructions for installation mention the 'need to install Docker on your machine and set up on the system path' to be able to use the plugin's features (Dias and Weatherford, 2018). By necessitating the Docker daemon installation and access to a Docker machine, plugin functionality is heavily constrained by the environment on which it is run. Many development environments are not configured for running Docker themselves — Docker is a platform enabling applications running within cloud computing environments, so it should not be necessary to install Docker locally just to build containerisation into a 'cloud' application. This provides an inhibition to its adoption, especially in enterprise environments where adding such software to local machines is difficult logistically.

- Instructions with references do not have their references checked. For example, `COPY`'s `from` parameter requires a reference to a `FROM` instruction in the file, but the value set for the parameter is not checked. This means a reference can be made to a nonexistent `FROM` without raising any warnings until the image is attempted to be built.

- Some errors raised by VSC are not errors which the Docker daemon or engine have issue with. For example, `LABEL "" ""` raises an error in VSC that 'LABEL names cannot be blank', but building an image from the Dockerfile with `docker build` does not have any difficulty, successfully creating an image. This is only a minor annoyance in comparison to other issues in this list, especially given that the errors raised are likely something which needs to be looked at anyway, but this inaccuracy can lead to wrong conclusions being drawn about the presence of an error in a file, such as rejecting continuous integration builds.

```
1   FROM scratch as a
2   COPY --from=b foo .
```

(a) Visual Studio Code not reporting error with flag value

```
$ docker build .
Sending build context to Docker
daemon 9.216kB
Step 1/2 : FROM scratch as a
 --->
Step 2/2 : COPY --from=b foo .
invalid from flag value b
```

(b) Docker engine reporting error with flag value

Figure 4.6
Visual Studio Code lacking check of reference

4.3 `docker build`

Docker can attempt to build an image from a Dockerfile with the `build` verb to the Docker daemon, providing error messages if necessary as it does so. This technique for identifying errors can work well for low-stakes local development image production, where images are built locally and quickly with few instructions and limited external requirements. With the build step a necessity in using a Dockerfile to create a container, it is natural that this is the most common place where issues are identified in small scale development environments.

However, such a technique has a major downside which prevents its use in real-world development for identifying issues with Dockerfiles: in many cases the image build process is being performed for real problems to be raised. As demonstrated in subfigure b (Figure 4.7), all instructions prior to the syntax error are executed for real until the syntax error in the file is reached. This requires all the necessary resources for the image itself just to identify issues with the file that would be used to create the image. It may not be feasible to locally recreate the environment in which the Dockerfile is to be run just to perform syntax checking on the file — such an endeavour would not only require any necessary hardware resources for the engine to perform its build steps, but also any local files provided in the build context for filesystem modification instructions to copy local files into the image (chapter 11), both of which may be costly in time and money.

```
$ nl Dockerfile
  1  FROM scratch
  2  LABEL "⏎
$ docker build .
Sending build context to Docker
daemon 2.048kB
Error response from daemon: LABEL
must have two arguments
```

(a) Docker daemon reporting syntax error immediately on file

```
$ nl Dockerfile
  1  FROM scratch
  2  LABEL " "⏎
$ docker build .
Sending build context to Docker
daemon 2.048kB
Step 1/2 : FROM scratch
 --->
Step 2/2 : LABEL " "
failed to process "\"": unexpected
end of statement while looking for
matching double-quote
```

(b) Docker engine reporting syntax error after reaching instruction with error during execution

Figure 4.7
`docker build` linting occurring in Docker engine after instructions are built

5 Tools

Various tooling assisted with the production of this report and the development of Alembicue. Appropriate selection and use of tooling to support development efforts are essential for producing timely and accurate deliverables. This includes correct choice and use of languages for various purposes (section 5.1), configuring a development environment (section 5.2) and setting up version control (section 5.3) for the project.

5.1 Languages

Various languages were used in the development of Alembicue, with the most appropriate language chosen for each specific use case. All languages were published by JetBrains, the developers of MPS, the language workbench chosen for this project (subsection 3.2.3). The languages used were as follows:

MPS (`jetbrains.mps.lang.*`) The MPS language is a modelling language used to describe and implement core components of a language, including its structure. This language also provides extensibility, which Alembicue uses for its declaration of primitive types including booleans, integers and strings. The framework for the aspects model (section 6.1) is provided with this language, used for the core of the editor, and for other functionality of the language to be built on top.

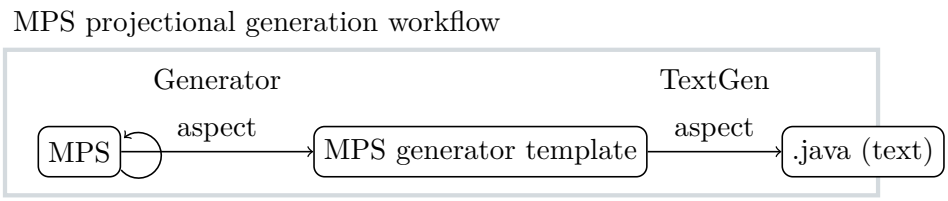


Figure 5.1
Transformation of MPS language to Java

Base Language (`jetbrains.mps.baseLanguage`) For programming of behaviour, BaseLanguage is a projectional counterpart to Java (Konopko, Shatalin and Pech, 2011). This language is used in Alembicue for the programming of actions, behaviours, constraints, intentions, text generation and type system checks.

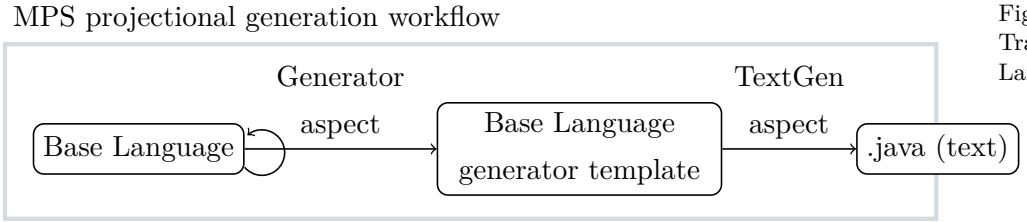


Figure 5.2
Transformation of Base Language to Java

Build Language (`jetbrains.mps.build`) In creating a workflow for compilation of a standalone application, the build language describes the steps taken in compiling and packaging the various components of the Alembicue language. The use of the build language is discussed in the compilation section of this report (section 16.2) as part of preparing Alembicue for release.

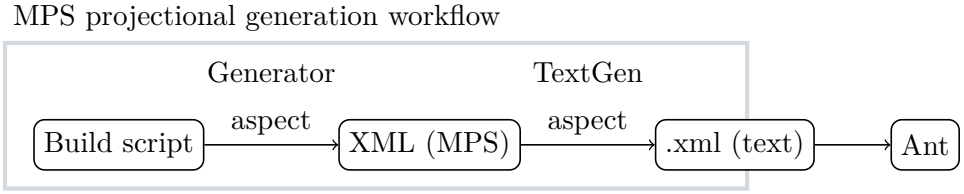


Figure 5.3 Transformation of build script language through to Ant

5.2 Environment

My development environment is macOS as this is my primary OS. Since production use of Docker is primarily with Linux due to its integration with Linux kernel features, my use of a UNIX® operating system for development and testing is advantageous for compatibility with a Unix-like Linux OS. Compilation of Alembicue will also be performed for Windows and Linux on macOS, tested with virtual machines running Windows and Ubuntu respectively.

5.3 Version control

For development of Alembicue, a version control system (VCS) was used to keep track of changes during the software development process and the writing of this report. The chosen VCS was Git — the exact choice of VCS is generally inconsequential as it is only for personal development processes. Git was chosen due to its familiarity and popularity (Stack Overflow, 2018). The Git repositories for software development and this report were self-hosted with Phabricator, which also served as the project management site. Tracking changes in the project with Phabricator is useful for the release (chapter 16), assisting with the compilation of release notes, as well as providing a log of process in terms of this project.

The project's Phabricator can be found at `https://phabricator.georgegarside.com`.

5.4 LaTeX

In writing this report, I used LaTeX to produce my report, with a customised document class based on Memoir. Various customisations aim to make the document more suitable in the context of a final year project.

5.4.1 Document class

- Setting margins to be smaller, with a wide 4 cm margin note width on the right. Margin notes are used throughout the document to provide supplementary content and context to the main narrative, including links and references to other parts of the document, and providing space for figure captions.
- Harvard style bibliography and inline citations powered by biblatex and defined by `biblatex-bath` with custom `lbx` for formatting. Harvard referencing provides sufficient information in an inline citation to be able to recognise familiar citations without needing to refer to the main bibliography, without being too verbose.
- Additional font families: Palatino for front cover title, Menlo monospace font for code, and `fontawesome` for icons.
- Formatting adjustments for preventing window and orphan lines, disabling hyphenation except where breaks are allowed in code, the removal of paragraph indent and an increase to paragraph skip to make paragraphs more consistent while easier on the eyes to follow a line of text.
- Table of contents numerical depth expanded to include subsections for easier referencing of content in the document, and dots changed to grey to be less distracting.
- Captions and sub-captions set up with positioning in the right margin note area beside each floating figure to avoid detracting from the narrative, as current location is the preferred placement, and which is ensured to be within the defined section when relocation is necessary.
- Page header enforced capitalisation is removed for natural casing which is easier to read, and rule lines removed to reduce clutter on the page.
- Title formatting to part, chapter, section, subsection and paragraph to adjust size to make the difference more prominent while fitting with the size of other content on the page. Positioning of titles was also modified with placement on the page alongside accompanying text and with numerical reference to the title positioned in the left margin for easy skim-reading to find the number, and to pronounce the start of a new heading.
- List label for enumerated and itemised lists moved into the left margin, and a supplementary description added which automatically calculates the label width based on the longest label declared as a new environment.
- Code environments declared for code formatting as a figure, and inline code commands to set colour and weight for various syntax.
- Menu paths and keyboard keys formatting with `menukeys` package and custom styles and colour themes for each, to highlight such content in a paragraph while not distracting from its context.
- Additional diagramming environments featuring Tikz, for use with language diagrams and hierarchy diagrams, including colour declarations for various editor components and syntax highlighting, as well as inline Tikz commands for including parts of diagrams in the text.
- Table formatting, both for those holding data and for those demonstrating a tabular cell layout in the projectional editor for a figure.
- Highlighting of text with various highlighter colours behind the text, and for highlighting type system messages (section 6.5) with wavy underlines of various colours.

5.4.2 Syntax diagram notation

In this report, diagrams of the language syntax use the following notation:

non-terminating 

State which cannot be terminated at

terminating 

State which can terminate the instruction

start 

Signifies the start point of the diagram

transition 

Transition from one state to another using the character labelled

keyword 

Exact text for an instruction's keyword

number 

Integer or decimal number `\d+(?:\.\d+)?`

string 

Text `\w+|"[\pL ]+"`

separator 

Separates elements within lists

operator 

Exact text for a symbol for a unary or binary operation performed adjacent state(s)

method 

Exact text representing an enum item

parameter 

Exact text for a parameter name, for the following state(s) to provide the value(s)

Part II:

Projectional Editor

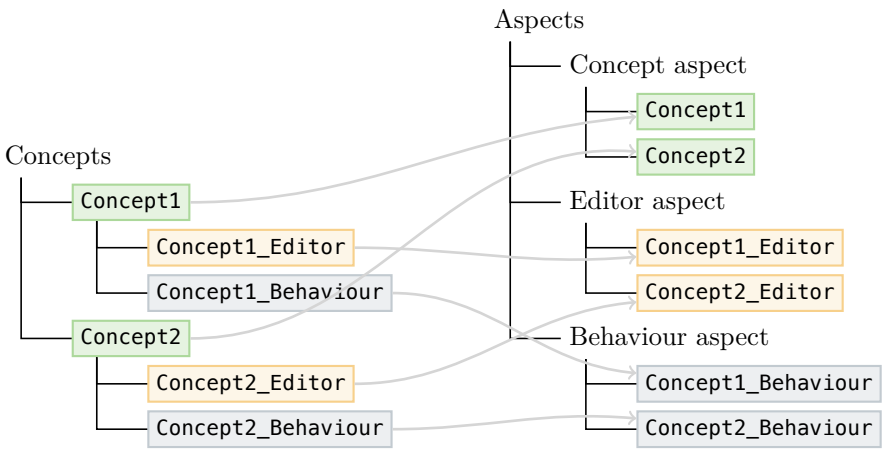
This part discusses the design and implementation of the language in Alembicue for its projectional editor. Beginning with coverage of the design and functionality of the editor component (chapter 6), all language concepts are then introduced in chapters grouped by the functionality they provide to the language, such as the type system (chapter 7), configuration of environment (chapter 8), container (chapter 10) and execution (chapter 9), as well as filesystem operations (chapter 11). For each instruction, both a description and explanation of the syntax of the language is provided, followed by a discussion of how the instruction and any associated functionality is implemented in Alembicue.

6 Design

The projectional editor is the core of Alembicue, providing the functionality for the modification of the abstract syntax tree representing each file. This chapter discusses the design of the editor, including core functionality like syntax highlighting (section 6.2), code completion (section 6.3), intentions (section 6.4), and warnings and errors (section 6.5).

6.1 Aspects

An aspect is a component in the design of Alembicue's projectional language. Each aspect has its own system which is responsible for solely for components of that aspect — even though each instruction (section 7.1) in the language is comprised of various parts, such as an editor, behaviours and constraints, the aspect's system is responsible for all parts of one type regardless of associated instruction.



(a) Example concepts hierarchy

(b) Example aspects hierarchy

Figure 6.1
Example mapping from concept construction to aspect systems

6.1.1 Concepts

A concept can be thought of as a type in conventional programming, and a class in object oriented programming. A concept defines the type for one or more nodes in the tree, including the structure of descendants.

`Concept`

Metadata Information about the concept such as `name`, `alias`, and short `description`, as well as references to any inherited concepts, whether extended or implemented in the case of another concept or interface concept respectively. A concept is `rootable` if it can be the root of the tree — see File (section 7.3) for more information on rootable nodes.

Properties Values for parameters stored in the concept. Properties have restricted types, namely primitives, enumerations, or strings matching a regular expression, termed constrained data types (JetBrains s.r.o., 2018c).

Children Concepts stored beneath the current concept on the tree, as child nodes, declared as optional or mandatory and with maximum of one or many of each.

References Similar to children, but links to existing nodes across the tree or to other trees, with range of possible nodes to link to dependent on concept-defined scope.

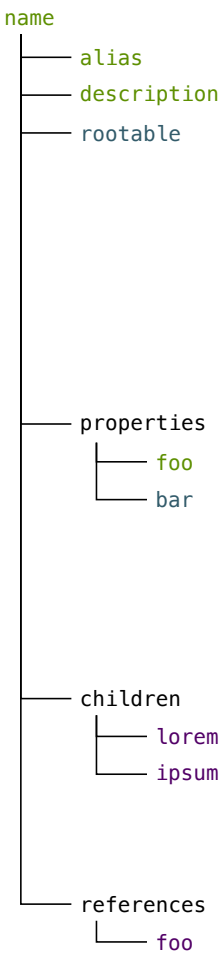


Figure 6.2
Example composition of a concept

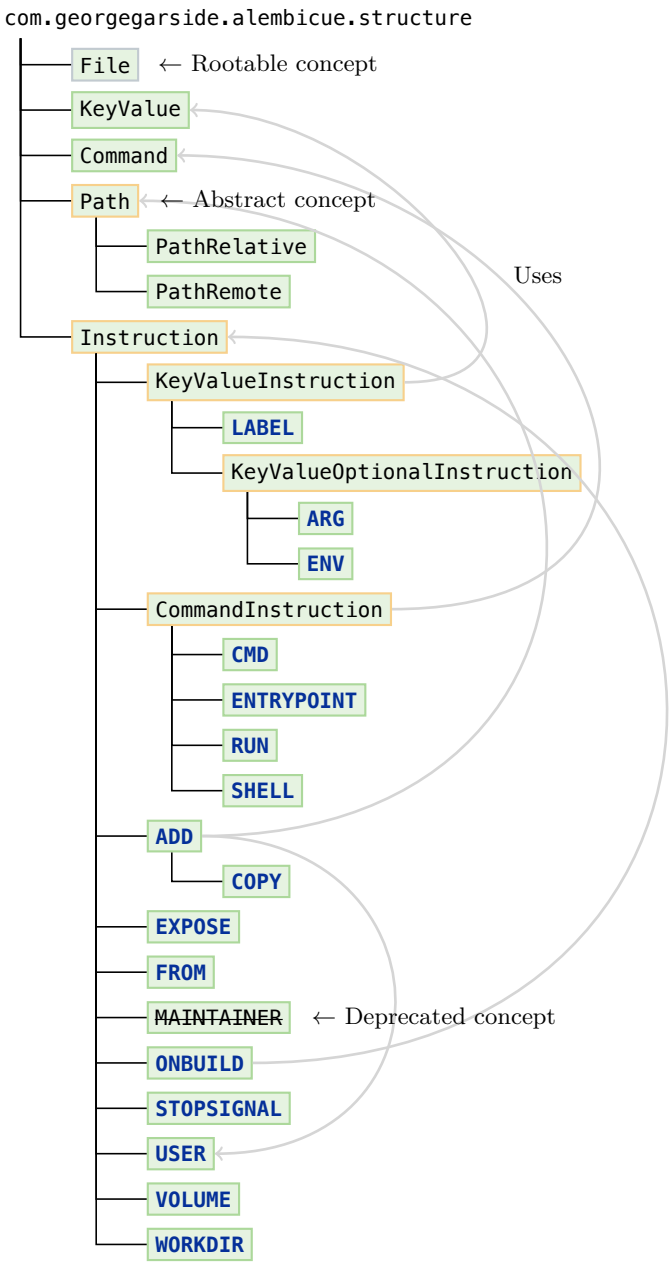


Figure 6.3
Language structure diagram

6.1.2 Editor

An editor is the visual interface for a concept or part of a concept's projection for the user. Such projections are cell-based, with various layouts (including indent, vertical and horizontal) to present the information contained within the concept in an appropriate manner. Editors can be split into editor components, which allow the editor to be constructed from multiple sections and be partially or wholly overridden by subconcepts.

`Editor`

6.2 Syntax highlighting

Alembicue utilises syntax highlighting in font weight and colour to distinguish different types of text projected in the editor, such as if the text can be edited. Unlike a regular text editor, which must parse text to determine syntactic elements and apply syntax highlighting, a projectional editor applies syntax highlighting by rules on cells. This allows the syntax highlighting to operate faster and simpler, since the entire text does not need to be parsed. Rules can be complicated without impeding the speed of syntax highlighting, since highlighting is performed in parallel on all cells, as cell boundaries are predefined by the editor rather than its context.

The following styles are defaults used in the editor:

`default` Generic text without special meaning, such as to provide an argument key or value.

`instruction` A keyword beginning (shown in uppercase) or within (shown in lowercase) an instruction. This text cannot be edited nor selected once it has been fixed in the editor.

`string` Free-form text provided by the user which is output as-is usually within quote marks, such as for a parameter to an external command call. For example, `/bin/bash -c`.

`number` A numeric string consisting of `[0-9]+` unless otherwise specified.

`comment` Text which does not affect the compilation process, provided for humans to read.

`tag` Text usually within a comment but which is parsed for special functionality, such as to indicate a version.

Comments in a projectional editor require special consideration as they are not usually stored in the resultant tree in a regular text document containing code. For more information about the implementation of comments, see Comments (section 7.5).

6.3 Code completion

Alembicue implements code completion for widespread use in the editor. Many of the fields in the editor support code completion, which aims to increase code velocity and accuracy by suggesting the completion of a word or phrase being entered in the editor, or even additional values to come after the entered text. Implementations of this throughout the language are discussed in this report alongside the instruction which introduces it.

Problems with existing code completion solutions are discussed in the Gap Analysis.

Code completion is activated with the shortcut `ctrl` + `space`. This shortcut was chosen as it is the apparent standard across editors, having tested the most popular editors which include code completion per the Stack Overflow (2018) developer survey.

The popup (Figure 6.4) is shown beside the cell with the insertion point currently located. Each option in the code completion popup has three components: an icon to distinguish the type of action which will be performed, a name for the action, and a short description to be shown on the right of the action.

Future work in this area involves using the icon to further categorise the action performed, since many cells have only one type of code completion, that of a node operation.

From that resulting popup, selecting an option will perform the relevant action, such as the insertion of a new initialised node of that type into the tree, or entering a value into a property in the tree for that node. Options can be selected with the mouse, or highlighted with `↑` and `↓` and selected with `⏎ enter`.

Figure 6.4
Code completion for instruction names, showing an abridged list based on context

6.4 Intentions

Intentions are contextual suggestions to mutate code at the current editor location. These can be used to optimise code and prevent errors (JetBrains s.r.o., 2019c). While working in the editor, the context is used to determine which intentions are available for a given editor state, using each intention's applicable conditions.

6.4.1 Declaration

Each intention is defined by the following components, alongside a `string name` to refer to the intention in code and a `string description` for the user to select the intention in the editor:

applicable concepts The intention requires the highest concept for which the intention could apply to be set. At a maximum, the intention can apply to this concept, and any sub-concepts if this is enabled for the intention. Providing a strict applicable concept reduces the number of invocations of the method to determine whether the intention is applicable for the context, if one is provided for the intention.

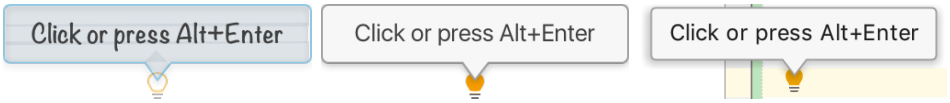
applicable children To reduce the applicable concept scope, a child filter can be applied to specify which children of the node the intention applies for. This is especially useful where the projection hierarchy is long and one may not be aware of the context they are in when invoking the intention for a concept far away from the location of the issue.

applicable context A closure to determine whether the intention is applicable for the given `editorContext` and `node`. Intentions can only be executed when the intention is applicable for the context, determined by this method returning `true`.

execution A closure to be executed when the intention is selected by the user in the editor area, or when the intention is executed automatically as declared as a quick fix.

6.4.2 Editor invocation

Where an intention is available, a light bulb is shown beside the editor area gutter (Figure 6.5). This button can be clicked to show a list of available intentions at the current context. Hovering the light bulb shows help text mentioning this interaction, along with a keyboard shortcut ⌥ alt + ⏎ enter.



(a) Rough sketch (b) Mockup design (c) Implementation

Figure 6.5
Intention help text on light bulb hover

Intentions list The list of available intentions is presented beside the current node as a list. This list can be navigated using the mouse by clicking on an available intention, or with the keyboard using the ↑ and ↓ arrow keys, with ⏎ enter to choose an intention to apply to the context. Where no intention is available for the context, the shortcut performs no function, as the light bulb does not show.

6.4.3 More options

For more options on an intention, the → arrow key can present an additional list of options on the highlighted intention (Figure 6.6). All intentions have two additional options: navigating to the declaration of the intention in the language, and disabling the intention.

Disabled intentions do not show in the list of intentions for any context. If a disabled intention would be the only applicable intention for a context if enabled, the light bulb is not shown. Intentions can be managed from Alembicue › Preferences › Editor › Intentions, where a full list of all intentions are available with checkboxes to enable or disable each intention or all intentions for a language.

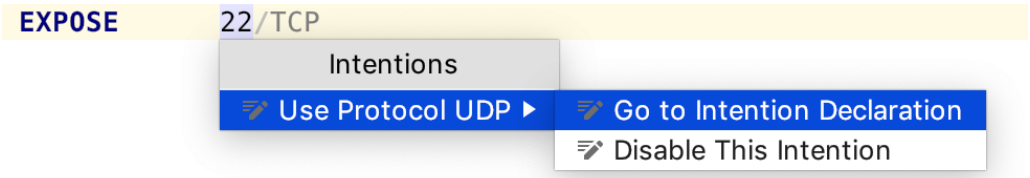


Figure 6.6
More intention options

6.5 Type system warnings, errors & info

Checking rules defined in the language may raise certain type system messages regarding the current tree. These messages can be one of three types:

`Checking rule`

- An error will block compilation of the file, or the instruction in the file, for an issue which has caused an invalid file.
- A warning should necessitate attention, but will not block compilation. Warnings can be to suggest preferable compositions of instructions or instruction components, rather than to state there is a error with the file.
- An informative message which may not necessitate attention, but which may warrant a reconsideration of the highlighted content in the file.

6.5.1 Resolving

These issues can be resolved with an intention or manually. With experience, one will likely know what the problem is and how it can be resolved without looking at the error description — this is where simply highlighting the issue in the editor can be helpful, without obstructing the user with a full error description. Manually resolving the error can take the form of using the keyboard or mouse to navigate to where the problem is, then entering keystrokes to resolve the problem. On each change to the syntax tree, the checking rules which may apply to the current context are re-checked, and if the checking rule no longer finds issue with the code, the error is instantly removed from the projection.

If an intention is used with ⌥ alt + ⏎ enter, the checking rule is immediately applied on selection of any available context-aware intention. Intentions also provide the ability to suppress any level of message (subsection 6.5.2).

This information is projected in the editor as a wavy underline on the relevant cells, with a colour appropriate to the type of message to allow the user to distinguish the level of message without navigating to the cell. Each message location is highlighted in the scrollbar of the editor area (Figure 6.7) per the colour of the highest level message at that location, and messages can be quickly jumped to by selecting the highlight in the scrollbar.

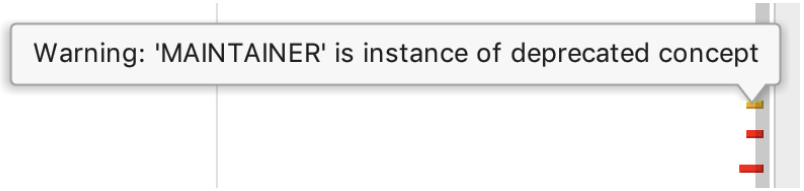
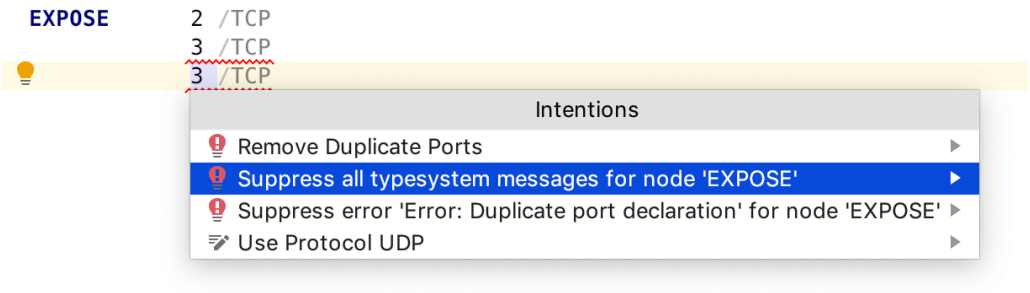


Figure 6.7
Warnings and errors shown by scrollbar in editor area

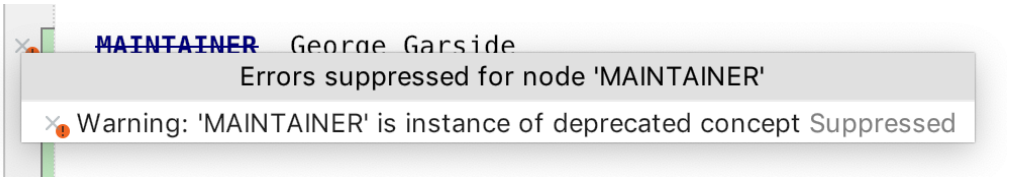
6.5.2 Suppressing

One can suppress errors and warnings in the editor. Suppression can apply to all type system messages for a node, or for a specific message on that node. To suppress an error, show intentions within the range of the error and choose the level of suppression.

Message suppression intention interoperability Suppressing an error or warning will automatically suppress the relevant suggested intention. Disabling an intention (subsection 6.4.3) does not suppress any corresponding message automatically, since the user may wish to be prompted to resolve a message but wish to resolve the message manually, rather than with an automatic intention.



(a) Intention to suppress all type system messages for node, or one or more individual errors currently raised.



(b) Gutter message indicating suppressed warning on node and option to unsuppress.

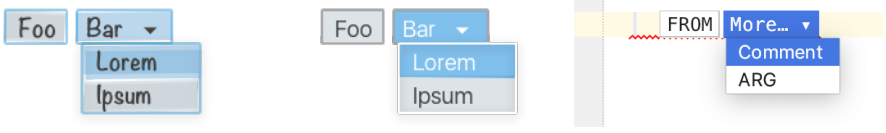
Figure 6.8
Intention to suppress type system messages

6.6 Context assistant

Alongside projected cells containing editable content, and cells for providing context such as keywords and separators, cells can also be used for the promotion of operations. Beside intentions (section 6.4) which focus on potentially useful suggestions for operations to be performed, perhaps in response to errors, it may be preferable to display options to the user directly in the editor, where they are prominently displayed for more likely use.

The context assistant is a collection of buttons and menus projected in context directly beside the cells. These buttons can be accessed by clicking on a displayed button or subsequent menu item, or with the keyboard shortcut ⌘ cmd + ⌥ alt + ⏎ enter (Ctrl + Alt + Enter for Windows) which moves focus to the context assistant and supports ← and → for selection, ↓ and ↑ for moving through a submenu, and ⏎ enter to select an option.

Due to the space consumed in the editor by actions presented in the context assistant, its use is restricted to times where such suggestions are very likely to be actioned, and where such options should not be hidden behind an additional button or keypress. Therefore, the context assistant is used for an empty file to begin adding instructions to it — the context assistant suggests adding a **FROM** instruction, or a comment or **ARG**, to an empty file (Figure 6.9).



(a) Rough sketch (b) Mockup design (c) Implementation

Figure 6.9
Example context assistant as implemented in Alembicue

7 Type system

The type system chapter of this report introduces various concepts which support the language, by providing the basis for the editor, concept behaviour and text generation of concepts, which can be inherited and overridden by individual concepts for their functionality. These meta-concepts provide structure to concepts, two of which are also introduced in this chapter: a concept to represent an empty line in the editor, named `BlankLine` (section 12.1). As with subsequent chapters, discussion of concepts includes both a description and explanation of syntax as used in and exported by Alembicue, and a description and explanation of how the implementation is carried out for the editor in Alembicue.

A concept is defined in the Background chapter of this report (subsection 6.1.1).

7.1 Instruction

An instance of `Instruction` represents a single instruction in a file to be executed.

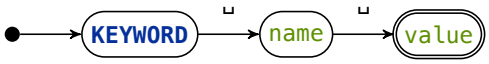


Figure 7.1
Example instruction diagram

7.1.1 Keyword

Every instruction begins with a keyword, identifying the type of instruction. The keyword is case insensitive, but conventionally written in uppercase (Docker, 2018c), so this editor follows convention. The keyword is projected in the **`KEYWORD`** style, which by default is bold dark blue as discussed in syntax highlighting (section 6.2). Separating the keyword from instruction parameters is whitespace. Following the keyword is the contents of the instruction, such as arguments to define how the instruction carries out its operation.

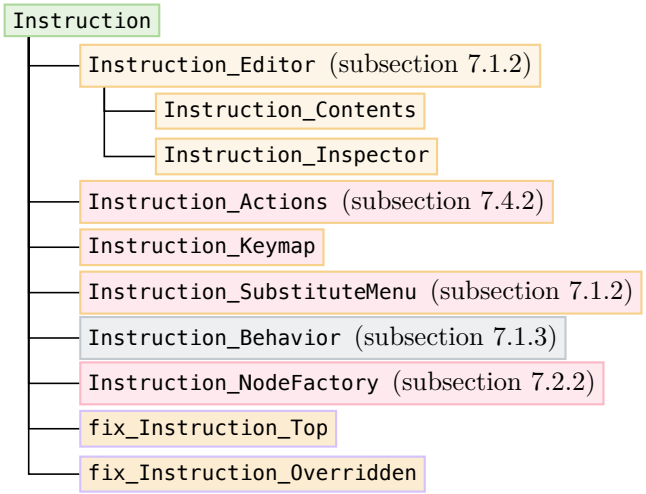


Figure 7.2
Instruction implementation hierarchy

7.1.2 Editor components

As noted in Keyword (subsection 7.1.1), every instruction begins with a keyword. This is set in the concept alias (subsection 6.1.1), and projected in the editor by the `Instruction_Editor` with the alias cell.

`Instruction_Editor`

`Instruction_Editor` is the wrapper for two editor components:

- `Instruction_Contents`

  The contents component provides the main editable content for the instruction. This is is projected to the right of the instruction's keyword (subsection 7.1.1).

- `Instruction_Inspector`

  The inspector components is shown in the inspector tool window while the node is selected in the editor. This has limited utilisation in Alembicue at present, which is an area to work on in future development (chapter 17).

For this component to be displayed in the editor, a cell projected by this node's editor must be focused, or a child of this node must have one of their cells focused and that node must indicate that this component should be placed in the location of the `next applicable editor` cell.

These can be overridden by subconcepts to `Instruction`, providing a projection tailored for the specific instruction.

Keymap It is standardised that ⇧ shift + ⌘ cmd + ⏎ enter complete the current statement and move the insertion point to a new line inserted below (JetBrains s.r.o., 2019a). With Alembicue, current statements do not require any further automatic completion (as there is no need to insert a semicolon to end a line or similar), so the only action is to move to a newly inserted blank line beneath the instruction. To perform this action on this shortcut, `Instruction_Keymap` defines the shortcut to execute a statement to add a `new next-sibling`[1] of a blank line, matching the file's element factory (subsection 7.3.1).

`Instruction_Keymap`

```
item description : <no description>
  keystrokes : <ctrl+shift> + <VK_ENTER>
  caret policy : ANY_POSITION
  show in popup : false
  menu always shown : false
  is applicable : <always>
  execute : (node, selectedNodes, editorContext)->void {
    node.new next-sibling(BlankLine);
  };
```

Figure 7.3
Instruction keymap for statement completion editor shortcut

Substitute menu Instruction's substitute menu is intended for providing code completion assistance to blank lines not currently containing an instruction. This is discussed further in the implementation of `BlankLine` (subsection 7.4.1).

`Instruction_SubstituteMenu`

7.1.3 Behaviour

In the development of a concept and its associated components, such as editors and type system definitions, additional executable code may be necessary to supplement those components or others in Alembicue. A behaviour is a collection of code, thought of like a class, which contains methods on a concept and an optional constructor for that concept. Further discussion of usage and implementation is made in this report where first utilised, such as for automatic calculation of build stage indexes for multi-stage builds (subsection 8.1.5).

`Instruction_Behaviour`

[1] Various methods provided by an MPS language for language construction support spaces as part of the method name, since space is not used as a delimiter in a projectional editor — there is no delimiter because the cell referring to a method name is self-contained.

7.2 Key-value

A key-value pair is a mapping from one data value to another. The first data value is known as the key, providing a unique identifier for the record (AWS, 2019). A set of such mappings forms an associative array. Such pairs are useful for recording variables and properties in the language, so this implementation is used as a type in various concept's children. The implemented type (subsection 7.2.1) is useful enough in various instructions for its own super-concept `KeyValueInstruction` (subsection 7.2.2).

7.2.1 `KeyValue`

Each pair is declared in the syntax tree with the concept `KeyValue`. This provides the editor for the pair: a `key` cell and `value` cell separated by an equals sign, representing the output of text generation on the concept. The equals sign is projected in the editor as a constant which cannot be selected nor edited, just providing guidance and context for the cells beside it.

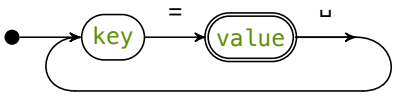


Figure 7.4
Key-value pair example

Syntax highlighting Depending on the type of the `value` for the pair, the text colour of the projected cell changes (Figure 7.5). This helps immediately distinguish the type of data stored within, and highlights automatic type conversation taking place, such as the string `tru` becoming boolean type when an `e` is appended to result in `true`.

string: `foobar` number: `1234` boolean: `true`

Figure 7.5
Value type highlighting

Editor actions The editor for `KeyValue` contains two components, key and value property cells. To make editing more natural, as well as `tab` and `shift` + `tab` moving between these cells, the backspace action on an empty value cell moves the insertion point to the end of the key cell. This imitates the action of pressing backspace in a regular text editor from the value cell, however unlike a regular text editor which would delete the equals sign, the projectional editor's equals sign does not form part of the underlying structure of the document and cannot be removed. The custom backspace action, used where the selected cell is not the key cell and the insertion point is at the leftmost position of the value, will jump the insertion point to the rightmost point of the key cell, skipping over the equals sign. Even though the equals sign still remains projected, the effect of being able to move cells using standard text-based input keystrokes makes the editor feel natural, imitating the response of a user who has moved to the value in error with a text editor.

[>	^{ key }	----->	=	^{ value }	<]

Table 7.1
`KeyValue` editor

```
action BACKSPACE
  can execute : (editorContext, node)->boolean {
    editorContext.getSelectedCell().getCellId() != "kv-key";
  }
  execute : (editorContext, node)->void {
    node.select[
        in: editorContext,
        cell: kv-key,
        selectionStart: node.key.length()
    ];
  }
```

Figure 7.6
`KeyValue_Actions` backspace to move from empty value to key

Text generation Text generation outputs the key and value wrapped in double quotes if the string contains spaces. Requiring this at the lowest level possible ensures this step takes place when necessary, so that all keys and values with spaces are wrapped with quotes in the output.

7.2.2 Key-value instruction

`KeyValueInstruction`

Various instructions support an associative array as a parameter, to provide data as values for the execution of the instruction. Such instructions extend the concept `KeyValueInstruction`. This concept sets the `value` children for the instruction, as a `[1..n]` (non-empty) array of `KeyValue` concept instances.

`Instruction _NodeFactory`

Value initialisation factory Each key-value instruction requires that at least one key-value pair is set for the `value`. To assist with this, each new instruction which is an instance of a key-value instruction concept should initialise a new instance of `KeyValue` and add it to the `value` list. This is performed with a node factory which initialises a new concept instance as follows:

```
newNode.value.add new initialized(<default>);
```

7.3 `File`

A rootable concept is a concept (subsection 6.1.1) which has been registered as being able to be at the root of the abstract syntax tree. This means that while the concept still has a hierarchical parent, instances of the concepts stored as nodes in the tree are not required to have a parent.

The file is the rootable concept in the language implemented in Alembicue. It provides the basis for all instructions, stored as children nodes in a `[1..n]` (non-empty) array `instruction`.

7.3.1 Element factory

While a node factory initialises a node for use in the tree (subsection 9.2.2), an element factory is concerned with creating new nodes in a list, where the list is an abstract type or super-concept of the intended concept to be added. The element

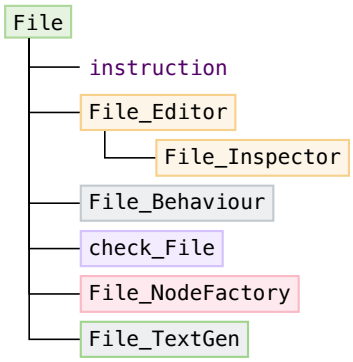


Figure 7.7
File implementation hierarchy

factory for File specifies that a new instance of a BlankLine concept is added as a node each time a new node is requested to be a child, for example on pressing ⏎ enter to mimic a text editor.

7.3.2 Behaviour

File_Behaviour

The addFROM File behaviour's purpose is to support the context assistant (section 6.6). This behaviour assists with the creation of the common first instruction in a File, that of **FROM** (section 8.1). When the file is empty, the context assistant suggests the insertion of a **FROM** instruction at the top of the file. Selecting this action calls this addFROM behaviour.

```
if (this.instruction.all({~it => it.isInstanceOf(BlankLine); })) {   If only blank lines
  this.instruction.forEach({~it => it.detach; });                    Remove blank lines
}
this.instruction.insert(0, new node<FROM>());                        Insert at the top of the file
```

Figure 7.8
File's addFROM behaviour

7.3.3 Text generation

As File is the rootable concept, any instances of this concept should produce a new output file. By supporting rootable nodes as beginning a file, more than one tree is supported in Alembicue by creating one file for each tree.

File_TextGen

The production of the output text file is controlled with a text generation aspect. This aspect controls metadata of the output file, as well as its contents.

File name The file name is set as the name of the node in the tree prepended with 'Dockerfile-', unless the name of the node is exactly 'Dockerfile' in which case 'Dockerfile' is set as the name. This convention is chosen as 'Dockerfile' is the standard file name (Docker, 2018c), and Alembicue supports more than one Dockerfile output requiring disambiguation. Alembicue itself does support more than one node with the same name, since the projectional editor refers to the underlying structure to disambiguate rather than the name of the node, but the plain text output text does not have this ability, so this step is taken to avoid having every file name be 'Dockerfile' in multi-file projects.

Instructions For generating text from the list of instructions stored under the File node, the text generation iterates the list and calls the text generation aspect of each instruction to create a line of text, then outputs this followed by a line break (Figure 7.9). Each instruction has a TextGen instance, either custom to the concept or inherited from a super-concept, which is responsible for the specific instruction instances being generated as text.

```
foreach line in node.instruction {                    Iterate all instructions
  concept switch (line.concept) {                     Concept-aware switch
    exactly BlankLine :                               Skip invoking text generation on empty line
      append \n ;                                     Blank lines are preserved in final output
      break;
    subconcept of Instruction :                       Bug fix
      continue;                                       Perform default behaviour
    default :                                         Default behaviour invokes instruction generation
      append ${line} \n ;                             Each instruction is line break separated
      break;
  }
}
```

Figure 7.9
File text generation for each instruction

7.4 `BlankLine`

`BlankLine`

Representing blank lines in a file, the `BlankLine` concept is an instruction which provides no output. Projected in the editor as an editable constant, blank lines support the typing of characters on top, for the purpose of creating an instruction. A blank line concept is used as this is regarded by the editor as a valid instruction, despite serving no programmatic purpose, as this allows the blank line to be preserved both in the editor without being rejected from the tree as an incomplete instruction, and in the output file as an empty line (Figure 7.9).

7.4.1 Instruction completion

`BlankLine_Editor`

While typing to begin an instruction, at first this string is highlighted with a red background to indicate text which does not create a valid instruction. Once the keyword for an instruction is typed, the `BlankLine` is replaced with the instruction matching the typed keyword. This transformation occurs as soon as a valid instruction keyword is completed. Until this point, the syntax tree has not been modified with the partial instruction line, and the output text does not contain this invalid string. This is demonstrated with **`FROM`** (Figure 7.10).

(a) `EmptyLine` instance

(b) Invalid partial left side transform entered

(c) Completed transformation to instruction instance

Figure 7.10
New instruction keyword automatic completion example

`BlankLine`
`_TransformationMenu`

Lowercase To further aid with completion of instructions, lower case instruction completion has been implemented. Since instruction keywords are defined to be uppercase (subsection 7.1.1), entering a lower case string for a keyword could not form a valid instruction without transformation. Therefore, lower case strings can be more aggressively completed, since it is not expected that the user is continuing to type at full speed without looking at the output on the screen, as may be the case with a partially completed upper case keyword. This more aggressive completion will create an instruction as soon as enough letters have been entered to uniquely identify a keyword for an instruction. For example, to enter the instruction **`ENTRYPOINT`** while **`ENV`** also exists as an instruction, typing '`ent`' would instantly complete the transformation to **`ENTRYPOINT`**.

```
section({ side transform : left }) {
  parameterized
    parameter type: mapping<string, concept<>>
    parameter values: (editorContext, node, model)->
        sequence<mapping<string, concept< >>> {
      list<concept<Instruction>> concepts = concept/Instruction/
          .sub-concepts(model)
          .where({~it =>        Only complete instructions, determined as concepts with keywords
              it.getName().toUpperCase().contentEquals(it.getName()); }).toList;
      map<string, concept<>> out = new hashmap<string, concept<>>;
      concepts.forEach({~it =>                                   Build list of completions
        string name = it.getName().toLowerCase();
        int lastUniqueChar = 0;                                  Index of disambiguating character
        do {                                 Calculate length of keyword prefix needed for uniqueness
          string substring = name.substring(0, ++lastUniqueChar);
          if (concepts.where({~it =>           Unique if substring matches exactly one concept
              it.getName().toLowerCase().startsWith(substring); }).size == 1)
                { break; }
        } while (lastUniqueChar < name.length());
        out[name.substring(0, lastUniqueChar)] = it.asConcept;
      });                                List contains unique substrings of disambiguation length
      out.toList;
    }
    action
      text (parameterObject, editorContext, node, model, pattern)->string {
        parameterObject.key;
      }
      can execute <always>
      execute (parameterObject, editorContext, node, model, pattern)->void {
        node.replace with(parameterObject.value.new initialized instance());
      }                          Ensures initialisation of new node is performed using factory
      action type (parameterObject, editorContext, node, model)->node<> {
        parameterObject.value.asNode;
      }                                Provides hint to editor of the type of action taking place
  }
```

Figure 7.11
Implementation of aggressive side transformation for instruction completion

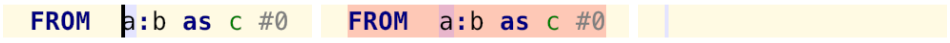
7.4.2 Instruction deletion

Now that the instruction is in the tree, it might want to be deleted. Deletion in a regular text editor usually involves moving the insertion point to the right-most position of the line, and 'backspacing' the line, or using a keyboard shortcut in an editor supporting such an action to remove a line or number of lines. In a projectional editor, the entire projection does not reside on one line, and neither does any part of the projection necessarily need to conform to such line-based conventions.

Deletion of an instruction in Alembicue can be performed in two ways. The least ambiguous method is through pre-selecting the range to be removed. Selecting a node in the editor first (such as with ⌥ alt + ↑), then pressing ⟵ backspace, will delete the node without confirmation, since the selection has been specifically made.

For more information on selecting nodes in Alembicue, see Selection (subsection 13.2.3).

Without pre-selecting a range of cells forming a node, simply pressing ⟵ backspace when there is no string character immediately preceding the insertion point is an ambiguous action, since there may not be a natural next character to be removed, or whether one or more nodes themselves should be removed. Where there is not an action aspect specific to an instruction or cell to be performed on the 'backspace' action, a deletion approval is shown in the editor (Figure 7.12). A deletion approval is shown as a light red background around the cells in scope to be deleted. Pressing ⟵ backspace once more confirms the deletion, removing the node from the tree.



(a) Example existing instruction

(b) Highlighted scope of deletion approval request

(c) Deleted instruction

Figure 7.12
Deletion approval of ambiguous deletion command for node `Instruction_Actions`

Replacement with `BlankLine` In a regular text editor, deleting the contents of a line (such as removing the instruction it contains) does not remove the entire line itself — in the case of 'backspacing' mentioned, an additional backspace keypress would be required to shift the insertion point to the line above.

However, in a projectional editor, simply removing the node from the tree would completely remove its projection from the editor, shifting the insertion point to the line above. Since this is likely to be unexpected behaviour, a backspace action was implemented. This action operates on the deletion of all instructions carried out without a scope being pre-selected (otherwise the insertion of a new node may be even more unexpected).

When backspacing a blank line, no confirmation is needed, so the action immediately removes the blank line the node from the tree. Otherwise, deletion approval is requested, highlighting the relevant nodes in the editor which are to be removed. Once approval is given, the instruction is replaced with a newly initialised blank line instance. By replacing the node on the tree in this manner, the ordering of the nodes and positioning of the insertion point is retained appropriately.

```
if (node.isInstanceOf(BlankLine)) {
  node.detach;                                  Deleting a blank line does not require confirmation
} else {
  if (node.approveDelete [in: editorContext])   Require second action for confirmation
     { return; }
  node.replace with new initialized(BlankLine);  Replace non-blank instruction
}
```

Figure 7.13
Replacing instruction instances with blank line on deletion

7.5 Comment

All line comments begin with a `#` and are followed by any string. All inline comments are surrounded by `/*…*/`. Comments in the editor are projected using the comment formatting as covered in Syntax highlighting (section 6.2).

```
# Lorem ipsum dolor sit amet.
/* Lorem ipsum dolor sit amet. */
```

Figure 7.14
Example line and inline comment

7.5.1 Line comments

`Comment`

For the implementation of line comments, an instruction (section 7.1) was created for this purpose. These instructions do not create a new layer in the resulting image, but are still represented as such in the editor, with the Comment concept inheriting from the Instruction concept. Comments have one property, a **`string`** `text` which contains the text of the comment. Line comments are output in the resulting Dockerfile in the same form they are projected in the editor.

[-	
#	^{ text }
-]	

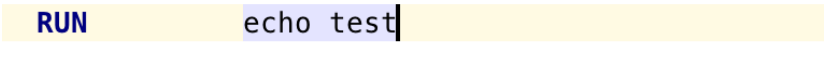
Table 7.2
`Comment` editor

Projection The octothorpe is purely a symbolic projection — as the document does not need to be parsed, the `#` serves no syntactic purpose unlike it would with programming languages based on plain text files. In a regular text document containing source code, comments are parsed and removed from the resulting syntax tree. However, when the tree is the source of the projection in the editor, comments do need to be stored in the tree. This concept is the support for instances of comments being stored as nodes in the abstract syntax tree of the document.

7.5.2 Inline comments

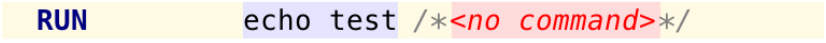
Unlike line comments, inline comments are not reflected in the output file, and are merely projected into the editor as a separate component. Sections of the tree can be commented and uncommented using ⌘ cmd + / , a shortcut for Code › Comment .

Furthermore, inline comments are not designed for freeform text, and instead are instances of syntactically correct partial abstract syntax trees, with an annotation that it is not part of the tree for validation or type system purposes. This ensures the tree remains correct while containing comments. For example, commenting a child node filling a required ([1]) child position will immediately instantiate a new instance of the child to be completed, ensuring the parent's requirement for a child is always fulfilled. This also applies in the other direction, where uncommenting the now commented child will comment the newly instantiated child, ensuring only one child is present at any time.



(a) Pre-comment

(b) Post-comment of command child and creation of new required child



(c) Uncomment of commented child comments new child

Figure 7.15
Example inline comment and uncomment actions

8 Environment configuration

Configuring the execution environment for an image begins with the **FROM** instruction specifying the base image (section 8.1), then arguments to the image are provided with **LABEL**, **ENV** and **ARG** instructions (section 8.2). This chapter introduces these concepts, how they are used in a file to configure an environment, and their implementation in Alembicue.

8.1 FROM

Indicating a new build stage, **FROM** specifies the image used.

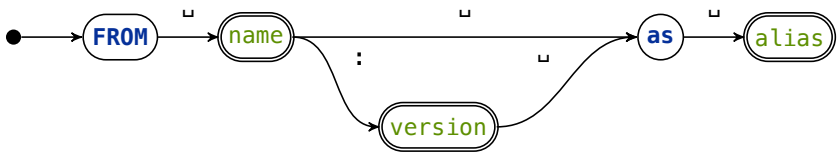


Figure 8.1
FROM instruction diagram

8.1.1 Parameters to a build stage

Parameters to the **FROM** instruction include the name and version of the image for the stage, and a name to refer to the stage later on in the build process in the case of multi-stage builds (subsection 8.1.5).

`name` The name of the image to base this build stage on, which matches the name of an image available in a repository or locally tagged (`docker run -t name`).

`version` The version of the base image to be pulled, matching an available version for the named image (`docker tag name repository/name:version`).

'pulling' an image refers to downloading an image from an image repository, such as Docker Hub or self-hosted repository.

`alias` A name to refer to this build stage later in the file in the case of multiple build stages.

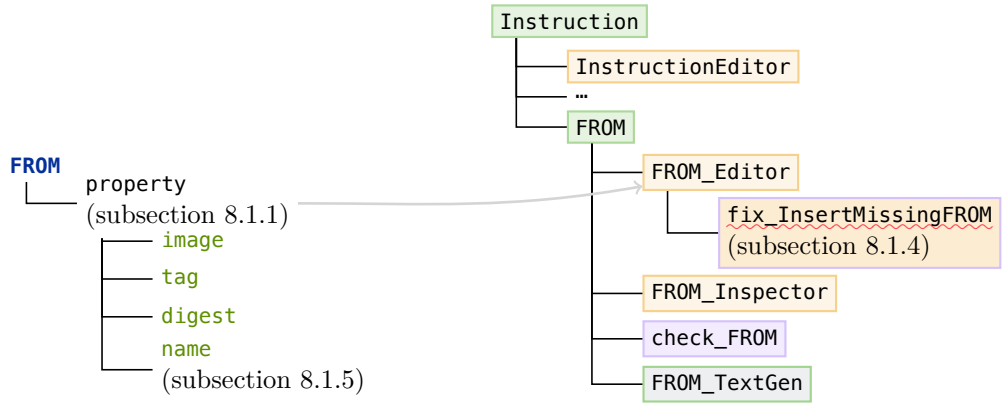
```
FROM image[:<no tag>    ]as default #0
          [@<no digest> ]

FROM mariadb:latest as db #1
```

Figure 8.2
FROM editor examples

8.1.2 Image version either/or implementation

The version of the image is given as a parameter to this instruction. The concept's two properties, `tag` and `digest`, are projected in the editor with a vertical cell layout, with light blue braces surrounding. tab and shift + tab move the insertion point between these two cells, among others. Following the image name, a colon precedes the version number or name, or an at sign precedes the revision hash.

When one of the two parameters has a value, the other parameter is hidden, alongside its corresponding prefix (`:` or `@`) and blue braces wrapping the



(a) `FROM` concept hierarchy (b) `FROM` implementation hierarchy

Figure 8.3 `FROM` implementation hierarchy

parameters. Since it is not possible to have an empty value for either property, these properties are marked as 'auto deletable' in the editor projection, so that an empty value automatically removes the property node from the tree rather than storing an empty string.

[>	{ image }	[/			as	^{ name }	#	*R/O model*	<]
			?[>	: ^{ tag } <]					
			?[>	@ ^{ digest } <]					
		/]							

Table 8.1 `FROM` editor

8.1.3 Postfix version property code completion

A common version name for a tag is 'latest' which ensures the latest base image is used, so this is provided in code completion. Furthermore, many images use the postfix '-alpine' on the version name to indicate that the version of image is itself based on the Alpine Linux distribution, so this too is provided in code completion (section 6.3) to append on the current value entered. In the manner of a transformation menu (subsection 12.2.1), using the common ctrl + space shortcut, 'latest' and 'latest-alpine' is offered as suggestion on an empty cell (Figure 8.4), and when the cell is partially filled the '-alpine' suffix to the current text is offered'.

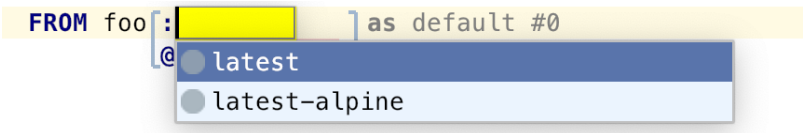


Figure 8.4 `FROM` instruction editor, 'latest' tag code completion

Implementation Code completion as mentioned for assisting with choosing common tags for images is available at any time when the insertion point is placed within the tag cell. As an addendum to the completion cases mentioned, there are two more cases to consider: if the current value of the cell matches 'latest', this is offered as a partial completion of the rest of the word, alongside that exact value with '-alpine', and if the current value is 'alpine' the '-alpine' suffix is no longer suggested ('alpine-alpine' is nonsensical). All these cases are combined into a menu part providing property values.

```
(_, node)->list<string> {
  list<string> values = new arraylist<string>;                List to store completion items
  values.add("latest");                                       Latest is always a standalone option
  if (node.tag.isNotEmpty &&                                  To append to current text, there must be existing text
      node.tag.endsWith("alpine")) {                          Don't keep appending when alpine is already selected
    values.add(node.tag + "-alpine'');                        Append suffix to existing value
  } else {
    values.add("latest-alpine");                              Offer to replace existing value
  }
  values;                                                     Return list of completions
}
```

Figure 8.5
Menu part providing property values for tag in `FROM` instruction

8.1.4 Placement of instruction

The placement of **FROM** in a file has two requirements, a requirement for inclusion and a requirement for relative ordering among other instructions.

Inclusion requirement To make sure the file contains a **FROM** instruction, the checking rule `check_File` contains a clause to ensure that there exists such an instruction within the File's children: `file.instruction.ofConcept<FROM>.isNotEmpty`.

`check_File`

To resolve the issue, the quick fix `fix_InsertMissingFROM` is available as an intention to add a **FROM** instruction to the file. Regardless of how simple an error is to fix, providing an intention to resolve the problem ensures a consistent interaction exists to resolve errors automatically for the user. The quick fix uses `Instruction`'s `getFile` method to traverse the tree up to an insertion point, then inserts a new **FROM** instance.

Ordering requirement To ensure all **FROM** instances in a suitable correct location given other instructions in the file, another checking rule in `check_File` exists. The checking rule (Figure 8.6) iterates over instructions up to the first **FROM** instruction if it exists, checking that all non-blank non-comment instructions, are allowable instructions: **FROM** or **ARG**. This is performed in two steps instead of a single filter on the list so that those nodes can be individually targeted with an error message.

`check_File`

```
node<FROM> fromIns = ins.ofConcept<FROM>.first;
ins.headList(ins.indexOf(fromIns))
  .where({~it => !(permitted meta); })
  .forEach({~it => if (!(permitted instructions)) {
    error "error message" -> it;
  }});
```

Figure 8.6
Checking rule pseudocode for ordering requirement on `FROM` instructions

8.1.5 Multi-stage builds

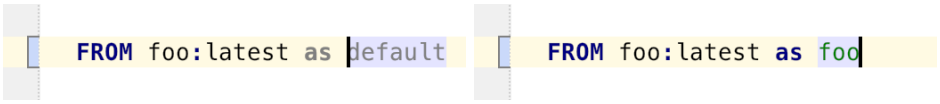
The **FROM** instruction begins a new build stage. Generally, a single file contains a single **FROM** instruction, and therefore a single build stage, which may create many layers from various subsequent instructions. A file may contain more than one build stage, denoted by more than one **FROM** instruction, which can be used

to perform actions with different base images. The ability to copy artefacts from one build stage to another without scripts outside of the build process allows for streamlined build pipelines and encourages a reduction in size of the final image.

To refer to the image with the base image defined by the **FROM** instruction, a name can be provided. This name is used in instructions which are able to perform actions with multiple images, for example **COPY** which can copy files from a previous image into the current image (section 11.4). This optional parameter is only useful if such instructions exist, and an info level message is shown in the editor if a name is given without being used, similar to warning messages in other IDEs which are used to highlight unused variables in code (JetBrains s.r.o., 2019b).

Editor implementation Unless multi-stage builds are required, it is not generally necessary to provide an alias for a build stage. Therefore, the editor does not need to require an alias be provided, or even encourage its use when it is not necessary.

The editor cell for providing an alias is pre-filled with a placeholder, and the cell is not highlighted red when empty. The editor cell can be navigated to with tab, and the placeholder is removed when the user types. Since the 'as' separator is not output when an alias is not provided, this word is projected greyed out until the alias is no longer empty, at which point it returns to blue.



(a) Projection without stage alias (b) Projection with stage alias

Figure 8.7
`FROM` editor projection of `as` separator

Numerical reference Multi-stage builds can also have stages referred to numerically, by the index of the stage. Each **FROM** instruction begins a new stage, incrementing the build stage index. The index of the stage is displayed to the right of **FROM**'s configuration, as a read-only numerical cell beside an octothorpe. The number is calculated from the tree, updating dynamically as the instruction changes position or as instructions around it change, such as by moving **FROM** or adding/removing **FROM**s above.

The calculation is automatic, using an Instruction behaviour (subsection 7.1.3) to determine the index. The behaviour, `int indexOfType()`, is calculated through File inspection (Figure 8.8).

`Instruction_Behaviour`

```
this.ancestor<concept = File>                  Tree ancestor traversal until concept is File
  .instruction                                                    All File instructions
  .where({~it =>                    Filter to create new list determined by predicate
    it.concept.equals(this.concept); })   Concept must match current node to be counted
  .indexOf(this);                                          Index of current node in new list
```

Figure 8.8
Instruction behaviour `indexOfType`

8.2 LABEL, ENV & ARG

Three instructions inherit `KeyValueInstruction` (subsection 7.2.2): **LABEL**, **ENV** and **ARG**. These instructions are conceptually and syntactically very similar, but provide different meaning per the schema.

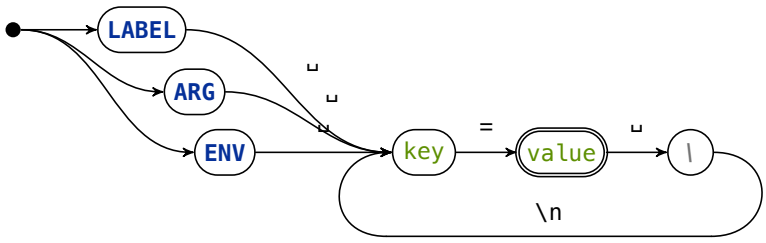


Figure 8.9
`LABEL`, `ENV` and `ARG` instruction diagrams

8.2.1 Comparison

LABEL Store key-value metadata about an image. for example a description. Labels apply to the entire image, not just instructions which follow. Labels apply to the image and are unavailable from the container (without reflection.)

Here, *reflection* is used as a reference to more advanced containerisation ideas, e.g. passing the Docker socket to allow the container to 'escape' its isolation. This is outside the scope of this language-based project. Mentioning it here provides context in comparing these instructions, as it would be incorrect to say that it is *not* possible to access labels from the container.

ENV Set environment variables for commands run in the container, such as with the **RUN** instruction. Environment variables also persist to the container (Docker, 2018c), therefore applying to the **ENTRYPOINT** and **CMD** instructions, as well as other commands run in an image (for example interactively). Environment variables can be overridden by orchestration: tooling used to run a container based on an image can modify the execution of the container without creating a new image, such as with `docker run --env foo=bar imagename` to be run in a command with **CMD** `${foo}`.

Environment variable expansion with **CMD** will use the shell expansion; dollar expansion is syntax for `sh` and `bash` among others.

ARG Set build variables for the image build process (subsection 3.1.2). These variables are available in subsequent instructions in the file, such as to provide parts of a **WORKDIR** (section 11.1) path or **CMD** parameter. Unlike the other two instructions, the value to a key-value pair for **ARG** is optional. This is supported in Alembicue with a `KeyValueOptionalInstruction` which extends `KeyValueInstruction` and overrides the requirements for a value in the editor projection and the text generation aspects. Values for build arguments can be given as part of image orchestration where a container is to be run but an image is needed, or when a specific image is requested to be built on-demand with custom parameters, such as with `docker build --build-arg foo=bar imagename`.

All three types of key-value instruction have their values presented when inspecting the image, and are inherited by subsequent build stages, whether in the same file (subsection 8.1.5) or from another file. Variables can be used in instructions with `${}` if following Dockerfile expansion, or the relevant expansion formatting of the shell being used if set with `SHELL` or specified as part of another command instruction (section 9.2).

8.2.2 `ARG` before `FROM`

An `ARG` instruction can precede the first `FROM` instruction. This is an exception to the checking rule of `FROM` which would otherwise prevent instructions appearing before the first `FROM`. This is useful in the case where the base image used in the first build stage should be parameterised using a build argument.

`check_FROM`

However, due to `ARG`'s scoping of not applying to the entire image but just the contained build stage, the variable is then unavailable in the actual build stage for instructions to use it, despite there not being such a build stage declaration above the first `ARG`. To use the build argument inside the build stage for future instructions, it is necessary to redeclare the variable with an additional `ARG` instruction repeating the name of the variable to reassign its scope.

```
ARG foo=bar
FROM abc:$foo
RUN echo $foo
```

(a) Undefined variable

```
ARG foo=bar
FROM abc:$foo
ARG foo
RUN echo $foo
```

(b) Repeated `ARG` declaration to redefine variable

Figure 8.10 Example of undefined build-time variable caused by `ARG` scoping

8.2.3 List folding

For the key-value pairs stored as children in these instructions, code folding can hide the potentially lengthy values stored and only display the keys. While folded, the keys of the pairs are shown comma separated in a single line, reducing the height in the editor consumed by the projection of the list.

The gutter edge shows arrows around the region which can be folded, and clicking either arrow button folds the list into a single line. Clicking again on the button in the gutter edge, or clicking on the folded list, unfolds the list back to its original form.

```
LABEL  test 1        = 2
       longer value  = with spaces
       boolean       = true
```

(a) Unfolded

```
LABEL  test 1, longer value, boolean
```

(b) Folded

Figure 8.11 Code folding of list of key-value pairs

9 Execution preparation

With a defined and configured environment for the container (chapter 8), commands can be executed within. This chapter introduces the super-concept of many command-based instructions which defines the structure and editor of a command. Also introduced are the instructions which extend this concept, including the ability to execute instructions at compile time of the image or runtime of the container.

9.1 Command

A command is a line of code which is interpreted and executed, such as by a shell. Docker images support running commands in a shell or with `exec`. Each command is comprised of a `string command` property which stores the command to be executed, and a `boolean isShell` which indicates whether the command is run in a shell or with `exec`, referred to hereinafter as the 'form' of the command.

[-	[>	?* *R/O model access* *	?"	^{ command }	?"	<]	-]

Table 9.1 `Command` editor

9.1.1 Changing command form

Execution occurs by default in a shell, such as `/bin/sh -c`. The option is given to execute the command without a shell by calling `exec` with the executable given. This alternative form is given to the build process as an array of strings, where the first string is the executable to be run, and subsequent strings are arguments to the executable; for example `["foo", "bar", "baz"]` where `foo` is the executable, and `bar` and `baz` are parameters.

Alembicue provides an abstraction level above this, removing the need to construct the array delimiters and separators in the string. The projectional editor allows for the entering of the raw command string in either shell or exec form, then signifying whether the command needs to be converted to an array of strings by setting `isShell`. This boolean can be changed with the following functionality available in the editor, provided by the aspect shown on the right of the item.

exec → shell A completion menu item is provided which converts the exec command to a shell command. This projects (non-editable, non-selectable) quotes around the string indicating the shell form is in use. `Command_Editor`

shell → exec Attempting to delete either quote, by pressing ⟵ backspace at the left-most point of the command string or pressing delete at the right-most point of the command string, will remove the quotes from both sides of the string and the projection of the shell. `Command_Actions`

exec → shell and shell → exec An intention is presented in either form which converts the command to the other form. This intention is always available when the insertion point is located within the command string.

`Command_ToggleForm`

These functionalities providing tree operations which simplify the editing experience of the projectional editor, providing natural editing reminiscent of a text editing environment.

(a) Command in exec form

(b) Command in shell form with shell projection (subsection 9.2.1)

Figure 9.1
Command editor projection of forms

9.1.2 Shell text generation

If a shell is required for a command, the command must be generated to text as a JSON array (Docker, 2018c). This is performed using text generation which splits the string on spaces and inserts the necessary delimiters (Figure 9.2).

`Command_TextGen`

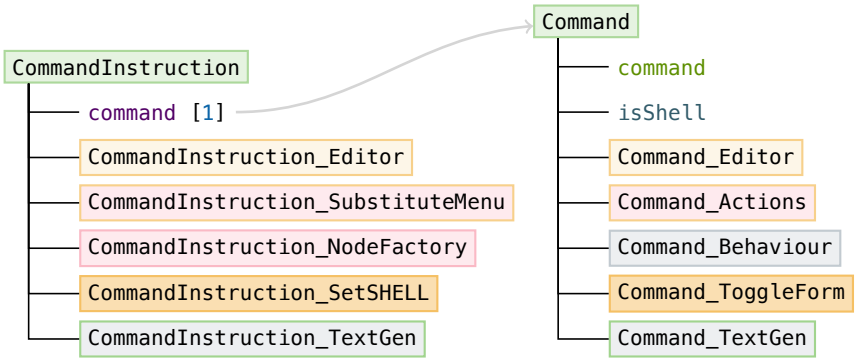
```
append
  {[ \\} \n { "}                                  Begin JSON array containing strings
  ${node.command.replaceAll(" ", "\", \\\\\n \"")}  Elements split by string delimiters
  {" \\} \n {]};                                  End final string and JSON array
```

Figure 9.2
Command shell form text generation

9.2 Command instruction

Instructions centring around a command which is to be executed by a shell extend `CommandInstruction`. This concept defines the required child `command` used to store the instance of the concept Command (section 9.1) to be executed.

`CommandInstruction`



(a) `CommandInstruction` hierarchy

(b) `Command` hierarchy

Figure 9.3
`CommandInstruction` implementation hierarchy

9.2.1 SHELL

The shell used for a command in shell form can be changed with the **SHELL** instruction (subsection 9.2.1).

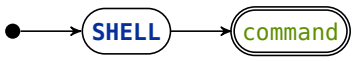


Figure 9.4
SHELL instruction diagram

Command shell projection To assist with editing a command in shell form, the shell which will run the given command is projected in the editor. This read-only model access queries the tree and returns command for the closest prior **SHELL** which would take effect if one exists, or returning the default shell command if there is no such instruction. Since the shell command prepended to the given command string is a projection of the relevant **SHELL** instruction's command, it always reflects the most appropriate value of the **SHELL** instruction.

9.2.2 Node factory

On creation of the command instruction, it is required that the `command` child contain an instance of a command. This is enforced by the language separately ensuring such a child is present and providing type system messages where this has been overridden. In addition to this, to make the editing experience smoother, a node factory is used to automatically create a child when the parent instruction is added to the tree, positioning the insertion point within the created command so that the user may begin typing a command string immediately.

`CommandInstruction_NodeFactory`

9.2.3 Change SHELL

To change the shell used, the exec form can be given the shell as the executable. For example, to run the command '`foo bar baz`' with `bash` instead of `sh`, the exec form of the command can be given '`/bin/bash -c foo bar baz`' which is output as text as '`["/bin/bash", "-c", "foo bar baz"]`'.

Alternatively, the shell can be configured with the **SHELL** instruction first, which changes the shell for all instructions which come after it. An intention is provided, available when the command in a `CommandInstruction` is in shell form, which creates a new **SHELL** instruction before the current instruction and moves the insertion point to the new instruction ready to be entered. This intention provides a natural method for editing an otherwise read-only projection of content in the editor.

`CommandInstruction_SetSHELL`

9.3 RUN, CMD & ENTRYPOINT

Three instructions extend the `CommandInstruction` concept, providing the ability to execute commands. Each of **RUN**, **CMD** and **ENTRYPOINT** take a command or part of a command to be executed during building an image or running a container.

RUN Executes a command at build time on top of the previous layer, saving the result as a new layer in the image.

CMD Sets a default command to be executed when the image is run. Only one **CMD** can take effect for an image (subsection 9.3.2).

ENTRYPOINT Sets the command to be passed the input command when running a container.

The simplicity of the implementation of these three instructions is testament to the implementation of `Command` and `CommandInstruction` being able to adapt to these semantic instructions and provide accurate text generation to the output file.

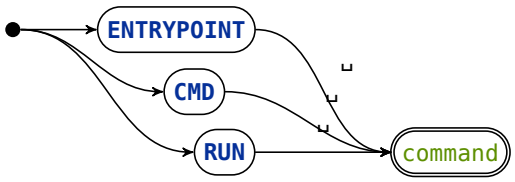


Figure 9.5
`RUN`, `ENTRYPOINT` and `CMD` instruction diagrams

9.3.1 Using CMD with ENTRYPOINT

When a container is run, the **CMD** instruction's command provides the command to be executed (Docker, 2018c). This command is provided as an argument to the entry point of the container, which by default is `/bin/sh -c` (Charmes, 2014). **CMD** sets default parameters for the container (Docker, 2018c), which can be overridden at runtime.

The entry point of the container can be changed with the **ENTRYPOINT** instruction, which provides the command given the parameters from the **CMD** instruction if one exists, or the runtime arguments if those exist. Parameters to the command given in **ENTRYPOINT** are discovered and used by the engine in the following priority order, from highest to lowest:

1. Runtime parameters, which could be passed from the command line invocation such as with `docker run -it name cmd`.
2. An orchestration engine providing a command, such as with the `command` key in the compose file for a service declaration (Docker, 2016).

3. The **CMD** instruction's command in the file if one exists, or the last **CMD** if more than one is found in the file.
4. None; **ENTRYPOINT** is used as-is.

Since only the last **CMD** is read, multiple **CMD**s throw a type system warning in Alembicue (subsection 9.3.2).

This means the value of the command cannot be known at build time because it can be changed later. In contrast, **ENTRYPOINT** is fixed at build time. Therefore, any parameters or flags *required* by the image should be provided in **ENTRYPOINT**.

9.3.2 Overridden CMD instructions

Only the last **CMD** in a file takes effect, with other instances of **CMD** instructions being ignored. Ignored **CMD** instructions are given a type system warning in the projectional editor from the **CMD** checking rules. This warning can be resolved with the generic quick fix intention on any ignored instruction, which when executed removes the instruction from the tree.

`check_CMD`

`fix_Instruction_Overridden`

While the type system check is specific to CMD, this intention to remove overridden instructions is written to be generic to any instruction which may be found to be overridden through data flow analysis of the file. Data flow analysis helps ascertain when certain instructions override the values of other instructions. This is an extension of existing tools for programming languages which can identify unreachable code — data flow analysis in Alembicue can be used to identify instructions which are reachable but have no effect on the resulting image or container. This is a powerful technique ensuring streamlined files where every instruction can have a noticeable impact on build time and complexity of the image layers, and reducing the likelihood of bugs by identifying issues which may have been overlooked.

10 Container configuration

With the environment of the image now configured (chapter 8), additional configuration can be set before or between execution of commands (chapter 9) or modification of the filesystem (chapter 11). These extra commands provide the ability to set the user which runs subsequent commands (section 10.1), add 'trigger' instructions to be executed in subsequent images (section 10.3), and configure the signal sent to the container on a subsequent orchestration message to stop (section 10.2). Much like the chapters that came prior, this chapter discusses these three instructions in terms of their syntax, editor, behaviour, and implementation in Alembicue.

10.1 USER

While the user of all commands defaults to root, it is possible to override this default using **USER** to configure the user and group.

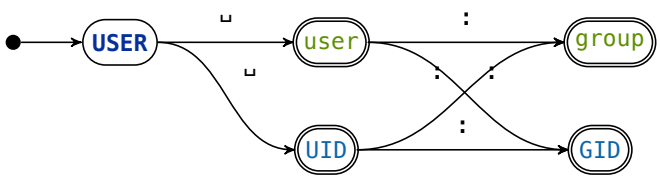


Figure 10.1
USER instruction diagram

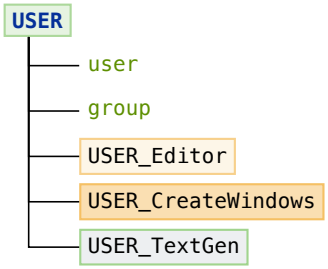


Figure 10.2
USER implementation hierarchy

10.1.1 User and group

USER takes one required parameter, `user`, and one optional parameter, `group`. Either parameter can be a **string** or **integer**, where the string is the name of the user or group, and the integer is the user ID or group ID respectively. No additional input is necessary to choose either data type (such as a conventional programming language using quote characters); instead, the type is implied, and the projection of each parameter denotes the implied type, with syntax highlighting for String or Number applied to the cell containing each parameter.

[>	{ user }	:	{ group }	<]

Table 10.1
USER editor

10.1.2 Creation of user

In a Unix-like container, the user and group given as parameters to the **USER** instruction are created if either does not exist in the inherited image. Therefore, no additional instructions are necessary to use a new user for subsequent commands.

However, in a Windows container, the user (and group if necessary) must be manually created before the user can be used. The operation to create the user can be generated from the parameters to the **USER** instruction. This automation is implemented as an intention which can be executed by the user with ⌥ alt + ⏎ enter. The intention adds a **RUN** instruction with a command to create a new user with `net user` (Figure 10.3).

USER _CreateWindows

```
node<Command> cmd = node                                  Get new command property
    .new prev-sibling(SHELL)                              Set shell before instruction
    .command.set new(<default>);                          Set new instance of command in instruction
cmd.command = "cmd /S /C";                                Set command to Windows equivalent
cmd.isShell = false;                                      Ensure command to set shell is not using previous shell
```

(a) Set shell for Windows

```
node<Command> cmd = node                                  Equivalent acquisition of command
    .new prev-sibling(RUN)                                Run command after setting shell
    .command.set new(<default>);
cmd.command = "net user /add " + node.user;               Create user
cmd.isShell = true;                                       Use shell set by previous instruction
```

(b) Create user in Windows

Figure 10.3 Automation of prerequisite instructions for `USER` on Windows

10.2 STOPSIGNAL

The **STOPSIGNAL** instruction defines the signal sent to the container when orchestration requires it to quit, to inform processes within of the impending shutdown of the container.

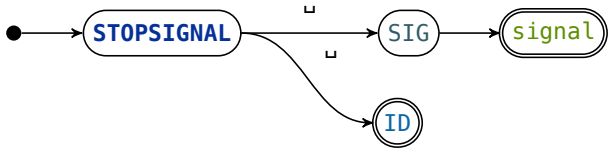


Figure 10.4 `STOPSIGNAL` instruction diagram

10.2.1 Completion

A signal can be provided numerically or as a string. Since the possible signals which can be sent are derived by the kernel of the container which is run, it is not possible to provide an exhaustive list of signals which could be sent.

IEEE's POSIX® defines a list of standard signals (Josey et al., 2004) which provide the opportunity for potentially relevant code completion. Various signals from this standard are incorporated into **STOPSIGNAL**'s projectional editor cell for the `signal` property as a menu part providing property values. This allows for the completion of common signals in kernels complying with this part of the standard.

STOPSIGNAL _Editor

10.3 ONBUILD

Instructions which would normally be executed as part of compiling the image can be delayed until the container is built from the image using **ONBUILD**.

Figure 10.5
ONBUILD instruction diagram

10.3.1 Trigger instruction

ONBUILD has one required child: a sub-concept of instruction (section 7.1). This contains the actual instruction, called the *trigger instruction*, which is to be executed when 'triggered' by a subsequent build of the next image based on the one containing this **ONBUILD** instruction.

When no instruction has yet been given for the trigger, the trigger instruction is set to an instance of `Instruction`. This provides the correct code completion and other functionality similar to that of a blank line in the document, while not being specific to any instruction or defaulting to any particular instruction instance.

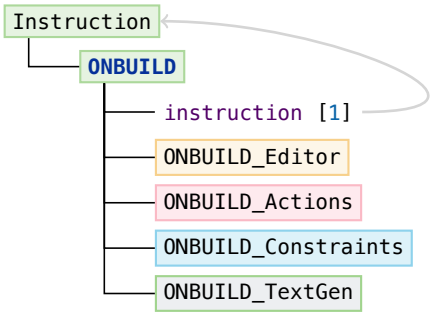


Figure 10.6
ONBUILD implementation hierarchy

10.3.2 Deletion action on child and parent

To correctly handle the deletion of the trigger instruction versus the **ONBUILD** instruction itself, a deletion action is defined which performs the scoping adjustments necessary to delete or request approval for deletion of one or the other.

`ONBUILD_Actions`

If a deletion is requested by the user, such as using ⟵ backspace actions, the defined action aspect checks whether a sub-concept of instruction is provided as an instruction child to **ONBUILD**. If that is the case, there exists an entered trigger, which should be deleted first (`node.detatch`). If not, a deletion approval is requested for the entire **ONBUILD** instruction, which if granted removes the instruction's node from the tree and replaces it with a blank line (Figure 10.7).

This custom action makes removal of nested instructions easier, without needing the user to pre-select a scope of nodes to be deleted. In a text editor, the instruction is written on one line (for example as **ONBUILD ABC** `xyz`), so using backspace to remove all characters to the right of **ONBUILD** would allow the insertion of a new instruction in its place. This deletion action imitates this

with the projectional editor nodes, performing the action first on the child of the node, then the node itself, providing a natural editing experience.

```
if (node.instruction.concept.isExactly(Instruction)) {
  if (node.approveDelete [in: editorContext]) { return; }
  node.replace with new initialized(BlankLine);
} else {
  node.instruction.set new(Instruction);
}
```

Figure 10.7 `ONBUILD` backspace action to handle deletion of contained trigger instruction versus the container `ONBUILD` instruction

10.3.3 Constraints for parent

`ONBUILD _Constraints`

The trigger instruction cannot be another **`ONBUILD`** instruction, or **`FROM`**, or a comment (Docker, 2018c). This is enforced with a parental constraint on the **`ONBUILD`** instruction.

Type constraints defines the concepts of the nodes permitted to be present as a child. By checking constraints throughout the user interface as editing is being performed, numerous abilities are restricted where constraints are in place, assisting code completion menus with the instructions which are permitted to be initialised in that location, as well as substitution menus and transformation actions. This integration between different aspects of the editor ensures syntax when combining multiple instructions, each with their own syntax that has already been checked.

11 Filesystem modification

The filesystem of the image can be modified using instructions to provide additional files and folders to the image from the current context. This is supported through a Path concept (section 11.3) — this provides filesystem navigation for **ADD** and **COPY** (section 11.4). For configuring the image, **WORKDIR** (section 11.1) and **VOLUME** (section 11.2) supplement filesystem interactions between the host and image or container respectively. This chapter introduces these instructions' syntax and implementation in Alembicue, alongside briefly covering their functionality to provide context to the decisions made in producing the language specification and projectional editor components.

The context is the directory structure in which the image is built.

11.1 WORKDIR

The working directory for subsequent instructions in the build phase is set with **WORKDIR**. Similar to the shell command 'cd', this instruction changes the directory used for relative paths within the image, such as a destination for a **COPY** (section 11.4). The instruction is the most basic of all instructions within the language, with one `path` property stored as a string.

The path constructs defined in the next section are not used for **WORKDIR**'s `path` string for two reasons:

- It is not possible to determine the state of the filesystem of the image before execution to know where the `path` is pointing to. Files and folders in the image filesystem are dependent on all previous instructions, including the base image used for building, and any commands executed previously which may create files, such as an installation using a package manager like `apt`.
- Unlike a local relative path which provides checking that the path exists on the filesystem, the **WORKDIR** path does not need to exist. If the path does not exist, it will be created.

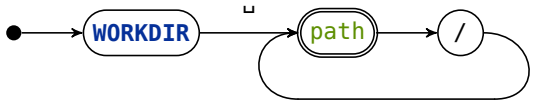


Figure 11.1
`WORKDIR` instruction diagram

11.2 VOLUME

A volume is a mounted path in a container. The image can define directories in the image's filesystem to be used as mount points for running containers of the image. The **VOLUME** instruction declares such paths to be mounted and stored outside the container. For more control over mount points and the location and type of the filesystem used for storing the volume in the host, container orchestration should be used alongside or instead of this instruction.

VOLUME takes path parameters representing the location in the image which requires mounting from outside the container, such as from a location on the host or a location in another container. Since this location is not within the realm of the image or container itself, it is not possible for the image definition to prescribe how the volume should be stored outside the container — this is the reason for the rudimentary nature of the **VOLUME** instruction.

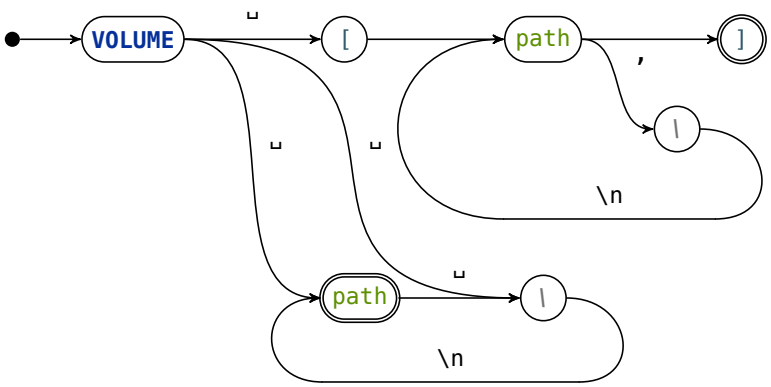


Figure 11.2
VOLUME instruction diagram

11.3 Path

A path is a string which specifies a location in a filesystem. The Path concept as implemented in Alembicue supports the use of paths as parameters for instructions, providing a projectional editor and text generation for path definitions, alongside behaviours to determine whether the path currently exists on the filesystem. There are two types of path implemented in the Alembicue language: local paths and remote paths.

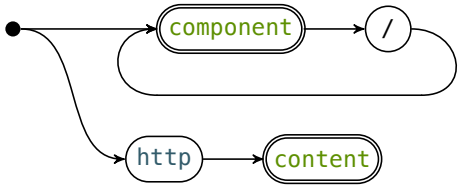


Figure 11.3
Path diagram

11.3.1 Local relative path

Local paths, realised with the `PathRelative` concept, represent a location in the filesystem of the current build context. Since the build context is the root of the filesystem tree provided to the image build process, all paths are relative to this location, hence the implementation name featuring the word 'relative'. These paths begin with a full stop indicating the current directory, then follows with any number of path components, each providing the name of a directory to follow, potentially with the last component providing a filename.

The `PathRelative` concept acquires much of its functionality from `BuildSourceProjectRelativePath` (Figure 11.5), a concept provided by Build Language. As well using languages to create functionality for which they were designed to represent, as discussed earlier in this report (section 5.1), the

individual concepts which are used to define the language can be used on their own. This helps simplify the implementation of various behaviours of the `PathRelative` concept.

Each path is defined by various components, each stored under the `compositePart` child. This optional child can store at most one part, which is another instance of a `PathRelative`. This creates a hierarchy of parts of the path which can be joined together to form an entire path. For example, the path `./foo/bar/baz` is stored using three `compositePart` children, where head stores the string of the current hierarchy position (`foo`, `bar`, `baz`) and tail stores the `compositePart` at the next hierarchy level down (Figure 11.4).

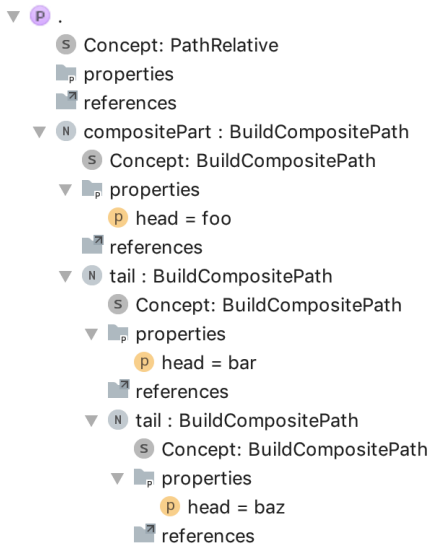


Figure 11.4
Node explorer displaying example `PathRelative` instance

Editor The projectional editor developed for a path mimic a normal string storing the path, with forward slashes delimiting path components. The `head` is the editable part of the projection containing the string of the current directory level, and the `tail` is the projection of the child nodes which each display their `head`, producing a projection of the entire tree.

To make editing smoother, transformations and actions are available in the editor. When the cursor is at the right-most point of the head, pressing `/` will create a new composite part and move the insertion point into the new path segment after the projection of the `/` character constant. To complement this, pressing `⟵ backspace` at the left-most point of an empty head will delete the composite part (removing the forward slash projection) and move the insertion point to the end of the previous head. These actions imitate the writing of a path in a normal text editor, by using the addition and deletion of the slash path delimiter, as would be expected with a normal path string, to control adding and removing child nodes in the tree.

`[-`	`^{ head }`	`?^`	`/`	`?AR%`	`tail`	`/empty cell:`	`[-`	`<<…>>`	`-]`	`%`	`-]`

Table 11.1
`PathRelative` editor

11.3.2 Remote path

A remote path points to a location accessible through HTTP. The Alembicue language supports such paths where applicable with the `PathRemote` concept. This provides the ability to download resources for use in a build context. Such paths are strings denoted by the prefix `http`, and point to a location which is downloaded with a GET request before being passed into the parameter where the path is given.

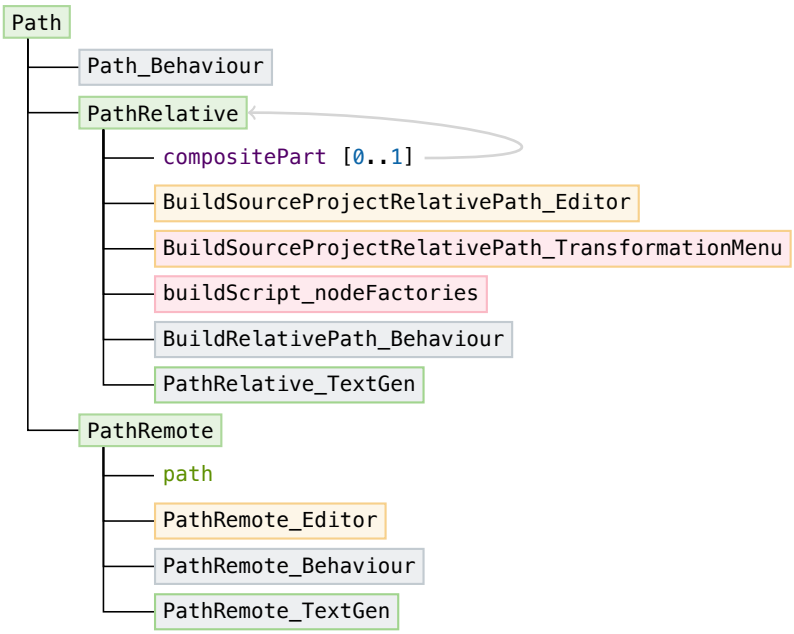


Figure 11.5 `Path` implementation hierarchy

11.4 `ADD` & `COPY`

To add files from a given path, the **`ADD`** and **`COPY`** instructions take the source path and provide the files to the image at the given destination path. Both instructions take a list containing one or more paths as `source` and a `destination` path string to place the files. Files and folders given in the path are required to be available in the build context (subsection 3.1.1), as they are copied into the resultant image and saved to disk as part of the image build process. While both instructions are similar, each has functionality that the other does not, as follows:

`ADD` With an **`ADD`** instruction, if a source path given is to an archive file, such as .zip or .tar, this archive is automatically extracted into the destination directory rather than being copied literally, removing the need to extract the archive manually using a **`RUN`** command.

`ADD` also supports remote paths as source paths which point to resources available over HTTP, which can be downloaded by Docker and copied (or extracted if an archive) into the destination. These automations simplify the addition of resources into the image.

`COPY` Using the **`COPY`** instruction can be regarded as a more literal copy from source to destination, without such automatic operations being carried out

like extraction or downloading. This also allows for one extra piece of functionality: the ability to copy files and folders from other build stages in a multi-stage build process (subsection 8.1.5).

As **COPY** is a similar instruction to **ADD**, the **COPY** concept extends the **ADD** concept, with additional aspects for its functionality (Figure 11.6).

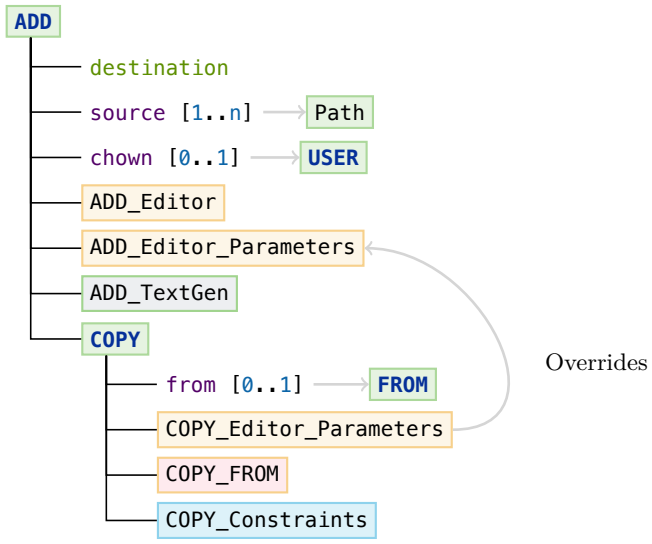


Figure 11.6
ADD and COPY
implementation hierarchies

11.4.1 Sources list

To make it clear what is being copied where, a vertical collection is used for storing the list, which is placed to the left of a projected right arrow pointing to the destination path. Where the list contains more than one source, the list is placed in grey brackets which help identify the list in the projection separately from the projection of the destination path, while simultaneously communicating that all the provided source paths will be copied to a single destination list.

Validation These paths are validated as existing on the filesystem, with red text used if the path does not exist at the current time, and dark purple if the path does exist. Black is reserved for path delimiters and not used to display the result of validation to distinguish between successful validation and no validation having taken place. Validation on the path name is provided as guidance only, since the lack of a valid file at the location is not a syntax error in the document itself, as the issue is located outside of the document. This means compilation cannot be prevented if the source does not exist as it may exist at a later date, or due to a different context location for the build.

Code completion for paths is provided, suggesting and completing names of files and folders currently existing in the project folder which by default is used as the context of the build. This helps the user provide the correct path to a location of their choice alongside validation of the path.

`COPY` local only The `COPY` instruction only supports local paths. To define this requirement in the language, a child constraint is defined on `COPY` which requires that children of the COPY concept are not instances of PathRemote. This removes remote paths from code completion in the sources list, prevents the typing of a remote path in the `COPY` instruction, and throws a type system error message if a remote path is otherwise somehow entered.

COPY
_Constraints

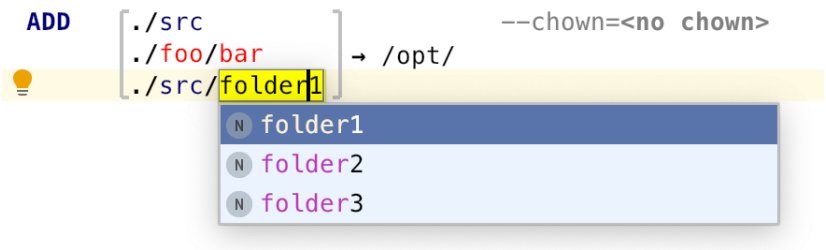


Figure 11.7
`ADD` sources list to be copied with path validation and completion for folder name

11.4.2 Change file owner

The `chown` command changes the owner of a file, such as with a `RUN` command. To simplify changing the owner for added files, parameters to perform a chown operation can be passed to the `ADD` or `COPY` instructions. This is incorporated into Alembicue by passing a `USER` instruction as a parameter.

On text generation, the parameters to the `USER` instruction are extracted from the node and printed to the output file as necessary for this instruction. Using the `USER` instruction shows a consistent editing experience, promoting the same editor component for the same case of providing a username and group name for the purposes of selecting a user. This demonstrates a benefit of a projectional editor, where the `USER` instruction keyword can be projected into the editor to guide the user into expected parameters, despite not including the `USER` keyword in the output file. In a regular text editor, the `USER` keyword would have to be parsed from the file and removed before the file could be used. This would be a potentially unsafe operation especially, in the case of a syntax error elsewhere in the document, where the `USER` keyword could be mistaken for a separate instruction instead of a parameter to `ADD` or `COPY` — not an issue that exists with a projectional editor notion.

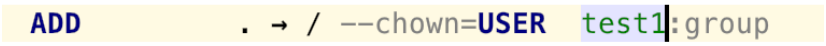


Figure 11.8
Performing 'chown' by passing `USER` to `ADD`

11.4.3 `COPY` from build stage

`COPY` overrides the editor component `ADD_Editor_Parameters` with its own parameters component to provide the additional parameter used to specify the build stage to copy from. This is appended to the list of parameters provided by `ADD` to the right side of the projection of the instruction's contents. The vertical list of parameters moves to the right as the source and destination increase in width. Once the width of the instruction contents becomes too wide, the parameters wrap to underneath the instruction contents.

`COPY _Editor_Parameters`

Table 11.2
`ADD` editor

Parameter for referencing The parameter to `COPY` to receive a reference to a build stage is provided as an editor component. This editor component is placed in the `ADD_Editor_Parameters` cell in the `ADD_Editor` which allows `COPY` to inherit `ADD`'s editor while providing its own additional customisations and functionality where necessary in an extensible fashion.

`COPY _Editor_Parameters`

Table 11.3
`COPY --from`

Parameter for referencing The main editable cell is the property (`from`) which accesses the reference stored by `COPY`. This is a model accessor (Figure 11.9) which makes reference to a model based on the input. If a numerical reference is given, the number must exist as a build stage. If a name is given, the name refers to the named build stage given after the 'as' in the referenced `FROM` instruction. The transformation menu presents the list of available nodes within scope for the user to choose from, providing completion for numerical and named referencing including swapping the number for a name in the case where a name is given for the stage number.

`COPY_FROM`

```
get{ (editorContext, node)->string {
  if (node.name.isNotEmpty) { return node.name; }          Use stage name if available
  String.valueOf(node.indexOfType());                      If stage is not named, use index of stage
}}
set{ (text, editorContext, node)->void {
  if (!text.matches("[0-9]+")) {
    node.name.set(text);                                   Set name of stage to name given if not numerical
  }
}}
validate{ (text, node, oldText, editorContext)->boolean {
  text.contentEquals(String.valueOf(node.indexOfType())) ||
     !text.matches("[0-9]+");
}}
```

Figure 11.9
Model accessor for `COPY`'s reference to build stage

Scope of reference The reference to a build stage for multi-stage builds is made to a build stage in the current file, denoted by a `FROM` instruction in the file. When providing a reference, scope constraints ensure ensure the reference exists and is available to be used for this instruction. This constraint on the `from` property is a reference scope provider, which provides a list of nodes that are within scope of the `COPY` instruction, to be checked when editing occurs. This list is created by looking at the current node's previous siblings at instruction level and providing any instances of `FROM` instructions to the reference. Restricting the search to previous siblings prevents the reference being made to future build stages declared in the file but which are not available yet.

COPY _Constraints

12 Metadata management

The two instructions introduced within this chapter do not directly affect properties of the built image nor execution in the container, but rather provide information for orchestration engines when running containers based on the image. These instructions can be ignored or replaced by orchestration at runtime — the intention is to provide helpful hints to software meant to aid with the running of container-based software. Providing the `MAINTAINER` of an image (section 12.1) is useful for organisational and responsibility purposes, and providing a list of network ports opened with the `EXPOSE` instruction (section 12.2).

12.1 `MAINTAINER`

The maintainer of the image can be set with `MAINTAINER`.

Figure 12.1
`MAINTAINER` instruction diagram

12.1.1 Parameter

This instruction takes one parameter, a name or email address of an individual or organisation representing the entity with maintenance responsibility of the image. This may be the author of the image, or a party responsible for its continued advancement. The maintainer given should be a contact point for queries and concerns regarding the image. RFC-822 provides a convention to follow for the value of this instruction's parameter, namely that the maintainer should be in one of the following two forms (Crocker, 1982):

	spec	*example*
phrase	`<addr-spec>`	`foo bar <foo@example.com>`
addr-spec		`foo@example.com`

12.1.2 Placement in file

`MAINTAINER` can be given at any location in the file, since it does not affect execution of any instructions preceding nor succeeding, and sets the maintainer for the entire file. This is one of few instructions which can be placed before flow issue the first `FROM` instruction in a file. The editor handles this instruction with even fewer restrictions than `ARG`, another instruction which can be placed prior to the first `FROM`. `MAINTAINER` is an excluded instruction in the checking rule for relative positioning of `FROM` in a file (Figure 8.6).

This is against the standard requirement that `FROM` be first in a file. For more details including implementation of this rule, see '`ARG` before `FROM`' (subsection 8.2.2).

12.1.3 Deprecation of instruction

Docker version 1.13.0 deprecated the **MAINTAINER** instruction (Vass, 2017) after discussion in 2016 regarding its usefulness given the more flexible **LABEL** instruction (Cormack, 2016).

Any uses of **MAINTAINER** are projected in the editor with the keyword strike through to indicate deprecation. A warning level message is displayed which highlights the issue to the user but does not block compilation, since the Docker engine still supports processing **MAINTAINER** instructions (Docker, 2018a).

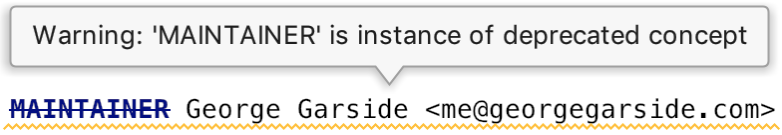


Figure 12.2
Warning: 'MAINTAINER' is an instance of deprecated concept

Deprecation implementation The mapping from an instance of the **MAINTAINER** instruction to a key-value (section 7.2) for the **LABEL** instruction is `maintainer=value`. Automatically mapping from one to the other is provided with the error intention `MAINTAINER_Convert`. This intention replaces a non-empty **MAINTAINER** instruction with the necessary key-value pair in a new **LABEL** instruction.

`MAINTAINER _Convert`

```
execute(node, _)->void {
  string author = node.author;                    Temporarily store current author
  node<KeyValue> label = node
      .replace with new(LABEL)
      .value
      .add new(<default>);                        Create new key-value pair child for replacement node
  label.key = "maintainer";                       Standard key equivalent for maintainer
  label.value = author;                           Apply author from maintainer to label
}
```

Figure 12.3
`MAINTAINER Replace MaintainerWithLabel` error intention

12.2 EXPOSE

The image can inform container orchestration of the ports a container which is based on this image requires with the **EXPOSE** instruction.

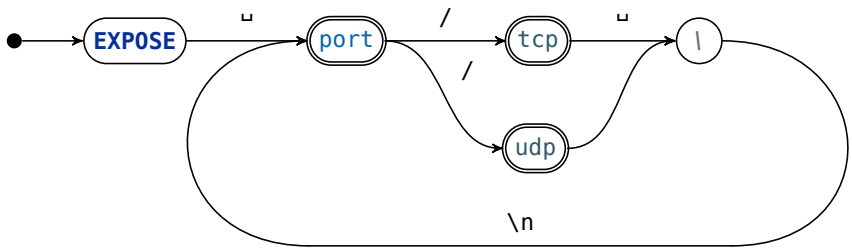
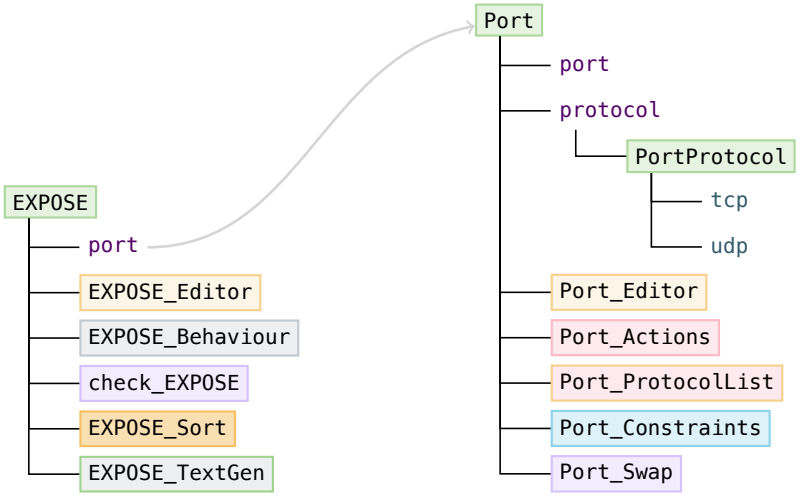


Figure 12.4
EXPOSE instruction diagram



(a) EXPOSE implementation hierarchy (b) Port implementation hierarchy

Figure 12.5
EXPOSE implementation hierarchy including Port & PortProtocol

12.2.1 Port & Protocol

To define network connectivity for a container, the image declares the ports opened with the **EXPOSE** instruction (section 12.2). The children of this instruction are instances of the PortProtocol concept, which encapsulates such a declaration. Each instance is composed of two properties, port and protocol.

A port number is an integer between 0 and 65535 inclusive. While port 0 is usually reserved, port 0 is seen as a request for an available port. The application is not granted port 0, but instead another port is allocated (Mitchell, 2019). A constraint on the property value for port ensures the value falls within the range required.

The two networking protocols supported are TCP and UDP. An enumeration encapsulates these options, providing onto complete for enumeration names and a name for projection in the editor, as well as enumeration values used in text generation of the concept.

The projection for a declaration of port with protocol mimics the text generation of such. The port number, followed by a forward slash, then the protocol name, is both how the text generation outputs the PortProtocol concept, and how the editor projection for the concept is read.

- Non-numeric characters are prohibited in the port property value cell, and typing such produces no output.
- Only enumeration names of protocols are permitted in the protocol cell, and code completion assists with the input of such. Also shown in the completion menu for protocol is an option for 'both' protocols; on selection, TCP is given as the protocol for the selected declaration, then the port number is used in a new declaration added as next sibling, with UDP given as the protocol.

`Port_ProtocolList`

Transformation menu To provide code completion (section 6.3) for `PortProtocol` declarations, a transformation menu is declared. A transformation menu provides augmented code completion for a given cell. A list of code completion options is generated from the enumeration (Figure 12.6), and choosing an option inserts the corresponding enumeration value into the tree at that location. Additional options are suggested where relevant, inserting multiple enumeration instances by duplicating the port declaration. This streamlines node insertions by avoiding the need to completely type the protocol name, or perform the multiple step process of creating an additional port and protocol declaration for an additional protocol on the same port.

```
enum/PortProtocol/.members.foldLeft(new arraylist<string>,{list<string> s, it =>
    s.add(it.name); s; })
```

(a) Generation of list of potential protocol values

```
enum/PortProtocol/.memberForName(parameterObject).getPresentation();
```

(b) Parsing of list for promotion to code completion

```
node.protocol.set(< TCP >);
node<Port> pair = node.new next-sibling(Port);
pair.port = node.port;    pair.protocol.set(< UDP >);
```

(c) Supplementary entry in code completion to duplicate the port declaration with its counterpart protocol

Figure 12.6
Presenting code completion list with transformation menu for `PortProtocol`

12.2.2 Sorting

`check_EXPOSE`

`EXPOSE_Sort`

`EXPOSE _Behaviour`

The list of port declarations should be sorted numerically increasing. The order is checked with the checking rule which provides a type system message at info level, and an intention is suggested to perform the sorting. This keeps ports in their standard order for preferable code style. The intention uses the behaviour aspect to determine intention eligibility.

```
if (this.port.isEmpty)
    { return false; }
int port = 0;
foreach declaration in this.port {
  if (declaration.port < port)
      { return true; }
  port = declaration.port;
}
false;
```

(a) `canSort` behaviour

```
if (this.port.isEmpty)
    { return; }
nlist<Port> ports =
    this.port.sortBy(
        { it => it.port; },
        asc
    ).toList;
this.port.clear;
this.port.addAll(ports);
```

(b) `sort` behaviour

Figure 12.7
`EXPOSE` behaviour aspect

Part III:

Integrated Development Environment

With the language syntax defined and incorporated into a projectional editor, increasing the usefulness of the editor is possible by creating an integrated development environment (IDE). This contains the language definitions and projectional editor, alongside other functionality such as version control system integration (section 14.4) and Docker orchestration. This part introduces and explains the Alembicue application functionality incorporated with the editor to provide a fully featured IDE.

13 Editor integration

The crucial part of the integrated development environment is providing an editor for files. The projectional editor designed and developed is integrated into the application to provide support for editing.

13.1 Document editor

Since the projectional editor is the core of the application, it takes up the most space in the application window. This area contains the functionality for performing the node operations discussed in implementation, such as adding and manipulating instructions.

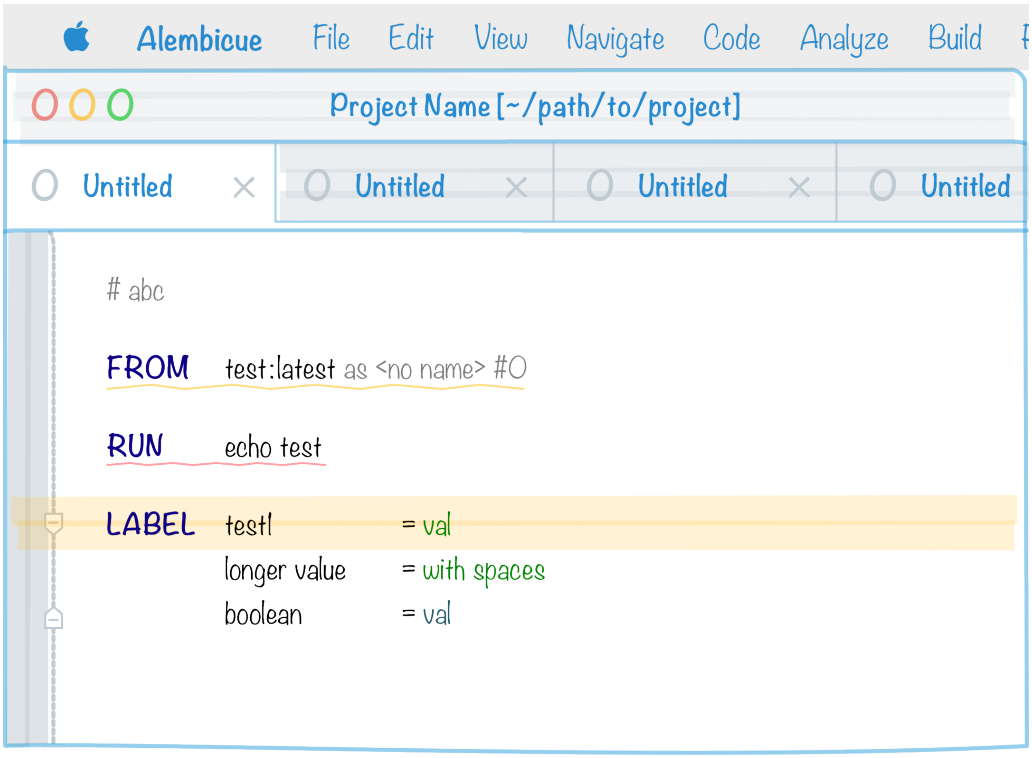


Figure 13.1
Mockup of editor interface

13.1.1 Current line

While the structure of the file projected in the editor is not entirely conformant to lines, text entered into cells are still horizontal in nature, so applying colour to the line containing the cell can be useful in highlighting where the insertion point is currently located and the scope of the edit. This highlight in Alembicue is a yellow background which is behind all other background colours, and extends into the gutter at the left of the editor.

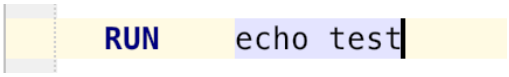


Figure 13.2
Highlight on line with insertion point

13.1.2 Node explorer

The underlying structure of the document is an abstract syntax tree. The node explorer provides the ability to see the tree structure as a hierarchy as it is recorded and stored without the projectional editor. To display the node explorer for a node, right-click or secondary click on a projection and choose 'Show Node in Explorer' from the context menu, or press ctrl + X. The node explorer shows the concept used for the instance of the node, as well as properties and children stored in the node, and references made to other nodes. Disclosure triangles show further nested information in each hierarchical level (besides clicking, → opens the triangle). This can be useful in determining exactly how the file is built up and confirming the correct information is being stored in the background, a useful debugging tool for confirming the file's values are received correctly by the build process.

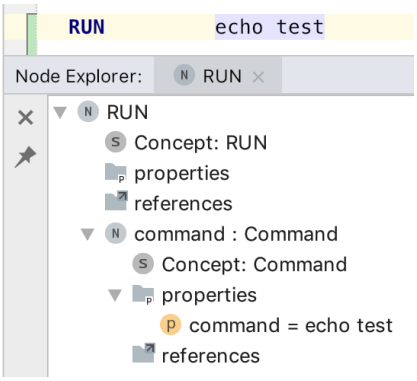


Figure 13.3
Node explorer example displaying contents of node for `RUN` instruction

13.2 Navigation

In the editor, navigation can be performed with the keyboard or with the mouse. Each cell projected by an editor aspect (section 6.1) has two properties to assist with supporting navigation, which are queried when navigation is attempted.

`selectable` A selectable cell permits navigation to place the insertion point placed within the cell. The default for cells is true, but this is overridden to false for all keywords, such as those to signify an instruction. By preventing selection of a keyword, the insertion point is ensured to be within the contents of an instruction, rather than in a fixed keyword, which assists with arrow key navigation by skipping the keyword (subsection 13.2.1).

`editable` An editable cell permits the typing of characters to be reflected in the cell. With this property false, all keyboard keys attempt to perform actions rather than inserting or modifying characters. For example, if the insertion point is placed in a selectable but not editable cell, and the ⟵ backspace key is pressed, a deletion approval is requested for the containing node rather than the character literally preceding the insertion point. Non-editable cells provide points to affix structure to for navigation, without being part of the underlying tree.

13.2.1 Arrow keys

Arrow key keyboard navigation is handled when the insertion point is located within a cell for a node's editor aspect.

↑ **and** ↓ Vertical navigation moves between vertical collections, such as that of a list of key-value pairs if present, then when this list is exhausted, moving between instructions. This navigation still ensures the destination cell to be navigated to is selectable per the `selectable` property.

← **and** → Horizontal navigation attempts to keep the insertion point within the current line (subsection 13.1.1) of the file, first moving the insertion point one character left and right respectively, then once the string of text in the cell has been exhausted in a particular direction, moving to the closest cell to the left or right in line with the current cell.

The destination cell where the insertion point is moved to must have the `selectable` property with a value of `true`. Where there are no more cells to either side, wrapping is performed, looking for the next selectable cell in the line above or below, such as a preceding or subsequent instruction's contents. Having the instruction keyword override selectable to be false makes this horizontal navigation between instructions more efficient by skipping the keyword.

Using ⌥ alt alongside the horizontal arrow keys, navigation with the arrow keys can be augmented. When the insertion point is placed within a selectable cell containing multiple words, ⌥ alt switches horizontal navigation from character increments to word increments. When the end of a cell is reached, ⌥ alt switches horizontal navigation to cell increments, and places the insertion point at the furthest end of the cell from the direction the cell has been reached from, such that subsequent ⌥ alt-augmented horizontal navigation is also incremented by cells, skipping the current content of any cell regardless of the number of words. This increases efficiency of moving through longer files while still iterating through every selectable cell in the projection.

Words are defined in this context as being strings of any characters separated by spaces.

13.2.2 Mouse clicks

Clicking with the mouse in the editor moves the insertion point to that location if the underlying cell has the selectable property. If not, the closest cell which has the selectable property, with horizontal preference, is selected for the insertion point, which is placed at the horizontally closest location in the cell. This mimics a text editor by being able to move the insertion point to the end of a line of text by clicking anywhere to the right of the currently entered text on the line, making the editor feel more natural for appending text.

13.2.3 Selection

More than one cell can be selected at once, including across nodes. The range of a selection can increase or decrease up and down the hierarchy, and to previous and next siblings. Keyboard shortcuts are available for creating a selection, which are as follows:

⌥ alt + ↑ Expand selection up hierarchy to words, cells, parent cells and parent nodes (Figure 13.4).

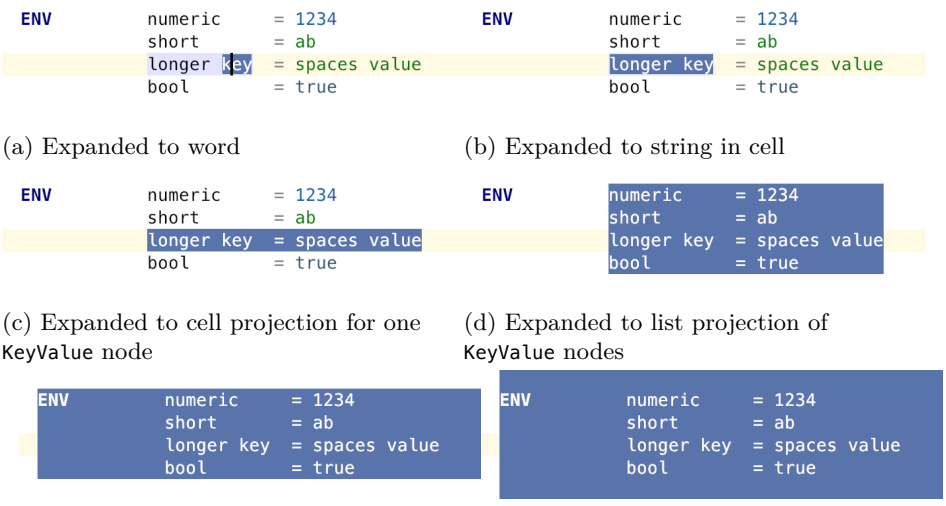
⌥ alt + ↓ Shrink selection down hierarchy back towards the original single insertion point from which the selection was expanded.

⇧ shift + ↑ Expand selection to previous siblings at current hierarchy elevation reached with ⌥ alt + ↑ or shrink selection to subsequent siblings at current hierarchy.

⇧ shift + ↓ Expand selection to subsequent siblings or shrink selection to previous siblings at current hierarchy.

These four commands can be thought of as moving a second selection point around beside the insertion point and performing a tree-based selection between the two points, where ⌥ alt moves the selection point up and down the hierarchy and ⇧ shift moves the selection point along siblings.

Selections in the abstract syntax tree are required to be contiguous, such that no siblings can be skipped in a selection. Therefore, this keyboard-based approach to selection manipulation, and the actions available with this keymap, are able to produce any valid selection range and scope, for further manipulation of the contents of the selection using other actions, such as ⟵ backspace for deletion, or moving (subsection 13.2.4).



(a) Expanded to word

(b) Expanded to string in cell

(c) Expanded to cell projection for one `KeyValue` node

(d) Expanded to list projection of `KeyValue` nodes

(e) Expanded to projection of `ENV` instruction containing children

(f) Expanded to file containing list of instructions

Figure 13.4
Steps of repeated invocations of selection expansion keyboard shortcut

13.2.4 Moving nodes

Nodes can be moved among their siblings with ⇧ shift + ⌘ cmd + ↑ or ⇧ shift + ⌘ cmd + ↓. This moves the currently selected node to the location of its previous sibling or subsequent sibling respectively. If there is no current selection, a minimal selection is made until a list can be identified for the sibling operation. If the end of the list has been reached, but another list supports children of the same type in the same hierarchy, the move is performed removing the node from its current list and appending or prepending it to the destination list. For example, a key-value pair can be moved from the start of an **ENV**'s list to the end of a previous **LABEL**'s list by moving the insertion point to the first key-value pair child of **ENV** and pressing ⇧ shift + ⌘ cmd + ↑.

13.2.5 Duplicating nodes

Nodes being projected in the editor can be duplicated by pressing ⌘ cmd + D. This copies the node, along with all its properties and children, and adds the copy as a subsequent sibling to the current node.

Initiating the shortcut with no selection duplicates the node most relevant for duplication: where the node is part of a list of other nodes and accepts a subsequent element. For example, a child node filling a single optional child declaration in a concept (`[0..1]`) will not be duplicated, because the maximum allowed is 1, so the duplication algorithm looks up the hierarchy until an ancestor is reached where there are n children allowed for a child declaration (`[0..n]` or `[1..n]`), then duplicates this node.

Duplications are also permitted with pre-selected nodes. Where a selection has been made (subsection 13.2.3), initiating a duplication will check whether the selection permits further nodes using the same checks as without a selection, then duplicates the entire selection of nodes. For example, selecting items at indices 1 and 2 in a list of 4 items, then duplicating with ⌘ cmd + D, will result in a list of $[0, 1, 2, 1, 2, 4]$ (where these number represents the index of the original item).

This allows for very fast tree mutations as all children are copied to the newly created node. A very useful example of duplication is in the source list for an **ADD** instruction — with any part of a path selected, ⌘ cmd + D will duplicate the path for a second source including all path components to be edited.

14 Project integration

Alembicue has the ability to create and manage different projects containing sets of files. These projects store files independently in different locations on the filesystem. This chapter documents the functionality of Alembicue regarding wrapping the editor in the full application interface including the dialogs for creating new projects and files in those projects (section 14.3), as well as version control system integration (section 14.4).

14.1 Out of box experience

The out of box experience (OOBE) is the experience presented on the very first launch of the application. This must be crucial to guide the user into the launch experience given in subsequent launches, without too many preliminary steps. The Apple Human Interface Guidelines are clear with onboarding guidance.

> *Avoid asking for setup information up front.* People expect apps to just work. Design your app for the majority and let the few that want a different configuration adjust settings to meet their needs. [...] If you must ask for setup information, prompt for it in-app the first time, and let users modify it later in your app's settings. (Apple, 2019a)

For providing such an experience, the only difference between OOBE and subsequent launches is one modal, asking whether settings should be imported from a location, or to set up as new. Settings can be exported from Alembicue with File › Export Settings... and imported with this OOBE modal or later with File › Import Settings.... If a new major version of Alembicue is released, this modal will automatically detect an older version's settings if stored in the default location and offer to import them without needing to manually specify a location. For other settings imports, it is necessary to choose a location of the settings file to import them. Once settings are imported, or the modal is dismissed indicating the user wishes to proceed with setting up as new, the standard Alembicue launch procedure begins.

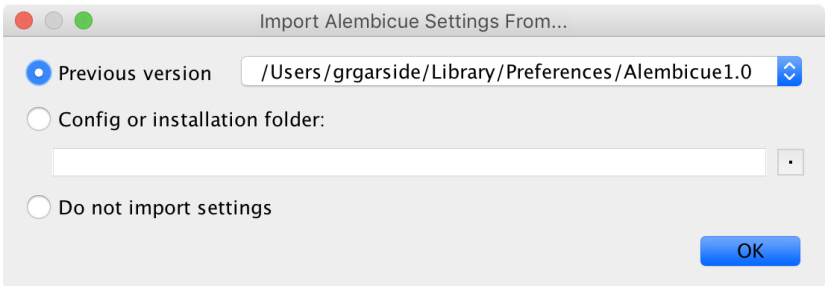


Figure 14.1
'Out of box experience' modal to import existing or exported settings

14.2 Welcome experience

If there is no current project open, the first window displayed by Alembicue is the Welcome dialog (Figure 14.2). This dialog displays the Alembicue logo, alongside the application name and version information.

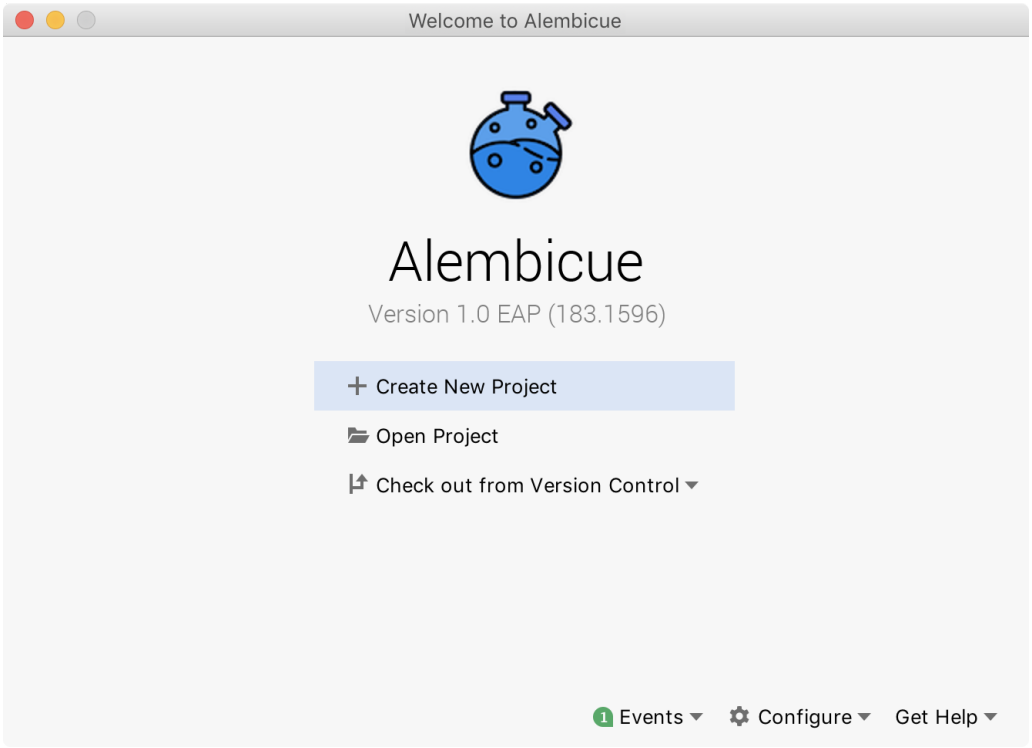


Figure 14.2
Alembicue IDE welcome dialog

14.2.1 Options

Three main options are presented for the user to choose from:

Create New Project Opens the new project wizard (Figure 14.4) to create a new project (section 14.3). This is discussed further in the next section.

Open Project Open the filesystem browser to find a project in the filesystem. Support local and remote destinations, any project whose files can be viewed in the operating system file browser can be opened directly through Open Project.

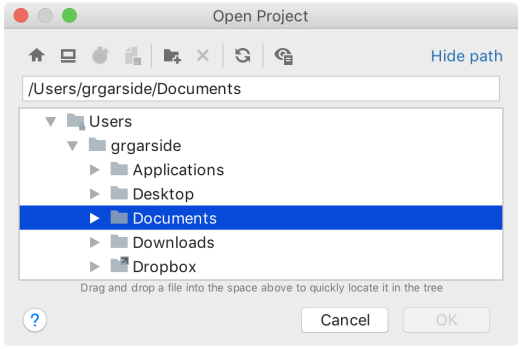


Figure 14.3
Alembicue IDE open project folder browsing dialog

Check out from Version Control Opens a dialog to clone a project (Figure 14.5) from a version control system repository by providing a URL source and filesystem path destination. For projects whose contents are tracked by VCS but which already exist on the filesystem, the normal filesystem browser accessible through Open Project should be used, and version control integration is automatically enabled on the project.

14.2.2 Additional options

Three additional options are also shown at the bottom of the dialog:

Events Additional information relevant to the current state of the IDE, such as information regarding available updates and application configuration, is accessible with the Events menu.

Configure Changing preferences, importing and exporting settings, and configuring the Java virtual machine can be performed using items in the Configure menu.

Get Help Additional support, including links to the Alembicue website (section 16.3) and extra tips for using the IDE, using the Get Help menu.

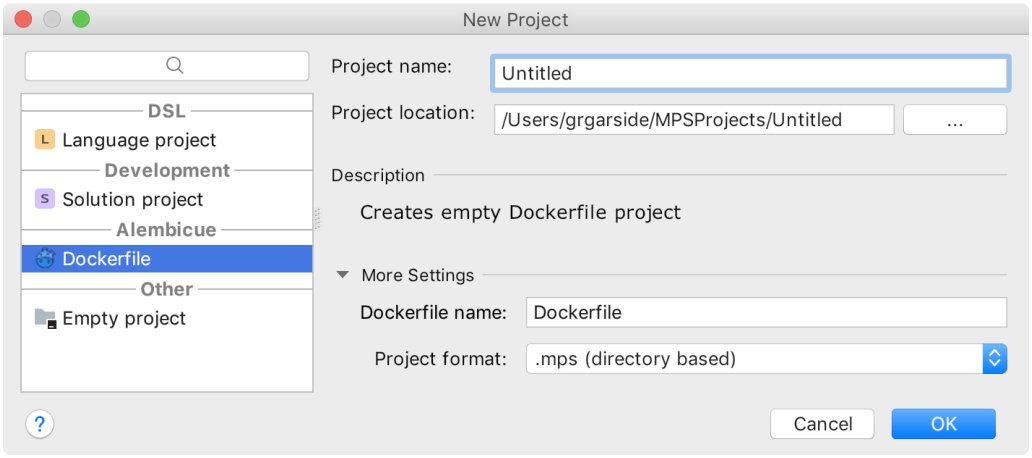


Figure 14.4
Alembicue IDE new project wizard

14.3 New project wizard

On creating a new project, either from the welcome dialog (section 14.2) or from File › New › Project…., the New Project wizard is opened (Figure 14.4). The wizard aims to simplify creating a new project by taking parameters to common options and generating a project which meets the requirements given.

14.3.1 Project parameters

For creating an Alembicue project containing a Dockerfile, the wizard takes the following parameters:

Project name
: The name of the project used to refer to the project in the future, and by default used as the directory name to store the project. Project name helps disambiguate projects in the recent projects list to open the project in the future, and to refer to the project in the cases where multiple projects are open at once.

Project location
: A location on the filesystem to store the project. Folders not currently on the path will be created, before the project is created within. This can be an empty folder or a folder currently containing files, the latter of which is common in cases where existing software should be containerised — any existing files are not disturbed and Alembicue can work alongside.

Dockerfile name
: The name of the Dockerfile to be saved to disk. By default, a Dockerfile has the name 'Dockerfile' (no extension), but in cases where multiple Dockerfiles may be in a project, it can be helpful to provide a name here to avoid confusion later. This name can be changed at any point using Refactor › Rename or ⇧ shift + F6 on the file.

Project format
: For compatibility with old systems, it is possible to change the format for saving the abstract syntax tree to disk from the default `.mps` format (XML contained in a directory) to `.mpr` (serialised XML concatenated into a single file). Choosing OK on the dialog creates and opens the project. If a project is already open, the user is prompted whether to replace the currently open project with the newly created project or to open the new project alongside the existing one.

14.3.2 Creation steps

In the creation of a new project, the wizard carries out the following steps. These steps could be performed manually without the wizard, which is just there to automate the process given the parameters from the user. For automation, these steps are carried out as a post-startup activity, delegated until model access is available (giving control first to the language workbench (subsection 3.2.3) to initialise the platform on the specific operating system).

1. Create a new empty project in the filesystem location given by the user in the wizard, with the model root set to `models/`. This sets the location for all files to be created using the language to be placed in this folder by default.
2. Create a new module in the project, with the generator output path set to `source_gen/`. This sets the text generation aspects to place their output in this folder by default when 'make' is performed on the project containing Alembicue files.
3. Create a new model in the module, with model (and therefore module) dependency on the `com.georgegarside.alembicue` language, available with the language definitions built into the environment.
4. Create a new root node in the model, of concept File (section 7.3). The File concept is marked rootable, which indicates the concept can begin a new abstract syntax tree. On the creation of a new root node, a new tree is generated ready for children to be appended.
5. Set the `name` property of the file to the parameter given in the creation of the project. By default, the name is set to 'Dockerfile'.
6. Add a new blank line to the file by creating a new instance of the BlankLine concept and adding the new node as a child to the new File instance.
7. Navigate the editor to the newly created file, giving it focus, ready for the user to begin editing. The insertion point is placed on the first line of the new file, with the context assistant shown (section 6.6).

14.4 Version control system

Alembicue integrates Git and Subversion version control systems at both an application level and directly with the editor.

14.4.1 Import from repository

Existing projects stored in a remote version control system can be downloaded and imported into Alembicue including all their revision history. This process is referred to as 'cloning' the repository. To begin the process, choose 'Check out from Version Control' in the welcome dialog (Figure 14.2). By providing a URL to the repository and a local directory to store the project, the repository located at the URL can be downloaded into the local directory, opened in Alembicue and automatically configured for integration of the version control system used. This allows you to get started with the project really quickly, whether as a tutorial project to follow along or in a multiple developer environment where a project has already been set up such as for production.

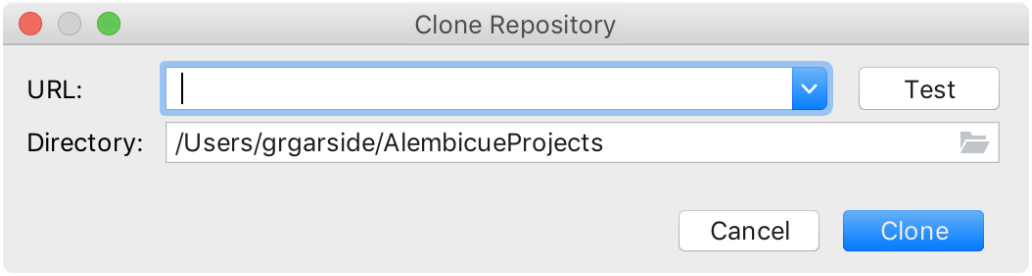


Figure 14.5
Version control system clone repository dialog

14.4.2 Code change hints

In a regular text editor, changes lines of text are marked as such by the version control system and can be presented to the user within the text editing environment, such as using a mark in the gutter. This usually makes it easier to see what has been changed in the working copy since the last commit.

To enhance this functionality, hints are provided throughout the language definition to suggest what has been changed, rather than just where the change has occurred. For example, the EXPOSE instruction provides VCS hints for changing, adding, and removing, for both ports and protocols (Figure 14.6).

Markers These hints are presented as coloured markers in the gutter of the editor. This makes comparing what has been changed in the editor in real time intuitive, without having to identify changed lines and then secondarily check what has been changed within.

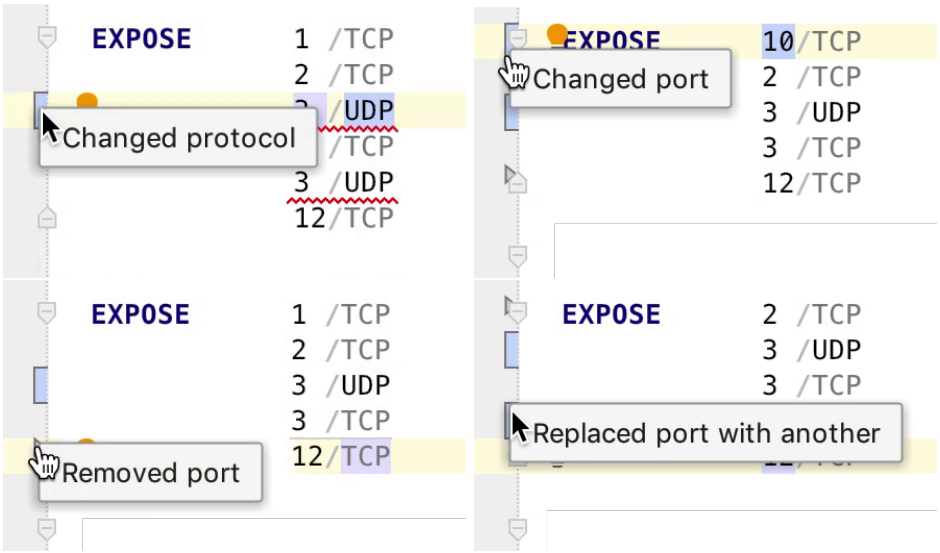


Figure 14.6
Version control system change hint examples shown in editor gutter

- Hovering a mark displays a brief summary of the changes contained within, and also highlights the cells in the projectional editor which represent the change (Figure 14.6). The colour helps identify the type of change that has taken place, with green for addition, blue for modification, and grey for removal.
- Clicking on a mark in the editor presents a popover with the previous revision of the code displayed before the change (Figure 14.7). A toolbar is shown in the popover which contains the additional functionality of navigating to the ↓ next change and ↑ previous change, as well as providing the ability to ↺ rollback current changes to the previous revision, ⇄ compare the changes using the model viewer (subsection 14.4.3), and ⧉ copy previous revision to the clipboard.

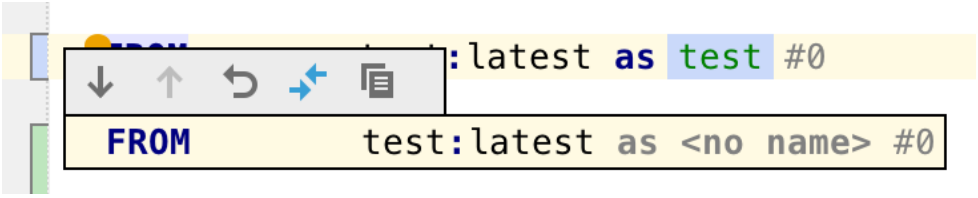


Figure 14.7
Version control change popover showing snippet of previous revision

14.4.3 Model comparison

A common feature used in a version control system (VCS) is the ability to compare one file with another. It is regarded that VCS are used with text files, due to the nature of how a VCS tool will 'record changes and determine conflicts on a line-by-line basis' (Ernst, 2012). This is against the nature of projectional tooling which does not have a direct mapping to a line concept, and therefore would be generally difficult to view changes and resolve conflicts working with the underlying tree structure as stored in XML text.

A possible solution is to use a VCS to track the output file, generated by Alembicue for use with Docker, for comparing changes between revisions and resolve conflicts. However, this means one looks at the output file to compare changes rather than using the projectional editor, a lesser viewing environment, and resolving conflicts would then need to be done by editing the output text file too, a lesser editing environment. The output text file should not be the primary storage method for information preservation long-term — the abstract syntax tree stored by and edited with Alembicue should be the focus of tooling.

Therefore, a custom viewer for changes tracked with a VCS would be optimal in assisting with version control of the abstract syntax tree.

Model viewer Since lines in a projectional editor are somewhat of a deception (subsection 13.1.1), it would be disingenuous to display the underlying abstract syntax tree as the changes made by the user to a file, since it is the projectional editor that is in use.

The model viewer provides this VCS-based comparison of projectional editor files in Alembicue. Providing the functionality of a 'diff' on the two files, the model viewer is displayed within the commit and diff windows. It is possible to use the model viewer to view changes in 'diff' views (such as changes between commits), pre-commit (changes between working copy and latest commit) and from any two files in the repository currently or in history.

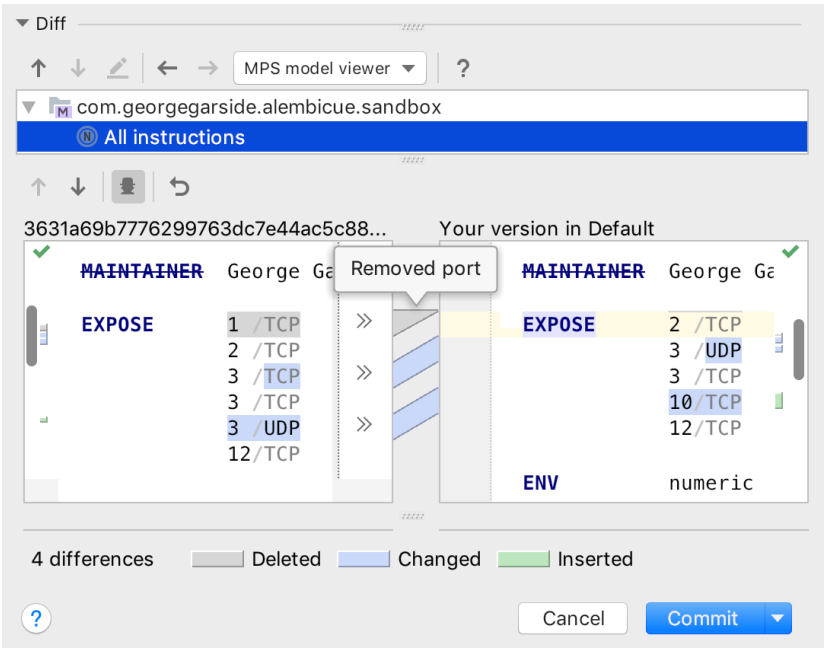


Figure 14.8
Alembicue model viewer displaying changes between working copy and last commit

Part IV:

Review

15 Testing

Alongside informal immediate developer testing of added components as part of development, two forms of formal testing were used: unit testing and integration testing. Unit testing (section 15.1) was run automatically and aims to identify problems with the language structure and aspects used to define the editor. Integration testing (section 15.2) was performed by the developer following test plans and incorporated regression testing towards the end of the project. Testing is an important component of software engineering by attempting to identify issues with individual code or the resultant behaviour from a combination of various aspects. The use of formal testing provided verifiable testing procedures to each instance of a test which could then be reproduced at a later date or by a different party. Both forms of testing are discussed in this chapter, including a summary of the tests that took place.

15.1 Unit testing

To test the language definition, projectional editor and other aspects, automated unit tests were run after each change was made to the code and at each release. Each test is defined by a name and description, identifying the test and what is being tested; the state of a portion of an abstract syntax tree, represented in the projectional editor alongside a recorded insertion point position; steps to be executed on the tree as part of the test, including keystrokes and actions by IDs; and the final state of the portion of the tree (Figure 15.1).

```
Editor test case FROM_Latest
description: Latest tag on image
before: <cell FROM image[:<no tag>;@<no digest>] as <no name> #-1>
result: <cell FROM foo:latest as <no name> #-1>
code:
  type "foo"
  press keys <any>+<VK_TAB> ;
  invoke action by id: jetbrains.mps.ide.editor.actions.Complete_Action
  press keys <any>+<VK_ENTER> ;
```

Figure 15.1
Example unit test
`FROM_Latest`

15.1.1 Running tests

All tests are run in a new minimal instance of Alembicue which is started without a GUI for the autonomous running of the tests. Each test can succeed or fail, depending on if any test prerequisites are not met, any constraints on the test fail, or the state of the tree at the end of the test does not match the given end state. Output is generated from the tests to summarise the results.

15.1.2 Test summary

All 36 automated tests passed by the end of the project, indicating that the components tested successfully carried out the task they were tested for.

Table 15.1
Unit tests

#	Aspect	Behaviour tested	Result
1	`BlankLine`	Transformation to instruction using fully entered name	✔
2	`BlankLine`	Transformation to instruction using partial entered lowercase name	✔
3	`BlankLine`	Creation of comment using transformation	✔
4	`BlankLine_Editor`	Deletion of blank line deletes current line with insertion point	✔
5	`BlankLine_SubstituteMenu`	Prevention of blank line from being inserted as instruction in the file	✔
6	`BlankLine _TransformationMenu`	Context assistant presented with correct options where circumstances meet requirements	✔
7	`Comment`	Approve deletion of comment replaces with blank line	✔
8	`KeyValue`	Entering and generation of key value pair	✔
9	`KeyValue_Editor`	Backspace from first position in value cell to last position in key	✔
10	`KeyValueInstruction`	Value is required and shows error when missing	✔
11	`KeyValueInstruction_Editor`	Creation of new key value pairs on return key to imitate new line	✔
12	`KeyValueInstruction _SubstituteMenu`	Prevention from being manually inserted as an instruction in the file	✔
13	`KeyValueOptionalInstruction`	Value is optional and does not show error	✔
14	`KeyValueOptionalInstruction _SubstituteMenu`	Prevention from being manually inserted as an instruction in the file	✔
15	`ONBUILD`	Supports necessary child instructions	✔
16	`ONBUILD_Editor`	Approve deletion of child instruction separately to parent instruction	✔
17	`ONBUILD_Constraints`	Prevent incorrect children from being created	✔
18	`USER`	Supports setting user and child	✔
19	`FROM`	Supports code completion for setting latest tag	✔
20	`Command`	Creation of command in exec and shell form	✔
21	`CommandInstruction`	Inclusion of command in instruction form	✔
22	`CommandInstruction _SubstituteMenu`	Prevention from being manually inserted as an instruction in the file	✔
23	`ADD`	Filesystem validation of path given exists on system	✔
24	`COPY`	Referencing to build stage inside and outside of scope	✔
25	`COPY_Constraints`	Ensure reference to `FROM` instruction is in scope	✔
26	`COPY_Constraints`	Prevent remote paths being provided	✔
27	`PathRelative`	Creation and removal of path segments using keyboard shortcuts to provide inputs to the editor	✔
28	`PathRemote`	Appropriate recognition of remote path instead of relative path for URL	✔
29	`EXPOSE`	Correct acquisition and display of enumeration for protocols	✔
30	`Port_Constraints`	Ensure port number given is within range	✔
31	`Instruction_Keymap`	Ensure creation of new blank line from insertion point position in child cells of previous instruction	✔
32	`Instruction _SubstituteMenu`	Restrict creation of instructions where lack of build context does not permit creation of such	✔
33	`Instruction _TransformationMenu`	Prevention from being manually inserted as an instruction in the file	✔
34	`CommandInstruction _NodeFactory`	Creation of child command alongside creation of instruction	✔
35	`Instruction_NodeFactory`	Creation of child key-value pair alongside creation of instruction	✔
36	`File_NodeFactory`	Creation of first blank line instruction in new file	✔

15.2 Integration testing

During development, new commits made into the version control system (section 5.3) included a test plan (Figure 15.2). This was written by the author to detail the steps which were taken to test the introduction of that specific commit on top of the existing code. Having already carried out these tests on integration, these test plans were followed manually once again as part of regression testing to ensure nothing released in an earlier commit had been broken by a later commit. Test plans are recorded on the project Phabricator in the Differential (code review) module, thus the test plans are referred to as Differential test plans.

In Differential, 75 revisions were recorded, of which 22 contained a test plan. These test plans were manually performed on the version of Alembicue ready for release (1.0.0 EAP) (section 16.4). All such test plans were passed successfully.

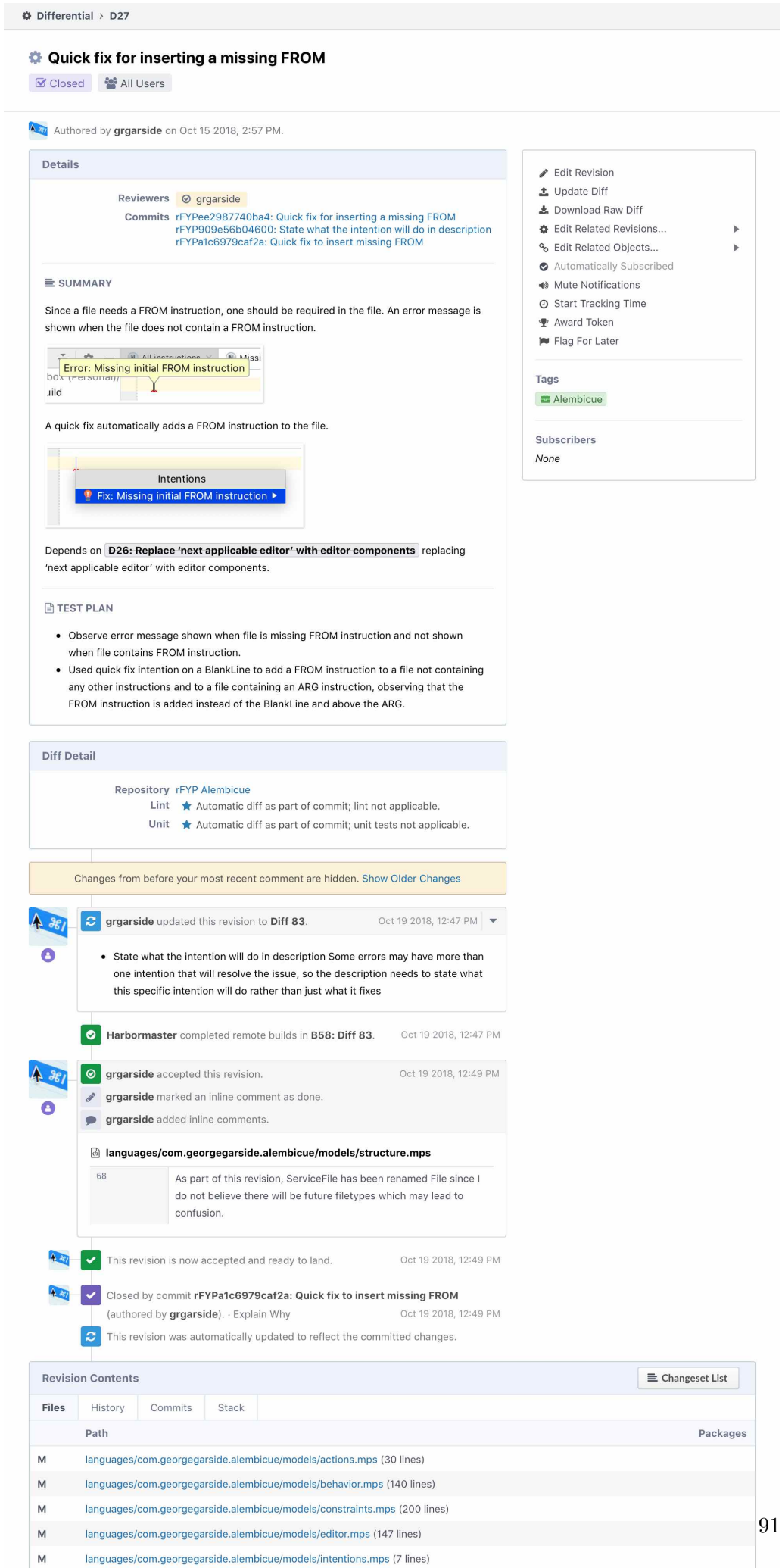


Figure 15.2 Phabricator Differential revision code review test plan

16 Release

To build Alembicue and its language as a fully functional IDE required some steps (section 16.2), and to release the software to be downloaded required more steps (section 16.3). This chapter documents the steps required and completed to take the language definition and IDE development to a released piece of software, including multiple versions (section 16.4).

16.1 Name

The name 'Alembicue' is a portmanteau of 'alembicate' and 'cue', where the idea of production and transformation from 'alembicate' refers to the manner in which abstract syntax trees are transformed and text files produced, and the idea of producing such text files from the 'cue' to articulate the tree with text generation in response to a request.

> **alembicate**, *v.* 1627. transitive. Chiefly figurative: to produce, refine, or transform (an idea, emotion, etc.) as if in an alembic. Cf. ALEMBICATED adj.
>
> **cue**, *n.*[2] 1553. 1c. A stimulus or signal to perception, articulation, or other physiological response.
>
> Oxford English Dictionary (1989).

16.2 Compilation

From language definition to standalone IDE application, there are a number of steps involved. These steps are documented in the following subsections. Compilation is based on Apache Ant, a framework for automating the building of applications from source code by specifying stages to be executed and pre-requisites required for those stages. MPS provides a custom projectional domain-specific language to interact with Ant, called 'Build Language' (`jetbrains.mps.ide.build`). This language is used as an abstraction of an Ant build script written with XML — the language's generator aspect transforms the declarations and projectional code to instances of concepts from MPS's projectional XML, which text generation uses to write out XML for Ant to ingest. Ant is used for development builds of the Alembicue language (subsection 16.2.2), and distribution builds of the Alembicue application package containing the language and integrated development environment (subsection 16.2.3).

16.2.1 Model checker

Before the language can be made, the model checker verifies that the language is valid using automated rules. These rules ensure the project and language structure is correct, that each reference target exists in the resultant tree after all tree operations, and that all language constraints and type system checks pass.

16.2.2 Build for development

The development build of the language includes the following steps:

1. The Alembicue language is compiled from the languages folder using the model definition of com.georgegarside.alembicue.mpl as the starting point to find and compile all language files for each aspect (section 6.1).
2. Resource files are loaded and added to the output. This is used for iconography, providing images used throughout the language, for example in the project wizard when selecting the language (section 14.2).
3. Dependencies to the Alembicue language are imported to the build and extracted into the output. Languages required for the compilation of Alembicue are jetbrains.mps .baseLanguage, jetbrains.mps.build and MPS.Editor.

With the language compiled, it is integrated into a 'plugin', which provides a wrapper to the language suitable for providing the projectional editor components into a non-projectional environment. This is performed as follows:

1. A blank plugin is created to set the metadata, including version, build number, and date.
2. Dependences to the Alembicue projectional editor and IDE enhancements are extracted, which includes `mpsStandalone`, `mpsVcs`, `mpsMakePlugin` and `mpsContextActionsTool`. These dependencies are provided by the MPS language workbench providing the base for the plugin to be developed on top of, thus providing packages which Alembicue is based upon.
3. The built language mentioned previously is extracted into the plugin.
4. Additional code for the UI not included in any language aspect is extracted, such as default settings for the IDE; branding including logos, splash screens and 'About Alembicue' information; and vendor information to link back to the website for the project.

16.2.3 Build for distribution

For the automation of building a distribution version of the software, Apache Ant provides a framework for scripting builds. A custom script was written to integrate the Ant build script for continuous integration in Phabricator. When a new release is committed and pushed to Phabricator, steps are performed to build all three types of application.

Ant build With the prerequisite of having built for development (subsection 16.2.2), the Ant build script for distribution is run (Figure 16.1). This is run in a Docker container set up for Ant build workflows, providing a portable environment containing the prerequisites required for building the software.

```
docker run
  --rm \                                  Automatically clean up container after build is run
  -v "$(pwd)":/app:delegated \            Mount current directory in container as /app
  sgrio/ant \                             Image for providing ant binary
  bash -c "
    /opt/apache-ant/bin/ant \             Location of ant defined in image
      -Dbasedir=/app -f /app/build.xml && Dbasedir ant option avoids cd
    /opt/apache-ant/bin/ant \
      -Dbasedir=/app -f /app/buildDistribution.xml"
```

Figure 16.1 Distribution build step 1: Ant built

Upload artefact Each artefact built, one for each of macOS, Windows and Linux, are uploaded to Phabricator's artefact tracking (Figure 16.2). This provides a link to download the artefact for use in the repository and on the webpage (subsection 16.3.3).

```
find app/build/artifacts/AlembicueDistribution/ \
  -name "Alembicue*" \
  -exec arc upload --json
```

Figure 16.2 Distribution build step 2: Phabricator artefact upload

PHID Each file artefact is given an incrementing ID number to refer to the artefact. However, to use the artefact with other systems as part of continuous integration, the artefact needs an identifier which is unique across all aspects of the continuous integration workflow. Phabricator provides a 'PHID', similar to a universally unique identifier (UUID), which identifies an item in Phabricator for reference by other Phabricator components or external tools. To determine the PHID of the uploaded artefact, a query is made utilising the Phabricator API. This PHID is used in the following step for association with another PHID.

```
echo '{"constraints": {"ids": [upload ID]}, "limit": "1"}' |   Search predicates
  arc call-conduit \                                           Make API call
  --conduit-uri https://phabricator.georgegarside.com/ \
  --conduit-token ${API token} \
  file.search |                                                Perform search on file artefacts
  jq -r '.response.data[0].phid'                               Parse response for PHID
```

Figure 16.3 Distribution build step 3: Get PHID of artefact from ID

Pin artefact to build With the artefact uploaded to the file storage and a unique identifier for the artefact, the file is associated with the build to provide a download link associated with the version control revision used to generate it. This provides a persistent link between the built application and the source code which was used to produce it.

```
echo "{
    \"buildTargetPHID\": \"build PHID to attach artefact\",
    \"artifactKey\": \"type of artefact, e.g. macOS/Windows\",
    \"artifactType\": \"file\",
    \"artifactData\": {\"filePHID\": \"artefact PHID (Figure 16.3)\"}
  }" |
  arc call-conduit \
  --conduit-uri https://phabricator.georgegarside.com/ \
  --conduit-token ${API token}
  harbormaster.createartifact
```

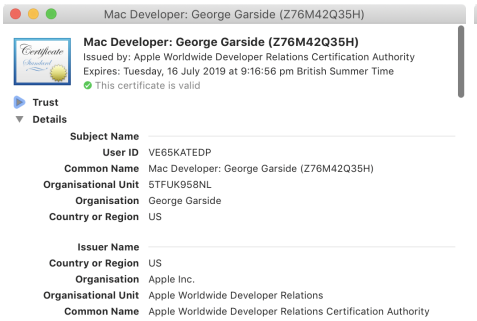
Create artefact association with build

Figure 16.4
Distribution build step 4: Associate artefact with continuous integration build

16.2.4 Code signing

Gatekeeper on macOS verifies developers of macOS software. Code signing attaches a digital signature to the application, verifying that it has been released without modification from its source. To perform code signing, the Developer ID certificate (Figure 16.5) containing the code signing purpose is attached to the released application.

With the Developer ID applying a signature to the application, Apple can verify that the software source is legitimate. To verify the authenticity of the software itself, Apple provides a notary service which ensures no malicious content is found in the application. The service publishes a list of tickets for authorised software, and provides a counterpart ticket which is 'stapled' to the released application, which Gatekeeper on macOS can check (Apple, 2019b).



(a) Certificate details

(b) Certificate extensions including code signing purpose

Figure 16.5
Apple Developer ID certificate for code signing

16.3 Webpage

To promote Alembicue and present the software for download, a webpage was created (Figure 16.11). This webpage advertises key features and functionality of Alembicue, with animated demonstrations of functionality.

The webpage is available at:
`https://georgegarside.com/apps/alembicue/`

16.3.1 Header

The header of the website (Figure 16.6), below the hero animation (subsection 16.3.2), features an attention-seeking headline followed by a brief introduction to Alembicue consisting of a few words of each unique selling point of the software. Each key feature is highlighted in a colour and with an appropriate symbol beside, such as an insertion point for scoped editing, a type system warning message underline for an available intention and a quick fix icon for an available error intention. A link to the code repository containing the source code of the project is placed beside the call to action download button, taking you to the download section of the page (subsection 16.3.3).

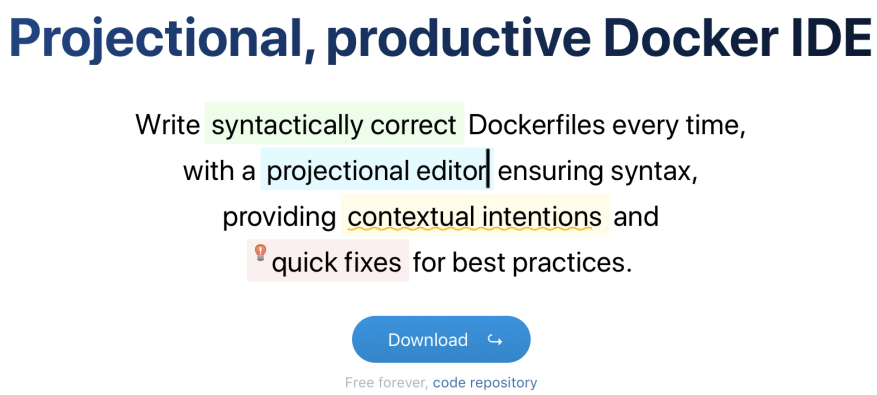
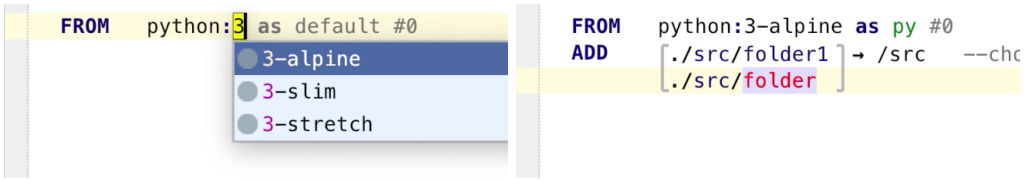


Figure 16.6
Webpage header

16.3.2 Animation

The website contains numerous animations which play through functionality of the editor. This attracts attention to various components of the editor which provides crucial indication of usage, alongside the textual description. These videos are very short, a few seconds in length, and autoplay and loop. Such videos include a demonstration of code completion and ‘multi-line’ projectional editor example (Figure 16.7), and a demonstration of type system messages and intentions (Figure 16.8).

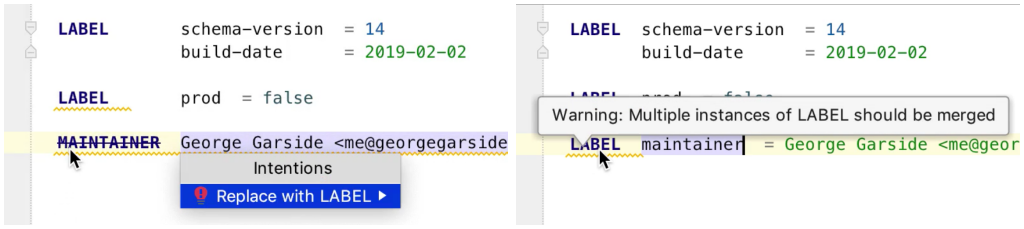
In a HTML5 and autoplay-blocked environment, autoplay is achieved by having the `muted` attribute set.



(a) `FROM version` property postfix completion

(b) `ADD` projectional editor multi-line example

Figure 16.7
Webpage video 1



(a) `MAINTAINER` deprecation replacement error intention

(b) Type system warning for multiple instances of `LABEL`

Figure 16.8
Webpage video 2

16.3.3 Download

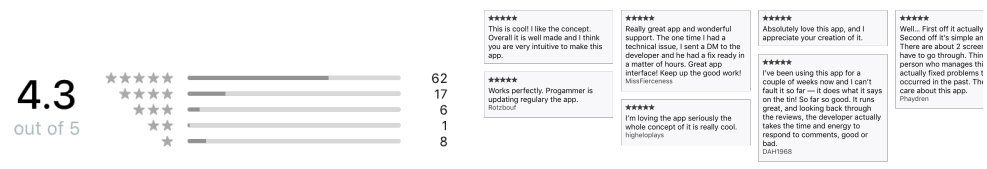
Links to download a pre-built package of the software are included near the close of the page content (Figure 16.9). Three buttons provide links to download Alembicue for macOS, Windows and Linux. These download a disk image (in the case of macOS) or a ZIP archive containing the application.

Figure 16.9
Webpage software download links

16.3.4 Testimonials

The webpage content concludes with an overview of star ratings received and testimonials received from users. Making submitted testimonials publicly available on the webpage helps visitors to the project see how useful the software has been for others, encouraging downloads for them to try it themselves.

Testimonials, reviews and comments are all provided using the form at the bottom of the page, consisting of a freeform textarea to receive messages from people. All messages were read and noted by the author, for analysis and observation, as well as bug fixing and identification of improvements. Testimonials from users provide very useful feedback for establishing whether the software is useful and which areas to focus on for future work. Assessing testimonials is a very useful evaluation technique, presented in this report's Evaluation section (subsection 17.3.1).



(a) Review stars

(b) Testimonials

Figure 16.10
Webpage review stars and testimonials

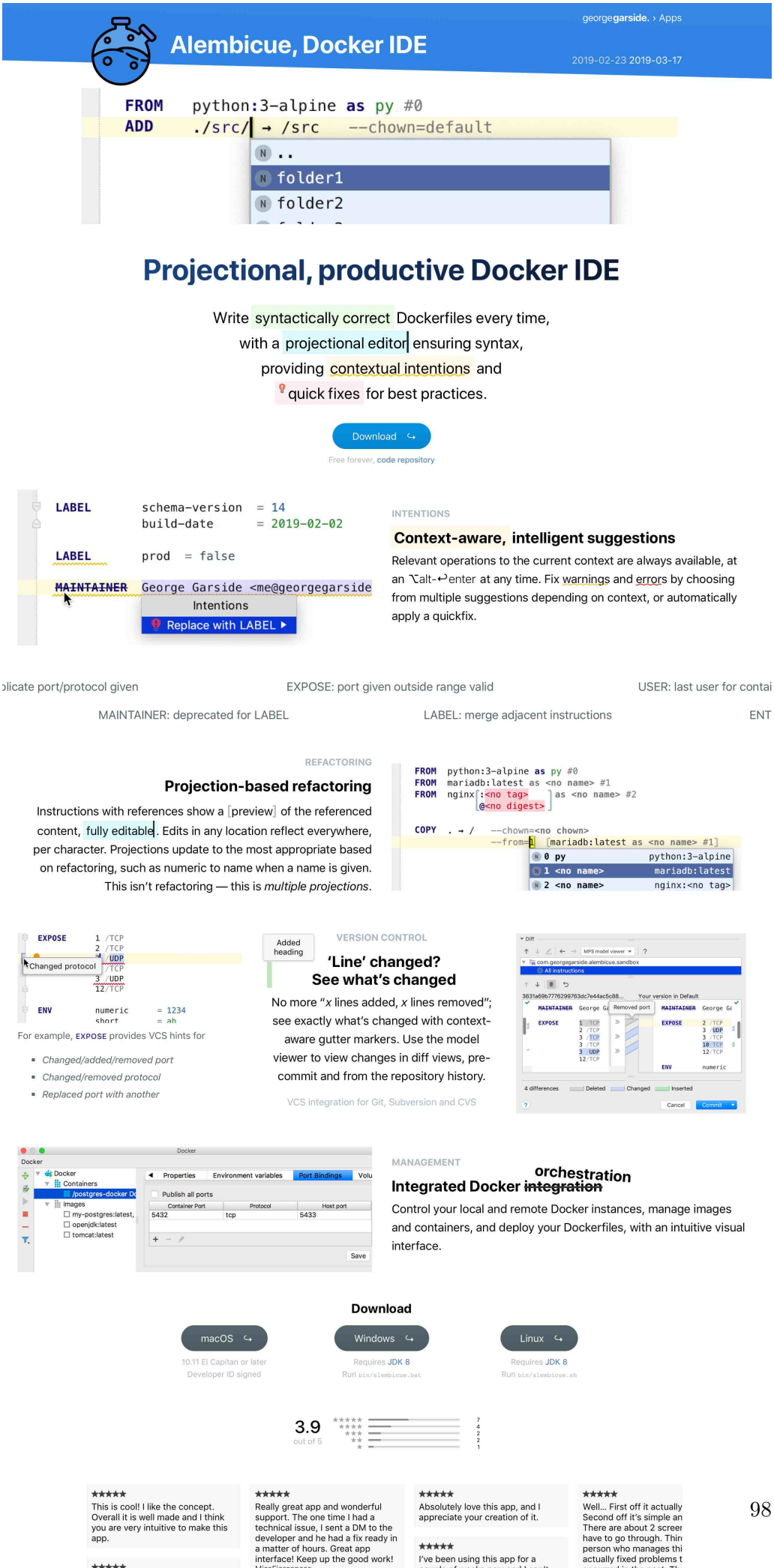


Figure 16.11 Screenshot of webpage for Alembicue

16.4 Versions

Semantic Versioning 2.0.0 specification is followed for versioning the pre-releases and releases of Alembicue. This helps communicate releases establishing new API and demonstrate backwards compatibility without the need to parse incomprehensible change logs, such as from the project version control (section 5.3), or lengthy release notes.

> Under this scheme, version numbers and the way they change convey meaning about the underlying code and what has been modified from one version to the next. (Preston-Werner, 2013)

While versioning numerically does signify various properties about the release, it is not a replacement for a textual description of the changes made in each revision. The changes made in each version are provided in Table 16.1 for versions prior to release and Table 16.2 for the release version and subsequent versions.

Alongside semantic versioning, 'EAP' is displayed beside all version numbers for current releases, indicating that these releases are part of the Early Access Programme. Such terminology refers to the 'ethical, compliant, and controlled mechanisms [to] outside of the [...] trial space and before the commercial launch' (Patil, 2016). These releases are meant for the public to use, as opposed to alpha and beta releases meant for internal and external testing respectively, but which also signifies the potential lack of feature completeness and other considerations. This encourages feedback on the software and promotes the active early development in the product's lifecycle.

Table 16.1
Changelog for pre-release versions

	Merge branch ‘`dev/composition`’
	Migrate common components to new language `alembicue.common`
0.1-a1	Service file with **`FROM`** instruction
	Require **`FROM`** to be first instruction
	Simplify common language removing empty models
	`FROM` text generation
0.1-b1	Editor refactoring & adjustments
	Add **`RUN`** instruction and transformation of `BlankLine` to instruction
	Add **`CMD`** instruction
	Add **`LABEL`** instruction
	Add **`MAINTAINER`** instruction and intention to use **`LABEL`**
	Add **`EXPOSE`** instruction with `Port` & `PortProtocol`
	Add **`ENV`** instruction and `KeyValueInstruction` interface
0.1	Add **`ADD`** instruction with `Path` concept & refactor `KeyValueInstruction` interface to concept
	Use indent buffer for Mapping TextGen
	Context Actions to set/unset chown on **`ADD`** instruction
0.2-b1	Instruction folding
	Add **`ENTRYPOINT`** instruction and create CommandInstruction
	Add Command for CommandInstruction and VOLUME instruction
	Add **`USER`** instruction and refactor stylesheets into common language
	Add **`WORKDIR`** instruction & create Windows user intention
0.2	Add **`ARG`** instruction with KeyValueOptional
	Refactoring TextGen for new lines in commands/etc
	Tidy inspection editor
	Set up Differential revisions
0.2.1	Replace ‘next applicable editor’ with editor components
	Quick fix for missing **`FROM`**
	Quick fix for inserting a missing **`FROM`**
	Port autocomplete to create both `TCP` and `UDP`
0.2.2	Fix empty file not showing `BlankLine`
	Trailing blank line info message
	Only check blank line if file contains **`FROM`**
	`getFile()` behaviour for Instruction
	`EXPOSE` check and fix for duplicates
0.3-b1	Deprecated strike-out in editor
	`BlankLine` transformation menu in context editor
	`EXPOSE` Port_Swap intention
	`FROM` editor show position in file
	`check_File` permit **`MAINTAINER`** before **`FROM`**
0.3	Add **`ONBUILD`** editor and constraint preventing **`FROM`** child

Table 16.2
Changelog for release versions

	Vertical grid for instructions in file
	ADD filesystem completion & validation
	Quick fix `fix_KeyValueInstruction_Merge`
	Move file checking rules to individual instruction checks
	ADD use reference to **USER** for `chown`
	`KeyValue` actions/keymap
1.0	**COPY** with reference to **FROM**
	Add type system instruction message suppression
1.0.1	Standalone IDE create project wizard
1.1	**SHELL** command
1.1.1	Fix backspace deleting previous node
1.1.2	Fix `chown` **USER** editor in **ADD** and add remote path URL
1.1.3	Fix Path text generation for both remote and relative
1.1.4	Fix transformations and actions for creation and deletion

17 Evaluation

Establishing that the project has met a success criteria (section 17.1) is an objective way to evaluate the success of a project. Furthermore, discussing the release of Alembicue (section 17.2) and subsequent reception from testimonials received (section 17.3) assists with evaluating the impact of the project's deliverables. This chapter details the successes and failures making up the project aim using objective criteria and subjective discussion.

17.1 Success criteria

To establish success of the project, it is necessary to demonstrate the project's ability to meet all the requirements set out as objectives (section 1.4) at the start of the project. The following is an evaluation of the project's work product in terms of these criteria.

1. Establish prerequisites to the project beginning, including its motivation, methodology and work plan to ensure the project stays on track.

   - ✔ The initial motivation of the project was established early, as was presented in its initial form in the project brief, before other parts of the project were considered. This ensured the project had the necessary basis to be regarded as a suitable final year project, as well as incorporating the necessary scope to make the project a viable solution to a real issue. The motivation was established further through the additional work which was done as part of this project report to demonstrate the motivation to the reader (section 1.2), establishing the usefulness of the deliverables.

   - ✔ The development methodologies (section 1.5) were also established before work on the project began, to ensure that all work carried out could be assessed appropriately at its conclusion. A plan of the work to be carried out was completed as part of this assessment, included in the report (subsection 1.5.2), to be followed through the project. Both the methodologies and the work plan were followed, with the work plan being adjusted as necessary as estimations could be made more accurate. This ensured the project was completed in a timely manner and to a high quality.

2. Research and review existing literature in the field for a greater understanding of the work which has taken place prior.

   - ✔ Existing literature was researched and reviewed as part of the completion of the literature review chapter of this report (chapter 2). This shaped the work on the project, providing a set of standards established in the past for the deliverables in this project to follow, as well as demonstrating areas which were found to be promising for this project to explore further. Alongside exploring the research, reviewing such work provided additional benefit by

critiquing and complementing the decisions and assumptions made, both of which were developed on further by this project in many areas.

- ➖ One area which was covered in the literature review was the integration of the editor with the compiler such as through compiler-specific application binary interfacing with the containing application. While the literature review did demonstrate that this was a promising area for future work, it was not explored in this project as it would arbitrarily link Alembicue to a specific existing compiler. Since the compiler is part of the BuildKit responsible for producing the image (Nephin et al., 2019), interfacing directly with it would require knowledge of the implementation of containerisation; an interesting topic by itself but which was not the focus of this language-orientated project.

3. Research, discuss and present conclusions, for an overview of containerisation necessary for the development of the containerisation language, and language workbenches for the implementation of the language.

   - ✔ Given an established focus on editing and language through literature review, establishing and understanding the specification of the language to be developed is crucial in creating appropriate deliverables for interoperability with this existing framework. Researching the containerisation process (section 3.1), so far as to understand the the applicability of the language to its provisioning of layers in the image build process, provides a greater understanding of how the language should be developed and the usage pertinent to developers. This also lead to the investigation and summarisation of the different language workbenches which could be used to develop various parts of the deliverables focused on language — this investigation was covered in the report (section 3.2) and conclusions were drawn providing the language workbench tooling decision used in the project.

4. Establish weaknesses in existing tooling for containerisation, and areas for innovation in Alembicue, through a detailed gap analysis.

   - ✔ The detailed gap analysis (chapter 4) performed in this project highlighted the shortcomings with many existing tools that provide a small subset of the functionality of the deliverables in this project. These issues with the existing tooling can be analysed objectively and contributed to the overall assessment of the software development efforts made in this area previously. By determining areas on which Alembicue can improve over these existing software, it can be shown that Alembicue is more suited to certain aspects of the development of containerisation.

5. Choose and set up tooling to begin development, including version control and continuous integration.

- ✔ Various languages needed to be learned for this project to take place effectively, given the choice of language workbench made for objective 3, and which were covered in this report (section 5.1). Establishing the learning effort for these languages and their applicability to the solutions ensured that effort was placed in the correct areas of the language suitable for use in development of deliverables for this project.

- ✔ Assistive development tools were used to configure an environment for working and developing in, including version control (section 5.3) and continuous integration (section 16.2). The successful use of these tools assisted with development by streamlining development, providing history for the project and ensuring quality of the released software through automated testing (chapter 15).

6. Design and implement a base level of the language, with core concepts such as the abstract syntax tree, and extensibility for continued development.

   - ✔ For the underlying structure of the language, core concepts were developed including the concept of an instruction (section 7.1), key-value pairs (section 7.2), a file concept (section 7.3), blank lines (section 7.4) and comments (section 7.5). These provided the necessary language foundation upon which the language acquired features through the use of these concepts to provide instructions. A successful implementation here was crucial in being able to provide a language which has semantic structure and with extensibility for future work, both in the scope of this project and in the future. This objective was met successfully, demonstrated through the implementation of the rest of the language making frequent and obligatory use of these concepts in providing functionality and syntax.

7. Design, develop and implement instructions for the language, including projectional editor components for each instruction, type system constraints and behaviours.

   - ✔ The basis of Alembicue was designed once all the prerequisites were addressed. This established important concepts, including the aspect model (section 6.1), and language functionality, such as intentions (section 6.4) and type system checking (section 6.5). The implementation of this was a success in providing extensibility for the rest of the language.

   - ✔ All instructions established in the language specification were implemented successfully, for configuring the environment (chapter 8), execution (chapter 9), container (chapter 10), filesystem (chapter 11) and metadata (chapter 12). Incorporating the syntax of the instructions into the language was carried out alongside the development of editing components for Alembicue's projectional editor. The implementation of each instruction was

the core of this project, and successfully completing this objective provided the necessary basis for the usability of the language outside of a technical demonstration of the capabilities of Alembicue to potential public release for use in production environments.

8. Develop and implement an integrated development environment, incorporating the projectional editor environment and Docker orchestration to run the language.

   ✔ Given the development of editing components for concepts in the language, incorporating the projectional editor into Alembicue provided the ability to edit documents written using the language (section 13.1). This provided the crucial functionality of the project in the software, ensuring its suitability for providing an accessible interface for the language, making use of the editor components developed alongside the instructions for the language.

   ✔ Docker integration in Alembicue provides orchestration capabilities to the software, adding the ability to control Docker to build the image from the Dockerfile written using Alembicue and to run a container based on the image. This provides end-to-end workflow integration from empty Dockerfile to running container using Alembicue, meeting this objective to provide an integrated development environment.

9. Perform suitable testing of each component of Alembicue and the overall application using established practices and methods.

   ✔ Testing of Alembicue was carried out alongside development (chapter 15), both manual and automated. By using both types of testing, an increased range of functionality of the software could be evaluated for mistakes and inaccuracies. All tests passed by the end of the project, successfully meeting this objective.

10. Release Alembicue for download and use by developers to write their own containerisation files.

    ✔ Alembicue was successfully released on its product webpage (section 16.3) and demonstrably in use by developers given the positive testimonials received as part of feedback solicited from users. The release of Alembicue was a success, measured in the release evaluation section of the report (section 17.2).

11. Evaluate the outcome of the project against each objective and the project's overall aim, including feedback received from Alembicue's release and possible future work to be carried out.

    ✔ Evaluation was performed as part of the review of the project, the findings of which are detailed in the evaluation section of the report (chapter 17). This evaluation also included reviews of Alembicue post-release, where feedback was incorporated into determining the reception of the software from the target market. Making sure each objective and deliverable was suitably met

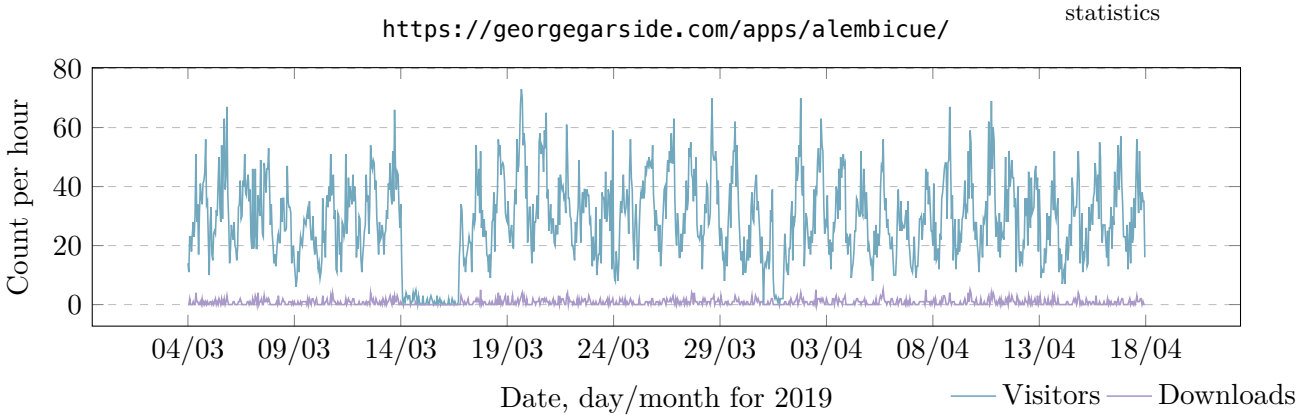
demonstrates the success of the project as a whole, incorporating all the necessary components of the project and providing value to those who will use Alembicue and work on it in the future.

17.2 Release

From release of Alembicue version 1.0 up to the time of analysis, a period of 1 month and 2 weeks has elapsed (from March 2019 to midway through April 2019). Additional updates and improvements were made to the project page and to the Alembicue application, which were rolled out during this time as covered in the release section of this report (section 16.4).

During this time, Alembicue's project page (`https://georgegarside.com/apps/alembicue/`) was visited 30,821 times (Figure 17.1). Acquisition of such visits can be broken down into two main categories: organic search, and direct or referral acquisition.

Figure 17.1
Alembicue webpage statistics



17.2.1 Organic search

87% of page acquisitions are accounted for by organic search, the majority source by far. According to reports generated by Google Search Console, the most organic clicks and impressions from a search engine results page are the result of queries on Google which can be categorised as follows:

Docker editing interest docker editor | docker ide | dockerfile editor

These queries are from people interested in solutions for editing Docker, namely Dockerfiles. Alembicue can be considered solution to this given its abilities to provide editing for Dockerfiles with its projectional editor (Part 5.4.2), and the regular placement of Alembicue in the search engine results pages (SERPs) for these queries demonstrates an established relevance to Alembicue.

Common Dockerfile issues dockerfile maintainer deprecated

Alembicue can help prevent common issues with Dockerfiles utilising intentions (section 6.4). The highest ranked query in this category regards the deprecation of the `MAINTAINER` instruction in the Dockerfile language specification, and Alembicue provides automation of the migration path to the correct instruction to be used instead (subsection 12.1.3) demonstrating its relevance to this query through the first feature advertisement on the webpage.

General concept interest projectional editor

Queries also relate to the general concept of projectional editing. With Alembicue providing an innovative implementation of this concept, it is relevant to many of these queries, and this is demonstrated through Alembicue's consistent appearance in various SERPs for queries in this category, leading to interest in the project and subsequent interaction (subsection 17.2.3). The regular appearance of Alembicue in search engine results for these terms shows that the project is applicable to many queries made by users. This demonstrates the success of the project in its application to these key areas through the provision of appropriate deliverables (subsection 1.4.1) as part of Alembicue's development and culminating in Alembicue's release.

17.2.2 Direct or referral

Visits to the Alembicue project page account for is approximately 4% of the total visits to the `https://georgegarside.com` domain over the same time period, with over 660,000 domain visits.

According to Quantcast (2019), the existing audience to the George Garside Network (which includes this domain) is heavily biased towards technology & computing, with audience interest in this category 3.8× average internet users. `georgegarside.com` features content on generic development and programming, and a large proportion of developers are familiar with or have an interest in cloud computing in general (Stack Overflow, 2019). prior to the addition of this project page there was no Docker-specific or containerisation-focused development content on the site. With 4% of site traffic, this is regarded as a very healthy page especially for a new project and for an audience lacking a demonstrative interest in Docker.

17.2.3 Conversions

The average visitor spent 4 minutes on the Alembicue project webpage. This demonstrates audience engagement with the content of the page, which can be attributed to presenting interesting functionality of Alembicue highlighting functions novel and unique to Alembicue, as well as being presented in an engaging format using animations demonstrating Alembicue's interface and interactions.

From engagement with the page, Alembicue was downloaded 855 times (Figure 17.1). This is a conversion rate from page view to download of almost 3%, which can be regarded very highly, as it is not expected that visits are from users with compatibility with the software, and some visits are from users who have already downloaded the software. With such a high conversion rate, it can be regarded that the content of the page is enticing visitors to try the software, which demonstrates both the engagement of the page and the intrigue presented by the concepts portrayed in the software.

The visits to the project page by desktop operating systems are distributed almost equally between macOS and Windows which account for the largest proportion, and the same distribution is apparent with downloads of Alembicue (Figure 17.2). This shows that the choice of building the software for the three platforms of macOS, Windows and Linux was a choice which has benefited the audience of the software, since excluding one of the platforms could have lost a large proportion of potential usage.

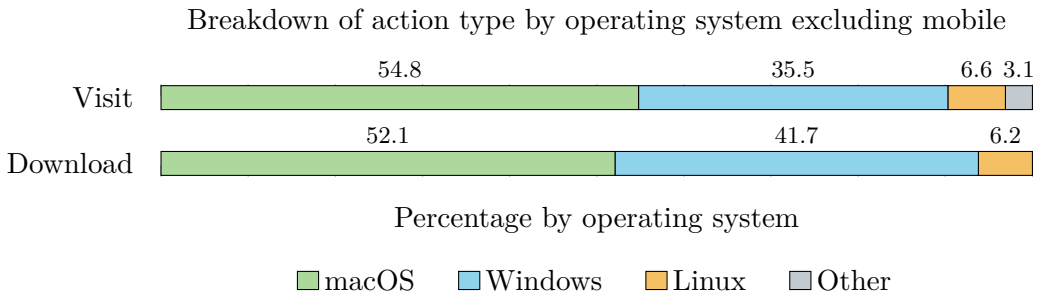


Figure 17.2
Downloads by desktop operating systems

17.3 Reception

Alembicue has been received well following release on the website. The feedback form on the page has received numerous comments and a star rating was given along with many of the comments. The ratings are tallied (Figure 17.3) and show a clear preference to higher ratings, with an average of 4.3 out of 5. This is a very positive response to such an early version of the application, and demonstrates the value of the product both to its current audience and potential market.

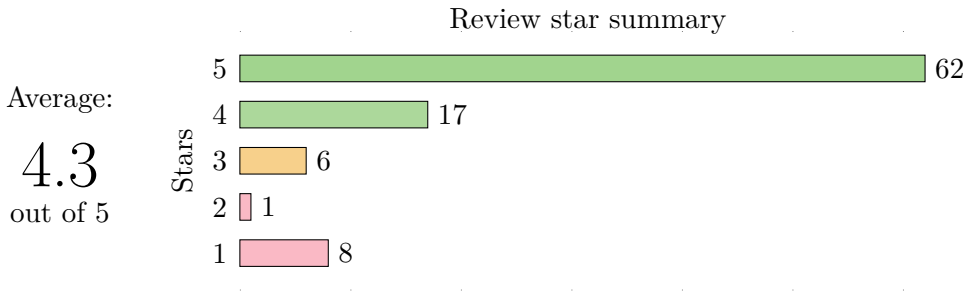


Figure 17.3
Summary of stars received in testimonials

17.3.1 New project issues

Various testimonials were received from users regarding the setup process of Alembicue following the initial release (version 1.0).

★ 'It just say loading for hours, pulling the screen changed to please wait few second after text gone nothing happen' — Dotfuscate, Indonesia

★★★ 'Would you please tell me why I getting blank page whenever I open the application??' — Ehsanviery, U.S.A.

★ 'Can't open the app I downloaded but when I log in, the screen turned all white.' — densajangel, U.S.A.

★ 'Your app doesn't work. It's just all white page.' — Saikie88, Philippines

Through feedback received which included the user agent string for the submission, and further testing carried out on virtual machines, it became apparent that on occasion there were issues with carrying out the creation of a new project in certain environments. Such issues included the creation of a project which lacked the necessary dependencies on the Alembicue language, therefore being unable to create a new file to begin work.

This was resolved through improved exception handling and user flow management through the new project wizard (section 14.3), which was released with version 1.0.1 16.2. Having resolved this issue and released a new version which could be downloaded from the webpage, many reviews were updated, demonstrating that the problem had been rectified.

★★★★★ 'The developer fixed the bug, and now the app works perfect!' — Nick-Snack, Argentina

★★★★★ 'App works like a charm now! Thank You!!! :D' — Wind Band FM, U.S.A.

17.3.2 Setup concerns

Some feedback received raised issues around getting started with Alembicue.

★★★★★	'I am new to this so can someone help me by telling me how I set up'	JustAedan, UK
★★★★	'I need a bit of help'	Sunlight Gaming, U.S.A.
★	'I don't understand what is this all about'	chris.pakalns, Latvia
★★★★	'I don't really understand how it work'	Mivang, U.S.A.

With the lack of detail as to what part of the application these people were having difficulties with, it's not possible to say for sure what could be improved. Various additional releases were made after the first public release of Alembicue (section 16.4), including to address issues on creating new projects (subsection 17.3.1) which could also be the source of these comments. Additional work could be to speak to users with issues and gain additional information on the cause of such problems, so that this could be addressed in a future feature release of the application.

On the contrary, some feedback directly contradicts the other feedbacks received regarding a lack of understanding of the application.

★★★★★	'Not bad at all. Seems a lot of people are not reading and understanding how the app works.'	russen., UK
★★★★★	'Great app I think most of the bad reviews here are from rival developers! Or people who don't understand. Very silly people. This is a great app.'	TheJoePiper, UK

It is apparent that the application requires some level of understanding before use, but alongside all the other commendations received (subsection 17.3.3), it can be said that this level is generally appropriate for the target audience. However, future work could involve the creation of a full user guide for the application which could help new users get started with the application more easily.

17.3.3 Commendations

These commendations demonstrate the usefulness of the software to the majority of those who have used it, which is a very encouraging sign that Alembicue has met its needs and is appropriately serving users.

★★★★★	'So far so good...'	JGL 1980, U.S.A.
★★★★★	'There's no need to complain'	Kanzai zhao, South Africa
★★★★	'Great app App works as intended'	J0725, U.S.A.
★★★	'I was pleased with this app up to this point. It wasn't good or bad, it had issues like every app but they were easily over looked.'	Satoshi king, U.S.A.
★★★★★	'Really great app and wonderful support. The one time I had a technical issue, I sent a DM to the developer and he had a fix ready in a matter of hours. Great app interface! Keep up the good work!'	MissFierceness, U.S.A.
★★★★★	'It Is what it says'	shadymcgavin, U.S.A.
★★★★★	'I have had no problems. Appears valid and legit. Thanks !'	TerraByte, India
★★★★★	'Works well, minimal interface'	Alexander Lynn, U.S.A.
★★★★★	'Great App! Enjoy using this.'	265run, U.S.A.
★★★★★	'So far so good I like it so far.'	Heddropolis, U.S.A.
★★★★	'*Funktioniert. Bin allerdings noch lange nicht soweit dieses auch zu testen.*'	Harald Grausam, Germany
★★★★★	'Love the app! It's easy and offers a lot of options.'	FernD0ll, U.S.A.
★★★★★	'I've been using this app for a couple of weeks now and I can't fault it so far — it does what it says on the tin! So far so good. It runs great, and looking back through the reviews, the developer actually takes the time and energy to respond to comments, good or bad.'	DAH1968, UK
★★★★	'*Nel complesso bella app*"	Andrydtd, Italy
★★★★★	'I'm not understanding the negative reviews when it comes to this app; it does exactly what the description of the app says it does and even more. Great app!'	Shaundebastion, U.S.A.
★★★★★	'Amazing! I do not know why everyone keeps rating this app low. It's amazing. I've been using it for a while and I have no complaints thank you for creating such an amazing app!'	Thefearjr, U.S.A.
★★★★★	'Great app and genuine very good app. Seems legit'	Thenameismaster, India
★★★★★	'*Se fa quel che promette è top App*'	Alex74gZ, Italy
★★★★	'Don't know why other users are having bad experiences. I have no issues at all.'	JeromeFami88, Philippines
★★★★★	'App seems to be working fine for me.'	Mynoraxis, Australia
★★★★	'Easy to use UI.'	Sir Jhom, Canada
★★★★★	'Why would anyone give this app anything less than 5 stars!?! What's not to love?!'	Gunner Stahl, U.S.A.
★★★★★	'Excellent app one of the best'	Arturo Larssen, Canada
★★★★★	'This is a nice app and I hope it continues to work.'	Kate Grr, UK
★★★★★	'Awesome Find Imo a must have for any enthusiast'	kadz, U.S.A.
★★★★★	'Waiting for an app like this. Still loving this app! Use it daily.'	G31573RF4HR3R, U.S.A.
★★★★★	'Simple, Functional, Works as intended. Good job.'	Hydroplosion, U.S.A.

17.3.4 Concept feedback

In addition to the general commendations received (subsection 17.3.3), some positive feedback made specific mention to the concepts used in Alembicue. Innovative concepts employed throughout include the projectional editor and the use of contextual guidance to assist the user with writing files. Such feedback is encouraging that the concepts are useful, providing helpful information and recommendations to the user along the way. This is the most encouraging feedback received as it demonstrates that future work in the area of projectional editing is worthwhile — something the market is looking for at the moment.

★★★★★ 'This is cool! I like the concept. Overall it is well made and I think you are very intuitive to make this app.' — *Anonymous*, U.S.A.

★★★★★ 'Nice app Im loving the app seriously the whole concept of it is really cool.' — higheloplays, U.S.A.

★★★★★ 'Seems like a very interesting concept & I'm excited to try it' — EzraRose, U.S.A.

However, some feedback on the concept appeared to demonstrate some difficulties with the application.

★★★ 'Dass man die manuell eintippen muss ist schon echt nervig, aber dass man nicht einmal korrigieren kann ohne gleich neu einzugeben reicht mir schon um zu erkennen dass es sich hier um eine sehr schlecht umgesetzte App-Idee handelt.' (Google Translate: 'That you have to type in manually is really annoying, but that you can not even correct without reentering is enough for me to realise that this is a very poorly implemented app idea.') — stefan4427, Germany

Due to a lack of detail in what exactly was causing difficulty, it is not possible to say for certain what aspect of the editor was problematic in this case. It is possible to speculate that the difficulty here was caused by instructions becoming fixed upon entering.

On successfully entering an instruction, the keyword for the instruction is fixed, and the instruction must be deleted in its entirety to change the keyword for the instruction. Unlike a regular text editor, where the keyword is just text on the page, the keyword in the projectional editor is a projection of the underlying node, and cannot simply be selected and edited. However, in a projectional editor, it would not make sense to be able to edit the keyword without changing the entire instruction, since the contents of the instruction is dependent on the type of instruction used. Removing the entire instruction is the only way to change the keyword because the keyword is a direct projection of the type of node used to represent that information in the tree.

Where possible, intentions provide automatic conversion between different types of node, such as from the deprecated `MAINTAINER` instruction to the `LABEL` instruction while keeping the contents of the instruction (transforming it to the necessary child type for the new instruction), and future work involves creating more of these contextual intentions available to the user.

17.3.5 Released versions

A few comments were made about the author releasing new versions of the software after development. This kind of feedback demonstrates the benefits of releasing new versions to fix issues and add functionality. For a full list of functionality added to the software after release, see Versions (section 16.4).

★★★★ 'Service contact efficace, j'avais un problème d'adresse et on m'a accompagner jusqu'à la résolution du problème. App agréable sans pubs et épurée, l'essentiel est là.' — lso3333, France

(Google Translate: 'Effective contact service, I had an address problem and I was accompanied until the problem was solved. Nice app without ads and refined, the essential is there.')

★★★★★ 'Bettered Interface. Recent updates have shown some great improvements.' — Corey_M_R1, UK

★★★★★ 'Great developer who listens to people fixed it. Thank you !' — Hannon96, U.S.A.

17.3.6 Suggested improvements

From the feedback received, some users suggested potential future work which could improve the application. While feedback on bugs and functionality issues were addressed where possible in the subsequent releases between the first release version of Alembicue and the time of writing this report, additional work suggested could not be put within the scope of this project and must be left to a later date.

★★★★★ 'Very Good App, Lovin it. Thank you for this app it works flawlessly. Looking forward for some fun settings to play with like dark theme etc.' — Radioactive, Pakistan

A dark theme for the editor interface could be implemented in a future version, as currently the background of the editor is white and the syntax highlighting is a fixed colour palette appropriate for the light background. A dark application interface is currently available through the appearance settings of Alembicue, but the editor remains light.

17.4 Future work

There are many possibilities for future work on this project given the extensibility of the language and the development software used. Providing the source code to the project as open source on Phabricator allows anyone to contribute improvements to the software, which can be compiled and released on the webpage for everyone else to use. Furthermore, incorporating the language editor into Alembicue removes the need for contributors to set up their own development environment and instead can focus on contributions to the language and functionality of the software. This helps encourages contributions to the software, as well as future work being done myself.

17.4.1 Language functionality

The needs of containerisation platforms change as new functionality is developed. Updating the language with more instructions as such needs change helps keep Docker at the forefront of the field. Therefore, updating the Alembicue language alongside ensures it provides this new functionality to everyone, not just those who wish to manually add instructions to Alembicue. Such addition of instructions can be made by extending existing instructions or extending the Instruction concept with new concepts that define their own parameters, as well as an editor component to provide the projectional editing environment for that instruction. Once these components are built, the existing functionality provides the instruction in the same manner as existing instructions, such as with code completion, without any additional steps from the developer. Such extensibility is a key benefit to Alembicue and will help keep the software relevant in the future.

17.4.2 Contextual assistance improvements

Context-aware behaviours are one of the most useful pieces of functionality provided by Alembicue on top of the language itself. Adding more of these behaviours, including intentions and more type system checking rules, warnings and errors, helps increase the editor's awareness of the code and encourages best practices in the development of such files.

While many intentions and contextual rules were added to the language as part of this project, the time constraints and scope of the project prevented the continued addition of more — the current Alembicue language definition and aspects (excluding the code necessary for the integrated development environment and application) is currently over 10,000 lines of Java, and the scope of the project within the timescales defined place a limit on how much assistant functionality can be built in. This is an obvious avenue to continue to develop Alembicue to make the software better at recognising improvements that can be made to the code.

17.4.3 Orchestration languages

In addition to the Dockerfile language used by Docker to build one image, other languages are available for other software which focus on other parts of the containerisation workflow.

Docker Compose is a tool for the creation of multi-container setups, each container based on a Dockerfile. Compose files only support a subset of YAML — certain syntax is valid YAML but invalid in compose files. Whilst tools exist to syntax check YAML documents, there does not exist an editor to assist with writing compose files, such that issues could be found prior to submitting the

file for composition. Docker provides `docker-compose config` to validate and view compose files, but this does not show all problems with the files, and also must be run regularly and manually to check the document for errors.

Extending Alembicue with support for Docker Compose, and by extension Kubernetes, will allow Alembicue to work with automated orchestration of images created using Dockerfiles also created with Alembicue, alongside manual orchestration using the existing Docker integration. This would provide the same benefits as discussed with Dockerfiles to more of the containerisation field and associated platforms.

18 Conclusion

The aim of the project was to design, development and release an integrated development environment, called Alembicue, for composing Docker images centred around a projectional editor for Dockerfiles. As this report has demonstrated, the project has been a success; all objectives were met, and deliverables were created to a high standard and released to great reception.

Success criteria evaluation (section 17.1)

As identified in a detailed gap analysis covering existing first-party and third-party tool support, no tool existed which was capable of providing sufficient syntax support for assisting developers in writing Dockerfiles. Alembicue, through a design of a complete language with specification of properties, constraints and type system, provides a formality and consistency that the existing solutions do not. Furthermore, using an innovative integration between syntax tree and editor, additional benefits were provided to the user of Alembicue including contextual intentions and suggestions for best practices. This functionality was provided alongside the revolutionary projectional editor which ensures syntax compliance and removes the need for parsing text in determining syntax and applying syntax highlighting — exceptional functionality both to this field of cloud computing and to programming in general.

Gap Analysis (chapter 4)

Design (chapter 6)

Type system (chapter 7), Environment (chapter 8), Execution (chapter 9), Container (chapter 10), Filesystem (chapter 11), Metadata (chapter 12)

The reception of Alembicue after release has been significant, encouraging and positive, demonstrating the successful development of the application and its ability to fill a gap in the market identified. This release was aided by previous work in developing a website with an audience interested in this field, which helped bring an existing audience of my previously released software to this new project, but analysis of the release also demonstrates the project acquiring a new audience separately; this is an encouraging identification of broad suitability to a popular field.

Reception (section 17.3)

Release (section 17.2)

While the scope of the project was large and the project itself covering many aspects, the completion of all objectives demonstrates that the scope was chosen well, of an appropriate difficulty keeping in mind the time constraints on the project. The time required for learning new programming languages necessary for the project was underestimated in the planning stages, but by planning at a high level until more information could be obtained for each development stage allowed adjustments to be made to the schedule to ensure a timely delivery.

Languages (section 5.1)

Methodology (section 1.5)

Alembicue provides a unique combination of language definition and projectional editor, with strong advantages over common parser-based text editing. This project has demonstrated the superiority of such editing paradigms and the favourable experience it provides to the entire field of programming.

Bibliography

Anlauff, M., Kutter, P.W. and Pierantonio, A., 1999. Tool support for language design and prototyping with montages. *International conference on compiler construction*. Springer, pp.296–300.

Apple, 2018. *Mac keyboard shortcuts* [online]. Available from: `https://support.apple.com/kb/HT201236` [accessed 5 November 2018].

Apple, 2019a. *Apple Human Interface Guidelines - Onboarding* [online]. Available from: `https://developer.apple.com/design/human-interface-guidelines/ios/app-architecture/onboarding/` [accessed 10 April 2019].

Apple, 2019b. *Notarizing Your App Before Distribution* [online]. Available from: `https://developer.apple.com/documentation/security/notarizing_your_app_before_distribution` [accessed 12 April 2019].

AWS, 2019. *What Is a Key-Value Database?* [Online]. Available from: `https://aws.amazon.com/nosql/key-value/` [accessed 15 March 2019].

Borras, P., Clément, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B. and Pascual, V., 1988. *Centaur: the system*. Vol. 13, 5. ACM.

Charmes, G., 2014. *What is the diffference between `CMD` and `ENTRYPOINT` in a Dockerfile?* [Online]. Available from: `https://stackoverflow.com/a/21564990/1549818` [accessed 11 February 2019].

Chen, M. and Nunamaker, J.F., 1989. Metaplex: an integrated environment for organization and information system development. *International conference on information systems: proceedings of the tenth international conference on information systems: boston, massachusetts, united states*. Vol. 1989, pp.141–151.

Collberg, C., 2009. *CSc 453 Compilers and Systems Software, 1: Compiler Overview* [online]. Available from: `https://www2.cs.arizona.edu/~collberg/Teaching/453/2009/Handouts/Handout-1.pdf` [accessed 31 October 2018].

Cormack, J., 2016. *Begin process of deprecating `MAINTAINER`* [online]. Available from: `https://github.com/moby/moby/pull/25466` [accessed 10 February 2019].

Crocker, D., 1982. RFC-822: Standard for the Format of ARPA Internet Text Messages. *Network Information Center*.

Dias, C. and Weatherford, S., 2018. *vscode-docker/readme.md* [online]. Available from: `https://github.com/Microsoft/vscode-docker/blob/master/README.md` [accessed 14 April 2019].

Docker, 2016. *Compose file version 2 reference* [online]. Available from: `https://docs.docker.com/compose/compose-file/compose-file-v2/` [accessed 11 February 2019].

Docker, 2018a. *Deprecated engine features* [online]. Available from: https://docs.docker.com/engine/deprecated/ [accessed 10 February 2019].

Docker, 2018b. *Docker glossary* [online]. Available from: https://docs.docker.com/glossary/ [accessed 12 March 2019].

Docker, 2018c. *Dockerfile reference* [online]. Available from: https://docs.docker.com/engine/reference/builder/ [accessed 2 November 2018].

Efftinge, S., 2015. *15 Minutes Tutorial* [online]. Available from: https://github.com/eclipse/xtext/blob/website-published/xtext-website/documentation/images/30min_editor.png [accessed 3 April 2019].

Efftinge, S. and Koehnlein, J., 1972. Xtext 2.9 - new & noteworthy. *EclipseCon NA 2016*, 7–10 February 2016 Reston. Virginia: EclipseCon.

Ernst, M., 2012. *Version control concepts and best practices* [online]. Available from: https://homes.cs.washington.edu/~mernst/advice/version-control.html [accessed 7 April 2019].

Evans Data Corp., 2018. Global Developer Population and Demographic Survey 2018.

Eysholdt, M. and Behrens, H., 2010. Xtext: implement your language faster than the quick and dirty way. *Proceedings of the acm international conference companion on object oriented programming systems languages and applications companion*. ACM, pp.307–309.

Fowler, M., 2005. *Language Workbenches and Model Driven Architecture* [online]. Available from: https://www.martinfowler.com/articles/mdaLanguageWorkbench.html [accessed 4 January 2019].

Fowler, M., 2008. *ProjectionalEditing* [online]. Available from: https://martinfowler.com/bliki/ProjectionalEditing.html [accessed 30 October 2018].

GNU, 2008. *GNU nano: Overview: 1.2 release* [online]. Available from: https://www.nano-editor.org/overview1.2.php [accessed 3 January 2019].

Gomez, L., 1988. Learning to use a text editor: Some learner characteristics that predict success. *Applied Ergonomics*, 19(1), p.76.

Gomez, L., Egan, D., Wheeler, E., Sharma, D. and Gruchacz, A., 1983. How interface design determines who has difficulty learning to use a text editor. *Proceedings of the sigchi conference on human factors in computing systems*. ACM, pp.176–181.

Hunt, A. and Thomas, D., 1999. *The Pragmatic Programmer: From Journeyman to Master*. Reading, Massachusetts. and et al.: Addison-Wesley.

JetBrains s.r.o., 2017. *MPS Meta Programming System* [online]. Available from: https://www.jetbrains.com/mps/ [accessed 4 April 2019].

JetBrains s.r.o., 2018a. *DevOps - The State of Developer Ecosystem Survey in 2018* [online]. Available from: https://www.jetbrains.com/research/devecosystem-2018/devops/ [accessed 21 April 2019].

JetBrains s.r.o., 2018b. *Domain-specific languages* [online]. Available from: https://www.jetbrains.com/mps/concepts/domain-specific-languages/ [accessed 4 April 2019].

JetBrains s.r.o., 2018c. *MPS Structure* [online]. Available from: https://confluence.jetbrains.com/display/MPSD20182/Structure [accessed 5 April 2019].

JetBrains s.r.o., 2019a. *Code completion: statement completion* [online]. Available from: https://www.jetbrains.com/help/idea/auto-completing-code.html#statements_completion [accessed 6 April 2019].

JetBrains s.r.o., 2019b. *Code Inspection: Unused local variable* [online]. Available from: https://www.jetbrains.com/help/rider/UnusedVariable.html [accessed 13 March 2019].

JetBrains s.r.o., 2019c. *Intention actions* [online]. JetBrains s.r.o. Available from: https://www.jetbrains.com/help/idea/intention-actions.html [accessed 28 March 2019].

Josey, A., Cragun, D., Stoughton, N., Brown, M., Hughes, C. et al., 2004. The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition. *The IEEE and The Open Group*, 20(6).

Kats, L.C. and Visser, E., 2010. The spoofax language workbench: rules for declarative specification of languages and ides. *ACM sigplan notices*, 45(10), pp.444–463.

Kelly, S., Lyytinen, K. and Rossi, M., 1996. MetaEdit+ a fully configurable multi-user and multi-tool case and came environment. *International conference on advanced information systems engineering*. Springer, pp.1–21.

Klint, P., 1993. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(2), pp.176–201.

Klint, P., Van Der Storm, T. and Vinju, J., 2009. Rascal: a domain specific language for source code analysis and manipulation. *2009 ninth ieee international working conference on source code analysis and manipulation*. IEEE, pp.168–177.

Knibbe, W., 1994. PIM for Macs speeds access to information. *InfoWorld*, 16(3), 22.

Ko, A., Aung, H. and Myers, B., 2005. Design requirements for more flexible structured editors from a study of programmers' text editing. *CHI'05 extended abstracts on human factors in computing systems*. ACM. Portland, Oregon, USA, pp.1557–1560.

Konopko, C., Shatalin, A. and Pech, V., 2011. *Base Language - MPS 1.5 Documentation* [online]. Available from: https://confluence.jetbrains.com/display/MPSD1/Base+Language [accessed 20 April 2019].

Koorn, J., 1992. *GSE: a generic text and structure editor* [online]. Amsterdam, The Netherlands: University of Amsterdam. Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.39.6948%5C&rep=rep1%5C&type=pdf [accessed 2 November 2018].

Kuiper, M. and Saraiva, J., 1998. Lrc—a generator for incremental language-oriented tools. *International conference on compiler construction*. Springer, pp.298–301.

Lamb, L. and Robbins, A., 1998. *Learning the vi Editor*. 6th ed. O'Reilly & Associates.

Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J. and Volgyesi, P., 2001. The generic modeling environment. *Workshop on intelligent signal processing, budapest, hungary*. Vol. 17, p.1.

McMahon, L., 1978. SED — A Non-interactive Text Editor.

Mernik, M., Lenič, M., Avdičaušević, E. and Žumer, V., 2002. Lisa: an interactive environment for programming language development. *International conference on compiler construction*. Springer, pp.1–4.

Mitchell, B., 2019. *What is port 0 used for?* [Online]. Available from: https://www.lifewire.com/port-0-in-tcp-and-udp-818145 [accessed 21 February 2019].

Moolenaar, B., 1998. *Vim reference manual: Vim Version 5.0 §syntax-highlighting* [online]. Available from: http://vimdoc.sourceforge.net/htmldoc/version5.html#new-highlighting [accessed 3 January 2019].

Nephin, D., Tiigi, T., Howard, J., Vass, T., Goff, B. et al., 2019. *Moby/builder.go at master moby/moby* [online]. Available from: https://github.com/moby/moby/blob/master/builder/dockerfile/builder.go [accessed 12 April 2019].

Oxford English dictionary, 1989. 2nd ed. Oxford: Clarendon Press.

Patil, S., 2016. Early access programs: benefits, challenges, and key considerations for successful implementation. *Perspectives in clinical research*, 7(1), p.4.

Performance Computing, 1984. Interview with Bill Joy. *Unix Review* [online]. Available from: https://web.archive.org/web/20120210184000/http://web.cecs.pdx.edu/~kirkenda/joy84.html [accessed 5 February 2019].

Preston-Werner, T., 2013. *Semantic Versioning v2.0.0* [online]. Available from: https://semver.org/spec/v2.0.0.html [accessed 11 April 2019].

Quantcast, 2019. *George Garside Network* [online]. Available from: https://www.quantcast.com/measure/profile/network/p-M86TpqD_mH-L8 [accessed 18 April 2019].

Rae Technology Inc., 1996. *Hierarchical structure editor for web sites*. US Patent US5911145A. 1999-06-08.

Smolander, K., Lyytinen, K., Tahvanainen, V.-P. and Marttiin, P., 1991. MetaEdit—a flexible graphical environment for methodology modelling. *International conference on advanced information systems engineering*. Springer, pp.168–193.

Söderberg, E. and Hedin, G., 2011. Building semantic editors using jastadd: tool demonstration. *Proceedings of the eleventh workshop on language descriptions, tools and applications*. ACM, p.11.

Solmi, R., 2011. *Whole Platform Project* [online]. Internet Archive. Available from: `https://sourceforge.net/apps/mediawiki/whole/index.php?title=Whole_Platform_Project` [accessed 3 April 2019].

Solmi, R., 2017. *Whole Platform* [online]. Available from: `https://whole.sourceforge.io` [accessed 3 April 2019].

Sorenson, P.G., Tremblay, J.-P. and McAllister, A.J., 1988. The Metaview system for many specification environments. *IEEE software*, 5(2), pp.30–38.

Stack Overflow, 2018. *Stack Overflow Developer Survey 2018* [online]. Available from: `https://insights.stackoverflow.com/survey/2018` [accessed 3 April 2019].

Stack Overflow, 2019. *Stack Overflow Developer Survey 2019* [online]. Available from: `https://insights.stackoverflow.com/survey/2019` [accessed 18 April 2019].

Sufrin, B., 1982. Formal specification of a display-oriented text editor. *Science of Computer Programming*, 1(3), pp.157–202.

Teichroew, D., Macasovic, P., Hershey, E.A. and Yamamoto, Y., 1980. Application of the entity-relationship approach to information processing systems modelling. *Proceedings of the 1st international conference on the entity-relationship approach to systems analysis and design*. North-Holland Publishing Co., pp.15–38.

Teitelbaum, T. and Reps, T., 1981. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Communications of the ACM*, 24(9), pp.563–573.

Turnbull, J., 2014. *The Docker Book: Containerization is the new virtualization*. James Turnbull.

Vass, T., 2017. *Moby/moby releases: v1.13.0* [online]. Available from: `https://github.com/moby/moby/releases/tag/v1.13.0` [accessed 10 February 2019].

Voelter, M. and Pech, V., 2012. Language modularity with the mps language workbench. *2012 34th international conference on software engineering (icse)*. IEEE, pp.1449–1450.

White, O., 2014. *IDEs vs. Build Tools: How Eclipse, IntelliJ IDEA & NetBeans users work with Maven, Ant, SBT & Gradle* [online]. Available from: `https://jrebel.com/rebellabs/ides-vs-build-tools-how-eclipse-intellij-idea-netbeans-users-work-with-maven-ant-sbt-gradle/` [accessed 3 April 2019].

Yuen, R., Yew, W., Hon, N. and Endo, S., 1997. Assembly language software development system. *Consumer Electronics, 1997. ISCE'97. and Proceedings of 1997 IEEE International Symposium on.* IEEE, pp.138–141.