

The IDE Portability Problem and Its Solution in Monto

Sven Keidel

TU Delft, Netherlands

Wulf Pfeiffer

TU Darmstadt, Germany

Sebastian Erdweg

TU Delft, Netherlands



Abstract

Modern Integrated Development Environments (IDEs) support multiple programming languages via plug-ins, but developing a high-quality language plug-in is a huge development effort and individual plug-ins are not reusable in other IDEs. We call this the *IDE portability problem*.

In this paper, we present a solution to the IDE portability problem based on a language-independent and IDE-independent intermediate representation (IR) for editor-service products. This IR enables IDE-independent language services to provide editor services for arbitrary IDEs, using language-independent IDE plug-ins. We combine the IR with a service-oriented architecture to facilitate the modular addition of language services, the decomposition of language services into smaller interdependent services, and the use of arbitrary implementation languages for services.

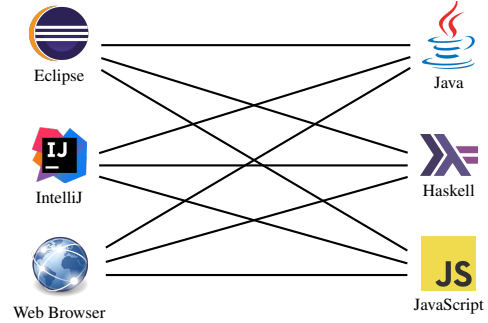
To evaluate the feasibility of our design, we have implemented the IR and architecture in a framework called Monto. We demonstrate the generality of our design by constructing language services for Java, JavaScript, Python, and Haskell and show that they are reusable in the Eclipse IDE and in a web-based IDE. We also evaluate the performance of Monto and show that Monto is responsive and has admissible performance overhead.

Categories and Subject Descriptors D.2.13 [Reusable Software]; D.2.6 [Programming Environments]: Integrated environments

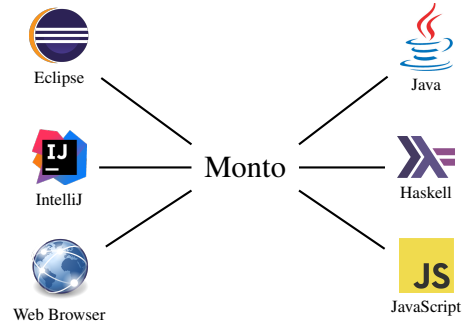
Keywords reusable software, integrated development environments

1. Introduction

IDEs aid software developers through tool support such as syntax highlighting, code completion, outline views, code refactorings, error reporting, and integrated debuggers. In-



(a) Current situation: For 3 IDEs and 3 languages, $3 \times 3 = 9$ language plug-ins have to be implemented.



(b) With Monto, the number of required implementations becomes linear: $3 + 3 = 6$.

Figure 1: The IDE portability problem.

stead of focusing on a single language, modern IDEs provide a plug-in mechanism that enables the integration of additional languages while reusing large parts of the IDE's infrastructure. Nevertheless, the development effort of a language plug-in is significant.

For example, consider the development effort¹ of language plug-ins^{2,3,4} for Scala in three different IDEs as of June 2016:

¹ Lines of code (LOC) measured with *cloc*.

² <https://github.com/scala-ide/scala-ide>

³ <https://github.com/JetBrains/intellij-scala>

⁴ <https://github.com/dcaoyuan/nbscala>

IDE	years	contributors	LOC Scala + Java
Eclipse	5	61	73k + 5k
IntelliJ	8	58	220k + 26k
NetBeans	3	11	23k + 122k

This table makes two points. First, developing a high-quality language plug-in is very costly. Second, each IDE requires a separate language plug-in; even for a single language there is no code sharing between plug-ins of different IDEs. More generally, when m IDEs support n programming languages, $m \times n$ separate language plug-ins have to be implemented, each with multiple years of development effort. We depict this situation in Figure 1a.

In the late 1960s, software developers faced a similar problem: How to compile high-level programming languages to assembly of different CPU architectures? Later, Richards referred to this problem as the *portability problem* [9]. For m target architectures and n languages, the traditional solution was to develop $m \times n$ separate compilers. Richards proposed a better solution where each language compiles to a machine-independent and language-independent intermediate representation (IR) and each architecture provides a translation from the IR to assembly. Modern compilers such as GCC follow this design and only require $m + n$ implementation artifacts. The challenge of this design is to provide a generic IR that is expressive enough such that all languages can be compiled to it, but that is also restrictive enough such that assembly can be generated for all target architectures.

In the context of IDEs, we call this problem the *IDE portability problem*: How to provide editor support for n programming languages in m different IDEs using only $m + n$ implementation artifacts. To solve the IDE portability problem, we propose an IDE-independent and language-independent IR called Monto IR for representing editor-service products (e.g., coloring information). For each language, we require a *language service* that takes source code and produces editor-support descriptions in the Monto IR. For example, the produced Monto IR can describe the syntax highlighting, error reporting, and code-completion proposals, but independent of the IDE. On the IDE side, we require a *Monto plug-in* that interprets the Monto IR and presents the editor support to the IDE user. Thus, as depicted in Figure 1b, for m IDEs and n languages, our design reduces the implementation effort from $m \times n$ language plug-ins to m Monto plug-ins and n language services. Moreover, our design effectively separates decisions about what to present to the user (language-specific) from how to present it (IDE-specific).

Modern IDEs not only provide editor support for languages, but also provide a platform for the integration of custom language services. But in our setting, an IDE only has a single plug-in that interprets the Monto IR. To retain the extensibility of existing IDEs, we propose a service-oriented architecture called Monto architecture for connecting language services and IDEs, where all messaging is done via the

network and all messages adhere to a simple protocol. This architecture enables (i) the modular addition of language services for new languages, (ii) the decomposition of a language service into multiple smaller services that can use each other’s results, and (iii) the implementation of services and IDEs in any implementation language. The central component of our architecture is the *message broker* that implements a service registry and routes messages between IDEs and language services.

In summary we make the following contributions:

- We identify the IDE portability problem and propose its solution based on a common IR.
- We present the Monto IR for describing editor support in an IDE-independent and language-independent style.
- We present the design of a service-oriented architecture for connecting language services and IDEs modularly.
- We evaluate the design of Monto by implementing Monto plug-ins for Eclipse and for a Web-Editor and language services for Java, JavaScript, Python, and Haskell with only 2 + 4 implementations.
- We show with measurements that the Monto architecture responds fast to user input, even if source documents are large.

2. Design Overview and Architecture

In this section, we first give an overview on the general workflow of Monto and then describe the major components of the architecture. Figure 2 illustrates the workflow of Monto through an example featuring two Java services. The parser service takes source code as input and produces ASTs. The outline service takes source code and an AST as input and produces an outline of the code.

When a user changes a source file ①, the Monto plug-in within the IDE sends a source message ② to the message broker. Each source message contains a unique ID called version, the file name, the language, and source code of the source file. Editor products include the version ID of the source, such that we can easily discard outdated products.

The message broker is responsible for distributing source and product messages to services that take them as input. The broker distributes messages to services in parallel but respects dependencies between services. In our example, the broker first forwards the source message to the parse service ③. The Java parser extracts the source code from the source message and parses the code into an AST. The parser then encodes the AST as a product message according to the Monto IR and sends the message back to the broker ④.

Now the broker has all information required to trigger the outline service and forwards the source and AST message to the outline service ⑤. The outline service computes an outline, encodes it according to the Monto IR, and sends it back to the broker.

The broker forwards all product messages to the IDE as soon as they arrive ⑥. Inside the IDE, the Monto plug-in interprets the Monto IR of the received product messages and updates the editor view of the IDE and the user observes the result ⑦. This workflow is repeated on every change to the source code.

Monto IR Monto’s IR is a serializable representation of visible IDE artifacts, like an outline view. The Monto IR for outline views uses JavaScript Object Notation (JSON) to represents the hierarchical tree-like structure of an outline. The following code is the Monto IR for an outline of a Java class:

```
[ { "label": "de.tu.darmstadt.graphics",
  "link": { "offset": 8, "length": 24 },
  "icon": "http://localhost:8080/package.png" },
  { "label": "Point",
    "link": { "offset": 48, "length": 5 },
    "icon": "http://localhost:8080/class-public.png",
    "children": [
      { "label": "x : int",
        "link": { "offset": 69, "length": 1 },
        "icon": "http://localhost:8080/field-private.png" },
      ...
    ]
  }
]
```

Each outline item contains a descriptive label, a link to a position in the code, an icon that describes the type of element and a list of child nodes. The outline view that corresponds to the IR code above is shown in Figure 3a displayed in Eclipse and Figure 3b displayed in a web-based IDE. The IR has to be flexible enough to support object-oriented languages like Java and functional languages like Haskell (see figure 3c and 3d). Furthermore, the IR has to be limited, so that all IDEs can still display the information of the outline. For example, if the location of icons would have been encoded with paths instead of URLs, a Web-Editor would not be able to display the icons, because direct file access is forbidden in browsers.

Language services In Monto, language services replace the role of language plug-ins. A language service takes a source document or products as input and returns Monto IR as output. Services receive the input as a source message and send their output as a product message via the network. The network guarantees loose coupling between services, broker and IDE. In particular language services can be implemented in a language that is different from the implementation language of the IDE. This property is required so that language services work with every Monto plug-in.

Services can be decomposed into smaller services that focus on a single task. This makes services easier to understand and increases their reusability. A service that is often reused is the parsing service, because many other services need an abstract syntax tree (AST) to produce their result. Services that need products of other services as input define a dependency on these products that is registered in the broker. The broker then sends requirements of a service bundled together. This allows services to respond directly to each request of the broker, without the need to cache products of other services.

Note that we currently assume that services produce their product by considering a single source file, but certain types of services need to consider multiple source files. We discuss in section 7 how the design of Monto can be extended to allow services to process multiple source files.

Monto plug-in Monto requires one Monto plug-in per IDE. Its main role is to react on source code changes, sending document updates in form of source messages to the broker, interpreting Monto IR and updating the user interface by calling the API of the IDE. After the plug-in received a response from the services, it might update the highlighting of code, update the outline view, or indicate new errors. The Monto plug-in communicates over the network with the broker to guarantee low coupling to the services. We require that a Monto plug-in not contain any language-specific code, otherwise new language services cannot be added without changing the code of the plug-in.

There are differences in the implementation between Monto plug-ins for different IDEs, e.g. a plug-in for a Web-based IDE has to dynamically generate HTML, whereas a plug-in for Eclipse simply calls an API method. Nonetheless, the plug-in should preferably use functionality that is provided by the API of the IDE rather than implementing functionality from scratch, to achieve a consistent user experience.

3. Monto IR

To separate language services from IDEs, we need to find an IR that language services can use to instruct an IDE. Specifically, Monto IR must satisfy the following design goals to ensure that it indeed solves the IDE portability problem illustrated in Figure 1:

- Monto IR must be reusable across IDEs such that different IDEs can interpret it. Consequently, the design of Monto IR must be *IDE-independent* and cannot rely on specific visualization styles.
- Monto IR must be reusable across languages, such that different languages can target it. Consequently, the design of Monto IR must be *language-independent* and expressive enough to support a wide range of languages.
- Monto IR must be *platform-independent* because existing IDEs and languages use different implementation languages.
- Monto IR must provide *unique identifiers* such that IDE requests and service responses can be correctly linked.

Our concrete design for Monto IR follows the policy that the IR should be *low-level and explicit*. The idea is that IDEs solely focus on rendering editor information and should not do any sophisticated processing. All such processing should instead happen in language services. For example, instead of providing regular expressions for syntax coloring, Monto IR declares the color of each character explicitly, such that the

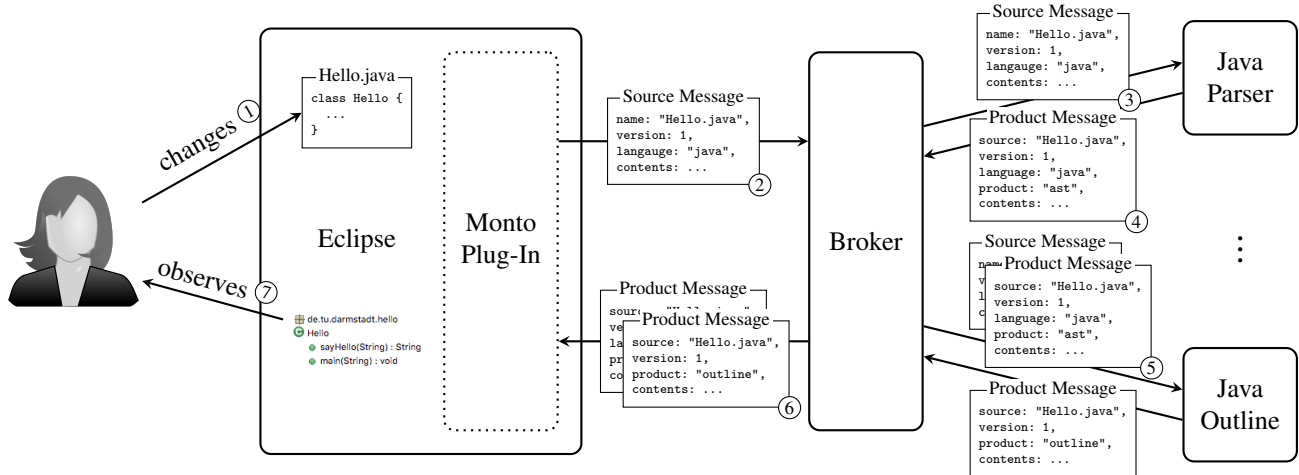


Figure 2: Communication between components in the Monto architecture.

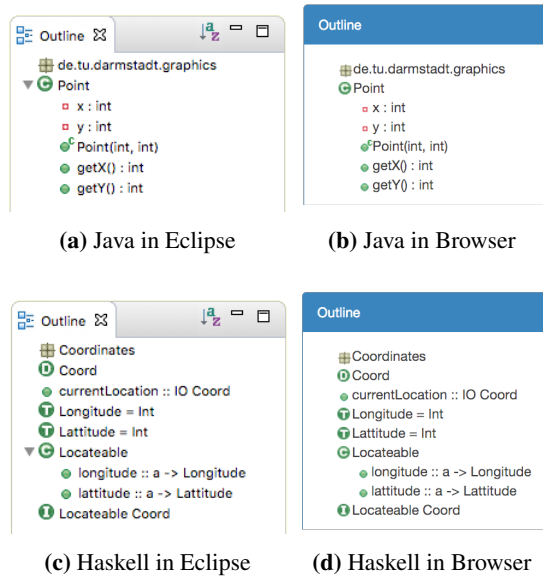


Figure 3: Outline views for different languages and IDEs.

IDE can overlay the coloring but does not have to process the source code. Using a low-level, explicit IR simplifies the IDE plug-ins and provides more flexibility to language services. We address platform-independence by encoding Monto IR messages in JSON.

In the remainder of this section, we describe the concrete design of Monto IR for source messages, syntax highlighting, ASTs, outline views, and error messages. As a running example, we use the following Java class:

```
public class Hello {
    String world = "World";
    public static void main(String[] args) {
        System.out.println("Hello " + new Hello().world);
    }
}
```

3.1 Source and Product Messages

An IDE sends a source message to the language services upon a change to the source code. The format of the source message is fairly straightforward:

```
Source ::= { name: String,
             version: Int,
             language: String,
             content: String }
```

That is, each source message contains an identifier `name` that uniquely identifies the source-code artifact, a unique ID `version` that identifies the current version of the source artifact, the language used in the source artifact `language`, and the content of the source artifact `content`. The Monto plug-in is responsible for using a fresh `version` number in every source message it sends. For our example Java class, the IDE sends the following message. To improve readability, we leave out quotation marks for JSON field names in this paper.

```
{ name: "~/project/src/Hello.java",
  version: 5, // running number
  language: "Java",
  content: "public class Hello {\n  String world = ... }"
```

Services send their result encoded in JSON as product messages back to the broker. Product messages have the following format:

```
Product ::= { name: String,
              version: Int,
              product: String,
              content: JSON }
```

The `name` and `version` field of a product message link it to a specific version of a source-code artifact the product was derived from. The field `product` describes the sort of product that is contained in this message and the `content` field contains the actual pay-load of the service.

3.2 Syntax Highlighting

Syntax highlighting selects different fonts for different parts of a source file. Typically, syntax highlighting associates

a font (family, color, weight, etc.) with specific syntactic or semantic classes of code elements featured in the edited language. The IDE then displays each code fragments in the selected font.

For Monto IR, we must abstract all language-specific and IDE-specific aspects of syntax highlighting. In particular, Monto IR cannot refer to language-specific classes of code elements. Instead, we opt for a low-level description of syntax highlighting using messages of the following format:⁵

```
Highlighting ::= Token*
Token ::= { offset: Int,
            length: Int,
            font: Font }
Font ::= { color?: Color, bgcolor?: Color,
           family?: String, size?: Int,
           style?: String, variant?: String,
           weight?: String }
Color ::= { red: Int, green: Int, blue: Int }
```

We describe syntax highlighting as a sequence of non-overlapping highlighted tokens. Each token is identified by its character offset in the source document and the number of characters it includes. We describe the font used for highlighting through attributes identical to those in CSS.

Not all language services require the full flexibility of our highlighting IR. However, services are free to confine themselves to use only a subset of the font attributes. Conversely, not all IDEs support the full flexibility of our highlighting IR. For example, many IDEs do not support multiple font families within a single text editor. However, IDEs are free to ignore those parts of the IR that they cannot render. Finally, the coloring of syntax sometimes is IDE-specific, for example, because the IDE uses a dark theme for its GUI components. As we describe in Section 4, language services in Monto can be configured by the IDE or its users.

For our example Java class, the Java service sends the following message:

```
[ // public class
  { offset: 0, length: 12, font: { /*bold blue*/ } },
  // Hello {\n String world =
  { offset: 13, length: 24, font: { /*black*/ } },
  // "World"
  { offset: 38, length: 7, font: { /*italic green*/ } },
  // ;
  { offset: 45, length: 1, font: { /*black*/ } },
  ... ]
```

3.3 Abstract Syntax Trees

An abstract syntax tree represents the syntactic structure of a program. While an AST is not directly useful for visualization in an IDE, it serves as an intermediate result that can be shared by services: A parser service can produce the AST and other services can require the AST as input. To support this scenario, we designed the following Monto IR for representing ASTs:

```
AST ::= { name: String,
          offset: Int,
          length: Int,
          children: AST* }
```

⁵ We write A^* to denote a JSON array of A elements and $k?: v$ to denote an optional field k in the surrounding JSON object.

Our AST representation mostly corresponds to s-expressions: AST nodes feature a name and a list of children. In addition, our AST nodes provide a pointer into the source document, encoded through offset and length. This way, other services can extract the original text from the source document when needed. For our example Java class, the Java parser sends the following message:

```
{ name: "CompilationUnit", offset: 0, length: 150,
  children: [
    { name: "ClassDeclaration", offset: 0, length: 149,
      children: [
        { name: "Modifiers", offset: 0, length: 6,
          children: [{name: "public", ...}] },
        { name: "Identifier", offset: 13, length: 5,
          children: [] },
        { name: "FieldDeclaration", offset: 23, length: 23,
          children: [ ... ] },
        { name: "MethodDeclaration", offset: 49, length: 98,
          children: [ ... ] } ] } ] }
```

3.4 Outline Views

An outline view provides a structural overview of a source document, typically in a hierarchical form. For example, an outline for Java shows type declarations, fields, and methods. IDEs often display a representative icon in front of each outline item to illustrate the kind of the item. Moreover, when the user clicks on an outline item, IDEs often move the cursor to the definition site of the item in the source document.

To support arbitrary languages, the Monto IR cannot make any assumption on the structure of the outline. For example, as was shown in Figure 3, the Monto IR must support languages that mostly consist of top-level declarations like Haskell as well as languages that use more deeply nested structures like Java. To this end, we designed the following IR for outline views:

```
Outline ::= Item*
Item ::= { label: String,
           icon?: URL,
           children: Item*,
           offset: Int,
           length: Int }
```

The root of the outline IR is a list of outline items, such that multiple top-level declarations can be displayed. Each item defines a text label for display in the outline. The label is free-form and can as in Figure 3 contain typing information in addition to the name of the declaration. We declare the icon of an outline item through an optional HTTP URL. This way, an IDE can load and cache the icons for outline items. To represent nested structures, an outline item can also declare a list of subitems. Finally, each outline item provides a pointer into the source document, encoded through offset and length. For our example Java class, the Java outline service sends the following message:

3.5 Error Reporting

Many language services detect and report errors, warnings, or other information about a program. Example services that produce such messages include parsers, spell checkers, type checkers, static analyses, and unit testing. Besides providing

```
[ { label: "Hello",
  icon: "http://localhost:8080/class-public.png",
  offset: 13,
  length: 5,
  children:
    [ { label: "world : String",
      icon: "http://localhost:8080/field-default.png",
      offset: 30,
      length: 5 },
      { label: "main(args) : void",
        icon: "http://localhost:8080/method-public.png",
        offset: 68,
        length: 4 } ] } ] ]
```

a list of such messages, IDEs typically highlight the location of the messages directly in the source code to provide visual feedback to the user. Moreover, IDEs typically display a detailed description of the message when the user hovers with the mouse over a highlighted location in the code. To represent error reports, we designed the following IR:

```
Report ::= Message*
Message ::= { offset: Int,
  length: Int,
  level: Level,
  category: String,
  description: String }
Level ::= "info" | "warning" | "error"
```

A report consists of a sequence of messages. Each message links to a region in the source code via offset and length and each message declares a severity level, which is either `info`, `warning`, or `error`. Finally, each message declares a message category (such as spelling, type, test, etc.) and each message contains a detailed description. The category enables users to filter for messages in the global message list within the IDE. For our example Java class, a service that applies a lint-style analysis could send the following report, where the first message marks an error and the second one marks a warning:

```
[ { offset: 0,
  length: 0,
  level: "error",
  category: "lint",
  description:
    "Class 'Hello' must not be in default package.",
  },
  { offset: 23,
    length: 12,
    level: "warning",
    category: "lint",
    description: "Field 'world' is public but non-final." } ]
```

3.6 Summary

The IR plays a central role in solving the IDE portability problem, where it serves as a standardized message format. We have presented standardized IR designs for different products: syntax highlighting, ASTs, outline views, and error reporting. In each design, we were careful to provide a generic representation such that the IR is reusable across IDEs and languages. So far, our work has focused on continuous editor services that run after every change to the source code. In Section 7, we discuss how our future work will add support for interactive editor services such as code completion or debugging.

```
DiscoveryResponse ::= ServiceDescription*
ServiceDescription ::= { service_id: String,
  label: String,
  description: String,
  options: Option* }
```

4. Service-Oriented Architecture

The service-oriented architecture of Monto enables (i) the modular addition of new language support, (ii) the decomposition of a language service into multiple smaller services that can use each other's results, and (iii) the implementation of services and IDEs in any implementation language. In this section, we explain why these properties are important for Monto and how Monto achieves them.

Modular addition of languages. Modern software projects often use multiple programming languages or domain-specific languages (DSLs) in the style of *language-oriented programming* [3, 13] to program different aspects such as the application logic, front-end, data persistence, or configuration and deployment. For this reason, existing IDEs support the modular addition of languages as plug-ins and we require the same flexibility for Monto. However, in comparison to traditional IDEs, a Monto language service simultaneously becomes available in all Monto-capable IDEs.

To support the modular addition of languages, Monto employs a service-oriented architecture where language services can be registered dynamically with the message broker (① in figure 4). To register a language service, the service sends an initial request of the following form to the message broker:

```
RegistrationRequest ::= { service_id: String,
  label: String,
  description: String,
  products: String*,
  dependencies: Dependency*,
  options?: Option* }
```

That is, a service identifies itself through a unique ID `service_id` and provides a user-readable label and description. It also declares the names of the products it produces, its dependencies on products of other services, and its configuration options. When the broker receives a registration message, it adds edges to a graph that represents the dependencies between services and products of other services. Moreover, the broker allocates a port for communication with the new service and informs the service:

```
RegistrationResp ::= RegistrationSucc | RegistrationFail
RegistrationSucc ::= { connect_to_port: Int }
RegistrationFail ::= { error: String }
```

After services successfully registered at the broker, the IDE can request information about running services by sending a discovery request ② to the broker. The broker responds with a list of descriptions of running services that contains an id, a label, a description text and a list of configuration options that can be displayed in a configuration menu to the user.

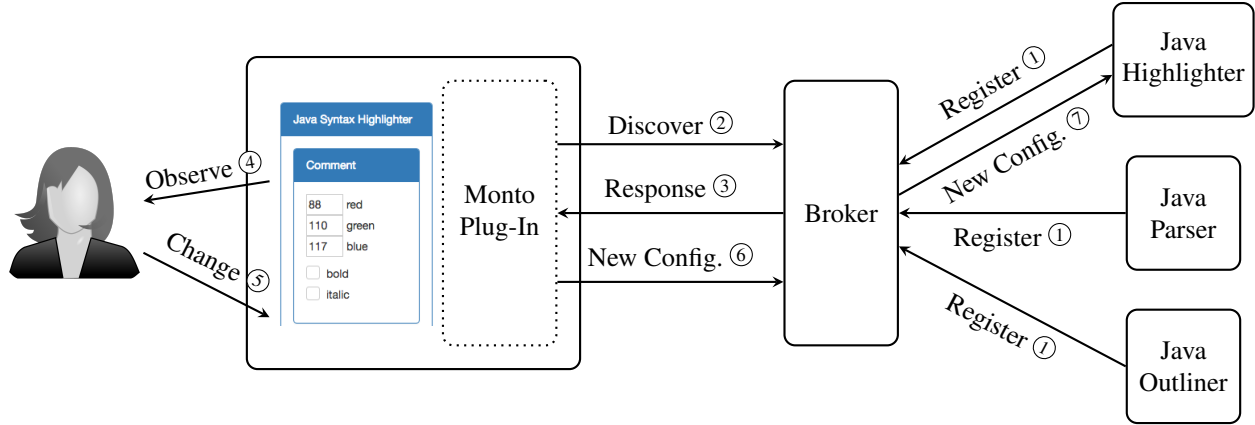


Figure 4: Service Discovery and Configuration in Monto.

Decomposition of services into smaller services. Monto services can provide multiple products. Thus, in principle, a single language service could implement all editor support for a language. However, there are often alternatives for individual aspects of a language. For example, there is a variety of type checkers (Flow, Strobe, Check) and static-analysis tools (JSLint, JSHint, ESLint) for JavaScript. To support their flexible replacement and composition, Monto supports the decomposition of a language into microservices. Specifically, multiple services can jointly implement the tooling for a single language, where different services produce different products.

A service can depend on source code or on the result of other services. Services declare their dependencies during registration:

```
Dependency ::= SourceDependency | ProductDependency
SourceDependency ::= { language: String }
ProductDependency ::= { service_id: String, product: String }
```

For example, the Java outline service depends on the source code and the AST. It sends the following registration request:

```
{ service_id: "javaOutliner",
  label: "Java Outline Service",
  description: "Produces an outline view for Java code",
  products: [ "outline" ],
  dependencies: [ { language: "java" },
    { service_id: "javaParser",
      product: "ast" } ] }
```

Different implementation languages. The implementation language of a language service is often different from the supported language and from the implementation language of the IDE. For example, Flow and Strobe are implemented in Ocaml, Check is implemented in JavaScript, and our Haskell language service uses the Haskell API of the Glasgow Haskell Compiler (GHC). The support for services and IDEs implemented in different languages is essential for facilitating the integration of new IDEs and language services and to avoid bias.

The service-oriented architecture of Monto integrates services and IDEs written in different languages through

the standardized Monto IR. We use JSON for the encoding of the IR because of the wide-spread availability of JSON libraries in different languages. For communication, Monto uses *ZeroMQ*, a messaging library that provides a minimal abstraction layer over raw network connections. *ZeroMQ* provides a small API that currently has bindings for 50 languages and thus does not restrict the implementation language of services or IDEs.

Service configuration. Services like syntax highlighting or spell checking need to be configurable to allow users to select a coloring scheme or a language. Like traditional IDEs, we would like to provide a graphical UI for users to adjust their configuration. Since a Monto IDE plug-in is unaware of the services it presents, we introduce a protocol for service configuration.

To display a configuration dialog, the IDE can request a list of running services from the message broker (2). The response (3) contains a list of available configuration options for each service:

```
Option ::= { type: "boolean", option_id: String,
  label: String, value: Bool }
| { type: "number", option_id: String,
  label: String, value: Int }
| { type: "text", option_id: String,
  label: String, value: String }
| { type: "select", option_id: String,
  label: String, value: String,
  values: String* }
| { type: "group", label: String,
  members: Option* }
```

Each configuration option has a unique ID, a user-readable label, and a default value. A service can declare options of different types, like boolean options which will be displayed as check boxes in the configuration dialog.

The Monto plug-in interprets the configuration description of the services and presents a configuration dialog to the user (4). The current configuration is stored and persisted on the IDE-side, to a per-IDE configuration. When the user changes the configuration (5), the IDE sends the new settings to the message broker (6) in the form of an associative array:


```
Configuration ::= { service_id: String, settings: Setting* }  
Setting ::= { option_id: String, value: String }
```

The broker forwards the configuration to the corresponding service ⑦, which validates the new configuration and changes its internal state. During discovery and configuration, plug-in and services communicate via the broker to reduce coupling.

5. Technical Realization and Case Studies

In the previous sections, we presented the design of the Monto IR and the Monto service-oriented architecture. In this section, we present two IDE bindings and four language services that we realized to evaluate the feasibility and generality of our design and to demonstrate that our design indeed enables language support across IDEs without code duplication. All of our source code is available online.⁶

As a basis for our evaluation, we developed an implementation of the message broker in Haskell. The broker implementation exactly follows the design outlined in Section 4, using *ZeroMQ* for message passing.

To demonstrate the generality of our design, we chose IDEs and languages of various character. We developed generic Monto plug-ins for the following IDEs:

- **Eclipse** is an IDE implemented in Java and supports a wide range of languages. We used Eclipse IMP [2] to build a generic Monto plugin-in that sends source code to the broker after any code change. The plug-in accepts product messages from the broker and can render all IRs from Section 3.
- **CodeMirror** is a text editor implemented in JavaScript and runs in the browser. As such, CodeMirror has limited file access and only supports *WebSockets*. We build a generic Monto plug-in that sends source code via *WebSockets* to a small proxy, which forwards the messages to the broker via *ZeroMQ*. The plug-in accepts product messages via the proxy and can render all IRs from Section 3. Since CodeMirror is just a text-editor pane, we added custom UI elements to render outline views.

The IDE bindings are language-independent and only depend on the Monto IR. That is, they can render source code of any language for which a language service exists. We developed language services for the following languages:

- **Java** is a statically typed, compiled, class-based object-oriented language. We provide separate services for syntax highlighting, parsing, outline viewing, and code completion (not described in this paper), where the latter two services both depend on the result of parsing.
- **JavaScript** is a dynamically typed, interpreted, prototype-based object-oriented language. We provide separate services for syntax highlighting, parsing, outline viewing, code completion (not described), type checking, and spell

checking. Both, the type and spell checker simply wrap existing tools Flow⁷ and Aspell to generate corresponding error reports in the Monto IR. The spell checker is configurable and allows users to select a language as well as to deactivate the checking of comments or string literals.

- **Python** is a dynamically typed, interpreted, multi-paradigm language. We provide separate services for syntax highlighting, parsing, outline viewing, and code completion (not described). We implemented the services for Java, JavaScript, and Python in Java, factoring out common code into a reusable library.
- **Haskell** is a statically typed, compiled, functional language. We provide a single service that supports syntax highlighting, parsing, outline viewing, and type checking. We implemented this service in Haskell by using the Haskell compiler GHC as a library and simply translating its outputs into the Monto IR.

Using Monto, our two IDE bindings can interoperate with any of the four supported languages despite the variety of supported languages and despite the implementation languages used for different components (Java for Eclipse; JavaScript for CodeMirror; Haskell for the broker; Java for Java, JavaScript, and Python; Haskell for Haskell).

6. Performance Evaluation

Monto's design requires language services to encode all information as JSON messages according to the Monto IR. Furthermore, Monto's service-oriented architecture requires network-based communication that is routed through the message broker. In this section, we empirically evaluate the impact of Monto's design on its performance. Specifically, what is the response time after a user edits a source file?

Representatively, we measured the response times of the Java tokenization, parsing, and outline services. To identify the overhead of Monto, we slightly modified the Java services to keep track of the inherent processing time used for tokenization, parsing, and outline extraction, but excluding the time used for translation into the IR, for serialization, and for deserialization.

To measure the response times of the Java services, we implemented a special Monto plug-in that runs stand-alone rather than within an IDE. The plug-in simulates user changes to the source file and sends a source message to the broker for each change, thus triggering the processing in a controlled fashion. After every sent source message, the plug-in waits and measures the time until it receives the service responses. We repeat the measurement 10 times for each source file and take their mean. After each measurement, we wait 100ms to allow the system to reach an idle state.

As Java corpus, we used the main source files from the *Apache Ant* project, version 1.9.7. The corpus contains 851

⁶ <https://github.com/monto-editor/>

⁷ <http://flowtype.org/>

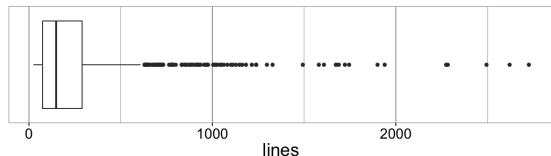


Figure 5: Distribution of *Apache Ant* file sizes.

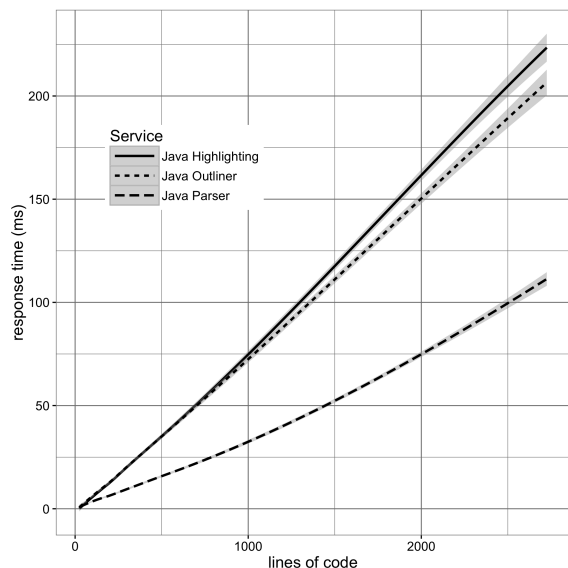


Figure 6: Response times of individual Java language services relative to the file size.

source files containing 222k lines of code. Individual files of the corpus contain between 24 and 2725 lines of code, with the median at 148 and the mean at 261. We show the distribution of file sizes as a box plot in Figure 5

Response time. Figure 6 shows the response times of the Java services for the source files of *Apache Ant*. We plot the response times for each service individually. The graph shows that for all files, the IDE receives the highlighting response first, followed by the parsing response (which is discarded by the IDE), and finally the outline response. While the outline service has to wait for the parsing service because it requires the AST, the parsing and highlighting services run in parallel. We consider a response to be complete when the IDE received the outlining response.

The graph shows that the response time grows linearly with the file size. For example, the system responds to files with ≤ 700 lines of code in under 50ms and to files with ≤ 1250 lines of code in under 100ms. To bring such numbers into context, Nielsen has shown that users do not perceive any notable delay if a system reacts in under 100ms [8]. For Monto, this means that there is no notable delay for files with ≤ 1250 lines of code, while larger files impose a notable delay that linearly grows with their size.

7. Discussion and Future Work

In this section, we discuss design decisions of the Monto architecture and possible extensions of the design.

Stateless services. One of the central design decisions of Monto was that language services should be stateless. The main benefit of this design is that it is easy to add new language services. For example, the outline service requires both the source code and the AST of the source code. Since the service is stateless, the broker is responsible to invoking the service with all required data as input. That is, the broker delays the invocation of the outline service until the AST is available, joins the source file and AST with corresponding versions, and submits them jointly to the outline service. A stateful service could have implemented the same logic, but we rather provide it once in the broker for all services. A disadvantage of stateless services is that services cannot support incremental processing of source-code changes, which requires state in general. However, as our performance evaluation showed, even non-incremental processing provides fast enough feedback. Nevertheless, for computationally involved services, we want to investigate how to support incrementality in future work.

Message Compression. File systems like ZFS and BTRFS use compression to increase the writing speed at the cost of higher CPU usage and slower reads. We could use a similar technique in Monto to reduce the size of source and product messages and the messaging overhead.

We evaluated this technique empirically by measuring the latency of sending messages uncompressed and the latency of compressing the message and sending the compressed message. As a compression algorithm we used LZ4, a fast compression algorithm that is also used by the file systems mentioned above. Our measurements showed that compression becomes viable only when source messages are larger than 30MB. However, source documents are typically much smaller. We conclude that compression is not worthwhile for Monto.

File Dependencies. The current Monto architecture assumes that a service can produce its result for a file without considering other files. In general, this assumption is untenable. For example, to type check a Java source file or to provide code completion, the service needs information about methods defined in other files that have been imported. Support for such file dependencies is work in progress; here we only outline our approach.

When a source file changes, the broker triggers relevant language services like it does now. The broker makes sure to delay language services that have a declared dependency on another service (e.g., outline on parsing), such that all relevant information is available when the service is called. For file dependencies, this is not possible because file dependencies are inherently dynamic and can only be detected when analyzing a concrete source files (e.g., by inspecting

the import statements). Thus, the broker cannot anticipate file dependencies. Instead, we allow language services to interrupt processing in order to request additional products from the broker. We plan to adopt an existing incremental algorithm [4] for reestablishing consistency in face of changes and dynamic dependencies.

Interactive Services. Monto as presented in this paper only triggers language services when a source file changes. While this is sufficient for a large set of editor services, it does not support services that react to user interactions other than file changes. For example, code completion and hover help are triggered by user interactions (key combination or mouse hovering) rather than by source-file changes. More sophisticated interactive services include read-eval-print loops and debuggers.

We believe that our design extends to interactive services, but we need to design new IR forms for communicating interactions. As in our current design, these interaction IRs must be IDE-independent and language-independent. This ensures that an IDE can submit interactions to the broker without knowledge of the language service, and a language service can interpret interactions independent of the IDE that submitted it. Similarly, we need to design new IR forms for presenting the results of interactive services, such as a list of code-completion proposals or a debugger state. In our implementation, we already support code completion, but the design of further interactions is future work.

8. Related Work

We discuss the relation of Monto to other systems that at least partially address the IDE portability problem.

Existing IDEs like Eclipse and IntelliJ provide plug-in mechanisms [1] for the modular extension with new features and programming languages. These plug-in mechanisms enable a single IDE to support multiple languages. However, as discussed before, plug-ins are closely coupled to a particular IDE and are not reusable in other IDEs. The development effort for Scala plug-ins in Eclipse, IntelliJ, and NetBeans that we showed in Section 1 confirms that the development of language plug-ins is costly and there is no code reuse across IDEs. Thus, existing IDEs do not solve the IDE portability problem.

Microsoft’s language server protocol⁸ (LSP) is a protocol between a client (like an IDE) and a language server. Compared to Monto, the client communicates directly with the language server and a central component like the Monto broker is missing. In contrast to Monto, the protocol is stateful and the language server has to maintain and update a copy of the document. A language server can signal the client if the whole document contents or incremental updates should be sent to the server. The main advantage of Monto compared to LSP is that language services can be split up into small

services that share results, whereas in LSP the sharing of results is not part of the protocol.

IDE-independent language processors like the *Scala presentation compiler* or the *Haskell ide-backend*⁹ provide APIs for retrieving editor-relevant information about source files. The API and the editor-service data structures are IDE-independent and can be reused in multiple IDEs. However, the APIs and data structures are language-specific. For example, the Scala presentation compiler and the Haskell ide-backend do not share a common interface or common IR. Consequently, a separate IDE plug-in is necessary for each language and there is little code reuse. Moreover, it is not clear how to apply these APIs in IDEs that use a different implementation language. Existing IDE-independent language processors do not solve the IDE portability problem.

Language workbenches are tools for the definition of new languages [5]. Given a (typically declarative) language definition, a language workbench derives a fully functional IDE plug-in for the language. Thus, language workbenches provide IDE support that is reusable across languages. However, language workbenches are coupled to a particular IDE. We discuss MPS, Spoofox, and Xtext in more detail below.

MPS [12] is a language workbench that generates projectional editors for language definitions. The language specification as well as the generated projectional editors are specific to MPS and cannot be used in other IDEs. In particular, MPS does not provide an IDE-independent IR and does not solve the IDE portability problem.

Spoofox [7] is a language workbench that generates editor support in Eclipse from declarative language definitions. While Spoofox is coupled to Eclipse and does not support other IDEs, it provides IDE-independent metalanguages for the definition of languages. Specifically, Spoofox provides metalanguages SDF3 for syntax, NaBL for name binding, TS for type checking, and DynSem for dynamic semantics [11]. It may be possible to derive IDE plug-ins for different IDEs from such language definitions. In contrast to using generic metalanguages, Monto relies on a generic IR for communicating concrete editor products in a language-independent and IDE-independent fashion.

Xtext [6] is a language workbench that generates editor support for Eclipse, IntelliJ, and three web-based editors. The generated editors for Eclipse and IntelliJ are regular plug-ins that are generated by two distinct generators. Neither the generators nor the plug-ins are reusable in other IDEs. To support web-based editors, Xtext provides a separate API based on HTTP requests that an editor can use to retrieve editor products. This API and the editor products are tailored for usage in web-based IDEs and do not appear to be reusable. Thus, Xtext does not solve the IDE portability problem.

While existing language workbenches are coupled to specific IDEs, it would probably be easy to adopt Monto for them: Instead of generating plug-ins directly, a language

⁸ <https://github.com/Microsoft/language-server-protocol>

⁹ <https://github.com/fpco/ide-backend/>

workbench could generate a language service that generates Monto IRs for source files. Our generic IDE plug-ins can then serve as a front-end to the language services derived from declarative language definitions.

Sloane et al. proposed what they called a *disintegrated development environment* [10]. While our work borrows some ideas from this Sloane et al.'s work, there are significant differences. Most importantly, the goal of the disintegrated development environment was to provide editor-independent tooling that derives additional information (i.e., separate text files) about a source file. In contrast, we are interested in integrated editor services that directly influence the behavior of the IDE. For example, the disintegrated development environment can show the AST of a file in separate buffer, but it cannot affect the highlighting of the source file itself. As consequence, Sloane et al. did not propose a generic IR, which is central to our approach. There are further technical differences. In particular, our message broker supports service dependencies and our services are stateless. As consequence, it is easier to add new language services in our design.

9. Conclusion

We defined the IDE portability problem, which says that the editor support for a language is currently tightly coupled to a single IDE. Contemporary systems require $m \times n$ implementation artifacts in order to support n languages in m different IDEs and there is almost no code reuse.

In this paper, we presented Monto, a solution to the IDE portability problem based on a language-independent and IDE-independent IR. Specifically, we standardized IR representations for syntax highlighting, ASTs, outline views, and error reporting, and we discussed how to support inter-active services. We presented a service-oriented architecture for Monto that enables the modular addition of language services, the decomposition of a service into smaller services, and the implementation of services and IDEs in arbitrary implementation languages.

We evaluated the feasibility and generality of Monto through case studies and showed that we can support four languages (Java, JavaScript, Python, Haskell) in two IDEs (Eclipse, Web browser) with only $2 + 4$ implementation artifacts. Monto enables the reuse of an IDE binding for all languages and the reuse of each language service for all IDEs. Finally, we empirically validated that the performance overhead induced by Monto is admissible and Monto-based IDEs provide rapid feedback to users.

Acknowledgments

This research was supported by DFG grant “Evolute” and Oracle Labs. We thank Anthony Sloane, Sylvia Grewe, Hans Becker, and Eduardo Souza who provided helpful feedback. Furthermore, we want thank Stefan Kockman for implementing the Python plugin for Monto.

References

- [1] Dorian Birsan. On plug-ins and extensible architectures. *ACM Queue*, 3(2):40–46, 2005.
- [2] Philippe Charles, Robert M Fuhrer, and Stanley M Sutton Jr. Imp: a meta-tooling platform for creating language-specific IDEs in eclipse. In *Proceedings of IEEE/ACM international conference on Automated software engineering*, pages 485–488, 2007.
- [3] Sergey Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains onBoard*, 1(2), 2004.
- [4] Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. A sound and optimal incremental build system with dynamic dependencies. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOP-SLA)*, pages 89–106. ACM, 2015.
- [5] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47, 2015.
- [6] Moritz Eysholdt and Heiko Behrens. Xtext: Implement your language faster than the quick and dirty way. In *Proceedings of ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309, 2010.
- [7] Lennart C. L. Kats and Eelco Visser. The Spoofox language workbench: Rules for declarative specification of languages and IDEs. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOP-SLA)*, pages 444–463. ACM, 2010.
- [8] Jakob Nielsen. *Usability engineering*. Elsevier, 1994.
- [9] Martin Richards. The portability of the bcpl compiler. *Software: Practice and Experience*, 1971.
- [10] Scott Buckley Tony Sloane, Matt Roberts and Shaun Muscat. Monto: A disintegrated development environment. *Software Language Engineering*, pages 211–220, 2014.
- [11] Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Neron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël Konat. A language designer’s workbench: A one-stop-shop for implementation and verification of language designs. In *Proceedings of International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (ONWARD)*, pages 95–111. ACM, 2014.
- [12] Markus Völter and Vaclav Pech. Language modularity with the MPS language workbench. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 1449–1450, 2012.
- [13] Martin P Ward. Language-oriented programming. *Software-Concepts and Tools*, 15(4):147–161, 1994.