Developing a zero-core modular IDE

Creating a zero-cost IDE; you get what you pay for

Nils Michael Fitjar

Master's thesis in Software Engineering at

Department of Computer science, Electrical engineering
and Mathematical sciences,
Western Norway University of Applied Sciences

Department of Informatics,
University of Bergen

1st June 2025

Western Norway
University of
Applied Sciences

Abstract

This paper introduces a modular, *zero-core*, application, to serve as an Integrated Development Environment (IDE) for experimental programming languages, addressing limitations in traditional IDEs. While standard IDEs are crucial in software development, their support for experimental languages is often inadequate. This can be mitigated by extensively using the module architecture of existing IDEs, by creating specific modules to address the shortfall of the host IDE. However, relying on *niche* modules or functionality is not beneficial for the longevity of the software. By analyzing the essential features of traditional IDEs a need for adaptability by IDEs to new paradigms and tools is highlighted. The solution, proposed by this paper, is to utilize a modular architecture to extend its lifespan and enhance support for experimental languages. Magnolia, a research programming language developed at the University of Bergen, serves as a case study, highlighting its unique characteristics and the necessity for a modular IDE. The primary research question explores how modularization facilitates the design and implementation of experimental programming languages. To showcase the usefulness of a modular approach, the modules needed to extend the core application to an IDE will be implemented.

Keywords: Modularization · IDE · Magnolia.

Acknowledgements

I would like to thank my supervisors, Magne Haveraaen and Mikhail Barash for their valuable feedback and discussions. Finally, I would also like to thank my friends and family, for their support throughout my studies.

`Nils Michael Fitjar`
1st June 2025

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Standard IDEs are indispensable tools in modern software development, offering features like early bug reporting, project outline visualization, code highlighting, and code completion, however, these IDEs may not adequately support the unique demands of experimental programming languages. Experimental languages could introduce new concepts like Abstract Semantic Representation (ASR) transformations, term algebras, mathematics of arrays, Syntactic Theory Functor (STF) [12], or other novel programming features. These are concepts from the academic community, and are not common in *mainstream* languages, and as such, have little to no support in modern IDEs. To solve this, researchers need ad hoc solutions for existing IDEs, adding the needed functionality to test out their language features. If this ad hoc solution is too extreme; outside the standard functionality supported by the developers of the IDE, it might be short-lived. As the IDE is maintained, updated and improved, the features used to solve the niche needs of the experimental language might be deprecated.

However, if the IDE has integrated support for extending the standard functionality of the application, then the ad hoc solution will be more stable. Such a system is known by many names. Plug-in architecture, extension Application Programming Interface (API), or add-on system, to name a few. The common factor amongst these systems, is that some component, be it a plug-in, an extension, an add-on, or a module, can extend the functionality of the application. This is a modular approach to extending the lifetime of an application; extending its software longevity. In many of these systems, said components, are composable, allowing for multiple components to work together in a modular fashion to add extra features to an application. This way of adding functionality to an application is commonly used in IDEs.

1.1 Modular architecture

A zero-core, modular IDE would assist in these ad hoc solutions. Even if a new feature from an experimental language is introduced, it is unlikely that this feature has no relation to existing features, and as such, it is easier to extend the application in such a manner to facilitate this new feature, with help of existing modules. However, if it is the case that this feature is paradigm-shifting, then there will still be existing functionality that can be used, re-used or extended to facilitate this.

Hypothesis 1. When an application is designed to be modular from the start, then features not thought of, by the original developers can be integrated into the application, and be stable. If an experimental research language introduces some paradigm shifting concept, then this can easily be tested in a modular IDE.

1.2 Zero-core architecture

Taking the modular architecture design to the extreme, the core application has no base features, everything is enabled by an external module. We call such a highly modular application, a *zero-core* application. To qualify for a *zero-core* application, the default application has no functionality; everything is acquired by modules. Such a design facilitates a modular approach, enabling a module-developer to only focus on the functionality they want to extend, not the entire core.

1.3 Thesis outline

Traditional IDEs encompass essential features such as syntax highlighting, code navigation, and hover-help, all of which play a crucial role in the software development process. However, their limitations become apparent when working with experimental languages. This paper advocates for modularization and composability as key design principles, demonstrating their ability to extend the operational lifespan of software by allowing for ease-of adoption to new paradigms and tools. The discussion revolves around Magnolia, an experimental research programming language developed by Bergen Language Design Laboratory (BLDL) at the University of Bergen. Magnolia is a way to experiment with

novel language features. It will therefore be a case study illustrating the need for a specialized IDE.

The focus point of this paper is to design a zero-core architecture, to develop and implement a modular IDE, where the target language will be Magnolia.

In chapter 2, we will introduce Magonlia, and features this language introduces that are difficult to encompass using standard IDEs. In chapter 3 we will explore the use case of the zero-core, modular IDE, focusing on the different users of this application. Chapter 4 will discuss the design and implementation of the IDE, mentioning different designs that were considered, some challenges that were encountered, and the modules developed to add the necessary functionality to qualify as an IDE. In chapter 5, we will discuss related works, amongst them how different IDE-vendors allow for extending of their core application. Chapter 6 will discuss the results of our implementation, and answer our hypothesis 1. Finally, in chapter 7, we will discuss the necessary work remaining.

Chapter 2

Background

In this chapter, we will introduce Magnolia in section 2.1, some of the interesting aspects the language introduces, and examples of how we can use it. We will then discuss IDEs in the following section 2.2, and mention some specific functionality they have that make them great tools for developers. Then, in section 2.3, we will cover our approach, and how it differs from standard modularity in other IDEs. We will also address the current Magnolia IDE. Finally, in the last section 2.4 we will discuss the challenges that exist due to the uniqueness of the Magnolia language.

2.1 Magnolia

Magnolia is designed to support a high level of abstraction and ease of reasoning. It was created with the purpose of being highly extensible, allowing for experimentation of language features, like functionalization, mutification, generated types and type partitions. With Bagge [1], we can summarize a Magnolia program to three fundamental ideas.

- *Concept*: A set of operations, type declarations and axioms.
- *Implementation*: Implementation of a concept.
- *Satisfaction*: Satisfaction of an implementation.

While Magnolia is inspired by things like abstract algebra and institution theory, it is quite trivial to understand on a conceptual level. Developers quite often work with sets of operations, type declarations and axioms, namely APIs.

2.1.1 Magnolia concept

Commonly, the term API is used specifically for Representational State Transfer (REST) APIs, but it also covers concepts, like those in Magnolia, interfaces in Java, traits in Rust, or type-classes in Haskell. What all of these variations have in common, is that they specify a method for two different procedures to communicate with each other. In a REST API this could be a microservice architecture, where several servers send and receive requests and responses, or in a programming project, it could be the **List** interface in Java, which informs consumers of that interface, which methods are needed to qualify as a **List**. In Magnolia a concept declare types, functions and properties which those functions need to uphold. A simple example of this would be a concept for addition with natural numbers.

```
concept NaturalNumbers = {
  type N;
  function zero(): N;
  function succ(number: N): N;
  function _+_(a: N, b: N): N;
  axiom unit(a: N) {
    assert zero() + a == a;
    assert a + zero() == a;
  };
};
```

Listing 2.1: Natural numbers (Magnolia)

In the listing 2.1, we are specifying concept called *NaturalNumbers*, which declares a type **N**, and three methods that act upon the type **N**. We have the function[1] **zero**, which takes zero arguments, and should return something of type **N**. With this constructor, we can instantiate our numbers. To get new numbers, we have the function **succ**, which should give the *succ*essor to the passed number. That way, we can represent 0 as **zero()**, 1 as **succ(zero())**, and 2 as **succ(succ(zero()))**. The final function, is an infix operator. + takes two arguments of type **N** and returns an **N**. Of course, this should be interpreted as addition, meaning **succ(zero())** + **succ(succ(zero()))** = **succ(succ(succ(zero())))**, or using numbers: $1 + 2 = 3$. Finally, the last statement in

[1] Also called a constructor

the concept is an axiom, stating that given any a, if we add **zero()** to a, we should get a, of type **N**.

This axiom is what allows for us to put constraints on our concepts, which allows for improvement in our API. Unlike other APIs, like traits or REST, such specific constraints can only be achieved by using unit tests, which is not enforced on the implementor. But with axioms, this is possible in Magnolia. This pattern is quite useful, since it allows for *reuse* of logic. The listings 2.2 specifies a list interface, we can instantiate a list, and do operations on it, such as getting adding an element to the list, or by concatenating two lists. We can also fetch the first element of the list, by using **head**, note the *guard* attached to the function statement, this guard ensures that when this method is invoked, the list we get the first element from, cannot be empty[2], which means that our function is not *total*, but *partial*. This means that for any argument, we might not have a corresponding result. If we did not have this guard, then what would happen if we *took* **head** of a list with no elements? In languages like Java, we would get null, but this does not exist in Magnolia. We could expand upon the list API by creating a non-empty variant of list, as shown in listing 2.3, which is the same as list, except to instantiate it, we need to supply an element, ensuring when we have a **NonEmptyList** variant, we can safely get an element from it, since there will at minimum be one element in the list, due to our constructor requiring one argument.

```
concept List = {
  type T;
  type List;
  function nil(): List;
  function cons(xs: List, x: T): List;
  function head(xs: List): T
    guard xs != nil();
  function concat(xs: List, ys: List): List;
};
```

Listing 2.2: List concept (Magnolia)

[2]Meaning it is equal to **nil()**

```
concept NonEmptyList = {
  use List;
  type NonEmptyList;
  function build(t: T): NonEmptyList;
  function cons(xs: NonEmptyList, x: T): NonEmptyList;
  function head(xs: NonEmptyList): T;
  function concat(xs: NonEmptyList, ys: NonEmptyList): NonEmptyList;
  function concat(xs: List, ys: NonEmptyList): NonEmptyList;
  function concat(xs: NonEmptyList, ys: List): NonEmptyList;
  axiom notEmpty(xs: NonEmptyList) {
      assert xs != nil();
  }
};
```

Listing 2.3: NonEmptyList concept (Magnolia)

However, the interpretation we have assigned to the *NaturalNumbers*, *List*, and *NonEmptyList* concept, depends on our implementation.

2.1.2 Magnolia implementation

As one can see in listing 2.4, we have implemented the concept specified in listing 2.1, by using concrete values for **N**. There is an implementation for all the functions, giving us the functionality we set out to specify with our concept, but there is nothing stopping us from straying away from the specification, by implementing it incorrectly. Since we are using primitive types[3] we have to use external code, which is another feature of Magnolia. In the listings 2.4 and 2.5, we are using an external implementation of numbers, from C++.

```
implementation implNaturalNumbers
  = external C++ CxxNaturalNumbers signature(NaturalNumbers);
```

Listing 2.4: Natural numbers implementation (Magnolia)

```
implementation implNaturalNumbersInvalid =
  = external C++ CxxNaturalNumbers signature(NaturalNumbersWrong);
```

Listing 2.5: Invalid implementation (Magnolia)

[3]In this case, integers, specifically 32-bit

We have defined the function **zero** correctly in listing 2.6. On line five, we return 0, which is what we expect, based on our axiom. However, on line 5, in listing 2.7, we instead return 1, which breaks our axiom, as $1 + 1 \neq 1$.

```
1  struct CxxNaturalNumbers_impl {
2    typedef int N;
3
4    N zero() {
5      return 0;
6    }
7
8    N succ(const& N number) {
9      return number + 1;
10   }
11
12   N plus(const& N a, const &N b) {
13     return a + b;
14   }
15 };
```

Listing 2.6: Natural numbers implementation (C++)

```
1  struct CxxNaturalNumbers_impl {
2    typedef int N;
3
4    N zero() {
5      return 1;
6    }
7
8    N succ(const& N number) {
9      return number + 1;
10   }
11
12   N plus(const& N a, const &N b) {
13     return a + b;
14   }
15 };
```

Listing 2.7: Invalid implementation (C++)

This is where the *satisfaction* comes in, it is what ties the concept and implementation together, by ensuring our axiom are upheld.

2.1.3 Magnolia satisfaction

```
satisfaction implNaturalNumbersIsValid
  = implNaturalNumbers models NaturalNumbers;
```

Listing 2.8: Satisfaction of the natural numbers (Magnolia)

When implementing a concept in Magnolia, one could do so incorrectly. In other programming languages, one might still have *imparted* some meaning in an interface, like that in Java, the interface **List** in the standard library, the method **addAll** is an associative operation, as shown in the following listing, 2.9, where we expect the assertion to be true.

```
public static void main(String[] args) {
  List<Int> xs = new ArrayList() { 1, 2 };
  List<Int> ys = new ArrayList() { 3, 4 };
  List<Int> zs = new ArrayList() { 5, 6 };
  assert (xs.addAll(ys.addAll(zs)))
         .equals(xs.addAll(ys).addAll(zs));
}
```

Listing 2.9: List concatenation that should result in the same list. (Java)

Of course, this is the case in the standard library to Java, but there is nothing that enforces this property on other implementers of the interface. Usually, programmers who want such a property on their interfaces, create unit tests, testing that it is the case that **addAll** is associative. But to enforce this, one would have to create a unit test for each implementation of the interface, while in Magnolia one writes a satisfaction as shown in listing 2.8, which in turn is *transpiled* to a format understandable by an Satisfiability Modulo Theories (SMT) solver. This SMT solver can *prove* that our implementation upholds our axioms in the implemented concept. Importantly, this can be done for any consumer of the API.

Satisfiability Modulo Theories solvers

A SMT solvers are programs that *prove* first-order formulas. Skogvik [17] showcases how Magnolia concepts can be translated into such first-order formulas, and be used by SMT solvers to verify them. Skogvik also lays out different SMT solvers and compare them against each other, with the Magnolia library as input. One of Skogviks conclusions are that while verification of some program is good to have, some features needed for a new IDE would be to integrate it with this functionality.

2.1.4 Mathematics and programming

Mathematics is everywhere, and useful. It's not always easy to notice this, but one thing that helps, is knowing the names of the structures one encounter. One can easily understand that knowing simple operations like addition, multiplication, etc. is useful but for more abstract mathematics, this is harder. An example of this is abstract algebra, which is the study of algebraic structures, which are often seen in programming. A programmer will use these structures more often than not, knowingly or unknowingly, and a good programmer will explicitly seek these structures out.

An important aspect of development, is logging. Knowing what actions have taken place is an essential tool when hunting down bugs. A common way to structure logs, would be composing them depending on when in the call stack they occurred. As a concrete example, let's say we are making a text editor, and are in the need of a logging manager, which, among other things, should compose different log statements. Assuming we have some type **Log(A)**, where the type **A**, is the result of the computation of a given function, we want to be able to compose different, related, computations. But, importantly, the order of composition of the **Log(A)**-type matters. Representing the composition of the **Log(A)**-type as $\odot$, and letting a, b, c be of type **Log A**:

Definition 2.1.1 (Log Composition)**.**

$$a \odot (b \odot c) = (a \odot b) \odot c \tag{2.1}$$

Now we have a good logger, as the logs of the entire call stack is available for us to read when something goes wrong. Moving on, a good feature of a text editor, is being able to undo and redo actions. These are the actions that a user should be able to do:

- Insert text at a position
- Delete text from a position
- Redo an action
- Undo an action

Same as in the logging example, composing is a reasonable thing to implement, and should result in another action. Similarly, the order matters; deleting text and then inserting, is not the same as inserting and then deleting. But what is different between the logging and editor example, is that we also want the *inverse* of an action, so for every action we want an opposite action that undos an action. Say, a is some action, and b is some opposite action, then our composition looks like this:

Definition 2.1.2 (Action Composition)**.**

$$a \odot b = U \tag{2.2}$$

Where U is an action representing *no-operation*. This could be inserting the empty string at any position, deleting the empty string at any position, or redoing or undoing any of the aforementioned actions.

Both of these examples are relatively easy to implement, but harder to verify, to ensure they satisfy our properties; that the *logic* holds. In Java and Rust, to ensure that we have implemented something correctly, one would create a unit test. But there is nothing to ensure that we create this test correctly, that we cover all the edge cases, or if we are testing the correct thing.

2.1.5 Logging example in Java, Rust, and Magnolia

In the Java listing (2.10) and Rust listing (2.11) we have implementations of logging which might not uphold our constraints. We can add unit tests, that ensure the implementations satisfy the definition 2.1.1, but this safeguard only exists in our project, and once our API can be implemented by third-party developers, we have no guarantee they will follow our constraints.

```
interface Log<T> {
  public Log<T> appendLogs(Log<T> a, Log<T> b);
}
```

Listing 2.10: Logging structure (Java)

```
trait Log {
    fn appendLog(self, other: Self) -> Self;
}
```

Listing 2.11: Logging structure (Rust)

In Magnolia, however, it is possible to constrain a consumer of our API. In the listing 2.12, we can add an axiom, which is the same as our requirement definition 2.1.2. For implementers, *consumers* of our API, we can now ensure they implement correctly, as

long as they add the simple declaration showed in listing 2.13. This will be ensured, because in a standard Magnolia work routine, a developer will invoke an SMT solver, which will ensure the concepts, that are implemented are sound; that they are satisfiable.

```
concept Log = {
  type Log;
  function appendLog(first: Log, second: Log): Log;
  axiom logComposition(a: Log, B: Log, c: Log) {
    assert
      appendLog(appendLog(a, b), c)
        ==
      appendLog(a, appendLog(b, c));
  }
};
```

Listing 2.12: Logging structure (Magnolia)

```
satisfaction loggingImplSatsLog = loggingImpl models Log;
```

Listing 2.13: Magnolia logging satisfaction, the implementation is left out for brevity

2.1.6 Editor example in Java, Rust, and Magnolia

The Java and Rust listings, (2.14, 2.15), also have no method of ensuring the satisfiability of future implementations. But what is more interesting, is that we can see, clearly in the case of the Magnolia listing 2.16, that there is some kind of relation between these APIs.

```
interface Action {
  public Action addText(String s);
  public Action removeText(String s);
  public Action redoAction(Action action);
  public Action undoAction(Action action);
  public Action combineAction(Action first, Action second);
}
```

Listing 2.14: Editor structure (Java)

```rust
trait Action {
    pub fn addText() -> Self;
    pub fn removeText() -> Self;
    pub fn redoAction(&self) -> Self;
    pub fn undoAction(&self) -> Self;
    pub fn combineAction(self, second: Self) -> Self;
}
```

Listing 2.15: Editor structure (Rust)

```
concept Action = {
  type Action;
  function noop(): Action;
  function addText(): Action;
  function removeText(): Action;
  function redoAction(a: Action): Action;
  function combineAction(first: Action, second: Action): Action;
  axiom noop(a: Action) {
    assert combineAction(noop(), a) == a;
  };
  axiom combineAction(a: Action, b: Action, c: Action) {
      assert
        combineAction(a, combineAction(b, c))
          ==
        combineAction(combineAction(a, b), c);
  };
};
```

Listing 2.16: Editor structure (Magnolia)

Both the logging example, and the text editor example, are some binary operation[4] over some set. In the first example, our set was all different log statements of the type **Log A**, and composing these logs, gave us another **Log A** type. While in the second example, we were working on the set of actions, which we could compose, which also gave us another action, but we also had an action representing no-operation, and an *inverse* operation, undoing an action. This is related to mathematics, specifically abstract algebra, the study of algebraic structures.

[4]Function with two arguments

2.1.7 Abstract algebra

In the first example, we are working with a *semigroup*, and in the second example, we are working with a *group*. These are known as algebraic structures, which is just some set, with a function that takes two inputs, and outputs one result, and some property on that function. The trivial example, is known as *magma*, and is defined in 2.1.4. The closure definition 2.1.3 simply specifies that we only work with one set.

Definition 2.1.3 (Closure)**.** For a set M, with a binary operation $\oplus$, $\forall a, \forall b, \exists c \in M$, such that $a \oplus b = c$.

Closure Addition with the integers is a kind of closure, as per the definition 2.1.3, since no matter what integer you put into the equation, you will still get an integer. And since this is the only requirement a magma has, this example is also a magma.

Definition 2.1.4 (Magma)**.** A magma is a set M, with a binary operation $\oplus$, which is *closed* by definition 2.1.3

We can *extend* the definition of magma, by adding associativity on the binary operation. The definition 2.1.5, as shown in the example 2.1.1, simply specifies that the order we evaluate our composition matters.

Example 2.1.1. Multiplication with the positive integers is associative, since no matter where we put parentheses; what order we evaluate this equation: $2 \times 3 \times 4$, we will get the same answer.

$$(2 \times 3) \times 4 = 2 \times (3 \times 4)$$

Definition 2.1.5 (Associativity Law)**.** For any binary operation $\oplus$, on a set M, $a, b, c \in M$. $a \oplus (b \oplus c) = (a \oplus b) \oplus c$, must hold.

This associativity gives us a semigroup, as shown in the definition 2.1.6, which is the structure that we modeled in our logging example.

Definition 2.1.6 (Semigroup)**.** A semigroup is a set M, with a binary operation $\oplus$, and $\oplus$ must uphold the definitions 2.1.3 and 2.1.5.

By simply requiring the identity law (2.1.7), we get a monoid (2.1.8), and adding the inverse law (2.1.9), we get a group.

Definition 2.1.7 (Identity Law)**.** For any binary operation $\oplus$, on a set M, $\forall a, \exists U \in M$, such that $a \oplus U = a$ and $U \oplus a = a$. This makes U is unique.

Definition 2.1.8 (Monoid)**.** A monoid is a set M, with a binary operation $\oplus$, and $\oplus$ must uphold the definitions 2.1.3, 2.1.5, and 2.1.7.

Example 2.1.2. To make a monoid, we can choose the binary operation to be $\times$, and our set to be the natural numbers, ($\mathbb{N}$). We know addition is closed, and associative, so choosing $U = 1$, we get a monoid. Any number from our set $\mathbb{N}$ multiplied with 1, gives us the number we choose.

Definition 2.1.9 (Inverse Law)**.** For any binary operation $\oplus$, on a set M, $\forall a, \exists U \in M$, such that $a \oplus U = a$, and U is unique. And $\forall a, \exists b \in M$, such that $a \oplus b = U$ and $b \oplus a = U$, and the mapping for $a \to b$ is one-to-one.

Definition 2.1.10 (Group)**.** A group is a set M, with a binary operation $\oplus$, and $\oplus$ must uphold the definitions 2.1.3, 2.1.5, 2.1.7, and 2.1.9.

The definition 2.1.10, of course is identical to the structure we used to model undo-redo, in our text editor example. We have our *combine* operator, which takes two variants of the *action*-type, and return another variant of the same type. It is therefore a closed binary operation. We also specified that *combine* is associative, it does not matter where one puts the parentheses, the resulting combination of *action*s are equivalent. *no-operation* gives us the *unit*, meaning our binary operation upholds the identity law. Finally, we have an inverse *action* variant, that is unique, for all other variants.

This, of course, is dependent on our implementation. It is quite common to make mistakes when developing. These mistakes are usually tackled by developing good unit tests. However, this can get quite tedious, as for everytime we re-implement these algebraic structures, we would have to re-create the unit tests. To avoid these common mistakes when implementing these structures, it would behoove a developer if they could encode these properties in something like an interface or a trait, however, this is not possible in either Java nor Rust. We cannot enforce things like the definition 2.1.5 on our operators.

2.1.8 Java: magma to group

This structure could *technically* be implemented in something like Java as shown in listings 2.17, 2.18, 2.19, and 2.20. But not practically. Note the empty interfaces; there is nothing that enforces the different laws; properties, on our method. This can only be done by unit testing, which is not enforced on an external consumer of the API.

```java
interface Magma<T> {
  public T binop(T a, T b);
}
```

Listing 2.17: Magma interface (Java)

```java
interface Semigroup<T> extends Magma<T> {}
```

Listing 2.18: Semigroup interface (Java)

```java
interface Monoid<T> extends Semigroup<T> {
  public T unit();
}
```

Listing 2.19: Monoid interface (Java)

```java
interface Group<T> extends Monoid<T> {}
```

Listing 2.20: Group interface (Java)

2.1.9 Rust: magma to group

The same issue with property enforcement exists in Rust.

```rust
trait Magma<A> {
    fn op(a: A, b: A) -> A
}
```

Listing 2.21: Magma trait (Rust)

```rust
trait Semigroup<A>: Magma<A> {}
```

Listing 2.22: Semigroup trait (Rust)

```rust
trait Monoid<A>: Semigroup<A> {
    fn identity() -> A;
}
```

Listing 2.23: Monoid trait (Rust)

```rust
trait Group<A>: Monoid<A> {}
```

Listing 2.24: Group trait (Rust)

2.1.10 Magnolia: magma to group

However, in Magnolia this can be required on the *interface*-level. The example code shown in listing 2.25, showcases a concept representation a binary operation, which has one function, *binop*, which takes in two values of type T, and returns T. Note that the actual implementation of this function is missing. This is because a concept encodes the properties of a users code. The actual implementation of the binary function needs to uphold the properties of the concept that is being implemented. Note that this is unlike the Java and Rust example, in which we have no way to encode the property of our binary function. So any consumer of our API would not be explicitly bound to our restriction of the associativity law 2.1.5, identify law 2.1.7, and the inverse law 2.1.9, required by semigroup and group. The closure definition, 2.1.3, however, can be encoded by the type system in Java and Rust.

```
concept Magma = {
  type T;

  function binop(a: T, b: T): T;
};
```

Listing 2.25: Magma concept (Magnolia)

In the example code shown in listing 2.26, the *magma* concept has been expanded upon, still following the same rules as before, but with the added property of associativity.

```
concept Semigroup = {
  use Magma;

  axiom associative(a: T, b: T, c: T) {
    assert
      binop(a, binop(b, c))
        ==
      binop(binop(a, b), c);
  };
};
```

Listing 2.26: Semigroup concept (Magnolia)

```
concept Monoid = {
  use Semigroup;

  function unit(): T;

  axiom identity(a: T) {
    assert binop(a, unit()) == a;
  };
};
```

Listing 2.27: Monoid concept (Magnolia)

```
concept Group = {
  use Monoid;

  axiom inverse(a: K, b: K, c: K) {
    assert binop(a, b) == unit();
    assert binop(a, c) != unit();
  };
};
```

Listing 2.28: Group concept (Magnolia)

So Magnolia facilitates reuse, and extension of logic. In Magnolia we can build upon existing proven algebraic structures like a group, and reuse their properties. One of the interesting properties of the group structure, is that certain evaluations are the same. In our editor example, the usefulness of this is clearer, as we can do optimizations on the editor actions. Since the user of the editor reacts slower than our code, we can batch several actions together, and evaluate them together. If the user keeps writing and deleting the same sentence over and over again, we could evaluate them as a *no-operation*, and not do the expensive operation of writing and removing text from a file.

2.1.11 Reuse in Magnolia

One of the most important features in any programming language, is the notion of *re-usability*. From the invention of the GO-TO-statement, with which we could repeat code statements N times instead of writing them N times, to functions, where we could run

the same piece of code several times in a program, with different inputs, reuse has been an essential tool for a programmer. It avoids *re-inventing the wheel*, as common functionality can be externalized and reused in several different places. This ensures fewer points of failure. Instead of having to test several different places in a project, we can test the function being used different places.

Reusability is also an important feature in Magnolia, but this reusability is in the entire language. In libraries in other languages, functions are reused, in an attempt to avoid common logical mistakes, but these mistakes could still be there, hiding in plain view. In Magnolia, one can re-use the *logic* of a function. The logging and group example can be rewritten using Magnolia concepts as shown in listing 2.29 and 2.30 respectively, by reusing the concepts we created for semigroup in listing 2.26 and group in listing 2.28.

```
1  concept Logging = {
2    type Log;
3
4    use Semigroup[binop => combine, T => Log];
5  };
```

Listing 2.29: Logging example (Magnolia)

```
1  concept Editor = {
2    type Action;
3
4    use Group[binop => compose, T => Action, unit => noOp];
5
6    function addText() : Action;
7    function removeText() : Action;
8    function redoAction(action: Action) : Action;
9    function undoAction(action: Action)
10 };
```

Listing 2.30: Editor example (Magnolia)

Indeed, reuse is so useful, that in Magnolia one can rename concepts one use. In the listings 2.29 and 2.30, we have renamed the type and function into something that makes more sense in the specific use case. On line four, in 2.29, we have renamed **binop** and **T** to **combine** and **Log**, respectively.

While it is useful for us developing the concept, to know that our logging concept is a semigroup, when using an implementation, this is less relevant. If a consumer of the

logging API read that logs where used everywhere in the project with **binop(log_a, log_b)**, it would be confusing, but when using context specific names, as renaming allows us to do, we could rename **binop** to **combine**, which makes the code easier to read.

2.2 Integrated Development Environment

Before IDEs where the standard development tool, all a developer had, was a terminal, an editor, and a compiler/interpreter. An editor to change the source code, a compiler/interpreter to compile/interpret, and a terminal to invoke them. But not all projects can be handled this way. In larger projects other programs like Make[5] for C/C++, or Gradle[6] for Java/Kotlin, was needed to build, test and package the project. So instead of manually adding each new C file to the compiler argument list, or manually compiling and zipping Java class files, an external program was used. But for C/C++ or Java/Kotlin, external libraries are used, which meant that among different developers, the environment could vary.

This meant that onboarding new developers took time, to ensure they had all the necessary dependencies to develop the project. Which could be as simple as downloading some library or program, to ensuring specific environment variables existed during compile time. Eventually all of these different dependencies, programs, environment variables and such where all integrated into a single application.

An IDE, aids a developer, as all the needed tools for development are integrated into one application. Some are more specialized than others, targeting specific language, like Eclipse and IntelliJ. Others are more generic, like VS Code. We will use the terms generic and specialized IDEs to differentiate the two variants.

2.2.1 Syntax highlighting

Highlighting important keywords, identifiers and more, makes the language easier to read for the developer, allowing them to spot easy to miss errors, like misspelling of keywords, functions, and variables. In the pictures 2.1, and 2.2 we can see the difference between having syntax highlighting. In 2.1 we can clearly see what is, and is not a latex command, if we wrote *begn* instead of *begin*, we can notice the discrepancy, as we expect it to, in this case, be purple. We are using visualization to make developers notice issues.

[5] `https://www.gnu.org/software/make/`
[6] `https://gradle.org/`

```
\begin{center}
  \includegraphics{./pics/syntax-highlightning.png}
  \caption{Syntax Highlighting (\latex)}
  \label{pic:stx}
\end{center}
```

Figure 2.1: Syntax Highlighting (LaTeX in Vim)

```
\begin{center}
  \includegraphics{./pics/syntax-highlightning.png}
  \caption{Syntax Highlighting (\latex)}
  \label{pic:stx}
\end{center}
```

Figure 2.2: No Syntax Highlighting (LaTeX in Vim)

2.2.2 Code autocompletion

Suggesting keywords, method names or even entire code snippets, is a powerful tool an IDE can have. This is possible to achieve, in some form, without being specialized, by for example, suggesting text that already exist in the document, but is most useful if it is specialized, and can suggest built-in methods. This allows a developer to not having to remember exactly how methods are named, is the method to split a string by some delimiter, *split_by* or *split_on*? As long as the developer writes *split*, the correct method name will be suggested.

2.2.3 Go-to-definitions

Being able to quickly navigate to methods and read their implementation is a useful tool for a developer, as less time has to be spent navigating the project structure, to figure out where some method was implemented, and more time can be spent actually developing.

Figure 2.3: VS Code suggesting appropriate methods (Rust)

2.2.4 Formatting and linting

When developing, an important process is code review, another developer ensuring that the suggested improvement is up to some standard, specified by the language and/or development team. Things like naming conventions, code style, unused variables, dangling doc-strings, bad variable names and commented out code, have no effect on the resulting program. Unused variables and comments are optimized away by the compiler, while variable names are mangled.[7] But for developers reading the source code, these issues can hide bugs because there is a lot of *noise*. Luckily, IDEs can detect these common issues, by the help of a *linter*, which can, on the invocation of the user, fix these issues. Linters are opinionated, since programming language specifications specify conventions on how the source code should look. In Rust for example, the compiler has a built-in formatter and linter, which, during compilation, warns the user of these mistakes, and can re-arrange the source code by formatting it to fit the standard.

2.2.5 Boilerplate code generation

An important process in development, are ensuring your code is correct. A good way to ensure this, is with unit testing. For object oriented programming, this is done by testing the methods on a class, and creating the necessary unit tests for this can be tedious. Luckily, IDEs like IntelliJ come with boilerplate code generation, creating a *skeleton* unit test, containing the empty tests ready to fail. An example for this can be seen in 2.4, where we are prompted with a checkbox for each method on the class; whether we want to include it in our unit test or not.

These are features also within a generic IDE, but most are available through the IDE module architecture. A generic IDE contains the features that are common among development across any programming language. But the following features are not exclusive to generic IDEs.

2.2.6 File explorer

Most project nowadays is larger than one file, so being able to visualize the project in a tree-like-structure, and navigate that, is useful. Similarly to syntax highlighting, it adds

[7] Unless they are constants

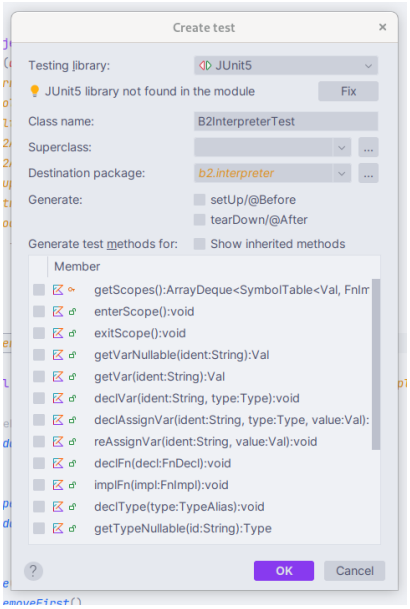


Figure 2.4: Window for generating boilerplate code for a unit test (Kotlin in IntelliJ)

Figure 2.5: File explorer in VS Code showing the Nmide source code

icons to showcase what is a file or folder, but also what file extension is used. In the picture 2.5, one can see how all the Rust files, the files ending with *rs*, have a crab[8], visually showing the developer that this file is a Rust source file. This makes it easier for a developer working on a polyglot project, as they can quickly find source files by language. This feature also comes with the ability to manipulate the project structure, by adding files, folders, moving files around, and deleting them, but most importantly, being able to open the file. By clicking on a file, it opens up in an editor, to be edited.

[8]Called *Ferris*

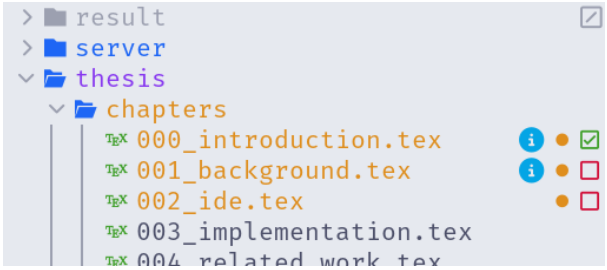


Figure 2.6: File explorer in Vim showing that the *result* folder is not controlled by Git (grey), and that the thesis folder have uncommitted changes, (purple), the files who have changes are marked (yellow), and there is one file which has a committed, but a non-pushed change, (green checked checkbox).

2.2.7 Version Control System integration

Version Control System is an integral part of development. Being able to sync ones work between different machines and developers is essential. It allows for cooperation between different programmers.

In the picture 2.6, we can see an example of this in action. There, we are using Git to version control our project, (this IDE). Vim can detect changes made to the project, with the help of Git, adding colours to our files based on their state, not controlled, uncommitted changes, marked changes, committed change.

2.2.8 Module market and module installation

An important part of the user experience in a modular IDE, is being able to seamlessly add, install and use modules. An integral part of this is the module marketplace. This is a term for the place where a user can find modules, and with the click of a button, install them. Depending on what feature this module adds, the installation process could be as simple as adding it during the runtime, or if it's a more complex or integrated feature to the IDE, requires a restart of the IDE. In the picture 2.7, we can see this in action.

2.3 Zero-core module architecture

A modular application, is an application which can be extended by other pieces of software. This extensibility is useful as features that the original developers of the application

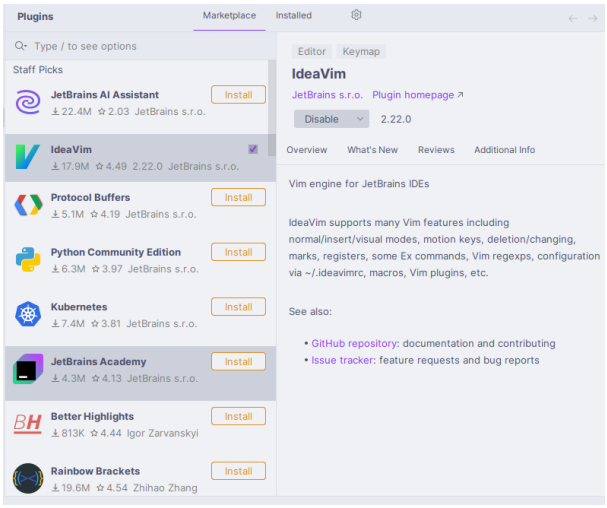


Figure 2.7: Module market window in IntelliJ showing popular modules that the user can install by clicking on the *install* button. Also seen in the window is that the user can disable already installed modules, or uninstall them.

did not think about, can be added. If this module architecture is well-designed, then this extension can be added without changing the core application.

There are different ways an application can be extended. The most common one uses so-called *live-reload*, in which, if a module drastic changes the functionality of an application, the application has to be restarted, or if it is a *minor* change, the module is simply loaded. Commonly, IDEs are extended during runtime. Another method would be *compile-time-extension*, in which modules are added before the application itself is compiled. There are some advantages and disadvantage in both approaches.

2.3.1 Compile-time module

As an example, a standard user of any application will expect the application to come bundled with all the needed functionality. This is best achieved with the *compile-time-extension* method, since the application can be installed with the expected modules during compile time, ensuring the resulting binary contains all the wanted features. This also comes with the benefit of the compiler being able to optimize the module-core interactions, since the module is directly integrated into the source code of the core application.

2.3.2 Runtime module

Runtime modules are usually also interpreted, but they can still be a library that is loaded during runtime. The benefit with a runtime module, is that a user can easily test out

different modules, as compiling the entire application before being able to test a module is a hassle. But it comes with some drawbacks, like having to do extra verifications on top of the module, ensuring invoking the module won't crash the entire IDE.

2.3.3 Zero-core IDE

A zero-core IDE takes the modular architecture that is standard IDEs and takes it to its extreme. When all features are modules, and modules interact by invoking each other, it can mimic a microservice architecture. A module invocation is some *request*, which the invoked module *responds* to. There is a provider-consumer dynamic. There is a provider, some module providing data, and some consumer, some module using the data.

If either the consumer or the provider is maintained by a third party, then there is some informal *contract* between them. The consumer expects the outputs of a provider to be in some certain format, which in the case for REST APIs usually is JavaScript Object Notation (JSON), and with specification formats like OpenApi[9], one can also specify the structure of the JSON response, with what fields and values it can have. One can also specify what values are valid in a request.

But even if one does use such a format, changes in scope can require the provider to change the API, which could affect the consumer, because the consumer assumed something about their informal contract, that all numbers provided are integers. And if suddenly the provider changes, and returns floating point numbers, the consumer could crash trying to parse a string as an integer.

The same concept applies to a zero-core IDE. Modules have an implicit, and informal contract between them. So the same measures, used in microservice architecture, to mitigate these issues, can be used in this zero-core IDE. Unit testing of code is essential in software development, especially when developing against third party systems. Instead, invoking these third party systems at test time, mocking is used. This mocking is part of how a consumer assumes a provider should act. Mocking of modules is trivial, as all one needs to mock is the state, the User Interface (UI), or that some event happens, triggering a module. At a larger scale we test the module family, to see if its change has affected the other modules.

This can be done with contract testing, as discussed by Gross and Mayer [11]. They propose a component architecture, where each module exposes some testing interface.

[9]https://swagger.io/docs specification/v3_0/about/

Table 2.1: Module Ecosystem per IDE

IDE	Module count
Eclipse	$\sim$ 1200
IntelliJ	$\sim$ 9500
VS Code	$\sim$ 71700

The same principals they discussed, while for a different, but similar architecture, can be applied to ours. But instead of having each module have a testing interface, we can instead load all modules composing the IDE in a test environment, where all interactions are recorded, and used to generate a dependency graph, showing what modules depend on whom. This can show that certain modules are more tightly connected than assumed, meaning they are in the same module family, it can also show what module families communicate with other module families, showing module communities.

2.3.4 Module ecosystem

In modern IDEs, with an extensive module architecture, there exists a vast module ecosystem. From simple modules that change the color scheme, or add file icons to more complex modules that add support for other languages. A good variety of a module ecosystem can help ensure the longevity of an IDE. In the table 2.1, we can see that IDEs have an extensive module ecosystem.[10]

VS Code is a popular IDE [5], and this could be due to the amount of modules it has to offer. The amount of modules attributed to VS Code could be the cause of a positive feedback loop. VS Code is popular because it has many modules to extend the functionality, making it able to cover many use cases. Since VS Code is popular, developers use, and make modules for VS Code. All of which strengthen the longevity of the IDE.

2.3.5 Granularity

When designing modules, the *granularity* of the combined modules has to be considered. As an example, if one where to extend the zero-core application with the needed functionality for it to be considered an IDE, this could be achieved by creating a singular

[10]Data found by looking at the marketplace for the modules, in order:`https://marketplace.eclipse.org/content/welcome-eclipse-marketplace`, `https://plugins.jetbrains.com/`, `https://marketplace.visualstudio.com/search?target=VSCode&category=All%20categories&sortBy=Installs`

module which does all the work. However, this is not a modular approach, as if one wants to change some specific feature in the IDE-module, one would have to re-create the whole module with that specific feature implemented. Instead, if this functionality was granular, that is to say, split into several modules, that together enable the needed features, then it would be *simpler* to modify the needed modules to achieve the wanted feature.

2.3.6 Module family

A module family are several modules enabling a single *feature*. A user of the IDE might think that being able to browse the project using a file explorer integrated into the IDE as a single feature but, in a modular system that facilitates reuse, this would be made up of several different modules.

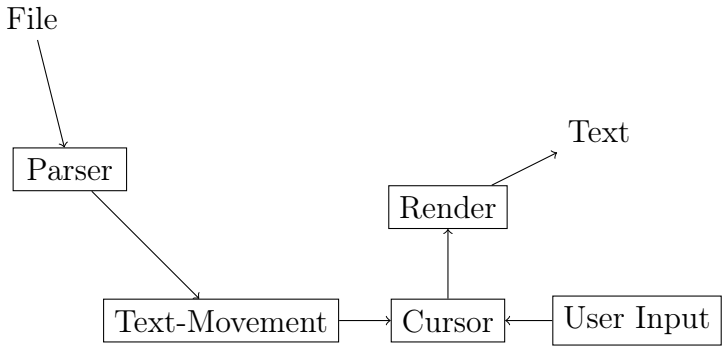


Figure 2.8: Diagram of a module family for a text editor

In figure 2.8, an input file is parsed to some structure which is used to translate user actions, into cursor movements. The cursor being the place in the file where text is written to by the user.

Module families naturally shows up in a *true* modular system. If several modules together enable some feature, then those modules can be treated as a singular module by an external module developer, depending on what they want to extend.

2.3.7 Existing Magnolia IDE

The current IDE for Magnolia [2], is a many-years-old version of Eclipse, using modules and functionality from the core Eclipse application, that has since been outdated.

The IDEs lifetime was limited by a dependency on external modules and features that where not maintained by the IDE-developers. This meant that for future development of Magnolia, an outdated IDE was needed, with outdated tooling. Furthermore, the Magnolia compiler was implemented as an Eclipse module, which means that development is limited to Eclipse, and only Eclipse, as a developer cannot compile Magnolia code without it.

Modularization will help to mitigate some of the issues with the current Magnolia IDE. Instead of maintaining an entire application, the needed and wanted features of the application can be maintained instead.

Experimental languages might have features which are not possible to be fully used in current IDEs. This is also the case for the current Magnolia IDE. The compiler for Magnolia, syntax highlighting, error reporting, and hover-functionality are functionality made in the Eclipse IDE, by using its plug-in architecture. Some of the functionality and plug-ins this implementation used, have been deprecated in later version of Eclipse. This means the Magnolia IDE is locked to an old version of Eclipse, which, as time passes, increases the complexity of installation, as the surrounding tooling and libraries needed by this version of Eclipse also becomes deprecated. Currently, in INF220, a course at the university, two weeks are set aside for students to be able to install the current Magnolia IDE.

2.4 Challenges imposed by Magnolia

In most programming languages, any type has a singular definition, retrieving the file and position in the file, where a type is implemented, is trivial, as there is only one position to show or go to. However, in Magnolia a singular type could have multiple definitions, and resolving this can be complex. Especially communicating this to a user in a helpful manner.

2.4.1 Renaming

Given our definition of a monoid[12], one can trivially see that string concatenation and list concatenation falls under this definition, and are therefore related. It is therefore quite

reasonable to reuse the monoid concept, within the string and list concepts. Along side their respective methods, like **toUpperCase** or **map**.

Even though they are related, it is more useful to have specific names for each concept. In 2.31 and 2.32 we are importing the monoid[11], and renaming the **unit** and **binop** operation to something that are specific to the concepts, **emptyString**, + and **emptyList**, ++ respectively.

```
concept StringConat = {
  use String;
  use Monoid[
    T => String,
    unit => emptyString,
    binop => _+_
  ];
}
```

Listing 2.31: String concatenation (Magnolia)

```
concept ListConat = {
  use List;
  use Monoid[
    T => List,
    unit => emptyList,
    binop => _++_
  ];
}
```

Listing 2.32: String concatenation (Magnolia)

In Magnolia, it is useful for a developer to visualize and follow the effect of renaming, that the compiler does. This visualization would enable a developer to know where a renaming happens. The structure of this is tree-like, and not functionality that is typically included in an IDE.

2.4.2 Dependency cycles

Programming languages have different ways to avoid the problem of imports of modules forming a cyclic graph. The easiest, is to simply disallow such import structures, which

[11] Definition 2.1.8

is something the Magnolia compiler does. All imports have to be Directed Acyclic Graph (DAG)'s. In most programming languages this is trivial to solve for developers, as if suddenly a project has a cyclic import, it can be solved quite easily. However, due to the heavy reuse in Magnolia, the cycles could be quite large and harder to reason about without a tool to visualize the dependency graph.

2.4.3 External software dependency

Magnolia depends on a compiler, like all compiled programming languages, but also an SMT solver. While the new compiler for Magnolia, at the time of writing is still under development [19], but once released will be quite stable. Writing modules that utilize the new compiler can be tightly coupled with the compiler. This is in contrast to the SMT solver environment. Skogvik [17] noted the different competitions for developers of SMT solvers, this means there might be a new and better SMT solver, which means it needs to be easy for a user of the IDE to change the SMT solver they are using to validate their program with.

Chapter 3

Magnolia IDE

This chapter will discuss the new Magnolia IDE, the different users of the IDE, and their possible experiences which was under consideration when developing the application. We will start with section 3.1 by covering the different users that exist in IDEs today, and how our architecture changes this user dynamic. In sections 3.2, and 3.3, we will cover the IDE users and module developers of our architecture, respectively. Specifically, we will address what the different users expect from an IDE in general, and what they might expect from ours. We will also cover how some of the different modules and tools we have implemented can be used to improve the user experience.

3.1 User Perspectives

This application has to consider different users. In IDE's like Eclipse or IntelliJ, there is the primary user base, the developers who are using the IDE to develop, and then there are the secondary user base, the developers whom develop modules for the IDE.

Being the primary user base, most of the new features implemented by either IDE are related to the development experience. There are still changes to that the module developers are interested in, namely API changes. IntelliJ for example, lists their *incompatible API changes*[1].

[1] `https://plugins.jetbrains.com/docs/intellij/api-changes-list-2025.html#intellij-platform-20252`

Breaking changes between IDE versions is something normal users of the IDE do not worry about. As usually when a new version is released, it means more features for the developer to utilize. While module developers have to ensure their modules still work. One of the reasons behind IDE version changes can break a module, is due to how they interact with their IDE. In IntelliJ a module is created by implementing a Java interface for the functionality one wants. This means a change in the interface can break the module.

A change in the IDE architecture could break a *plugin* for the newer version of the IDE, as with Bagge's Magnolia IDE [2]. While in this zero-core IDE module developers are quite important, as they are the ones who add the functionality to the IDE. Therefore, the core API has to be more stable, and it is by virtue of not having much functionality.

3.2 IDE users

As mentioned in chapter 2, modern IDEs come with an integrated module architecture. Which is used to extend/change the IDE, from as simple as to change the theme, to more drastic changes, like changing all key binds to *vim-motions*. In any case, a user expects certain functionality to already exist in an IDE, like text editing. A maintainer of a zero-core IDE could supply modules added at compile time, meaning the expected functionality is there out of the box, while more thematic modules could be supplied as runtime modules. Furthermore, a maintainer can supply more framework-like modules, which enable other modules to easier implement their functionality.

We have implemented a few modules, some more functionality based, other more framework-like. In the table 3.1, we can see these modules, their names, and the functionality they supply.

Table 3.1: Table of modules

Module name	**Functionality**
ide_framework	Creates a UI framework other modules can use
ide_explorer	File explorer functionality
ide_pm	Adds the menu bar, with buttons and dropdown
ide_tabs	Responsible for the tabbing system
ide_editor	Editor
ide_fsa	Exposes File System Operations (FSA) to other modules

ide_framework This module sets up the general UI layout, which other modules depend on. In the figure 3.1, we have laid out the naming convention we use when referring to different *places* in the IDE.

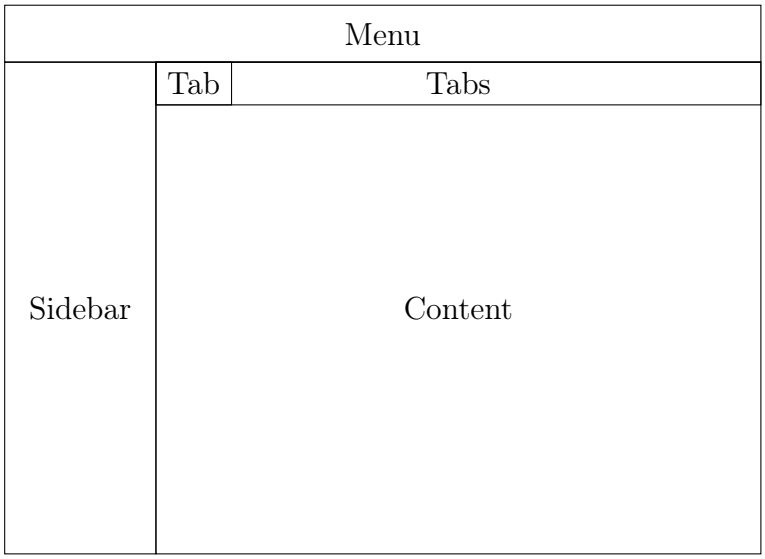


Figure 3.1: Diagram of the layout of different areas in our IDE

ide_explorer In picture 3.2 we can see the module, *ide_explorer* in action, showing the Magnolia library visualized as a tree-like structure with collapsible folders. These folders are rendered in the *sidebar*, on the left. When we click on the *File* button in the menu, a dropdown appears, where we can click on a button, *Open Folder*, which invokes the ide_fsa module, the module in charge of handling file system operations, where we get in *response*, all the folders and files in the path we selected. Which we transform into HTML, and along with some Cascading Style Sheets (CSS), we get the collapsible folders.

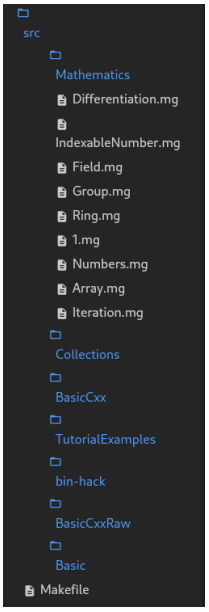


Figure 3.2: ide_explorer module, showing the Magnolia library.

ide_pm Is a module responsible for the menu bar at the top of the IDE. It just simplifies the creation of interactive UI elements for other modules. The Module has functionality for handling dropdown menus, which are common in IDE-UIs.

ide_tabs This module handles pagination of the IDE, where other modules can add their own content to different tabs, that this module can cycle through. By clicking on a file in the file explorer, a tab is made, where the contents are managed by the *ide_editor* module.

ide_editor The editor module is coupled with the IDE framework, and the ide_explorer module. We can open and edit files using the combination of these modules. By clicking a file in the tree, we can invoke the ide_editor module, which invokes the ide_tabs module, creating a tab with a text editor in. In picture 3.3 we can see this in action, as the editor is created in the *content* place, in the center of the IDE, along with a *tab*, with the name of the file being edited as the title of the tab.

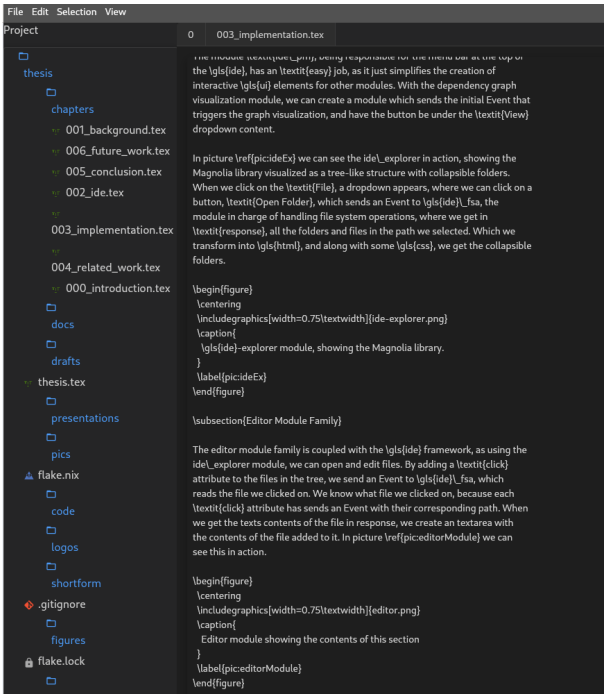


Figure 3.3: Editor module showing the contents of this section

ide_fsa Since this IDE can target different Operative System (OS)es, we need some form of File System Abstraction (FSA). This is achieved by our module, ide_fsa, which enables other modules to do file system operations without having to worry about what OS they are on.

3.2.1 Magnolia dependency graph visualizer

In Magnolia, as in many other languages, one cannot have a cyclic dependency. This means that the dependency graph of a Magnolia project should be a DAG. And since Magnolia has such a focus on reuse, the dependency graphs in a Magnolia project could be quite large. Which means the cycles could be quite long, which would make resolving the cyclic dependency issue complicated. One way to help a developer, would be to give them a tool to visualize the dependency graph, so that they could see what modules are connected. Using the Magnolia library as the input, we can create a visualization of the dependencies in Magnolia. Using two modules, one for *parsing* the Magnolia library, finding all packages, and their dependencies, and another for visualizing this.

The module responsible for rendering the graph, uses *D3*[2], a visualization library for JavaScript. In the picture 3.4, we can see the finished rendering of the dependency graph

[2] `https://d3js.org/`

of the Magnolia basic library. As mentioned earlier, Magnolia has a lot of re-use, and therefore dependencies. That makes this visualization quite *noisy*, as there are a lot of crossing between the dependencies.

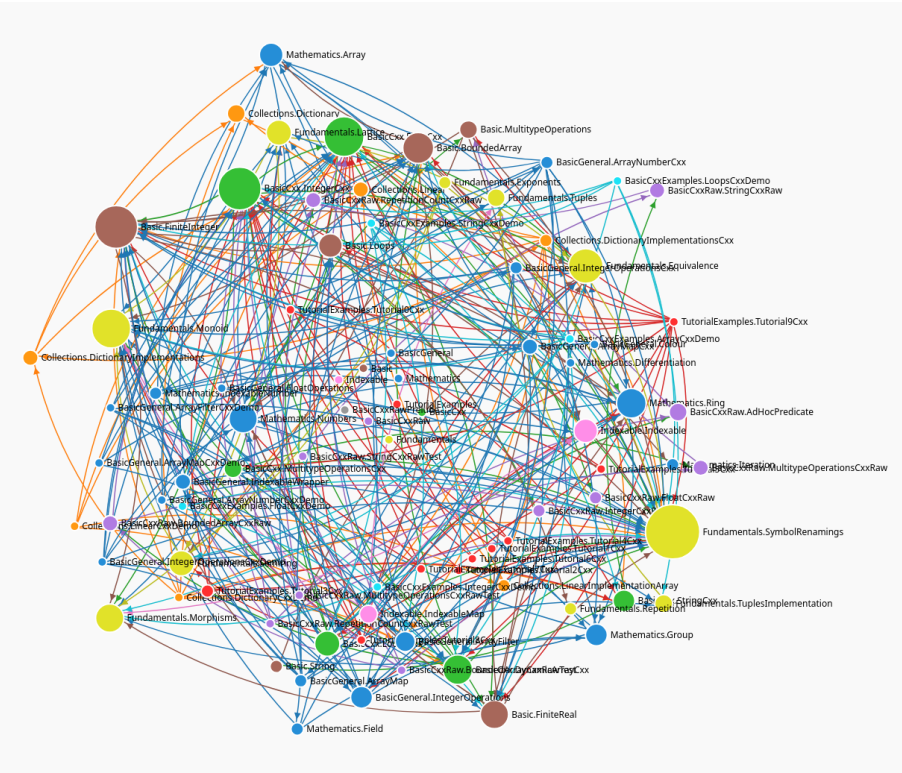
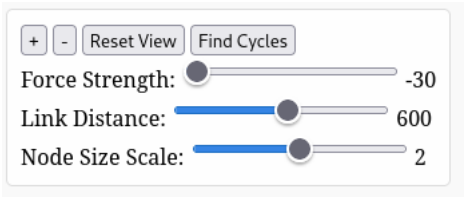


Figure 3.4: A module that visualizes the dependency graph in the Magnolia basic library. Each colour represents a package, which contains several modules. The size of the nodes vary depending on the amount of dependents a module has. The module utilizes an external JavaScript library to create this visualization.

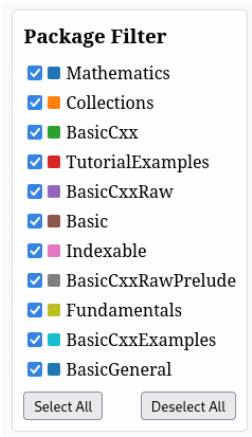
Luckily, with D3 and CSS, we can mitigate some of the noise. In the picture 3.5a, we can see the control-panel that our graph module has created. With the control panel, we can zoom in and out on the graph[3], and reset our view. This, along with the node size scale, scaling how big a node is depending on how many dependents it has, ensures this visualization tool can be used for other programming libraries, not just Magnolia.[4] Furthermore, we can highlight the packages we care about, using the filter panel the module created. In picture 3.5b, all the different Magnolia packages have been detected, and their corresponding colour has been added. We can then enable, or disable them.

[3]This can also be done with the mouse

[4]Given a proper parser module, for the target programming language



(a) Control panel, with buttons and sliders for controlling the graph view



(b) List of packages in the graph, that can be toggled

Figure 3.5: The control panel and package legend, created by the visualizer module.

In the picture 3.6, we can see the graph after we have disabled all other packages, except the *Fundamentals* package.

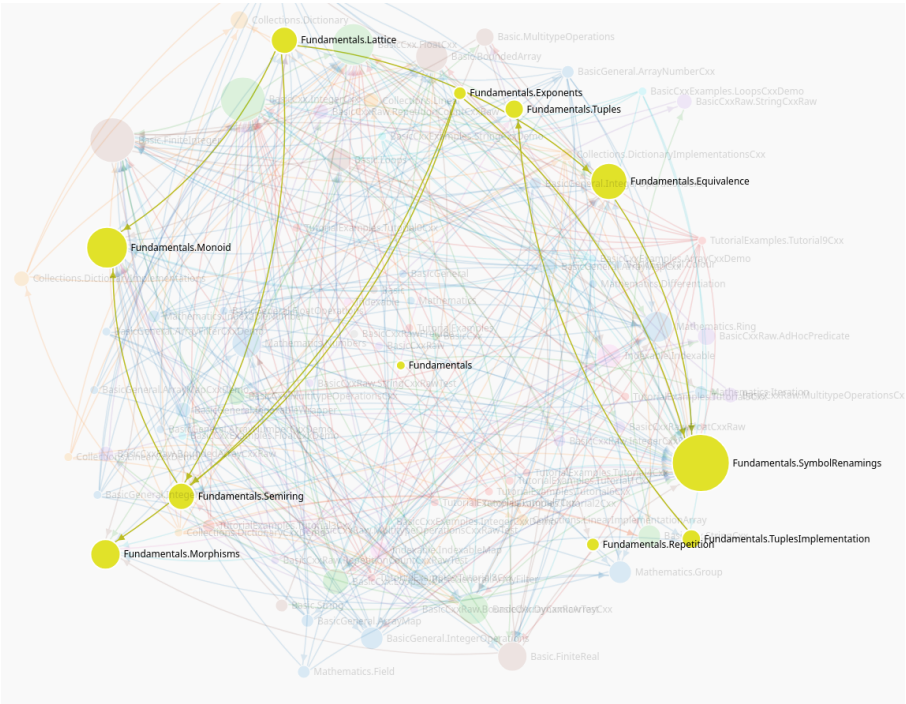


Figure 3.6: A module that visualizes the dependency graph in the Magnolia basic library, with just the *Fundamentals* package highlighted.

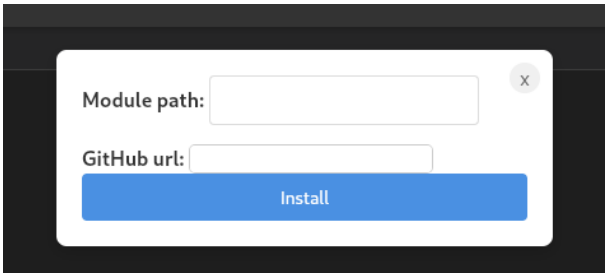


Figure 3.7: A module installation form, where a user can supply a path to a module on disk, or a URL to a GitHub repository containing the module binary.

3.2.2 Developer

Most users just want an IDE, and do not spend, nor want to spend, much time configuring their IDE. This can be achieved by adding the necessary modules to qualify as an IDE at compile time. If one is a lecturer, teaching something that is used by a *niche* programming language, the lecturer can add the needed modules to a configuration file, *Modules.toml*, and then compile it to an IDE. Before the IDE is compiled, it finds the mentioned modules in the configuration file, and directly integrates them into the core, ensuring that the resulting binary is a fully fledged IDE. And then this IDE can be distributed to the students, who can still extend the IDE with runtime modules at their own digression.

3.2.3 Module installer

Module markets are an important part of the IDE user experience. Being able to install a module with the click of a button essential. Not having a dedicated module market like VS Code, has not been detrimental for Vim, as with a *plugin manager*, a user can install a module by simply supplying a URL to a GitHub repository. Similarly, we allow users to either install modules from disk, or by supplying a URL to a GitHub repository. In the picture 3.7, we can see such a form. A user can either install a module from disk, in which case, we simply copy the module binary at the given path to the runtime module folder, or if a URL has been supplied, we get the latest released binary from that GitHub repository, and copy this binary instead.

3.3 Module developer

Being a zero-core application; all functionality comes from modules, the module developer experience is the most important. To achieve this, documentation is important. If a

module developer has a question about how the core might react, it should be answered by the documentation. In Eclipse, this is in the form of *Javadoc*s, which specify, with examples how the Eclipse runtime handles *plugins*[5].

The documentation for *plugin* developers, in both IntelliJ and Eclipse has to be large, due to of how *plugins* interact with the IDE; it is a large API. In a zero-core IDE it is smaller, simply due to the fact that the core IDE offers fewer features, as features necessarily come from modules in a zero-core architecture.

3.3.1 Language agnosticism and foreign modules

A limiting factor in module oriented applications, is the *language barrier*. Most applications limit what language one can extend an application with, like in VS Code, where its JavaScript/HTML/CSS. Or IntelliJ, where one can use Java or Kotlin. But what does language agnostic mean in the context of our modular architecture? Semantically alike modules.

Translating a module from one programming language to another should be trivial. This is achieved by the models used in the core. The *primitive* types, are the same as in JavaScript, the notion of an empty value, numbers, strings, lists and *objects*, can be serialized/deserialized to/from any language. So the manipulation on these types can be extracted and rewritten in another preferred language. Therefore, to be fully language agnostic, modules should be syntactically translatable between each other. The same two modules, one implemented in JavaScript, the other in Rust, should be semantically the same.

Of course, this language agnosticism also extends to our libraries. Utility functions, for manipulation the UI or the primitive types in our state, should be equivalent, par for naming conventions, as this is a syntactical difference, not semantically different.

3.3.2 Module developer tools

When developing against a module architecture, having tools to help debug issues is useful. Common issues when developing in a modular architecture, where modules can

[5] `https://help.eclipse.org/latest/rtopic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/core/runtime/Plugin.html`

invoke other modules, are incorrect invocation, as in invalid arguments or return type. Being able to manually invoke modules during runtime is a great tool for debugging. In picture 3.8 we can see this a prototype module where module developers can manually invoke other modules.

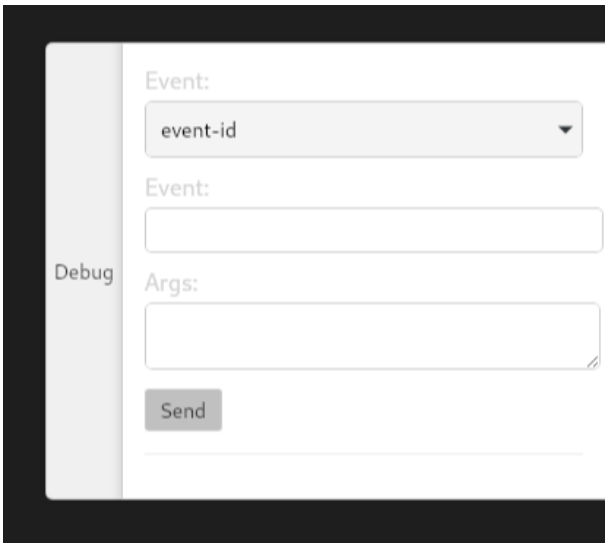


Figure 3.8: Module that adds a simple pop-up menu for manually invoking other modules

When mocking in REST-API development, one creates the expected response, which is usually a JSON-file. The same is done here, where the *Args* field in this form expects the argument to be in JSON. This is helpful, as other testing libraries, like *Playwright*, does the same and the IDE logs the arguments in the same formatting, meaning we can simply copy-paste the argument we want to mock from the logs, into the field. Another helpful tool, is the one shown in the picture 3.9. Here we can see a module which visualizes the current state of the IDE.

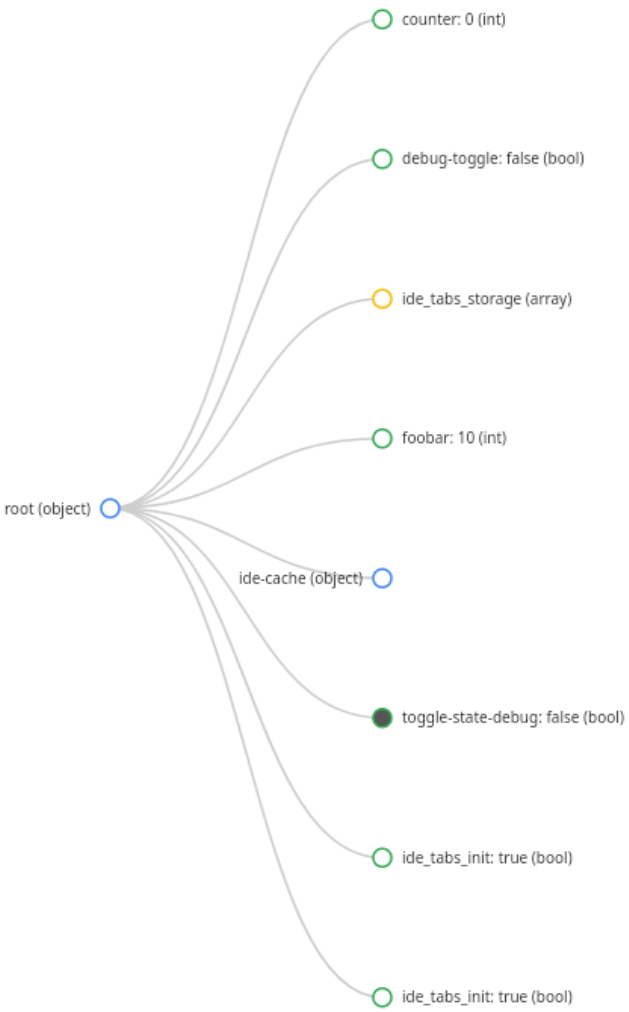


Figure 3.9: Module that creates a visualization of the current IDE state. The visualization is made using an external JavaScript library.

3.3.3 Module dependency visualization

When developing against a modular architecture, it is useful to be able to see what different module families appear, and what the different dependencies between the modules are. In picture 3.10 we can see the resulting graph of the IDE modules.

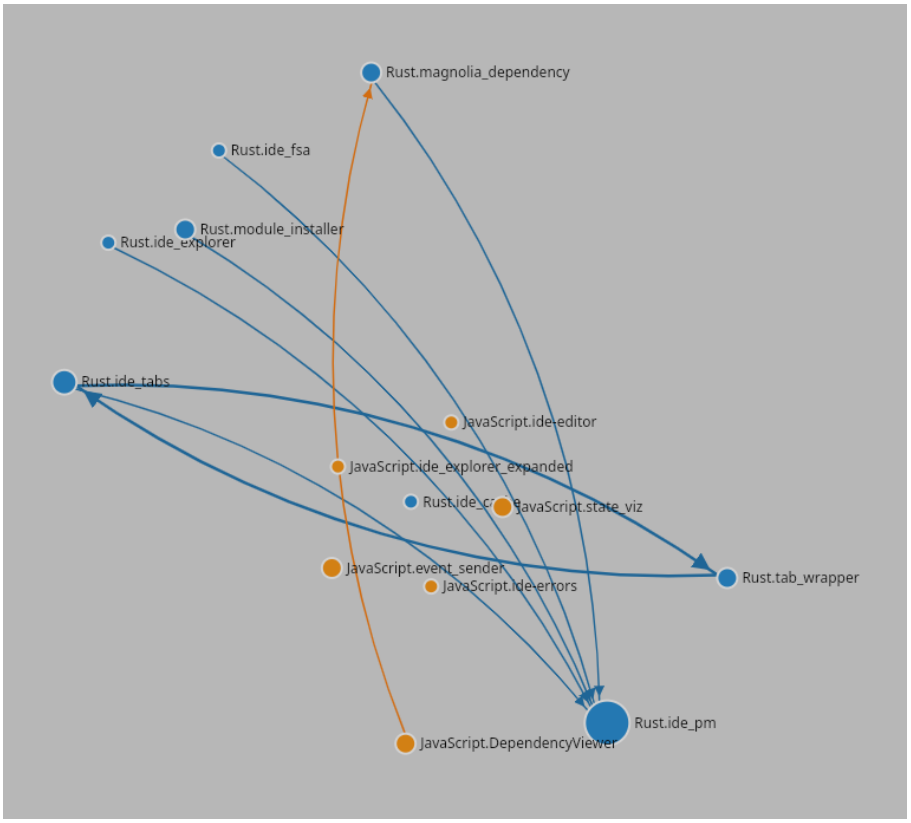


Figure 3.10: The different modules and their dependencies. Graph was created by using the same module shown in picture 3.4.

Chapter 4

Implementation

This chapter will focus on the implementation of the zero-core IDE. In section 4.1, we will mention technologies used, and why they were chosen. In section 4.2 and 4.3, we will discuss the different iterations the application architecture had, and why they were sub-par, compared to section 4.4, which is the implementation of the zero-core IDE. Section 4.5 will explain the necessity of testing when using such a modular design, and explore the ease of which functionality can be tested in such a modular architecture. Section 4.6 and 4.7, will discuss the implementation of IDE specific functionality, how modules are installed, and module development tools, respectively.

4.1 Tech stack

A module can extend an application at either compile time, or during runtime. This could be achieved by using an interpreted language like JavaScript or Python. The issue with using a dynamically typed language like Python or JavaScript, is that it enhances the risk for runtime issues occurring, and when dealing with scenarios like writing to files, or running long processes like compiling a program, it is important to avoid such issues. So using a type safe language, that can *transform* runtime errors into compile time errors, is preferred. Furthermore, to be able to support modules in foreign languages, necessarily means that the core IDE needs good Application Binary Interface (ABI) support, and therefore should be implemented in a low level language. But what does *low level* language mean? And what is an ABI?

4.1.1 Low level languages

Programming languages has changed over time. In the beginning, a program was a series of ones and zeros, representing instructions a computer should do. Since then, we have moved several abstraction layers above what is commonly referred as *bare metal* programming. From writing in hexadecimal instead of binary, to machine instructions, to more generic programming language, like C. What was different with C, compared to writing direct machine instructions, was that an external program, a compiler, could translate C code to machine instructions specific to the computers' Central Processing Unit (CPU) architecture, this meant a single program, written in C, could be compiled to many different computers. So, at the time C came out, it was considered a *high level* programming language, because the language a developer was writing in, had a higher level of abstraction.

Today this notion of *low* and *high* level languages has changed. A *low level* language is close to how a CPU *thinks*, which has traditionally meant that C is a low level programming language, but some [6] argue that this is no longer the case. In any case, we will use *low level* to mean a programming languages like C, where direct memory manipulations is a feature of the language.

4.1.2 Application Binary Interface (ABI)

An ABI is an low-level interface, a kind of API, between two programs. Such as C program and its dynamic library dependencies. The ABI defines how data is laid out in memory, how functions are invoked, and other machine level details. Both the C program and the dynamic libraries must agree on the ABI, otherwise misinterpreted data or invalid function calls could lead to Undefined Behavior (UB).

Undefined behavior In programming UB occurs when a program violates the language specification in a manner that is not defined by the specification. This can be the results of ABI mismatch, like if the layout of a struct in memory differs between the parties, in a manner which leads to breaking of type safety, or direct violations of the language rules, like null pointer dereference. UB is dangerous because the compiler might optimize the binary unpredictably or the program may behave arbitrary. It is also a vector of attack for hackers.

4.1.3 Rust

Rust is a general purpose programming language, designed for, amongst other things, type safety, memory safety, and concurrency. When programming in Rust, the bugs common in other languages, like null pointers, buffer overflow and data races are detected at compile time. Most of these are features of Rusts ownership rules. These rules, enforced by the compiler, ensure that values are safely dropped[1], this ensures that all variables referenced in Rust have a value, and can be safely evaluated. It works by simply dropping values when they are out of scope.

The example in listing 4.1 showcases the ownership. The **name** variable is declared, and used as an argument in the **greeting** function. We cannot call the function again with **name**, since at the end of **greeting**, before it returns, **name** is dropped, since once we called **greeting**, the **main** method no longer *owned* **name**, as the ownership was transferred to **greeting**. We could *fix* this by changing the argument type from *name: String*, to *name: &String*[2] and adding the borrow symbol to the argument in the method invocation, as shown in listing 4.2.

```
fn main() {
    let name: String = "Nils".to_string();
    greeting(name);
    // Not allowed
    greeting(name);
}

fn greeting(name: String) {
    println!("Hello, {}", name);
}
```

Listing 4.1: Ownership example, this code snippet will not compile because we are violating the ownership rules. (Rust)

[1] Called *freed* in Rust

[2] Since String is a dynamic heap string type, when we have a reference to it, it is equivalent to an *&str*, being a reference to an immutable sequence of bytes.

```rust
fn main() {
    let name: String = "Nils".to_string();
    greeting(&name);
    // Now this is allowed
    greeting(&name);
}

fn greeting(name: &str) {
    println!("Hello, {}", name);
}
```

Listing 4.2: Ownership example with reference (Rust)

Now, the *greeting* function is borrowing the variable, which means restrictions are placed by the compiler, on what the function can do. It cannot mutate the variable, which means that several different functions can read the same data concurrently. We can also only mutate the variable if there are no one borrowing it. This principle ensures the other mentioned features of the language, including performance, can be guaranteed by the compiler.

Another Rust feature are so-called *macros*. A macro is some code that is evaluated and executed at compile time, that may change the source code. An example of this, can be seen in listing 4.1. The **println!** is a macro invocation. **println!** is used so that the developer does not have to format the expressions being used, this is handled by the macro. The macro knows the type of the different variables passed, and therefore how to print it to the terminal. Macros are helpful because redundant work can be automated.

Furthermore, Rust has good cross-platform support, ensuring we can write OS agnostic code, and compile it to specific targets, without much hassle. Since Rust is low-level, it has good bindings to C, ensuring compatibility with future models, made in other languages, by use of the Rust ABI.

Rust Application Binary Interface

Rust's ABI is not stable, as it is not supported by their semantic versioning. This means even a bug fix in the compiler, could break the ABI. So if an application, written in Rust, is compiled in version 1.8.0, if this application relies on a Rust library that is compiled in version 1.8.0, everything is okay. But if the application is later recompiled with a

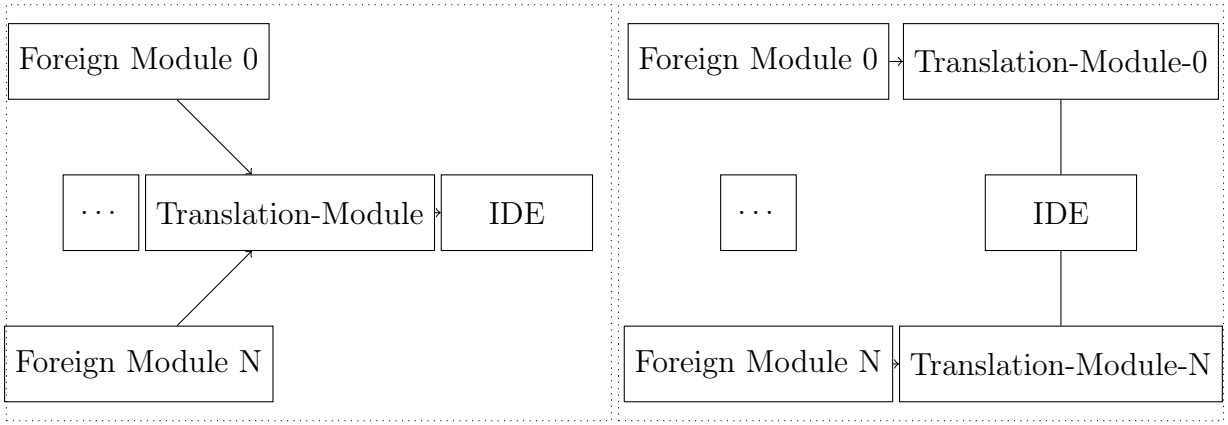
compiler in version 1.8.1, then UB could occur. One of the ways UB was avoided, was using the *abi_stable* crate[3], which enables *safe* loading of external libraries.

With the abi_stable crate, if the types in the IDE change, either by expansion or renaming or such, the crate would crash the application during startup, because the existing module would have a different expectation of what types existed, which again, could lead to UB. But, this due to the implementation of a runtime module using the *abi_stable* crate, as one could design a module to be expanded in the future, safely. This would mean we could only add new functionality to the module, not change the old ones. But due to the stability of the API, this was deemed unnecessary.

An interesting side effect of using the abi_stable crate, is that the crate *forces* the types and functions to be implemented in a certain way, and that way meant that they are closer to the C-ABI. Which means integration with foreign languages are easier.

Module ABI

Any programming language that has C bindings, can create a module, as that is the bridge between the original, *foreign* language, and Rust. We could bridge the languages by simply creating a module which does the translation to and from the different languages, either for each foreign module, or a singular one, acting as a translator, translating the data flowing between the core and foreign-modules as shown in figure 4.1a and 4.1b respectively.



(a) Foreign modules being invoked by a singular translation-module

(b) Foreign modules being invoked by an individual translation-module

Figure 4.1: Two diagrams showing the different methods for implementing a translation module

[3] `https://github.com/rodrimati1992/abi_stable_crates/`

A Haskell module, for example, would require some translation module, either a unique translator for each module, where the Rust module would start the required Haskell runtime[4], when invoking the Haskell module, and exit before returning, avoiding UB. The reason Haskell is of interest, is that the new compiler is written in Haskell [19], and as such, if one where to develop modules capable of features like error reporting, go-to-definitions, and similar for the IDE, a translation module from Rust to Haskell could be necessary.

4.1.4 Tauri

Tauri[5] is a framework for Rust, which enables us to create a cross-platform application. Any frontend framework that compiles to HTML, JavaScript and CSS can be used as the Graphical User Interface (GUI). Such a GUI is commonly referred to as a *web view*. This framework also adds support for invoking Rust methods in the frontend framework, and vice-versa. This allows for support of JavaScript modules, without much fuzz. Tauri archives with Inter-Process Communication (IPC), which allows for isolated processes to communicate securely. For JavaScript to Rust, this is achieved with something called *Commands*, which acts as an abstraction on top of the IPC, which turns the invocation to a frontend-backend architecture.

Tauri also comes built-in with a lot of utilities, for when it comes to application development, like bundling the application for different OSes, abstracting away file system operation, making them independent of OSes, general application security, and creation of dialogs. These dialogs allow for, amongst other things, a user to chose file paths. A user can be prompted with a file selection, click on a file, and click open, and this will be returned as a path in Tauri, which we can use to manipulate the file.

4.1.5 Security

This framework gives a lot of security which is needed in an application which runs third party code. An example of this, would be the so-called *isolation pattern* that Tauri supports. Since we allow for evaluation of JavaScript code in the IDE, we allow for third-party-code to access all of Tauris API. This API is quite powerful, allowing for features such as access to the system shell, the file system, etc.

[4]https://ghc.gitlab.haskell.org/ghc/doc/users_guide/runtime_control.html
[5]`https://tauri.app/`

Being powerful, it is dangerous to expose this to third-parties, but we can intercept and modify all Tauri API calls sent from the JavaScript side, before they reach Tauri. We can therefore, depending on the perceived threat or sensitivity of a Tauri API call, choose to disregard the invocation of the API. We could also choose to disregard *all* invocations that comes from the JavaScript side, effectively only allowing Tauri API invocations through the IDE itself, meaning only Tauri functionality exposed by the IDE itself are callable.

This of course has *some* performance implications, as we now have some overhead on each Tauri API invocation, but this is negligible[6].

TypeScript

TypeScript offers a lot of features over JavaScript, amongst them being able to *type* functions, ensuring null/undefined-safety, at least in our own project. Making TypeScript an ideal candidate for our frontend implementation. Furthermore, by using crates like *ts-rs*[7], Rust types can be annotated with attribute macros, which create a one-to-one mapping between the Rust type, and the serialized JSON object, to be used in TypeScript, allowing for even more type safety, and ensuring that the types used in the IDE only have to be defined one place.

Since the UI is managed by JavaScript, it means any JavaScript library can be used in our IDE. Allowing for any JavaScript library to be used, enabled a low development time of UI components. Since UI is a well explored field, there exists a lot of existing UI libraries, especially for JavaScript. Npm[8], for example, is a package registry for JavaScript. It contains around 3.4 million libraries, all of which are usable in this architecture. If the functionality that these libraries are useful for the application, is another question. This functionality allows for quick development time for modules, which means features that are standard in IDE can be quickly and easily added. But this also means we have to validate modules written in JavaScript, as we can only guarantee the safety of a module during runtime.

[6]According to Tauri themselves `https://v2.tauri.app/concept/inter-process-communication/isolation/#performance-implications`

[7]`https://github.com/Aleph-Alpha/ts-rs`

[8]https://www.npmjs.com/

Module validation

Running third-party-code can dangerous. If this code is not validated or does not come from a trusted source, it could be an attack vector. Luckily, Tauri does some of this work for us, allowing us to analyze all module to core communication, but even if we have validated that a module comes from a source the user trusts, we still need to ensure the module is implemented correctly.

The Rust compiler can ensure that the Rust modules are valid during compile time, for runtime this is a bit trickier. But for JavaScript modules, which the IDE supports out-of-the-box, this is a bigger issue. This lead to the development of two systems. Rust Module System (RSMS) and JavaScript Module System (JSMS).

It was necessary to distinguish the different module systems, due to the way they would be loaded and invoked by the core application. Since the core is written in Rust, the RSMS doesn't have to do any validation or translation when communicating with compile time modules. With runtime modules this also ended up being trivial, with the abi_stable crate, but will be discussed more in depth later.

In the JSMS, managing of modules can lead to exceptions being thrown. Since third party code is being run, nothing can be trusted. All module invocations and outputs needs to be sanitized before it can be used in the core application. This is achieved by wrapping all invocations in a *try-catch*, and using the *io-ts*[9] library to decode types during runtime.

This enables us to safely invoke modules, as we can translate all computations into a product type, where it is either a success, giving us the wanted computation from the module, or an error. But even with types, we cannot verify functions. Since during runtime, we are in the JavaScript environment, we can only validate if something is a function, using the `typeof` operator. It is possible to do *some* verification on functions in JavaScript, but this can only tell us that it is a function. Nothing about the typing of the function can be ascertained at runtime, without explicitly invoking the function.

[9] `https://github.com/gcanti/io-ts`

4.2 Module v1

We did not attempt at first, to create a zero-core application; this was a *natural* conclusion to the existing problem. The first attempt was a simple generic IDE, in which the module architecture was a concern from day one of development. The general plan was this:

1. Create an IDE
2. Extend the IDE, to allow for a module architecture
3. Modules call the application using some Domain Specific Language (DSL)

Since any JavaScript frontend framework could be used, React was chosen, one of the reason for this choice was due to its popularity, which again, would speed up the development time of the application, atleast the UI. But also due to the way React renders the HTML. Between two different re-renders of the application, React can check the difference between the Virtual Document Object Model (VDOM), which is React's representation of the Document Object Model (DOM). It then only changes what is needed in the DOM, instead of re-creating the entire DOM, which makes the render time quick.

This was the more straight forward way to work, because as we could model it of existing IDEs, like VS Code or Eclipse. Another advantage is that when implementing the application, one necessarily gets a better understand of how eventual modules should extend the application.

When we came to the final step, it immediately became clear that naively creating a DSL for functionality was impractical. We started with the idea of, try to implement a feature of the IDE, so extend the DSL, but this approach did unfortunately not lead to a truly modular application. Similar issues to existing IDEs, how does one allow for *everything*? Furthermore, anything created this way, would be subpar to existing software, which would lead to the next maintainer having to fix the core application. This in turn, would add a lot of complexity, which the maintainers would have to deal with.

4.3 Module v2

- Everything is a module

Instead of developing features that make up an IDE, and attempting to ensure it is implemented in such a manner that it can be modified in the future, make everything modular. The only thing the IDE can do, is to manage modules. All features, from the file explorer to the text editor, everything is a module that can be enabled or disabled. A zero-core, modular architecture.

4.3.1 The Elm architecture

An inspiration for the new module architecture is Elm-Lang [7]. Elm is a functional language, aimed at frontend web development, but its architecture is quite interesting. As one can see in figure 4.2, the Elm-runtime translates the Elm code into DOM manipulations, and translates DOM events into *Msg* which is handled by the Elm code. This was the inspiration for the new module architecture. A module is managed by the runtime, which is the *core* application. But with some inspiration from Model-View-Controller (MVC), where instead of the module keeping its own state, this is again managed by the core, allowing for multiple modules to read and react to states updated by other modules, allowing for more interactivity between modules, and therefore being more modular.

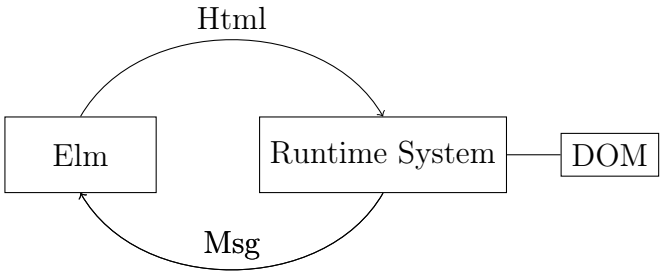


Figure 4.2: Elm Architecture (Figure adapted from [8])

4.3.2 Module architecture

In this architecture, the Elm-code is a module, while the runtime system is the core itself. The core invokes all modules, all of which, should have these three functions, `init`, `update`, and `view`.

Init Returns a collection of key-value-pairs, which represent the state of the core.

Update Returns a collection of key-value-pairs, which overwrite existing key-value-pairs in the state, or are appended to the state. Invoked every time a *Msg* is sent.

View Returns a collection which represents HTML, which is rendered by the core.

A module is initialized by invoking the **init** method, which returns a state. This can be seen in figure 4.3. After the state initialization, the modules' **view** method is invoked, which initializes the UI for the user, which can be seen in figure 4.4.

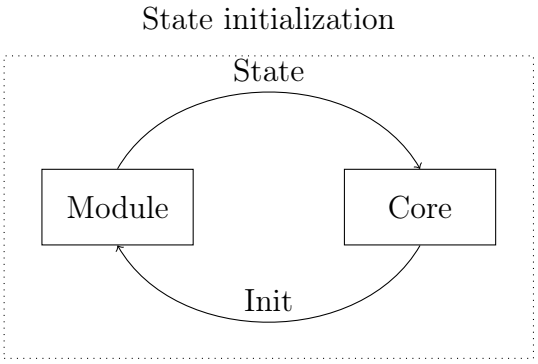


Figure 4.3: Module state initialization stage

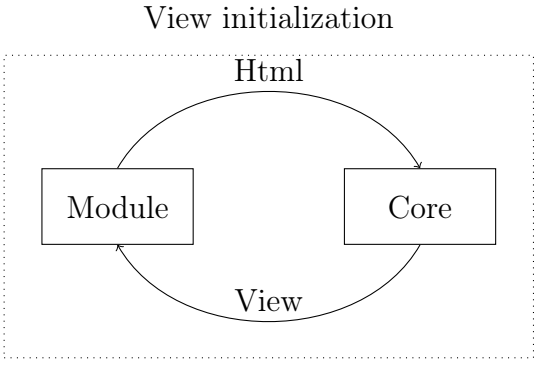


Figure 4.4: Module view initialization stage

Since the IDE is written in both TypeScript and Rust, a method of encoding type information when crossing between the TypeScript and Rust environment was needed. It was achieved by simply typing JSON objects, so while the state could be represented as any JSON object, it was instead represented as nested JSON objects, where, all values,

except **null**, where encoded as an object with one field, being the type of the object, and then the value. So an int would be **{ int: 0 }**.

The reason for representing a JSON object as key-value pairs, is that this could be easily translated to a Rust representation of the same type, using the *Serde* crate. This allows for creating Rust structs which represents JSON objects, and creates an automatic encoder/decoder between Rust and JSON. Using the ts_rs crate, we could also automatically create the TypeScript type that represents the automatically encoded/decoded JSON. This ensures a good cooperation between the *frontend* and *backend*.

4.3.3 Module lifecycle

The general idea was that for each possible DOM-event, there would exist a way to send a Msg. Each Msg contains a Msg name, and some value, which enabled pattern matching on Msg, similar to Elm, for modules, so each module could choose to act on a Msg or not. So, after the initialization of the IDE, any time the user interacted with the GUI the modules would react to the Msg. The trivial plugin, would simply return an empty state on **init** and **update**, while on **view**, it would return a *frag* element, which is a React element that evaluates to no DOM change.

In listing 4.3, an example of a counter module can be seen. This module initializes a state, containing the field **"counter"**, with the value **VInt 0**. The module is initialized, by invoking the function on line two.

```
1  init :: State
2  init = [("counter", VInt 0)]
3
4  update :: Msg -> State -> State
5  update (Msg { msg: "counter", val: VInt i }) model =
6    case lookup "counter" model of
7      Just (VInt j) ->
8        insert "counter" (VInt (j + i)) model
9      Nothing -> insert "counter" (VInt 0) model
10 update _ m = m
11
12 view :: State -> HTML
13 view model = Div [] [Text "Hello, World!"
```

```
14    , Btn [OnClick $ Msg { msg: "counter", val: VInt 1 }] []
15    , Text $ putStrLn $ lookup "counter" model
```

Listing 4.3: Counter Module (Haskell)

The *update* function the module exposes, on line five, pattern matches on a "counter" msg, with a **VInt i** value. If the given msg matches this, then the module adds to the "counter"-field, the value from the msg, which is 1. The module is invoked for all msgs being sent, so it is up to the module developer to ensure the match on the correct msg. We can see that we have a *catch-all* for all Msg that we do not care about, on line ten. Here we just return the input state, with no changes.

Finally, the *view* function, on line 12, renders a button, which when pushed by a user, sends the *counter-msg*, which we pattern match. This is done by adding an *eventListener*, to the HTML button, where we emit the passed msg on every click.

Module purity

One important thing in this architecture, is the pureness of module. The state of a module needs to be kept in the core application, and not in the module itself. The reason for this is twofold. It allows for the possibility of the core to be optimized in the future, as modules which do not react to a certain msg-state combination, can be noticed, and ensure modules are not unnecessarily invoked. It also lowers the complexity for module developers, as it is easier to reason about modules if *all* they do is read or write to some state.

If we have the modules A and B, where their relationship is $A \rightarrow B$, meaning A *invokes* B by sending some msg, which B reacts too, and we want something to happen before B reacts, we can add a new module, C, which also reacts to the same msg, but if we know the name of the module B, we can set the name of module C to be *above* the order, relative to B, ensuring that C always triggers before B.

4.3.4 Module v2 cons

While this approach was better than the first iteration, we still encountered issues, some of which could not be solved with this approach.

Not modular This setup is also not really modular, as a single module cannot invoke another module without being impure. The only way to invoke/trigger another module, is to throw a msg, which would trigger an update → view → cycle. So a module cannot *listen* for a single message, all modules are triggered by the same msg, and handled accordingly, synchronously. This is clearer if we visualize the architecture. In the figure 4.5, we can see a diagram of the architecture. Note the arrows, signaling how they interact with each other.

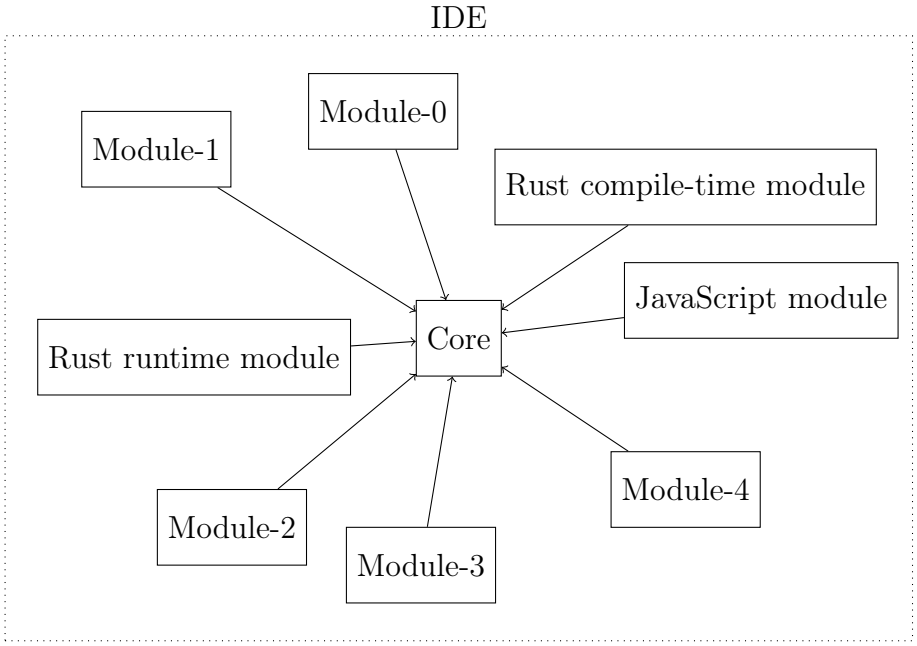


Figure 4.5: Module v2 architecture diagram, showing how all modules can only interact with the core.

They all go through the core, there is no way for $module_0$ to invoke just $module_1$.

Synchronous module invocation If a msg triggers a computational heavy method, the IDE will *hang*, and act *sluggish* until the computation has finished. This would also affect *all* modules, since they are invoked in order, regardless of if they actually change the state or view.

Ever-growing state There was no way to remove a field on the state, the state is appending/overwriting -only, which was a side effect of the coalescing of states, as we looked at the differences between the previous state, and the new state, and if the new state did not have a field that the previous one has, we kept it. If they had the same field, the new state overwrote the old one. So since we only did tree comparisons, there was no way to encode removal of a field.

State collision

A state collision occurs when two or more modules updates the same field, during the same update-cycle. This issue also occurs when folding two states. After any update-cycle, we were left with a list of states, which needed to be coalesced into a singular one. There are several different ways to correct a collision between two states:

1. If the states are of same type:
    1.1. If the value from one of the colliders are unchanged from the previous state:
        i. Keep the new value OR Keep the old value
    1.2. Else
        i. Apply the types' semigroup operator to the fields.
2. Else
    2.1. If the value from one of the colliders are unchanged from the previous state:
        i. Keep the new value OR Keep the old value
    2.2. Else
        i. Keep the left-hand side value OR Keep the right-hand side value

Since the states are ordered by the name of the module they come from, we have a consistent ordering of left-hand side and right-hand side. Due to the fact that module invocation is synchronous, and ordered. If the same modules give a collision on the same input[10], the resulting state will be the same every time.

The problem is that applying some semigroup operator on the values could be an unwanted way to resolve collisions. It is clear for some of the types what the operation is, like for the number types, it could be addition, but that would be a choice we are taking away from the module developer, and not a behavior they could change.

Therefore, the standard way will be to log the collision, and then drop both states. Even if two states have A and B amount of fields, and just one collision, we would drop $A + B$ amount of fields.

This problem of resolving state collision only occurs because each module returns a subtree of the state. We then have to analyze the new tree coalesced from all modules, to figure out if there occurs any collision. And then notifying the module developer of which field this collision occurred on, and which modules tried to modify that field.

[10]Given that all modules are pure

4.4 Module v.3

To solve the issues with the previous iteration, we decided to add another requirement, and the structure of a module:

- Everything is a module
- Modules can *invoke* modules

A module now only exposes two functions:

Init Returns nothing

Handler Returns nothing

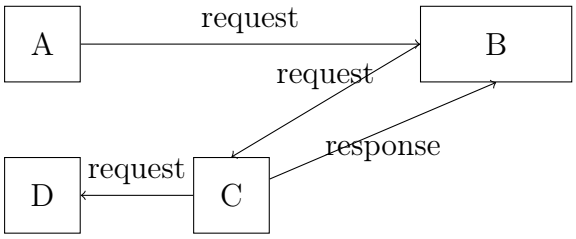
But given how neither of these functions return anything, how can modules affect the core? A module can *send* a set of instructions, which we call core modifications, to modify the core. These modifications are *sent* using an Multi Provider Single Consumer (MPSC) channel, which enables communication between two different threads, allowing for concurrent core modifications.

In the previous architecture version, each module directly changed the state, which caused issues. Instead, each modification a module does, *acts*, as a direct modification, but is in fact, translated to a DSL which can be analyzed for possible collisions. This was discovered to be a need, as in the new version, the UI was also restructured, to allow for less re-rendering, and this restructuring, made it clear that changing the state, or changing the UI is just tree manipulations, which will be discussed more later.

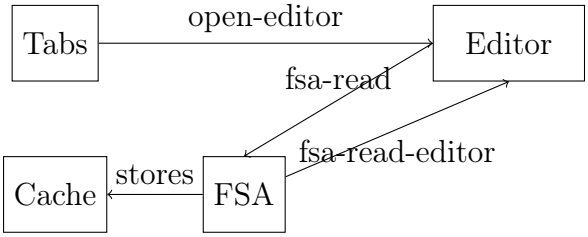
4.4.1 Zero-core architecture and microservice architecture

The new plan came with a change of viewpoint. Think of *everything being a module*, this pushed for a modularization between the then tightly coupled parts, the *frontend* and *backend*. As mentioned, having two different languages could allow for easier support of modules written in different programming languages, but for this to work in an optimal way, both the *frontend* and *backend* should be loosely coupled.

Furthermore, modules themselves should also be loosely coupled, where they invoke or are invoked by other modules. This is an equivalent architecture to microservices. In the figures 4.6a and 4.6b, we can see the equivalence side-by-side.



(a) Basic diagram of a microservice architecture



(b) Diagram showing communication between modules.

Figure 4.6: Two diagrams showing the similarities between a microservice and zero-core module architecture

4.4.2 Vanilla TypeScript

Instead of using React as the frontend framework, TypeScript was chosen, which simplified the integration between the backend and frontend, as the complexity of React's state management could be avoided, along with React's hydration. Given the rendering was now more *hands-on*, the core could expose a lot of the functionality for rendering, which modules could change. This would increase the difference between the JSMS and RSMS, as the backend was not privy to this API, but this was not seen as an issue, as this API would turn module non-pure. Reacts state management with *useEffect* and *useState* where needed to handle efficient re-rendering of the DOM, but since we were working directly on the DOM, this state management was just in the way. So moving to just TypeScript meant that it would enable easier to develop the JSMS.

4.4.3 Core modifications

Learning from the issues outlined in section 4.3.4, instead of a module returning the new core, it will rather return a set of instruction on *how* the core is to be modified, resulting in what the module developer wants the core to be. The reason for turning it around in this manner, is that, the new architectural change also came with a change on how the UI is modeled, as it is now up to the core to figure out an inexpensive way to do rendering. Since the core has UI-structure which is a representation of what the DOM should be, it

can be treated as a Virtual-DOM, similar as to how React does it. This also means that there could be a collision on UI-change, as well as on a state-change. Instead of solving the equivalent problems twice, it was decided to try to treat the issues with collisions in state and UI as the same issue; it is some form of tree-manipulation. We could therefore reduce the amount of needed methods on the module instance, to two. One for initializing the state, and one for handling events. In 4.4, we have a **CoreModification** type, which has two fields, one for the state, and one for the UI.

```
pub struct CoreModification {
    state: Instruction<Value>,
    ui: UIInstr,
}
```

Listing 4.4: Core Modification struct, it contains two fields, state and ui. Using a instruction type, parameterized on value, we represent state. The ui field contains a type alias for a triple, containing the instruction type, but parameterized on a *html*, *attribute* and string variants. (Rust)

4.4.4 Instruction based tree manipulation

This restructure changes the way the view is rendered. Instead of the view being re-rendered for each state-update, the view, or UI-hierarchy, is only modified by modules. This modification is similar to the earlier state modification, so a unified algorithm to solve this can be used. If there is an easy way to translate a UI modification to a state modification, and back again. To solve this, instead of having a module return the actual modifications, meaning, the updated core, a module returns a set of instructions of what to do with the core.

```rust
pub enum Instruction<T> {
    /// No Operation, results in no change to the state
    #[default]
    NoOp,
    /// Adds the given T where the id and/or class is found.
    Add(String, T),
    /// Removes the given T where the id is found.
    Rem(String, T),
    /// Combines two instruction into one
    Then(Box<Instruction<T>>, Box<Instruction<T>>),
}
```

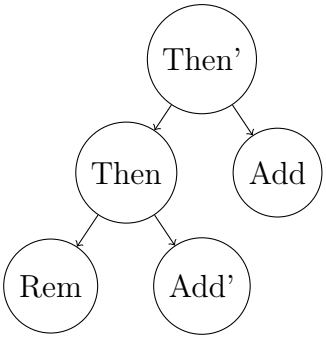
Listing 4.5: Instruction (Rust)

In the listing 4.5 we can see the Rust implementation of our instructions. Given how this is a recursive data-type, it is a safe assumption to make that this forms some kind of algebraic structure. As we have a kind of set, where the elements are variants of our instructions. This algebraic structure of the instruction-set comes more clear, if we remove the syntax, and just focus on the semantics.

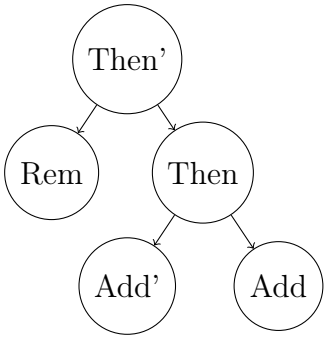
We have three instructions, $NoOp$, Add_T, and Rem_T. Those are our variants in our set, where the subscript T indicates which type our instruction are parameterized by. If $Then$ is our binary operation, combining our instructions, it is clear that our operator makes a tree structure out of our instructions. We want this to be associative, which means the equation 4.1 should hold.

$$Then'(Then(Rem_T, Add_T), Add'_T) = Then'(Rem_T, Then(Add_T, Add'_T)) \tag{4.1}$$

In the figure 4.7, we can see diagrams of two trees, in figures 4.7a and 4.7b, which are a representation of the different trees from the left and right side of the equation 4.1.



(a) Diagram of the resulting tree from $Then'(Then(Add_T, Add'_T), Rem_T)$



(b) Diagram of the resulting tree from $Then'(Add_T, Then(Add'_T, Rem_T))$

Figure 4.7: Diagram of the tress from right and left sides of the equation 4.1.

It is clear that these are two different trees, but when we evaluate the tree, we can see that the order of operations are the same. If we evaluate right side first, we get this order: Add_T, Add'_T, Rem_T, and if we evaluate the left side first, we get this order: Rem_T, Add'_T, Add_T. They are different orders, but regardless of which order we choose, both sides are equal. Furthermore, it is clear that $NoOp$ is our neutral element, giving us a monoid, as defined in 2.1.8. But we also have *idempotency* on our instructions, if we apply the same Add_T, or the same Rem_T, several times, it is the same as applying it just once.

It is very useful to think about what kinds of properties our structures have, as with this knowledge, we utilize existing theories to improve our software. Since this instruction structure is used for UI rendering, it is useful to minimize the amount of needless renders, which we can do by applying the idea that our instruction set is an algebraic structure with properties.

But these properties are not possible to encode in Rusts type system, but when implementing *combine*, we can map the variants along with the specific fields being added (Add), modified (Mod), or removed (Rem), to get a more optimized instruction set. If we are modifying a value on field $foobar$, but in the same instruction set, remove it, then the modifying instruction is an $NoOp$. This optimization can be found in appendix A. The result is that we can utilize algebraic structures to reduce the amount of data we send to the user.

Instruction utility functions

Like writing direct binary to develop a program, writing **instruction**s to change the core is quite abstract for most developers so to facilitate development of modules, a helper

class was created, which *translates* modifications to instructions. As shown in listing 4.6 and 4.7, a module developer simply invokes different methods on the builder, eventually building a **CoreModification**, to be sent.

```
async fn init(&self, core: Box<dyn Core>) {
    let mod = CoreModification::default()
        .set_state(UIBuilder::new().add(None, None, Html::Div()));
    core.get_sender().await.send(mod).await.unwrap();
}
```

Listing 4.6: UI Builder (Rust) showcasing how to add an empty HTML div element to the root HTML element.

```
async fn init(&self, core: Box<dyn Core>) {
    let mod = CoreModification::default()
        .set_state(StateBuilder::new().add("count", 0));
    core.get_sender().await.send(mod).await.unwrap();
}
```

Listing 4.7: State Builder (Rust) showcasing how to add a *count* field to the state, also showcasing how Rust can infer that the i32 type 0 is a **Value::Int** type.

This allows for an ergonomic way for module developer to create modifications on the core, without having to understand the syntax of the **Instruction**-set.

4.4.5 Backend agnostic frontend

Since we are using the framework Tauri to implement the IDE, the IDE is split to two, loosely coupled parts. The *frontend* and *backend*. The frontend acts as a thin wrapper around the core API, enabling different *runtimes* to handle module management, while the frontend waits for events, and renders the GUI. This structure allows for future maintainers of the IDE to be able to *trivially* switch runtime, if they wanted to use some other language to implement the runtime system in, like PureScript, Gleam or Haskell, all of which can target JavaScript, then they could. Indeed, this is ingrained in the base library, as everything that is Tauri dependent, is wrapped in a trait, so that external consumer of the base library can implement the necessary functionality given by Tauri. An example of this, would be to implement the IDE as a web application. Ignoring, for

a moment, the RSMS, all modules exist on the client side, so with minor tweaking, the application can be served as a web app. Some more modifications and workarounds are needed, to support the RSMS, as the modules would exist as a singular instance across multiple different users, but if the modules are *pure*, this would not matter. Then the only issue is to ensure consistency between the state and UI, which would be stored on the client side.

4.4.6 Making the core evaluate modifications asynchronously

Due to Rust first class focus on concurrency, it was trivial to make the core modifications run asynchronously. In previous iterations, the core evaluated one event at a time, waiting until all modules had finished their computations, before emulating the change and allowing for the next event to be evaluated. But this caused a noticeable *lag* if an event was long.

This was solved by changing the core modification evaluation from a simple method to be invoked, to an MPSC channel system. Using *tokio*[11], a Rust crate for asynchronous development, a channel for core modifications was created, and instead of the core collecting all modifications, each module is invoked and *awaited* for in a separate thread, where in each module, if they have a core modification, sends the modification to the core channel, which works on a first come, first server basis. Here the core can evaluate the changes, also on a separate thread.

This restructure was trivial to add, as due to Rusts borrow-checker, we know that there are no race conditions and other similar concurrency issues in our application.

4.5 Testing

A zero-core IDE is equivalent to a microservice architecture, so we can use the similar lessons learned in microservice architecture, when developing ours. In a microservice architecture testing is important to ensure changes in one service does not inadvertently affect another. This is commonly achieved by using *pipelines*, a part of the Continuous Integration and Continuous Delivery (CI/CD) process, where we run several *jobs* whenever we make a change to our microservices.

[11] `https://tokio.rs/`

In our case, with modules, if we are bundling different modules together, and serving that as an IDE, we want to ensure that a change to a module does not negatively affect the other. This is where *pipeline jobs* come in, as each *job* test some part of our IDE. While it is cheaper to *spin* up an instance of the IDE in a pipeline, than an application dedicated to serve millions of users, we still want to avoid doing this unnecessarily. This is why we split up our testing into different stages.

4.5.1 Mocking

Due to the *pureness* of modules, mocking can be achieved easily, and therefore, modules can be tested alone, which is good, because testing a singular module is inexpensive. There are several ways to do this mocking in our architectural setup. For both the JSMS and RSMS, there are *mock-cores*, which can mock the expected functionality of the core instance, which we can extend to evaluate actual modifications, ensuring we can assert that some state or UI change has occurred after an event has been sent.

4.5.2 Unit testing

A module developer should create unit tests for their module. This can easily be done, and tested many times, due to the light-weightiness of a module. This, together with mocking, ensures we can test our modules, as if they were in a IDE. Which means we can ensure changes made to a module is non-breaking.

This also applies to maintainers of the IDE, as maintaining the core functionality and API of our library, means documenting possible breaking changes, which unit tests can help ensure.

Furthermore, we also need testing of our core libraries. Since we are aiming to support many different languages; to be language agnostic; we need different core libraries providing utility functions that are widely used when developing modules for our IDE. An example of this, could be the builder pattern we use when creating the core modifications, as shown in listing 4.4.

To ensure these unit tests are uniform, and cover the same edge-cases, we have designed test data, that the different libraries use when testing. For instance, when developing with foreign languages, the data sent between the IDE and modules, or in between

modules, is in the JSON format. But in a Rust module, this would be deserialized into a Rust equivalent struct, before being serialized back to JSON when sent out from the module.

Similarly, in JavaScript, we have to ensure that we can actually handle the data the IDE sends[12]. Using the same test input, we can verify that both the Rust and JavaScript library correctly handle the serialization and serialization of the data. Since much of the base logic exists in the backend, this is where the brunt of the unit tests exist.

4.5.3 Library testing

We also need to ensure that the libraries we serve to our module developers are correct. The easiest way for a maintainer to achieve this, is to test in production; letting module developers report issues. This is not a good experience for module developers. But the second-easiest way is to create unit tests that ensure edge cases are handled correctly. We have designed test data for serializing and deserializing data between the different languages supported by the IDE, ensuring that any library developed can be verified to work correctly. The test data is modular, meaning we can easily create new edge cases. In the picture 4.9, we can see that there are 135 different test cases for the JavaScript library. In picture 4.8b, we can see that there are 5 serializations tests, which is because we cannot create a parameterized test from a file, like we can with the JavaScript testing framework, Vitest[13]. But they both use the same test data as input.

```
✓ optimization_test_cases > nested_then_flattening 0ms
✓ edge_cases > remove_with_special_chars 0ms
✓ edge_cases > then_with_single_operation 0ms
✓ edge_cases > deeply_nested_then 0ms

Test Files  1 passed (1)
     Tests  135 passed (135)
  Start at  20:07:24
  Duration  271ms (transform 67ms, setup 0ms, collect 86m
```

(a) Picture showing the output of running the JavaScript library with the test data as input.

```
running 5 tests
test serialization_tests::attr_test ... ok
test serialization_tests::html_test ... ok
test serialization_tests::event_test ... ok
test serialization_tests::value_test ... ok
test serialization_tests::instr_test ... ok

test result: ok. 5 passed; 0 failed; 0 ignored;
```

(b) Picture showing the output of running the Rust library with the test data as input.

Documentation testing

In Rust any function annotated with a doc string, can contain code examples. If these code examples are written as Rust code, and use assert statements, then this code is run,

[12]Since the types are specified on the backend

[13]`https://vitest.dev/`

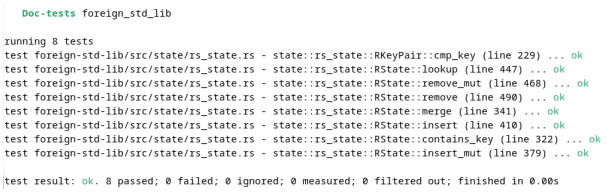


Figure 4.9: Picture showing the output of running the JavaScript library with the test data as input.

during testing, as if it was an actual test. Meaning the saying *code is documentation*, can be *documentation is code* in Rust. In picture **??**, we can see some *doc-tests* being run on the library for runtime Rust modules.

UI testing

We can also combine this with existing testing libraries, like Playwright[14], which can enable us to create tests specifically for UI behavior. In the case of Playwright, our UI testing is dependent on that our *mock-core* has the necessary functionality to transform the **Html** type we have implemented, to actual HTML, which can be rendered on a webpage, or *headless*, in Playwrights case, so that Playwright can assert the state of our DOM.

4.5.4 Module family testing

If a module changes some feature, let's say in the editor functionality, the module family tree encompassing this functionality needs to be tested, to ensure nothing breaks. This means creating tests that use all the modules in a family, and asserting that the state and UI behave as expected. In the case of the editor functionality, that after the event *open-file* is sent with a path to some file, that there exists a *textarea*-HTML-element in the DOM, with the same contents as the file.

Contract testing

As a module developer, on is designing some kind of API, but the developer has no say in how a consumer of the API consumes it. In a microservice architecture, the common way

[14] `https://playwright.dev/`

to work around this, is to version control the API by prefixing $v*$ in front of all endpoints in the API, where star, (*), is the version of the API. This way, the API designer can develop new APIs, without worrying about breaking functionality that consumers of the API depend on. This, however, usually means having to maintain equivalent APIs in parallel, until one decides to deprecate an older less used version, forcing consumers to move on to the newer version of the API.

Instead of relying on such a versioning system, module developers could use *contract testing*.

Contract testing Imagine some API, and several consumers. The API developer is serving some data, in this case an integer number, which all the consumers use. One day, the developer finds out that using integers is not optimal, and want to move on to using floating point numbers instead. Changing the API outright could bring issues, as the consumers might rely on the API being an integer, instead of a float. But the change is needed, or wanted, at least. In this scenario, it is *easy* to inform all the consumers of the API, but if the consumer count increases tenfold, this is more difficult. A notice can still be sent, but it is not feasible to ensure all consumers commit time to change their ways. Contract testing ensures that, if a change like this occurs, the maintainer of the API is notified by which consumer this change breaks.

The issue is to create these contracts. Using frameworks like Pact[15], a developer creates a DSL test, where they describe how the provider or consumer reacts to certain interactions. But since everything is a module, we can automate this.

4.5.5 Automating contract testing

This process could be partially automated, as all modules have to register the event they want to handle. Furthermore, all events thrown are also explicitly done through the core instance, meaning a *test-core* could be created, which registers which event is thrown from what module, and all dependencies between modules can be noted. This has been partially achieved. By loading all specified modules, we can note their dependencies, by looking at the different consumer and providers.

A module has two different states, initialization, and handling. During initialization, a module cant have any dependency on other Modules, but it can register for an event,

[15] `https://docs.pact.io/`

meaning the module will be invoked once the specified event is triggered, or it can throw an event. Registration for an event can always be directly analyzed, as it happens on the core instance. The way a registration occurs, is that a module supplies a string for a specific event name to trigger on, meaning after the first initialization, the consumer graph looks like this: $module \rightarrow event$.

To find the providers, we can initialize a module, and see what possible events, if any are thrown. If an event is thrown, then that module is a provider of that event. The mapping between event and module is not unique, as several modules can provide the same event. Another way a module can provide an event, is through user interaction. A module can create some UI, which a user can interact with, which would trigger an event. The UI can be analyzed for such triggers, and a mapping between that event and module would be created.

During a handling of a triggered event, a module could register for new events, trigger another event, or do nothing. To find more dependencies, a module is triggered with the events it subscribed to, and analyzed for new registrations or triggers. If the list all subscribed events have been exhausted, and no new registration or trigger has occurred, that module is considered *exhausted*, and won't be analyzed anymore.

Since modules are asynchronous, and can possibly spawn their own threads, there is a possibility for deadlocks to occur. There is no way to avoid this, without restricting module developers, which is not wanted. Therefore, it is up to the user of the module dependency graphing tool to avoid this, by supplying a timeout value, ensuring that if, after some time N, the analyzation of a module is still occurring, the analyzation is killed.

Modules could be depending on a certain state, before triggering or subscribing to an event, this is not really possible to know without doing static code analysis, so this is out of scope for this tool.

With this, we can detect possible contracts between different modules, ensuring we know that testing between them is needed.

4.5.6 End-To-End testing

The final step in the testing pipeline, is to test the entire application together. This is known as End-To-End (E2E). E2E is expensive, compared to the other steps, as we have to load the entire application in the pipeline, and test all interactions. This, of course, is

the easiet way to cover all edge-cases, but since it is the whole application being tested, harder to figure out what caused a failure. Our IDE can be saturated with events in the E2E step of a pipeline, as all user interactions are translated into events, this ensures a module developer can narrow down what modules are at fault, by what modules *subscribe* to that event.

4.6 Modules

Event type The event type allows for modules to pattern match on specific events, and unlike, as in the previous version, modules can *subscribe* to specific events to react to. This changes the structure of the module architecture to go from one wherein the core is a terminal object, to a more *complicated* one, in which module families can form.

This forms our *request* and *response* type, making the equivalence with a REST API obvious. The clients and consumers of the API, are the modules which can be seen in listing 4.8.

```rust
#[async_trait]
pub trait Module: Send + Sync {
    fn name(&self) -> &str;
    async fn init(&self, core: Box<dyn Core>);
    async fn handler(&self, event: Event, core: Box<dyn Core>);
}
```

Listing 4.8: Module trait (Rust)

A module can interact with the core, by getting the state, UI, *throwing* an event, registration themselves to *handle* an event, or to *send* a **CoreModification**. In listing 4.9, we can see this core trait.

```rust
#[async_trait]
pub trait Core: Send + Sync {
    async fn state(&self) -> State;
    async fn ui(&self) -> Html;
    async fn throw_event(&self, event: Event);
    async fn add_handler(&self, event: String, handler: String);
    async fn send_modification(&self, modification: CoreModification);
    async fn appdir(&self) -> PathBuf;
}
```

Listing 4.9: Core trait (Rust)

Since a module updates the core by *choosing* to send a **CoreModification**, through a MPSC-channel, a module can run an expensive computation on another thread, while *ending* their invocation, ensuring a smooth IDE experience.

4.6.1 Magnolia dependency graph visualizer

In Magnolia, as in many other languages, one cannot have a cyclic dependency. This means that the dependency graph of a Magnolia project should be a DAG. And since Magnolia has such a focus on reuse, the dependency graphs in a Magnolia project could be quite large. Which means the cycles could be quite long, which would make resolving the cyclic dependency issue complicated. One way to help a developer, would be to give them a tool to visualize the dependency graph, so that they could see what modules are connected. Using the Magnolia library as the input, we can create a visualization of the dependencies in Magnolia. Using two modules, one for *parsing* the Magnolia library, finding all packages, and their dependencies, and another for visualizing this.

```rust
    async fn init(&self, core: Box<dyn Core>) {
        core.add_handler("get_magnolia_graph".to_string(), MODULE_NAME.
            to_string())
            .await;
    }
```

Listing 4.10: Magnolia library parser module *subscribing* to an *get_magnolia_graph* event (Rust)

In listing 4.10, the module is invoking the *add_handler* method on an object that implements the *Core* trait, (4.9), and passing *get_magnolia_graph* and *MODULE_NAME*.

This means that events with the event name *get_magnolia_graph*, will trigger this module. It's *this* module, because we have to pass the name of the module handling the event.

We can therefore invoke this module by simply triggering the subscribed event.

Due to Rust's type safety, there is a lot of *noise*, especially because we are working with recursive data structures, with optional values. In listing 4.11, we can see a simplified version of the module handler, but this is not valid Rust code.

```
1  async fn handler(&self, event: Event, core: Box<dyn Core>) {
2      match event.event_name() {
3          "get_magnolia_graph" => {
4              let path = event.args();
5              let key = format!("graph:{path}");
6              match core.state().get(&key) {
7                  Some(existing_graph) => {
8                      core.throw_event(Event::new("graph", existing_graph));
9                  }
10                 None => {
11                     let graph = get_graph(&path);
12                     core.throw_event(Event::new("graph", graph));
13                     let mods = CoreModification::state(
14                         StateBuilder::add(key, graph)
15                     );
16                     core.send_modification(mods);
17                 }
18             }
19         }
20         _ => (),
21     }
22 }
```

Listing 4.11: Simplified magnolia library parser module (Rust)

The above code (4.11), we are handling events with the name *get_magnolia_graph*, (line 3), and getting the path that is supplied in the event argument, (line 4). In line 4 we then create a *key*, which we use to check if this graph has been created yet, by checking the state (line 6). If this graph does exist, we *respond* by throwing an event with the existing graph (line 7 to 9). If it does not exist, we create it by calling the *get_graph* function, (left out for brevity), which recursively finds files in the supplied path, using RegEx to find the packages, and their dependencies. We then end by *responding* with

the created graph, (line 12), and store it in the state, with the key (line 13 to 16). The resulting response can be seen in listing 4.12

```
{
  event: {
    event: "graph",
    args: {
      list: [
        {
          obj {
            name: { str: string },
            dependencies: { list: [{ str: string }] }
          }
        }
      ]
    }
  }
}
```

Listing 4.12: Magnolia library parser response (TypeScript)

The module responsible for rendering the graph, uses D3, a visualization library for JavaScript. D3 expects *nodes* and *links*, specified in listing 4.13.

```
type Node = { id: string, name: string };
type Link = { source: string, target: string };
```

Listing 4.13: D3 expected input (TypeScript)

But due to how types are encoded in our Value type, as seen in 4.12, some translation is necessary. We have to go from the type *Value*, to *list of objects, with two fields, name and dependencies, of type string and list of string, respectively* This translation can be seen in 4.14. Left out, are the steps verifying that the event has an argument, (since its optional to pass one), and that the argument is of the list variant of *Value*. Since the list variant can contain any *Value* variant, we filter the list by whether it is an object variant, (line two), we then transform each element in the list, into an intersection of the types D3 expects, (4.13). The question mark syntax on line four, where we declare the *id* variable, means that if any of the expressions on the left are undefined, the resulting expression is undefined. Since the object variant of *Value* does not necessarily contain the *name* field, or is of the kind *str*. On line nine to thirteen, we are getting all the dependencies from the object, with a helper method, *tObjLookUpOr*. This method does a *lookup* on

the supplied field on an object, and a type check. If the object does not exist, or is not of the correct type, the passed fallback value is returned instead, in this case, an empty list (line ten). Since we know the value is a list, we can safely access it, (line eleven), and filter by the string variant, and transforming the *Value* to a string primitive, (line eleven to thirteen).

```
1     const data = args.list
2       .filter(v => isTObj(v))
3       .map(obj => {
4         const id = obj.obj["name"]?.["str"];
5         return {
6           id,
7           name: id,
8           source: id,
9           targets: tObjLookupOr<ValueList>("dependencies")
10            (tList([]))
11            (obj).list
12            .filter(v => isTStr(v))
13            .map(s => s.str),
14        };
15      });
```

Listing 4.14: Snippet from the dependency visualizer module, showing how we translate from our built-in representation of values, to the one expected by the D3 library (TypeScript).

4.6.2 File System Abstraction (FSA)

Many of the features needed in an IDE, and in the modules showcased thus far, are dependent on file system operations. The module which expose such functionality to other modules, is called *ide_fsa*. It is written in Rust, as Rust abstracts away the differences between OS paths. As the same path on a Windows system is represented differently on an unix system, since the path separator is
and / respectively. The following are a list of file OS system operations that are made available to modules, with the help of the ide_fsa module.

- Open a file
- Create a file/folder
- Read from a file

- Move a file/folder
- Copy a file/folder
- Delete a file/folder
- List the contents of a folder

As shown in the picture 3.10, from chapter 3, many of the modules developed are dependent on ide_fsa. This was a deliberate choice, as not only would it mean we could rely on Rusts OS abstractions, ensuring the IDE would work on different OSes, but it also meant we could delegate error handling from files to this one module. As doing such file system operations[16], are inherently error-prone. This is encoded by Rust, into a *Result* type, which we utilize to send Events, but to different modules depending on the result variant.

Ide_fsa listens for the following events:

- fsa-read
- fsa-write
- fsa-dir

Which, depending on what argument the consuming module passed, maps to a specific Input/Output (IO) operation, which may fail. We have decided to not handle the failure on the consumer side, rather the provider side. So, if a user attempts to open a file which does not exist, it will simply fail, and a notification will appear. We decided for this approach, as it allows for simpler UI focused modules. In a fully fledged implementation of the prototype modules showcased, proper, expected, UI behavior should be implemented, like showing explicitly to the user that the action they just did, clicking open file, caused an error.

[16] Also called Input/Output (IO)

```
1   async fn handler(&self, event: Event, core: Box<dyn Core>) {
2       let result = match event.event_name() {
3           "fsa-write" => fsa_write(&event, &core).await,
4           "fsa-read" => fsa_read(&event, &core).await,
5           "fsa-dir" => fsa_dir(&event, &core).await,
6           _ => Ok(()),
7       };
8
9       if result.is_ok() {
10          return;
11      }
12
13      let obj = Value::new_obj()
14          .obj_add("error_event", Value::Str(event.event_name().to_string
                ()))
15          .obj_add("error_args", event.args().cloned().unwrap_or_default
                ())
16          .obj_add(
17              "error_msg",
18              Value::Str(format!("{:?}", result.unwrap_err())),
19          );
20
21      core.throw_event(Event::new("fsa-error", Some(obj))).await;
22  }
```

Listing 4.15: Snippet from the ide_fsa module, showing how it handles events. If any of the functions from lines 3 to 5 return an error then we create an event, sending the error to the ide_error module.

Looking at the fsa-read event, we can see in listing 4.15, that the events are matched by their event name, and mapped to the corresponding function. In the case of fsa-read, it expects there to be a path argument, which we use to read the contents from the file, if it exists, before *responding* to the consumer by sending an event with *fsa-read-module*, where *module* corresponds to the module name passed in the original event argument. This happens if there occurs no errors[17] during the IO operation. If it does fail, say because the file does not exist, then an event is sent to the ide_error module, displaying the error to the user.

[17]It is an error when the method **is_ok**, on line 9, returns false

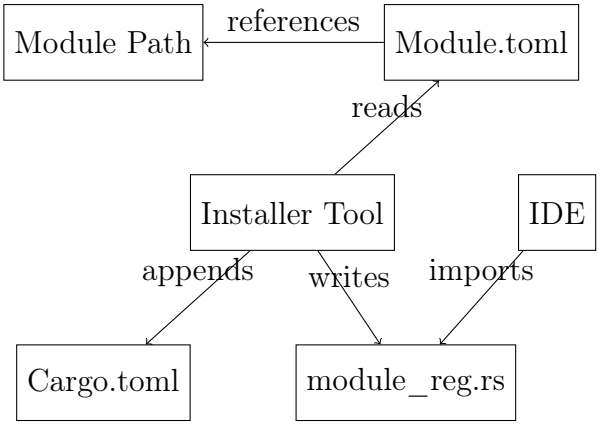


Figure 4.10: Diagram of compile time module integration, we can see that when we invoke the installer tool, it reads the Modules.toml config, gets the necessary module paths, and *installs* them by writing them onto a file, *module_reg.rs*, which is imported by the IDE. All imports need to be specified in *Cargo.toml*, which is also handled by the installer tool.

4.7 Module installation

Manually installing compile-time modules can be tedious, so we automated the task. We are using the term *installation* to mean adding the module to the IDE. This can be as simple as adding a script tag to the DOM, with the source being a JavaScript file, or more complex, loading a library which has the correct bindings to be a Rust runtime module.

For JSMS, it is quite trivial, simply bundle the JavaScript code into a single script, and import it into one file, and, as a final step, bundle this file. This ensures that all JavaScript modules are included, and loaded after the core system.

For the RSMS, it is less trivial. A compile-time Rust module is a Rust crate, and since we are using Cargo[18] we need to specify all our imported crates in a *Cargo.toml* file. We then also need to explicitly import and add the module into a list of all compile-time modules. With a build script, we have automated this. In the figure 4.10, we can see a diagram of this process.

For runtime modules, the installation is divided into two steps. First, we have to add the source file to the application directory[19], before actually *installing* the module. Depending on what kind of file it is, this installation process can vary.

[18]Build tool for Rust
[19]Located in *%APPDATA* on Windows, or */.share* on Unix

4.7.1 Rust runtime module installation

We use the term runtime here, to strictly mean post compilation. When this IDE is running in user-space, we cannot use newly installed Rust runtime modules. This is because the IDE only looks for Rust runtime modules during startup, so a restart is necessary. Given that a Rust runtime module is a simple library, we know what functions it exposes, and the calling conventions. Along with the abi_stable crate, we can ensure no undefined behavior happens during runtime.

4.7.2 JavaScript runtime module installation

Unlike the Rust runtime modules, JavaScript runtime modules can be installed and used post-startup. This is because there is no difference on how a JavaScript module is installed, regardless if the installation happens during compile time or runtime. Since all a JavaScript module is, is a script element on the DOM, we can add new ones whenever. The only thing that differentiates the two, is that for runtime modules, we have to call the **init** method after installing it, instead of during compile-time, when all modules are invoked after every module has been installed.

Chapter 5

Related Work

In this chapter, we will first address the module architectures in other IDEs. We will then, in section 5.2 discuss how much of the functionality that an IDE has, comes from the Language Server Protocols. In section 5.3, we will look at the complexity which arises from GUI development, and some other works which help mitigate some of this complexity. We will then look into ways to automate testing of GUIs, in section 5.4. In sections 5.5, and 5.6, we will cover STF and abstracts algebra, respectively. Finally, in section 5.7 we will discuss language workbenches.

5.1 Existing module architectures in IDEs

In this section, we will cover how other IDEs have designed their module architecture. We will also highlight instances which are similar to our architecture.

5.1.1 Eclipse

IDEs are one of the most common application that supports extensions by third-party code. IDEs like Eclipse are specialized for working with Java, but they can still support other languages with the help of modules. A module in Eclipse for instance, could extend Eclipse with functionality like syntax highlighting, code completion, Go-to-definitions, debugging, and more, for standard programming languages. A lot of this functionality, comes from module-to-module extension, as in Eclipse modules can extend modules, with the use of the Eclipse Rich Client Platform (Eclipse RPC) [15].

Eclipse Rich Client Platform

Eclipse RPC is a platform for building desktop applications. Eclipse being an example of this platform in action. A plug-in could for example be responsible for setting up the general UI layout, similar to our module, *ide_framework*, and another plug-in could then modify this UI by adding a file explorer, similar to our module *ide_explorer*.

5.1.2 NetBeans

An important part of NetBeans core architecture, is the NetBeans Module API. This API is responsible for supporting the *runtime environment*, which is the minimum amount of modules needed to run the NetBeans application. A module in NetBeans is a JAR file, in which a module can, amongst other things, list their public packages. This means that other modules can directly invoke methods provided by this package.

5.1.3 IntelliJ

The extensibility IntelliJ has, is achieved by its extensions points architecture. This is an API for plugins to integrate with the IDE. Plugins use this API to register their implementations, which the IDE then use. Between different versions of IntelliJ, plugins may be broken, due to a breaking change in their plugin API.

5.1.4 Visual Studio

There are two kinds of extensions in Visual Studio, *VSPackage* and *MEF* extensions. *VSPackage* are mainly used to extend functionality like tool windows and projects, while *MEF* extensions are used to customize the text-editor.

5.1.5 Visual Studio Code (VS Code)

Visual Studio Code is an extensible IDE. It achieves this extensibility with its extension API. This API allows for extensions to modify the look and behavior of the IDE. In fact, many of the core IDE features are possible due to built-in extensions.

5.1.6 Vim

In Vim, plugins are written using VimScript. All plugins located in Vims plugin folder are loaded when Vim starts. This is where a user of Vim would place their configuration plugins, adding custom commands, key binds, themes, and more. In the ftplugin folder, plugins that are specific to file extensions are stored. Assuming we have a *python.vim* file in ftplugin, that file would be loaded when we open a python file.

Plugin management

Most users use Vim with a plugin manager, a plugin, which manages other plugins. These plugins can do mundane tasks like:

- Download and install plugins from repositories
- Configure plugins
- Handle updates
- Handle enabling and disabling of plugins

5.1.7 Emacs

Emacs[1] is an extensible text editor. Almost all of Emacs functionality is achieved by writing code in Emacs Lisp, a dialect of Lisp[2]. Unlike other IDEs, Emacs does not restrict modules to a certain API, instead they are free to modify anything, as the functionality from the Emacs Lisp, *sits* atop of a core written in C, which abstracts away platform specific code, and enables Emacs to be turned from an a *simple* text editor, to write and send emails, multimedia management and much more. Another interesting thing about Emacs, is that all files are in *buffers*, meaning the representation of a file shown to the user, is not necessarily the contents of the file, some rendering is can be done by a plugin, before the file contents is shown to the user.

[1] `https://www.gnu.org/software/emacs/`
[2] `https://lisp-lang.org/`

5.1.8 Theia

"The Theia IDE is a modern IDE for cloud and desktop built on the Theia Platform"[3].

Eclipse Theia is a highly extensible IDE, supporting *extensions* from VS Code, their own extensions and plugins, and *headless* plugins. Theia differentiates between extensions and plugins, where a plugin is installed during runtime, and an extension is installed during compile-time.

Theia reuses components from VS Code, like their extensions API, which enables them to support VS Code extensions. Theia plugins share similarities with VS Code extensions, but are not restricted to *just* the VS Code API.

Theia's extensions are designed to add or remove existing core functionality in Theia. They also have access to the entire core API.

An headless plugin runs without access to the frontend, meaning they are suited for Commandline Interface (CLI) interactions, or similar use cases where a frontend is not needed.

Theia is both a desktop IDE and a web IDE. Where there is no real distinction between the two, both abstracting it to a frontend and backend. This is similar to our zero-core IDE, as we can make this abstraction due to our usage of a web view for the desktop IDE.

5.2 Language Server

The most important features in a modern IDE are possible due to the LSP[4]. LSP is a protocol for a language server and editor, (the client), in which they communicate, allowing for many of the features mentioned in section 2.2, and explicitly mentioned in table 5.1. LSP being the standard since the 2020s, is a sign of modularity being preferred, as now a single LSP can be created, and used across several different applications, like IntelliJ, VS Code and Vim. While useful for *standard* language, this is the limiting factor when it comes to supporting experimental languages, as not only does a new set of

[3] `https://theia-ide.org/`

[4] `https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/`

Table 5.1: IDE features enabled by LSP

IDE Feature	LSP-method
Go to Declaration	textDocument/definition
Go to Implementation	textDocument/implementation
Auto-completion	textDocument/completion
Hover	textDocument/hover
Warnings	textDocument/publishDiagnostics
Rename	textDocument/rename

protocols need to be appended to a language server, the editor itself needs to be changed to actually use these protocols. This creates a lot of work, for both the IDE developer and for the compiler developer. Here is where a modular approach can help both. If some new functionality or feature is added to the experimental language, this off course means the compiler/interpreter has to be expanded and/or modified, but for the IDE, a module could be added and/or modified to utilize this change, instead of having to change the entire application.

An example of this in action, say a developer is working on a file *main.ts*, in their Typescript project. They hover over a type imported from, and defined in *types.ts*. This is what happens:

1. The editor detects the user is hovering over a *special* word
2. The editor sends a request to the Typescript LS
3. The LS responds
4. The editor formats the response into a small window showcasing the documentation and implementation of the type

5.3 Graphical User Interface development

A common complexity within application development, is GUI development. Using the MVC-pattern as an example, GUIs can represent structures such as lists, which users might want to manipulate in some fashion, like appending or rearranging the items in the list. Managing such a change, especially one that involves GUI widgets can be a challenge, since a change in the view should be reflected in the model, and encoding this can be very involved.

Another issue in GUIs is optimizing performance in regard to events triggered by user actions, such as scrolling, resizing or typing. These events could happen many times in a second, while in theory user speed is trivial for a computer to keep up with, there are instances where a module family could be quite large, meaning many different modules are triggered by the same event many times. There are techniques, called event coalescing, for handling this, like debouncing and throttling.

Debouncing Debouncing is a technique where you delay the sending of an event until after some time period T has passed. Once the event is triggered T_0 starts counting down. If the same event is re-triggered while $T_0 > 0$, T_0 is reset by $T_0 = T$. If $T_0 = 0$, then the event is sent. Ensuring that T is not too large, is important, as if T is above some threshold, the user of the GUI will notice, and it will make the application *feel* slow.

Throttling Throttling is a similar technique to debouncing, except instead of delaying the event by some time T, the event is only sent when $T_0 = 0$. Meaning the event is sent at regular intervals, and could be sent at the exact same point in time when the user triggered the event, or it could happen at most, T units after the user action.

5.3.1 Flushable promises

Debouncing and throttling work in less complex GUI structures, but as the amount of features in an application increases, the complexity will also increase. These event-coalescing-strategies are a source of subtle bugs, as event coalescing can easily break modularity. In a JSMS, this issue could be solved by using *flushable promises* [4]. This could have solved our issue, where we had some event handler that took noticeably longer time to return, but since this was a Rust module, we could *solve* this by doing this computation on another thread. If it was a JavaScript module we could have solved it by using *flushable promises*.

If we implement a LS-client in JavaScript, *flushable promises* could allow for a smoother experience, as things like *looking up* renaming in a Magnolia project is a more involved process for the compiler, and in larger projects, could take a noticeably long time.

5.3.2 Multi-way Dataflow Constraint System

Luckily, there exists frameworks that make this task easier. *WarmDrink*, [18], [14] is a JavaScript framework that allow a developer to declarative specify structural changes in an application. The framework can guarantee GUI behavior, by utilizing a Multi-way Dataflow Constraint System (MCDS), ensuring it can create constraint systems. A constraint system is a representation of how different variables are dependent on eachother. If one variable is needed in the computation of another, they have a dependency relation. By using a DSL, a developer can declare a constraint system. But, similar to how working on our **Instruction**-set without tooling is quite complex, tools have been created to ease the creation of such constraint systems [13]. Given how WarmDrink is a JavaScript framework, it is quite well suited for the web, aswell as our IDE. A runtime system specifically for a MCDS could be implemented for JavaScript modules. Furthermore, there also exists a Rust implementation of this framework, which means a similar system can be created for Rust modules.

5.4 Automated testing

Due to the extensive modularity of the application, all modules can be tested individually, by *mocking* the expected state and events. This means that breaking changes in one module can be detected before E2E testing, which is expensive. But this can only verify the general logic of a module and module family, not the UI. To achieve such automation, one could rely on an automated testing framework, like the one in [3]. Or if one is working with a *simple* JavaScript runtime, one could use third party software like *Playwright* for creating tests, as it can auto generate the DSL, while the developer uses the module or entire IDE if it is an E2E test. This would help a module developer to discover behavior that a user might not expect [16].

5.5 Abstract algebra

Magnolia is a kind of algebraic specification language, like CafeOBJ [9]. An algebraic specification language, is a language where one can develop similarly as to how one might create an algebraic structure. We specify some set and some functions that take elements from the set as inputs. Finally, we specify the behavior of our functions, if they are associative, if there are any predicates that the arguments to the functions need to fulfill, etc. As shown in the development of this IDE, this can be quite useful way of thinking.

5.6 Syntactic Theory Functor

STF is a framework for creating, reusing and restructuring specifications[12], specifications from algebraic specification languages like CafeOBJ [9]. STFs are also used by the new Magnolia compiler[5], to resolve renaming in Magnolia and *flattening* of the ASR to be shown to the developer [19].

5.7 Language workbenches

Language workbenches are tools for creation and use of computer languages [10]. An IDE is a kind of language workbench, as it can be used both for a language, as described in this thesis, but also for creating languages, as this too is a software project. More specific are the tools created by JetBrains, Meta-Programming System (MPS)[6], for creating programming languages, specifically DSLs. What makes tools like MPS different from standard IDEs, is that in standard IDEs we work on the source file of a program, while language workbenches work on the Abstract Syntax Tree (AST) of the program. Programming languages were made so that we programmers could read closer to how we think, than how a computer thinks. So before a computer can run our code, it needs to be translated. Generally, this translation is done by parsing the source file into some tree-structure, and then interpreting that tree, transforming it into an AST. As a concrete example, in figure 4.7 we have a visualization of an AST, for our **Instruction**-set.

So language workbenches then, enables a developer to work on this abstract representation of the language one is creating.

Similarly, in Magnolia, it is useful for a developer to visualize the effect of the compiler flattening the ASR-tree, especially in regard to renaming. An ASR-tree is similar to an AST, except it has extra information on each node.

[5] As of May 2025, still in development
[6] `https://www.jetbrains.com/mps/`

Chapter 6

Discussion and conclusion

In this chapter, we will address some of the issues encountered when developing this modular IDE. In section 6.1, we will discuss our experience developing modules for this IDE, and how that was affected by the development of the architecture itself. We will continue in a similar vein, by covering some of the issues with the API, in section 6.2, and how they were *solved*. Section 6.3 will discuss language agnosticism, while section 6.4 will highlight the challenges for development of foreign modules. And finally, we will make a conclusion from our hypothesis 1.

6.1 Modular development

In this thesis, we have shown that developing against a zero-core modular architecture is trivial. By utilizing separation of concerns, a module developer needs to only understand the feature they want to extend, or if it is an entirely new feature, find out what has been done before.

6.1.1 Unstable API

Developing against an unstable API is difficult, and when developing a module architecture, it is like an unstable API when it is not *mature*, e.g. when it does not have settled modules to develop against. Since this is the case, there are a lot of issues with the existing modules, making the user experience less than competing IDEs.

Most of these issues are minor, and can be fixed with some minor revisions to the existing modules, for instance, when closing the IDE, unsaved changes are discarded, with no information given to the user. Or how what project a user was working on, is not saved between instances, so a user has to re-open the project they worked on. This is a side effect of the development plan, and not the architecture.

To fully test out this architecture, it was thought that a wide range of modules should be implemented, to quickly iron out issues with the implementation of the architecture, and to figure out what functionality Tauri has, that we can expose, like the file selection.

So not only were modules needed to cover the necessities to qualify as an IDE, but they were also needed to *test* the implementation. Not having a developer dedicated to only implement modules, meant that module development was usually dropped for other things. As every time a module where worked on, it would eventually lead to a discovery, that the current API needed some change, which would enable the module feature to be easier implemented. A concrete example of this, is the editor module.

An essential part of an editor, in an IDE at least, is being able to utilize a LSP. Most of the communication between a client and LS, require information about *where* the user is in the text. This information is available in a *textarea*-element, but to get this information, we needed to change how events were being sent.

We made events gather information about the DOM-event they were triggered by, so in the case of *click* attribute, we know the DOM-event is of type *MouseEvent*, which can give us some, information. And if the *target*, (a field on *MouseEvent*) is an instance of *HtmlInputElement* or *textarea*, we know that the *selectionStart* and *value* field exist on the target. With which, we can manually calculate the position of the click. Implementing this meant adding a breaking change to the API, which deprecated different modules, so more time was spent on re-implementing them.

6.2 Inconsistent UI and ad-hoc solutions for lackluster API

There is an issue in the current API on maintaining consistency between the UI in the frontend, and the UI representation the RSMS has access to. This means that there is no good way to achieve saving of an edited file. But this is still a feature the editor

module supports. The way saving was implemented, was to add plain JavaScript to the save button, where when the user presses it, the contents of the textarea are sent to the ide_fsa module, which can save the contents. This is an action that bypasses the core IDE, which was necessary due to the lacking API.

6.3 Lacking language agnosticism

Not really achieved, because we cannot syntactically translate between a JavaScript module and a Rust module. This is due to the differences between the utility libraries created for JSMS and RSMS. When JSMS modules where created, they were primarily made for using existing JavaScript libraries, to showcase this interoperability. So, much of the HTML elements were created using JavaScript, so the utility library primarily focused on this, having builder pattern for creating HTML. There is a similar builder pattern in the Rust utility library, but it is not a one-to-one mapping, meaning there are some semantic differences between two modules doing the same.

But *installation* of the modules also differ. In JavaScript, module developers can simply invoke the *installModule* function, with their created module, to install the module. The reason this works, is that when we bundle all JavaScript modules during compile time, it ends up as a script-tag in the DOM. The same is for the case of runtime JavaScript modules. The result, in either case, is that the entire contents of the JavaScript file is evaluated, meaning even though we are simply importing a JavaScript file, and not explicitly invoking anything, it ends up with the modules being installed.

This is not the case in Rust, importing another Rust crate does not mean we invoke it. That is why we need the extra steps of creating a *ModuleBuilder*, which has to implement the *ModuleBuilder* trait, so that we can build the module.

6.4 Foreign modules

Languages like Gleam and PureScript, which compile directly to JavaScript can be trivially added. But for languages that can target the C-ABI, this is less trivial. This is because of how the core-IDE was designed. We decided to use a *Rust-y* approach, meaning we utilized many of the features that made interoperability between the Rust-ABI

and C-ABI more complex. An example of this, can be found in the listing 6.1 and 6.2, where we have the *standard* value variant, and then the *C-safe* variant.

```
    Null,
    Int(i32),
    #[ts(as = "f32")]
    Float(NotNan<f32>),
    Bool(bool),
    Str(String),
    List(Vec<Value>),
    #[ts(type = "Record<string, Value | undefined>")]
    Obj(HHMap),
    Html(Html),
}

#[derive(Default, Debug, Clone, PartialEq, Hash, Eq)]
```

Listing 6.1: Value variant (Rust)

```
#[repr(C)]
#[derive(StableAbi)]
pub struct RValue {
    pub(crate) kind: RValKind,
    pub(crate) val: RValueUnion,
}
```

Listing 6.2: C-safe value variant (Rust)

Note the #**[repr(C)]** macro attribute, and the two fields, *kind* and *val*. The macro attribute specifies to the Rust compiler that it should *do what C does*. This is in regard to order, size and alignment of fields of a structure. Since we cannot have the same enum structure as we can in Rust, the work-around was an enum that specifies what kind of value we are working with (*val*), and a union, that holds the specific value. A union in both C and Rust, has the same size in memory, as the largest possible value it can store. In listing 6.3 we can see this union. Accessing a field is inherently an *unsafe* action, as we cannot tell the compiler if the bytes we are reading are actually and integer, or is a list of values. We can see this, as in the listing 6.4, on line three, we have to use the *unsafe* keyword in Rust, which essentially means the compiler cannot promise what we are doing in this code block is *valid*.

```rust
#[repr(C)]
#[derive(StableAbi)]
pub union RValueUnion {
    pub(crate) _int: i32,
    pub(crate) _float: f32,
    pub(crate) _bool: bool,
    pub(crate) _str: ManuallyDrop<RString>,
    pub(crate) _lst: ManuallyDrop<RVec<RValue>>,
    pub(crate) _obj: ManuallyDrop<RVec<RKeyPair>>,
    pub(crate) _html: ManuallyDrop<RHtml>,
}
```

Listing 6.3: Union used to hold the values the C-safe value can have (Rust)

```rust
    pub fn int(&self) -> Option<i32> {
        if self.kind == RValKind::Int {
            Some(unsafe { self.val._int })
        } else {
            None
        }
    }
```

Listing 6.4: Accessing a value in the C-safe value variant is inherently unsafe (Rust)

But with the starting point of the runtime Rust module system, a C module system could be developed. One would just have to ensure that the differences between the modules are syntactical, and not semantics.

6.5 Conclusion

Developing against an unstable API, means that modules can be deprecated. It also means that module language agnosticism can quickly disappear, since that depends on having multiple different libraries in sync with an unstable one. In fact, many of the issues that we have claimed to be innate with IDEs, appear in this stage of our modular architecture. But, our API is bounded, we have some types, and some operation on those types. We have chosen to have a larger set of operations, simply due to the fact that this enhances the module developer experience. But to figure out what utility functions are necessary, we need to develop modules. Once the satisfactory functions are developed, our API is stable. Which means modules are no longer in danger to be deprecated. Which means that module language agnosticism can be corrected for. Finally, this means that

future changes coming from the outside, being a paradigm shift on what is necessary to have in an IDE, or a language, the necessary modules can quickly be developed and integrated into the current solution.

Chapter 7

Future Work

In this chapter, we will discuss improvements that can be made to this implementation, and improvements that similar architecture should consider.

7.1 Testing

There should be tests made to ensure that a module behaves the same, if written in Rust or JavaScript. This can be an issue, because the way a module interacts with the IDE is through a core, which is implemented separately for the different module systems. Test modules should be made, ensuring that the end state of the IDE is the same, when the module does the same action. But this would require the modules being semantically the same for the different test cases. In any case, difficult to ensure all edge cases have been covered.

7.1.1 End-To-End testing

We cannot guarantee that all the utility methods in the JavaScript and Rust libraries work as expected. A proper test scenario that covers a wide array of functionalities offered by the different libraries should be implemented. An example scenario, is one where two modules, one in the Rust ecosystem, the other in the JavaScript ecosystem, sending back and forth different values, and modifying them, using their own implemented methods.

1. Both modules receive an event with some argument
2. Both modules respond with an event, with some complicated argument

We can then assert if the manipulation is like expected.

7.2 Language agnosticism improvements

Steps should be made to mitigate the shortfall of this solution, with regard to language agnosticism. The differences in installation for RSMS and JSMS are mainly due to how trivial it is to install JavaScript modules, compared to Rust modules. JSMS should enforce a similar system of module building as RSMS, not only to ensure less semantic differences, but also to ensure safety, as restricting the JSMS is good.

7.3 Attribute and instructions

It is not possible to remove *eventListener*s. The reason for this is due to how event listeners are created. Since they send an event when they are triggered, they need to have the event to send as an argument, and to remove them from the DOM, we need the same reference to the function. If we want to remove them, we would have to require the module developer to pass the exact same event they used to create the event listener, which could be complicated.

A solution could be to store the created function, by mapping it to the event name used in the creation, but what if the user has two events to send on a button click, with the same name, but different arguments? How would that be solved in a meaningful manner?

7.4 Key presses

A common feature of IDEs is being able to have certain keybindings for different actions. For example, in VS Code, one can hit $CTRL + n$ to open up a new tab, with a new file. This system is not yet possible in the IDE, but this is due to a lack of a supporting module family. But, given that this is a common feature in IDEs, this should be a priority.

7.5 Inconsistent UI representation

Difficult to keep the UI representation consistent with the DOM. An example of this, is that the UI representation in the IDE does not store information like the possible *value* an HTML might have. So for the editor module, there is no efficient way to know what text is in the editor. Another example, is for the module installer, there is no way for the module to *query* the UI for information about the form it presents the user, seeing what values are in the fields. A workaround to this was used, where depending on what element an *eventListener* was added to, the sent event would be *sticky*, meaning it would add extra arguments to the *args* field of the event, like attribute information, id, value, etc. But this would not update the UI stored in the IDE, but rather give modules a peek at the current UI state. A better solution would be to somehow keep track of *all* user interactions to the DOM, and somehow bubble these changes down to the backend, where the UI representation is managed.

7.6 Unify the tooling

A user has to specify what kind of module they are adding, the language and package manager if it is a JavaScript module. This is trivial to detect by a program. A user should be able to simply invoke the tool with either a URL or a path to the module, and then the tool can infer what kind of module it is, and add it to the configuration file correctly. This should be integrated into the IDE, but in a manner where the different installers are modules themselves, which would enable other module developers or maintainers to extend this functionality. Similarly, other tooling, like generating the module dependency graph should be integrated into the core. By using the Rust conditional compilation system, we can conditionally include or exclude functionality, like these CLI tools, into the IDE, allowing us to serve the IDE without this functionality if wanted.

7.7 Modular editor

The prototype editor module develop for this IDE is subpar compared to existing ones. A new one should be developed, in tandem with a LS client. This will ensure that this IDE can support more languages. This editor should then utilize existing technology that

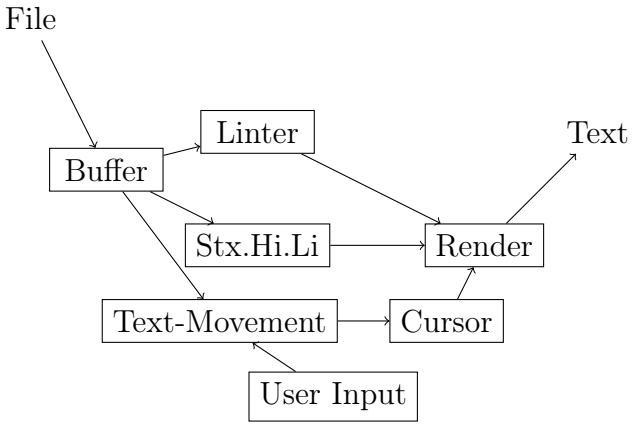


Figure 7.1: Diagram of a module dependency graph for an editor module family. The contents of a file are stored in a buffer, which different modules, linter, syntax highlighting and text movement, can handle the text in their respective format.

is already used by other IDEs, like the tree-sitter[1] parsing system, which amongst other things, can help with syntax highlighting.

7.7.1 Editor buffers

One of the reasons the prototype editor module is subpar, is that it uses a textarea HTML-element as input and renderer of text content. This should be replaced by a buffer system, similar to how other IDEs do. In the figure 7.1, we can see a diagram of a module family which enable a better, more modular, editor.

The general idea is for a buffer to contain the contents of a file. The final step is for some render module to transform the text into some HTML that is actually rendered by the IDE to the user. The linter and syntax highlighter modules add colour to keywords, depending on what language the file is written in, while the linter adds indications to a user if what they have written is incorrect, if there is a warning or error. Given that files are just a long string of bytes, some cursor module is needed, to keep track of where the user is inserting text. This also allows for other modules, like the text movement module to register user key presses as text movement.

7.8 Improvements to the module architecture

Other popular IDEs have a form of module architecture. For instance, in NetBeans, modules can directly invoke methods of other modules. This is due to the modules being

[1] `https://github.com/tree-sitter/tree-sitter`

written in the same language, and can have the same underlying ABI. We circumvented this issue, by having our IDE be the intermediary, but this restrictions means we can only indirectly invoke the methods of other modules. An improvement to this architecture, would be for modules to directly interact with each other, and for the IDE itself to be entirely made out of modules, the only thing the *core* of the zero-core does, is initialized modules. Letting modules be in charge of their own cycles, when to be invoked, who to invoke, etc.

Acronyms

ABI Application Binary Interface.

API Application Programming Interface.

ASR Abstract Semantic Representation.

AST Abstract Syntax Tree.

BLDL Bergen Language Design Laboratory.

CI/CD Continuous Integration and Continuous Delivery.

CLI Commandline Interface.

CPU Central Processing Unit.

CSS Cascading Style Sheets.

DAG Directed Acyclic Graph.

DOM Document Object Model.

DSL Domain Specific Language.

E2E End-To-End.

Eclipse RPC Eclipse Rich Client Platform.

FSA File System Abstraction.

GUI Graphical User Interface.

HTML HyperText Markup Language.

IDE Integrated Development Environment.

IO Input/Output.

IPC Inter-Process Communication.

JSMS JavaScript Module System.

JSON JavaScript Object Notation.

LS Language Server.

LSP Language LS.

MCDS Multi-way Dataflow Constraint System.

MPSC Multi Provider Single Consumer.

MVC Model-View-Controller.

OS Operative System.

REST Representational State Transfer.

RSMS Rust Module System.

SMT Satisfiability Modulo Theories.

STF Syntactic Theory Functor.

UB Undefined Behavior.

UI User Interface.

VDOM Virtual Document Object Model.

Bibliography

[1] Anya Helene Bagge. 'Constructs & Concepts: Language Design for Flexibility and Reliability'. PhD thesis. Bergen, Norway, 2009. ISBN: 978-82-308-0887-0. URL: http://www.ii.uib.no/~anya/phd/.

[2] Anya Helene Bagge. 'Facts, Resources and the IDE/Compiler Mind-Meld'. In: *Proceedings of the 4th International Workshop on Academic Software Development Tools and Techniques (WASDeTT'13)*. Ed. by Mark van den Brand et al. Montpellier, France: WASDeTT, 2013. URL: http://www.ii.uib.no/~anya/papers/bagge-wasdett13-ide.html.

[3] Karl Henrik Elg Barlinn. 'Automating User Interfaces for a Multi-way Dataflow Constraint System'. MA thesis. University of Bergen, 2022. URL: https://hdl.handle.net/11250/3001144.

[4] Maria Katrin Bonde. 'Declaratively Programming the Dynamic Structure of Graphical User Interfaces'. MA thesis. University of Bergen, 2024. URL: https://hdl.handle.net/11250/3147681.

[5] Pierre Carbonnelle. *Top IDE index*. 2023. URL: https://pypl.github.io/IDE.html (visited on 20/04/2025).

[6] David Chisnall. 'C Is Not a Low-level Language: Your computer is not a fast PDP-11.' In: *Queue* 16.2 (Apr. 2018), pp. 18–30. ISSN: 1542-7730. DOI: 10.1145/3212477.3212479. URL: https://doi.org/10.1145/3212477.3212479.

[7] Evan Czaplicki. 'Elm: Concurrent frp for functional guis'. In: *Senior thesis, Harvard University* 30 (2012).

[8] Evan Czaplicki. *The Elm Architecture*. 2021. URL: https://guide.elm-lang.org/architecture/ (visited on 23/04/2025).

[9] Răzvan Diaconescu and Kokichi Futatsugi. 'Logical foundations of CafeOBJ'. In: *Theoretical Computer Science* 285.2 (2002). Rewriting Logic and its Applications, pp. 289–318. ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(01)00361-9. URL: https://www.sciencedirect.com/science/article/pii/S0304397501003619.

[10] Sebastian Erdweg et al. 'Evaluating and comparing language workbenches: Existing results and benchmarks for the future'. In: *Computer Languages, Systems & Structures* 44 (2015). Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014), pp. 24–47. ISSN: 1477-8424. DOI: https://doi.org/10.1016/j.cl.2015.08.007. URL: https://www.sciencedirect.com/science/article/pii/S1477842415000573.

[11] Hans-Gerhard Gross and Nikolas Mayer. 'Built-In Contract Testing in Component Integration Testing'. In: *Electronic Notes in Theoretical Computer Science* 82.6 (2003). TACoS'03, International Workshop on Test and Analysis of Component-Based Systems (Satellite Event of ETAPS 2003), pp. 22–32. ISSN: 1571-0661. DOI: https://doi.org/10.1016/S1571-0661(04)81022-3. URL: https://www.sciencedirect.com/science/article/pii/S1571066104810223.

[12] Magne Haveraaen and Markus Roggenbach. 'Specifying with syntactic theory functors'. In: *Journal of Logical and Algebraic Methods in Programming* 113 (2020), p. 100543. ISSN: 2352-2208. DOI: https://doi.org/10.1016/j.jlamp.2020.100543. URL: https://www.sciencedirect.com/science/article/pii/S2352220820300286.

[13] Mathias Skallerud Jacobsen. 'Tool support for specifying multi-way dataflow constraint systems'. MA thesis. University of Bergen, 2022. URL: https://hdl.handle.net/11250/3045556.

[14] Jaakko Järvi Knut Anders Stokke Mikhail Barash. 'A domain-specific language for structure manipulation in constraint system-based GUIs'. In: *Journal of Computer Languages* 74.101175 (2023). DOI: 10.1016/j.cola.2022.101175.

[15] Andreas Kornstadt and Eugen Reiswich. 'Composing Systems with Eclipse Rich Client Platform Plug-Ins'. In: *IEEE Software* 27.6 (2010), pp. 78–81. DOI: 10.1109/MS.2010.138.

[16] Daniel Svalestad Liland. 'Least Surprising Dataflows in Constraint Based Graphical User Interfaces'. MA thesis. University of Bergen, 2024. URL: https://hdl.handle.net/11250/3147685.

[17] Beate Skogvik. 'Verification of Guarded Magnolia Satisfactions'. MA thesis. University of Bergen, 2024.

[18] Knut Anders Stokke. 'Declaratively Programming the Dynamic Structure of Graphical User Interfaces'. MA thesis. University of Bergen, 2020. URL: https://hdl.handle.net/1956/22882.

[19] Sander Wiig. 'Third times the charm; a new compiler for the Magnolia Research Language'. Unpublished master thesis. 2025.

Appendix A

Optimizing instructions sets

The following appendix discusses how we can optimize an instruction set. When we use the term *optimize*, we mean to reduce the amount of instructions, by pruning the instruction tree.

A.1 Semantics of the instruction data type

Instruction is a recursive data type, parameterized by **T**. Since the **Instruction** data type, semantically forms a monoid[1] structure, we can apply some of the theory from algebraic structures on our **Instruction** variant, optimizing it. The goal of this optimization, is to see if an **Instruction** tree contains any no-operations, and prune it.

As a recap, in listing A.1, we can see the **Instruction** implementation in Rust.

```rust
#[derive(Debug, Default, Clone, Deserialize, Serialize, TS, PartialEq)]
#[serde(rename_all = "camelCase")]
pub enum Instruction<T> {
    /// No Operation, results in no change to the state
    #[default]
    NoOp,
    /// Adds the given T where the id and/or class is found.
    Add(String, T),
    /// Removes the given T where the id is found.
    Rem(String, T),
    /// Combines two instruction into one
    Then(Box<Instruction<T>>, Box<Instruction<T>>),
}
```

Listing A.1: Instruction (Rust)

[1] See section 4.4 in chapter 4

A.1.1 T agnostic function

The optimalization function is not parameterized by a strict variant of T, meaning it works the same regardless of T. In the IDE the **Instruction**s are parameterized by **Value**, **Html**, **Attr** and **String**. The last three are specific to UI modification, and the last two are modifications on specific **Html** instances. This means that if an **Instruction** parameterized by **Html**, is of the **Rem** variant, we know all other UI **Instruction**s pertaining to the removed **Html** variant are **NoOp**s.

In Rust we can have generic data types, as shown in listing A.1, by the type parameter, T, but we have to restrict the type T to a type that implements the trait **PartialEq**, which means we can use equality on it. We need this restrictions, because the attribute macro **Instruction** has. These macros generate the needed code to implement the different traits:

- Debug: Enables the implementer to be printed to *stdout*
- Default: Implements a default variant of the implementer type, in this case, NoOp
- Clone: Implements a simple **clone** method, to create an owned instance of a borrowed value
- Deserialize & Serialize: Implements the needed methods for encoding and decoding a variant to a JSON representation
- TS: Enables automatic TypeScript type generation of the variant

A.2 Pruning the instruction tree

An *unnecessary* operation is one that leads to an **NoOp**, which is the case of inversable **Instruction**s. This inversability does not occur on all **Instruction**, hence it forming a monoid and not a group, but there are cases where we have inverses, meaning we get **NoOp**s. If we do a remove instruction, and then an add, the remove instruction is an **NoOp**. If, however, we do the opposite, first add, then remove, we get an **NoOp** for both instructions.

Furthermore, since we have the idempotency property on add and remove, regardless of how many times we repeat the same instruction, it is the same as doing it just once, meaning we can reduce a series of the same instructions into a series of no-operations, and a singular instruction.

A.2.1 Optimization rules

This gives us four rules which we can apply to our instruction tree, where $\circledast$ is our binary function.

$$Rem_T \circledast Add_T \implies NoOp \circledast Add_T \implies Add_T \tag{A.1}$$

$$Add_T \circledast Rem_T \implies NoOp \tag{A.2}$$

$$Add_T^1 \circledast Add_T^2 \circledast \cdots \circledast Add_T^N \implies Add_T^1 \circledast NoOp^1 \circledast \cdots \circledast NoOp^{N-1} \implies Add_T^1 \tag{A.3}$$

$$Rem_T^1 \circledast Rem_T^2 \circledast \cdots \circledast Rem_T^N \implies Rem_T^1 \circledast NoOp^1 \circledast \cdots \circledast NoOp^{N-1} \implies Rem_T^1 \tag{A.4}$$

A.3 Implementation

We first start the pruning, by removing all NoOps, and then flattening the instructions, by using the **opt** and **flatten** methods, shown in listings A.2 and A.3 respectively.

```
        let ys: Vec<Instruction<T>> = xs
            .iter()
            .filter(|i| !matches!(i, Instruction::NoOp))
            .cloned()
            .collect();
        if ys.is_empty() {
            return Self::NoOp;
        }

        fn _opt<T: PartialEq + Clone + Eq + Hash + Debug>(ys: &[
            Instruction<T>]) -> Instruction<T> {
            match ys {
                [] => unreachable!("Inputted list in nested function is always
                    non-empty"),
                [z] => z.clone(),
                [Instruction::NoOp, zs @ ..] => _opt(zs),
                [z, zs @ ..] => match z {
                    Instruction::NoOp => _opt(zs),
                    Instruction::Then(fst, snd) if matches!(*(fst).clone(),
                        Instruction::NoOp) => {
                        _opt((*(snd.clone())).flatten().as_slice())
                    }
                    Instruction::Then(fst, snd) if matches!(*(snd).clone(),
                        Instruction::NoOp) => {
                        _opt((*(fst.clone())).flatten().as_slice())
                    }
                    Instruction::Then(fst, snd) => Instruction::Then(
                        Box::new(_opt((*(fst.clone())).flatten().as_slice()
                            )),
                        Box::new(_opt((*(snd.clone())).flatten().as_slice()
                            )),
                    ),
                    _ => Instruction::Then(Box::new(z.clone()), Box::new
                        (_opt(zs))),
                },
            }
        }
        _opt(ys.as_ref())
    }
}
```

Listing A.2: Opt method (Rust): Uses a match statement and a guard to match on a *slice*, (reference to a Vec). The guard lets us add a predicate to our branch, in this case, if y *matches* an NoOp. If it is an empty slice, it's a NoOp, otherwise, it will be an Instruction with all NoOps recursively removed.

```
pub fn flatten(self) -> Vec<Instruction<T>> {
    match &self {
        Instruction::NoOp | Instruction::Add(..) | Instruction::Rem
            (..) => vec![self],
        Instruction::Then(f, s) => {
            let mut xs = f.clone().flatten();
            xs.append(&mut s.clone().flatten());
            xs
        }
    }
}
```

Listing A.3: Flatten method (Rust): Note the lack of return statements, this is because the last expression in a function in Rust, is returned, if the expression does not end with a semicolon.

In listing A.4, we then iterate over each instruction in the sequence, and map each field and value to a counter. If it's an Add instruction, the counter is incremented, if it's a Rem instruction, the counter is decremented. We don't have a way to inform the compiler that we have removed all NoOp and Then instructions, and we need complete match-statements, so we add a catch-all with an *unreachable* macro, which will *panic* with the supplied message. When a Rust program *panic*s, it exits the program with the supplied panic message, if any.

```rust
let mut fv_map: HashMap<(String, T), i32> = HashMap::new();

for instr in sequence.clone() {
    match instr {
        Instruction::Add(f, v) => {
            let key = (f, v);
            match fv_map.get(&key) {
                Some(v) => {
                    fv_map.insert(key, *v + 1);
                },
                None => {
                    fv_map.insert(key, 1);
                }
            }
        }
        Instruction::Rem(f, v) => {
            let key = (f, v);
            match fv_map.get(&key) {
                Some(v) => {
                    fv_map.insert(key, *v - 1);
                },
                None => {
                    fv_map.insert(key, -1);
                }
            }
        }
        _ =>
            unreachable!("'Then' or 'NoOp' instruction should never occur
                in a flattened instruction set"),
    }
}
```

Listing A.4: Modification counting (Rust)

Finally, in the listing A.5, we *unflatten* the sequence of instructions, and check the count for each Add and Rem Instruction. If it is above 0, then that means we have added that field-value pair more times than removing it, but we can still only add it once, so we set the count to 0, and return a Then instruction, since we have the accumulated instructions along with the current Add instruction. If the count is less than 0, then it means we are removing it more times than adding it, similarly, we can only remove it once, so we set the count to 0, and combine the accumulated instruction, with the Rem instruction. Because of our **combine** implementation, we can be sure that the initial NoOp element is removed as soon as possible.

```
    sequence
        .into_iter()
        .fold(Instruction::NoOp, |acc, instr| match instr {
            Instruction::Add(f, v) => {
                let key = (f.clone(), v.clone());
                let i = *fv_map
                    .get(&key)
                    .expect("Should be initialized in the previous pass");
                if i > 0 {
                    fv_map.insert(key, 0);
                    acc.combine(Instruction::Add(f, v))
                } else {
                    acc
                }
            }
            Instruction::Rem(f, v) => {
                let key = (f.clone(), v.clone());
                let i = *fv_map
                    .get(&key)
                    .expect("Should be initialized in the previous pass");
                if i < 0 {
                    fv_map.insert(key, 0);
                    acc.combine(Instruction::Rem(f, v))
                } else {
                    acc
                }
            }
            Instruction::NoOp | Instruction::Then(..) => unreachable!
                (
                "`NoOp` or `Then` instruction should never occur in a flattened
                    instruction set"
            ),
        })
}
```

Listing A.5: Instruction folding (Rust)