Standard front page

IT-Universitetet i København

Course exam with written work

Class code (Holdkode):
SMDP

Name of course:
Model Driven Development Project

Course manager:
Andrzej Wasowski

Course with e-portfolio, link:

Thesis/project exam with project agreement

Supervisor:
Thorsten Berger

Thesis or project title:
SuperIntents

Name(s):	Birthdate and year:	ITU-mail:
1. Anders Emil Bech Madsen	311289	aebm@
2. Nikolaj Aaes	181189	niaa@
3. Niklas Schalck Johansson	121288	nsjo@
4. Anders Bech Mellson	310780	anbh@
5.		@
6.		@
7.		@

IT-UNIVERSITETET I KØBENHAVN

Model Driven Development 2013 - SuperIntents

Anders Bech Mellson, Anders Emil Bech Madsen, Nikolaj Aaes, and Niklas Schalck Johansson
{anbh, aebm, niaa, nsjo}@itu.dk

IT University of Copenhagen, Rued Langgaards Vej 7, 2300 Copenhagen S, Denmark

Abstract. Application development for the Android platform involves a mechanism called *Intents*. Intents make it possible to call other applications on a device in order to achieve certain functionality such as making a phone call. This mechanism in its current state is difficult to use for developers, as a lot of tacit knowledge is required in order to fully utilize Intents.

We propose a plugin for Eclipse that attempts to alleviate this problem by allowing developers to choose from a list of Intents and have the necessary code auto-generated without leaving their development environment to do Intent-research. The plugin was developed using Model-Driven Development (MDD) practises, not only to solve the various tasks involved but also to be able to evaluate whether MDD is a beneficial tool in development efforts of limited size. We evaluated the plugin using both qualitative and quantitative tests with application developers, measuring speed enhancements and perceived complexity.

We conclude, that the produced plugin significantly improves the speed at which Intents are used by application developers. Our experiments even allowed a developer to utilize Intents he would otherwise be incapable of without the plugin.

Keywords: Android, Intents, SuperIntents, Eclipse plugin, AST, M2M, Model-driven Development.

When developing applications for the Android platform, there is often a need to interact with other applications. For example, when an application wants to show a website to the user, it launches a browser that is installed on the device. To enable this communication between applications, the Android platform has a sophisticated feature for handling such interactions called *Intents*.

An Intent is an abstract description of an operation to be performed [1]. It allows applications to start other applications with specific parameters defined in the Intent. Applications can actively listen to certain types of Intents by exposing *filters* that the operating system will use to choose a suitable application. Alternatively they can be explicitly invoked so that they open with a specified application.

With third-party Intents, however, the issue remains how to know which features are available to the phone in question, as these vary depending on which applications are installed on the device, and which applications can be installed in order to fulfill a certain need. To alleviate this, various online resources for available Intents have been created, such as OpenIntents [11].

We claim, that Android developers currently experience difficulty utilizing Intents, and that third party registries such as OpenIntents define their Intents too loosely to be of much real value; all fields are freetext, and leave little-to-none formatted data for code-generation. Knowing what data types are accepted, what callbacks are available and what errors to catch is not always apparent and must be investigated by each individual developer.

In this scientific case study using Model-Driven Development (MDD) tools, we will attempt to alleviate this difficulty by providing a stricter and more informative definition of Intents, enabling code-generation for the Intents in question. As an added incentive, this case study also attempts to evaluate the usefulness of MDD in projects of limited size.

This report describes the implementation and evaluation of an Eclipse plugin, aptly named SuperIntents, that makes it easier to use third-party Intents from the applications listed in an external resource. The plugin allows users to choose Intents from a list in their IDE, generating and inserting appropriate code to call the chosen Intent and handling its callback.

The goal of this plugin is to minimize the knowledge each developer needs about a specific Intent in order to utilize it, thus enabling more developers to use Intents and allowing Intents to be used with increased ease and speed. This also allows developers to easily reuse code instead of writing the same implementation of Intents multiple times.

The plugin was developed using MDD, primarily utilizing Model-to-Model (M2M) transformations to achieve our various conversions. The overall methodology started with a knowledge-gathering exercise where we gained domain-specific knowledge on Intents by implementing some ourselves, which allowed us to specify the requirements and to scope our solution. We then re-engineered the Intent DSL and used that in our code generation. The produced plugin was evaluated by conducting a speed-comparison test, comparing the use of the plugin to manual implementation. To ensure the correctness of the generated code, we compared it to its specification. In order to obtain more qualitative information, test subjects were asked to fill out a questionnaire about Intents.

As such, this project contributes the following;

- The design of an improved DSL for Android Intents.
- The implementation of a plugin that enables Intent code to be auto-generated.

– A user-experiment to evaluate the plugin.
– Reflections on the usefulness of MDD in smaller projects based on the evaluation and results of this case study.

In Section 1, we will provide the necessary background information, while Section 2 will describe the implementation of the various components of the plugin. In Section 3, we go on to evaluating the benefits of the plugin, after which we will address discovered threats to validity in Section 4. Ideas for future work are discussed in Section 5. Finally, related work is noted in Section 6 and the project is concluded upon in Section 7.

1 Background

1.1 Intents

Intents are asynchronous messages which allow Android components to request functionality from other components. Intents can be used to signal to the Android system that a certain event has occurred. Other components in Android can register to this event via an *Intent filter*.

There are two ways of calling Intents; *explicitly* and *implicitly*. Explicit Intents define which activity should handle the request, using the Java class as an identifier. Implicit Intents leave it to the underlying system to find a suitable application to handle the request. As an example, the following implicit Intent tells the subsystem to find an Intent filter that reacts to the action `ACTION_VIEW` and give it an `URI` for Google as a data parameter. On an Android phone, this would display a list of possible applications that can open an `URI`.

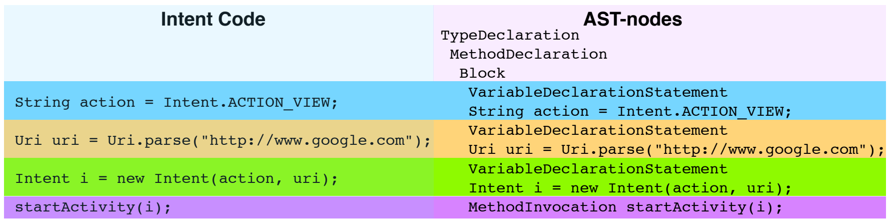
```
String action = Intent.ACTION_VIEW;
Uri uri = Uri.parse("http://www.google.com");
Intent i = new Intent(action, uri);
startActivity(i);
```

Intents can return results, which is why there are two different ways of starting an Intent; `startActivity(..)` and `startActivityForResult(..)`. When an Intent is called with the latter, its result is returned to an overriden method called `onActivityResult(..)` in the activity that calls the Intent. A result returned in the callback method can be matched against its calling Intent using an identifier. Intents contain a number of properties, which all need to be handled in the Intent-generating plugin. For a full reference of properties, see the Android Intent page [1], and supported properties by our Intent DSL in Section 2.3.

1.2 Java Development Tools' Abstract Syntax Tree

One of the most used extensions for Eclipse is the *Java Development Tools* (JDT). JDT is what turns Eclipse into a full featured Java IDE. JDT provides an API to access and manipulate the source code in the form of an *Abstract Syntax Tree* (AST) [8]. We use this API to inject our Intents as AST-nodes following an M2M transformation. We describe transformations further in Section 2.2. The internal structure of the AST can be seen using the plugin ASTView [6].

To illustrate our usage of this API we will transform our snippet from Section 1.1 into AST-nodes. We have selected vital nodes from this transformation and you can see how it maps to the original code in Fig. 1.



Intent Code	AST-nodes

	`TypeDeclaration`
	` MethodDeclaration`
	`  Block`

`String action = Intent.ACTION_VIEW;`	`VariableDeclarationStatement`
	`String action = Intent.ACTION_VIEW;`

`Uri uri = Uri.parse("http://www.google.com");`	`VariableDeclarationStatement`
	`Uri uri = Uri.parse("http://www.google.com");`

`Intent i = new Intent(action, uri);`	`VariableDeclarationStatement`
	`Intent i = new Intent(action, uri);`

`startActivity(i);`	`MethodInvocation startActivity(i);`

Fig. 1. Mapping from Intent code to AST-nodes

2 Realisation

2.1 Requirements and Use Cases

When designing the solution we established requirements and use cases for the plugin. The plugin must:

– Present a list of Intents.
– Generate code corresponding to a selected Intent.
– Enable the user to select where the generated code should be inserted.

The plugin's window shows a list of Intents. When the user double-clicks an Intent the plugin generates and inserts the code. It gets inserted into the AST node closest to the current caret offset. See Fig. 2 for a use case diagram of this scenario and Appendix B for the full list of use cases.

2.2 Overall Architecture

The solution comprises four parts; (1) an *Intent DSL* with an accompanying XMI-definition that allows the user to define new Intents, (2) an *Intent Loader* to load these XMI-instances into Java classes generated from the same DSL, (3) a transformation-component named *jIntent2AST* capable of transforming the Java Intents to AST-nodes. The nodes are inserted by (4) *JDTInserter*. Part 1 is shown in Fig. 4. Parts 2, 3 and 4 are shown in Fig. 3.

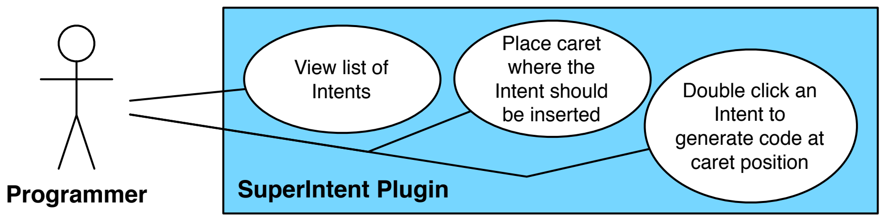

Programmer
SuperIntent Plugin
View list of Intents
Place caret where the Intent should be inserted
Double click an Intent to generate code at caret position

Fig. 2. Use case diagram using IBM notation[13]

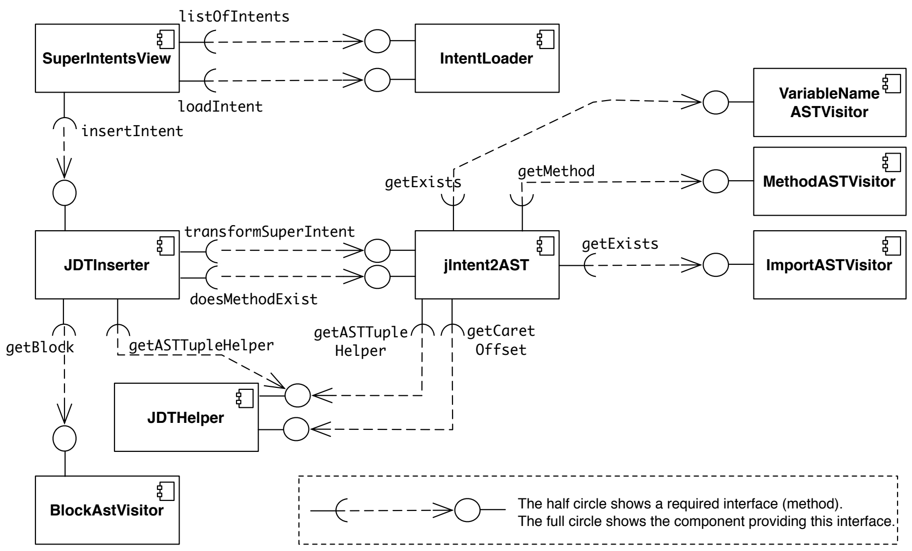


Fig. 3. Architecture Component Diagram using IBM notation[12]

Code Transformations We utilise several model transformations for this project. First, we use a Text-to-Model (T2M) transformation to convert XMI into the EMF-Generated Java Classes, which we then transform into AST-nodes using a M2M transformation. Finally, using Eclipse's built-in Model-to-Text (M2T) transformation to convert AST-nodes into actual Java code. This scheme is depicted in Fig. 5.

As depicted in the figure, we rely on new Intents being defined in XMI format, which means manual transformations from Android Intent definitions to XMI are neccesary prior to any of our transformations. Primarily, this is due to the fact that Intents from OpenIntents are loosely documented. As an example, all descriptive input fields of an Intent definition are freetext, leaving little to be automatically mapped.

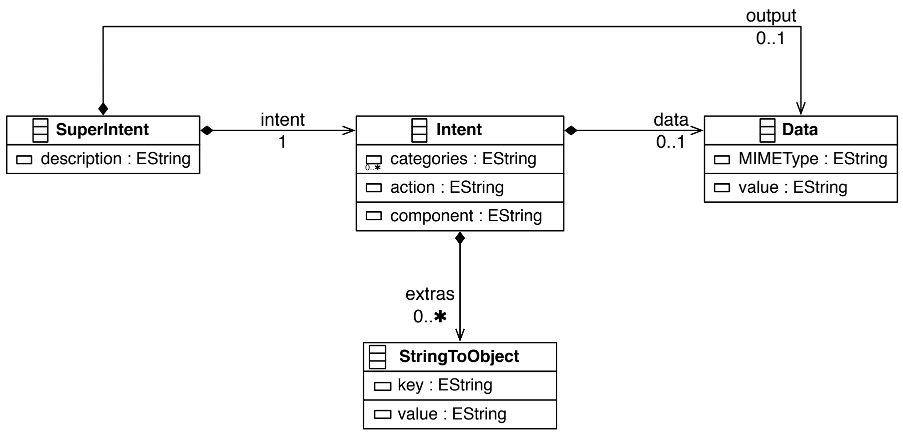


Fig. 4. Intent model in Ecore

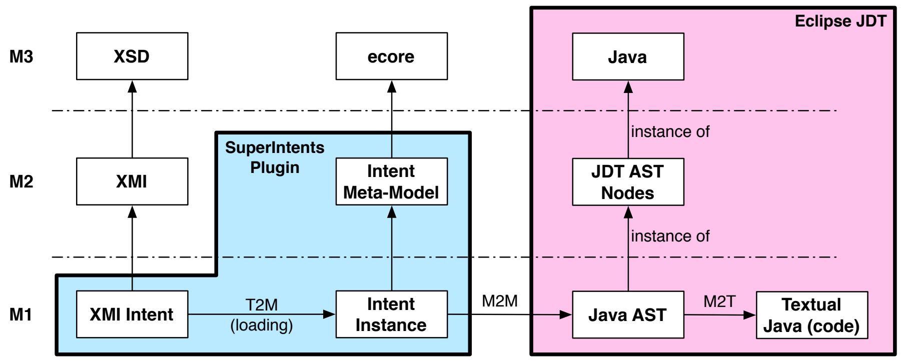


Fig. 5. Model transformations used

Manual transformations are done using the generated Ecore editor, that is generated from our Ecore model. See Fig. 4 for Ecore model. Using this editor it is possible to create `.Intentmodel` files which are concrete instances of our Ecore model. In the editor it is possible to add child nodes to a `SuperIntent`, such as `Intent` and `Data`, as well as setting the properties defined in the model. The underlying representation for `.intentmodel` is XMI. The set of all defined model instances emulate a local repository of Intents. Ideally this information would be retrieved from a database or webservice.

2.3 The Android Intent DSL

For the purpose of this project, an Ecore model of the Android Intent API was created, see Fig. 4, to provide us with an overview of Intents as well as provide us with a simple way of defining Intents from which code generation can be performed. The purpose of the DSL was to create a better structured Intent definition, which would enable code-generation.

At the top level of our model we have the `SuperIntent` class. Even though this class is not strictly necessary to model an Intent it serves two purposes: adding a description and connecting an Intent to the expected output. The description property makes it possible to elaborate on Intents that have e.g. similar names, where the difference is not obvious such as `Call` and `Dial`. The output makes it possible to check if the callback `onActivityResult(..)` is necessary when generating the code, as well as being specific on the return type. The Intent itself is modeled to exactly represent how any given Intent is created in code. Our Intent has five properties; `data`, `extras`, `action`, `categories` and `component`. `data` and `extras` are used to provide an Intent with input before it is started. `extras` is represented as a `Map` in Java, and as such, is represented as a dictionary in our model. `action` is used to specify exactly what the Intent should do, once it is started. `component` is used to specify an explicit Intent's target. `categories` is used to specify further information about the action to be executed.

The model allows us to generate Java representations of the domain, which in turn can be transformed into AST-nodes, a process which is elaborated in Section 2.5.

2.4 Defining Intents in XMI

The Ecore model created for this project is represented in an *XML Schema Definition* (XSD). XMI can use the XSD to validate specific instances of our Intent model. We use our model to generate Java classes and an editor for concrete instances of the model. Using the generated editor we can create specific instances in concrete syntax. We use the description of Intents at OpenIntents which is easily translated into concrete syntax in our editor. The instances are automatically validated against our model, and saved in an XMI format with an `.intentmodel` extension.

Whenever we need the Java representations of these instances we instantiate the XMI as an `.intentmodel` and from that create a `SuperIntent` Java object. Thus the process from the XMI format to `SuperIntent` Java objects required almost no code, and was done with almost no knowledge of the intermediate representations.

2.5 From Java Intents to AST

To transform instances of the EMF-generated Java classes into nodes that the JDT AST [5] can understand, we implemented our own M2M transformer in Java. The general idea was that each of the attributes in our EMF metamodel would translate into one or more lines of code and this code would be dependent on what attributes were available for each SuperIntent. Since JDT already provides an API for generating these AST-nodes as well as interacting with the Eclipse JDT AST, we chose to generate the JDT AST-nodes directly from Java code instead of using a modelling framework.

As a alternative, we could have used tools like JaMoPP [10], which has the ability to transform Java source code into EMF models and vice versa. JaMoPP combined with EMFText [7] could provide the necessary framework we needed to convert our own EMF generated Java classes into valid Java code.

Another alternative would have been to use JaMoPP in conjunction with ATL [3] that can perform M2M transformation between EMF models. By utilising this, we could have been able to transform the `.intentmodel` instances directly into valid Java code.

While using both of these approaches would have provided us with a more model-driven way to transform our initial EMF metamodel into Java code, they would only provide an M2T transformation. Even if the tools could provide a way to generate AST-nodes, we still needed Java code to actually inject these nodes into the AST of the current editor.

2.6 SuperIntents Plugin

We have chosen to realize our solution as a plugin for Eclipse. The plugin consists of a view, an Eclipse window, some internal storage, and model transformers. The plugin-view has a list of Intents which we support from OpenIntents. Eclipse has a project template for making a plugin, which we have built our plugin upon.

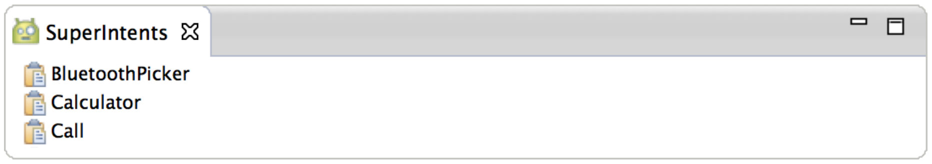


Fig. 6. Screenshot of SuperIntents view in Eclipse

Our plugin works inside a Java Text Editor, in the view depicted in Fig. 6. The user can double click on an Intent in our plugin view, which will generate the code needed to use that Intent. The plugin uses the caret's placement inside the editor to find the right insertion point in the AST. We perform AST-node insertions instead of direct text. This will make it easier to maintain, if something like the syntax of the involved languages should change.

3 Evaluation

To evaluate our claims we conducted a user test. The subjects performed a series of test cases that involved using our plugin to implement certain Intents. We measured the speed of this implementation and had the subjects fill out a questionnaire, in order to obtain some qualitative results.

3.1 Usage Experiment

A number of four test subjects of roughly equal Android-programming competency were chosen. They would all first be using the plugin to utilise certain Intents, and then use the conventional approach for the same tasks. This allowed us to compare the speed and relative ease of both methods. The test cases were considered successful if the Intents in question was implemented. The test cases can be found in Appendix A.

The first test case asked the subject to implement the `Call` Intent and had the purpose to evaluate whether the subjects could use the basic functionality of the plugin. The Intent required the user to alter the permissions in the Android Manifest file and this was mentioned in a comment in the generated code. The second test case asked the subject to implement two implicit Intents, `Scan` and `Calculator`. This test case had the purpose to evaluate whether the callback method generation was useful to the subject.

3.2 Qualitative test

Following the usage experiments, subjects were asked to fill out a questionnaire. The questions involved their experience with Java and their general opinion of both Intents and our plugin.

3.3 Test Subjects

Our test subjects were intended to be in the *Competent* category of Dreyfus' model of skill acquisition [4], which would indicate that they had knowledge and experience with Intents, but could not simply call one without much thought.

3.4 Results

Users were first asked to complete the tasks using our plugin, after which the same tasks would be performed without the plugin. This meant that there were learning-factors to take into account. As our plugin was used first, any learning effects would only influence the speed of implementation without the plugin, which only adds additional validity to our case.

Comparing the time spent implementing task one with and without our plugin, there is a definite trend towards the plugin resulting in significant improvements in terms of speed. One user managed to perform the task marginally faster without the plugin, but this was largely attributed to the fact that he managed to obtain the entirety of the code by searching the Internet, and had learned a critical lesson on Manifest-settings by using the plugin beforehand. The remainder of our test subjects, however, experienced an increase in speed of between 55-70%. For an entire listing of results, see Appendix C.

For task two, only one user managed to actually finish the task without the plugin. He completed the task 635% faster with the plugin. Another user was unable to implement task two without the plugin due to lack of skills, and another failed due to an unrelated technical issue with the test-computer. He was, however, already significantly past his implementation time recorded with the plugin at the time of failure.

Conclusively, all users experienced a dramatic decrease in implementation time using our plugin, and some users managed to perform tasks they were otherwise unable to perform. An interesting discovery was that users seemed to learn very little about the code they were implementing when the plugin generated it, and were unable to replicate the code manually, even though they had seen it moments before. This leads to the conclusion that users might be able to perform more complicated tasks using this tool, but it also turns out to be a hindrance to their learning experience.

Users also answered a survey concerning their experience with the plugin and their general opinion of Intents. As a general rule, users found Intents to be less complicated while using our plugin, indicating that using Intents was either *simple* or *very simple*, as opposed to the majority of users claiming Intents being either *complicated* or *very complicated* to use without our plugin.

Additionally, half of the subjects said they had personally abandoned using an Intent because they did not know how to call it. All subjects said that they would consider using the plugin for their daily development. For a full reference, see Appendix D.

3.5 Evaluation of Model-Driven Development in projects of limited size

The benefits of using MDD was not apparent at the beginning of the project. Representing Intents using Java itself appeared to be the easiest and fastest solution. It simply did not seem feasible to use MDD technologies over traditional programming technologies for the small amount of work needed. However, once we settled on a plugin design, the benefits of the MDD tools became more apparent. Ecore provided a useful approach to our specific problem, by allowing us to investigate the domain through modelling. It further provided us with an editor for defining model instances, allowing us to use the model at little-to-no cost. Any alterations needed in the model would also quickly be disseminated to the remainder of the project as auto-generation came into play. In conclusion, for a small scale project like this, MDD can make sense and can provide some useful tools for building and developing robust software. Developers should, however, evaluate the overhead involved to estimate whether it will provide a substantial enough payoff in their specific case.

3.6 Benefits of Intent-Code Generation

Quality of Code By using generated code we eliminate simple typo errors and suggest a code structure to the developer that contains all the necessary code for using the Intents. The developer no longer needs to investigate whether the Intent needs a callback method, as well as manually construct unique request codes to correspond to each call.

Reuse of Code Being able to generate the Intent code also speeds up the workflow of the developers that already know how to implement Intents. Instead of having to type the same code over and over, the plugin just generates the necessary code.

4 Threats to Validity

4.1 Threats to Internal Validity

Usage Experiments There is a risk that we have misinterpreted our results and that the measurings were inaccurate.

Learning effects By subjecting our test subjects to the same test cases both with our plugin and without they could have learned how to perform the test case faster the second time.

Bias The test cases only involved three different Intents that suited our purposes for the tests. It is possible that our bias towards having good test results have led us to choose Intents that were not the most representative. This could have been countered by having more test with more Intents or having an unbiased source create the test cases.

Testing environment During our test, the subjects were all in the same room which would have increased the learning effects. While they did not directly observe each other they often thought aloud which gave pointers to the other subjects. This could have been avoided by having a more sterile testing environment.

4.2 Threats to External Validity

No actual Intent Registry in this Format Exists Since no structured registry of Intents exists, our test was conducted using instances of Intents we created ourselves based on Intents found on OpenIntents. This limits the usefulness of the plugin, as users would only have a limited amount of Intents to choose from.

Low Amount of Test-Subjects Because we have performed our tests with a low amount of test subjects, one could argue that the error margin is too large. The sample pool is simply too small to be able to generalize the results to developers as a whole.

Java/Android Experience By choosing test subjects with prior Android and Java experience we cannot claim that our plugin will help newly started Android developers.

5 Discussion

Our plugin is designed in a modular fashion as can be seen in Fig. 3. This provides the basis for expanding and exchanging modules to obtain improved or additional features.

In the current version of our plugin, we load Intents from files that are bundled inside the plugin jar file. It could load Intents from other sources, like local Android projects or an online repository. To facilitate this change only the `IntentLoader` class needs to be modified.

However our implementation is not without faults. Our Ecore model seen in Fig. 4 is too simple and do not support things like the Android Manifest file, errors and exceptions. This is because our model is based on a combination of OpenIntents and Androids Intent models. OpenIntents uses a note field to specify special needs, such as permissions in a Manifest. We have mapped this to a comment being inserted. But we could, and maybe should, have supported more in our own model. If we were to make these changes, the MDD tools we are using could help. One big advantage these tools gives us is to have our changes synchronized between models in a semi-automatic fashion.

Sadly, OpenIntents does not have an API for retrieving their Intent registry. Our definition in Xtext could provide the basis for another online repository that did have an API. Our definition could be even stronger if we implemented OCL constraints. This could enforce non-empty strings and rule out other sources of errors.

Code-completion is a common tool for any developer working in an IDE like Eclipse. Therefore, it would make sense to add this feature to a future version of the plugin. The current design would allow for easy completion of Intent names and the callback method needed by Intents returning data.

6 Related Work

In the following sections, we present related work. First, we present a paper that is relevant when developing for existing frameworks, like our plugin for Eclipse. We then present a paper that is relevant when considering the use of MDD. We finish by presenting two technologies that could have been used in this project to further promote the use of MDD, and contribute to the development of our plugin.

6.1 Framework-Specific Modelling Languages with Round-Trip Engineering

Paper by Michał Antkiewicz and Krzysztof Czarnecki [2]. They introduce *Framework-Specific Modeling Languages* (FSML), which is a Domain-Specific Modeling Language that is designed for a specific framework. An FSML consists of an abstract syntax, a mapping of the abstract syntax to the framework API, and, optionally, a concrete syntax. Round-Trip Engineering is used to keep several parts, such as a model and code, in sync. A change in one part propagates to the others. It relates to this project by dealing with ways of stricly specifying *Framework Completion Code*, which is essentially what Intent-calling code is. That way, Intent-calling code could be validated by its accompanying FSML. Even though our plugin does generate correct syntax, it is up to the definition of the XMI-representation whether it is valid Framework Completion Code.

6.2 Empirical Assessment of MDE in Industry

Paper by Hutchinson, Whittle, Rouncefield and Kristoffersen [9]. The paper seeks to provide empirical data for the use of Model-Driven engineering (MDE) in projects. The survey is carried out using a public questionnaire, and followed up by interviewing 20 respondents. The research showed that there are are substantial benefits to gain using MDE, however the success is widely dependant on technical, social and organizational factors. The research further showed that the use of MDE is often decided to be carried out on important projects, because means are not available for a parallel trial project. The research further indicated that MDE was often hindered or promoted by developers perception of MDE abstractions. Some developers had an easy time using MDE tools, others were never able to fully understand them, and most MDE tools were still considered immature. MDE required extensive training, but could be applied to both new and experienced developers. MDE was found to be a great way to quickly respond to necessary changes to an existing system, but not that great to respond to new opportunities.

The results of this paper can be related to some of the experiences achieved during this project. First, this project required extra training in the use of MDE tools . Second, members of this project had difficulty seing how MDE could aid in the development, and the benefits did not become apparent until the end of the project. Last, MDE tools were considered helpful when responding to changes internally.

6.3 JaMoPP: The Java Model Printer and Parser

JaMoPP is a framework that consists of three components; a complete Java5 Ecore Metamodel, a complete Java5 EMFText Syntax, and an implementation of Java5's static semantics analysis. We could have used JaMoPP to generate Java code from our XMI `SuperIntent` instances. The disadvantage of this is that the transformation would be a M2T transformation. JaMoPP would enable us to do a M2M transformation from our XMI instances into JaMoPPs EMF model of Java and to Java code using EMFText. JaMoPP does not contain a model of the Eclipse JDT and could not have helped us generate the AST-nodes we use in our plugin.

6.4 Atlas Tranformation Language

The Atlas transformation language (ATL) is a model transformation language that is specified both as a metamodel and as a textual concrete syntax. It enables M2M transformations between EMF models through a series of rules that define how elements are transformed to form new elements. This could have been beneficial in two ways for this project. Firstly, we could have used this in conjunction with JaMoPP instead of EMFText to facilitate the M2M transformation between our XMI instances and JaMoPPs EMF model. Secondly, we could have created our own EMF model of all the existing AST-nodes and used ATL to transform our XMI instances directly into AST-nodes.

7 Conclusion

In this report, we have presented an Eclipse plugin aimed at alleviating the difficulties Android developers face when implementing Intents. We have described the realisation of this plugin, including what transformations are done to generate Java code from Intents defined in XMI. We tested the plugin through a usage experiment. This showed that implementing Intents was remarkably faster. In some cases implementation was done more than 600% faster. The plugin also enabled subjects to utilise Intents that they were unable to implement without it. We argue, that these results were created through simplification of the usage patterns of Intents, and by auto-generating Intent-calling code.

Finally, we note that Model-Driven development in smaller scale makes sense. It provides a useful approach to a given problem by obtaining domain knowledge through modelling. Some aspects of the code such as creating a language for defining Intents was also given at little-to-no cost through the produced model.

References

[1] *Android Library on Intents*. 2013. URL: http://developer.android.com/reference/android/content/Intent.html.

[2] Michał Antkiewicz and Krzysztof Czarnecki. "Framework-specific modeling languages with round-trip engineering". In: *Model Driven Engineering Languages and Systems*. Springer, 2006, pp. 692–706.

[3] *ATL*. 2013. URL: http://www.eclipse.org/atl/.

[4] Stuart E. Dreyfus and Hubert L. Dreyfys. "A five-stage model of the mental activities". In: (1980).

[5] *Eclipse Corner Article: Abstract Syntax Tree*. 2006. URL: http://d.pr/56gM.

[6] *Eclipse JDT AST*. 2013. URL: http://www.eclipse.org/jdt/ui/astview/.

[7] *EMFText*. 2013. URL: http://www.emftext.org/.

[8] *Help - Eclipse Platform*. 2013. URL: http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FAST.html.

[9] John Hutchinson et al. "Empirical Assessment of MDE in Industry". In: *Proceedings of the 33rd International Conference on Software Engineering*. ACM New York, 2011, pp. 471–480.

[10] *JaMoPP*. 2013. URL: http://www.jamopp.org/.

[11] *OpenIntent.org Intent Library*. 2013. URL: http://www.openintents.org/en/intentstable.

[12] *UML basics: An introduction to the Unified Modeling Language*. 2004. URL: http://www.ibm.com/developerworks/rational/library/dec04/bell/.

[13] *UML basics: The component diagram*. 2003. URL: http://www.ibm.com/developerworks/rational/library/769.html.

Appendix A

Test Cases

Test case ID 1

Test case description Create an Intentaaaaaa that does not return a value when called and call the Intent. In this test case we use the Intent “Call”.

Test steps

- Open specified Android Project (CallProject).
- Implement Call Intent in method "PerformCall".
- Test implementation on device.
- If errors exist, correct them.

Related requirement(s) The plugin must be able to generate and insert the appropriate code for an Intent that does not return a value.

Pass/fail The test case is passed if the user is able to call the Intent successfully.

Remarks None.

Test case ID 2

Test case description Create two Android Intents that both return a value when called and call both of these Intents. In this test case we use the Intents “Calculator” and “ScanQR”.

Test steps

- Open specified Android Project (MultipleIntentProject).
- Implement Calculator in method "PerformCalculation".
- Implement ScanQR in method "PerformScanQR".
- Test implementation on device.
- If errors exist, correct them.

Related requirement(s) The plugin must be able to generate and insert the appropriate code for multiple Intents that returns values.

Pass/fail The test case is passed if the user is able to call the Intents successfully.

Remarks If a user is able to create and call two Intents that returns values he is assumed to also be able to do the same task with any number of Intents.

Appendix B

Use Cases

The following use cases were considered relevant for the solution:

Use case 1 Implementation of basic Intent.
Actor An Android developer.
Goal To implement an Intent without a return value.
Scenario

- Open the IDE with the plugin installed.
- Place the caret in the editor where the generated code will be inserted.
- Double click the desired Intent to generate the corresponding code.

Use case 2 Implementation of basic Intent with a return value.
Actor An Android developer.
Goal To implement an Intent with a return value.
Scenario

- Open the IDE with the plugin installed.
- Place the caret in the editor where the generated code will be inserted.
- Double click the desired Intent to generate the corresponding code.
- Examine the callback method and handle the returned value.

Appendix C

Quantitative Test Results and Notes

User	1 - Thomas	2 - Marcus	3 - Mikael	4 - Stefan
Test 1, plugin	3:42	1:21	2:08	3:38
Test 2, plugin	9:05	2:15	1:25	2:32
Test 1, no plugin	not performed	2:27	1:48	5:10
Test 2, no plugin	not performed	not completed because of test environment	9:05	failed test - gave up after 13 min

0.1 User 1 - Thomas - with plug-in

It took some time to figure out that permissions needs to be set in the manifest file. 2 mins.
Was told he needed to do one more thing. 1st intent, 3:45

Start of second intent 4:38 Got help pointing the device at a QR code. Finished 13:48 2nd intent, 9:10

Subject is unsure what the completion of task 2 is. He thinks that it is because he didn't read the description thoroughly. Thinks he has been finished 3-4 mins.

Test 1
with plugin
time 3:40
time 3:45

Test 2
with plugin
time 8:53
time 9:00

0.2 User 2 - Markus - with plug-in

Subject has problems understanding the description of the assignment. Needs to be explained what the assignment is, implementing the call intent.

Test 1
Problems with permissions - doesn't see the inserted description
Time 1 1:21
Time 2 1:22
"Distracts" (User 3 says: OK?) user 3 when he is finished ahead of user 3.

Test 2
Subject
Time 1 Delta 2:14 / 3:35 (Unsure if he is done)
Time 2 2:17

0.3 User 2 - Markus - without plug-in

Test 1
Time 1 2:27
Time 2 2:28

Test 2
User 3 tries to distract user 2 by saying he is ahead. User gets frustrated by code doing unknown things. User asks for help (6 min) Gets told it is in our interest that it takes a long time without the plug-in. User asks for help again (8 min), gets help again. Needs to stop the test.

This test has failed because of our test environment failing. Time1 not completed Time2 not completed

0.4 User 3 - Mikael - without plug-in

Test 1
Time 1: 2:08
Time 2: 1:25

Test 2
Time 1: 1:48
Time 2: 9:05

Did not notice the permission comment initially. Googled needed permission after first error on device. Knew exactly what to look for. Didn't use the permission in the code.

Had problems with Mac keyboard -> ctrl-s and ctrl-v on Win, and cmd-s. cmd-v on Mac.

0.5 User 3 - Mikael - without plug-in

Test 1
Time 1: 2:08

Time 2: 1:25

Test 2
Googled intent syntax first. Had a hard time understanding how to properly start the Calculator. Used several iterations to implement onActivityResult() to distinguish the two intent calls.

Time1: 1:48
Time 2: 9:05

0.6 User 4 - Stefan - with plug-in

Asks where do I test? Gets help. Has problems with the permissions. Sees the comment.

Task 1: 3.38 (Problems with manifest permissions. Everything else went fast.)
Task 2: 3.38->6.12

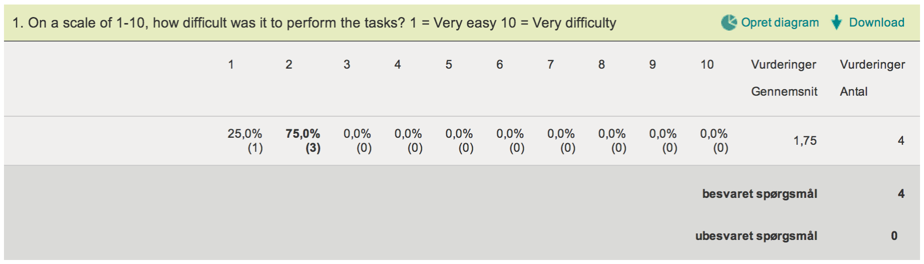
0.7 User 4 - Stefan - without plug-in

Gets told he can use the internet for help Gets frustrated because he has been working for a few minutes in a wrong project. Gets frustrated and stuck. Gets help. Gets help to do a syso. Gets frustrated and stuck again. Can't complete the test.

Task 1: 5.10 (Googles instantly, the results from the last person helps - modified permissions in Project 2... (2 min)
Task 2: 5.10->18.11 (Intent knowledge regarding action begin a string took some time - ended up giving up.)

Appendix D

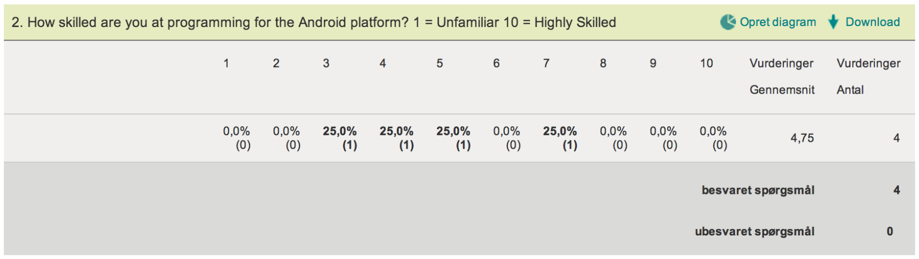
Qualitative test results



1. On a scale of 1-10, how difficult was it to perform the tasks? 1 = Very easy 10 = Very difficulty

Opret diagram Download

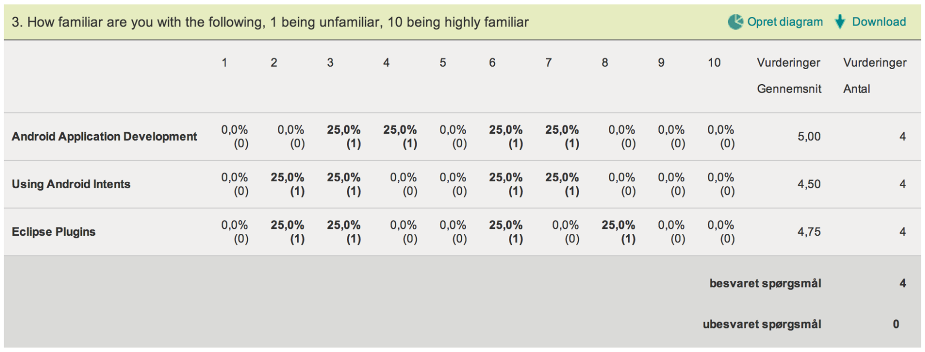
	1	2	3	4	5	6	7	8	9	10	Vurderinger Gennemsnit	Vurderinger Antal
	25,0% (1)	**75,0% (3)**	0,0% (0)	0,0% (0)	0,0% (0)	0,0% (0)	0,0% (0)	0,0% (0)	0,0% (0)	0,0% (0)	1,75	4
											besvaret spørgsmål	**4**
											ubesvaret spørgsmål	**0**



2. How skilled are you at programming for the Android platform? 1 = Unfamiliar 10 = Highly Skilled

Opret diagram Download

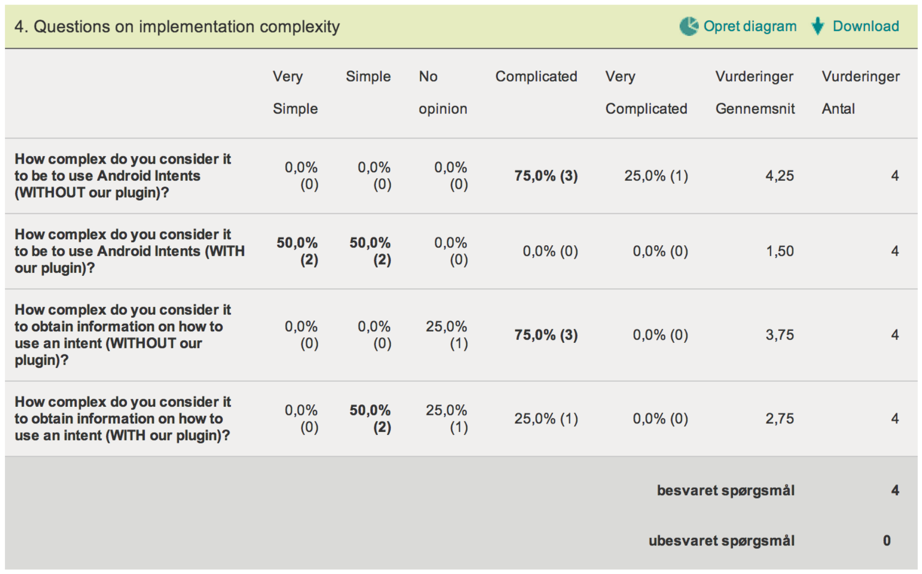
	1	2	3	4	5	6	7	8	9	10	Vurderinger Gennemsnit	Vurderinger Antal
	0,0% (0)	0,0% (0)	**25,0% (1)**	**25,0% (1)**	**25,0% (1)**	0,0% (0)	**25,0% (1)**	0,0% (0)	0,0% (0)	0,0% (0)	4,75	4
											besvaret spørgsmål	**4**
											ubesvaret spørgsmål	**0**



3. How familiar are you with the following, 1 being unfamiliar, 10 being highly familiar

Opret diagram Download

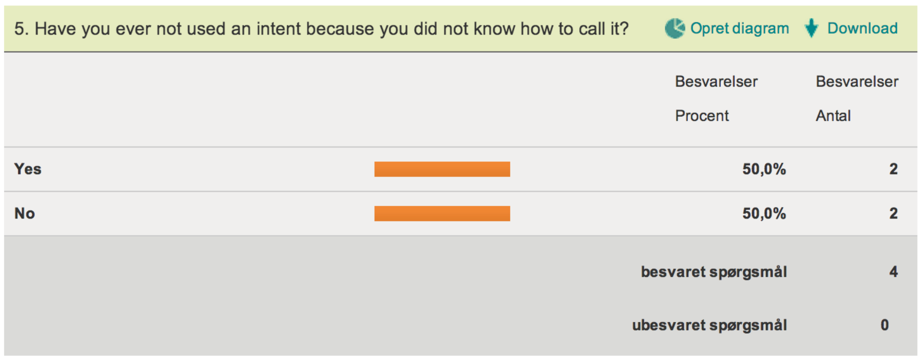
	1	2	3	4	5	6	7	8	9	10	Vurderinger Gennemsnit	Vurderinger Antal
Android Application Development	0,0% (0)	0,0% (0)	**25,0% (1)**	**25,0% (1)**	0,0% (0)	**25,0% (1)**	**25,0% (1)**	0,0% (0)	0,0% (0)	0,0% (0)	5,00	4
Using Android Intents	0,0% (0)	**25,0% (1)**	**25,0% (1)**	0,0% (0)	0,0% (0)	**25,0% (1)**	**25,0% (1)**	0,0% (0)	0,0% (0)	0,0% (0)	4,50	4
Eclipse Plugins	0,0% (0)	**25,0% (1)**	**25,0% (1)**	0,0% (0)	0,0% (0)	**25,0% (1)**	0,0% (0)	**25,0% (1)**	0,0% (0)	0,0% (0)	4,75	4
											besvaret spørgsmål	**4**
											ubesvaret spørgsmål	**0**



4. Questions on implementation complexity

Opret diagram Download

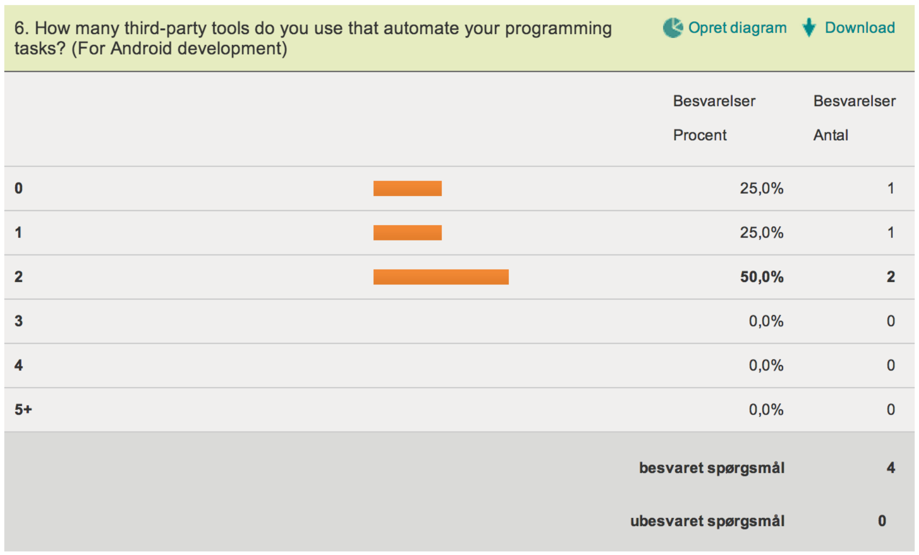
	Very Simple	Simple	No opinion	Complicated	Very Complicated	Vurderinger Gennemsnit	Vurderinger Antal
How complex do you consider it to be to use Android Intents (WITHOUT our plugin)?	0,0% (0)	0,0% (0)	0,0% (0)	**75,0% (3)**	25,0% (1)	4,25	4
How complex do you consider it to be to use Android Intents (WITH our plugin)?	**50,0% (2)**	**50,0% (2)**	0,0% (0)	0,0% (0)	0,0% (0)	1,50	4
How complex do you consider it to obtain information on how to use an intent (WITHOUT our plugin)?	0,0% (0)	0,0% (0)	25,0% (1)	**75,0% (3)**	0,0% (0)	3,75	4
How complex do you consider it to obtain information on how to use an intent (WITH our plugin)?	0,0% (0)	**50,0% (2)**	25,0% (1)	25,0% (1)	0,0% (0)	2,75	4
						besvaret spørgsmål	**4**
						ubesvaret spørgsmål	**0**



5. Have you ever not used an intent because you did not know how to call it?

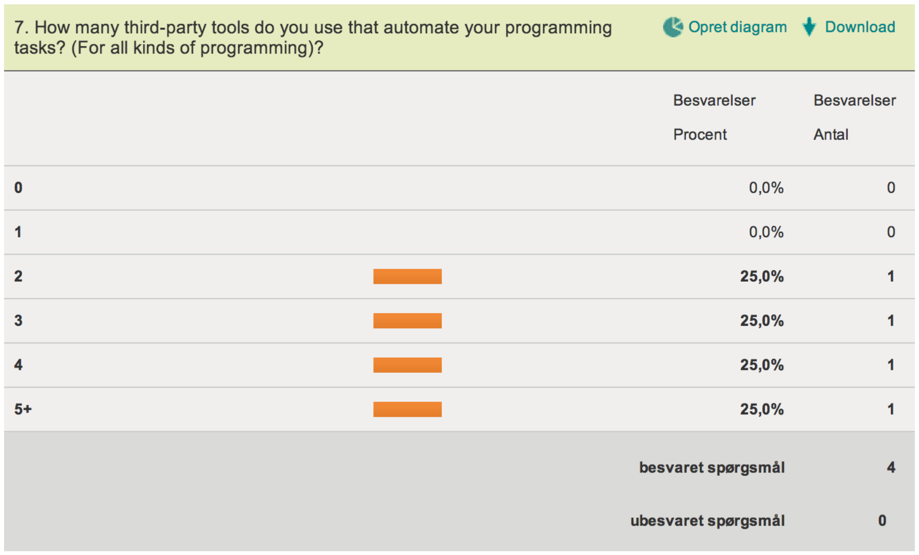
Opret diagram Download

	Besvarelser Procent	Besvarelser Antal
Yes	**50,0%**	**2**
No	**50,0%**	**2**
besvaret spørgsmål		**4**
ubesvaret spørgsmål		**0**



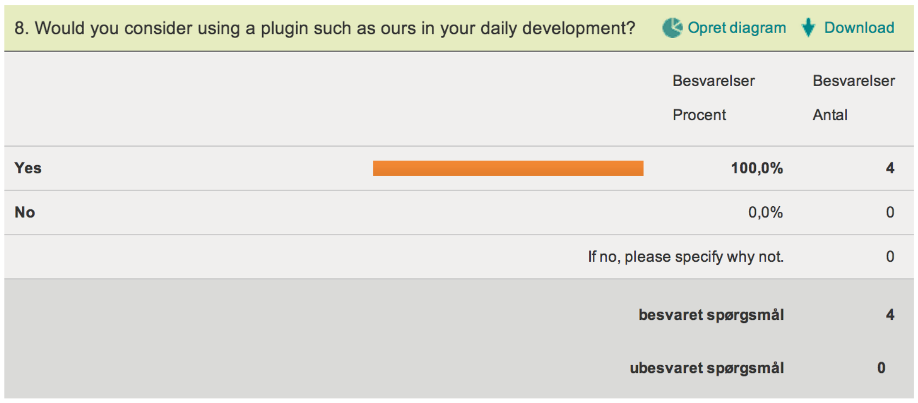
6. How many third-party tools do you use that automate your programming tasks? (For Android development)

	Besvarelser Procent	Besvarelser Antal
0	25,0%	1
1	25,0%	1
2	**50,0%**	**2**
3	0,0%	0
4	0,0%	0
5+	0,0%	0
	besvaret spørgsmål	**4**
	ubesvaret spørgsmål	**0**



7. How many third-party tools do you use that automate your programming tasks? (For all kinds of programming)?

	Besvarelser Procent	Besvarelser Antal
0	0,0%	0
1	0,0%	0
2	**25,0%**	**1**
3	**25,0%**	**1**
4	**25,0%**	**1**
5+	**25,0%**	**1**
	besvaret spørgsmål	**4**
	ubesvaret spørgsmål	**0**



8. Would you consider using a plugin such as ours in your daily development?

Opret diagram Download

		Besvarelser Procent	Besvarelser Antal
Yes	▬	**100,0%**	**4**
No		0,0%	0
		If no, please specify why not.	0
		besvaret spørgsmål	**4**
		ubesvaret spørgsmål	**0**