

inCode.Rules: An Agile Approach for Defining and Checking Architectural Constraints

Abstract—The size and complexity of software systems is constantly and abruptly increasing, as well as the size of the teams who develop them. Although software systems usually start with a clean design and an unitary architecture, preserving the design quality in the final product, especially its modularity and reusability, depends on the programmers’ ability to understand, implement and maintain the initial architecture of the system; in other words, it depends on the ability to preserve a common vision about the high-level design *i.e.*, to preserve the *architectural integrity*. This is an essential problem, and it has led to a number of approaches that help maintaining architectural integrity by allowing for the specification and checking of architectural rules (constraints) in code. Unfortunately, these approaches are rarely used in practice because of their excessive complexity, lack of flexibility and absence of integration with the actual development environment. In this paper we propose a new, agile approach to defining and checking architectural constraints. The proposed solution consists, on one hand, of the `inCode.Rules` language that offers a highly intuitive, yet flexible, means for defining architectural rules. On the other hand, `inCode.Rules` is far more than a language specification: in the paper we show how architectural rules can be automatically checked using the `inCode.Rules` interpreter; and how they can be easily edited using the full-fledged editor that we created. Both the interpreter and the editor are tightly integrated in the Eclipse IDE. Furthermore, `inCode.Rules` has grown beyond being just a prototype as it has been already successfully applied on large-scale systems of over 1 MLOC. Thus, `inCode.Rules` provides a more agile approach to architecture verification, as it brings the evolution of code and architecture closer than ever before.

Keywords: quality assessment, architectural integrity, Eclipse, software engineering,

I. INTRODUCTION

In the last decades the size and complexity of software systems has been growing exponentially. While the object-oriented paradigm has come with the promise of providing an adequate means to manage the complexity of large-scale software systems, time has shown that object-oriented design is not so easy to learn and very hard to master [1]. As a result, we are nowadays confronted with a large number of object-oriented legacy systems that are hard to maintain and evolve and consequently the maintainability of large-scale software system gained an important place in the software development process. Maintainability is not just about repair work, or (a more popular, more informal expression) fixing bugs. Maintainability is also about further development of the system, adding new behavior, new features, adapting the system in order to work in a different environment (e.g. a different operating system).

The Problem: Most of the causes of the maintainability problems are directly related to the poor design of the software

system. The quality of the final product, especially its modularity and its reusability all depend on the programmers ability to *understand, implement and maintain the initial architecture* of the system. Problems appear due to the parallel evolution of the architecture and of the source-code. The problem is known as *architectural mismatch i.e.*, “at the time the system architecture is published it is already obsolete” [2]. This problem is very important, especially in the industry [3] where the engineers must deal for up to 15 years [4] with the same system, and thus with the same architecture [5]. The problem of software erosion in general [6], and preserving the architectural integrity of a system in particular, becomes even more difficult as the number and geographical dispersion of developers (*e.g.*, by outsourcing) working on the same system increases [7]. With a widely spread team consisting of people with heterogeneous profiles it is very hard to maintain a common vision, to preserve the architectural integrity of a system.

If the development team is lacking the means to maintain a close connection between the source-code and the architecture, then software erosion will gradually creep in and make changing the software significantly harder and less predictable, which can eventually lead to the cancellation of the entire project [7]. As we will see in more detail in the coming section, this issues has led in the last decade to a number of approaches that allow for the specification and automatic checking of architectural constraints. Unfortunately, these approaches are rarely used in practice because some of them are highly complex, while are others are too rigid and when it comes to defining specific rules; and, on top of that, almost all lack a strong integration with the development environment, which is essential, as otherwise the architecture and the actual development remain two separate worlds.

Our Approach: In this context, this paper proposes a new, agile, approach to defining and checking architectural constraints. The proposed solution consists, on one hand, of the `inCode.Rules` *language*, a domain-specific language (DSL) that we created in order to provide a simple yet flexible means for defining architectural rules. On the other hand, our approach is not only theoretical, as we implemented the language in a manner that makes it smoothly integrated in the Eclipse IDE. More precisely, *inCode.Rules*, together with a full-fledged editor, are implemented as Eclipse plugins, facilitating thus a more agile approach that brings the evolution of code and architecture closer as ever before. The implementation is based on the the Eclipse modeling framework [8] and it comes as an extension of `inCode` [9], the software analysis platform that we have previously created to

enable a continuous quality assessment.

Outline.: The rest of the paper is organized as follows: in the next section we summarize the related work. In Section 3 we describe the key concepts of the `inCode.Rules` language, pointing out its simplicity and flexibility traits. In Section 4 we show how `inCode.Rules` is smoothly integrated in the Eclipse development environment. In the last part of Section 4 we present the time performance of `inCode.Rules` when running on systems of various sizes and discuss the factors that influence the execution times. We conclude in Section 5 by wrapping-up the contributions of the paper and by discussing the enhancement perspectives.

II. RELATED WORK

As mentioned in the beginning, in order to address the issue of architecture decay, a number of specifications languages – for defining architectural constraints in a formal manner – have emerged, as well as a series of static-analysis tools, that allow checking the architectural integrity. The goal has been to eliminate any ambiguities that might occur.

Architectural Description Languages (ADL): The most representative approaches of this category are Wright [10], which enables architects to specify temporal communication protocols, SADL [11] that formalizes architectures in terms of theories, Rapide [12] which supports behavioral specification and event-based simulation of reactive architectures, Darwin [13] that provides support for distributed architectures that change dynamically. ArchJava [14] is similar to Darwin, but it has the advantage that it can also check for communication integrity. Although these languages have support for sophisticated analysis, they are not able to check consistency with source code, except for ArchJava. However, ArchJava requires modifying the source code and compiling it with its own custom compiler.

Static Analysis Tools: There are a number of tools that are aimed to assess design quality at the architectural level. A perfect example is Lattix [15] which determines and verifies dependencies between the subsystems or packages of Java systems. This solution is widely used in the industry and is based on the "dependency structure matrix" (DSM). DSM is a method that has its roots in the manufacturing industry. It has been developed to optimize the production process: "In the development of a product, a collection of tasks is performed. These tasks have dependencies on one another, either because of physical objects that must flow from task to task, or because of information that one task requires and which another task provides.[...] The term dependency structure matrix refers both to a particular representation of such dependencies, and to algorithms for reorganizing the dependencies by reordering and clustering tasks." [16]. The Lattix LDM tool has the advantage of scalability and good integration with the development environment. On the other hand one of the disadvantages of this tool is that one needs to review the source code quite often in order for the architecture to stay consistent with the code. Another

disadvantage is the fact that it can check only one type of rule- that regulates what an entity is allowed or isn't allowed to access. And lastly, learning to properly use the tool requires a not negligible amount of effort.

Summarizing the limitations of previous work on defining and verifying architectural constraints, we define a set of *three mandatory success criteria* for a new ADL language, that would have made ADLs more agile and thus increase the chances of a *wide adoption by developers*:

- 1) **Simplicity**. An ADL must make it easy to write (and re-write) rules; not only for trained architects, but for any developer. The language should not have a steep learning curve. For this, the language must be as close as possible to way such constraints would be expressed in the *natural language*. If an ADL supports the creation of *human-readable* specifications then the specification serves not only as an input for the automatic checking of the architectural rules, but also as a proper *architecture documentation*.
- 2) **Flexibility**. Many ADLs allow the specification of architectural rules only a coarse-grained level, making an oftentimes arbitrary separation between architecture and design [17]. Thus, we consider it mandatory that an ADL should allow one to define rules both in terms of "packages" as well as in terms of "classes" and "methods". Furthermore, the language needs flexibility not only in terms of the involved entities, but also in terms of the granularity at which *relations* can be defined.
- 3) **Integration**. One of the biggest problems with the documents describing architectures is the gap that separates them from the place where the actual development takes place: the IDE (Integrated Development Environment). Thus, the best way to create ADL implementation and tool support (editor, interpreter, rule checker etc) is to integrate it tightly and smoothly with the IDE itself.

III. INCODE.RULES: THE LANGUAGE

This section will present the key language mechanisms that we defined in `inCode.Rules`. The point of this section is not to provide an exhaustive description of the language, as this is beyond the scope of the paper¹; The point is to reveal how `inCode.Rules` simultaneously fulfills the *simplicity* and the *flexibility* criteria.

The language supports two different sets of rules: (i) *usage rules* and (ii) *property rules*. The usage rules are meant to provide the designer the ability to break-down the system into components, or modules, as well as to specify the usage relationships between the components. The beauty of this rule type is that one can define components that overlap, thus allowing the designer to specify more than one *modularization view* of the same system. Property rules on the other hand have another role, they allow the designer to enforce rules using filters and properties that are already defined in `inCode`.

¹For a complete description of `inCode.Rules` please refer to [18]

A. Usage Rules

This first type of rule is used to specify how the design entities (*e.g.*, packages, classes *etc.*) interact with each other. This category of rules is the one most supported by all ADLs, being oftentimes the only type of rules specifiable by an ADL. A usage rule consist of three elements: **Subject**, **Action** and **Target**. The *subject* is the design entity (or entities) to which the rule refers to, while the *target* is the specification of the entity (or entities) that is/are related to the subject by means of the *action*. For example:

Listing 1: Rule Example

```
package named "X" must not use
package named "Y" ;
```

Subjects and Targets: . The specification of both subject and targets follows the same structure (see Figure 1). In the previous example the subject is: `package named "X"` ; and the target is specified as `package named "Y"` . In spite of their identical specification, there are two obvious differences between the subject and the target: (i) the subject is the entity who takes the “blame” if the rule is broken; and (ii) the orientation of the action is always from the subject to the target (*e.g.*, in the previous example, while “X” is not allowed to use “Y”, there is no restriction the other way around).

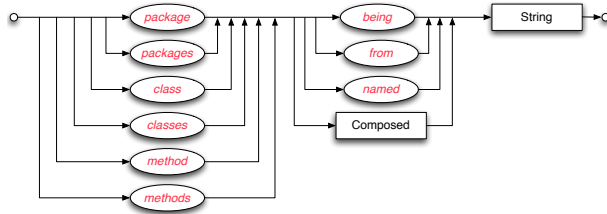


Figure 1: Specification of Subjects and Targets

So, what is the “anatomy” of subject and target entities? We first notice that the word `package` which is the subject’s **entity type**. In contrast with other ADLs `inCode.Rules` supports not only packages as subject entities, but also finer-grained design entities, namely: classes and methods.

The second part of the subject/target is called a **selector**. The selector is used for specifying which concrete design entities (instances) of the aforementioned **entity type** are actually the subject of the rule. In our example, the selector is a named selector, which selects packages by their name, that can be specified either as full-names or by means of *regular expressions*. Apart from the named selector, `inCode.Rules` supports three other types of selectors:

- **The being Selector:** selects entities based on any *design property* (*e.g.*, the presence of a design flaws). These design properties are made available by the underlying software analysis infrastructure (*i.e.*, `inCode`), as described in the next section (see Figure 5). An example of a rule based on the the `being` selector is the one below which states that the subjects of the rule are all classes that have the *Data Class* design flaw [19]:

```
classes being "Data Class" ...
```

- **The from Selector:** is similar to the `named` selector in the sense that it selects the subject entities based on name – either full names or names based on regular expressions. The difference is that the name specified in the `from` selector refers to the container entity (*e.g.*, the container of classes are packages, and the containers of methods are classes). In other words, it searches for the type of Java design elements that contain those design elements specified by the *entity type*. In the example below the subject of the rule consists of all classes from all the `org.eclipse.ui` package and all its “sub-packages”²:

```
classes from "org.eclipse.ui.*" ...
```

- **Composed Selectors.** The atomic selectors specified so far can also be combined to build a complex selection condition. Selectors can be combined using the two main logical operators: `and` and `or`. Obviously, the `and` operator denotes that both operand selectors must be applied *i.e.*, the returned elements must fulfill both criteria; in contrast, the `or` operator means that the operand selectors are applied in parallel *i.e.*, every one of the returned elements must be according to at least one of the selectors.

It should be also noted that `inCode.Rules` allows for a nesting of composed selectors for allowing to express even more complex selection criterions (*i.e.*, a composed selector can be an operand for another composed selector). In the example below, the subject of the rule consists of all classes that have their names ending in `Dialog` and belonging to the `org.eclipse.ui` package or to one of its “sub-packages”³:

Listing 2: Usage of Composed Selectors

```
classes from "org.eclipse.ui.*"
and named "*Dialog" ....
package named "Y" ;
```

Action: An action is composed of two parts: the **relation specifier** and **relation type** (see Figure 2). While the *relation type* indicates the type of dependency between subject and target that is considered relevant in the context of the rule, the *relation specifier* indicates the *sense of the relation*: (i) `must` – if the existence of the relation is mandatory; (ii) `must not` – the the relation is completely prohibited; (iii) `may` – the relation is neither mandatory nor prohibited⁴.

While many ADLs don’t distinguish among dependency types, in `inCode.Rules` we distinguish between the following *relation types*, which makes it possible to define architectural constraints in a more precise manner:

²In an unrelated matter the reason why we put the word *sub-packages* in quotes is based on the fact that, syntactically, in Java there is no such thing as sub-packages; however, from a logical point of view they have this meaning.

³The reader should note the heavy use of regular expressions.

⁴The `may` relation is especially useful in writing *exception clauses* as described in Section III-C)

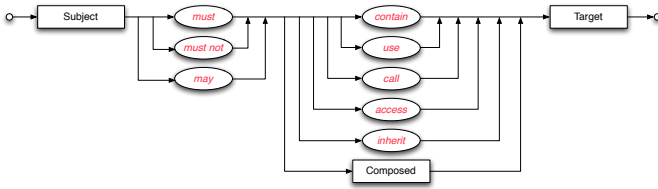


Figure 2: Specification of Actions

- use **relation**, which means that the subject entity references in some way the target entities – by means of calls, data access and/or inheritance – either directly or indirectly *i.e.*, the entities *contained* in the subject entity references entities contained by the target entities; for example: methods in a class reference attributes in a package.
- call **relation**, means that methods defined within the subject entity call methods defined within the target entity; or, if the target (or subject) entities are methods then the relation refers to these method entities.
- access **relation**, in which the subject references attributes defined by the target.
- inherit **relation**, which means that classes defined by the subject inherit classes defined by the target.
- contain **relation** *i.e.*, the target entity is declared in the subject entity (*e.g.*, subject is a class and target is a method defined within that class). It is important to note that the contain relation is not the same as the *containment* relationship between classes known from object oriented programming, where it means that a class A has an attribute of type B. In that context, in most cases, type B is defined outside of class A. In contrast, in `inCode.Rules` "entity A contains entity B" means that the *definition* of entity B is part of the definition of entity A. For example: `class A contains class B` means that class B is an inner-type defined in class A. In other words, in `inCode.Rules`, `contain` refers to the actual Java code, rather than the system modeled by the code.

Composed Actions.: Just like the subject or the target, the action can also be composed of two (or more) actions, by means of the `or /` and operators. For instance, we might want to specify that *neither calls nor accesses* are made from package named "a.b" to package named "x.y". This can be specified as follows:

Listing 3: Composed Action Rule

```
package named "a.b" must not (call or access)
package named "x.y";
```

B. Property Rules

In contrast to a *usage rules*, a *property rule* is asymmetric: it consists only of a subject and an action (Figure 3). While the subject is specified exactly as in *usage rules*, the action is specified differently, namely by the `have` keyword.

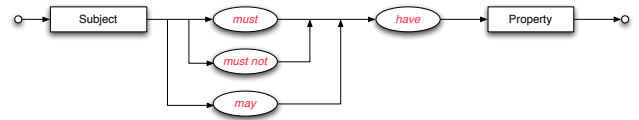


Figure 3: Specification of Property Rules

Below is an example of a *property rule* stating that the system is not allowed to contain any classes that have the *Data Class* design flaw, namely classes that are dumb data holders without complex functionality, on which other classes strongly rely in terms of data-access [20]:

Listing 4: A Simple Property Rule

```
classes must not have "Data Class";
```

As mentioned before, these properties (*e.g.*, "Data Class") are made available to `inCode.Rules` by the underlying software analysis infrastructure (*i.e.*, `inCode`), as seen in Figure 5. Properties are expressed as *property strings* and they correspond to a continuously growing set of quality assessment analyses defined in `inCode`[9]. This means that `inCode.Rules` is also continuously enriching its "vocabulary" as it may use any of those externally available quality properties.

Furthermore, properties can be **composed** using the `or` and `and` composition operators, which consequently allows us to write more complex *property rules* like in the following example:

Listing 5: Composed Filter

```
classes must not have
( "Data Class" or "God Class" );
```

C. Defining Exceptions

"*Rules are meant to be broken*". This saying holds in no other engineering field better than in software development. *Change* is an intrinsic property of software, and it would be foolish to think that a set of design decisions (let alone rules) will be valid and respected throughout the entire lifecycle of the system. This is the main reason why the `inCode.Rules` language supports the concept of **exceptions**.

There is also a second reason for introducing exceptions: these language constructs increase the expressivity of the language, increasing thus their readability. For instance, consider a package `org.x` with four classes A, B, C and D. The design states that package `org.x` is not allowed to use package `org.y` except for class D. If exceptions did not exist we would have to write three rules to code the design, one for each class except class D. With exceptions we only need to write one rule and one exception. At first this might not seem that much of an improvement, but if the package `org.x` contained

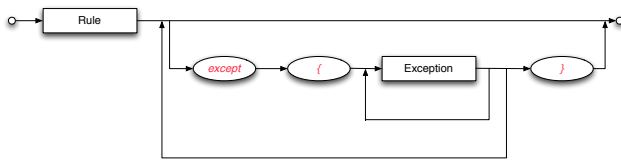


Figure 4: Specification of Exceptions

20 classes, it becomes clear that the language would simply not scale without the concept of rule exceptions.

Exceptions are an optional part of a rule and they appear after the rule definition, between braces, and before the semicolon (see Figure 4). Let's see how exceptions are used in a concrete example :

Listing 6: Exception

```
package named "org.x" must not use package named
  "org.y"
except {
  class named "org.x.ThisClass"
  may use class named "org.y.ThatClass"
};
```

One remarkable thing in the previous example is that the definition of an exception looks a lot like a rule. Actually, from the grammar point of view, *exceptions are rules*. It is the way that they are interpreted which makes them exceptions. There is only one constraint that applies to exceptions: the *action specifier* of the exception must be *opposite or neutral to the action specifier from the rule that contains the exception*. For example, if the main rule has a *must not* action specifier, then every exception attached to this rule has *must use* either *may* or *must* as an action specifier. The table below captures the acceptable matches between the action specifiers in the main rule and its exceptions:

Rule	Exception
must	may / must not
must not	may / must

Because the `inCode.Rules` language supports two types of rules (*i.e.*, usage and property rules), there are two types of exceptions as well. Actually, since exceptions are rules, it is quite easy to remember that *usage rules* can only have exceptions that are themselves *usage rules*, while the exceptions of *property rules* must be *property rules* as well.

Nested Exceptions: Since exceptions are essentially rules, it means that they too can have their own exceptions! This can best illustrated by an example: package `org.x` is not allowed to use package `org.y`. However, class `org.x.A` is allowed to use package `org.y` ... except for method `foo()` of class `org.x.A` which is not allowed to use anything defined in package `org.y`. To specify such a complex rule in `inCode.Rules` is quite simple :

Listing 7: Exception to an exception

```
package named "org.x" must not use
package named "org.y"
```

```
except {
  class named "org.x.A"
  may use package named "org.y"
  except {
    method named "org.x.A.foo"
    must not use package named "org.y" };
};
```

This exception mechanism is recursive, which means that an architect can write as many nested exceptions as she likes. Having this said, in our experience it is neither easy nor meaningful to nest more than three levels *i.e.*, the main rule plus two levels of exceptions. Furthermore, it is also not recommendable to use a deep nesting of exceptions as this would significantly diminish the understandability of the rule. Based on our experience we recommend the following use of the exception mechanism:

- Keep exception imbrications at a minimum, three (or four) levels should be the maximum any rule should have.
- When writing an exception, the designer usually refers to a subset of the target or subject. Therefore it is acceptable to just write the new subject and copy the rest of the rule. However, one should not forget to change the action qualifier.

Based on the previous recommendations, we introduced in `inCode.Rules` two *syntactical simplifications*, for the sake of a better readability: (i) if a rule has only one exception, surrounding the exception rule by braces is optional; (ii) exceptions to *property rules* might specify only the subject as the action and property can be inferred from the main rule.

The example below illustrates the increased readability due to the two syntactic simplifications:

Listing 8: Exception with exception

```
package named "org.x"
  must not have "Data Class"
  except class named "org.x.A";
```

IV. INCODE.RULES: THE ECLIPSE-INTEGRATED TOOL

The previous section has illustrated the simplicity and the flexibility of specifying architectural constraints using `inCode.Rules`. Now, we will present the tool support that accompanies the language specification, and which insures its immediate usability and agility. This section will reveal tight integration of `inCode.Rules` with the Eclipse development environment, and thus, the way it bridges the gap between architecture specification and actual development.

A. The Architecture

Figure 5 displays a the architecture of how `inCode.Rules` connects with other Eclipse plugins. At the lowest level it relies on the Eclipse infrastructure, which is far more complex than perceived from the IDE. Primarily, we rely on Xtext [21] for specifying the `inCode.Rules` language. Xtext is a framework used for the development of Domain Specific Languages (DSLs). Xtext is based on the Eclipse Modeling Framework (EMF), and it integrates technologies such as the Graphical Modeling Framework,

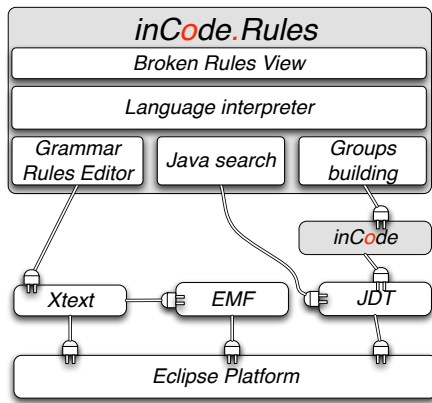


Figure 5: Architecture of `inCode.Rules`

Model to Text (M2T) and some parts of the Eclipse Modeling Framework Technology. In order to use Xtext we wrote the grammar of the `inCode.Rules` language using the Xtext notation, which in return allows Xtext to generate all the needed features. On top of XText we built the *Grammar Rules Editor* and the *Language Interpreter*. Furthermore, as `inCode.Rules` is focused on verifying the architecture of Java project, we use Eclipse’s Java Development Toolkit (JDT), as well as its powerful Java Search feature, for getting access to information about the actual code of the analyzed project. From a user perspective the interaction with `inCode.Rules` is provided by the *Broken Rules View* as described in Section IV-C.

Last, but not least, `inCode.Rules` builds on top of the `inCode` Eclipse plugin [9], which provides developers with the support for the detection and correction of a large set of design flaws (e.g., God Class, Data Class, Feature Envy etc). The design problems are detected using metrics-based rules, called detection strategies [22]. The set of these detection rules is continuously growing and, as we have shown in the previous section, `inCode.Rules` can benefit from that as any of the detection rules can be used in the context of a *property rule*. The specificity of `inCode` is that it allows the detection of design flaws in real time. The programmer is warned immediately after the file save that a design flaw has been detected.

B. The `inCode.Rules` Editor

Although the `inCode.Rules` language is very close to natural language, writing architectural constraints in `inCode.Rules` is made even easier by the full-fledged editor integrated in the Eclipse environment. Starting from the `inCode.Rules` grammar and based on the support offered by the Xtext framework, we created a dedicated editor with advanced capabilities like syntax highlighting and a powerful auto-complete feature.

Advanced Auto-Complete: From the grammar, Xtext generates a content proposal system that is able to determine which keywords could come next. This content proposal system can work because the parser is a LL parser, this means

that it can create a partial AST of a rule, even if that rule is not complete. It can even create the AST (obviously a incomplete AST) of a rule that has only the first word. Figure 6, which is a screenshot of the `inCode.Rules` editor demonstrates the capability of the content assist system to determine the next possible keywords. It ‘knows’ that after the keyword “class” the rule can continue with the optional selector (and it proposes “(”, “named”, “being”, “from”) or directly with the action (and it adds to the proposed keywords the “may” and “must” options). After creating the AST the content proposal system can easily determine which of the keywords (or other grammar elements, like strings, or semicolon) could come next.

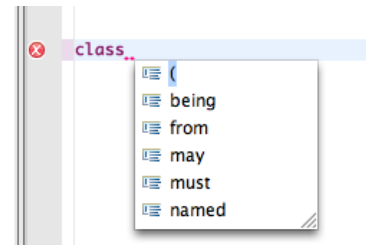


Figure 6: Standard code completion

However, there is a type of problems that Xtext cannot handle just by using the grammar. Its the case of auto-completing the *strings that represent Java element names or the selector and property names*, for which Xtext needs further information from JDT or `inCode`. As seen in Figure 7 we extended the standard capabilities of the editor with the possibility of auto-completing even the names of the selectors and properties that are loaded by `inCode` at startup through a reflection mechanism. This means that the selectors and properties may vary from one `inCode` installation to another. This also affects the proposal mechanism as only the available code proposals will be presented to the architect who is writing the constraints.

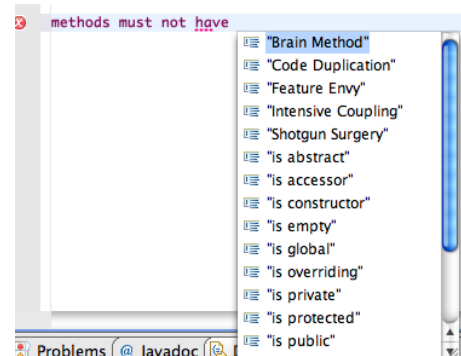


Figure 7: Advanced code completion for properties

Syntax Highlighting: The `inCode.Rules` editor is compliant with the guidelines of the Eclipse UI. By default the keywords have the same default color of the java language keywords. The same goes for the ID terminals and for the string terminals.

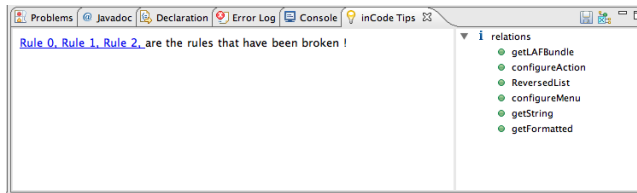


Figure 9: The Broken Rules View

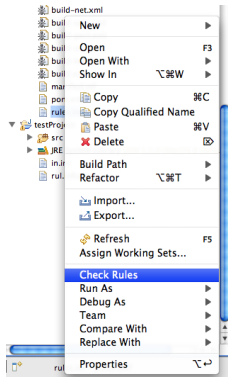


Figure 8: Running the `inCode.Rules` Checker

C. Running the Rule Checker

After the architectural constraints have been edited in an `inCode.Rules` file, the rules can be checked for conformance in the source-code. To run the interpreter on a rules file the user must right click (Figure 8) the rule file in the Package Explorer View, or any other Eclipse view that displays files and select "Check Rules". As a result, the rule interpreter visits each rule and checks if the code is compliant to the rule. After the rules have been checked, the *Broken Rules View* (Figure 9) opens automatically and shows the results. If there are no broken rules a simple message is displayed saying so. If, however there are broken rules, they are displayed in a list using either their names (ids) or their order number (if the rule does not have a name).

As shown in Figure 9 the broken rules are enumerated in a list separated by a comma and their names (or rule numbers) are hyperlinks. Clicking one of the hyperlinks brings up the tree on the right. The tree on the right side of the view is a one level tree that shows the "relations" that caused the rule to break. For example if the rule was violated by a method call, in the tree, the name of the called method appears. When double clicking the rule `inCode.Rules` opens the editor and highlights the exact location of the method call.

D. Performance Assessment

In order to assess the time performance of checking architectural rules with `inCode.Rules` we analyzed two well-known open-source systems, namely: JHotDraw 7 (cca. 100.000 LOC) and Eclipse.JDT.UI (cca. 1.000.000 LOC). For each we defined a set of 20 architectural rules at various levels of granularity *i.e.*, some rules were coarse-grained (any type of dependencies at package level), while others were very fine-

grained (specific types of dependencies at the method/attribute level). All experiments were run on a machine with Intel Core2Duo 2.3GHz processor and 2 GB of RAM.

Although, all tests did successfully complete (even on the 1 MLOC system!) the time performance had large variations, going from 26 seconds (JHotDraw, with coarse-grained rules only) to 17 minutes (Eclipse.JDT.UI with all rules included).

By analyzing these variations of execution times we found out the following:

- Execution times *increase proportionally* with the number of rules, if all rules are defined at the same granularity level.
- Execution times *increase significantly* when rules are more fine-grained.
- One of the causes is the latency of Eclipse JDT. For instance, it takes about 13 seconds to gather a group of about 6000 classes from the 1 MLOC Java project.
- Another cause is the time performance of the `inCode` plugin, especially when it comes to computing the different metrics needed in the *property rules*. In this context, by doing some explorations on a potential caching solution we were able to get a performance increase of up to 400% (from 17 minutes to 4 minutes). However, this solution needs yet a more thorough investigation and testing.

V. CONCLUSIONS.FUTURE WORK

In this paper we presented the `inCode.Rules` ADL for describing and checking architectural rules for Java systems. Our approach has followed three main characteristics: (i) *simplicity*, (ii) *flexibility* and (iii) *integration* of architecture description with the actual development. The language allows one to write two types of design rules: *usage rules* and *property rules*. It also has an exception mechanism that is used to describe the architecture more accurate and to allow the architecture to be flexible enough so that it can be extended and modified with ease. The language is implemented as an Eclipse plugin, and thus is part of the Eclipse ecosystem, probably the most widely used environment for Java development.

We plan two types of future work related to enhance `inCode.Rules`:

- 1) *Continuous checking of architectural constraints*. Our first major future is to reach the supreme agility in architecture verification, namely checking architectural rules in *real-time*. In order to reach this goal, we will have to find ways to improve the time performance even more. Currently, although we already successfully used `inCode.Rules` on very large projects (over 1

MLOC), execution time is still an issue when the system is very large and the set of rules is very large and at the same time the rules are too fine-grained. Thus, we are currently exploring solutions based on disk caching to address this issue.

- 2) *Adding language support for reflective rules.* A reflective rule would sound something like : "methods from 'org.x' must access their own classes". This would mean that each method must respect the rule but only with respect to its own class. Two things are required for this: (i) iteration over the subject group, as we have to apply the rule to each method separately and (ii) the reflective part, as we need to identify the target by using the entity in the subject. Removing this limitation is one of the top priorities in the evolution of the language.

ACKNOWLEDGMENT

We gratefully acknowledge the financial support of the Romanian National Science Foundation (CNCSIS) for the project "Methods and Tools for Continuous Quality Assurance in Complex Software Systems" (PNII-IDEI 357/1.10.2007).

REFERENCES

- [1] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. Thesis, University of Illinois, 1992. [Online]. Available: <ftp://st.cs.uiuc.edu/pub/papers/refactoring/http://www.laputan.org/pub/papers/opdyke-thesis.pdf>
- [2] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural mismatch: Why reuse is so hard," *IEEE Softw.*, pp. 17–26, 1995.
- [3] M. Feilkas, D. Ratiu, and E. Juergens, "The loss of architectural knowledge during system evolution: An industrial case study," *ICPC 09: Proc. of the 17th IEEE International Conference on Program Comprehension*, 2009, to appear.
- [4] R. W. Schwanke, V. A. Strack, and T. Werthmann-Auzinger, "Industrial software architecture with gestalt," *IWSSD '96: Proceedings of the 8th International Workshop on Software Specification and Design*, 1996.
- [5] K. Mens and R. Wuyts, "Declaratively codifying software architectures using virtual software classifications," *In Proceedings of TOOLS-Europe 99*, pp. 33–45, 1999.
- [6] D. L. Parnas, "Software aging," in *Proceedings 16th International Conference on Software Engineering (ICSE '94)*. Los Alamitos CA: IEEE Computer Society, 1994, pp. 279–287.
- [7] M. Dalgarno, "When good architecture goes bad," *Methods Tools*, Spring 2009.
- [8] F. Budinsky, S. A. Brodsky, and E. Merks, *Eclipse Modeling Framework*. Pearson Education, 2003.
- [9] R. Marinescu, G. Ganea, and I. Verebi, "inCode: Continuous quality assessment and improvement," in *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*. IEEE Computers Society Press, 2010.
- [10] R. Allen and D. Garlan, "A formal basis for architectural connection," *ACM Transactions on Software Engineering and Methodology*, July 1997.
- [11] M. Moriconi, X. Qian, and R. A. Riemenschneider, "Correct architecture refinement," *IEEE Trans. Software Engineering*, 21 (4), April 1995.
- [12] D. C. Luckham and J. Vera, "An event-based architecture definition language," *IEEE Trans. Softw. Eng.*, pp. 717–734, 1995.
- [13] J. Magee and J. Kramer, "Dynamic structure in software architectures," *SIGSOFT Softw. Eng. Notes*, pp. 3–14, 1996.
- [14] J. Aldrich, C. Chambers, and D. Notkin, "Archjava: connecting software architecture to implementation," *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, 2002.
- [15] I. Lattix, "<http://www.lattix.com/news/articles/lattix50.php>."
- [16] N. Sanga, E. Jordan, V. Sinha, and D. Jacksion, "Using dependency models to manage complex software architecture," 1995.
- [17] M. Fowler, "Who needs an architect?" *IEEE SOFTWARE*, vol. 20, no. 5, pp. 11–13, 2003.
- [18] G. Ganea, "Defining and checking complex architectural rules in eclipse," Master's thesis, Politehnica University of Timisoara, 2009 <http://loose.upt.ro/download/papers/incoderules.pdf>.
- [19] A. Riel, *Object-Oriented Design Heuristics*. Boston MA: Addison Wesley, 1996.
- [20] M. Lanza and R. Marinescu, *Object Oriented Metrics in Practice*. Springer, 2006.
- [21] H. Behrens, K. Wannheden, S. Efftinge, and S. Zarnekow, "<http://wiki.eclipse.org/xttext/documentation>."
- [22] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *20th IEEE International Conference on Software Maintenance (ICSM'04)*. Los Alamitos CA: IEEE Computer Society Press, 2004, pp. 350–359.