File dependencies in a disintegrated development environment

Dateiabhängigkeiten in einer reintegrierten Entwicklungsumgebung
Bachelor-Thesis
Author: Stefan Kockmann

Day of submission: March 8th, 2016
Examiner: Prof. Dr.-Ing. Mira Mezini
Supervisors: Dr. rer. nat. Sebastian Erdweg
              M.Sc. Sven Keidel

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Department of Computer Science
Software Technology Group

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-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 8. März 2016

_____

(Stefan Kockmann)

Abstract

Among developers the utilization of Integrated Development Environments (IDE) is widely conducted to improve their work performance. IDEs offer a conglomerate of coding related tools and features to help the developer in his endeavour including source code highlighting, context dependent completion and compilers. Various IDEs exist offering different collections of features. This leads to situations where a developer is missing a feature like support of a programming language. A lack of feature can usually be encountered by implementing a plug-in to an Application Programming Interface (API). These plug-ins however are generally IDE dependent and can therefore not be used by other IDEs. Hence the quality of the same offered features is different between IDEs. This leads developers into using multiple IDEs or using a single IDE knowing to relinquish best possible support.

This deficiency is tackled by the framework Monto. It decouples IDEs in different accessable components. Features for example are encapsulated into services and can thereby be reused by different IDEs. These services are partitioned in a way that they might be dependent on other services. The contributions of this thesis are to enhance the existing services by implementing support for a new programming language, decrease the overhead between services and analyse the runtime response times of different implemented services.

Inhaltsverzeichnis

List of Figures

List of Tables

1. Introduction

To be more efficient and less error prone developers commonly use IDEs. These IDEs offer functionalities such as highlighting, completion, debugging and often compiling. The functionalities like highlighting and completion offer visual enhancement and faster development times and are part of an actual editor. Debugging and compiling though provide functionalities through toolchains that a user can utilize. The functionalities differ thus greatly in kind. An IDE is therefore a congestion of different functionalities. To enable community driven improvements IDEs often offer an API. Utilizing an API plug-ins can be developed. A common reason to implement a plug-in is to provide support for a programming language. As the number of different programming languages is vast and the number of different IDEs is increasing this leads to problems: new IDEs need to implement support for every programming language and the quality of a programming language support an IDE has to offer various hugely. To be more precise: assuming for every programming language support a plug-in would be needed, n programming languages exist and m IDEs exist. This leads to n * m plug-ins needed with great variety in quality of support. To address this *Sloane, et al.* [1] developed an architecture, that represents a decomposition of IDE functionalities. The IDE functionalities get encapsulated into services that can be accessed via a broker. Hence the complexity is reduced to n + m. This original version has been enhanced by *Keidel* in his master thesis [2]. The services are proportioned into components with single responsibilities which leads to potential dependencies to other services.

This thesis aims to enhance the services of the Monto framework further. Python Services are developed and presented in chapter 3. In addition for currently active base service types a hypothesis of their behaviour at runtime is proposed. To handle potential performance issues further more a Distributor is created that limits the overhead between services and is discussed in chapter 4. The hypothesis proposed in 3.1.6 is discussed and validated on the basis of a runtime test in chapter 5. In the same chapter a developed testing tool is presented. Chapter 6 focuses on related works to the content of this thesis, while in chapter 7 possible future works following up the outcome of this thesis are proposed. In chapter 8 the work and the results of this bachelor thesis are discussed.

2. Background

In this chapter the basics of the Monto framework modified and outlined by *Keidel* [3] are described. A brief overview of the architecture as well as the function of the components is given. For a more in-depth view on why the components have been partitioned in this way it is recommended to read "A disintegrated development environment" [1] and "Monto: A disintegrated development environment" [2], which is the original proposal of the framework. Furthermore basic technologies used are described and justified why they have been chosen in this bachelor thesis.

2.1 Architecture of Monto

Monto is a framework consisting of several useful components which can be used by an IDE through an implemented Monto plug-in like in [2] or through the implementation of a client like in [4]. Four of these components can be identified as main parts. They are namely broker, source, sink and services. While their functions are described more in detail in later sections of this chapter, this section will focus on how they interact with each other. Plug-in/client side sources send Source Messages to the broker. Content of these Source Messages is amongst other things source code. The broker then distributes the Source Message to services, which have solely a dependency to this source. Contrary to services, which have additional dependencies. Tokenizer(a sort of Service) for example has a sole dependency to Source Messages. The by the broker addressed services return Product Messages, which are again send to the broker. It then computes for which services the dependent messages are present and sends these messages to the service, which in return send Product Messages to the broker. This procedure is repeated until all dependency constellations, that could have been fulfilled, have actually been fulfilled. While the broker is distributing Product Messages to dependent Services, it is also distributing them to sinks of the plug-in/client. The described interaction can be seen in Figure 1.
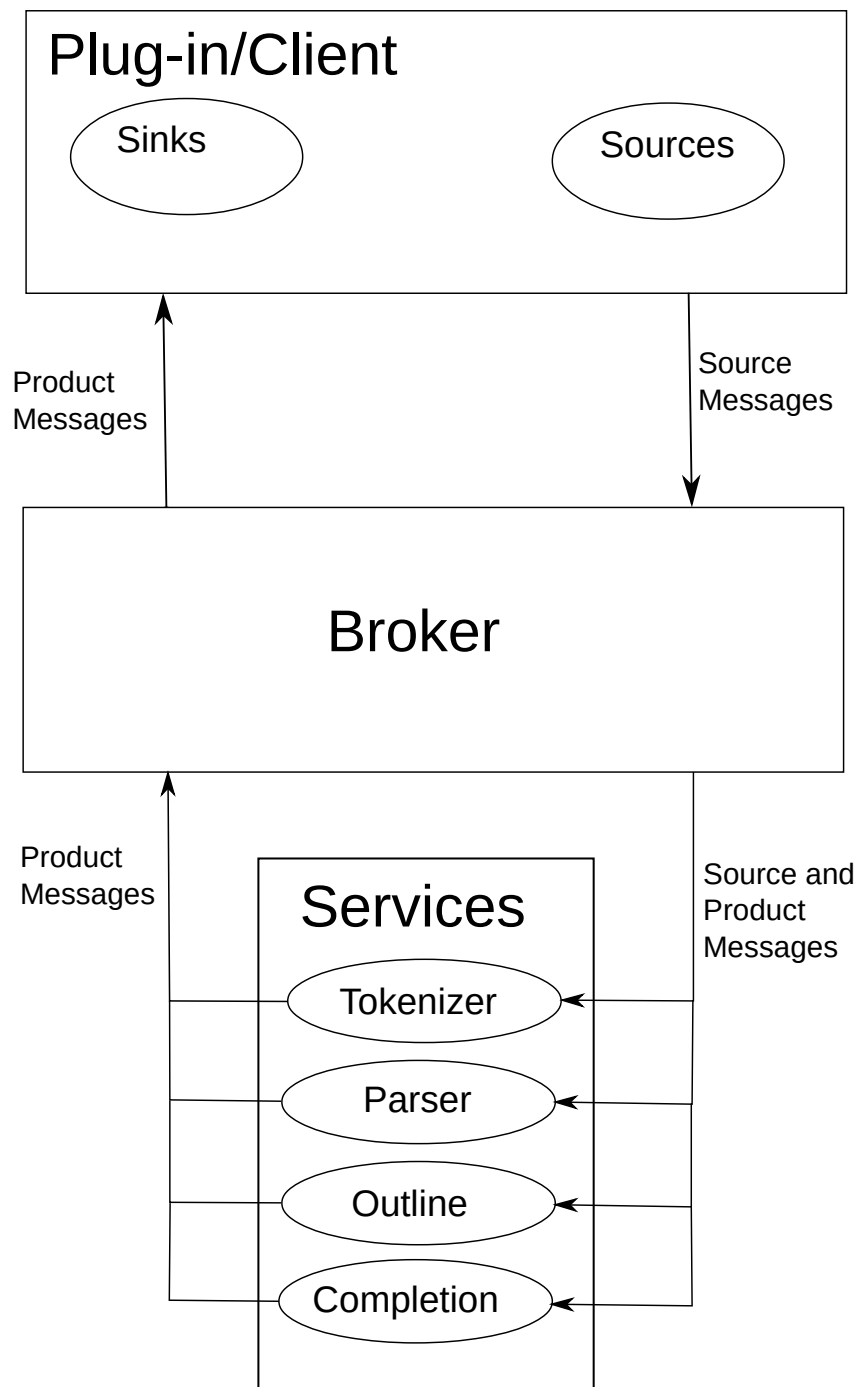
Figure 1: Monto Architecture

Used technologies include ZeroMQ (ZMQ) [5] and the JavaScript Object Notation format (JSON) [6]. Through the ZMQ framework the communication is chanelled. It offers lightweight communication abilities, a variety of message patterns and compatibility for the most common programming languages. JSON has been chosen for message interaction due to it being widely adapted by developers and being both human and machine readable.

2.2 Monto Source

Monto Sources are part of the plug-in/client. Their main responsibility is to announce changes in a source file to the broker. A changed source file potentially results to different products of services and therefore effects the client. Used communication pattern between broker and sources is publish-subscribe [7]. In this pattern messages are not directly send to a receiver, but instead being published with a topic. Interested parties can subscribe to certain topics and in this way receive desired messages. In Monto sources publish and broker subscribe to these messages. The type of sended messages is Source Message. An example for a Source Message is pictured in Table 1.

```
1   {
2       "source": "hello.py",
3       "version_id": 1,
4       "language": "python",
5       "contents": "print \"Hello World!\"",
6       "selections": []
7   }
```

Table 1: Example of a Source Message

A Source Message contains various informations. The field "source" is used to identify for which source file products should be computed. As the state of a source file continuously changes an additional field is needed to identify which Product Messages are connected to which Source Message. Therefore "version_id" has been introduced. The "language" field is e.g. used by Services to check, if the incoming Source Message has the expected language. "contents" and "selections" are used to compute actual products. "contents" contains the source code and "selections" contains a selection in the source code, which is e.g. used by the Code Completion Service.

2.3 Monto Sink

Like sources Monto sinks are part of the plug-in/client. They as well share the usage of the publish-subscribe pattern. Unlike sources Monto sinks subscribe to messages from the broker, which in this case is the publisher. The broker publishes the Product Messages of Services to the according subscribing sinks. Technically a ZMQ Socket is used to which the broker publishes and the sink subscribes.

```
1   {
2     "product": "completions",
3     "contents": [
4       {
5         "insertionOffset": 262,
6         "icon": "",
7         "description": "method: stop",
8         "replacement": "top"
9       }
10    ],
11    "service_id": "javaCodeCompletioner",
12    "invalid": [],
13    "language": "java",
14    "version_id": 11,
15    "source": "java-10lines.java",
16    "dependencies": [
17      {
18        "product": "ast",
19        "language": "java",
20        "tag": "product",
21        "version_id": 11,
22        "source": "java-10lines.java"
23      }
24    ]
25  }
```

Table 2: Example of a Product Message

Services process various informations, which the client requires. To communicate needed informations Product Messages are used. An example of a Product Message created by a Code Completion Service is pictured in Table 2. One attribute that transports mentioned informations is "contents", in which the main output of a service is encapsulated. Others are "product" which contains the name of the product, "product_id" which is similar to "version_id" and "dependencies" which contains Product Messages that have been used to process the output of the service [8]. In these dependent Product Messages the field "contents" is left out to reduce traffic. Furthermore the Product Message includes informations similar to the Source Message, from which it was originally derived. They are namely "language", "version_id" and "source" and fulfill the same purpose as in the Source Message.

An additional responsibility of sinks is to potentially process the product message. This is determined by checking certain fields in the Product Message. The sink could for example decide when reading "product": "outline" with "language": "python" to visualize the according content, which is a list of outlines. A different value of the field "language" might lead to denial of processing, if the client is accordingly configured. This depends on the implementation and configuration of the client.

2.4 Monto Broker

Monto broker communicates with a publish-subscribe mechanism to sinks and sources of the plug-in/client. To sources the broker subscribes, while to sinks it publishes. Communication to services is done through a bidirectional ZMQ Pair Connection. The task of the broker in the Monto architecture is to resolve dependencies of services and distribute messages to sinks and services. Therefore it constructs a dependency-graph. The dependency-graph depends directly to the services, whose dependencies it should represent. That's why the dependency-graph cannot be build until the services have registered to the broker.

2.5 Monto Services

Monto services solely interact directly with the broker. They can't interact with each other. To communicate with the broker ZMQ pair sockets are used, which provide a bidirectional connection. Their basic function is to process input, which can be Source Messages or Product Messages, and turn the result into a Product Message, which is sent back to the broker. Inside the Product Messages informations are encapsulated which the client can use to realise various features. The product of the Tokenizer Service can be used for example to highlight certain locations in the source file. The Code Completion Service can be used to potentially offer completion suggestions for the user. The services are typically packaged regarding the support they offer for one programming language.

2.6 Technologies in testing tool Benchmark

The audience, for which Benchmark is designed, are developers of the Monto-Editor. Based on this technologies were chosen regarding easy configuration of the startparameters and easy processing of the outcome. For configuration a file in xml-format is used. The recorded outcome of the process is written in CSV files, one file per service package(e.g. programming language), that was tested.

2.6.1 XML Format

```xml
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <benchmark>
3    <timeout>5000</timeout>
4    <repetitions>2</repetitions>
5    <waitTime>2000</waitTime>
6    <doesPrint>false</doesPrint>
7    <services>
8      <service>
9        <name>python</name>
10       <path>/home/stefan/workspace/benchmark/services/services-python.jar</path>
11       <messages>
12         <message>
13           <name>10lines</name>
14           <path>/home/stefan/workspace/benchmark/testVersionMessages/python-10lines.py</path>
15           <selection>
16             <startOffset>2</startOffset>
17             <length>1</length>
18           </selection>
19         </message>
20         <message>
21           <name>100lines</name>
22           <path>/home/stefan/workspace/benchmark/testVersionMessages/python-100lines.py</path>
23           <selection>
24             <startOffset>139</startOffset>
25             <length>0</length>
26           </selection>
27         </message>
28       </messages>
29     </service>
30     <service>
31       <name>java</name>
32       <path>/home/stefan/workspace/benchmark/services/services-java.jar</path>
33       <messages>
34         <message>
35           <name>10lines</name>
36           <path>/home/stefan/workspace/benchmark/testVersionMessages/java-10lines.java</path>
37           <selection>
38             <startOffset>2</startOffset>
39             <length>1</length>
40           </selection>
41         </message>
42       </messages>
43     </service>
44   </services>
45 </benchmark>
```

Figure 2: Example of a possible config.xml

The Extensible Markup Language (XML) provides a widely used format which main advantages are to be both human-readable and machine-readable. An example is shown in Figure 2. In Benchmark it is used to configure parameters with which the Benchmark process should be executed. The markup <benchmark> has five lower tiered markups: In <timeout> the time which will be at most be waited for an answer of a service is specified in milliseconds, <repetitions> is used to configure how often each message will be repeatedly send to the service, <waitTime> is a parameter specified in milliseconds used to regulate the waiting time between two repetitions of sending the same message, with <doesPrint> a boolean value can be specified which determines if the console-log of the started services should be printed or not and in <services> multiple <service> can be configured. <service> consists of three lower tiered markups: In <name> the name of the service package, usually the name of the programming language, is specified, <path> is used to state the path to the executable jar-file of the service package and in <messages> multiple <message> can be configured. Each of these messages will be used in the execution of Benchmark. <message> has three lower tiered markups: <name> to specify the name of a test message (it will reoccur inside the CSV file), <path> to configure the path to the actual source file and <selection> which marks a location in the source file used to fill the field "selections:" in Source Message.

<selection> consists of <startOffset>, the position in the source file, and <length>, the length of the selection.

2.6.2 CSV Format

The Comma-seperated value (CSV) format is widely used by developers and researchers. The name reflects it core and simplicity: Values can be stored in a file just by seperating them with a comma. The first line of a file contains typically a header to relate the values to. Like in this bachelor thesis CSV files are often used as input for statistical analysis due to its simplicity being both human and machine-readable.

In Benchmark for every set of services tested a new CSV file is created. An example picturing a fragment of a CSV file is shown in Table 3. The first row "id" is solely to have an unique identifier for each line. "Mode" documentates which services have been active at the test run. As in the example shown they are logged as abbreviation derived from the parameters used when starting the services. "-t -p -o -c" as in the example states, that the services "Tokenizer", "Parser", "Outliner" and "CodeCompletioner" have been started. A translation can be seen in Table 4. As there can be multiple files used to test, the "TestSource" column documents which file was used for the measurement. The contents of the columns "TokenizerDelay", "ParserDelay", "OutlinerDelay" and "CodeCompletionerDelay" are the difference in time between sending a Source Message and receiving a corresponding Product Message of the named service measured in milliseconds.

Id,Mode,TestSource,TokenizerDelay,ParserDelay,OutlinerDelay,CodeCompletionerDelay

0,-t -p -o -c ,10lines,37,41,52,66

1,-t -p -o -c ,10lines,14,19,21,22

2,-t -p -o -c ,10lines,6,15,18,17

3,-t -p -o -c ,10lines,9,11,16,16

4,-t -p -o -c ,10lines,13,17,21,19

5,-t -p -o -c ,10lines,5,15,20,18

6,-t -p -o -c ,10lines,7,12,16,16

7,-t -p -o -c ,10lines,5,8,10,10

8,-t -p -o -c ,10lines,5,8,10,10

9,-t -p -o -c ,10lines,4,7,9,9

10,-t -p -o -c ,10lines,4,11,11,11

11,-t -p -o -c ,10lines,4,7,8,8

12,-t -p -o -c ,10lines,4,6,8,8

13,-t -p -o -c ,10lines,3,6,8,8

14,-t -p -o -c ,100lines,60,94,109,111

15,-t -p -o -c ,100lines,42,59,65,68

16,-t -p -o -c ,100lines,33,55,65,68

17,-t -p -o -c ,100lines,23,40,46,48

18,-t -p -o -c ,100lines,23,36,44,41

19,-t -p -o -c ,100lines,23,44,51,48

20,-t -p -o -c ,100lines,20,40,45,43

Table 3: Extracted example content of a CSV file

3. Services

Services offer products, which can be used by a consuming client or plug-in to be processed into features for the user. They are implemented to assist in supporting a programming language. Typically a set of services is packaged to support one programming language.

3.1 Basic Service Types

The original design of Monto services by *Keidel* covers four different types of services [2]. They are designed to be comparatively small and have a single responsibility to fulfill. This design favors extensibility by simply adding a new service to a given service set if a new functionality is desired. As the product dependencies between services need to be resolved by the broker this potentially leads to longer processing times due to a bigger overhead. This is later evaluated with multiple tests.

The services are implemented using the programming language Java. Originally they were part of an Eclipse IDE plug-in for Monto. Later they have been refactored and outsourced into an own Eclipse project by *Pfeiffer* [4], which can be used as a base to implement new services [9]. MontoService is an abstract superclass containing common functionality and an API for communication to the broker. New services have to extend and implement primary onRequest(), which handles incoming Source and Product Messages and onConfigurationMessage(), which handles incoming Configuration Messages.

3.1.1 Tokenizer Service

The Tokenizer Service is used for highlighting in a source file. Therefore it is dependent on the source message, which is parsed with a Lexer of Another Tool for Language Recognition (ANTLR). An ANTLR Lexer and Parser can be generated by the tool ANTLR [10] using an ANTLR grammar. As there are grammars for every common programming language it has been used by Java [11], JavaScript [12] and Python Services [13]. The original ANTLR Tokens stream is converted into a Monto Token stream with a position, expressed through offset and length, and a category, which is later used for custom highlighting in the client.

3.1.2 Parser Service

The Parser Service is solely dependent to the Source Message and is used to create an Abstract Syntax Tree (AST). An AST is a tree representation containing the abstract syntactic structure to corresponding source code. While the AST directly isn't typically used by the plug-in/client, other services depend on it like for example the Outline Service or the Code Completion Service. Like the Tokenizer Service the Parser Service utilizes ANTLR for parsing. The ANTLR parser produces an ANTLR AST. This ANTLR AST is transformed into a Monto AST, which is passed inside a Product Message to the broker as its content.

3.1.3 Outline Service

The Outline Service is dependent on the Source Message and on the AST product of the Parser Service. It is used to identify potentially fundamental parts in the analysed source code like for example variable names or function names, which can be visualized in the client/plug-in for an enhanced overview of a source file. The incoming AST is reduced with

usage of a visitor to elements potentially interesting for an overview. The Source Message is needed to get details in the source code that have been lost when transformed into an AST like the name of a variable for example.

3.1.4 Code Completion Service

The Code Completion Service's dependendencies are analog to the Outline Service: dependent on the Source Message and on the AST product of the Parser Service. It is used to suggest completions to an incomplete statement. For example while an user is adding new code to the source file a moment will appear, when a function call is being typed, but incompletely written. The Code Completion Service can suggest completions based on the previous variable and function names. To identify the potential completion suggestions the service traverses over the AST and uses the source code similar to the Outline Service.

3.1.5 Viable Service Combinations

A given set of services like the Java Services [11], Python Services [13] and JavaScript Services [12] can be started with explicitly stating which services should be started. Although the JavaScript Services offer additional services developed by *Pfeiffer* in [4], we will constrain the possible services to the four basic types, which every currently set of services offer. It is not necessary to start all possible services at once. Due to the fact that dependencies of services to the products of other services exist not all possible service combinations are viable. For example it wouldn't be productive to start the Outline Service without starting the Parser Service as the Outline Service would never yield results since it's dependencies are never fulfilled. From the four basic Services a fixed set of possible combinations can be derived. These are listed in Table 4 with the startparameters needed to specify when starting the service set.

Startparameter	Enabled Services
-t -p -o -c	Tokenizer Service, Parser Service, Outline Service, Code Completion Service
-t -p -c	Tokenizer Service, Parser Service, Code Completion Service
-t -p -o	Tokenizer Service, Parser Service, Outline Service
-p -o -c	Parser Service, Outline Service, Code Completion Service
-p -c	Parser Service, Code Completion Service
-p -o	Parser Service, Outline Service
-t -p	Tokenizer Service, Parser Service
-p	Parser Service
-t	Tokenizer Service

Table 4: Mapping of Startparameter to set of enabled services

3.1.6 Hypothesis

As shown in the previous subsection the number of started services can vary. How this might effect the runtime response times of each single service is unclear. In the scope of the bachelor thesis this behaviour is tested. As the services are all started in the same process the hypothesis is proposed, that service response times increase, if more services are active. This is checked and evaluated in the chapter 5 Evaluation.

3.2 Implementation of Python Services

To increase the support of programming languages in Monto it was planned to implement a set of services with the purpose to enable support to the programming language Python. Python exists in two maintained versions: Python2 and Python3, which have slight differences in grammar. The implementation for this bachelor thesis focuses on Python3 as having the better perspective to keep on being maintained in the future. A grammar offered by ANTLR [14] has been utilized to generate an ANTLR lexer and an ANTLR parser. These are used in the Tokenizer Service, the Parser Service, the Outliner Service and the Code Completion Service as mentioned in the previous section.

4. Distributor

In this chapter the Distributor is presented. It is designed to tackle potential performance issues inside the Monto framework. First the situation before the introduction of the istributor into Monto is described. Following the concept and implementation are discussed.

4.1 Goal and Hypothesis

In the original design of Monto *Keidel* planned services to provide single products. The services are limited to supply one functionality and fulfill one responsibility. Additional functionality is provided by adding new services in contrast to modifying existing services. As services depend potentially on the products of other services this leads to overhead and might influence the overall performance and response times of services. To erase the overhead would only be possible by merging all functionalities into one service, which is not feasible as it would reduce extensibility and maintainability. A different measure is to reduce the size of the overhead. The size of the overhead between services is defined through the size of Product Messages, which are send by services and received by dependent services via the broker. The goal of the Distributor is to reduce the size of the overhead. The corresponding hypothesis is that this has a clear impact on the response time of services.

4.2 Concept of the Distributor

In the general concept of services designed by *Keidel* it is not defined in which process services for one programming language are running. The implementations of Java Services [11], JavaScript Services [12] and Python Services [13] though share the fact they are running in the same process of the operating system (OS). This is an important precondition for the function of the Distributor as it aims to reduce the overhead by providing a cache for a part of the Product Message. The part being the value of the "contents" field. The size of "contents" varies the most depending on the size of the source code as it can be assumed bigger source files lead to bigger product contents. Caching the value of the contents field effectively limits the at most size of a Product Message as the other values are comparatively fixed in size. The cached "contents" is replaced by a key. Therefore the name of the field is changed to "contents_key". Product Messages with "contents_key" instead of "contents" are send to the broker. The broker publishes these Product Messages to services and sinks. In the operating system process of the services the Distributor receives the Product Messages and substitutes "contents_key" with "contents" including their value by executing a look-up. For services input and output remain the same: Product/Source Messages with "contents" field. The Distributor handles the caching and distribution of "contents".

Its general structure is pictured in Figure 4. As you can see the services Tokenizer Service, Parser Service, Outline Service and Code Completion Service don't interact directly with the broker anymore. In the picture they are abbreviated with their first letter.
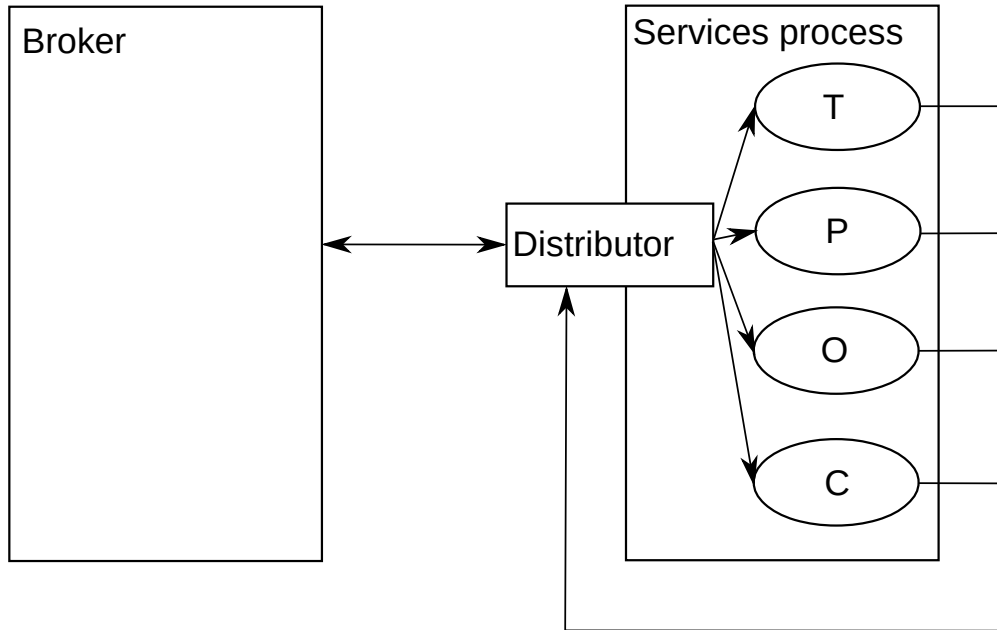
Figure 3: Distributor Concept

4.3 Implementation of the Distributor

Like in the concept defined the Distributor plans on intercepting and adjusting in- and outgoing Product Messages of services. To achieve this goal the implementation of the Distributor utilizes the MontoService class of services-base-java [9]. As it is extended by every implemented service and encapsulates the handling of incoming and outgoing messages, it offers the ideal place to hook into the implemented base classes.

In Figure 4 the thread inside the MontoService class is shown handling incoming and outgoing messages. The highlighted parts showcase the adjusted parts to realize the Distributor. In lines 14 to 20 the handling of incoming Product Messages is shown. First the incoming JSON Object is decoded. As it is uncertain, if they include "contents" or "contents_key", a visitor is used to ensure a Product Message with "contents" field. In the visitor a look-up on the Distributor is executed which throws an InvalidKeyException, if for the value of the "contents_key" field no value inside the cache could been found. Valid Product Messages are forwarded onto the next highlighted part in lines 28 to 34. On each message the "onRequest" method is executed returning the resulting ProductMessage, which has a "contents" field. The value of the "contents" field is stored into the Distributor returning a key as an Integer value. A new ProductMessageWithKey is created with the original Product Message and the key. Lastly the message is encoded and published to the broker.

```
 1  serviceThread = new Thread() {
 2      @Override
 3      public void run() {
 4      Distributor distributor = new Distributor();
 5          while(running)
 6              that.<JSONArray,List<Message>>handleMessage (
 7                      serviceSocket,
 8                      messages -> {
 9                          List<Message> decodedMessages = new ArrayList<>();
10                          for (Object object : messages) {
11                              JSONObject message = (JSONObject) object;
12
13                          if(message.containsKey("product")){
14                              try {
15                                  IProductMessage prdMsgWithKeyOrContents = ProductMessages.decode(message);
16                                      ProductMessage prdMsgWithContents = prdMsgWithKeyOrContents.accept(new ContentsVisitor(distributor));
17                                      decodedMessages.add(prdMsgWithContents);
18                                  } catch (InvalidKeyException e) {
19                                      e.printStackTrace();
20                                  }
21                              } else{
22                                  decodedMessages.add(SourceMessages.decode(message));
23                              }
24                          }
25                          return decodedMessages;
26                      },
27                      messages -> {
28                          ProductMessage prdMsgWithContents = onRequest(messages);
29
30                          Integer contentsKey = distributor.put(prdMsgWithContents.getContents());
31                          ProductMessageWithKey prdMsgWithContentsKey = ProductMessages.constructMsgWithKey(prdMsgWithContents, contentsKey);
32
33                          String toBeSend = ProductMessages.encode(prdMsgWithContentsKey).toJSONString();
34                          serviceSocket.send(toBeSend);
35                      }
36                  );
37      }
38  };
```

Figure 4: Distributor Implementation in MontoService

5. Evaluation

In chapter 3 a hypothesis is proposed: An increase in the number of active services leads to longer response times of each service. To validate this a testing tool for automated repeated message sending and time measuring has been implemented and is discussed in the first section. Following the test procedure is introduced. After that the measured times are presented and discussed.

5.1 Test Procedure with Testing Tool Benchmark

To validate the hypothesis of chapter 3 a test was designed: three different Source Messages per programming language varying in size are published repeatedly to the broker. This is executed for every possible set of active services as specified in 3.1.5. In the test that was performed for the following analysis each Source Message was published 100 times for each service combination. The time difference between sending a Source Message and receiving a Product Message was recorded for each service in each case. To provide valid and reproducible data a main requirement to the test is to be noisefree and easily repeatable. User tests utilizing the plug-in/client wouldn't be feasible as the sending of messages need to be repeated comparatively often to yield clear results. Therefore a testing tool was developed to automate the procedure.
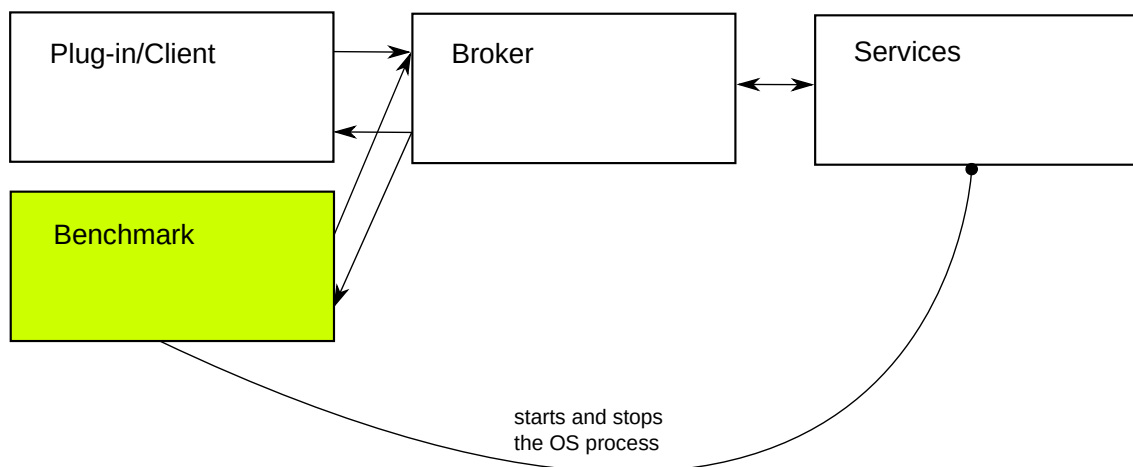
Abbildung 1: Benchmark scheme

The testing tool is named Benchmark. Abbildung 1 shows the interaction of Benchmark with the other modules. Benchmark registers to the ZMQ sockets of sources and sinks similar to the plug-in/client. That leaves the plug-in/client bypassed. Source Messages are published to the broker and Product Messages are received through subscribtion. As the active services vary through out the execution of one test run Benchmark needs to start and stop the active set of services as required during the runtime. As a precondition for successful execution the broker needs to have been started.

5.2 Analysis

The discussed test is executed with three Source Messages per programming language of different lengths. Basing on the assumption bigger files lead to longer response times files have been chosen from open source projects with the rough graduation of a length of 10, 100 and 1000 lines of code. The code is partly generic to achieve the condition of fulfilling 10, 100 or 1000 lines of code. The content of the files is diverse including comments etc. While choosing the test files no demand to the substance other than being executable was made. It should lead simply to a rough differentiation between the source files.

The test provides 100 data sets per Source Message per enabled set of active services per programming language. This can be calculated to 100*3*9*3 = 8100 data entries and 36 different possible plots (four per Source Message per programming language). In the following subsections a selection of the data is presented and discussed.

The test was performed on an Lenovo Y50 running Ubuntu 15.10 64-bit. The corresponding hardware includes a memory of 11,7 GiB Ram, the processor Intel® Core™ i7-4710HQ CPU @ 2.50GHz * 8 and the graphic card Intel® Haswell Mobile. The used files can be found at [15].

The following plots have been created using the programming Language R in version 3.2.3 and ggplot2 version 2.0.0. Only a selection of the data representing plots are discussed. The others can be found in the Appendix 10.

5.2.1 File length comparison

The assumed impact described in the hypothesis (An increase in the number of active services leads to longer response times of each service) might not take effect with small files, as they are processed potentially faster than longer files. To compare the influence of the file length three plots are presented where the programming language and the service examined are the same in Figure 5, Figure 6 and Figure 7. As a programming language Java is chosen. This selection is rather arbitrary as the choice of the programming language should not make a notably difference. As a service the Parser Service is selected. This is due to the fact that most viable services include the Parser Service. The selected plots therefore offer the most content.
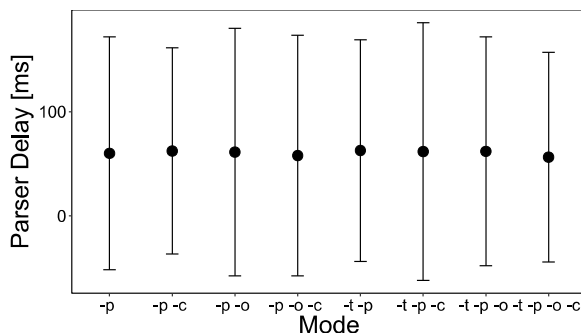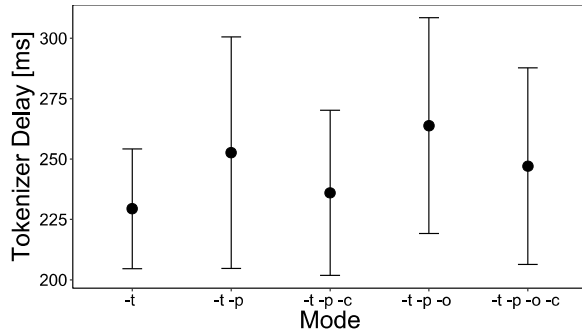

Figure 5: Java Parser Delay with 10 lines test file
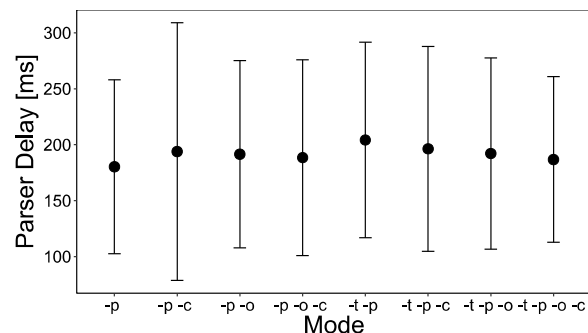
Figure 6: Java Parser Delay with 100 lines long test file
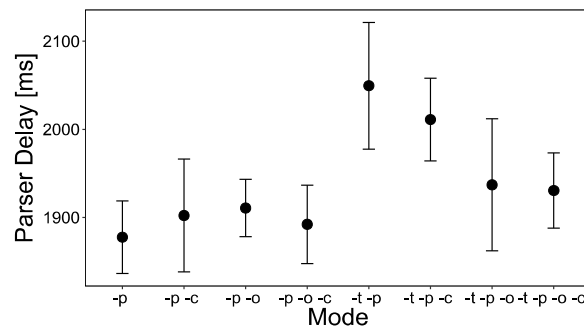
Figure 7: Java Parser Delay with 1000 lines long test file

The delay pictured in Figure 5 shows no clear variation in mean or standard deviation between the different running sets of active services. The standard deviation even range into the negative area. The response times are for an user not noticeable. Minor differences between the modes should even decrease, if the number of repetitions in the test would have been higher. The hypothesis in 3.1.6 can not be confirmed so far.

The mean of the Parser Delay pictured in Figure 6 shows a similar variation as the mean in Figure 5. The value is a bit higher than for the 10 lined file, which has been expected. No clear differences in the standard variation are visible. The mode -p -c has a bit higher standard deviation than the others. It is not enough to derive a real difference.

The delays associated with a 1000 lines long file pictured in Figure 7 are considerably higher than with the other two files. Response times vary around 2000 milliseconds. Clear differences both in mean and standard deviation can be observed. The highest standard deviation is connected to the mode -t-p, while the lowest standard deviation is monitored with the mode -p -o. The modes with a started Tokenizer Service tend to have bigger delays than the sets of active services without a tokenizer.

Comparing the three figures confirms our presumption, that the file length considerably impacts the response time of a service. We can determine that response times for file lengths up to 100 lines are likely to be not noticeable by the user, while the duration of 2 seconds waiting time should lead to a poor user experience. A critical turning point in perception of this delay times should be with a file length between 100 and 1000 lines. Our original hypothesis from 3.1.6 Hypothesis can not be confirmed. The waiting time for a service response is not proportional to the number of started services, though a tendency could be recognized: The response time of the Parser Service increases, if the Tokenizer Service is active.

5.2.2 Service comparison

The performed test recorded additional to the Parser Service response times of the Tokenizer Service, the Outline Service and the Code Completion Service. As we have seen in the previous subsection: for small files no clear difference in the waiting time could be perceived. Therefore response times between services are compared using the comparatively large 1000

lines long file. For a better comparability the programming language is arbitrarily chosen to be JavaScript for each examined service.

Figure 8: JavaScript Tokenizer Delay with 1000 lines long test file
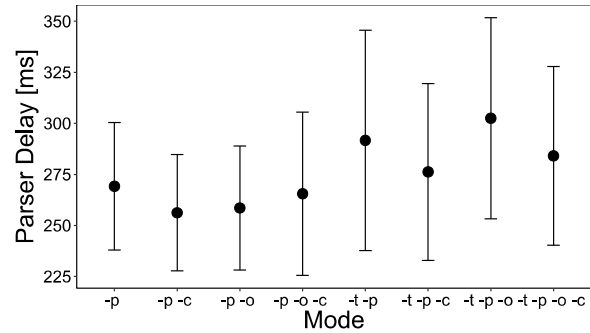
Figure 9: JavaScript Parser Delay with 1000 lines long test file
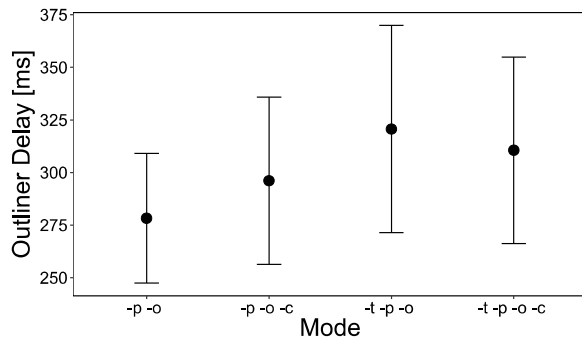
Figure 10: JavaScript Outline Delay with 1000 lines long test file
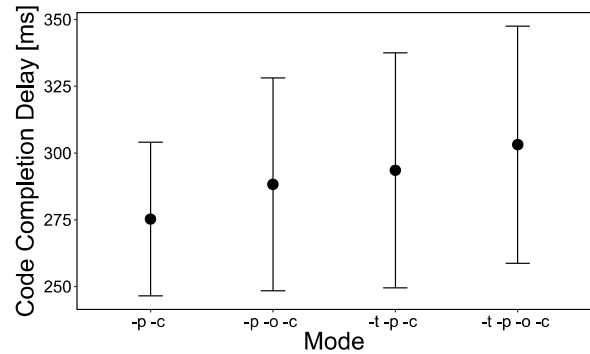
Figure 11: JavaScript Code Completion Delay with 1000 lines long test file

The delays of the Tokenizer Service pictured in Figure 8 show little differences. The means of every measured mode is in range of the standard deviation of each other measurement. Standard deviations are nearly the same with a slight tendency of a smaller standard deviations for the mode -t. The JavaScript Parser Delays showcased in Figure 9 indicate the tendency of a slightly higher response times in the modes where the Tokenizer Service is started. It is a slight tendency though, as the calculated means are in close range to eachother. Standard deviations are very similar with a slight tendency to be smaller for the sets of active services started without a Tokenizer Service. The same tendencies can be observed in Figure 10 even a bit slighter. The mode with the least number of active services, -p -o, has the lowest mean and the lowest standard deviation. The measured values are still in a close range to eachother so a clear regularity can not be derived. Response times to the JavaScript Code Completion Service shown in Figure 11 are similar to those of the Outline Service. The mode -p -c has the smallest mean and smallest standard deviation. The calculated means and standard deviations of the other modes are slightly higher with "-t -p -o -c" having the highest values. Comparing the results of the four observations it can be determined that no clear regularity can be derived. Though for each service type the same slight tendendies have been discovered: a smaller number of active services tend to lower

standard deviations and means. Especially enabling the Tokenizer Service tend to a slight higher response times of the other services. The data is still not clear enough to confirm 3.1.6 Hypothesis.

5.2.3 Programming Language comparison

The third varying factor while performing the test is the programming language. To determine if the response times behave differently depending on the programming language three test cases are compared as a showcase with each other. The Parser Service is chosen as the service type in each test case due to the fact it offers the most test data. As smaller files do not provide clear data the chosen files size is 1000 lines.

The Python Parser Delay pictured in Figure 12 shows no clear regularity. Still tendencies can be observed like in the previous subsections. Modes including an active Tokenizer Service tend to have higher means of response time and slightly higher standard deviations than modes without an active Tokenizer Service. The tendencies of the Java Parser with a 1000 lines long test file are discussed in 5.2.1 File length comparison and are pictured in Figure 7. The plot to JavaScript case is discussed in 5.2.2 Service comparison and is visible in Figure 9.
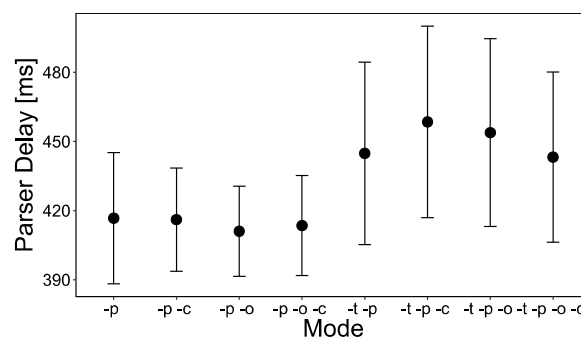
Figure 12: Python Parser Delay with 1000 lines long test file

Comparing the test results with eachother it can be noticed they don't show a clear regularity. They all share the same slight tendency though: service sets with active Tokenizer Service tend to have higher response times and bigger standard deviations than those without active Tokenizer Service. As the impact of the varying modes are similar for each programming language it can be stated that the absolute response times are massively different. For Java the response times vary around approximately 1950 milliseconds while they vary around 275 milliseconds for JavaScript and around 440 milliseconds for Python. This can be explained as Java has a considerably more complex grammar than the other two languages and the utilized ANTLR Parser and Lexer being not optimized for runtime.

6. Related Work

The implementation of the Distributor is an adaption of the Broker Pattern, which is presented in [16]. The communication between two parties, Monto Broker and Monto Services, is chanelled through a broker, the Distributor. To not confuse Monto Broker and Distributor with one another, the Distributor can be seen as a service broker and the Monto Broker as a system broker as it coordinates the communication between all components in the Monto architecture. The Broker Pattern is a commonly used pattern, hence multiple other research projects have been conducted.

Francu et al have developed an Advanced Communication Toolkit (ACT) [17]. The included tools aim to provide technologies that let the Broker Pattern be implemented in a structured way. They claim through using ACT to be able to for example replace primitive communication mechanisms, protocols and data marshalling. Patterns which can be seen as simplified versions of the Broker Pattern should be able to implement including Client-Dispatch-Server or Forwarder-Receiver[18].

A way of "Securing the Broker Pattern" [19] is examined by *Morrison et al*. Decoupling into multiple components leads to the challenge to secure communication between the components. As the current implementation of the Monto architecture is running on a single system, this is not an issue. If the architecture would run on a distributed system however, means to authenticate and authorise components to each other have to be implemented. The "Secure Broker" presents an approach to tackle this problem by introducing identities of components, providing authorization facilities and use Reference Monitors, that handle access by checking the rights of a requester.

7. Future Work

In chapter 3 the four current base service types are described. State-of-the-art IDEs though offer essentially more features. To close this gap more services need to be developed. As the services rely on Product Messages to be transmitted, these have to be equally enhanced, which has to be done by reworking product conventions. A small collection of possible new services would be: refactoring facilities, auto completion of programming language dependend expressions and a debugger.

Beside new types of services the amount of supported programming languages could be enhanced. With three different types it offers still comparatively little support. Beside programming language support the support for web application frameworks like AngularJS, BackboneJS or TypeScript could be introduced as they are based on JavaScript but add new expressions to extend programming abilities.

The testing tool presented in section 5.1 provides the capability for automated runtime tests. Behaviour of the system at runtime is already tested and analysed as a part of this bachelor thesis. As the Monto framework is continuously refined and therefore changes it's structure the results of chapter 5 might soon not be up-to-date anymore. The tool offers the possibility to gather constant performance data. It could be enhanced to offer more than just recorded delay times of services. For example it could monitor the system processes of broker and services, when stressed with continuous send Source Messages. Thus it has the potential to be a constantly used testing tool at development time.

8. Conclusion

In this thesis the Monto framework developed by *Keidel* is enhanced. It is a relatively new framework decoupling an IDE into reusable parts. Most of the IDEs currently used by developers have been improved by multiple people for quite some time. Therefore a lot of catching-up has to be done to make Monto able to compete with established IDEs.

The first improvement provided and presented in this thesis is the implementation of Python Services increasing the number of supported programming languages to three. The Python Services cover the four basic service types: Tokenizer Service, Parser Service, Outline Service and Code Completion Service. For these a hypothesis is proposed: The number of active services effect the responding time of each active service. This is analysed utilizing test data previously recorded. Although the collected data does not reveal a clear regularity, slight tendencies could be observed: the Tokenizer Service tends to delay response times and increase the standard deviation of other services.

While the presented test aims to get an insight of possible performance issues, the Distributor introduced intents to improve the performance by limiting the overhead between services. Product Messages published before contain the product as the value of the "contents" field. Therefore the size of these Product Messages are proportional to the size of the product. The size of the new Product Messages containing the field "contents_key" is fixed, which leads to decrease of overhead and thus should lead to a better performance for bigger source files.

9. Acronyms

IDE	Integrated Development Environment
API	Application Programming Interface
XML	Extensible Markup Language
CSV	Comma-seperated values
ANTLR	Another Tool for Language Recognition
-t	Startparameter to activate Tokenizer Service
-p	Startparameter to activate Parser Service
-o	Startparameter to activate Outline Service
-c	Startparameter to activate Code Completion Service
OS	Operating System
ACT	Advanced Communication Toolkit

10. Appendix

10.1 Plots for Java test data
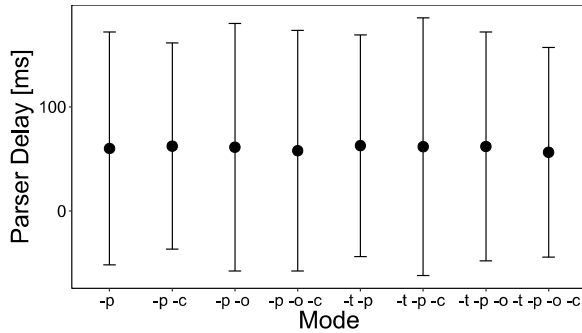
AppendixFigure 1: Java Parser Delay with 10 lines long test file
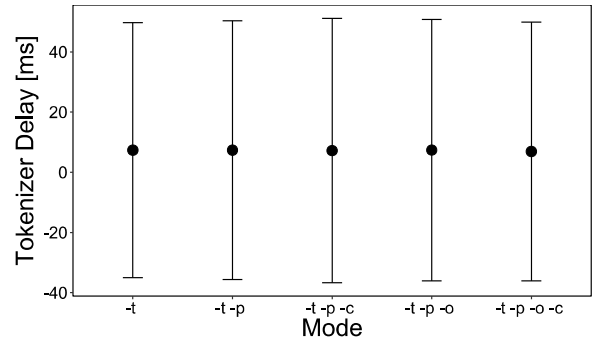
AppendixFigure 2: Java Tokenizer Delay with 10 lines long test file
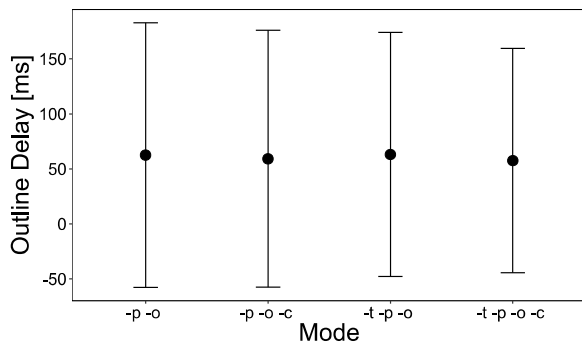
AppendixFigure 3: Java Outline Delay with 10 lines long test file
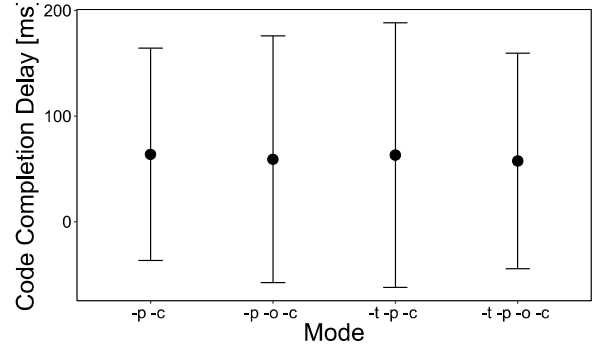
AppendixFigure 4: Java Code Completion Delay with 10 lines long test file
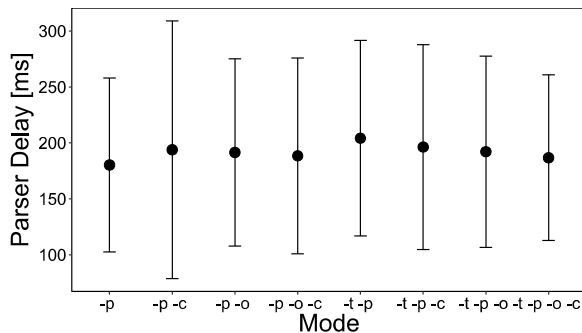
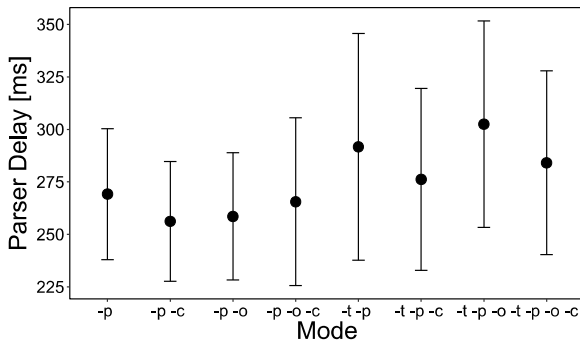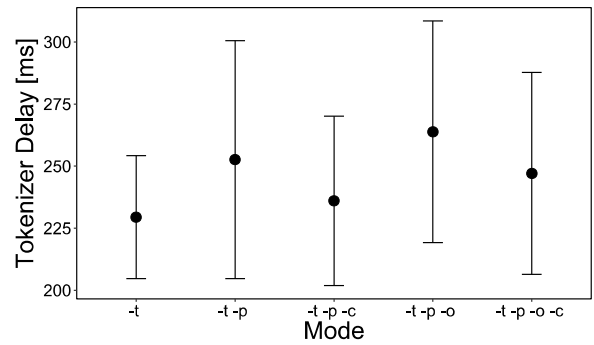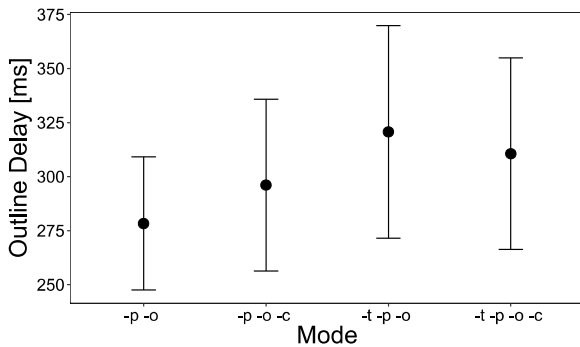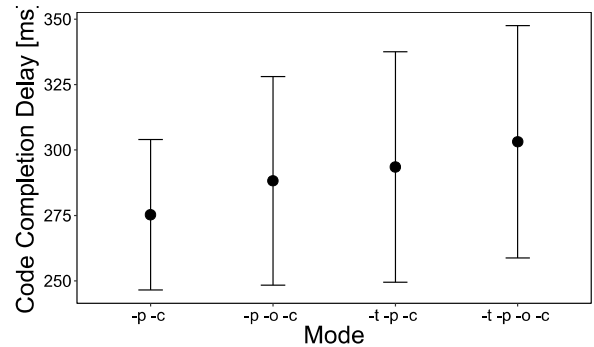AppendixFigure 5: Java Parser Delay with 100 lines long test file
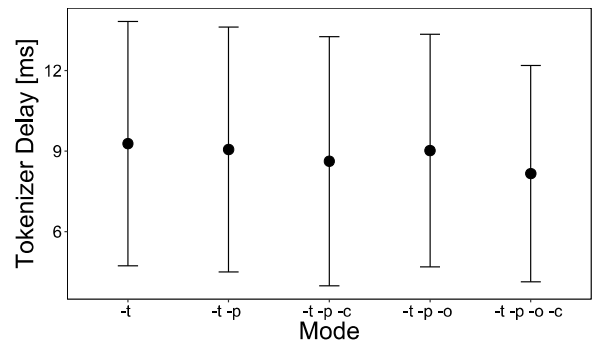
AppendixFigure 6: Java Tokenizer Delay with 100 lines long test file
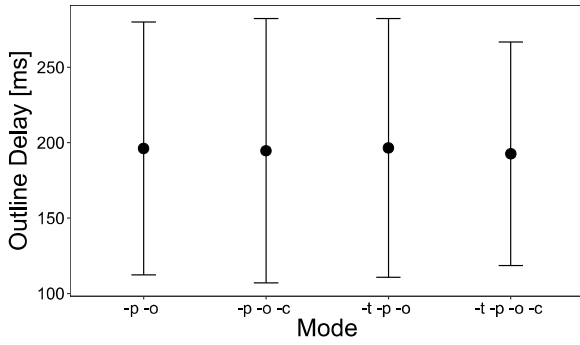
AppendixFigure 7: Java Outline Delay with 100 lines long test file
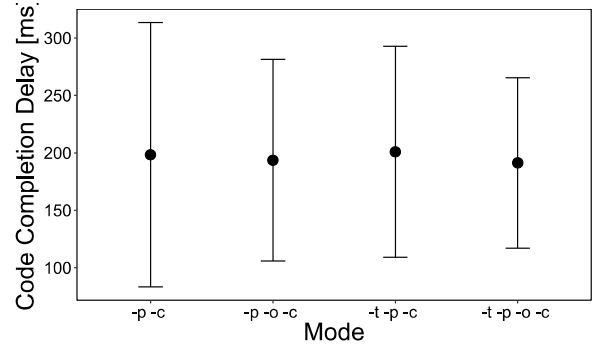
AppendixFigure 8: Java Code Completion Delay with 100 lines long test file
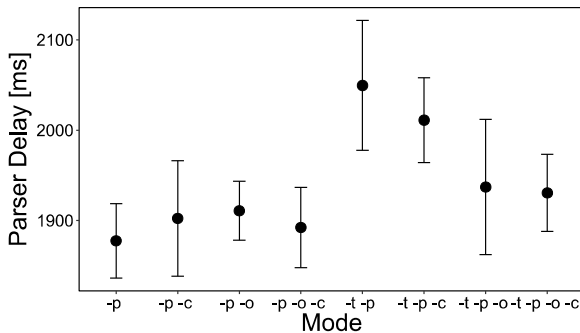
AppendixFigure 9: Java Parser Delay with 1000 lines long test file
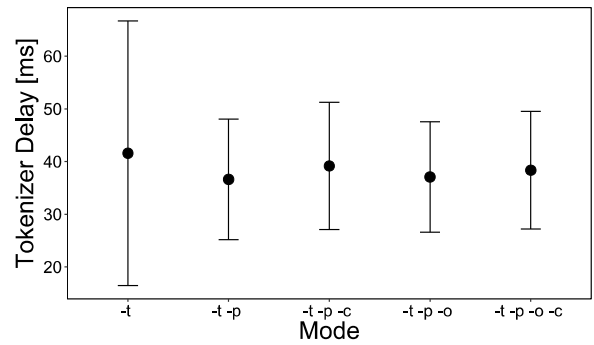
AppendixFigure 10: Java Tokenizer Delay with 1000 lines long test file
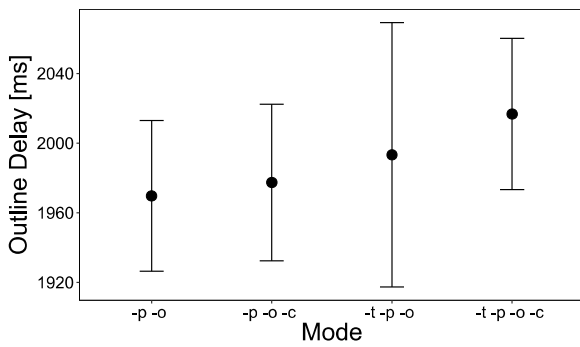
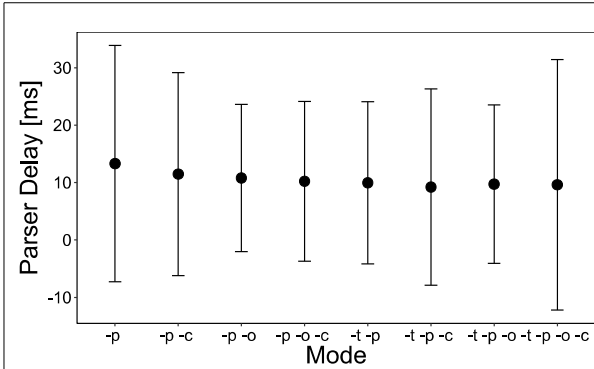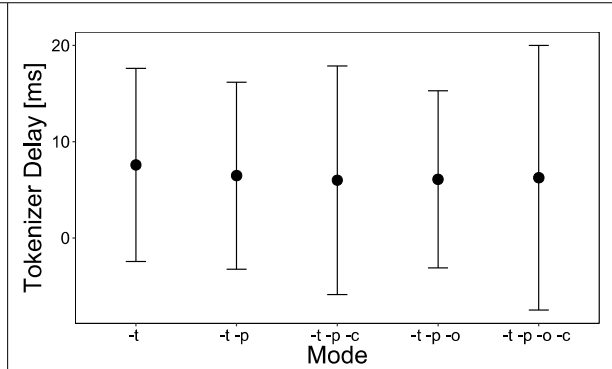AppendixFigure 11: Java Outline Delay with 1000 lines long test file
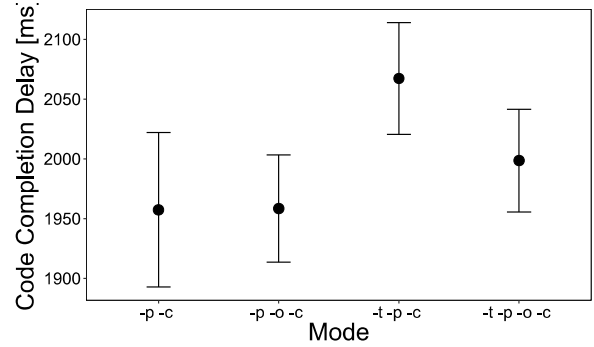
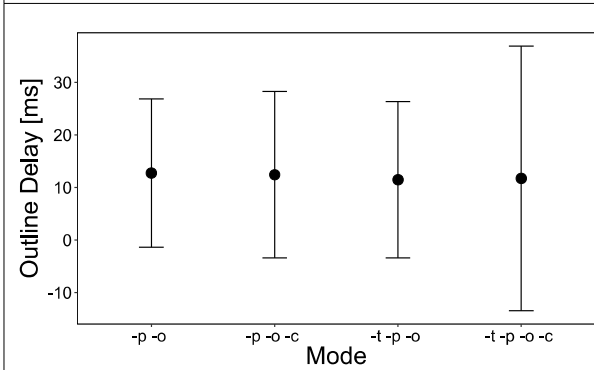AppendixFigure 12: Java Code Completion Delay with 1000 lines long test file
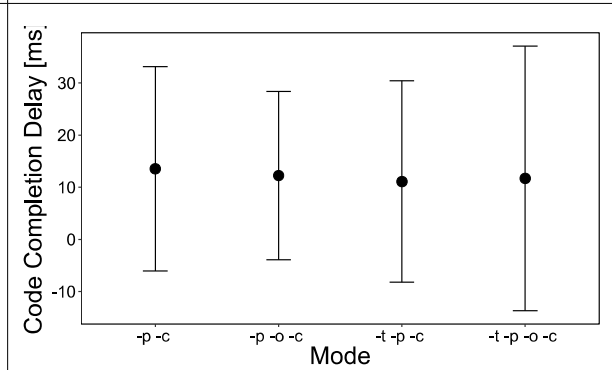
10.2 Plots for JavaScript test data
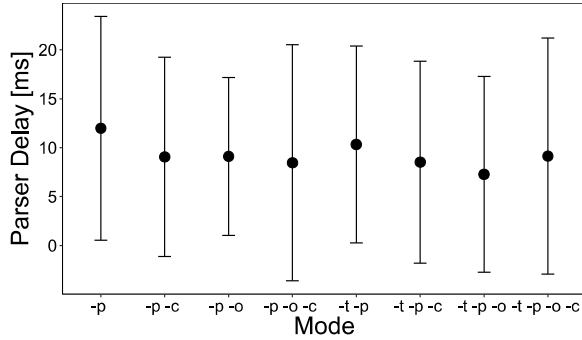
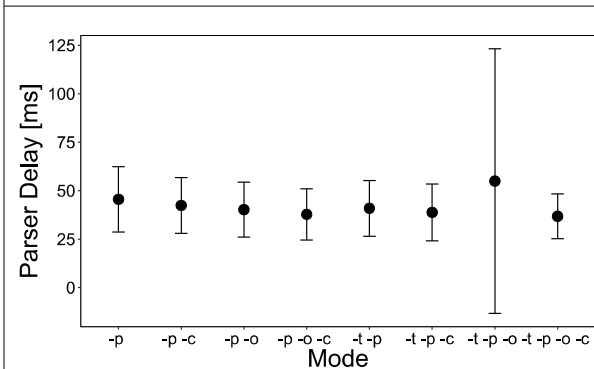AppendixFigure 13: JavaScript Parser Delay with 10 lines long test file
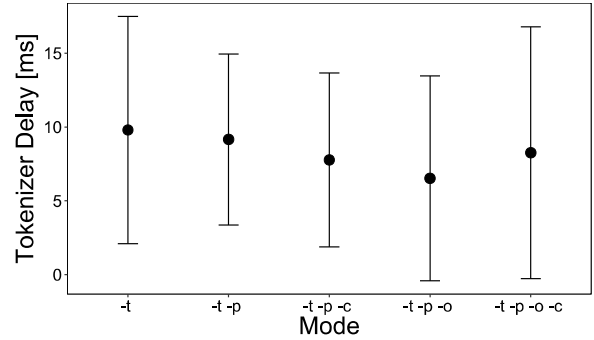
AppendixFigure 14: JavaScript Tokenizer Delay with 10 lines long test file
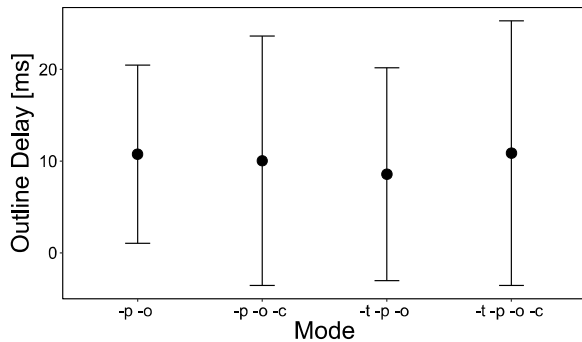
AppendixFigure 15: JavaScript Outline Delay with 10 lines long test file
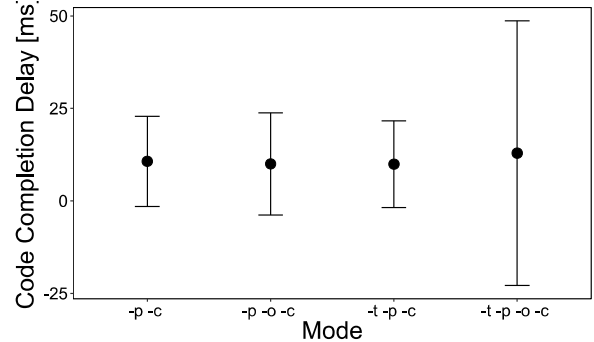
AppendixFigure 16: JavaScript Code Completion Delay with 10 lines long test file
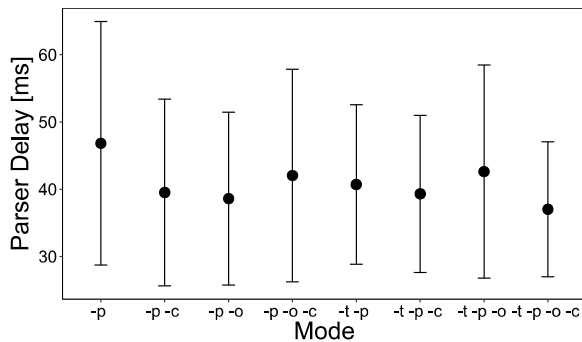
AppendixFigure 17: JavaScript Parser Delay with 100 lines long test file
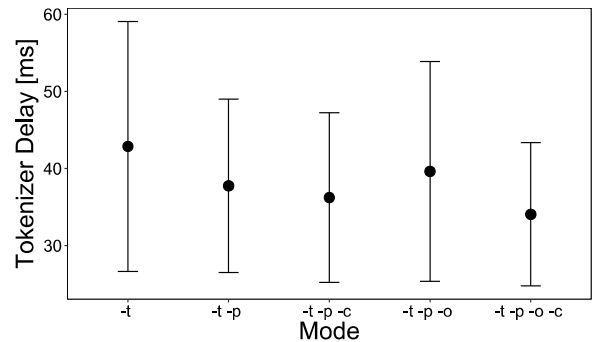
AppendixFigure 18: JavaScript Tokenizer Delay with 100 lines long test file
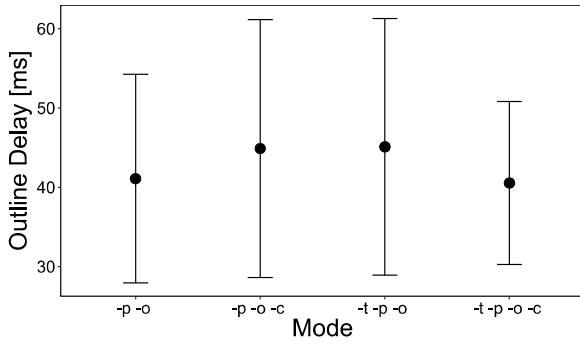
AppendixFigure 19: JavaScript Outline Delay with 100 lines long test file
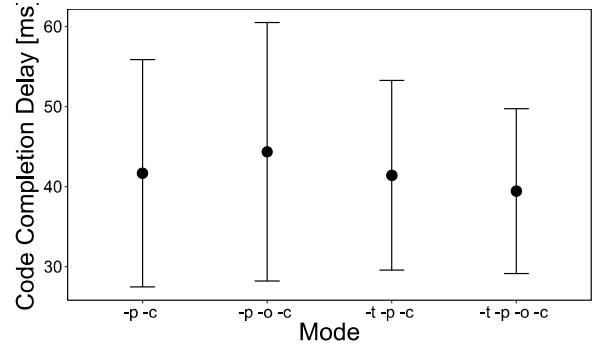
AppendixFigure 20: JavaScript Code Completion Delay with 100 lines long test file

AppendixFigure 21: JavaScript Parser Delay with 1000 lines long test file

AppendixFigure 22: JavaScript Tokenizer Delay with 1000 lines long test file

AppendixFigure 23: JavaScript Outline Delay with 1000 lines long test file

AppendixFigure 24: JavaScript Code Completion Delay with 1000 lines long test file

10.3 Plot for Python test data

AppendixFigure 25: Python Parser Delay with 10 lines long test file

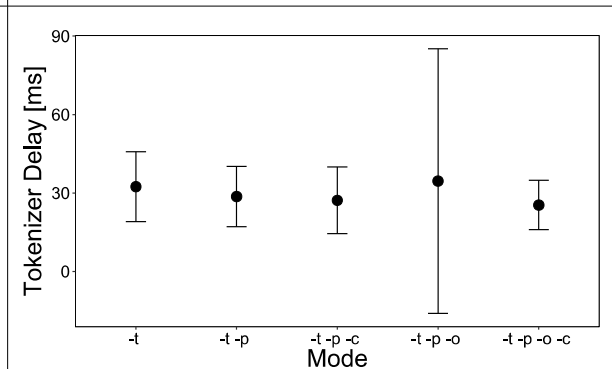AppendixFigure 26: Python Tokenizer Delay with 10 lines long test file

AppendixFigure 27: Python Outline Delay with 10 lines long test file
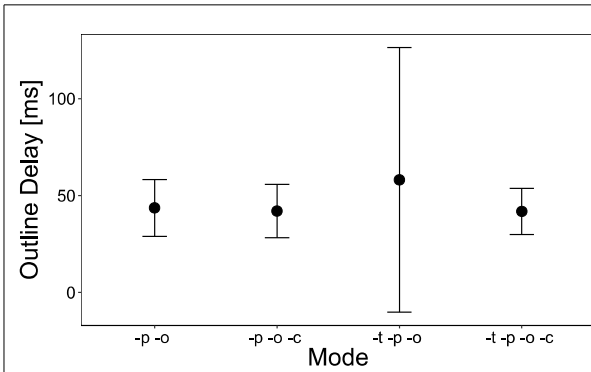

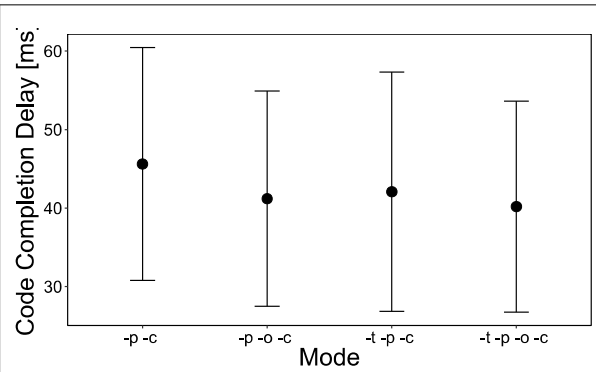AppendixFigure 28: Python Code Completion Delay with 10 lines long test file

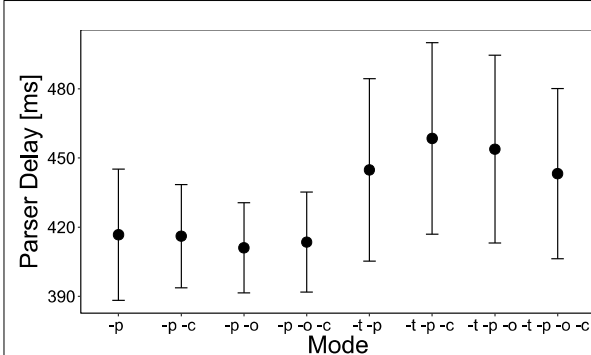AppendixFigure 29: Python Parser Delay with 100 lines long test file

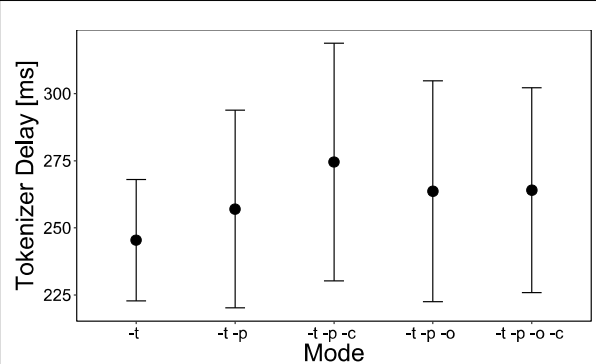AppendixFigure 30: Python Tokenizer Delay with 100 lines long test file

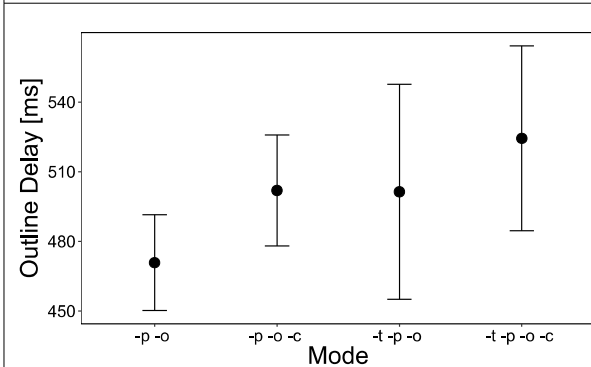AppendixFigure 31: Python Outline Delay with 100 lines long test file

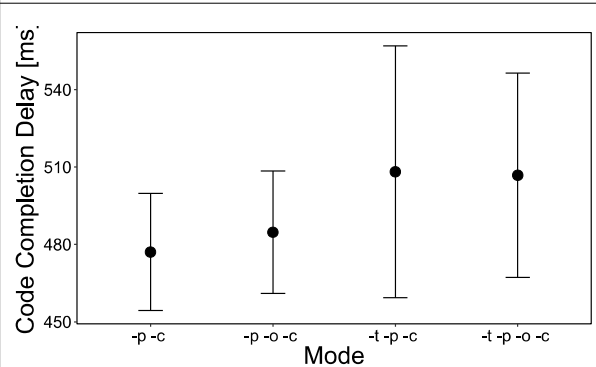AppendixFigure 32: Python Code Completion Delay with 100 lines long test file

AppendixFigure 33: Python Parser Delay with 1000 lines long test file

AppendixFigure 34: Python Tokenizer Delay with 1000 lines long test file

AppendixFigure 35: Python Outline Delay with 1000 lines long test file

AppendixFigure 36: Python Code Completion Delay with 1000 lines long test file

Bibliography

[1] : Scott Buckley Tony Sloane Matt Roberts and Shaun Muscat. "Monto: A Disintegrated Development Environment" . In: Software Language Engineering. 2014

[2] : Keidel, Sven. A disintegrated development environment. Technische Universität Darmstadt. April 2015

[3] : Monto-Editor. https://github.com/monto-editor/, Accessed: 23.02.2016

[4] : Wulf Pfeiffer. A web-based code editorusing the Monto framework. Technische Universität Darmstadt. October 2015

[5] : ZeroMQ. http://zeromq.org/, Accessed: 23.02.2016

[6] : JavaScript Object Notation format. http://json.org/, Accessed: 23.02.2016

[7] : Frank Buschmann, Kevlin Henney and Douglas C. Schmidt. "Publish-Subscriber". In: Pattern-oriented Software Architecture. 214-220

[8] : Monto Message Conventions. https://github.com/monto-editor/message-conventions, Accessed: 23.02.2016

[9] : Java Base Service Package . https://github.com/monto-editor/services-base-java, Accessed: 23.02.2016

[10] : Another Tool for Language Recognition. http://www.antlr.org/, Accessed: 23.02.2016

[11] : Monto Java Services. https://github.com/monto-editor/services-java, Accessed: 23.02.2016

[12] : Monto JavaScript Services. https://github.com/monto-editor/services-javascript, Accessed: 23.02.2016

[13] : Monto Python Services. https://github.com/monto-editor/services-python, Accessed: 23.02.2016

[14] : ANTLR Python3 grammar. https://github.com/antlr/grammars-v4/blob/master/python3/Python3.g4, Accessed: 10.10.2015

[15] : Used files to perform the Benchmark test. https://github.com/skockmann/bachelorthesis, Accessed: 08.03.2016

[16] : Frank Buschmann, Kevlin Henney and Douglas C. Schmidt. "". In: Pattern-oriented Software Architecture. 73-75

[17] : Cristian Francu and Ivan Marsic. "An Advanced Communication Toolkit for Implementing the Broker Pattern". Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on. 1999.

[18] : Frank Buschmann, Kevlin Henney and Douglas C. Schmidt. "Forwarder-Receiver". In: Pattern-oriented Software Architecture. 218

[19] : Patrick Morrison and Eduardo B. Fernandez. "Securing the Broker Pattern". EuroPLoP.

2006.