Interactive services in a disintegrated development environment

Interaktive Dienste in einer reintegrierten Entwicklungsumgebung
Master-Thesis von Hans Becker
Tag der Einreichung:

1. Gutachten: Prof. Dr.-Ing. Mira Mezini
2. Gutachten: Prof. Dr.-Ing. Guido Salvaneschi
3. Gutachten: Sven Keidel, M. Sc.

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Software Technology Group

Interactive services in a disintegrated development environment
Interaktive Dienste in einer reintegrierten Entwicklungsumgebung

Vorgelegte Master-Thesis von Hans Becker

1. Gutachten: Prof. Dr.-Ing. Mira Mezini
2. Gutachten: Prof. Dr.-Ing. Guido Salvaneschi
3. Gutachten: Sven Keidel, M. Sc.

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den December 1, 2016

_____

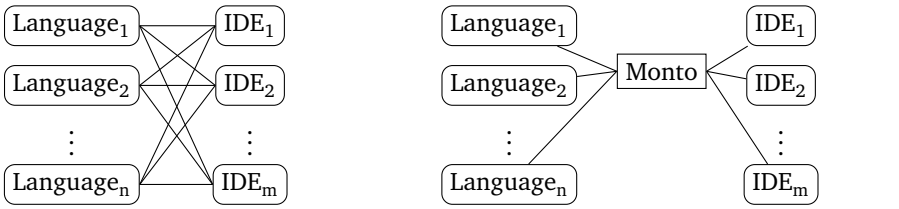
(Hans Becker)

Abstract

Software developers use Integrated Development Environments (IDEs) to help them work faster and more efficiently. IDEs provide a wide varying range of features. To name a few: Syntax highlighting, code completion, code analysis tools, debugger support, etc. Also, there is no shortage of IDEs to choose from. A developer may find him or herself switching between IDEs to have the best experience depending on the language since not every IDE supports every programming language or feature. This complicates the life of language developers, because if they want their language to be adopted by the community, good IDE support is a key factor. Since no modern IDEs share a unified programming interface, the task to integrate one language in every popular IDE is immense.

The Monto framework tries to solve this insufficiency. It dis-integrates features from IDEs and moves them into to IDE-independent services. In a perfect world, where every IDE would support Monto, implementing one IDE feature for a language once, would enable all IDEs to take advantage of this new feature. There have been several contributions to Monto since its first introduction, but interactive services have not been integrated into Monto. This thesis will contribute two important features to the Monto toolset: code completion and debugger support. The foundation to integrate interactive services into Monto also has to be laid out.

Contents



(a) Past situation **(b)** Monto as mediator between languages and IDEs

Figure 1: IDE plugin situation in the past and with Monto [2]

1 Introduction

Integrated development environments (IDEs) are an essential part of a software developer's workflow. They save time, ease otherwise cumbersome tasks and help to find and fix mistakes. Some essential IDE features are:

- **Syntax highlighting** assigns color and text syles information to source code elements. This makes the code easier to read, because similar groups of keywords appear in the same color. It also allows developers to find mistakes more easily, e.g. when a keyword is misspelled it does not get the expected highlighting.
- **Code completion** predicts, suggests, and completes incomplete user input in a code editor to save the developer time. It also is able to show all available functions or properties of a specific code object.
- **Error reporting** shows incorrect or unrecognized code elements and proposes solutions for the problem.
- **Hints** about incorrect or unusual usable of programming languages or APIs to prevent unwanted behaviour and runtime exceptions or crashes.
- **Refactoring** options help to rename and restructure written code to improve readability and reusability.
- **Outline views** visualize elements of a source code file and navigation to these elements. A tree structure is often used as a visualization method.
- **Build management** releaves the developer from compiling source code and managing dependencies by hand.
- **Debugging** written applications is an efficient way to investigate malfunctions and identifying their causes.

Established IDEs like Eclipse or IntelliJ offer extension points to allow new features and new programming languages to be integrated. The development of these extension is complex and time consuming. This is even more unfortunate because these extensions are not interchangable between IDEs. If a language developer does not want to spend a considerable amount of time support all IDEs perfectly, he/she is faced with the decision to either support multiple IDEs rudimentarily or focus on one IDE and support it thoroughly. Either way the success of the new language will be affected negatively, either by bad IDE extension or unavailability of extensions in a particular IDE. One the other side, a developer of a new IDE is affected by a similar problem: His/her new IDE does not immediately support all programming languages. Each languages has to implemented separately. Here again is the success of the IDE limited by missing or bad language support.

This incompatibility and non-reusability between IDE extensions let most IDEs suffer from the "IDE portability problem" [1]. The problem is visualized in Figure 1a and can more generelly be expressed in the following way: Assuming, that there are n languages and m IDEs, with traditional IDEs there need to be $n \cdot m$ implementations if each IDE should support all languages.

The Monto framework[1] solves this problem by introducing an language- and IDE-independent intermediate representation (IR) for IDE features. The actual logic and features like code highlighting, completion or parsing are outsourced from the IDEs to independent services, leaving the IDE as a simplified instrument for user interaction and result displaying. The language specific features are "disintegrated" from the IDE. A service-oriented architecture allows easy integration of new services, but also the reusability of existing ones. Once an IDE has integrated Monto, all programming languages supporting Monto can be edited in it. Vice versa, once a programming language has Monto services, it can be edited in all Monto-supporting editors. Monto reduces the number of implementations in the IDE portability problem to $n + m$ by introducing the Monto IR (see Figure 1b).

The Monto project was started in 2014 by Anthony Sloane et al. by introducing the concept of a disintegrated development environment, the Monto architecture and proving its practicality with a prototype implementation [3]. Sven

[1] https://monto-editor.github.io/about/

Feature \ Language	Java	JavaScript	Python	Haskell
AST[2] generation	✓	✓	✓	✓
Syntax highlighting	✓	✓	✓	✓
Outline view	✓	✓	✓	✓
Spell checking		✓		
Type checking		✓		

Table 1: Previously implemented Monto services per language

Keidel's master thesis extended the functionality of Monto by introducing dependencies, simplified the development of new services and developed an Eclipse-based Monto editor [2]. Wulf Pfeiffer implemented a web-based Monto editor and improved the portability of Monto by adding a discovery and configuration function to services [4]. Support for the python programming language was contributed by Stefan Kockman bachelor's thesis, as well as analytics and improvements on response times [5]. Keidel, Wulff, and Sebastian Erdweg gave an overview of the evolved Monto architecture and processes in a research paper for the 9th Conference on Software Language Engineering [1]. This paper is considered the starting point of this thesis. Table 1 lists which services for which language already exists. Two central services are missing from this list: code completion and debugging. This thesis is aiming at providing a code completionier and debugger service, as well as extending the Eclipse Monto editor to use these new services.

Code completion predicts and suggests code, that the user currently types. Usually a drop-down menu let the user choose between different suggestion and selecting it, inserts the text. Especially for novice users code completion is also a great tool for exploring and learning a language or an API in a language. Experienced users can take advantage of code completion features to save time while writing code with e.g. mechanisms that suggest whole blocks of code.

One central part of each reputable IDE is an integrated debugger, which allows the inspection of written programs during their execution. Breakpoints can be defined at code locations where the executing should halt temporarily. Once a breakpoint is hit, names and values of accessible variables should be displayed. Execution can be controlled manually when a breakpoint was hit in different step sizes or resumed completely normally. Debugging is a very valuable tool at the hand of developers because it greatly simplifies finding und fixing bugs.

Monto should include a language- and IDE-independent services, that provide code completions and support debuggers in the same fashion as existing services. However, existing services operate stateless. Code completions are not stateless, because they depend on the cursor position in a source file. A debugger is by design a heavily stateful tool because as one example its instructions build upon each other ("start debugging", "set breakpoint", "step over", ...). Giving meaning to these instructions is only possible when the previous ones are also known. Stateful services currently cannot be integrated into Monto, because of the lack of stateful equivalent required to represent them.

In summary these are the contributions of this thesis:

- Create an IR that allow stateful operations in Monto
- Create an IR for code completions and implement code completion services
- Create an IR for debugging support and implement debugger services

2 Code completion service

The code completion service should provide useful completions based on the position in a source file. For simplicity reasons only identifier names (in Java that would be class, interface, enum, function, variables, field names and imported elements) were considered as completions. When these are found, the cursor position in the source file is used to filter relevant identifiers. The scoping ranges of identifiers are not considered. Also language keywords (e.g. in Java `"if"`, `"while"`, `"class"`, `"int"`, `"throws"`, `...`) are not completed.

The completion workflow can, therefore, be split up into two separate parts and consequently two language services:

1. `IdentifierFinder` searches through an abstract syntax tree (AST) and finds identifiers.
2. `CodeCompletioner` extracts the string next to cursor and filters out valid identifiers.

Before diving into details about the IR and services, a brief technical overview of Montos architecture needs to be given.

2.1 Overview of Monto's architecture

A more complete and detailed description can be found the paper "The IDE Portability Problem and Its Solution in Monto" [1]. Three types of components make up Monto:

1. IDE
2. Broker
3. Language services

Language services run in the background as standalone servers and perform language-specific tasks. The **IDE** acts as the interaction point for the user and displays results of language services (called products) to the user. A central background service, the **broker**, redirects, caches and dependency-resolves messages sent between IDEs and language services. All communications between these three components are sent over network sockets using the ZeroMQ[3] library. All messages are sent in the universal JavaScript Object Notation (JSON) format to ensure maximal compatibility with most programming languages and platforms, which ZeroMQ also provides. All of the Java, Python and JavaScript language services are written in Java using a base project[4], that includes shared components. The broker, as well as the Haskell language services, are written in Haskell.

One typical communication could look like this: The user changes a source code file, which prompts the IDE to send a `SourceMessage` to the Broker. It distributes the `SourceMessage` as a request to all services, that subscribed to `SourceMessages`. The services then process the `SourceMessages` to products, which can be anything from an AST to a syntax highlighting. Products are sent to the broker as `ProductMessages`, which forwards them to the IDE. The IDE decides if it wants to act on the received `ProductMessage` and may display its content to the user and the cycle can start over.

2.2 Static dependencies of language services

The Monto SLE research paper [1] mentions, that services can declare dependencies on `SourceMessages` or products but does not go into detail. Management of dependencies is important to the code completion and identifier finder service, as well as every other service. That is the reason, why they are now explained further. The dependency management and resolution are solely handled by the broker, services just have to register them. The object of dependencies can either be sources (the raw code of a file) or products of other services.

Static dependencies (also called product dependencies) can be declared by services on registration with the broker. They are intended to be used when a service's product does always depend on the source code or another product. Only if all dependencies have arrived at the broker, a request containing these dependencies is send to the service.

These dependencies are stored in a directed acyclic graph (DAG). Nodes in the graph represent ServiceIDs of services, edges represent the actual dependency. A static dependency is defined by a (Product, Language) tuple. It is possible, that one service depends on multiple products of one other service, which leads to the edges having a list of (Product, Language) tuples as a type. Edges point from the service that declared the dependency to the service that produces the dependent product.

Figure 2 shows one exemplary static dependency DAG. The source is not produced by any real service (send by the IDE as a `SourceMessage`), but it has to be part of the graph so that other services can depend on it. To assure that, the "source" node is always added to the graph. The exemplary graph has four services: The `JavaHighlighter` and

[3] http://zeromq.org/

[4] https://github.com/monto-editor/services-base-java

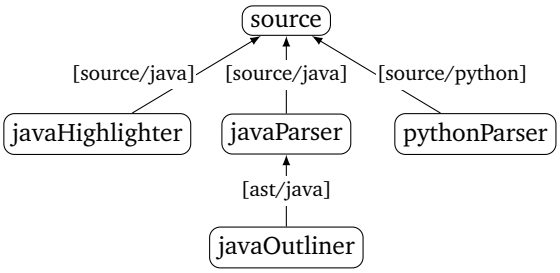


Figure 2: Example static dependency graph

`JavaParser` services depend on Java sources, which is represented by the [source/java] edges from the service nodes to the source node. There is also the `PythonParser` service, which also depends on sources, but only python ones, achieved by the edge having the type [source/python]. The "source" on the edges normally should be products produced by service, but because the source is not a real product, "source" is used as a substitute. The `JavaOutliner` service depends on the AST product of the `JavaParser`, resulting in the edge from `JavaOutliner` to `JavaParser` with type [ast/java]. If any service would depend on multiple products of a service, the edge type would grow to e.g. [ast/java, errors/java]. Every time a new source or product arrives at the broker, it saves the received element and checks if there are services, whose dependencies are now fulfilled. If so, it sends out requests to those services containing their requested dependencies. This process for the perspective of a service is exemplified in Section 2.4

In most situations, the graph is a directed connected tree with the root being the source node because the source is the starting point of all existing Monto services so far. But it is also possible, that a service creates a product, that does not depend on the source, which would make the graph no longer connected and therefore no longer a tree.

2.3 Identifiers IR

As mentioned in the beginning of this section, the code completion process is split up into two separate services that build upon each other. Services that handle the first process are called `IdentifierFinders`. The IR that is used by `IdentifierFinder` services will be defined now. They are expected to create a product with name `"identifiers"`. This name would appear in a `ProductMessage` as the `product` field. To describe the JSON object structure of the `contents` field in a `ProductMessage`, the same syntax as in [1] is used here. `"identifiers"` products are expected to provide a JSON element in this format:

```
Identifiers ::= Identifier*

Identifier ::= { identifier: String,
                 type: String }
```

`Identifier*` denotes a JSON array of type `Identifier`. The `"identifier"` field contains the actual name of the identifier (e.g. a class or variable name), `"type"` categorizes the identifier further into `"class"`, `"interface"`, `"import"` and others (used to display icons based on the origin of each identifier). To exemplify the usage of this IR, the `IdentifierFinder` written for the Java language is documented now.

2.4 IdentifierFinder service for Java

The source code of the Java class `Class1` is shown in Listing 1. This class will be used as en example to illustrate how the `JavaIdentifierFinder` works, as a representative of any service with static dependencies. `JavaIdentifierFinder` has a static dependency on Java sources and Java AST products from the Java parser service. The resulting `RegistrationRequest` JSON (IR in [1]) is shown in Listing 2. The `"dependencies"` array shows the source dependency as the first item and the AST product dependency, produced by the `"javaJavaCCParser"` service, as the second item. The broker writes this dependency into the DAG and waits until both the `SourceMessage` and `"ast"` `ProductMessage` of one source arrive and then sends them to the `JavaIdentifierFinder`. The resulting `Request` is displayed in Listing 3. The two requested dependencies are included in the `requirements` field. The actual AST is hidden, because of its larger size. The exact makeup of the AST depends on the parser service, but follows the AST IR given in [1]. `JavaIdentifierFinder` then visits each node in the AST and looks for nodes that define identifiers. Once such a node is found, the source code from the source dependency is used to extract the actual name of the identifier as a string, because the AST node only references the position in the code rather then the string itself. All these strings are collected and when all AST nodes are visited, a `"identifiers"` `ProductMessage` is sent out. Listing 4 shows the `ProductMessage` JSON that `JavaIdentifierFinder` produces when it finished processing `Class1`.

One major drawback of depending on the AST product is, that if the source file contains syntax errors no AST can be constructed and therefore also no identifiers can be extracted. The code completion would then only work with a valid source file, which is particularly unfortunate while writing code because most sources are not error-free while typing. To bypass this issue, the `JavaIdentifierFinder` was equipped with a fallback that works on the raw source code instead of an AST. The fallback operates in the following way:

1. All comments are removed.
2. All strings and characters are removed.
3. All non-alphanumeric characters are replaced with spaces.
4. All numbers are removed (including scientific notation, explicit double, float, long values, and hexadecimals)
5. Source code string is split on spaces.
6. All duplicates are removed.
7. All Java keywords and literals are filtered out.

The remaining strings are assumed to be identifiers.

Listing 1: Java source code of `pak1.Class1` class

```
package pak1;

public class Class1 {
    public String field1 = "1";
    public void function1() {}
}
```

Listing 2: `RegistrationRequest` of the `JavaIdentifierFinder` service with two static dependencies

```
{
  "service_id": "javaIdentifierFinder",
  "label": "Identifier Finder",
  "description": "Tries to find identifiers from AST, but can also find codewords from source message, if AST is not
       available",
  "options": [],
  "products": [{
      "product": "identifiers",
      "language": "java"
  }],
  "dependencies": [
    {"language": "java"},
    {
      "service_id": "javaJavaCCParser",
      "language": "java",
      "product": "ast"
    }
  ],
  "commands": []
}
```

Listing 3: `Request` to process textttpak1.Class1 sent to the `JavaIdentifierFinder` with a source and AST product dependency

```
{
  "source": {
    "physical_name": "TestEcl/src/pak1/Class1.java",
    "logical_name": "pak1.Class1"
  },
  "service_id": "javaIdentifierFinder",
  "requirements": [
    {
      "contents": "package pak1;\n\npublic class Class1 {\n    public String field1 = \"1\";\n    public void function1() {}\
           n}\n",
      "language": "java",
      "source": {
        "physical_name": "TestEcl/src/pak1/Class1.java",
        "logical_name": "pak1.Class1"
      },
      "id": 2
    },
    {
```

```
      "contents": {
        <AST of Class1>
      },
      "service_id": "javaJavaCCParser",
      "language": "java",
      "source": {
        "physical_name": "TestEcl/src/pak1/Class1.java",
        "logical_name": "pak1.Class1"
      },
      "id": 2,
      "product": "ast"
    }
  ]
}
```

Listing 4: `Identifier` `ProductMessage` of source `pak1.Class1`

```
{
  "id": 2,
  "source": {
    "physical_name": "TestEcl/src/pak1/Class1.java",
    "logical_name": "pak1.Class1"
  },
  "service_id": "javaIdentifierFinder",
  "product": "identifiers",
  "language": "java",
  "contents": [
    {
      "identifier": "Class1",
      "type": "class"
    },
    {
      "identifier": "field1",
      "type": "field"
    },
    {
      "identifier": "function1",
      "type": "method"
    }
  ]
}
```

When looking at the source code of the Java class `Class2` (Listing 5), an `import` statement on `Class1` can be seen. This dependency on another class should also be considered by the `JavaIdentifierFinder` because identifiers of imported classes should also be part of code completions in the importing class. Therefore, all identifiers defined in `Class1` should also be identifiers of `Class2`. This is no longer a static dependency but a dynamic one because it orginated from the source code. But Monto is equipped to handle dynamic dependencies.

Listing 5: Java source code of `pak2.Class2` class

```
package pak2;

import pak1.Class1;

public class Class2 {
    protected boolean yesOrNo = true;
    public float field2 = 9_4_00e5f;

    public interface ITest {}

    public static void main(String[] args) {
        System.out.println("testing");
        Class2 test = new Class2();
    }
}
```

2.5 Dynamic dependencies of language services

Dynamic dependencies (also called file dependencies) can be declared at any point in time after the service registered. They are intended to be used when concrete source files depend on other source files, e.g. when a Java class uses import statements.

The dynamic dependency graph is very similar to the static dependency graph with the only difference being, that nodes no longer represent just ServiceIDs, but (Source, ServiceID) tuples. Edges still stand for lists of (Product, Language) tuples. Figure 3 depicts a built dynamic dependency graph. To better understand the circumstances, that led to this graph, the example with `Class1` and `Class2` from Section 2.4 is continued here. The `JavaIdentifierFinder` should not only contain identifiers for the requested source, but also identifiers of imported sources. If `Class2` imports `Class1`

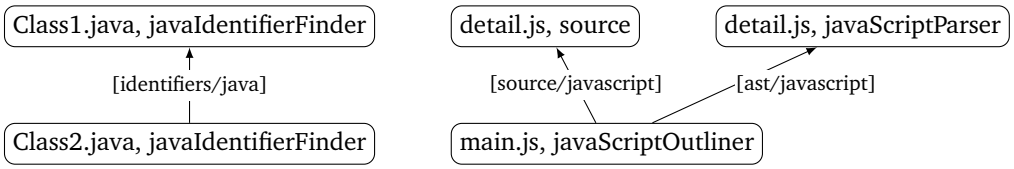


Figure 3: Example dynamic dependency graph

and the `JavaIdentifierFinder` should generate identifiers for `Class2`, it can create a dynamic dependency on the `"identifiers"` product of `Class1` instead of sending the requested `"identifiers"` product for `Class2`. In other words, it delays the delivery of a product by creating a dependency. The IR for registering dynamic dependencies is the following:

```
RegisterDynamicDependencies :: = { source: Source,
                                   serviceID: String,
                                   dependencies: DynamicDependency* }

DynamicDependency ::= { source: Source,
                        serviceID: String,
                        product: String,
                        language: String }
```

The `Source` type will be explained later in Section 3. For the sake of this section, it can be assumed to be a simple string. `RegisterDynamicDependencies` can be sent from services to the broker. Listing 6 shows the concrete JSON message sent from the `JavaIdentifierFinder` to the broker for registering a dynamic dependency from `Class2` on `Class1`. Now the depicted [identifiers/java] edge from (Class2.java, javaIdentifierFinder) to (Class1.java, javaIdentifierFinder) gets added in the broker's dynamic dependency graph. Static and dynamic dependencies are intended to coexist. Once the `"identifers"` product of `Class1` is generated, the broker searches for services with fulfilled static and dynamic dependencies and notices the `"identifiers"` product for `Class2` can be requested again. The `JavaIdentifierFinder` then receives a `Request` for `Class2` similar to the one in Listing 3 but also containing the `"identifiers"` product of `Class1` in the `"requirements"` array. Now `JavaIdentifierFinder` can search the identifiers in `Class2` and simply append identifiers of `Class1` to successfully generate the completions of `Class2`. The resulting `ProductMessage` is shown in Listing 7. The last three identifers in the `contents` field are the ones imported from `Class1`.

Dynamic dependencies are saved in the broker and not removed when the dependencies are fulfilled once. If a source was changed, the broker will respect and wait for previously requested dependencies before sending a request to a service. If a service notices that dynamic dependencies are no longer correct and redeclares them, the previous dependencies get overridden.

Listing 6: `"identifiers"` `ProductMessage` of source `pak2.Class2`

```
{
  "source": {
    "physical_name": "TestEcl/src/pak2/Class2.java",
    "logical_name": "pak2.Class2"
  },
  "service_id": "javaIdentifierFinder",
  "dependencies": [
    {
      "source": {
        "physical_name": "TestEcl/src/pak1/Class1.java",
        "logical_name": "pak1.Class1"
      },
      "service_id": "javaIdentifierFinder",
      "product": "identifiers",
      "language": "java"
    }
  ]
}
```

Listing 7: `"identifiers"` `ProductMessage` of source `pak2.Class2`

```
{
  "id": 2,
  "source": {
    "physical_name": "TestEcl/src/pak2/Class2.java",
    "logical_name": "pak2.Class2"
  },
  "service_id": "javaIdentifierFinder",
```

```
  "product": "identifiers",
  "language": "java",
  "contents": [
    {
      "identifier": "main",
      "type": "method"
    },
    {
      "identifier": "yesOrNo",
      "type": "field"
    },
    {
      "identifier": "Class2",
      "type": "class"
    },
    {
      "identifier": "test",
      "type": "variable"
    },
    {
      "identifier": "ITest",
      "type": "interface"
    },
    {
      "identifier": "field2",
      "type": "field"
    },
    {
      "identifier": "args",
      "type": "variable"
    },
    {
      "identifier": "Class1",
      "type": "class"
    },
    {
      "identifier": "field1",
      "type": "field"
    },
    {
      "identifier": "function1",
      "type": "method"
    }
  ]
}
```

With the introduction and usage of dynamic dependencies the `JavaIdentifierFinder` is completed. `IdentifierFinder` services for the Python and JavaScript languages were also implemented as part of this thesis. They do not include the fallback option of the `JavaIdentifierFinder` and only operate on the AST. Now the second stage of the code completion process needs to be defined.

2.6 Need for stateful messages in Monto

Up until now, all services are triggered by the arrival of a new source file or product. This allows services to be stateless because for each trigger element (source or product) always the same product is generated. They do not depend on any other previous operation. But IDEs usually only show code completions when the user invokes a keyboard shortcut like `Ctrl+Space`, so there is no actual change to the source file itself. Therefore having the arrival of a new source file or product as the only trigger for language services does not fit the need of interactive services like the code completion.

Debugger support in Monto is a main contribution of this thesis. When considering debuggers, the need for stateful messages and triggers becomes even clearer. A debugger is a prime example of an interactive service. It operates on a dialog-like request-response principle. Starting a debug session requires the debugger service to save the state of the session, which makes it stateful. To trigger debugger actions (like stepping commands, adding or removing breakpoints) a new type of communication with services needs to be created. The code completion process can also be represented by a session. A command triggers the code completion process and creates a temporary session that ends with the completions being send out. There is no other command interacting with the session except the one that creates it.

Two main concepts need to be provided by the Monto architecture so that interactive services can operate properly: sessions and commands. The concept of sessions is rather intuitive in stateful operations. Commands represent interactive requests to a session.

2.7 CommandMessage IR

The introduction of `CommandMessages` implements the concept of sessions as well as commands in the Monto framework to enable services to be interactive. A `CommandMessage` is defined as:

```
CommandMessage ::= { session: Int,
                     id: Int
                     command: String,
                     language: String,
                     contents: JSON,
                     requirements: Message* }

Message ::= SourceMessage | ProductMessage

SourceMessage ::= { id: Int,
                    source: Source,
                    language: String,
                    contents :: String }

ProductMessage ::= { id: Int,
                     source: Source,
                     service_id: String,
                     product: String,
                     language: String,
                     contents: JSON }
```

Now a description of the individual fields of `CommandMessage`:

- The **`id`** field distinguishes between `CommandMessages` of one session.
- Sessions are identified by an integer and referenced by `CommandMessages` with the **`session`** field. The value of `session` has to be set by one component in Monto to assure the uniqueness. Just like the `id` of a `SourceMessage`, the IDE determines the `session` number.
- **`command`** acts like a tag for the whole message, it describes what the message intent is (e.g. `"completeCode"` or `"addBreakpoint"`)
- **`language`** is necessary in combination with `command` to find the service of the correct language to process this `CommandMessage`.
- **`contents`** can contain any JSON element or `null`. It is used to transmit additional information required so that the command can be executed (e.g. opened source file and position in the source file when code completions should be generated). `contents` is similar to the `contents` field of `ProductMessages`.
- **`requirements`** contains requested dependencies (see Section 2.9) and serves the same purpose as the `requirements` field of a `Request`.

A `CommandMessage` can be sent by the IDE to the broker when the user indicates, that an action, other than a change to a source file, should be initiated. Like stated before, this can be a signal to start a debug session, to execute a step command in the debugger, or to prompt code completions. The broker now has the task to relay the `CommandMessage` to the correct service/services. But without further changes this cannot be achieved, so, to begin with, services need to specify which `CommandMessages` they want to receive. To accomplish this, the `RegisterServiceRequest` from [1] was extended by the `commands` field to:

```
RegisterServiceRequest ::= { serviceID: String,
                             label: String,
                             description: String,
                             options?: Option,
                             products: ProductDescription*,
                             commands: CommandDescription*,
                             dependencies: ProductDependency* }

ProductDescription ::= { product: String,
                         language: String }

CommandDescription ::= { command: String,
                         language: String }
```

```
ProductDependency ::= { service_id: String,
                        product: String }
```

The "language" string of CommandDescriptions distinguishes between e.g. commands for Python or Java files, because services are usually split up by language. The broker maintains a Map<CommandDescription,[ServiceID]>, which gets updated when new services register. Once a CommandMessage reaches the broker, it extracts command and language, creates a CommandDescription to query the Map for services, that have registered for this CommandMessage, and sends out the CommandMessage. This finishes the distribution of CommandMessages using the publish-subscribe pattern.With the concept of CommandMessages beeing introduced, the IR for CodeCompletioners can be determined.

2.8 Completions IR

The CodeCompletioner services are not triggered on source file or product arrival, but with CommandMessages. To receive these CommandMessages they need to subscribe to "completeCode" commands in their respective language while registering with the broker. "completeCode" commands are expected to contain a JSON element of the following format:

```
CompletionRequest ::= { source: Source,
                        selection: Region }

Region ::= { offset: Int,
             length: Int }
```

Region is an existing format in Monto and describes a selected area in a source with the selection starting "offset" characters from file start and being "length" characters long. On reception of a "completeCode" command they are expected to generate a product of name "completions" with content:

```
Completions ::= Completion*

Completion ::= { description: String,
                 replacement: String,
                 deleteBeginOffset: Int,
                 deleteLength: Int,
                 icon: URL }
```

deleteBeginOffset and deleteLength describe a region in the source file that should be deleted by the editor before inserting replacement. No specific way is preset on how CodeCompletioner services produce this "completions" product.

The CodeCompletioner implemented in this thesis use the "identifiers" product of IdentifierFinders to generate "completions". But there is no way to request dependencies when receiving CommandMessages. It should be possible to define dependencies in a similar fashion to dynamic dependencies (Section **??**). This problem is tackled in the next subsection.

2.9 Dependencies of CommandMessages

When processing a CommandMessage, a service can realize that additional information is needed to complete the command. This additional information is specified with a new type of dependency: a CommandMessage dependency. The static and dynamic dependency graph cannot be reused to manage this new type because the CommandMessage itself has to be saved until all dependencies are fulfilled. The object of CommandMessage dependencies is still either a source or a product. To register dependencies, services can send a message to the broker in the following format:

```
RegisterCommandMessageDependencies ::= { command_message: CommandMessage,
                                         dependencies: DynamicDependency* }

DynamicDependency ::= { source: Source,
                        service_id: String,
                        product: String,
                        language: String }
```

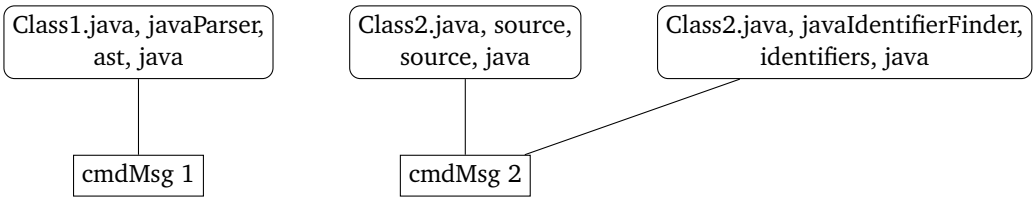


Figure 4: Example `CommandMessage` dependency graph

It became apparent that the new dependency graph is an undirected bipartite graph. Nodes of a bipartite graph can be divided into two disjoint sets and edges of a bipartite graph can only appear between nodes of separate sets. One of these node sets represent `DynamicDependencies` and the other one `CommandMessages`. Edges mark affiliations between `CommandMessages` and dependencies. `service_id` and `product` of the `DynamicDependency` tuple take on the value `"source"` when a source is the object of the dependency, otherwise, it is the actual depending product and the ID of the service producing the product.

In addition to the static and dynamic dependency graph, the `CommandMessage` dependency graph now has also to be checked for fulfilled dependencies, when new sources and products arrive at the broker. Upon registration of the dependency, it is possible, that it is immediately fulfilled because the source or product already arrived at the broker. This case was considered and is implemented correctly. This logic is handled in the functions `commandMessagesWithSatisfiedDependencies` and `isCommandMessageSatisfied` in `Broker.hs`. When all dependencies of a `CommandMessage` are satisfied, they are embedded into the `requirements` field of the `CommandMessage` (see IR in Section **??**) and send out to subscribed services. Once sent out, the `CommandMessage` gets deleted from the dependency graph because there is no need to persist dependencies of one `CommandMessage` since they are non-repeating requests. This is a difference to the static and dynamic dependency graph, where dependencies are kept.

A bipartite graph with node sets `A` and `B` can be modeled with two maps of type `Map<A, B>` and `Map<B, A>`, so that efficient lookup of adjacent nodes from each node set is possible with a complexity of $O(\log n)$. Such a graph data type is constructed in `DependencyGraphCommandMessages.hs` together with functions that allow updating of the graph while making sure that both maps are kept in sync. All of this functionality is implemented in the broker and covered by extensive test cases in `DependencyGraphCommandMessagesSpec.hs`.

2.10 CodeCompletioner services for Java, Python and JavaScript

Now that `CommandMessages` and dependecies of `CommandMessages` are introduced, the `CodeCompletioners` for Java, Python and JavaScript implemented as part of this thesis can explained further. The `JavaCodeCompletionier` will be used as an representative but the Python and JavaScript counterparts work almost the same. It will respect and use the IR defined in Section 2.8.

The first step to accomplish code completions is to register the `JavaCodeCompletioner` using the new RegisterServiceRequest and taking advantage of the new `commands` field to subscribe to Java `"completeCode"` commands. Listing 8 shows the JSON send to the broker. The next step has to be taken by the user or, more specifically, by the IDE. It has to send a `"completeCode"` `CommandMessage` (Listing 9). It contains the source and cursor position in the source at which the user wants to get code completions. In this example to user just typed `"f"` and requested completions. In this specific message, completions for `Class2`, mentioned earlier in this section (Listing 5), at a concreate position are requested. The broker then forwards this `CommandMessage` to the `JavaCodeCompletioner` because it has subscribed to it. `JavaCodeCompletioner` should new generate a `"completions"` product but cannot do so, because the identifier product of `Class2` is not available. To make it available, a `RegisterCommandMessageDepenencies` message has to be constructed and send to the broker. Beside the identifier product, a dependency on the source code of `Class2` is also requested, because it will allow extraction of the text next to the cursor. The `RegisterCommandMessageDepenencies` message is shown in Listing 10. The broker receives the message and inserts the dependency in the `CommandMessage` depenency graph. Figure 4 shows the graph after insertion. Just as a quick reminder, there are two distinct sets of nodes in the graph: The top three nodes are `DynamicDependency` in (Source, ServiceID, Product, Language) tuple form and the remaining two nodes at the bottom are `CommandMessages`. The graph can be read as:

- `CommandMessage` 1 depends on the AST product of the javaParser service of `"Class1.java"`. This dependency was inserted previously.
- `CommandMessage` 2 depends on the Java source file `"Class2.java"` and the identifiers product of the javaIdentiferFinder service of `Class2.java`. This dependency is the just inserted one.

Either the two dependencies are already cached in the broker, in which case the `CommandMessage` get send out immediatelly, or the two dependencies arrive after the dependency registration. Sooner or later, the CommandMessage arrives at

the JavaCodeCompletioner a second time. This time the requested dependencies are included in the requirements field. Listing 11 shows this completed CommandMessage. JavaCodeCompletioner can now use the offset of the CompletionRequest's region as the cursor position; length is ignored. The string to the left of the cursor position is extracted, delimited by the first non-alphanumerical character. In this example it will be just the character "f". All identifiers starting with "f" are considered relevant for this position in the file. Relevant identifiers are enriched with additional informations like an icon and the region in the source code, that should be deleted before insertion. They are therewith converted to Completions, which can be sent out as a ProductMessage. This ProductMessage is shown in listing 12. One small quirk is, that ProductMessages have to include a source, which usually contains a filename, but CommandMessages were introduced to allow source-file-independent actions. So the source of a ProductMessage cannt be the filename of a source. In all of the implemented services so far, using the session id here has proven to be the most useful and logical choice. This completes the JavaCodeCompletioner. The "completions" product now arrives at the IDE, where it shown to the user. The next subsection explains how completions were implemented into the Monto Eclipse editor.

Splitting the code completion process into two services allowed almost complete reusablility of the JavaCodeCompletioner for the Python and JavaScript language. Shared code was extracted to the Java class CodeCompletioner, which is now part of the Java base project.

Listing 8: "RegisterServiceRequest" message to register JavaCodeCompletioner with one CommandDescription subscription

```
{
  "service_id": "javaCodeCompletioner",
  "label": "Code Completion",
  "description": "A code completion service for Java",
  "options": [],
  "products": [
    {
      "product": "completions",
      "language": "java"
    }
  ],
  "dependencies": [],
  "commands": [
    {
      "command": "completeCode",
      "language": "java"
    }
  ]
}
```

Listing 9: "completeCode" CommandMessage sent from IDE

```
{
  "session": 0,
  "id": 0,
  "command": "completeCode",
  "language": "java",
  "contents": {
    "source": {
      "physical_name": "TestEcl/src/pak2/Class2.java",
      "logical_name": "pak2.Class2"
    },
    "selection": {
      "length": 0,
      "offset": 300
    }
  },
  "requirements": []
}
```

Listing 10: RegisterCommandMessageDependencies message sent from JavaCodeCompletioner to request identifiers and source of Class2

```
{
  "command_message": {
    "session": 0,
    "id": 0,
    "command": "completeCode",
    "language": "java",
    "contents": {
      "source": {
        "physical_name": "TestEcl/src/pak2/Class2.java",
        "logical_name": "pak2.Class2"
      },
      "selection": {
```

```
        "length": 0,
        "offset": 300
      }
    },
    "requirements": []
  },
  "dependencies": [
    {
      "source": {
        "physical_name": "TestEcl/src/pak2/Class2.java",
        "logical_name": "pak2.Class2"
      },
      "service_id": "javaIdentifierFinder",
      "product": "identifier",
      "language": "java"
    },
    {
      "source": {
        "physical_name": "TestEcl/src/pak2/Class2.java",
        "logical_name": "pak2.Class2"
      },
      "service_id": "source",
      "product": "source",
      "language": "java"
    }
  ]
}
```

Listing 11: "completeCode" CommandMessage resend by broker with completed identifiers and source dependency

```
{
  "session": 0,
  "id": 0,
  "command": "completeCode",
  "language": "java",
  "contents": {
    "source": {
      "physical_name": "TestEcl/src/pak2/Class2.java",
      "logical_name": "pak2.Class2"
    },
    "selection": {
      "length": 0,
      "offset": 300
    }
  },
  "requirements": [
    {
      "contents": [
        {
          "identifier": "Class2",
          "type": "class"
        },
        {
          "identifier": "ITest",
          "type": "interface"
        },
        {
          "identifier": "main",
          "type": "method"
        },
        {
          "identifier": "test",
          "type": "variable"
        },
        {
          "identifier": "args",
          "type": "variable"
        },
        {
          "identifier": "yesOrNo",
          "type": "field"
        },
        {
          "identifier": "field2",
          "type": "field"
        },
        {
          "identifier": "Class1",
          "type": "class"
        },
        {
          "identifier": "field1",
          "type": "field"
```

```
        },
        {
          "identifier": "function1",
          "type": "method"
        }
      ],
      "service_id": "javaIdentifierFinder",
      "language": "java",
      "source": {
        "physical_name": "TestEcl/src/pak2/Class2.java",
        "logical_name": "pak2.Class2"
      },
      "id": 2,
      "product": "identifier"
    },
    {
      "contents": "package pak2;\n\n//import pak1.Class1;\n\npublic class Class2 {\n    protected boolean yesOrNo = true;\n
              public float field2 = 9_4_00e5f;\n\n    public interface ITest {}\n    \n    public static void main(String[]
           args) {\n        System.out.println(\"testing\");\n        Class2 test = new Class2();\n        f;\n    }\n}\n",
      "language": "java",
      "source": {
        "physical_name": "TestEcl/src/pak2/Class2.java",
        "logical_name": "pak2.Class2"
      },
      "id": 2
    }
  ]
}
```


Listing 12: "`completions`" `ProductMessage` produced by `JavaCodeCompletioner` containing only relevant identifiers (ones starting with "`f`")


```
{
  "tag": "product",
  "contents": {
    "id": 2,
    "source": {
      "physical_name": "TestEcl/src/pak2/Class2.java",
      "logical_name": "pak2.Class2"
    },
    "service_id": "javaCodeCompletioner",
    "product": "completions",
    "language": "java",
    "contents": [
      {
        "description": "field2",
        "replacement": "field2",
        "delete_begin_offset": 299,
        "delete_length": 1,
        "icon": "http://localhost:5050/field.png"
      },
      {
        "description": "field1",
        "replacement": "field1",
        "delete_begin_offset": 299,
        "delete_length": 1,
        "icon": "http://localhost:5050/field.png"
      },
      {
        "description": "function1",
        "replacement": "function1",
        "delete_begin_offset": 299,
        "delete_length": 1,
        "icon": "http://localhost:5050/method.png"
      }
    ]
  }
}
```


2.11 Eclipse editor integration

The Eclipse Monto editor is in large parts base the IMP platform developed by Charles, Fuhrer and Sutton [6, 7]. Is is a meta-tooling platform with the target to ease the development of new Eclipse-based IDEs. Even though it relieves developers from having to use complex interfaces, it permits significant customization of IDE appearance and behavior. The Eclipse Monto editor already uses IMP to relay parsing and highlighting informations to Eclipse, without having to directly interface Eclipse.

IMP also includes an easy to use interface to propose code completions to the user. `IContentProposer` defines the method `getContentProposals()`, which is realized by the ContentProposer class in the Monto editor. When this

(a) AST wasn't available

(b) AST was available

Figure 5: Screenshot of the code completion feature of a Java source in the Monto Eclipse editor

method is called by IMP, the currently opened source file and position in the file in packaged into a `"completeCode"` `CommandMessage`, respecting the IR in Section 2.8. The `"completions"` product is awaited with a blocking call, the `Completions` get translated into their Eclipse counterpart and returned. The blocking call waits at most 100 milliseconds, before returning no completions to avoid blocking the UI for too long.

Figure 5 shows a screenshot of the finished code completion feature of a Java source in the editor window of Eclipse. Two different version of code completions were used, because the Java code completion can work with or without a valid AST. Displaying of icons and the just typed `"m"` being recognized as an identifier are the noticeable differences between the AST-based and fallback identifier extraction method.

2.12 Summary

In this section a language- and IDE-independent code completion service was designed and integrated into Monto. Existing Monto dependency management solutions were described and exemplified. But the interactive code completion service was not able to be integrated with existing mechanisms. To enable interactive services to correctly operate in Monto, the concept of `CommandMessages` was introduced. They allow stateful services to exist and communicate. A new type of dependency had to be introduced, so that `CommandMessages` can take advantage of existing services. IDE support for code completions was integrated into the Eclipse Monto editor using the IMP platform.

In the next section, a new and completely interactive service will be implemented using the just introduced features: A runner service, that enables executing of code written in the Monto framework. It will be a step of preparation before tackling the debugger integration.

3 Runner service

Executing written applications is an essential and quick way to test functionality that was just implemented by a developer. They do not replace separate unit test but can confirm if a chosen solution works like intended. They are also a valuable tool, when testing out new APIs, for which exact details are not known or documented. Usually console output and information about exceptions is shown in the IDE.

This process should be available in Monto to provide users with the just mentioned advantages. The distributed nature of Monto suggests that running applications should be the task of a new language service so that all Monto-supporting editor can reuse it. Services that execute written code are called `Runner` services. To make that possible an intermediate representation of messages has to be defined, that will work with most programming languages. This intermediate representation is then tested by implementing a `JavaRunner` service and integrating into the Eclipse editor of Monto.

Before defining an intermediate representation, a change to an existing Monto data structure has to be made. The reason for the change can be most easily observed, when considering the steps required to execute Java code from a terminal:

```
$ javac com/company/MyClass.java
$ java com.company.MyClass
... program output ...
```

Compilation requires the file name on disk, but executing requires the fully qualified class name. Monto previously just considered the the file name on disk. To fix the problem `Source` has to be redefined and a new service has to be implemented.

3.1 Logical name extractor service

`Sources` previously just consisted out of one String, the name of the in the IDE opened file. This is not sufficient anymore because some services may require the logical name of the file. Compilatio, execution, but also dependency resolvement are use cases where logical name can be useful to know. In Java a logical name would be the fully qualified class name. The type `Source` cannot be just a String anymore and has to be extended to:

```
Source ::= { physical_name : String,
             logical_name?: String }
```

The task of finding the logical name of a `Source` is not trivial and should therefore not be handled by the IDE, but a separate language service. The IDE continues to send out `SourceMessages` without the optional `logical_name` field. A logical name extractor service receives these `SourceMessages` and extract their logical name. A `Source` with populated `physical_name` field is sent out as a new `"logicalSourceName"` product to the IDE. The only Monto component that sends out `SourceMessages` is the IDE. So when it receives a `"logicalSourceName"` product, a new `SourceMessage` with the complete `Source` is sent out again. This means the same source code is sent out twice, which also leads to duplicate product generation, but it is a simpler and cleaner solution than to allow language services to send `SourceMessages`.

Not all services require the logical name of a `Source`, so a logical name extractor service does not have to be implemented for every language immediatelly. A logical name extractor service was implemented for Java sources (`JavaLogicalNameExtractor`). It uses two regular expressions to find class and package name in Java source code files, combines them and sends the new `Source` out as an `"logicalSourceName"` product.

Now that logical names are available in the Monto framework, an intermediate representation for executing application can be determined.

3.2 Runner IR

`Runner` services are only initiated when an IDE user wants to execute an application. Therefore it is interactive and should use `CommandMessages` as defined in Section 2.7. The starting point of each program execution is marked by an `CommandMessage` with a `command` of `"run"`. An entry point to the written code has to be included in this `CommandMessage`. Its contents are expected to contain a `LaunchConfiguration` JSON object in the format of:

```
LaunchConfiguration ::= { main_class_source: Source }
```

`LaunchConfiguration` can in the future be extended to include fields for command line arguments, environment variables, or working directory to specify launch conditions further. With this information `Runner` services are expected to start the execution of the program. If the program writes to `stdout` or `stderr`, a `"streamOutput"` product should be created. Its contents are expected to be in the form of:

```
StreamOutput ::= { source_stream: String,
                   data: String }
```

source_stream either has value "OUT" or "ERR" for stdout or stderr. data contains the actual output written to the stream as a String. Besides output of the program, IDE users additionally want to be informed when the process terminates. This is signalized with the "processTerminated" product with content:

```
ProcessTerminated ::= { exit_code: Int }
```

If a program does not respond any more or does not terminate on its own, it can be terminated forcefully. If the user wishes to do so, a "terminate" CommandMessage with no content can be sent, answered again with a "processTerminated" product, if the program then terminates.

Usage of this IR is illustrated in the next subsection by implementing a Runner service for the Java language. But the IR is designed so that Runner services for other languages like Python and C could also be easily implemented.

3.3 Java runner service

As described in Section 2.7 interactive services need to subscribe to CommandMessages they want to receive. The JavaRunner has to subscribe to Java "run" and "terminate" CommandMessages to fulfill the IR detailed in the last section. This registration will not be explained further because they greatly resemble the one listed in Section 2.10. Once a "run" CommandMessage is received (Listing 13), compiling the main class is the next action, but doing so requires the source code - a CommandMessage dependency has to be requested. Once the dependency arrives, compiling the main class could be done using the javac tool, but it would require writing the source code to a file on the hard disk. This could introduce bothersome synchronization issues. The JDK includes the JavaCompiler[5] interface that allows from-memory-to-memory compilation. No in-memory alternative could be found for the actual execution, so the java tool has to be started in a separate process. This also means the compiled bytecode has to be written to disk. JavaCompiler is flexible enough to also allow from-memory-to-file compilation. All code regarding compilation of Java files is managed in the CompileUtils class.

As just mentioned, the java tool has to be started in its own process for the actual execution of Java programs. The Java class Process[6] serves that purpose. Two InputStreams are provided to read outputs from stdout and stderr. These are read with two InputStreamProductThreads, that convert any read bytes to a string and sent it out in a "streamOutput" product (Listing 14). Besides output of the program, IDE users additionally want to be informed when the process terminates. One additional thread (ProcessTerminationThread) waits for the Process to terminate to then send a "processTerminated" product (Listing 15).

The JavaRunner is the first Monto service with state, because it has to keep track, which processes are started using a Map<Integer, ProcessTerminationThread>. Keys in the maps correspond to the session id of the process-starting "run" command. If the session id was chosen to be 4, consequtive products reference this session id by setting the products source to "session:4". If the IDE wishes to interact with a running session using CommandMessages, the session field of the CommandMessage has to be set to 4 as well. CommandMessages with non-existing session ids will be ignored. At the moment the only time that the Map has to be used, is when a "terminate" CommandMessage is received (Listing 16). To terminate the process of the referenced session, the ProcessTerminationThread is interrupted, which leads it to invoke the destroy() method on the started process. ProcessTerminationThread then again waits for the process to terminate and, if so, sends a "processTerminated" product.

Listing 13: "run" CommandMessage to start a Java execution session

```
{
  "session": 4,
  "id": 1,
  "command": "run",
  "language": "java",
  "contents": {
    "main_class_source": {
      "physical_name": "TestEcl/src/JRunnable.java",
      "logical_name": "JRunnable"
    }
  },
  "requirements": []
}
```

[5] https://docs.oracle.com/javase/8/docs/api/javax/tools/JavaCompiler.html

[6] http://docs.oracle.com/javase/8/docs/api/java/lang/Process.html

Listing 14: stdout "streamOutput" product send by the JavaRunner

```
{
  "id": -1,
  "source": {
    "physical_name": "session:4",
    "logical_name": null
  },
  "service_id": "javaRunner",
  "product": "streamOutput",
  "language": "java",
  "contents": {
    "source_stream": "OUT",
    "data": "I started."
  }
}
```

Listing 15: "processTerminated" product send by the JavaRunner

```
{
  "id": -1,
  "source": {
    "physical_name": "session:4",
    "logical_name": null
  },
  "service_id": "javaRunner",
  "product": "processTerminated",
  "language": "java",
  "contents": {
    "exit_code": 244
  }
}
```

Listing 16: "terminate" CommandMessage to end a Java execution session

```
{
  "session": 6,
  "id": 0,
  "command": "terminate",
  "language": "java",
  "contents": null,
  "requirements": []
}
```

3.4 Eclipse integration

Eclipse's plugin mechanism offers extension points for integrating launch support of custom languages. IMP, the base of the Monto Eclipse editor, greatly simplified previous editor development, but it does not include any extension point for launching programs. So Eclipse's own extension points had to be used directly. A detailed description of all relevant extension points and interfaces can be found in the Eclipse documentation [8] and the article "We Have Lift-off: The Launching Framework in Eclipse" [9]. An abbreviated summary of the most important features is given here now.

3.4.1 Launch configuration and user interface

Just like launching normal local Java application in Eclipse's Java Development Tools (JDT), the first step in launching applications, is to specify the entry point (main class) and other launch conditions. Eclipse handles this by allowing plugin developers to specify their own launch configuration tabs, that create user interface elements and write their launch configuration parameters into a key-value-pair-like data structure. In the Monto Eclipse editor `MainClassLaunchConfigurationTab` handles these tasks. In the method `createControl(...)` user interface elements are created and in `void performApply(ILaunchConfigurationWorkingCopy configuration)` three parameters are written to `configuration`:

1. Physical name of the main class `Source` (including the project folder name in the Eclipse workspace)
2. Logical name of the main class `Source`
3. Language of the main class `Source`

Figure 6 shows the three text input elements of the Monto launch configuration tab.

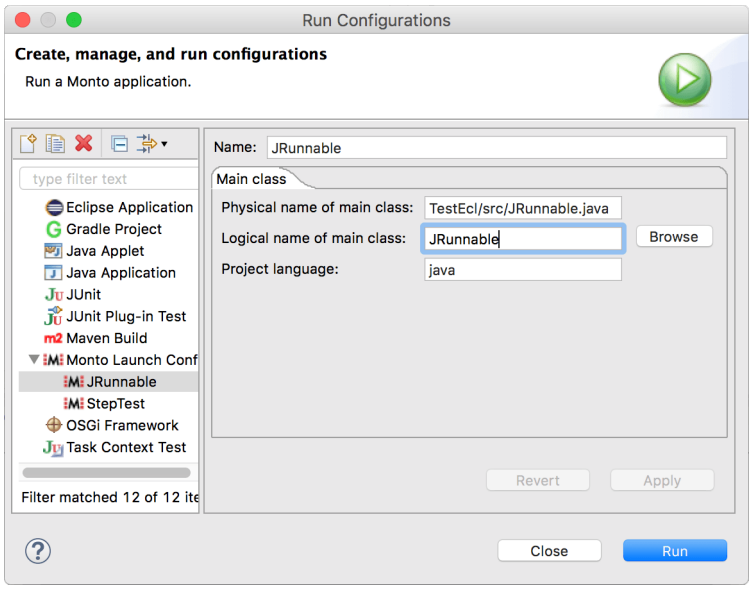


Figure 6: Screenshot of the Monto launch dialog user interface

3.4.2 Process model and console output

After the launch configuration is created, the user can select it in the Eclipse toolbar and run it by clicking the play-button-shaped run button. If registered correctly, Eclipse calls the `launch(ILaunchConfiguration configuration, String mode, ILaunch launch, IProgressMonitor monitor)` method of a `LaunchConfigurationDelegate` class to signalize "The user wants to run a application now". `configuration` is the by the dialog configured instance of the launch configuration, which includes the main class. `mode` is either `"run"` order `"debug"` (see Section **??**). Now all information needed to construct a `"run"` `CommandMessage` (see Section 3.3) has become available, and the `CommandMessage` can be sent off.

`launch` allows attachment of an `IProcess` instance. `IProcess` is the Eclipse interface for showing console output. It's most important methods and references classes are listed in a UML class diagram in Figure 7. Three Eclipse interfaces are shown on the left side and their realizing Monto class counterpart on the right side. `IProcess` references one `IStreamsProxy`, which references two `IStreamMonitors` (one for `stdout` and one for `stderr`). `IStreamMonitors` allow registration of `IStreamListeners` that are notified when new text is appended to a stream. `MontoStreamMonitor` keeps an internal `StringBuffer` to which all text gets appended, when `fireEvent()` is called, as well as notifying all registered listeners. `IStreamProxy` also allows writing to `stdin` via `void write(String input)`, which is ignored by the Monto implementation at the moment. `IProcess` also includes methods tackling process termination (`getExitValue()`, `terminate()`, ...).

After the `"run"` CommandMessage is sent out, an instance of `MontoProcess` and it's contained classes is created and attach to the `launch` instance. Listeners for `"streamOutput"` and `"processTerminated"` `ProductMessages` are installed to call `onProcessTerminatedProduct()` and `onStreamOutputProduct()`. These then forward the contents of the `ProductMessages` to the correct place, so that Eclipse can display them. Figure 8 shows a finished run session and the console output in the Monto Eclipse editor.

3.5 Summary

Using `CommandMessages` and their dependencies, this section documented and illustrated the IR construct to enable Monto to execute applications. A `JavaRunner` service was implemented and the Eclipse Monto editor now supports running applications and showing their output. Almost all of the IR and Eclipse extensions will be used in the next section, when the debugger service is designed and implemented. The `Runner` section can be seen as a first step towards the support of debuggers in Monto.

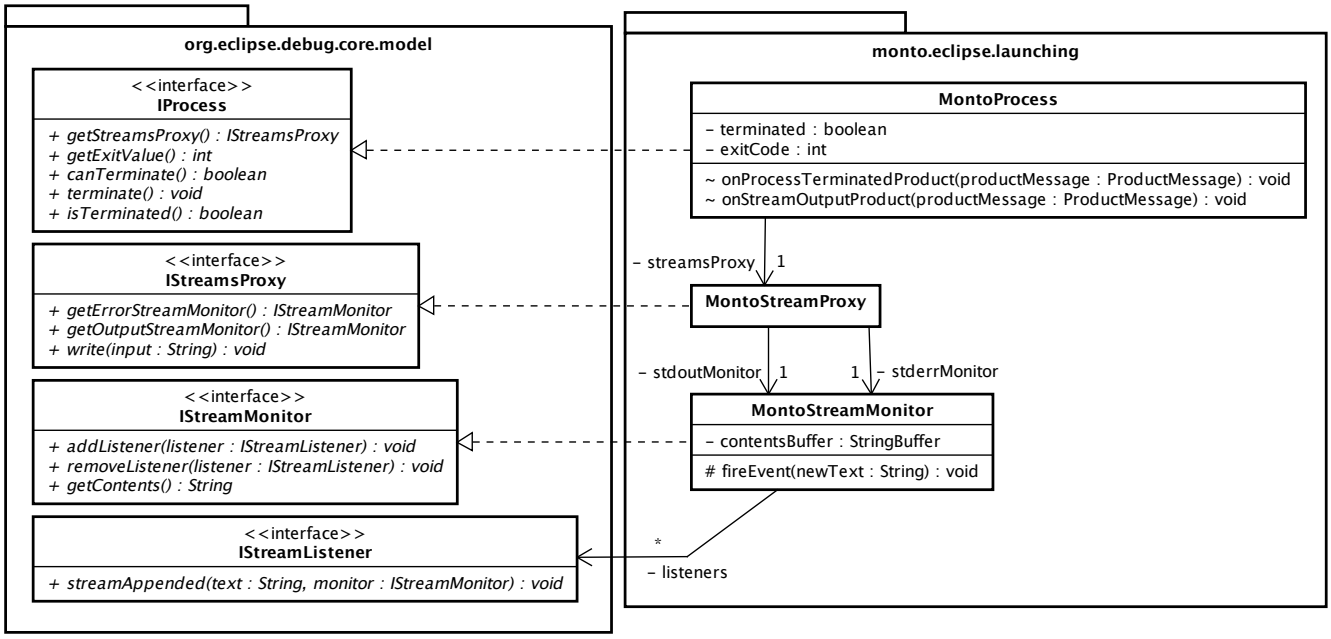


Figure 7: UML class diagram of Eclipse process interface and Monto's realizing classes

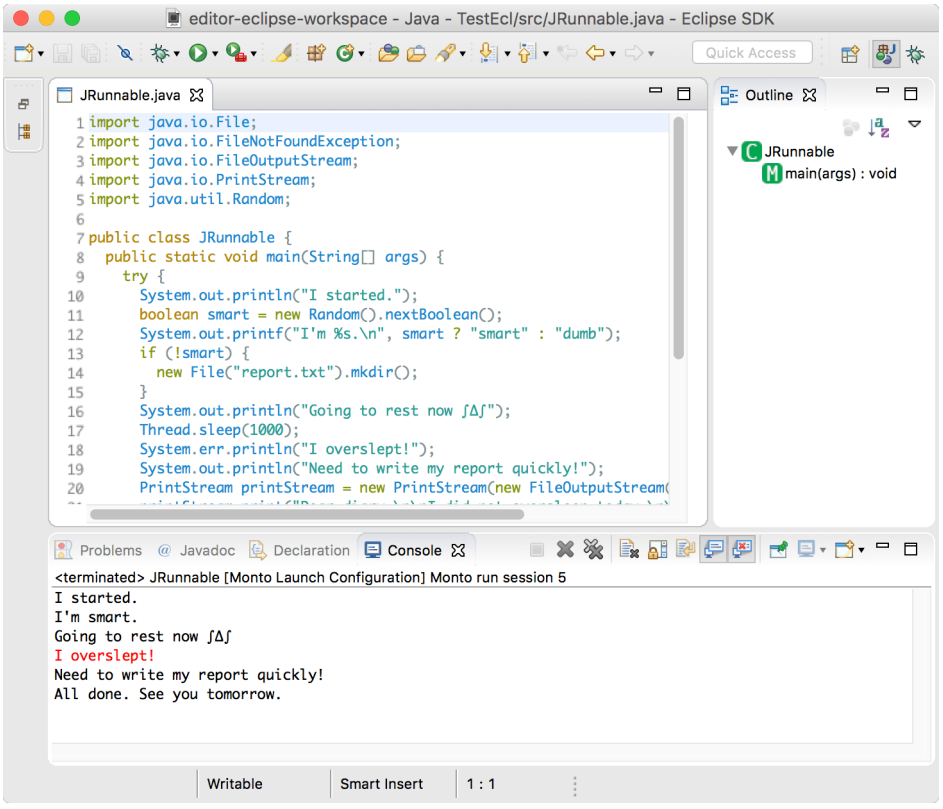


Figure 8: Screenshot of the stream output of a Monto "`run`" session

4 Debugger

Debugging code is a valuable tool available to developers when the execution of a program needs to be investigated. A reason for this could be a malfunction in the program that cannot be explained by just looking at the code. Debugging, as the name suggest, serves the purpose to fix bugs. IDEs greatly simplify the user interaction necessary the start a debug session. Debuggers halt the execution on user defined positions in the code, so-called breakpoints. Once a breakpoint is hit, the call stack, a structure containing the functions that led to the current point in execution, is visualized in the IDE. Currently relevant variable names and their values are also displayed. The future execution can be controlled in different step ranges, to jump to a specific area of interest.

Providing debugger support in Monto would make a vital addition to the framework. This section will aim to provide an IDE- and language-independent integration for debuggers in Monto. The previous section handled the design and implementation of executing programs in Monto. This will serve as the basis for the "Debug" feature because most of the products and commands can be reused in this section. New IRs for debugging have to be designed and tested by implementing a new language service. Integration into an IDE will also be covered using the example of the Monto Eclipse editor.

4.1 Feasibility analysis of a generic debugger interface

As a first preparation step three popular languages and their debugger interfaces were analyzed for similarity to see if debuggers can be reduced to a common denominator. But instead of directly diving into the raw debugger interface, easier to use command line debugger tools were used, because the features of the command line tools are usually a subset of the full debugger interfaces. So if similarities between command line tools can be found, the corresponding full debugger interfaces are also at least as similar. Java, Python, and C/C++ and their command line debugger tools `jdb` (Java debug bridge), `pdb`, (Python debug bridge) and `gdb` (GNU debug bridge) are used for this analysis.

Comparable debugger commands of these tools are listed in Table 2. Similar commands could be found to:

1. set, list and delete breakpoints.
2. move through call stack and inspect their content.
3. show variable names and values.
4. manipulate execution once breakpoints were hit.
5. list available function or methods.
6. evaluate expression when a breakpoint was hit.
7. stop the debugging.

Some disparities can be noticed when setting and removing breakpoint. `jdb` only references qualified class names, while `pdb` and most of `gdb` uses filenames. The new, more granular type of `Source` with a physical and logical name (see Section 2.4), should allow abstraction over this issue. These commands will be considered when designing debugger `CommandMessages` and their IRs.

4.2 Eclipse's debug model

Before designing a debug model from the ground up, the debug model of the Eclipse platform was considered. Eclipse's developers faced the same problem when designing an interface to allow language developers to integrate their debuggers into Eclipse. A class diagram in Figure 9 shows interfaces of the Eclipse debug framework, their associations, and multiplicities. `ILaunch` and `IProcess` were already mentioned in Section **??**. `ILaunch` allows registration of multiple `IProcesses` and `IDebugTargets`. `IDebugTarget` poses as the topmost entity of an Eclipse debug session. It can reference multiple `IThreads`, depicting all running threads in a debug process. `IThreads` can contain multiple and `IStackFrames` and one `IBreakpoints`, that led to the thread's suspension. `IStackFrames` can again include multiple `IVariables`, of which each must reference one `IValue`. `IValues` can refer to multiple `IVariables`, representing contained fields (e.g. attributes of a Java class or elements of an array). All interfaces but `ILaunch`, `IProcess`, and `IBreakpoint` extend `IDebugElement`, whose main purpose is to reference one common plug-in identifier. More detailed information on this model can be found in the article "How to write an Eclipse debugger" [10] and the Eclipse documentation [8]. This model will also influence the design of the Monto debugger IR.

Action	jdb command	pdb command	gdb command
Set breakpoint	`stop at package.Class:22`	`break file.py:10`	`break file.c:10`
Remove breakpoint	`clear package.Class:22`	`clear file.py:10`	`clear file.c:10`
List breakpoints	`clear`	`break`	`info breakpoints`
Get current stack trace	`where`	`where`	`backtrace`
Move up one stack frame	`up`	`up`	`up`
Move down one stack frame	`down`	`down`	`down`
Get arguments	`locals`	`args`	`info args`
Get locals	`locals, fields package.Class`	`locals()`	`info locals`
Get type of variable	`eval obj.getClass()`	`whatis` *var_name*	`whatis` *var_name*
Step until different source line is reached	`step`	`step`	`step`
Step until current methods returns to its caller	`step up`	`return`	`finish`
Step until next source line in reached	`next`	`next`	`next`
Resume normal executing	`cont`	`cont`	`continue`
Get callable functions	`methods package.Class`	`dir()`, `dir(`*var_name*`)`	`info functions`
Evaluate given expression	`eval` *expr*	`print` *expr*	`print` *expr*
Stop debugging	`exit`	`quit`	`quit`

Table 2: Comparable actions of command line debuggers `jdb`, `pdb` and `gdb`

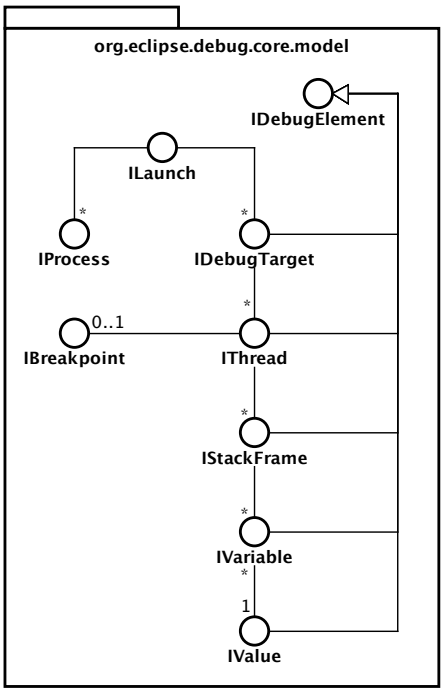


Figure 9: UML class diagram of Eclipse's debug framework interfaces (simplified) [10]

4.3 Debugger IR

Using the insights gained from the last two sub sections, Monto's debugger IR was designed and is documented now.

`Debugger` services are interactive, meaning they are triggered by `CommandMessages`. The beginning of a debugger session is marked by a `"debug"` command with a contents JSON object in form of:

```
DebugLaunchConfiguration ::= { main_class_source: Source,
                               breakpoints: Breakpoint* }

Breakpoint ::= { source: Source,
                 line_number: Int }
```

`breakpoints` contains breakpoints already set by the user before the debug session was even started. `Breakpoint` represents a line based breakpoint in a Monto `Source`. A `Debugger` services is expected to start debugging once a `"debug"` command is received. The entry point to the application is provided by `main_class_source`. The given breakpoints should be installed, so that the execution halts when they are hit. Outputs to `stdout` and `stderr` are send by `"streamOutput"` products, just as in Section 3.2.

Breakpoints can also be added after a debug session is started with a `"addBreakpoint"` command, which contains a `Breakpoint` JSON object in the `contents` field. Removing breakpoint from a running debug session should also be possible. This is requested by a `"removeBreakpoint"` command again containing a `Breakpoint` as content.

Once a breakpoint was hit, a `ProductMessage` with `product` `"hitBreakpoint"` and the following content is expected to be send out:

```
HitBreakpoint ::= { hit_thread: Thread,
                    other_threads: Thread* }
```

`Threads` are the topmost elements in the chosen debug model defined as:

```
Thread ::= { id: Long,
             name: String,
             stack_frames: StackFrame*,
             suspending_breakpoint?: Breakpoint }

StackFrame ::= { source: Source,
                 line_number: Int,
                 region: Region,
                 variables: Variable* }

Variable ::= { name: String,
               type: String,
               value: String }
```

All `StackFrames` and `Variables` are expected to be extracted, when a breakpoint was hit. No commands like the one in Table 2 are used to inspect `StackFrames`. After a breakpoint is hit, stepping should be available to navigate through the code. A `"debugStep"` command triggers one step and is expected to contain this data structure as content:

```
StepRequest ::= { thread_id: Long,
                  step_range: String }
```

A step command always refers to one thread that should be stepped with `thread_id`. They are always executed in the most recent stackframe. `step_range` can assume these value:

- `"INTO"` (step until different line number is reached)
- `"OVER"` (step until next source line number is reached)
- `"OUT"` (step out of the current function/method to its caller)

When the step was executed, a "threadStepped" product is supposed to be sent out with an updated Thread element and as content. Stepping only operates on one Thread, so there is no need to include the other Threads again, like in the "breakpointHit" product.

Completely resuming normal execution after a breakpoint was hit should also be possible. This can be triggered by a "debugResume" command with no contents and is acknowledged by the Debugger services with a "threadsResumed" product with no content. Termination of debug session is also possible. The command "terminate" and product "processTerminated" defined in Section 3.2 are reused for this purpose.

Commands were mostly based of Table 2, while the Thread model was taken and modified from Eclipse's architecture. Monto's debug model differs from Eclipse's in the following points:

- No cyclic references (e.g. Eclipse IStackFrame references IThread, Monto StackFrame doesn't reference Thread)
- No separate Value object (the value of a Variable was integrated into the Variable itself)
- IValues possibility to references contained fields isn't part of the Monto IR.
- Eclipse-specific classes (IDebugTarget and super class IDebugElement) were removed because other IDEs won't have these classes

This intermediate representation was used to implement a Debugger service for the Java language and to integrate it into the Eclipse Monto editor. The next two sub sections describe and document the development process.

4.4 JavaDebugger

When writing a debugger language service, it is preferable to use a dedicated debugger API instead of the command line tools presented in Section 4.1. The Java Debugger Interface (JDI)[7] is such an API for the Java language. Python's debugger framework bdb[8] accomishes the same tasks in Python. For C/C++ gdb/mi[9] (gdb machine interface) is available, but not as convienient as JDI or pdb because it is a line-based text interface, which involves printing to and parsing from text streams. This subsection documents the development of the JavaDebugger service using JDI, the so far only implemented debugger service on Monto.

Analog to the JavaRunner service, JavaDebugger is a completely interactive service controlled by CommandMessages. A new debug session is started by a "debug" command (Example JSON in Listing 17). Also analog to the JavaRunner the source code of main_class_source has to be requested with a dependency before compiling. To be able to access all features of JDI, the "-d" parameter has to be passed to JavaCompiler, so that all debug symbols get added to the bytecode. If this parameter is missing, local variables will not be visible when inspecting stack frames. With the bytecode generated and written to disk, a new debug session can be started. JDI offers several different connectors[10], that allow connecting to running Java virtual machines to debug them. The default connector is com.sun.jdi.CommandLineLaunch, which not only connects to a virtual machine, but also takes care of starting it (it is a LaunchingConnector[11]). This connector was use in the JavaDebuggerService. A Map with launching parameters is passed to the launch(Map<...>):VirtualMachine method, which in the JavaDebugger includes the bytecode-containing directory and the main class (again as a fully qualified class name, provided by the logical_name field in Source). The returned VirtualMachine instance can then be launched. It provides a Process instance via VirtualMachine.process(), whose InputStreams are read with two InputStreamProductThreads and whose termination is monitored with a ProcessTerminationThread to send "streamOutput" and "processTerminated" products, just like in the JavaRunner. Notification on debug events can be requested in the EventRequestManager accessible via VirtualMachine.eventRequestManager(). Events are delivied via a EventQueue provided by VirtualMachine.eventQueue() which relies on blocking calls. EventQueueReaderThread makes the EventQueue make more conveniently accessible using the observer pattern. Most of these objects are saved to a JavaDebugSession instance and added to the debugSessionMap:Map<Integer, JavaDebugSession> of JavaDebugger, so that later CommandMessages can interact with this debug session.

Breakpoints can be added in two ways: Either at the start of a debug session as part of DebugLaunchConfiguration JSON object or after the start of a debug session using the "addBreakpoint" command (Example JSON in Listing 18). Independent of how the breakpoint was added, attaching it to JDI is handled in JavaDebugSession.addBreakpoint(Breakpoint):vo
JDI requires a Location[12], describing the breakpoint's position. Locations can be retrieved from a ReferenceType[13] using

[7] https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/
[8] https://docs.python.org/2/library/bdb.html
[9] https://sourceware.org/gdb/onlinedocs/gdb/GDB_002fMI.html#GDB_002fMI
[10] http://docs.oracle.com/javase/8/docs/technotes/guides/jpda/conninv.html
[11] https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/connect/LaunchingConnector.html
[12] https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/Location.html
[13] https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/ReferenceType.html

locationsOfLine(int):List<Location>. ReferenceTypes are retrievable from VirtualMachine.classesByName(String):List< using the fully qualified class name, but only in the class in question was already loaded by a ClassLoader[14]. If a class was already used during the runtime of a Java application, it was also loaded by a ClassLoader, is which case the breakpoint can be installed correctly. If not, the installation of the breakpoint has to be delayed until the class get loaded. deferredBreakpoints:List<Breakpoint> holds all these delayed breakpoint in a JavaDebugSession. Luckily, ClassPrepareEvents[15] can be requested using EventRequestManager. These get fired when the class is about to be loaded by a ClassLoader, at which point a ReferenceType to it exists and breakpoints can be installed.

Once a breakpoint was hit, a BreakpointEvent[16] gets triggered (in EventQueueReaderThread using an EventQueue). In this event JDI provides a reference to the thread (as a ThreadReference[17]), in which the breakpoint was hit. A hierarchy similar to Eclipse's and Monto's debug model can be retrieved from ThreadReference and is converted to the Monto Thread IR in JavaDebugSession.convertJdiThreadTreeToMontoThreadTree(ThreadReference, Breakpoint):Thread. Additional ThreadReferences, retrieved from VirtualMachine.allThreads(), are also converted and packages together to the "breakpointHit" ProductMessage (Example JSON in Listing 20). Removing breakpoints when a "removeBreakpoint" command is received (Example JSON in Listing 19), is handled in JavaDebugSession.removeBreakpoint(Breakpoint):void.

After a breakpoint is hit, stepping can be initiated with a "debugStep" command (Example JSON in Listing 21). JDI allows stepping via the method EventRequestManager.createStepRequest() and triggers StepEvents[18], when the step finished. When this event was triggered, a "threadStepped" product is sent out with the updated Thread element and its children as content (Example JSON in Listing 22). The "debugResume" command (Example JSON in Listing 23) signalizes that the execution should no longer be paused and resumed normally. This is handled in JavaDebugSession.resume():void and acknowledged with a "threadsResumed" product (Example JSON in Listing 24). Termination of debug session is possible with the "terminate" command, which results in the interruption of the ProcessTerminationThread, the same as in the JavaRunner.

Listing 17: "debug" CommandMessage that starts a debug session

```
{
  "session": 1,
  "id": 1,
  "command": "debug",
  "language": "java",
  "contents": {
    "breakpoints": [
      {
        "line_number": 18,
        "source": {
          "physical_name": "TestEcl/src/JRunnable.java",
          "logical_name": "JRunnable"
        }
      }
    ],
    "main_class_source": {
      "physical_name": "TestEcl/src/StepTest.java",
      "logical_name": "StepTest"
    }
  },
  "requirements": []
}
```

Listing 18: "addBreakpoint" CommandMessage

```
{
  "session": 1,
  "id": 0,
  "command": "addBreakpoint",
  "language": "java",
  "contents": {
    "line_number": 14,
    "source": {
      "physical_name": "TestEcl/src/JRunnable.java",
      "logical_name": "JRunnable"
    }
  },
  "requirements": []
}
```

[14] https://docs.oracle.com/javase/8/docs/api/java/lang/ClassLoader.html
[15] https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/event/ClassPrepareEvent.html
[16] https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/event/BreakpointEvent.html
[17] https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/ThreadReference.html
[18] https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/event/StepEvent.html

Listing 19: "removeBreakpoint" CommandMessage

```
{
  "session": 1,
  "id": 0,
  "command": "removeBreakpoint",
  "language": "java",
  "contents": {
    "line_number": 14,
    "source": {
      "physical_name": "TestEcl/src/JRunnable.java",
      "logical_name": "JRunnable"
    }
  },
  "requirements": []
}
```

Listing 20: "hitBreakpoint" ProductMessage (only one Thread object is shown)

```
{
  "id": -1,
  "language": "java",
  "product": "hitBreakpoint",
  "service_id": "javaDebugger",
  "source": {
    "logical_name": null,
    "physical_name": "session:1"
  },
  "contents": {
    "hit_thread": {
      "id": 1,
      "name": "main",
      "stack_frames": [
        {
          "line_number": 14,
          "region": {
            "length": 14,
            "offset": 293
          },
          "source": {
            "logical_name": "StepTest",
            "physical_name": "TestEcl/src/StepTest.java"
          },
          "variables": [
            {
              "name": "power",
              "type": "int",
              "value": "102"
            },
            {
              "name": "message",
              "type": "java.lang.String",
              "value": "\"something\""
            },
            {
              "name": "threshold",
              "type": "float",
              "value": "4.0"
            }
          ]
        },
        {
          "line_number": 8,
          "region": {
            "length": 46,
            "offset": 149
          },
          "source": {
            "logical_name": "StepTest",
            "physical_name": "TestEcl/src/StepTest.java"
          },
          "variables": [

          ]
        },
        {
          "line_number": 3,
          "region": {
            "length": 30,
            "offset": 71
          },
          "source": {
            "logical_name": "StepTest",
            "physical_name": "TestEcl/src/StepTest.java"
```

```
          },
          "variables": [
            {
              "name": "args",
              "type": "java.lang.String[]",
              "value": "instance of java.lang.String[0] (id=104)"
            }
          ]
        }
      ],
      "suspending_breakpoint": {
        "line_number": 14,
        "source": {
          "logical_name": "StepTest",
          "physical_name": "TestEcl/src/StepTest.java"
        }
      }
    },
    "other_threads": [
      {
        "id": 92,
        "name": "Signal Dispatcher",
        "stack_frames": [],
        "suspending_breakpoint": null
      }
    ],
    ...
  }
}
```


Listing 21: "debugStep" CommandMessage


```
{
  "session": 7,
  "id": 0,
  "command": "debugStep",
  "language": "java",
  "contents": {
    "range": "OVER",
    "thread_id": 1
  },
  "requirements": []
}
```


Listing 22: "threadStepped" ProductMessage (StackFrames removed)


```
{
  "id": -1,
  "source": {
    "physical_name": "session:7",
    "logical_name": null
  },
  "service_id": "javaDebugger",
  "product": "threadStepped",
  "language": "java",
  "contents": {
    "id": 1,
    "name": "main",
    "stack_frames": [
      ...
    ],
    "suspending_breakpoint": null
  }
}
```


Listing 23: "debugResume" CommandMessage


```
{
  "session": 7,
  "id": 0,
  "command": "debugResume",
  "language": "java",
  "contents": null,
  "requirements": []
}
```


Listing 24: "threadsResumed" ProductMessage


```
{
  "id": -1,
```

```
  "source": {
    "physical_name": "session:6",
    "logical_name": null
  },
  "service_id": "javaDebugger",
  "product": "threadsResumed",
  "language": "java",
  "contents": null
}
```

4.5 Eclipse integration

Eclipse extensions written for the launching support in Section 3.4 are completely reusable when integrating Monto's debugging concept, but have to be extended. IMP's authors wrote this paragraph regarding debugger support in IMP:

> *As of this writing, IMP does not provide any framework components or meta-tooling for developing or interacting with language runtimes and debuggers. Although IMP's agnostic position with respect to language technology makes defining runtime and debug-time tooling more difficult (requiring a bridging layer of semantic descriptions), we believe it is still eminently possible. This is an open area for research and further development. [7]*

Just like when integrating launch support in the Eclipse Monto editor, Eclipse's extensions points have to be used, because IMP has no equivalent ones. Entry point of a debug session in Eclipse is still `launch(ILaunchConfiguration configuration, String mode, ILaunch launch, ...)` in `LaunchConfigurationDelegate`. It is called, when the user selected a previously created Monto `LauchConfiguration` and clicked the debug button in Eclipse's toolbar. But `mode` has now a value of `"debug"`. A `"debug"` `CommandMessage` referencing the main class and already set breakpoints is sent out (see IR in Section 4.4). A `MontoProcess` instance is created and add to the `ILaunch` argument, to show console output, just like before. But to provide Eclipse with a debug model, a `MontoDebugTarget` instance is created and also attached to the `ILaunch` argument. Section 4.5.1 contains more details on the debug model.

4.5.1 Debug model

Monto's debug IR (Section 4.3) now has to be integrated into Eclipse's debug model (Section 4.2). All interfaces shown in Figure 9 are implemented by Monto classes in package `monto.eclipse.launching.debug`. Allmost all logic is handeled in `MontoDebugTarget`, for example the translation between Monto IR and Eclipse model in method `convertMontoToEclipseThread(MontoDebugTarget, Thread):MontoThread`. `MontoDebugTarget` contains a list of `MontoThreads` which acts as the root for the debug model tree. This tree can be seen in the top left window in Figure 10. The line number of the currently selected stack frame is highlighted in the editor window and variables in the selected stack frame are show in the top right window.

`MontoThread` and `MontoStackFrame` contain several getter methods like `isTerminated():boolean`, `isStepping():boolean`, `canStepInto():boolean` or `canResume():boolean` to manage capabilaties and state. `MontoStackFrame` delegates all these methods to `MontoThread` and `MontoThread` delegates most of them to `MontoDebugTarget`. Eclipse uses them to enable or disable button in the user interface, but also shows different icon based on the returned state. User actions result in the invokation of method like `MontoStackFrame.stepInto():void`, `MontoThread.stepOver():void`, `MontoThread.terminate():void` or `MontoStackFrame.suspend()`. Again `MontoStackFrame` delegates to `MontoThread` and `MontoThread` to `MontoDebugElement`, where the actual `CommandMessages` are sent. More details on the Eclipse debug model integration are available in the article "How to write an Eclipse debugger" [10] and the Eclipse documentation [8].

4.5.2 Breakpoints

`IBreakpoint`[19] describes breakpoints in the Eclipse framework. `LineBreakpoint`[20] is an abstract class provided in the Eclipse Framework, that implements `IBreakpoint` and takes care of most platform related code when dealing with line-based breakpoints. Monto only supports line-based breakponts and there extends `LineBreakpoint` with the class `MontoLineBreakpoint`. Breakpoints are in Eclipse always attached to a `IResource`[21] object, in our case representing

[19] http://help.eclipse.org/neon/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/debug/core/model/IBreakpoint.html

[20] http://help.eclipse.org/neon/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/debug/core/model/LineBreakpoint.html

[21] http://help.eclipse.org/neon/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/core/resources/IResource.html

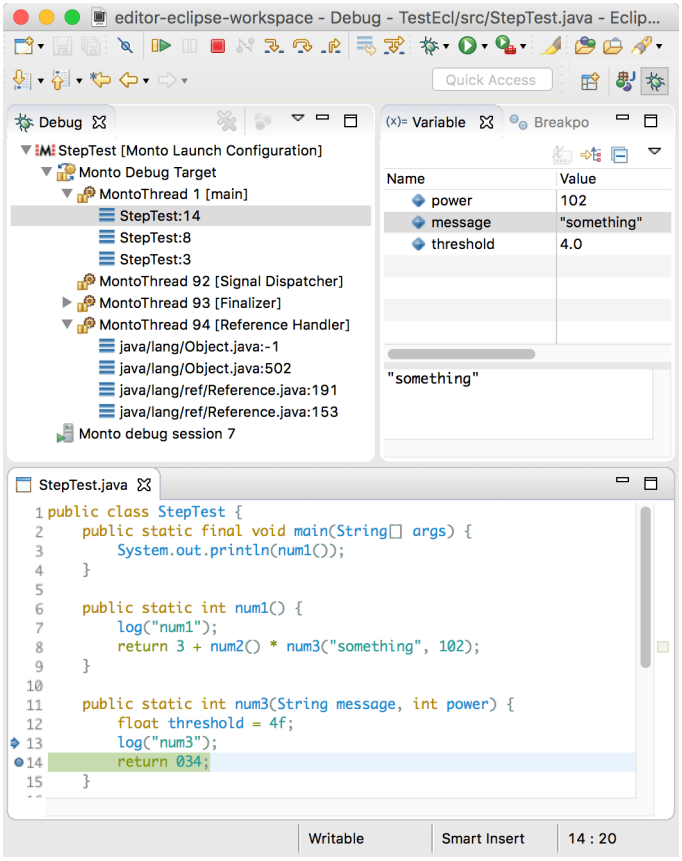


Figure 10: Screenshot of the Eclipse Monto editor, that shows a thread tree with stack frames

a workspace file and also saved between Eclipse sessions. A `IMarker`[22] belonging to the `IResource` object is a data structure, in which key-value pairs can be presistent. Properties of a Monto `Breakpoint` are persisted in a `IMarker`, so that they can be restored, even if the Monto Eclipse editor restarts.

`MontoLineBreakpoint` is registered as a new type a breakpoint in the `plugin.xml` file, but Eclipse requires another extension point, so that the registered breakpoint can be created from a editor window by double-clicking the line number. A `IToggleBreakpointsTargetFactory`[23] instance has to be used to create `IToggleBreakpointsTargets`[24]. The factory can decide, which type of breakpoint is used based on which editor window requests toggling of breakpoint. In Monto only one type of breakpoint is used, so `MontoToggleBreakpointsTargetFactory` always creates instances of `MontoToggleBreakpointsTarget`. The `IToggleBreakpointsTarget` declares its capabilities and decides in method `toggleLineBreakpoints(ISelection):void` if a new breakpoint should be created for the given `ISelection`[25] or if an existing breakpoint should be deleted.

4.5.3 Source code lookup

A important feature of every debugger is to jump to and hightlight to currently executed line. This was achieved in the Monto Eclipse edtior by implementing and registering two interfaces: `ISourceLocator`[26] and `IDebugModelPresentation`[27]. `ISourceLocator` translates `MontoStackFrames` to `IFile`[28] objects, that exists in the

[22] http://help.eclipse.org/neon/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/core/resources/IMarker.html

[23] http://help.eclipse.org/neon/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/debug/ui/actions/IToggleBreakpointsTargetFactory.html

[24] http://help.eclipse.org/neon/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/debug/ui/actions/IToggleBreakpointsTarget.html

[25] http://help.eclipse.org/neon/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/jface/viewers/ISelection.html

[26] http://help.eclipse.org/neon/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/debug/core/model/ISourceLocator.html

[27] http://help.eclipse.org/neon/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/debug/ui/IDebugModelPresentation.html

[28] http://help.eclipse.org/neon/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/core/resources/IFile.html

Eclipse workspace. This is easily achieved, because `StackFrames` reference the `Source` they are in and therefore also the `physical_name` of the source file. `IDebugModelPresentation` takes that `IFile` or a `MontoLineBreakpoint` and points to the editor, that should open these and creates the `IEditorInputs` for them. Both elements participated in the creation of the screenshot in Figure 10.

4.6 Summary

By inspecting command line debugger tools and the Eclipse debug framework, common actions and structures of the debugging process were identified and used to design a Monto debugger IR. The Runner IR presented in the previous section could be reused almost completely. To test the debugger IR, a debugger service for the Java language was implemented and integrated into the Eclipse Monto editor. The Monto framework is now able to debug applications, set and hit breakpoint, stepping through code and inspecting stack frames. This concludes the contributions provided by this thesis. The next section will conclude this work and provide some aspects that could be implemented in the future.

5 Related work

1. Emacs Grand Unified Debugger https://www.gnu.org/software/emacs/manual/html_node/emacs/Debuggers.html
2. Eclipse Che http://www.eclipse.org/che/
3. Truffle WebDebugger https://vimeo.com/161942246

6 Future work

- Web editor CommandMessages and debugger support
- More debuggers in other languages
- Project dependencies
- Move source of truth for `Sources` to broker
  - Currently `Sources` are send, when IDE opens file
  - Send on IDE start all `Sources` to Broker
  - Makes resolution of dynamic dependencies more robust, because dependencies get fulfilled right away / is no longer dependent on order of opening files
- Compiler services
- Extend debugger features
  - Introduce capabilities of debuggers (similar to configuration options), so that IDE can disable UI elements, that aren't support by a debugger service of a particular language
  - Specify thread suspend and resume policy. Right now: Suspend and resume all
  - Support suspend during debug session without breakpoints
  - Variables should be able to contain other variables to support fields of objects or elements of array
  - Display threads, even when program is not resumed

7 Conclusion

This thesis aimed to integrate interactive services into Monto. As two examples of interactive services, code completion and a debugger should be implemented. Before interactive services could be approached, the foundations in the framework needed to be constructed.

The example of a code completion service was used to explain which kind of services were already possible and the missing medium for stateful services was pointed out. A solution in the form of `CommandMessages` was introduced, but it needed to be extended by dependencies so that they could take advantage of already existing services. Now, with no more obstacles in the way, the code completion intermediate representation was designed. Services that complete Java, Python, and JavaScript code were implemented using this IR and IDE support for Monto code completions was integrated into the Eclipse editor.

The next step was to tackle the debugger, but it turned out to be more sensible to realize a code execution service first. By implementing the code completioner, all the framework tools necessary to implement an execution service were already available. A service that runs Java code used a newly designed execution IR to integrate the first execution service into Monto. IDE support for code execution was also added to the Eclipse editor.

The debugger service could reuse most of the IR design for the execution service. However, breakpoint management, stepping instructions, and a debug model needed to be added, so that debuggers can operate properly in the Monto framework. Command-line debugger tools of common programming languages were analyzed for similarities to find this IR. Together with the debug model of the Eclipse framework, they inspired the debugger IR. The Monto Eclipse editor was extended to support the Monto debugger model and a service that debugs Java applications was implemented.

The Monto project is still in its early stages, but the possibility to integrate interactive services helped the platform to be even more diverse.

References

[1] S. Keidel, W. Pfeiffer, and S. Erdweg, "The IDE Portability Problem and Its Solution in Monto," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2016. ACM, 2016, pp. 152–162.

[2] S. Keidel, "A disintegrated development environment," Master's thesis, Technische Universität Darmstadt, 2015. [Online]. Available: http://erdweg.org/teaching/thesis-keidel.pdf

[3] A. M. Sloane, M. Roberts, S. Buckley, and S. Muscat, *Monto: A Disintegrated Development Environment*. Springer International Publishing, 2014, pp. 211–220.

[4] W. Pfeiffer, "A web-based code editor using the Monto framework," Bachelor's thesis, Technische Universität Darmstadt, 2015. [Online]. Available: http://erdweg.org/teaching/thesis-pfeiffer.pdf

[5] S. Kockmann, "File dependencies in a disintegrated development environment," Bachelor's thesis, Technische Universität Darmstadt, 2016. [Online]. Available: http://erdweg.org/teaching/thesis-kockmann.pdf

[6] P. Charles, R. M. Fuhrer, and S. M. Sutton, Jr., "IMP: A Meta-Tooling Platform for Creating Language-Specific IDEs in Eclipse," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. ACM, 2007, pp. 485–488.

[7] P. Charles, R. M. Fuhrer, S. M. Sutton, Jr., E. Duesterwald, and J. Vinju, "Accelerating the Creation of Customized, Language-Specific IDEs in Eclipse," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09. ACM, 2009, pp. 191–206.

[8] Eclipse documentation - Platform Plug-in Developer Guide. [Online]. Available: http://help.eclipse.org/neon/topic/org.eclipse.platform.doc.isv/guide/debug.htm?cp=2_0_17

[9] J. Szurszewski, "We Have Lift-off: The Launching Framework in Eclipse," 2003. [Online]. Available: https://eclipse.org/articles/Article-Launch-Framework/launch.html

[10] D. Wright and B. Freeman-Benson, "How to write an eclipse debugger," *Eclipse Corner, Fall*, 2004. [Online]. Available: https://eclipse.org/articles/Article-Debugger/how-to.html