Scripting an IDE for EDSL awareness

Ilya Sergey
DistriNet
Katholieke Universiteit Leuven
Belgium
ilya.sergey@cs.kuleuven.be

Peter Gromov
JetBrains Inc.
peter@jetbrains.com

Dave Clarke
DistriNet
Katholieke Universiteit Leuven
Belgium
dave.clarke@cs.kuleuven.be

Abstract

Modern dynamic programming languages provide various mechanisms to implement embedded domain-specific languages (EDSLs), usually based on the meta-object protocol or delegation. The main disadvantages of this approach are the difficulty of statically analyzing domain-specific constraints and providing reasonable code navigation in an existing integrated development environment (IDE), even when the IDE is aware of the host language's semantics. In this paper we present GroovyDSL, a flexible framework for describing semantics-based code assistance for custom EDSLs. GroovyDSL is based on the IntelliJ IDEA environment and allows a developer to add new rules to implement EDSL-aware references resolution and smart code completion. We present a fully implemented small language to describe such rules in a natural way for an EDSL, based on the Groovy programming language, abstracting from the IDE's internal language representation.

Categories and Subject Descriptors D.2.6 [*Software Engineering*]: Coding Tools and Techniques; D.2.6 [*Software Engineering*]: Programming Environments; D.3.4 [*Programming languages*]: Processors

General Terms Languages, Code assistance, Analysis, Frameworks, Context

Keywords IDE, domain-specific languages, dynamic programming languages, Groovy, DSL implementation, end-user programming

1. Introduction

Embedded domain-specific languages (EDSLs) are becoming increasingly popular for rapid program development and prototyping in a wide range of application domains. By providing a powerful and simple technique to describe the semantics of a domain of concern as syntactic sugar in a host language, EDSLs allow the developer to leave all the implementation issues for the native compiler or interpreter and thus concentrate on the description of a problem. Being based on some host language, such as Ruby, Groovy [4, 6] or Clojure, EDSLs may significantly speed up the development of specific components, even by customers not fully familiar with the original language. Modern dynamic programming languages provide various mechanisms to implement EDSLs, mostly based on the meta-object protocol and delegation. This allows small EDSLs to be implemented as part of large programming frameworks, such as Rails, RSpec, Cucumber for Ruby, Grails, Griffon, Gant, Gradle in Groovy and many others. Working with predefined naming or environment conventions allows one to use the rich behaviour of a framework's components, which would not be possible if such conventions were not present.

Nowadays developing large projects without using advanced programming tools such as integrated development environments (IDEs) is unthinkable. However, an IDE cannot take into account the dynamic semantics of all possible EDSLs to provide an adequate on-the-fly analysis, code completion and others features typical of language-supporting tools. Thus, developers and end users of embedded DSLs must decide whether it is worthwhile to use a small and expressive DSL to describe their problem yet lose reasonable tool support, or whether it will be faster to implement the same logic in the original programming language.

In order to fill the gap between on-the-fly static program analysis and the dynamic semantics of embedded DSLs, we introduce GroovyDSL, a framework to describe DSLs written in Groovy. GroovyDSL is available as a part of the Groovy language support for the IntelliJ IDEA environment via the bundled JetGroovy plugin.[1]

GroovyDSL provides a domain-specific language designed to refine the run-time types of expressions in end-user DSLs. These refinements are script files that are executed by the IDE on the fly, bringing new reference resolution and code completion logic into the scope of a project. GroovyDSL relies on the Program Structure Interface (PSI), a set of internal classes and interfaces of IntelliJ IDEA, which allow all programming languages to be described in a uniform way. Due to Groovy's meta-programming capabilities, the developer of DSL descriptions in GroovyDSL generally need not be aware of how PSI works. Interoperation with PSI is handled by calls to methods and properties of GroovyDSL scripts.

Based on recent research on implementing refactorings in IntelliJ IDEA (presented at the Second ACM Workshop on Refactoring Tools [2]), this work is the first attempt to make the internal interfaces of an IDE more intelligible for end users who are typically not experts in writing IDE plug-ins, thus allowing the IDE's functionally to be extended without implementing heavy-weight plugins. The internal API is generally very hard to use by non-expert users and GroovyDSL offers an elegant façade around it.

The main contribution of this paper is the GroovyDSL framework, which allows the static approximation of the semantics of dynamic method resolution in Groovy-based DSLs. In the remainder of the paper we give a short survey of Groovy's meta-programming techniques, and describe the GroovyDSL language and how it is

[1] `http://www.jetbrains.com/idea/`

integrated with the IDE. We also provide several examples using GroovyDSL to describe the augmented expression types in DSLs implemented in Groovy.

2. Meta-programming in Groovy

Groovy offers a large variety of ways to inject behaviour into classes dynamically. Using capabilities of Groovy's meta-object protocol (MOP) one can intercept and change existing methods, add new ones, and even synthesize methods dynamically while a program runs [6]. Groovy offers the following features for dynamically changing the behaviour of classes and objects:

Using ExpandMetaClass Every instance in a Groovy program has a `metaClass` property, which returns the map of methods associated with it. It is possible to add or redefine methods for both Java and Groovy classes and for Groovy instances by replacing the necessary entries in their meta-class.

Implementing invokeMethod() and methodMissing() methods If a method definition is not found in an instance's meta-class, Groovy's method resolution strategy will invoke `invokeMethod()`, if it exists. Otherwise, `methodMissing()` is invoked, if it exists. One can add new logic just by implementing these two methods. As with other methods, they also may be injected into some instance's meta-class. Such an approach can be used to implement before/after method logic, as well implementing 'fake' methods which are not actually present in an object's class.

Using Categories In contrast to the use of a meta-class object, which allows one to globally inject new behaviour into instances, Groovy's Categories provide a way to confine behavioural changes to within the control flow of a specific call of the `use()` method. Categories in Groovy are classes with static methods taking at least one parameter, which is treated to be a receiver of an appropriate method. The `use()` method takes a list of category classes and the closure inside of which they take effect. After leaving the closure block, the injected methods are removed. Categories are an instance of the so-called Context-oriented programming (COP) approach [3] which enables bindings to by modified for a dynamic scope. Clarke and Sergey provide formal semantics for this style of COP [1].

Changing delegation chain for closures Closures in Groovy are implemented as inner classes. They rely on a default strategy to resolve method calls. Methods are sought successively in a closure's body, closure's *owner* meta-class and, finally, in the meta-class of a closure's *delegate* object, which is normally the same as an owner. Groovy provides a way to redefine a closure's owner and delegate, allowing their methods to be invoked unqualified in a closure's body (see Figure 3). This is a typical elegant way to implement the Builder pattern in Groovy.

In custom Groovy frameworks, certain scripts or methods are supposed to be executed in specific contexts, such as in the presence of particular category classes, to augment an instance's meta-class or redefine a closure's delegate. GroovyDSL allows this dynamic behaviour to be approximated by describing it in the form of short scripts. It makes the IDE more aware of the dynamic semantics of method invocations to give more adequate code assistance.

3. GroovyDSL workflow

In this section we give an overview of GroovyDSL's execution model. First, we describe the mechanism of executing GroovyDSL scripts immediately after writing them in an IDE editor with appropriate error handling. Second, we present the main methods and properties of scripts, which are used to define dynamic behaviour in the context of an IDE.

```
0 def ctx = context(ctype: "java.lang.String",
1                   filetypes: ["gtest"],
2                   scope: closureScope())
3
4 contributor ctx, {
5   findClass("java.lang.Throwable")?.
6             methods?.each {add it}
7 }
```

Figure 2. Example definition of a context and a contributor

3.1 Enhancing an IDE on the fly

Every GroovyDSL script is just a plain Groovy file with the `.gdsl` extension that is supposed to be executed in a specific environment. A GroovyDSL script is a DSLs to describe the rules of run-time reference resolution for another DSL. The execution workflow for GroovyDSL script processing is depicted in Figure 1.

After being made available to the IDE, GroovyDSL scripts are evaluated lazily, whenever some unresolved code reference occurs in an editor or whenever the "Complete Code" action is invoked. New procedures obtained by parsing scripts are cached and applied in various contexts to determine whether to add new behaviour to the execution environment.

3.2 Writing scripts: Contexts and Contributors

As in plain Groovy, GroovyDSL scripts are allowed to define variables and functions, use common control structures, and access JDK and Groovy SDK classes. GroovyDSL scripts operate upon two main entities: Contexts and Contributors.

Contexts in GroovyDSL describe a place in the program text where some behaviour will be available, without any reference to the kind of behaviour. Our contexts are related to, yet more general than, the notion of context in programming language semantics [5]; a context may be considered as a Groovy program with a "hole" that remains for some expression's method or property invocation.

Contexts are first-class citizens in GroovyDSL, so they may be stored in local script variables and passed as parameters to functions. Figure 2 shows the definition of a context `ctx`. This context is described by a target class type `java.lang.String`, file extension `.gtest` and affects only bodies of closures. We use the predefined function `context()` with a set of optional named parameters such as `ctype` and `filetypes`.

Contributors are responsible for describing DSL-specific behaviour. A contributor takes the list of contexts it will be applied to and a closure that describes the logic of the provided new behaviour. An example contributor providing simple delegation logic by adding all methods of the class `Throwable` to the class `String` is shown in Line 4 of Figure 2. A new contributor is defined by invoking the function `contributor()`. The previously defined context `ctx` is passed as a one-element list. The second argument, a closure with no parameters, is invoked for all places in a program that match one of the contexts provided in the first argument to `contributor()`.

In the example, the utility method `findClass()` is invoked to obtain a representation of a class in a project, if it is found, or **`null`** otherwise. We use safe-dereferencing operator `?.` to avoid throwing of a `NullPointerException`. Property `methods` gives a list of methods available in the found

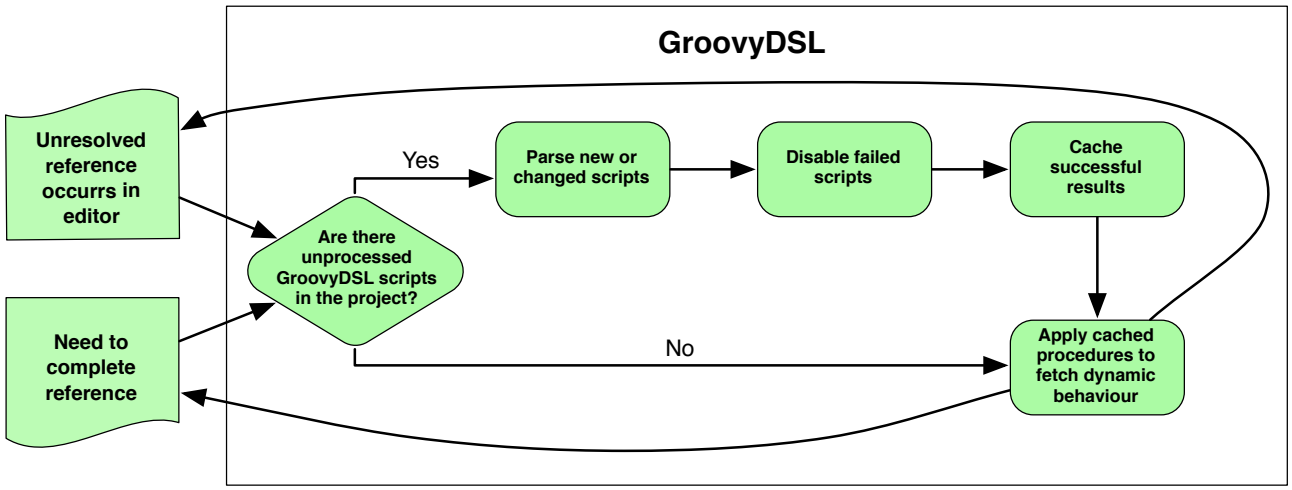

Figure 1. GroovyDSL workflow overview

class, so it is possible to iterate through this list with a closure via `each()` method. To register new behaviour we invoke the method `add`, which adds method to the given place.

3.3 Extending the GroovyDSL framework

The core of GroovyDSL functionality is concentrated in the body of the closure passed as the second parameter to the `contributor()` method. All invocations of functions and properties inside of it are calls to wrappers around IntelliJ IDEA's internal PSI. There are two kinds of wrappers. The first kind are unqualified, such as `findClass()`. The second kind are methods added to the original PSI classes via the mix-in category mechanism. For example, the `methods` property (in Figure 2) is added externally by the GroovyDSL environment to the core PSI's class `PsiClass`, representing classes in Java-like languages.

Both of these sets of functions may be extended with new ones using IntelliJ IDEA's plug-in mechanism. To provide new unqualified methods one should provide a class implementing the `GdslMembersProvider` interface and register it by a plug-in descriptor. All methods of such newly added components will be used as possible delegates for unqualified method calls. New methods and properties may be added to PSI by implementing the `PsiEnhancerCategory` interface. Its registered implementations will be used as categories (see Section 2) and "wrapped around" the body of a contributor's closure at the moment of its execution.

```
0  class MyDelegate {
1    def saySomething(String str) {
2      println str
3    }
4  }
5
6  class Runner {
7    def run(obj, Closure cl) {
8      cl.delegate = obj
9      cl()
10   }
11 }
12
13 def runner = new Runner()
14 runner.run(new MyDelegate()) {
15   saySomething("hello!")
16 }
```

Figure 3. Example of closure delegation. Method `run()` of class `Runner` substitutes its first parameter for the delegate of the closure passed as its second parameter. This allows the unqualified method call `saySomething()` of class `MyDelegate` on line 15 to be correctly resolved at the runtime.

4. Example describing dynamic semantics

In this section we give two examples of using GroovyDSL in practice.

4.1 Describing delegation

The first example describes a context-dependent delegate substitution to a closure. Figure 3 shows the definition of the class `Runner`, whose method `run()` takes an object `obj` and a closure `cl` as parameters. It redefines the default delegate of the given closure and assigns `obj` to its `delegate` property. When passing a closure to the method `run()` as a parameter, according to `run()`'s definition, it is allowed to invoke unqualified methods an properties of its first parameter inside of the body of a closure (see Section 2).

The GroovyDSL script describing this behaviour is shown in Figure 4. In this case we are interested in contexts placed inside of closure bodies. This fact is denoted by passing `closureScope()` as a parameter of the described context `ctx`. When defining a contributor, we use utility method `enclosingCall()`, which returns the call expression of method with name `run` that wraps the analyzed closure context, if it exists, or **`null`** otherwise. If an appropriate method call is found, we try to resolve it to a method definition by calling the `bind()` method. The last check we perform is to ensure that the full name of the receiver class is `Runner`. The last call to the `delegatesTo()` method adjusts the delegate of the given closure to the actual class type of the first argument. So, the delegated call to the `MyDelegate`'s `saySomething()` method will be no longer highlighted in the IDE's editor as a possibly unresolved and will be suggested in IDE as possible variant for a code completion inside the closure's body.

```
0 def ctx = context(scope: closureScope())
1
2 contributor(ctx, {
3   def call = enclosingCall("run")
4   if (call) {
5     def method = call.bind()
6     def clazz = method?.containingClass
7     if ("Runner".equals(clazz?.qualName)) {
8       delegatesTo(call.arguments[0])
9     }
10  }
11 })
```

Figure 4. Description of delegation semantics in GroovyDSL

4.2 Adding custom methods and properties

Using Groovy's meta-classes, custom methods and properties may be added to existing classes on the fly. Within the context of a GroovyDSL contributor, the predefined methods `method` and `property` simplify this task. Figure 5 gives an example. The context `ctx1` targets class `java.lang.Number`, which will then be augmented in all possible scopes. In the contributor's closure, the method `property` is used to add new properties to the target class. This method take several named parameters. The parameter `name` defines the name of the property to be injected and the `type` parameter defines its type.

```
0 def ctx1 = context(ctype: "java.lang.Number")
1 contributor(ctx1) {
2   property name: "eur", type: "test.Money"
3   property name: "usd", type: "test.Money"
4 }
5
6 def ctx2 = context(ctype: "ReentrantLock")
7 contributor(ctx2) {
8   method name: 'withLock',
9          type: 'void',
10         params: [closure: {}]
11 }
12
13 def ctx3 = context(ctype: "org.test.Employee")
14 contributor(ctx3) {
15   ['Senior', 'Expert', 'Junior'].each {
16     method name: "findAll${it}Employees",
17            type: 'java.util.Collection'
18   }
19 }
```

Figure 5. Injecting custom methods and properties into classes

The function `method` is used to add synthesized methods to a class. It takes named parameters specifying the name of the defined method, its return type, and a list of parameters with their types. Note that the parameter `closure` of method `withLock` uses `{}`, which is GroovyDSLs syntactic sugar for `'groovy.lang.Closure'`.

To create families of dynamic methods, we can use all the power of Groovy collections and higher-order functions. On line 15 of Figure 5 we create a set of methods with names `findAll*Employees` for *each* kind of employee from the list. Such a pattern is applicable for Groovy-based frameworks, such Grails, where specific methods are generated using the class structure and some naming conventions.

5. Conclusion

In this paper we presented GroovyDSL, a framework to refine runtime expression types in embedded DSLs based on the Groovy programming language. GroovyDSL is implemented in Groovy and Java as a part of the JetGroovy plugin for the IntelliJ IDEA 9 environment. The presented approach allows the scope of behavioural changes to be statically described to obtain IDE support for applications written in the context-oriented programming style using Categories. Code assistance for GroovyDSL's internal language is described in GroovyDSL itself as well as support for other EDSL's for popular Groovy-based frameworks. GroovyDSL was successfully applied to describe the semantics of basic compile-time Groovy AST transformations[2] and properties of Gant[3] scripts.

We are working on the implementation of a declarative language that should allow inspections and constraint checking to be described in Groovy programs. In addition, we are going to extend the DSL-based approach to describe the IDE's behaviour to support other languages. Designing a rule-based rewriting specification language to describe refactorings and their validation before and after transformation is an interesting direction for future work.

Full documentation for GroovyDSL with examples is available from `http://www.jetbrains.net/confluence/display/GRVY/Scripting+IDE+for+DSL+awareness`.

Acknowledgments

Thanks to Václav Pech who suggested vivid examples for method injections and gave some insightful comments on this work. We also thank Dimitri Van Landuyt for comments concerning the GroovyDSL activity diagram .

References

[1] D. Clarke and I. Sergey. A semantics for context-oriented programming with layers. In *COP '09: International Workshop on Context-Oriented Programming*, pages 1–6, Genova, Italy, 2009. ACM.

[2] D. Dig, R. M. Fuhrer, and R. Johnson. The 2nd workshop on refactoring tools (WRT'08). In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 859–860, Nashville, TN, USA, 2008. ACM.

[3] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology, March-April 2008, ETH Zurich*, 7(3):125–151, 2008.

[4] G. Laforge. Groovy: An agile dynamic language for the Java Platform, 2008.

[5] A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Trans. Comput. Logic*, 9(3):1–49, 2008.

[6] V. Subramanian. *Programming Groovy*. The Pragmatic Bookshelf, 2008.

[2] `http://groovy.codehaus.org/Compile-time+Metaprogramming+-+AST+Transformations`

[3] `http://gant.codehaus.org/`

A. Presentation outline

Decreasing the amount of boilerplate code is one of most important purposes of both internal and external domain-specific languages. There are different ways to hide behaviour, unrelated to the main logic of an application. As an example of such techniques, let's look at compile-time AST transformations in the Groovy programming language. The code in the Figure 6 gives an example of the `LockableList` class, which combines functionality of a plain list and a lockable data structure.

```
0  import java.util.concurrent.locks.*
1
2  class LockableList {
3      List list = []
4      Lock lock = new ReentrantLock()
5  }
6
7  def aList = new LockableList()
8
9  aList.lock.lock()
10 try {
11     aList.list << 'Groovy'
12     aList.list << 'Grails'
13 } finally {
14     aList.lock.unlock()
15 }
16
17 assert list.size() == 2
```

Figure 6. Traditional implementation of `LockableList`

The typical way to get rid of annoying references to `list` and `lock` fields on lines 11, 12 and 14 is to inherit both `java.util.concurrent.locks.Lock` and `java.util.List` interfaces, or, at least, implement all necessary methods, such as `unlock()`, to make them delegate to appropriate `List` and `Lock` fields. Such typical procedures may be automatized by adding the `groovy.land.Delegate` annotation to the named fields. Figure 7 shows the same example with annotated fields, which now look more concise.

```
0  import java.util.concurrent.locks.*
1
2  class LockableList {
3    @Delegate private List list = []
4    @Delegate private Lock lock =
5                  new ReentrantLock()
6  }
7
8  def aList = new LockableList()
9
10 aList.lock()
11 try {
12     aList << 'Groovy'
13     aList << 'Grails'
14 } finally {
15     aList.unlock()
16 }
17
18 assert list.size() == 2
19 assert list instanceof Lock
20 assert list instanceof List
```

Figure 7. An implementation of `LockableList` with `@Delegate`-annotated fields

Using the `@Delegate` annotation allows one to use an instance of `LockableList` as both an instance of `List` and of `Lock`, whereas the actual definitions may be marked as **`private`**.

Since Groovy provides freely extensible compiler infrastructure and a variety of meta-programming techniques, it is not a big problem to describe rules to augment existing or new classes with some implicit behaviour at runtime. A programmer may provide his own annotations and enhancement algorithms to inject new method or properties into classes, even into Java classes.

Some problems may occur when working in big projects and trying to refactor with an aid of some tools. Let's imagine temporarily that the `java.util.concurrent.locks.Lock` interface is not in Java SDK and one can freely rename its methods, delete them or add new ones. After renaming, for instance, `lock()` method in a modern IDE, all its invocations should be renamed as well to avoid unresolved references. But unlike Java, Groovy is dynamically-typed language and since by default an IDE is unaware of custom annotations' semantics, calls such as `aList.lock()` in Figure 7 will be kept unchanged. Such broken code will be compiled even without warnings, but it will raise an `InvocationError` at runtime.

To avoid such problems and provide appropriate information about the reference resolution semantics to an IDE, one should somehow describe the semantics of target language transformations. There are at least two ways to do it:

1. Use a native compiler as to obtain information about references and types. Such an approach is supposed to work well in the case of custom annotations, as we described via an example. But it cannot handle runtime code transformations, such as so-called "monkey patching" and dynamic methods injections, for obvious reasons. Also, to the best of our knowledge, the reuse of a native compiler may significantly complicate the implementation of refactorings in multi-language projects, written, for example, in Java and Groovy.
2. Provide a mechanism to (partially) describe the semantics of dynamic code transformations to augment an IDE's logic for handling references in code. This approach demands full control of a project code through some uniform representation to be able to handle definitions and references in multi-language codebase. The language to describe the dynamic semantics should be sufficiently expressive to handle different cases of dynamic behavioural changes. It addition, it should be quickly computable to prevent an IDE from hanging while performing background analyzes.

Implementing support for Groovy-based domain-specific languages in IntelliJ IDEA 9's programming environment, we have chosen the second approach. The framework we present is called GroovyDSL and it is designed as a part of Groovy language support. GroovyDSL provides a language to describe dynamic code transformations as simple declarative scripts in the IntelliJ IDEA itself. For example, the code in Figure 8 gives a definition of the semantics that `@Delegate` annotations bring to a program.

```
0 contributor(context()) {
1  classType?.fields?.each {
2   if (it.hasAnnotation("groovy.lang.Delegate")){
3    delegatesTo(it.classType)
4   }
5  }
6 }
```

Figure 8. Description of the `@Delegate` annotation semantics in GroovyDSL

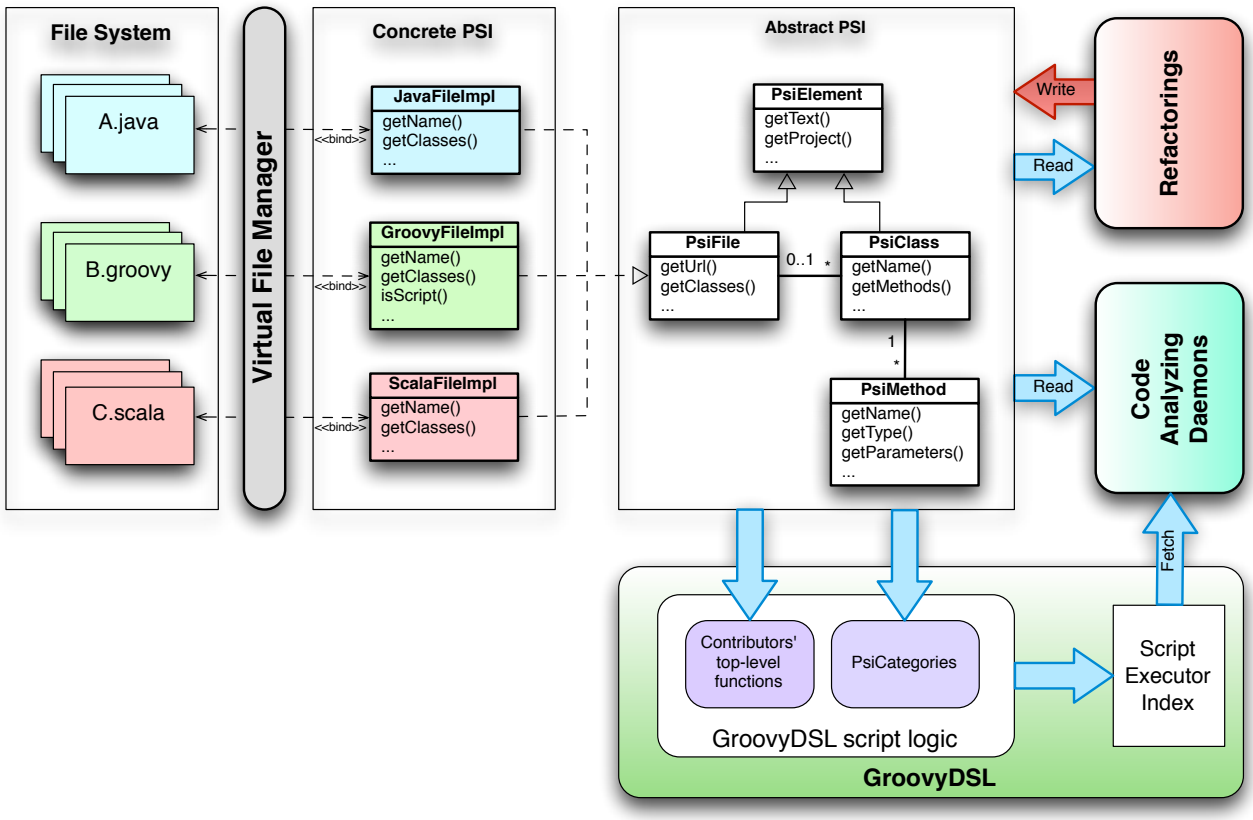

Figure 9. Program Structure Interface with GroovyDSL architecture overview

Let's leave the definitions of `contributor()` and `context()` functions aside for a moment. Literally, this description may be read as "For *each* field of some class, if this field is annotated with the `groovy.lang.Delegate` annotation, then the original class is enhanced with the behavior of this field's type."

A GroovyDSL script is nothing but a plain Groovy script that is supposed to be executed in the specific environment where all its references will be resolved. Due to a variety of patterns to define behaviour injection in Groovy, we gave up implementing such logic in the form of a dialog with settings. Instead of this we provide to end-users a front-end for operating with IntelliJ's *Program Structure Interface* (PSI). PSI is a set of classes and interfaces to describe the overall structure of a project, loaded in IDE, including its sources, libraries and other dependencies. Being sufficiently general, it allows various languages to be described in a uniform way, which eases cross-language reference resolution, analysis and refactoring. All issues of synchronization with a file system are left under the hood, but normally, to get the most flexible control over programs in some particular language, one should implement a front-end compiler for it from scratch. This how the support for Java, Scala, Groovy, Clojure, Python and many other languages is implemented in IntelliJ IDEA.[4] Dealing with an already supported language one needs to have an access only to a small part of PSI, namely, the one responsible for the structure of program files, their arrangement in a project and the type system.

A coarse overview of IntelliJ IDEA's PSI, its place in the application structure and its interaction with GroovyDSL is depicted in Figure 9. IntelliJ IDEA framework maintains two-way synchronization between source files with registered filetypes through a set of *Virtual File Manager* classes. There is at least one implementation of the `PsiFile` interface, corresponding to each file type. Three concrete implementations for source files in Java, Groovy and Scala are depicted. There are many clients of PSI, though in the picture only two are shown, namely, *Refactorings* and *Code Analyzing Daemons*. Various refactoring actions deal with PSI by reading and modifying it during the appropriate code transformations. Code analyzers check whether there are unresolved references in project code, malformed or untyped expressions, and collect information about available types of expressions to build a list of variants for code completion in every place of a program.

In the presence of the GroovyDSL framework, information about expression types and available references is obtained by code analyzers not only by scanning the actual PSI of a project but also by 'asking' GroovyDSL components. GroovyDSL implements its own daemon, which maintains an index of all `.gdsl` files in a project via the *Script Executor Index* structure. After being made available to the IDE, GroovyDSL scripts are evaluated lazily whenever some unresolved code reference occurs in an editor or whenever the "Complete Code" action is invoked. Besides standard Groovy variables and functions at the top level, GroovyDSL scripts contain declarations of two entities, namely Contexts and Contributors.

[4] `http://www.jetbrains.com/idea/documentation/idea_5.0.html`

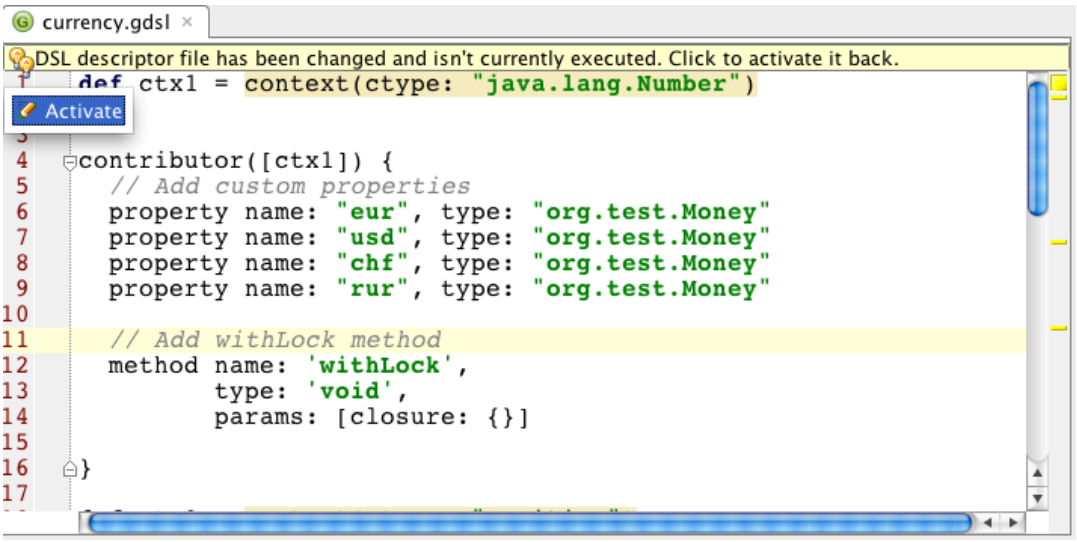

Figure 10. GroovyDSL script is changed and should be reloaded

Contexts in GroovyDSL are entities that give an answer to the question "Where?" In other words, a context describes a place where some behaviour will be available, without any reference to the kind of behaviour. Contexts are first-class citizens in GroovyDSL, so they may be stored in local script variables and passed as parameters to functions. The built-in function `context()` initializes a new context. It may take various parameters to restrict the actual context's scope of applicability.

Contributors are responsible for describing DSL-specific behaviour. A contributor takes the list of contexts it will be applied to and a closure that describes the logic of the provided new behaviour.

A GroovyDSL script is executed by clicking on the "Activate" button in IntelliJ's editor window (see Figure 10). If some error occurs during script evaluation, the script will be disabled and an exception stack trace will be provided to analyze in the IDE. In case of successful script evaluation, all defined contributors and contexts will be stored in Script Executor Index to be invoked on demand. This may occur, for example, when "Code Completion" is invoked in an editor window. Figure 11 gives an example of such code completion within an environment where the script from the Figure 10 had already executed.

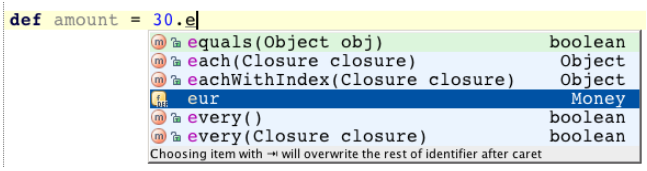

Figure 11. Code completion action is invoked in editor with Groovy script opened

As it may be seen from Figure 9, GroovyDSL scripts interact with PSI not only directly, but also through two sets of components: contributors' top-level functions and PSI categories. This is the 'heart' of GroovyDSL. Normally the main part of script logic written by an end-user is located inside of the closure passed to the `contributor()` call as an argument. That is why we implemented a number of functions, which are available only in the body of the closure. Most parts of them delegate implicitly to PSI classes and interfaces, providing a declarative way to analyze program structure or add new methods and properties into types. In the Table 12 we give definitions of several useful methods and properties from the top-level API of contributor. Methods taking an instance of the `Map` as an argument are supposed to be called with named arguments, such as `name` and `type` for `property()` method from Figure 10.

`findClass(String name)`	Looks for a class with the full-qualified name `name` in the current project
`property(Map args)`	Adds a property to the given type
`method(Map args)`	Adds a method to the given type
`add(PsiMember member)`	Adds its `member` parameter to the given type
`place`	A reference to the expression being analyzed
`classType`	The inferred type of the expression being analyzed
`enclosingCall(String mn)`	Looks for a call to method with the given name `nm`, enclosing an actual context
`delegatesTo(PsiClass c)`	Adds all accessible methods of the class `c` to the actual type

Figure 12. Some top-level methods and properties of contributor's closure argument

The full actual list of top-level methods and properties is available on the GroovyDSL manual page.

Besides the implementation of top-level functions, GroovyDSL also provides new methods and properties to PSI interfaces themselves to simplify the analysis of programs. Such augmentations of PSI are implemented in the form of Groovy Categories, which are 'wrapping' the call of the contributor's closure argument. We chose this way instead of `MetaClass` modifications to confine be-

```
0 public class GroovyDslDefaultMembers implements GdslMembersProvider {
1
2  @Nullable
3  public PsiClass findClass(String fqn, GdslMembersHolderConsumer consumer) {
4   final JavaPsiFacade facade = JavaPsiFacade.getInstance(consumer.getProject());
5   final PsiClass clazz = facade.findClass(fqn, GlobalSearchScope.allScope(consumer.getProject()));
6   return clazz;
7  }
8
9  // Other top-level methods
10 }
```

Figure 13. The class `GroovyDslDefaultMembers` implements basic top-level methods of contributor's closure parameter.

havioural changes within the control flow of a specific closure. Categories in Groovy are classes with static methods taking at least one parameter, which is treated to be the receiver of an appropriate method, so for better productivity one may implement them in Java. Moreover, this way we restrict access from a script to methods of the so-called "write PSI", i.e. the ones that allow program structure to be modified, since GroovyDSL scripts are supposed to analyze files only, not modify them.

Both sets of top-level methods and enhancing PSI categories may be extended for GroovyDSL via the provided plugin API. To implement new top-level methods one should provide a class implementing the `GdslMembersProvider` interface, and register it using a plugin descriptor. All methods of such newly added components will be used as possible delegates for unqualified method calls from inside of contributor's closure parameter. All methods of the `GdslMembersProvider` implementation that are supposed to be used as delegates must take an instance of the `GdslMembersHolderConsumer` class as their last parameter to get information about the context they act within. Figure 13 gives an example of such an implementation.

New methods and properties may be added to PSI by implementing the `PsiEnhancerCategory` interface. Its implementations, registered using a plugin descriptor, will be used as categories "wrapped around" the body of a contributor's closure at the moment of its execution. Figure 14 gives an example of adding the missing method `getQualName` into the `PsiClass` interface.

```
0 public class PsiClassCategory
1        implements PsiEnhancerCategory {
2
3  @Nullable
4  public static String getQualName(PsiClass c) {
5   return c.getQualifiedName();
6  }
7
8  // Other category methods
9 }
```

Figure 14. A part of the `PsiClassCategory` class, implemented in Java, which adds the `getQualName()` method to the `PsiClass` class

Summarizing, in the described approach an end-user describes his DSL's runtime types working transparently with three different sets of functions:

- 'Pure' PSI itself. For example in Figure 8, on line 1, `fields` property is the default method `getFields()` of the `PsiClass` class, returning an array of its fields.
- Predefined top-level methods of the closure passed as an argument to a `contributor()` call. In the mentioned example this is a call to the `delegatesTo()` function.
- Method and properties 'injected' into PSI using GroovyDSL's category components. For example, see the injected method `hasAnnotation(String name)` of the `PsiMember` class, used on line 2 of the example in Figure 8.

Let's look now at an example of more sophisticated contexts. Fo performance reasons we made the definition of context as simply to compute as possible. Internally every context is a set of filters, which are checked in all places where it is possible to build a 'hole' according to its definition. But passing additional named arguments to the call of the `context()` function it is possible to refine the context for particular types of files, scopes, and types to be augmented. Figure 15 shows part of the file `metaDsl.gdsl`, which is bundled into IntelliJ IDEA 9 and describes the available methods for a contributor's closure.

```
0 def contributorBody = context(
1           scope: closureScope(isArg: true),
2           filetypes: ["gdsl"])
3
4 contributor contributorBody, {
5  if (enclosingCall("contributor")) {
6   method name: "method",
7          type: "void",
8          params: [args: [:]]
9
10  property name: "classType",
11           type: "com.intellij.psi.PsiClass"
12  // Other methods
13 }
14 }
```

Figure 15. Description of GroovyDSL top-level methods

Here we restrict context's scope to make it spread only to expressions located inside closure's bodies. Moreover, we are interested only in closures that are passed to some method or other function as arguments, so we pass a named parameter `isArgs: true` to the `closureScope()` call. Finally, our context `contributorBody` is valid only in files with the `.gdsl` extension, so we pass an extra parameter `filetypes: ["gdsl"]` to the context constructor. The defined context is assigned to the local script variable contributorBody and used later in the contributor, which adds method `method`, property `className`, and other members to the closure's top-level.

Concluding, we observe that GroovyDSL was successfully used as a part of IntelliJ IDEA's Groovy language support to describe,

in particular, the Gant build tool and compile-time Groovy AST transformations: [5]

- Delegate transformation: `delegateTransform.gdsl`
- Category and Mixin transformations: `categoryTransform.gdsl`
- Newify transformation: `newifyTransform.gdsl`
- Singleton transformation: `singletonTransform.gdsl`
- Bindable and Vetoable transformation: `bindableTransform.gdsl`, `vetoableTransform.gdsl`

All these scripts are bundled into the latest builds of IntelliJ IDEA 9's distribution.

[5] `http://groovy.codehaus.org/Compile-time+Metaprogramming+-+AST+Transformations`