

# ezDL – Developer Guide

Matthias Jordan, Thomas Beckers<sup>1</sup>

Sunday 29<sup>th</sup> April, 2012, 11:59 (last build)

<sup>1</sup>Working group Information Engineering, University of Duisburg-Essen



# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>Building</b>	<b>7</b>
2.1	Maven on the command line . . . . .	7
2.2	Maven in Eclipse . . . . .	8
2.2.1	Preparations . . . . .	8
2.2.2	Configuring the project . . . . .	8
2.2.3	Building . . . . .	9
<b>3</b>	<b>How we code</b>	<b>11</b>
3.1	Write self-documenting code . . . . .	12
3.1.1	Guidelines . . . . .	17
<b>4</b>	<b>Architecture</b>	<b>19</b>
<b>5</b>	<b>Agents</b>	<b>23</b>
5.1	General architecture . . . . .	23
5.2	Requests and request handlers . . . . .	25
5.3	Implementing an agent that handles a request . . . . .	27
<b>6</b>	<b>Wrappers</b>	<b>29</b>
6.1	General architecture . . . . .	29
6.2	Solr wrappers . . . . .	30
6.3	Toolkit wrappers . . . . .	30
6.4	Requirements for all wrappers . . . . .	30
6.5	Query processing in wrappers . . . . .	31
6.5.1	Capability.PROXIMITY_SEARCH . . . . .	32
6.5.2	Capability.SIMPLE_YEAR_RANGE . . . . .	32

6.5.3	Capability.FULL_BOOLEAN_YEAR_RANGES . . . . .	32
6.5.4	Capability.SINGLE_WILDCARDS . . . . .	33
6.5.5	Capability.MULTI_WILDCARDS . . . . .	33
6.5.6	Capability.FULL_BOOLEAN_SYNTAX . . . . .	33
6.5.7	Capability.COMPLEX_PERSON_NAME_SEARCHES . . . . .	33
6.5.8	Capability.PHRASES . . . . .	34
6.6	How to implement a new Toolkit wrapper . . . . .	34
<b>7</b>	<b>Communication</b>	<b>35</b>
7.1	Messages . . . . .	35
7.2	Message content types . . . . .	37
7.3	Infrastructure . . . . .	37
7.4	The important role of the MTA in the communication between client and back end . . . . .	37
7.5	Communicators . . . . .	38
<b>8</b>	<b>The graphical user client</b>	<b>39</b>
8.1	Tools . . . . .	39
8.2	Implementing a tool . . . . .	39
8.2.1	Communication between tools and components . . . . .	42
8.3	Infrastructure . . . . .	42
8.3.1	The export service . . . . .	43
<b>9</b>	<b>Human Interface Guidelines</b>	<b>45</b>
9.1	The desktop . . . . .	45
9.1.1	The title bar . . . . .	46
9.1.2	The menu bar . . . . .	46
9.1.3	The status bar . . . . .	47
9.1.4	The tool bar . . . . .	47
9.2	Tools . . . . .	47
9.2.1	Menu items . . . . .	47
9.2.2	Schwebende Werkzeuge . . . . .	48
9.3	Informationsobjekte . . . . .	48
9.4	Elemente . . . . .	49
9.4.1	Eingabe-Elemente . . . . .	49
9.4.2	Labels . . . . .	49
9.4.3	Gruppierungen . . . . .	50
9.4.4	Buttons . . . . .	50

<i>CONTENTS</i>	5
9.5 Nutzereingaben . . . . .	50
9.5.1 Eingabefelder . . . . .	50
9.6 Steuerung . . . . .	51
9.6.1 Dialoge . . . . .	51
9.6.2 Drag-n-Drop . . . . .	51
9.6.3 Kontextmens . . . . .	51
9.6.4 Programmweite Tastaturbefehle . . . . .	51
<b>10 Tools</b>	<b>53</b>
10.1 Wrapper Toolkit . . . . .	53
10.2 . . . . .	53



# Chapter 1

## Introduction



# Chapter 2

## Building

Before changing anything in the code, you will want to make sure that the code in your repository actually builds and passes the tests. This chapter will walk you through building the project.

EzDL is built using Maven. This offers two different options: using Maven on the command line and using it in Eclipse through a plugin. The following section introduces building using the command line. The usage of the Eclipse plugin is demonstrated afterwards.

### 2.1 Maven on the command line

Once you have the ezDL project checked out in your file system, you are ready to build it.

The build process is fully automated. If everything works as expected, Maven will first download resources that the build process or ezDL itself depend on („dependencies”). Then the system is built and tested. To start the process, open a shell (command line interface), change into the ezDL project directory and type the following<sup>1</sup>:

```
$ mvn clean install
```

The process will take a few minutes—mainly because of the tests.

---

<sup>1</sup>In this and the following examples, „\$” is the command line’s prompt and is not to be typed.

## 2.2 Maven in Eclipse

To build the project in Eclipse, the Maven plugin has first to be installed and the project has to be configured to use Maven properly. These steps, outlined in the following two sections, have only to be performed once per freshly cloned ezDL project. The build process, described in section 2.2.3, can be performed repeatedly without repeating the first steps.

### 2.2.1 Preparations

Install a Maven plugin for Eclipse using the „Install New Software” option in the „Help” menu.

Make sure that the plugin is configured to allow for nested modules. This option can be found in the Preferences under „Maven” and is named „Support multiple Maven modules mapped to a single Eclipse workspace project”. You can use the Preferences dialog’s search feature to find the option by entering a few keywords of that phrase.

### 2.2.2 Configuring the project

After installing and configuring the Maven plugin, the first step is to configure the Eclipse project to use Maven. Then we import the submodules as Eclipse projects then we are ready to build. The steps are the following:

1. Right-click on the ezDL project and select „Maven→Enable Dependency Management” and wait until Eclipse is idle again.
2. Right-click on the ezDL project and select „Import→Existing Maven Projects”, select all submodules and wait until Eclipse is idle again.
3. Right-click on the ezDL project and select „Maven→Update Project Configuration” and wait until Eclipse is idle again. In some cases you have to refresh the whole project and afterwards again select to „Update Project Configuration”. This step might take some time because Eclipse changes the layout of the project entirely and starts a build process.

Important: The text encoding should be set so „UTF-8” and the line breaks to „Unix” in the properties dialog of the project.

The developers of ezDL agree on a coding style that is supposed to make the code clearly readable and independent from the taste of the individual developer. Also, this makes patches and diffs (and thus, merging) much easier.

To make formatting conforming to the coding style easier, two Eclipse configs are shipped with the project.

In the Preferences, go to „Java→Code Style→Formatter”. Click on ”Import”. Navigate to your project and go to the sub-directory ”eclipse”. Select ezDL-code-formatting-eclipse.xml. Then click „Apply”.

Next, go to „Java→Code Style→Clean Up”. Click on „Import”. Navigate to your project and go to the sub-directory `eclipse`. Select ezDL-cleanup.xml. Then click „Apply”.

Now you can auto-format your code usign Ctrl-Shift-F and clean up by Ctrl-Shift-O.

### 2.2.3 Building

To actually build the project, right-click on the project and select „Run As→Maven install”.

Actually, there are a few different things to chose from the „Run As” context menu:

**Maven clean** will remove all generated code and binaries from the Maven project.

**Maven package** will build the whole project and test it but omit certain checks.

**Maven install** will perform several checks (e.g. for the right license headers in each .java-File), build the whole project and test it.

When things look dicey and you want to be sure that there aren’t any old binaries interfering with your new changes, you should perform a „Maven clean” before proceeding with the build.

There are versions of the Maven plugin that are not very well integrated into Eclipse. This may lead to problems after building the Maven project such as classes or resources not being found when starting test cases right out of Eclipse (e.g. using „Run as→JUnit Test”). In these situations you should rebuild the project in Eclipse, too, using the options provided by

the „Project” menu in Eclipse. This can be done by switching to auto-building the project (this is the default) and performing a clean, resulting in an automatic rebuild. After this you should be able to run and debug the code in Eclipse.

# Chapter 3

## How we code

Please be aware that there are two target audiences for your code. The one that probably first comes to mind is the compiler since code is the only way to communicate with it. The compiler's requirements concerning the code are very strictly defined but easy to follow: the syntax has to be right. The compiler doesn't care about much more. The second target audience, though, is much harder to write for: it's the next guy responsible for maintaining your code. This guy can also easily be yourself. Especially in long-term programming efforts like ezDL programmers frequently revisit their own code months after writing it and get angry at the stupid person how wrote that illegible piece of nonsense. The only way to get around that is writing for two target audiences at once: the compiler and human beings that have no knowledge about the code and the rationale behind it for what ever reason. This means that on the one hand code should communicate clearly what it is supposed to do and obey a common writing style to reduce the noise that would arise if each programmer used only her own sense of elegance.

Therefore, just about every software project has its own style of how code is written—that is, if you count total chaos as a style. In ezDL we have a set of formatting rules and some writing guidelines we try to follow. The formatting rules are stored in an Eclipse config file file that is shipped with the project. See section 2.2.2 for details. The question how the code should communicate with fellow developers are higher-level guidelines that cannot be externalized to a file very well. This has two consequences: One is that there is (currently) no way to automatically transform badly written code to good code. The other one is that there is also (currently) no good way to automatically recognize either bad or good code.

That means that on the other hand programmers have to follow some rules when writing the code and that other programmers have to make sure that the code complies to the standards. But what exactly are those standards? We try to summarize them in this chapter by starting with badly written pieces of code and show how to improve them so they would pass a code review.

### 3.1 Write self-documenting code

Every programmer loves self-documenting code because if the code documents itself the programmer doesn't have to. Right? Wrong. Self-documenting code is not code that the reader has to figure out by herself. "Self-documenting" means that the programmer uses techniques to document the way the code works that exclude inline comments.

Inline comments are sometimes used when code isn't really obvious at first glance. Like this:

Listing 3.1: selfdoc1.java

```

1 public Object getStructure(Document document, Map<String, Object> props) {
2     ...
3     int cmp = input.length();
4     if (cmp > 10000) {
5         System.out.println("This_page_is_very_large_Structuring_"
6             + "will_take_at_least_" + ((cmp / 10000)) + "_minutes!!!");
7     }
8     input = input.replaceAll("<!--.*?-->", ""); // remove all comments!
9     String part1 = "";
10    String part2 = "start!";
11
12    // process the input
13    while ((!input.equals("")) && (counter < 40) && (!part2.equals("")))
14        && (shift >= 0)) {
15        ...
16    }
17    ...
18 }

```

This code is taken from an early version of `ToolkitAPI.java`. Several questions arise that should be addressed. And some questions are already answered by using inline comments. Some open questions are:

- What happens to `cmp` later?
- What does the 10000 mean? Why isn't 3 used? Or 10000000?

- What do `part1` and `part2` hold? What object are these part of?
- What do the 80 lines of code (with lots of nested `if` and `for`) in the `while` loop do? To what input? What is the output?

Inspecting the code we find out that `cmp` is only used in comparisons (thus the name, we guess) as a shorthand for `input.length()`. So we rename the variable properly and use the `final` modifier to signify that the variable is never modified<sup>1</sup>:

Listing 3.2: selfdoc2a.java

```

1  final int inputLength = input.length();
2  if (inputLength > 10000) {
3      System.out.println("This_page_is_very_large..Structuring_"
4          + "will_take_at_least_" + ((inputLength / 10000)) + "_minutes!!!");
5  }

```

Now we address the 10000. Analyzing the output, the 10000 might be a constant with the dimension  $min^{-1}$ , holding how many characters of input per minute the code can process. Since this value is fixed, we give it a name and use it:

Listing 3.3: selfdoc2b.java

```

1  final int inputLength = input.length();
2  final int charsPerMinute = 10000;
3  if (inputLength > charsPerMinute) {
4      System.out.println("This_page_is_very_large..Structuring_"
5          + "will_take_at_least_" + ((inputLength / charsPerMinute))
6          + "_minutes!!!");
7  }
8  }

```

Please note that this code still has the problem of having the speed wrong on most machines. If the estimate was really important, you would want to calculate the speed first.

<sup>1</sup>The `final` modifier is used to make the code clearer. It also saves a few CPU cycles under certain conditions but this is just a nice side-effect here. The following steps towards better legibility make the code less efficient—e.g. by computing things that might not be needed. So there is a trade-off between legibility and performance. To quote Donald E. Knuth: “Premature optimization is the root of all evil”. You should first make sure that your code can be maintained and bug-fixed easily. The standard here is both other programmers and you in three months. If the code seems correct but slow, look for performance bottlenecks and optimize them when you know where they are. Just optimizing stuff to death that might be executed only twice per decade doesn’t help performance-wise and makes the code less easy to understand.

But the code is still not well documented. The comparison in the `if` clause is strange. The input size is compared to a speed. That doesn't sound right. What is meant instead, is to calculate how many minutes the code needs to process the input. If that time is longer than a minute, the warning should be printed. We notice that this calculation is actually done later in the `println` statement. So we reformulate the code a bit to get it right:

Listing 3.4: selfdoc2c.java

```

1   final int inputLength = input.length();
2   final int charsPerMinute = 10000;
3   final int processingTimeMinutes = inputLength / charsPerMinute;
4   if (processingTimeMinutes >= 1) {
5       System.out.println("This_page_is_very_large_Structuring_"
6           + "will_take_at_least_" + processingTimeMinutes + "_minutes!!!");
7   }
8   }

```

The name of the new final variable, `processingTimeMinutes` combines a name for the content and the unit of the value. If `Minutes` was omitted here, the reader would have to analyze the code in order to find out if the calculation is about seconds, minutes, hours, days or years. It is a good habit to use the unit as a part of the name of a variable—especially for private static fields.

On to the next snippet:

Listing 3.5: selfdoc3a.java

```

1   public Object getStructure(Document document, Map<String, Object> props) {
2       ...
3       input = input.replaceAll("<!--.*?-->", ""); // remove all comments!
4       ...
5   }

```

Nice comment. But what if the programmer later finds out that this doesn't suffice to remove all comments?

Listing 3.6: selfdoc3b.java

```

1   public Object getStructure(Document document, Map<String, Object> props) {
2       ...
3       input = input.replaceAll("<!--.*?-->", ""); // remove all comments!
4       input = input.replaceAll("^//.*$", ""); // remove line comments!
5       ...
6   }

```

And it gets uglier. Unless the code is really time-critical a nicer way to do it is this:

Listing 3.7: selfdoc3c.java

```

1 public Object getStructure(Document document, Map<String, Object> props) {
2     ...
3     input = removeAllComments(input);
4     ...
5 }
6
7 String removeAllComments(String input) {
8     input = input.replaceAll("<!--.*?-->", ""); // remove all comments!
9     input = input.replaceAll("^//.*$", ""); // remove line comments!
10    return input;
11 }

```

Now, reading `getStructure()` is easier. It is clear that comments are removed from the input but the exact way is no longer important. Please note that the new method, `removeAllComments()`, is package visible to help with unit testing but not public since it is not part of the outside view of what the object does.

The code can be further restructured if the inline comments in `removeAllComments` are not wanted:

Listing 3.8: selfdoc3c.java

```

1 String removeAllComments(String input) {
2     input = removeXMLComments(input);
3     input = removeLineComments(input);
4     return input;
5 }
6
7 String removeXMLComments(String input) {
8     input = input.replaceAll("<!--.*?-->", "");
9     return input;
10 }
11
12 String removeLineComments(String input) {
13     input = input.replaceAll("^//.*$", "");
14     return input;
15 }

```

What is nice about these new methods is: they can both be tested separately and they can be commented on the method level, giving information on the inputs and outputs and maybe preconditions and postconditions. Using commented methods also shows the scope of the comments. If there are 100 lines of spaghetti code with a comment like “calculate foo” at line 20, nobody would know how many lines are needed to calculate foo, unless the following code is analyzed further. Moving the calculation to a new method and commenting it makes it all clearer and the calculation is even reusable in other methods of the class. This kind of JavaDoc is also way better supported by IDEs such as Eclipse.

There is another part up for being edited:

Listing 3.9: selfdoc4a.java

```

1 public Object getStructure(Document document, Map<String, Object> props) {
2     ...
3     // process the input
4     while ((!input.equals("")) && (counter < 40) && (!part2.equals(""))
5         && (shift >= 0)) {
6         ...
7     }
8     ...
9 }

```

The while loop. As mentioned above, the code inside the while loop is 80 lines long and deeply nested. 80 lines of nested code is often a warning signal. The whole while loop should be moved into a new method that has a name that documents what the loop actually does. But even then we have a problem: the condition in the while loop is not easy to get. What does it mean that input equals the empty string? Why should counter stay below 40? Why not 42? And so on.

To get around that, we can use the above tactic of giving literals symbolic names and rename variables so that it is clear what they are about.

Listing 3.10: selfdoc4a.java

```

1 public Object getStructure(Document document, Map<String, Object> props) {
2     ...
3     // process the input
4     final int maxIterations = 40;
5     final int minNestingLevel = 0;
6     while ((!input.equals(""))
7         && (iterations < maxIterations)
8         && (!part2.equals(""))
9         && (nestingLevel >= minNestingLevel)) {
10        ...
11    }
12    ...
13 }

```

If loop conditions are too complex, it might be better to refactor the whole condition into a new method and document the method. Sometimes, a complex loop condition is a sign for the loop being improvable by simplification.

At this point we still don't know about the reasons for the parts of the conditions that compare against the empty string<sup>2</sup> but we leave that as an exercise to the reader.

---

<sup>2</sup>something that should be done with at least `String.length()` or better `String.isEmpty()` depending on the Java version

### 3.1.1 Guidelines

After giving some examples on how to solve particular problems, we'd like to give you a list of guidelines that should lead to more comprehensible code.

**Use sensible names.** This starts with using nouns for classes and verbs for methods. It also means that variable names such as “foo” and “bar” or “i” should be avoided even if they show that you probably know your fair share of hacker culture. Variables like “i” can be used in `for` loops if it is absolutely clear what the variable stores. When choosing a variable name, try to make sure that the name you're going to use is actually a good representation of the named entity. E.g. a method that adds an item to a list as long as that item has a certain property (like not being null) should not be named `add` because the user might believe that also null references can be added. A name like `addIfNotNull` is much clearer, if longer, and leads to fewer misunderstandings when it comes to corner cases of input. Other examples for bad naming are `getDate` when the method returns a string representation of a date and ..

**Use methods instead of commented code blocks.** If you have a long method that performs several steps on some input, consider breaking the steps down into new private methods and call them. That way you can get around commenting code (with the inherent ambiguity of not saying where the code stops that the comment is about) and have a clearly defined interface.

**Make sure each entity is about exactly one thing.** Classes should deal with one entity or one kind of behaviour. Having classes perform different things like searching for documents and retrieving document details depending on a part of the input is difficult. It's difficult to read because the class might get large, it's difficult to extend, it's difficult to remove behaviour because nobody knows which properties are affected and so on. Instead write one class to search and one class for the detail retrieval. If you don't need the detail retrieval (or want to experiment with a different way to do it) you can delete the class or rewrite it.



# Chapter 4

## Architecture

The architecture aims for good object-oriented design. Part of that goal is the decoupling of components. In ezDL this is done by using message passing for communication between larger components (e.g. between agents in the back end or between the client and the back end) and even between smaller ones (like tools in the front end). See figure 4.1 for a diagram of the top-level view. On the right side there is the back end that does the grunt work in ezDL like searching remote digital libraries for documents. On the left side is a client application that communicates with the back end through a Message Transfer Agent (MTA). An MTA serves as an interface between clients and the back end and hides the inner workings of the back end from the client. A client asks the MTA for documents that have “algorithm” in the title and the MTA answers with a list of such documents. How the MTA does that is not of interest to the client.

How the back end performs the tasks (currently) can be seen in figure 4.2. On the left are the MTAs that make the system border. The long box at the bottom is a communications infrastructure like a CORBA ORB or a JMS broker. The other boxes are agents. Agents are software systems each of which runs in their own JVM and is responsible for one well-defined thing or family of things. The user agent, e.g., is the place where information about users is stored: their login credentials, their preferences, and so on. The search agent searches, the repository agent stores documents and the query history agent stores query histories. The directory agent is special because it facilitates communication between agents by knowing which agents are available and what kind of service they offer. Agents can ask the directory for the name of an agent that offers a specific service. This is the reason why

starting the back end is performed in the following order:

1. Communication infrastructure (e.g. JMS broker)
2. Directory
3. Agents and wrappers
4. MTA's

The MTA's are started last because otherwise they would offer an interface to a system that is not yet ready for serving requests.

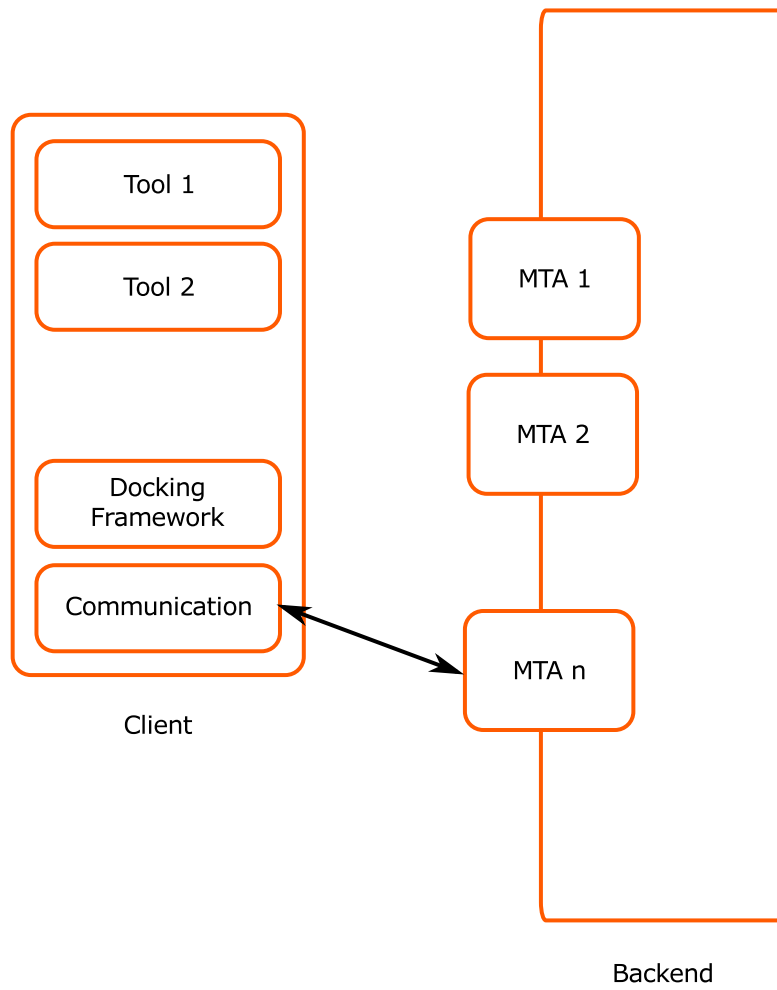


Figure 4.1: The architecture of ezDL on the highest level

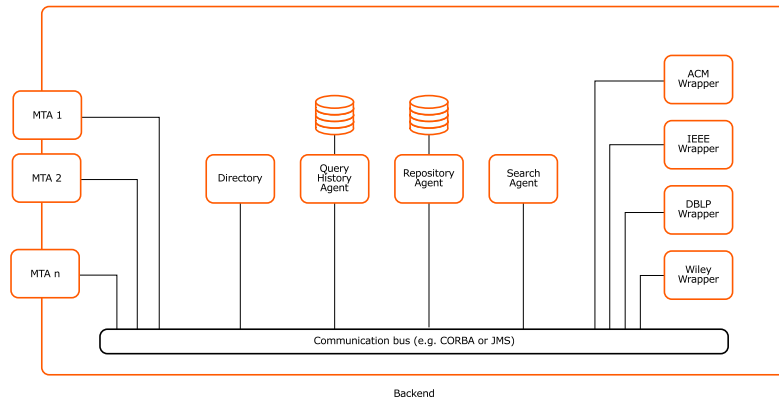


Figure 4.2: The architecture of the ezDL back end

# Chapter 5

## Agents

### 5.1 General architecture

The active components in the back end operate as agents, meaning they are processes that communicate with each other to fulfill their tasks. The communication is realized using a common mechanism that is beyond the scope of this section and, basically, beyond the scope of the agents themselves because the communication is hidden behind interfaces. This allows developers and operators to use the middleware that suits them best. Available and implemented middlewares are CORBA and ActiveMq.

The first thing an agent does when it is started is registering with the directory. The directory is a central agent that knows about all the other agents and their capabilities.

The registration message contains the name of the agent, the name of the service that it implements and possibly more information. If the agent is a wrapper (see chapter 6), the registration also contains information about the wrapped remote site and performance metrics.

The service that an agent implements is the kind of action that it performs. The service names are structured hierarchically in the form of UNIX directory names. E.g. all common services are under `/service` and wrappers are under `/wrapper`. Wrappers that wrap digital libraries are under `/wrapper/dl` and so on.

See figure 5.1 for an overview of some subcomponents of agents. At the top is the `Agent` interface that every agent has to implement. Below that is the `AbstractAgent`, a default implementation of the core behaviour of

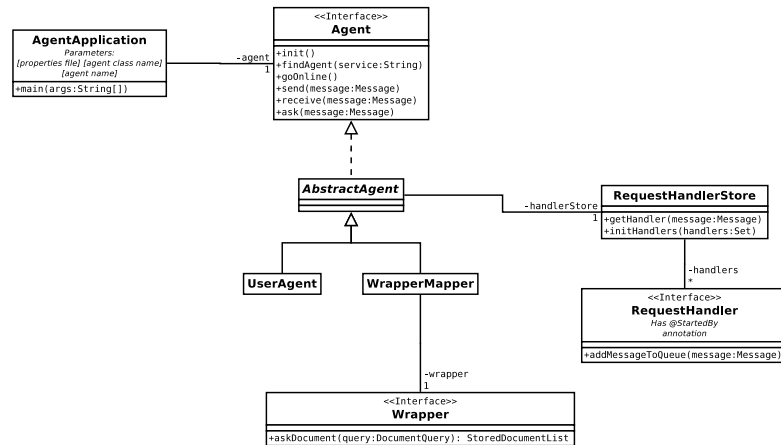


Figure 5.1: The architecture of an Agent

agents. `AbstractAgent` implements basically everything but “production behaviour”. It takes care of registering with the directory agent, handles incoming messages, sends messages, caches information about agent names and so on. The one thing that the `AbstractAgent` does not know how to do is actually handle specific messages. Messages are handles by `RequestHandler` implementors. See figure 5.2 for more details.

The class in the upper left corner is `AgentApplication`, a helper class that has a `main` method and, thus, can actually be run. `AgentApplication` takes care of reading properties files and instantiating an agent class.

An agent (read: `AgentApplication` with an `Agent` in it) can be started multiple times as long as the names of all instances are unique. It is, e.g., possible to start multiple instances of the search agent using names like `SearchAgent1` and `SearchAgent2`. Since the service name is hard coded in the source of the search agent (because the service type only changes if the source is changed dramatically), all instances will be found under `/service/search`. A client can then ask the directory agent for all names of agents that implement `/service/search` or get just one (load-balanced) name.

Agents can use the method `findAgent(String service)` to get a name of an agent that implements the given service.

There is only one agent that doesn’t implement a service and whose name cannot be resolved using the directory agent: it’s the directory agent itself.

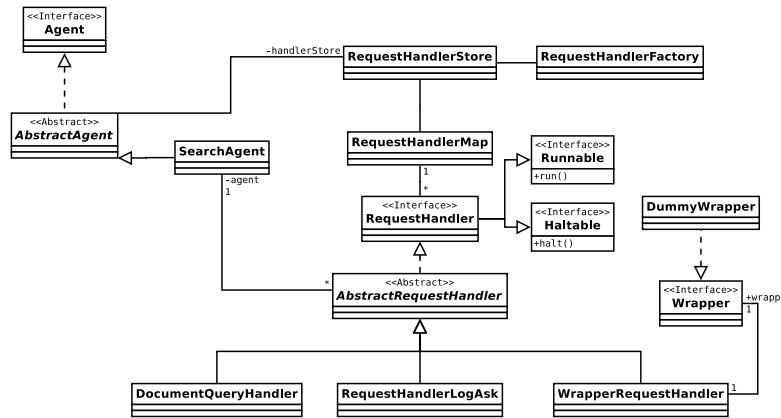


Figure 5.2: The architecture of RequestHandler

The name of the directory agent is configured using the properties files and can be determined in an agent by calling `getDirectoryName()`.

## 5.2 Requests and request handlers

In ezDL, a request is a thing that some entity in the system (the user, the client, some agent) wants to get done. Entities communicate requests by sending messages to remote parties. The receiver uses a `RequestHandler` to handle the request. As explained in section 5.1, this is done by implementing the `RequestHandler` interface. Figure 5.2 shows more details about the relationship between agents and request handlers. It also shows that there is an `AbstractRequestHandler` that can be used for implementing a request handler. The `AbstractRequestHandler` takes care of all the mundane aspects of handling requests and lets the developer concentrate on actually handling the request.

Request handlers are announced to the `RequestHandlerStore` by the `AbstractAgent`. The only thing the agent implementor has to do is override the method `Agent.setupRequestHandlers()` to return a list of request handlers that should be used. The default implementation already returns a number of handlers for common things like responding to status requests so the super implementation should be called.

The `RequestHandlerStore` has to know how to deal with a given request

handler. Since it only gets the class names of request handlers it should take care of, the classes have to be annotated. There is one annotation that is mandatory since it tells the store which message content types the handler actually handles. The other one is optional.

**StartedBy** is the mandatory annotation the store uses to determine which request handler class can be used to handle an incoming message content object. It can be used like this: `StartedBy(AliveAsk.class)`.

**Reusable** is the optional annotation that marks a class as being thread safe. If a request handler is thread safe, it can handle multiple requests concurrently in the same object. In this case the store does not have to create a new object for each incoming message but can reuse an existing object. Request handlers that are not thread safe must not be marked by this annotation. This way each incoming message is handled by a freshly created request handler that can use internal state to deal with more complex protocols like sending messages itself and waiting for answers in order to process the request.

If the request handler store handles an incoming message, it first checks if the message is about a request that is already being handled by an existing request handler. Maybe the request handler sent a message to another agent and now waits for the result. To decide this, the request ID in the message is used: request handlers are tracked by the ID of the request they are processing. So if a request handler has to talk to another agent, it should use the ID of the current request for the new message so answers are routed to the request handler that sent the request. If a new request ID were used, the remote agent would answer correctly. The agent, though, would get a `Tell` message (e.g. `DLObjectDetailsTell`). It would ask the store to find a request handler for that request. Since no handler is there to handle answers (for obvious reasons) it would check if an existing request handler is running for the same request. The request ID of the incoming message being different, no existing request handler can be found and the message is finally dropped.

## 5.3 Implementing an agent that handles a request

As outlined in the previous sections, implementing a new agent is a two step process: first the agent has to be implemented and then the request handler has to be implemented. Listings 5.1 and 5.2 show the code to get a working dummy agent that lacks any but the standard functions. In the example, the agent is implemented to announce `/service/dummy` to be the service it offers and to process `DummyAsk` messages by sending a `DummyTell` to the sender. Please note that the request handler is marked reusable by the second annotation.

Listing 5.1: DummyAgent.java

```

32 public class DummyAgent extends AbstractAgent {
33
34     @Override
35     public String getServiceName() {
36         return "/service/dummy";
37     }
38
39
40     @Override
41     protected Set<Class<? extends RequestHandler>> setupRequestHandlers() {
42         Set<Class<? extends RequestHandler>> h = super.setupRequestHandlers();
43         h.add(DummyRequestHandler.class);
44         return h;
45     }
46
47 }

```

Listing 5.2: DummyRequestHandler.java

```

29 @StartedBy(DummyAsk.class)
30 @Reusable
31 public class DummyRequestHandler extends AbstractRequestHandler {
32
33     /* <work> */
34     @Handles(DummyAsk.class)
35     public void handle(Message message, DummyAsk dummyAsk) {
36         DummyTell tell = new DummyTell();
37         Message reply = message.tell(tell);
38         getAgent().send(reply);
39     }
40     /* </work> */
41
42 }

```



# Chapter 6

## Wrappers

An important part of ezDL is the connection to data sources that—in the general case—are outside of the ezDL back end. This connection is established by so called “wrappers”. In ezDL, the term “wrapper” can mean two similar things, depending on the context. When talking about object-level architecture, a wrapper (or better: `Wrapper`) is a class that implements the `Wrapper` interface so that it can handle requests for documents. When talking about the general back end architecture, a wrapper is actually an agent (see chapter 5) that uses a `Wrapper` object to perform its intended task.

### 6.1 General architecture

Figure 6.1 shows the high-level architecture of the agents that serve as wrappers.

Figure 6.2 shows more details about how the actual wrapper mechanism is implemented.

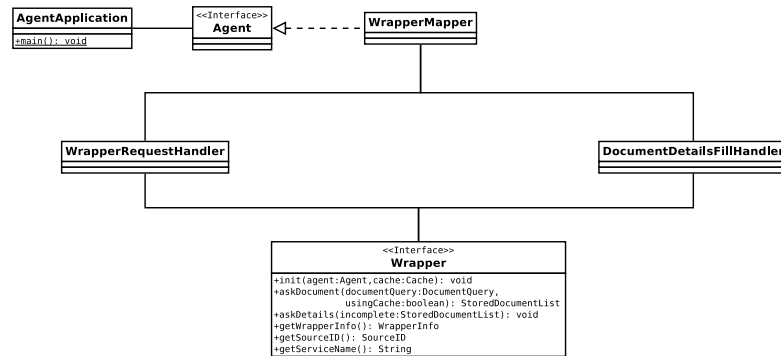


Figure 6.1: The general architecture of wrapper agents

## 6.2 Solr wrappers

## 6.3 Toolkit wrappers

## 6.4 Requirements for all wrappers

To ensure consistency of behaviour between different wrappers, all wrappers have to adhere to a common standard. This prevents the user from wondering why her query returns different kinds of results from different remote services.

**Parse completely** Wrappers have to parse pages as completely as possible in all situations without assuming that certain pieces of information are already known. In particular all the items that make OIDs (title, all authors, year) have to be parsed. E.g. a wrapper should not assume that the title has already been collected earlier because it might be that the new document has not been found using a search result page (that would have included the title) but using a direct link to that page (e.g. in the context of a citation search).

**Detail URLs** Detail URLs (`Field.URLS`) are presented to the user to enable her to visit a details page directly. These URLs have to be reduced to the part that is minimally necessary to identify a resource. I.e. the full host name has to be a part of the URL but the query string has to be cleaned from session IDs and similar information to make merging detail URLs easier/possible. Failing to do so will result in documents

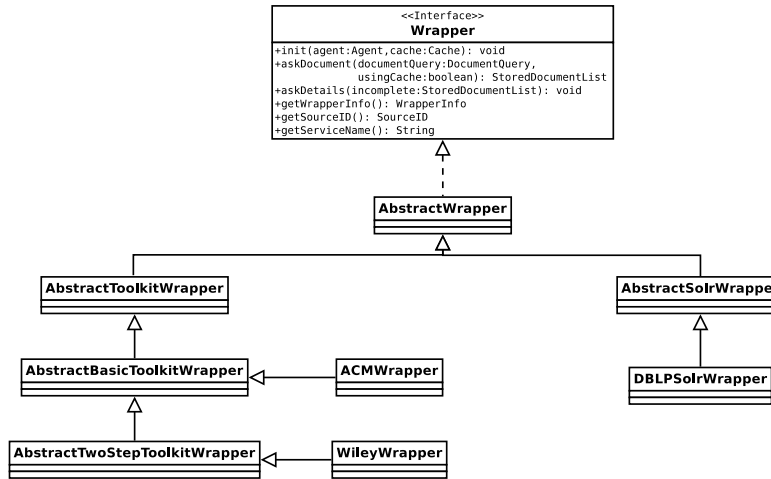


Figure 6.2: The architecture of wrappers

having multiple different detail URLs all of which actually point to the same page which is a waste of both storage space and the user’s attention.

## 6.5 Query processing in wrappers

A *Wrapper* implementation gets a *DocumentQuery* when it is asked for documents that are relevant for a certain query. The remote service (e.g. web sites or a Solr server) does not understand these query objects so they have to be translated into queries that are understood by the remote service.

To make this a bit easier and wrapper classes more legible, this process takes place in an implementation of the *QueryConverter* interface. These classes have to translate the query from ezDL—i.e. what the user can express in ezDL—to something the remote resource understands. Problems arise when there is no way to transform the original query to the target language without loss of information. This often occurs in cases where a DL offers only some basic querying features.

Handling these situations consistently is important to give the user a consistent feedback to their query. The following subsections show how each possible wrapper capability should be handled if the remote resource does not support it.

### 6.5.1 `Capability.PROXIMITY_SEARCH`

A proximity query searches for documents that contain two terms in a certain maximum distance. If this isn't directly supported by the remote resource, query the super set and filter afterwards. That means: transform the query to one that searches for documents containing the two terms in any distance (regular term search). Then use the `Filter` to throw out documents that don't fit the maximum distance requirement. Please note that querying the super set means to get a lower precision and since result sets are often limited to a fixed number of documents it also means a lower recall. If possible, use a larger maximum number of documents if you have to expand the query to return the super set to get a greater chance of finding some documents that match the original query.

### 6.5.2 `Capability.SIMPLE_YEAR_RANGE`

Some remote resources don't allow searching for a year range at all. Queries in ezDL can be arbitrarily complex and feature year parts like `<1990 OR >2005 OR 2001-2002`. If no year range search is possible, treat it like any other impossible to search for field and just drop it from the query. The results (if any) should then be filtered according to the original query.

### 6.5.3 `Capability.FULL_BOOLEAN_YEAR_RANGES`

Some remote resources only allow searching for a simple year range using a start year and an end year. In this situation the query should be broken into components for each year range defined in the query. To make this easier, transform the query to DNF and process each conjunction individually. Be aware that this might take much longer than processing the query as is: a query with three year range parts like in the above example would result in at least three different conjunctions—depending on the rest of the query. So the number of pages retrieved from the remote resource should depend on the number of conjunctions to be queried and the maximum amount of time the user is willing to wait, as indicated in the query.

#### 6.5.4 `Capability.SINGLE_WILDCARDS`

The single wildcard `$` is a place holder for exactly one character. In the interest of the user, wildcarded terms should be expanded if direct translation to the target syntax is not possible. In the case of `$` wildcards, expand cautiously to wildcard-less terms taking both statistical character frequencies and the maximum allowed query length into account. You can use the `WildcardExpander` for this.

#### 6.5.5 `Capability.MULTI_WILDCARDS`

The wildcard `#` expand to infinitely large sets of terms (place holder for zero to arbitrarily many characters). In the interest of the user, wildcarded terms should be expanded if direct translation to the target syntax is not possible. If the remote resource cannot handle a wildcarded term with a `#` wildcard it should be dropped from the query since there is no way to expand it. If the only `#` is the last character, drop it from the term and hope that the remote resource can work with prefixes.

#### 6.5.6 `Capability.FULL_BOOLEAN_SYNTAX`

If boolean syntax is not supported but only e.g. a list of query terms, the list of positive terms in the query (i.e. no negated terms) should be extracted and used. In general it might be a good idea to also convert the query into DNF and run each conjunction against the remote resource individually, collecting all results. Then the results should be filtered using `Filter` to make sure they comply with the original query.

#### 6.5.7 `Capability.COMPLEX_PERSON_NAME_SEARCHES`

If person name searches are only partially supported (e.g. searching for last names only), the query terms should be simplified and ran against the remote resource. Filtering the results must not be done based on anything but wildcard search. In other words: an ex-post filtering of the results is only legitimate if it is done to support wildcard searches such as for “`A# Author`” in cases where the remote service does not natively support wildcards.

### 6.5.8 Capability.PHRASES

Break the phrase into single terms. Then filter the returned results using the `Filter`.

In general, if possible it is a good idea to increase the number of result items that are requested from the remote resource if the query is extended in some way so the lower recall can be compensated.

## 6.6 How to implement a new Toolkit wrapper

Create new class that implements `AbstractBasicToolkitWrapper`.

Create a new test class in the test module. The test class inherits from `ToolkitWrapperTestBase`.

Implement methods.

# Chapter 7

## Communication

Communication between components and subcomponents in ezDL is done by passing messages around. This chapter introduces basic concepts and explains the message content types.

### 7.1 Messages

Messages are used to pass information in three different contexts:

- between tools in the front end
- between front end and back end
- between agents in the back end

In each of these contexts, addressing works differently. For this reason, there are three different message types available.

A message type acts as an envelope for The Actual Message Content. Actual message contents are stored in classes that implement the `MessageContent` interface, which is merely a tagging interface.

Figure 7.1 shows how the various message and message content types are interconnected.

**BackendEvent** is one of the events used by tools in the front end to communicate with each other. A `BackendEvent` is fired if a tool wants to communicate with the back end.

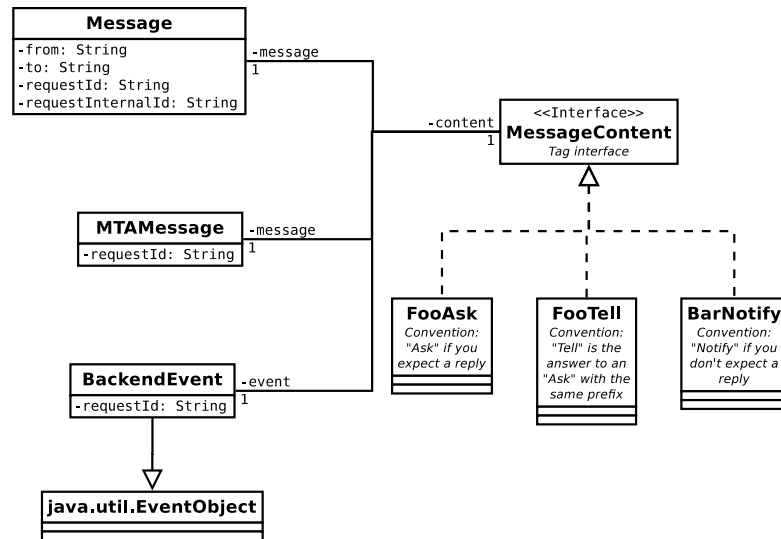


Figure 7.1: The architecture of messages

**MTAMessage** is the envelope that is used by the communication component in the front end to communicate with the back end. The name comes from the MTA (Message Transfer Agent) that is involved in the communication. Since front end and back end are connected using a 1:1 connection (currently a TCP connection), both sender and receiver are known and don't have to be specified. This makes sure that no conflicts can arise between the known parties on both ends of the connection and the specified parties.

**Message** is the envelope used by back end agents to communicate with each other.

Each of these envelope classes contain a **MessageContent** implementor that holds the actual message payload. Decoupling the payload from the addressing also helps with repackaging payloads between system borders—e.g. in the MTA that gets an **MTAMessage** object from the client and has to forward the payload using a **Message** object.

## 7.2 Message content types

There are lots of `MessageContent` types in `ezDL`. To make the protocols they are used in clearer, a naming convention is used. At the end of the class names either `Ask`, `Tell` or `Notify` are suffixed.

**Ask** is the suffix that indicates that a content of this type implies an answer to be requested. E.g. `AliveAsk`, the question “are you still there?” should be answered using a class `AliveTell` (see below under “tell”).

**Tell** is the suffix used for an answer to an `Ask` message content using the same prefix. E.g. a content type `DocumentDetailsTell` is the answer to `DocumentDetailsAsk`.

**Notify** is used to suffix message content types that are just sent in an FYI manner—no answer needed. Examples are `CancelRequestNotify`. The intent is to tell the remote party to stop working on a request because the answer is not needed anymore, so no answer is expected.

Even though this is just a naming convention it is strongly recommended to follow it.

## 7.3 Infrastructure

How are messages transferred?

## 7.4 The important role of the MTA in the communication between client and back end

The client only communicates with the MTA. So the MTA has the responsibility to deal with incoming requests. This implies that new kinds of requests (e.g. the `DummyAsk` used in the examples in this documentation) have to have a representation in all MTA’s that are supposed to act on that request.

In the `AbstractGatedMTA` there is a method that deals with forwarding incoming message content objects to agents. You’ll find the first few lines in listing 7.1. One way of making sure your new message content type is

forwarded to the right agent is to add a new `if` clause using the others as a template as seen in the last few lines of the listing.

Listing 7.1: The `transform()` method in `GatedHttpMTA.java`

```
99     private MessageRouting transform(MessageContent content) {
100         MessageRouting mt = null;
101         try {
102             if (content instanceof AvailableWrappersAsk) {
103                 mt = new MessageRouting(getDirectoryName(), content);
104             }
105             if (content instanceof DLObjectQueryAsk) {
106                 mt = new MessageRouting(findAgent("/service/search"), content);
107             }
108             if (content instanceof DLObjectDetailsAsk) {
109                 mt = new MessageRouting(findAgent("/service/repository"), content);
110             }
111             if (content instanceof DummyAsk) {
112                 mt = new MessageRouting(findAgent("/service/dummy"), content);
113             }
114         }
```

## 7.5 Communicators

# Chapter 8

## The graphical user client

In this chapter we introduce some general concepts of the user client. You will also find helpful notes on how to develop tools (see section 8.1) and how to use the rich set of features already in the graphical client in your own tools (see section 8.3).

### 8.1 Tools

From the user's point of view, tools are collections of views (sub-windows) that perform specific tasks. Every tool has at least one view because a view is needed to interface with the user. Tools can have many views. The search tool, e.g. has the search form view, the library choice view and the result view.

### 8.2 Implementing a tool

In order to implement a tool that the user can interact with, two interfaces have to be implemented: `Tool` and `ToolView`. To speed up the development process, abstract implementations of these interfaces are available. See figure 8.1 for an overview of the architecture of tools and listings 8.1 and 8.2 for minimalistic sample implementations.

The code in listing 8.2 creates a new view that has a button with the label "test" in it.

The code in listing 8.1 creates a tool that uses the `DummyToolView`. It does not, though, react to events. If event were to be handled, the method

`handleEzEvent(EventObject)` had to be implemented accordingly.

Listing 8.1: DummyTool.java

```

33 public class DummyTool extends AbstractTool {
34
35     @Override
36     public boolean handleEzEvent(EventObject ev) {
37         return false;
38     }
39
40
41     @Override
42     public List<ToolView> createViews() {
43         List<ToolView> views = new LinkedList<ToolView>();
44         views.add(new DummyToolView(this));
45         return views;
46     }
47
48
49     @Override
50     protected IconsTuple getIcon() {
51         return IconFactory.createIconsTuple(Icons.DEFAULT);
52     }
53
54
55     @Override
56     protected String getI18nPrefix() {
57         return "tools.dummy.";
58     }
59
60 }

```

Listing 8.2: DummyToolView.java

```

27 public class DummyToolView extends AbstractToolView<DummyTool> {
28
29     private static final long serialVersionUID = 1L;
30
31
32     public DummyToolView(DummyTool parentTool) {
33         super(parentTool);
34         add(new JButton("test"));
35     }
36
37 }

```

In the state listed here, the tool and view don't do anything but sit there.

In order to have the button send a message to the backend we have to implement a suitable message content type. See listing 8.3 for a very boring content type.

Listing 8.3: DummyAsk.java

```

25 public class DummyAsk implements MessageContent {

```

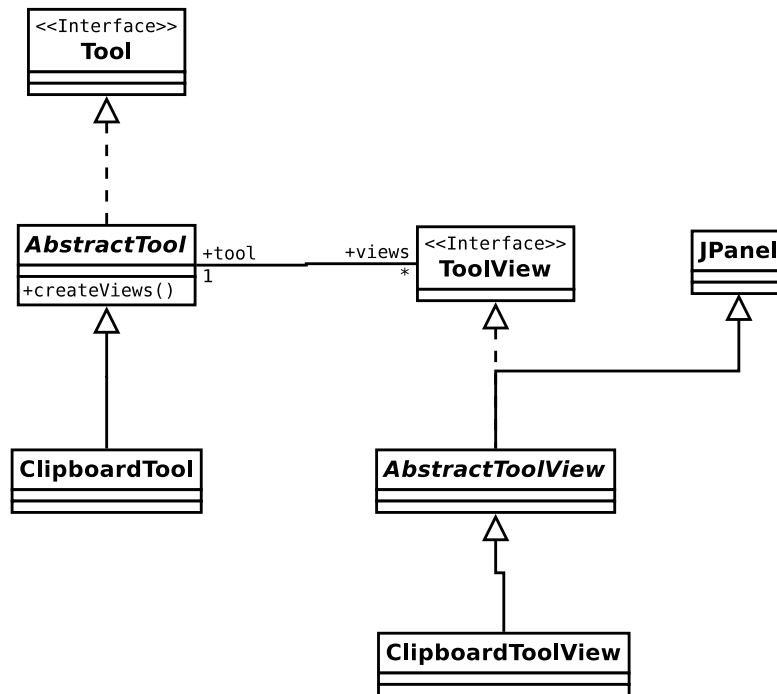


Figure 8.1: The architecture of tools

```

26
27     private static final long serialVersionUID = 1L;
28
29 }

```

We can edit the view to make it send a message to the back end when the button is pressed. See listing 8.4 for details.

Listing 8.4: SendingToolView.java that sends a message

```

35 public class SendingToolView extends AbstractToolView<Tool> {
36
37     private static final long serialVersionUID = 1L;
38
39
40     public SendingToolView(Tool parentTool) {
41         super(parentTool);
42         add(new JButton(new DummyAction()));
43     }
44
45
46     private class DummyAction extends AbstractAction {
47

```

```
48     private static final long serialVersionUID = 1L;
49
50
51     @Override
52     public void actionPerformed(ActionEvent e) {
53         String id = RequestIDFactory.getInstance().getNextRequestID();
54         DummyAsk content = new DummyAsk();
55         BackendEvent be = new BackendEvent(this);
56         be.setContent(content);
57         be.setRequestId(id);
58         Dispatcher.postEvent(be);
59     }
60
61 }
62
63 }
```

### 8.2.1 Communication between tools and components

The main paradigm that absolutely has to be followed by all tool implementations is that no tool should depend on any other tool. E.g. tools should not assume that there is a search tool and try to find clever ways of calling the search tool's methods directly. Such a design would lead to huge problems if the remote tool is replaced by a different tool or dropped entirely.

Communication between tools is done by message passing. If a tool needs services from any other tool, it has to fire an event using the dispatcher service. If there is a tool that is able to handle the event (e.g. perform a search) then everything is okay. If not, nothing will break either.

Please note that this requirement only applies to dependencies between tools. If you come to a situation where some component in a tool view needs something from the tool itself, invoking the methods of the tool is entirely good practice. To do that, just pass a reference to the tool to the component. After all, a component should know which tool it is working for. This also helps using the right tool (and thus the right tool's state) in the case that multiple instances of a tool are running at the same time.

## 8.3 Infrastructure

There are many components that the graphical client offers to tool implementors. This section will introduce the most important once and show how they are used.

### 8.3.1 The export service

In quite a few cases the user will want to use information from within ezDL outside. E.g. after a search the user ended up with a tray full of relevant documents and she wants to use these for citing in a paper. To facilitate this, ezDL offers an export function that can save information in multiple formats such as CSV, HTML, RIS and BibTex.

To use the export service, all you have to do is attach the `ExportAction` to some UI component - e.g. a button. The `ExportAction` needs a reference to a `SelectionGetter` implementation. `SelectionGetter` enables the export service to retrieve the list of items to be exported so the tool or view that has to export items has to implement the interface somehow and pass the reference to the an object of that class to the export action.

This will make the button active if exportable items are selected, inactive if not and export the selected items if the activated button is pressed.



# Chapter 9

## Human Interface Guidelines

The Human Interface Guidelines describe the structure of the elements of the ezDL user interface and common patterns of interaction.

All specifications regarding positions refer to LTR languages (e.g. English). If RTL languages (e.g. Arabic) is to be implemented in ezDL, these specifications have to be adapted according to the conventions used in the target language.

Links:

- <http://www.qalabs.com/resources/newsletters/archive/newsletter37.htm>
- <http://library.gnome.org/devel/hig-book/stable/index.html.en> - Gnome HIG 2.2
- <http://wiki.openusability.org/guidelines/> - KDE HIG
- 
- <http://msdn.microsoft.com/en-us/library/aa511440.aspx> - Windows User Experience IG
- <http://developer.apple.com/documentation/UserExperience/Conceptual/AppleHIGuidelines/XH> - Apple HIG

### 9.1 The desktop

EzDL uses a tiled windows interface. The main window (“Desktop”) is divided into individual “tiles” that contain subordinate windows and whose

size can be changed. The subordinate windows will be called “tools”. If a tile contains multiple tools the user can change between the tools using tabs. The tab of the tool in the front is highlighted.

The configuration of the desktop regarding the number and place of the tiles and the assignment of tools to tiles is defined by task-dependent “perspectives”. The user can edit and save the perspectives.

Only the desktop has the following interface elements:

- a real title bar
- a menu bar
- a status bar

### 9.1.1 The title bar

The title bar shows the name of the program (“ezDL”) and the name of the user.

### 9.1.2 The menu bar

The menu bar contains always the following sub-menus, which are named in the language that is currently set.

- File
  - Quit
- Bearbeiten Edit
  - Preferences
- Ansicht View
  - Perspectives
- Hilfe Help
  - Help about ...
  - Helptopics
  - Tutorial

– About ezDL

The menu bar contains menus of tools that are docked (see section 9.2).

In high levels of user support, menu items for advanced options are hidden by default (not disabled/greyed-out) and have to be activated to be usable.

### 9.1.3 The status bar

The status bar is located at the lower border of the desktop to the right of the tool bar. It shows all status and error messages of the program. Error messages are highlighted by an error symbol and red font color. Progress bars for time consuming actions are shown in the status bar.

The status bar has a pop up window that shows the last couple of status messages on mouse-over.

### 9.1.4 The tool bar

The tool bar is at the lower border of the desktop to the left of the status bar. It contains icons sized 24x24 for all tools that are active in the current perspective. The icon of the active tool is highlighted.

## 9.2 Tools

Tools can be detached from the desktop. They are then called “floating”. As long as they are part of the desktop (they are then called “docked”), tools don’t have neither title bar, menu bar nor status bar. Being docked, each tool has a tab with a 16x16 icon on the left side and the name of the tool.

Tabs inside of tools have to be positioned sideways.

Tools may contain multiple regions. Each region has a horizontal “label” above it. It is highlighted noticeably by a different color and describes the region or its purpose in at most three words. This also applies to tools that have only one region.

### 9.2.1 Menu items

Tools can have two kinds of menu items. When docked, all menu items of the active tool are shown as part of the desktop menu.

1. Einträge für die vier immer sichtbaren Untermens, z.B. Datei -j Drucken, Datei -j Speichern im Suchwerkzeug oder in der Detailanzeige oder Ansicht -j Einfache Anfrage, Ansicht -j Anfrageformular im Suchwerkzeug
2. ein werkzeugspezifisches Untermenü mit allen Aktionen, die sich auf das Werkzeug und nicht auf einzelne Informationsobjekte beziehen

### 9.2.2 Schwebende Werkzeuge

... (? eventuell brauchen schwebende Werkzeuge ihr eigenes Menü und ihren eigenen Statusbalken ?)

## 9.3 Informationsobjekte

Als "Informationsobjekte" werden innerhalb von ezDL alle Objekte bezeichnet, die nicht Teil eines Werkzeugs oder des Desktops sind.

Werden Informationsobjekte in Listenform angezeigt, so befindet sich links neben dem Bezeichner stets ein Icon, das die Art des Informationsobjektes anzeigt.

Arten von Informationsobjekten sind:

- Dokumente (Artikel)
- Terme (Suchbegriffe, extrahierte Begriffe, Schlagworte, Klassifikationen, ...)
- Personen (Autoren, Nutzer, ...)
- Anfragen
- Zeitschriften
- Konferenzen
- Filme
- Musikstücke, Musikalben

## 9.4 Elemente

### 9.4.1 Eingabe-Elemente

- Gruppen von Eingabe-Elementen sind links- und rechtsbndig
- Untereinanderstehende Eingabe-Elemente haben einen vertikalen Abstand von 6 Pixeln

### 9.4.2 Labels

- Nicht fett ("normal")
- Endet bei Eingabe-Elementen mit einem Doppelpunkt, der direkt am letzten Wort abschliet (z.B. "Label:")

#### Kurz-Labels

- Maximal 3 Wrter lang
- Sind rechtsbndig links neben dem Element anzuordnen mit Abstand zum Element von 6 Pixeln
- Die Grundlinie ist bndig mit der Grundlinie der Texte innerhalb der Elemente

#### Lang-Labels

- Knnen lnger als 3 Wrter sein und sich ber mehrere Zeilen erstrecken
- Sollen nicht lnger als das zugehrige Eingabe-Element sein
- item Stehen ber dem jeweiligen Eingabe-Element mit 6 Pixeln Abstand
- Linksbndig zum Eingabefeld

#### Gruppierungs-Labels

- Fett
- Linksbndig

- Ohne Doppelpunkt
- Stehen innerhalb von waagerechten Linien bis zum Ende der Gruppe (“– Login —————”)

### 9.4.3 Gruppierungen

- Zusammengehörige Elemente sind zu gruppieren
- Erhalten ein Label (siehe “Labels”)
- Sollen nicht geschachtelt werden
- Erhalten keinen Rahmen

### 9.4.4 Buttons

- Buttongruppen stehen rechtsbndig in Dialogen
- In Konfigurationsdialogen stehen Buttons in der Reihenfolge “Abbrechen”, “bernehmen”, “OK”
- In anderen Dialogen stehen Buttons in der Reihenfolge “Abbrechen”, “OK”
- Buttons sollten innerhalb von Gruppen gleich gro sein
- Button-Labels sind fett und enthalten ggf. ein Icon links neben dem Text
  - Abbrechen: rotes X
  - OK: grner Haken
  - Rest siehe Icon-Liste

## 9.5 Nutzereingaben

### 9.5.1 Eingabefelder

Eingabefelder besitzen in allen Untersttzungsniveaus auer dem niedrigsten einen “Eingabehinweis”. Der Eingabehinweis wird im Eingabefeld in einer

blassen Farbe angezeigt, die deutlich von Nutzereingabe zu unterscheiden ist, und besteht aus einem kurzen, erklärenden Satz oder einer Beispieleingabe. Der Eingabehinweis verschwindet genau dann, wenn das Feld aktiv ist oder Nutzereingabe enthält.

## 9.6 Steuerung

### 9.6.1 Dialoge

- Dialoge können mit ESC abbrechend verlassen und mit RETURN/ENTER bestätigend verlassen werden.

### 9.6.2 Drag-n-Drop

Informationsobjekte können mittels Drag-n-Drop auf Werkzeuge, Werkzeugbereiche, einzelne Werkzeugelemente (z.B. ein Eingabefeld oder einen Knoten in einem Baum) oder Werkzeug-Icons in der Werkzeugleiste gezogen werden. Gltige Ziele werden hervorgehoben. Beim Fallenlassen wird die Standardaktion für dieses Werkzeug (oder den Werkzeugbereich) ausgeführt, z.B. Hinzufügen zu einer Anfrage oder Abspeichern in einem Ordner.

### 9.6.3 Kontextmenüs

Informationsobjekte besitzen ein Kontextmenü mit werkzeug- und objekt-spezifischen Einträgen. Alle Einträge des Kontextmenüs beziehen sich auf das markierte Informationsobjekt. Das markierte Informationsobjekt wird in einem nicht-aktiven, ausgegrauten Titelseintrag des Kontextmenüs angezeigt. Die Einträge des Kontextmenüs werden durch "Trenner" logisch gruppiert.

### 9.6.4 Programmweite Tastaturbefehle

- F1: Hilfe zum aktiven Werkzeug
- vielleicht aus Gnome Shift-F10 (Kontextmenü zum aktiven Objekt) und Ctrl-F1 (Popup zum aktiven Objekt) übernehmen??



# Chapter 10

## Tools

### 10.1 Wrapper Toolkit

### 10.2