Ensuring Well-Behaved Usage of APIs through Syntactic Constraints

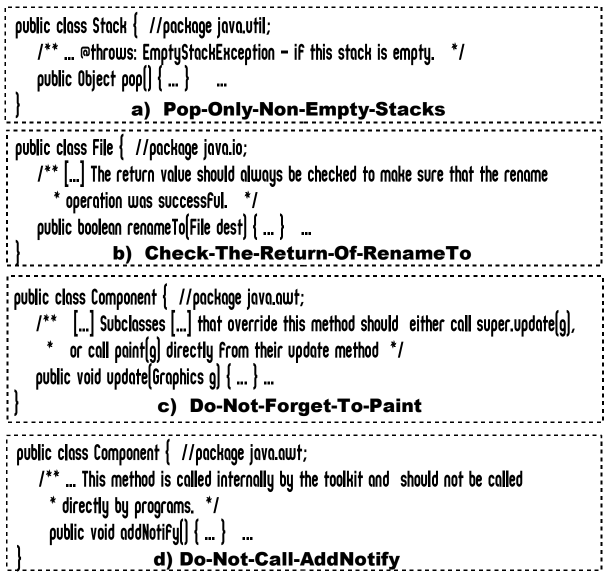
Martin Feilkas and Daniel Ratiu
Institut für Informatik
Technische Universität München
Boltzmannstr. 3, D-85748 Garching
`feilkas|ratiu@in.tum.de`

Abstract

Libraries are the most widespreaded form of software reuse. In order to properly use a library API, its clients should fulfill a series of (many times implicit) assumptions made by the API programmers. Failing to fulfill these assumptions leads to a misuse of the library and thereby to defects in the client's code. In this paper we present a method for checking a well-behaved usage of an API through a set of context-sensitive syntactic constraints over the API clients. These constraints restrict the set of programs that can be written with an API only to programs that fulfill the API assumptions and thereby represent a well-behaved and valid usage of the API. In this paper we present a set of typical assumption classes made by API providers about their clients. We define a framework for formalizing the context-sensitive constraints over the API client code and propose a typical constraint for each class of assumptions. Thereby we provide a mechanism that allows the provider of an API to describe the knowledge of how an API is intended to be used in an automatically checkable form. We present our experience with parts of the Java Standard APIs.

1 Introduction

Domain-specific libraries represent the most widely used form of software reuse. They offer implementations for the concepts of a certain domain through program abstractions such as classes and methods. Most of the times the API providers and their users are totally decoupled. Thereby, the users are left alone to understand the library and to properly use the library-defined abstractions for implementing their programs. Today programming languages provide a series of language mechanisms – e.g. checked exceptions, static typing, visibility rules – for enforcing a good usage of an API. However, only by writing programs that compile, the users of a library can not be sure that they used the library in a correct, safe and well-behaved manner. Besides the knowledge made explicit through the programming language, an API contains a considerable amount of tacit knowledge about its domain. This knowledge is captured implicitly in a series of assumptions about the clients code and is most of the times only partially reflected, if ever, in documentation, tests or informal API usage examples. In Figure 1 we present examples of such assumptions encountered in the Java API. Using an API non-conform to its foreseen usage scenarios can lead to misuses of the API. This in turn not only generates (latent) bugs or extensibility problems but also leads to problems in understanding and maintaining the API clients.



```
public class Stack {  //package java.util;
   /** ... @throws: EmptyStackException - if this stack is empty.  */
   public Object pop() { ... }    ...
}
```
a) Pop-Only-Non-Empty-Stacks

```
public class File {  //package java.io;
   /** [...] The return value should always be checked to make sure that the rename
      * operation was successful.  */
   public boolean renameTo(File dest) { ... }   ...
}
```
b) Check-The-Return-Of-RenameTo

```
public class Component {  //package java.awt;
   /**   [...] Subclasses [...] that override this method should  either call super.update(g),
     *   or call paint(g) directly from their update method  */
   public void update(Graphics g) { ... } ...
}
```
c) Do-Not-Forget-To-Paint

```
public class Component {  //package java.awt;
   /** ... This method is called internally by the toolkit and  should not be called
      * directly by programs.  */
      public void addNotify() { ... }   ...
}
```
d) Do-Not-Call-AddNotify

Figure 1. Assumptions in the Java API

In this paper we propose a framework (Section 3) to describe API assumptions as a set of API usage patterns that reflect the envisioned well-behaved usage of the API. We advocate that often the assumptions are expressible as context-sensitive constraints over the syntax of the API clients (Section 4). In Section 5 we present our experience with several assumptions in the Java library. Section 6 presents the related work and Section 7 concludes the paper.

2 Well-behavedness in using the Java APIs

Below we present examples of assumptions about the well-behaved usage of the Java standard API. The examples are categorized in assumption classes, each class being described in a paragraph. Each of the following examples represent a case of tacit knowledge about how to properly use the API.

Assumption about the system state. The properties of the state of the system before and after a method is executed are of capital importance. They are expressed in plain English through phrases such as: *"before calling X check that ..."*. The API provider makes an assumption that before calling a method the clients should check that the system is in the required state. A well known case where assumptions about the system's state are present is the method 'pop' of the class 'java.util.Stack' (Figure 1a – *"Pop-Only-Non-Empty-Stacks"*). The Java API providers assume that 'pop' is not called on an object representing an empty stack. Otherwise the contract of the 'pop' method is broken and this results is an Java runtime exception. This is a latent bug which manifests only at runtime.

Assumptions about the communication. Another class of assumptions about the clients regards the communication between modules through the parameters and return values of functions. These assumptions constrain the arguments that should be passed to methods or how the caller should deal with the return value. As an example we present the case of the method 'renameTo' of the class 'File' (Figure 1b – *"Check-The-Return-Of-RenameTo"*). In the javadoc of this method is a strong advice to check the value returned by this method since it informs about the success or failure of the rename operation. This is an example of a case in which the return values of methods have special meanings and should not be ignored. Failing to fulfill this assumption leads to latent bugs in the case when the rename fails. Our manual inspection of the previous versions of the API revealed that this comment was introduced only since Java 1.4.2. [1].

Assumptions about the extensions. The third class of assumptions is related to extensions that are performed to an API. The extensions are very frequent especially in the case of frameworks. In plain English these assumptions are expressed through sentences like: *"whenever you extend X you should do Y"*. For example, the subclasses of the class 'Component' that override the method 'update' should take care themselves about the painting of the component (Figure 1c – *"Do-Not-Forget-To-Paint"*). Failing to meet this assumption leads to components that are not painted properly. Our manual inspection of the previous versions of the

```
ClsDecl ::= Visibility "class " Name " extends " ClsInh"{" (FieldDecl |MethDecl | MethOv)* "}"
MethOv ::= MethDecl
MethDecl ::= Visibility RetType Name "(" [(ParamDecl ",")* ParamDecl]") {" StmtBlock "}"
StmtBlock ::= MethInv | IfStmt |WhileStmt | ...
IfStmt ::= "if(" Expr ") {" StmtBlock "}" ["else{" StmtBlock "}"]
MethInv ::= [ObjName "." ] Name "(" [(Arg ",")* Arg] ")"
```

Figure 2. Fragment of the Java grammar

API revealed that this comment was introduced only since Java 1.4.2.

Scoping assumptions. The fourth class of assumptions regard architecture. In plain English these assumptions are described through sentences like: *"method X should not be called by the API clients"*. These cases are many times instances of the well known limitation between public vs. published interfaces [5]. For example, the method 'addNotify', even if public, is not intended to be directly called by the API clients (Figure 1d – *"Do-Not-Call-AddNotify"*). Failing to address this assumption can result in anomalies in the framework's operation. Our manual inspection of the previous versions of the API revealed that this comment was already introduced in the early Java API version 1.1.

3 Anatomy of domain-specific constraints

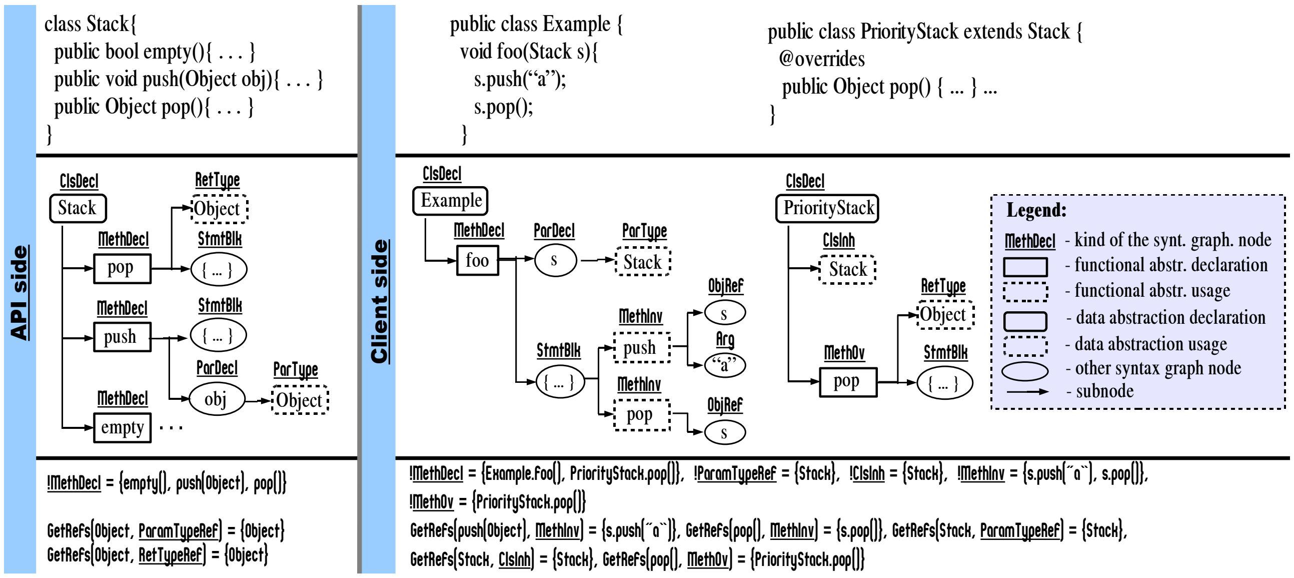
In this section we introduce a formal framework for defining syntactical constraints over the clients of a domain-specific API. Firstly, we give a brief introduction of constraints that a language provides for restricting the set of valid programs. Secondly, we define a formalism that enables the definition of constraints over the domain abstractions defined in an API. In Figure 3 we present an example of our framework: in the upper part is the Java source code, the corresponding syntax graphs are sketched in the middle and in the lower part we present examples of our formalism. In Section 4 we instantiate this framework to express syntactic constraints that can be used for automatically checking the assumptions presented in Section 2.

Constraints in programming languages. A programming language provides a set of language constructs. In this paper we focus on the following set of constructs of Java, shown also in Figure 2:

$$C_{Java} = \{\underline{ClsDecl}, \underline{MethDecl}, \underline{MethInv}, \underline{StmtBlock}, ...\}$$

Each of these constructs corresponds to the kinds of nodes that may appear in an abstract syntax graph of a Java program. Implementing a program is done by instantiating the constructs of the language. The grammar of a language defines the basic structure of construct compositions that are acceptable in programs – e.g. the BNF fragment of the Java grammar presented in Figure 2 ensures that the declaration of fields ($\underline{FieldDecl}$) occurs only inside a class declaration ($\underline{ClsDecl}$). However, the context-free grammar of the language is not capable of expressing all the constraints that

[1]The version 1.3.0 of the API does not have this comment `http://java.sun.com/j2se/1.3/docs/api/java/io/File.html`



```
class Stack{
  public bool empty(){ . . . }
  public void push(Object obj){ . . . }
  public Object pop(){ . . . }
}
```

```
public class Example {
  void foo(Stack s){
    s.push("a");
    s.pop();
  }
```

```
public class PriorityStack extends Stack {
  @overrides
  public Object pop() { ... } ...
}
```

!MethDecl = {empty(), push(Object), pop()}

GetRefs(Object, ParamTypeRef) = {Object}
GetRefs(Object, RetTypeRef) = {Object}

!MethDecl = {Example.Foo(), PriorityStack.pop()}, !ParamTypeRef = {Stack}, !ClsInh = {Stack}, !MethInv = {s.push("a"), s.pop()},
!MethOv = {PriorityStack.pop()}
GetRefs(push(Object), MethInv) = {s.push("a")}, GetRefs(pop(), MethInv) = {s.pop()}, GetRefs(Stack, ParamTypeRef) = {Stack},
GetRefs(Stack, ClsInh) = {Stack}, GetRefs(pop(), MethOv) = {PriorityStack.pop()}

Figure 3. Defining and using stacks

are needed in order to ensure that the constructs are only used correctly. In order to capture these situations, context-sensitive constraints like: "every variable must be declared before it is used" or "the methods a class implements from an interface must be public" are added to the programming language and enforced by the parser (during so called semantic analysis).

`Notation`: **1)** In this paper we use only the language constructs of Java that are defined in the (highly simplified) fragment of a Java grammar from Figure 2. When referring to these language constructs we will always use an underlined font. **2)** We use a Java-like notation to navigate over the syntax graph. So we will write $x.Y$ for the set of nodes of type $Y \in C_{Java}$ that are connected to a specific AST node x – e.g. if x is a node of type $\underline{MethDecl}$ then $x.\underline{Visibility}$ represents the visibility of the method. **3)** We write $!c$ for the set of instances of the construct in c $(\subseteq C_{Java})$ within a program – e.g. $!\underline{ClsDecl}$ is the set of all class declarations in a program. **4)** To make our notations more readable we assume that every $\underline{Name}$ (Identifier) in the abstract syntax graph is fully-qualified. **5)** For the same reasons we also introduce the notation $x \sqsubset y$ which means that x is contained as a subgraph in y – e.g. we will write $inv \sqsubset exp$ meaning that an instance inv of a method invocation appears in the subgraph of an expression exp.

Constraints over the usage of domain-abstractions. Programming languages enable programmers to represent concepts that are needed to describe their domain of interest. The abstraction mechanisms of Java (a set $AM \subset C_{Java}$) are central for the definition and the usage of domain concepts through libraries. An abstraction mechanism $(\in AM)$ contains two kinds of language constructs:

– *Concept declarations* (AM_{decl}) enable programmers to define new domain-specific concepts. These constructs are used by library programmers to provide the implementation of domain concepts to their clients.

– *Concept bindings* (AM_{use}) enable programmers to use a domain-specific concept. This kind of constructs are used by the API clients to access and use the implementation of domain concepts in an API.

In the following we consider the core object-oriented abstractions mechanisms (data abstraction and procedural abstraction):

$$AM_{decl} = \{\underline{ClsDecl}, \underline{MethDecl}\}$$
$$AM_{use} = \{\underline{ClsInh}, \underline{MethInv}, \underline{ParaTypeRef}, \underline{MethOv}\}$$

In Figure 3 (left) we present a class declaration and three method declarations as examples of elements of AM_{decl}. On the client side (right) we show different kinds of usages of the declarations such as an invocation and an overriding of the 'pop' method.

Let C_D be the set of domain-specific concepts that are defined using the abstraction mechanisms (AM) – e.g. in Figure 3 (left) we have: $C_D = \{Stack, empty, pop, push\}$. The compiler accepts every usage of a concept implementation as long as it adheres to the language rules – e.g. every invocation of the 'pop' method is allowed even if the stack could be empty. However, many concept implementations that are provided by APIs cannot be used properly in every situation that the programming language would accept. This happens because the domain-concepts come along with specific constraints. Just like the programming language defines context-sensitive constraints to allow only correct usage of its built-in constructs, the implementation of domain-concepts C_D also come along with certain restrictions that cannot be expressed within programming languages.

In order to identify the references to a certain declaration in a program, we define the function $GetRefs$ as follows: for a declaration $d \in !AM_{decl}$ and a usage mechanism $u \in AM_{use}$, $GetRefs(d, u)$ returns the nodes in the syntax graph of type u that are usages of d. As Figure 3 illustrates we can use this function for example to select all the usages of the 'pop' method that appear as method invocations ($\underline{MethInv}$) or all the usages of pop due to overriding ($\underline{MethOv}$):

$$GetRefs(pop, \underline{MethInv}) = \{s.pop()\}$$
$$GetRefs(pop, \underline{MethOv}) = \{PriorityStack.pop()\}$$

Domain-specific constraints. Domain-specific abstractions are always defined using a declaration mechanism. To use a concept an adequate usage mechanism is needed. For the definition of a concept-specific constraint we need three ingredients: the declaration of the concept, the usage mechanism that should be covered by the constraint and a specification of the context in which the usage of the concept is allowed. Formally, we define a constraint $\mathcal{C}$ to be:

$$\mathcal{C} = \langle Decl, Use, Ctxt\rangle$$

where $Decl$ is a concrete instance of a declaration $\in !AM_{decl}$ (e.g. java.util.Stack.pop); Use is a usage mechanism $\in AM_{use}$ that is applicable to the type of Decl (e.g. $\underline{MethInv}$); $Ctxt$ is a predicate calculus formula that defines the context of the usage (e.g. see Equation 1).

The semantics of a concept-specific constraint is:

$$\forall GetRefs(Decl, Use) : Ctxt$$

In plain English, this means that for every usage of a specific declaration a certain context must be present. We will use the identifier "use" as a variable within the $Ctxt$ of a constraint to access the usage that should fulfill the constraint. A constraint that enforces a well-behaved usage of pop is:

$$\begin{aligned} PopInIf = \langle &\mathbf{Decl} : java.util.Stack.pop(),\ \mathbf{Use} : \underline{MethInv}, \\ &\mathbf{Ctxt} : \exists i \in !\underline{IfStmt} : \exists m \in \underline{MethInv} : \\ &m \sqsubset i.\underline{Expr} \wedge use \sqsubset i.\underline{StmtBlock} \wedge \\ &m.\underline{ObjName} = use.\underline{ObjName} \wedge m.\underline{Name} = \text{“}empty\text{”}\rangle \end{aligned} \tag{1}$$

Intuitively, this constraint demands that every invocation of the 'pop' method must reside in an if-statement that contains an expression which checks that the stack might be empty.

4 Expressing assumptions as constraints

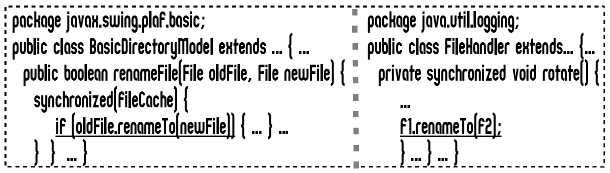
In the following we instantiate our framework presented in the previous section to define constraints on the API client code in order to tackle the assumptions from Section 2. To each class of assumptions corresponds a family of constraints over the syntax as described in the following paragraphs. We present a formalization for each constraint, give an intuitive explanation and discuss the possible variations and limitations of the constraints.

Control-flow Constraints. The Java programming language accepts every sequence of method invocations. But in practice, the anticipated use of many API methods depends on the state in which the system is. If the system is not in the anticipated state then this usually causes exceptions at runtime. Thus, before the clients invoke such methods, they must assure themselves that the system is in the desired state. For example, the "Pop-Only-Non-Empty-Stacks" assumption can be made explicit through the $PopInIf$ constraint (as defined in Equation 1). By enforcing the client code to be compliant to this constraint, the maintainers see explicitly (in the syntax) that the client code took the case that the stack might be empty into account. Additionally to the $PopInIf$ constraint we could add more flexibility by allowing also while-loops that check for emptiness. The constraint might also be further refined by allowing 'pop' invocations if 'push' has been invoked directly in front of them.

Data-flow Constraints. The communication between modules is modeled through data-flow. We use the data-flow constraints to address the communication assumptions presented in Section 2. We tackle the "Check-The-Return-Of-RenameTo" assumption through the following constraint:

$$\langle \mathbf{Decl} : java.io.File.renameTo(File),$$
$$\mathbf{Use} : \underline{MethInv}, \mathbf{Ctxt} : \exists i \in !\underline{IfStmt} : use \sqsubset i.\underline{Expr}\rangle$$

Intuitively, this constraint makes sure that the 'renameTo' method invocations are always within the condition of if-statements. A variation of this constraint is the situation when the return value is saved into a local variable that is further checked in a if-expression (extract-local-variable refactoring). However, due to the lack of space we do not tackle this case here. In Figure 4 we provide two examples of the well-behaved (left) and non well-behaved (right) usage of the 'renameTo' method in the Java library.



```
package javax.swing.plaf.basic;
public class BasicDirectoryModel extends ... { ...
  public boolean renameFile(File oldFile, File newFile) {
    synchronized(fileCache) {
        if (oldFile.renameTo(newFile)) { ... } ...
    } }  ... }
```

```
package java.util.logging;
public class FileHandler extends... {...
  private synchronized void rotate() {
      ...
      f1.renameTo(f2);
      } ... } ... }
```

Figure 4. Clients of the "renameTo" method

Inheritance Constraints. Inheritance is one of the most important extension mechanisms of object-oriented languages. Inheriting from a class often implies that the subclass must be compliant to certain invariants assumed by the super-class. We use the constraints on inheritance in order to address the extension assumptions from Section 2. The "Do-Not-Forget-To-Paint" assumption is made explicit through the constraint:

$$\begin{gathered}\langle \mathbf{Decl} : java.awt.Component.update(Graphics), \\ \mathbf{Use} : \underline{MethOv}, \mathbf{Ctxt} : \exists m \in !\underline{MethInv} : \\ m \sqsubset use.\underline{StmtBlock} \wedge \\ m.\underline{Name} = \text{“}update\text{”} \wedge m.\underline{ObjName} = \text{“}super\text{”}\rangle\end{gathered}$$

Intuitively, each method that overrides ‘update’ from the class ‘Component’ must call ‘update’ on the ‘super’ reference. In order to keep the constraint relatively simple, we restrict only the form of the client in the sense that the update method should be called – we do not completely check whether this method is properly called. Our analysis of the subclasses of ‘Component’ in ‘java.awt’ revealed (not surprisingly) that all of them are well-behaved.

Scoping Constraints. Many times the difference between public and published interfaces can be made explicitly checkable through an advanced scoping mechanism such as the one given by the following constraint that tackles the “Do-Not-Call-AddNotify” assumption:

$$\begin{gathered}\langle \mathbf{Decl} : java.awt.Component.addNotify, \\ \mathbf{Use} : \underline{MethInv},\ \ \mathbf{Ctxt} : false\rangle\end{gathered}$$

Intuitively, this constraint expresses the fact that, given a set of API clients, the method ‘addNotify’ should never be called (‘false’ expresses that there is no valid context from which the method can be called). This constraint is very simple but nevertheless it is very common since many methods of frameworks should not be ‘published’ to the clients although they are declared ‘public’.

5 Experience

Pervasiveness of the constraints. Our programming experience tells that the proper usage of an API contains many assumptions expressible as constraints over their clients' syntax. In order to check how pervasive the need for constraints is, we inspect the constraints that are explicitly documented in the javadoc of the Java standard API. We identified the assumptions in the javadoc by searching for the following phrases: “must always”, “must not”, “must be called”, “should always”, “should not”, “should be called”. For each hit we manually inspected the method and decided whether it is a case of an API usage constraint that can be expressed as restrictions over the API clients syntax. In Table 1 we summarize our findings: the first column represents the part of the Java API that was inspected, the second column (Hits) represents the number of places in the javadoc that contain one of the above phrases. The third column presents the number of assumptions that we identified to be expressible as syntactic constraints after the manual inspection. On the right hand of our table we present the number of constraints in each of our categories (i.e. CF – control-flow, DF – data-flow, IN – inheritance, AR – architecture). We remark that from the places documented in the javadoc, on average over 30% represent cases of constraints of a syntactic nature. The bigger part of the other ones have been descriptions of the semantics of the APIs. Furthermore, we are convinced that there are many assumptions that we did not identify using our search method and there are also assumptions not even documented in the javadoc (as shown in Section 2 some of the assumptions are documented only in very late versions of the Java API).

Package	Hits	Synt	CF	DF	IN	AR
Java Util	23	6	1	0	4	1
Java IO	8	6	0	4	2	0
Java AWT	45	19	5	6	1	7
Eclipse JDT	37	12	4	1	4	3

Table 1. Assumptions in the Java API

Restrictiveness of the constraints. The question that we aim to answer in the following is: in what measure are the usages of the API well-behaved and comply to the syntactic constraints and in what measure are our constraints too strict? We answer this question by using our tool – the Java Constraints Checker. This tool is based on the Eclipse technology and enables us to define the constraints over API abstractions and check the client code whether it complies with our constraints or not. In order to perform our experiments we used the following sequence of steps: 1) define a syntactic constraint, 2) run the analysis and 3) inspect the results for false positives. The steps 1 and 3 are manual and the step 2 is done fully automatic with the help of our tool.

The following table presents the results of analysing the usage of Java API parts in Eclipse[2]. It shows how many of the usages of a certain concept have been found (Hits) and how many of these fulfilled the constraint (well-behaved).

By our manual inspection of the non-well-behaved usages of ‘renameTo’ we identified that seven of them have been found in test code. Within test code the violation of the constraint is less problematic than in system code. The other non-well-behaved usages ignore the renameTo-constraint and can be regarded as true-positives. We did not find any call to ‘addNotify’ so this constraint was always fulfilled. However, in the case of ‘pop’ we remark that most of the usages are not well-behaved. However, 109 of the 125 violations of our constraint are wrapper methods that assume that in turn their client code takes care about the

Constraint	Hits	Well-Beh.	Non-Well-Beh.
pop	144	19	125 (109 wrappers)
renameTo	20	11	9 (7 in tests)
update	4	4	0

Table 2. Evaluation of syntactic constraints

[2]Eclipse Core (org.eclipse.core.*) and JDT packages (org.eclipse.jdt.*)

state of the stack. So these warnings should be suppressed and a constraint similar to the 'pop' constraint should be introduced to these methods. In the other cases our manual inspection revealed that these usages occur in highly algorithmic pieces of code such as parsers. Although the stack may be used correctly, these situations are most often very long methods that are hard to understand. These results show that most of the times the usage of the domain concepts are well-behaved. If they are not well-behaved they are latent bugs or hints to code smells.

6 Related work

Dynamic approaches. David Parnas [2] presented a method for the specification of the behavior of software module interfaces using trace assertions. The traces that can be monitored during the execution of a system and then be used to verify the correctness of the usage of an interface. In contrast to our work trace assertions usually cannot be checked statically. There is much recent work [1, 11, 9] in mining the specifications of an API. These approaches concentrate on reverse engineering typical behavioral usage-patterns of APIs using tracing techiques. These usage-patterns are close to our control-flow constraints. A key difference is that we define the usage-patterns up-front and that we address a wider spectrum of possible usage of an API.

Constraint Languages. The closest work to our paper is the constraint language SCL [8, 7]. This language is intended to be used to define and check properties that client code of object-oriented frameworks has to fulfill. In this work we advance on this in two directions: Firstly, we point out and classify typical kinds of assumptions that the API developers make about their clients and relate them with kinds of syntactic constraints that the API users should fulfill. Secondly, we develop a formal framework for defining syntactic constraints. Our constraint framework is purely based on the grammar of a language, thus it is possible to specify constraints that regard domain concepts as well as language constructs. Furthermore the concept of syntactic constraints as presented in Section 3 is language independent and can be easily applied to other paradigms.

Static analysis. The properties that are checked by common static analysis like Findbugs[3], PMD[4] and others are on the abstraction level of the programming language and partially cover some of the low-level libraries (e.g. [6]). Although we have chosen examples from the Java API since it has many clients and is well known, our approach is targeted towards domain-specific APIs. Design-by-Contract methods like JML [10] also annotate constraints onto APIs using pre-/postconditions and invariants. By simply fulfilling these constraints the clients, even if they are provable to be semantically correct (e.g. using [4, 3]), they are not necessarily well-behaved and thereby it is not obvious for code-reviewers or maintainers to understand that the code is really correct. By restricting the form of the API clients we aim to increase the readability and support comprehension of the fact that the API is correctly used.

[3]http://findbugs.sourceforge.net/

[4]http://pmd.sourceforge.net/

7 Conclusions

Library developers make assumptions about the usage scenarios of the domain abstraction implemented in their APIs. But usually these assumptions are only tacit and are expressed (if ever) in textual documentation. A client of an API can not be sure that he uses the API in a way that fits to the expectations of the provider. Failing to use an API in a well-behaved manner can lead to latent bugs and problems in the comprehension and evolution of the clients. In this paper we identified a series of classes of assumptions commonly made by API programmers about their clients. We developed a framework for expressing these assumptions as syntactic constraints that restrict the form of the clients to the valid usages of the API. This enables the client to automatically check if his usage of the API is well-behaved.

References

[1] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1):4–16, 2002.

[2] W. Bartussek and D. L. Parnas. Using assertions about traces to write abstract specifications for software modules. In *2nd Conf. of the European Cooperation on Inf.* Springer, 1978.

[3] D. Detlefs, K. Rustan, M. Leino, G. Nelson, and J. Saxe. Extended static checking. Technical Report #159, Compaq SRC, 1998.

[4] C. Flanagan, K. Rustan, M. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for java. In *PLDI*. ACM, 2002.

[5] M. Fowler. Public versus published interfaces. *IEEE Softw.*, 2002.

[6] D. Gregor and S. Schupp. Stllint: lifting static checking from languages to libraries. *Softw. Pract. Exper.*, 36(3):225–254, 2006.

[7] D. Hou. Scl: Static enforcement and exploration of developer intent in source code. In *ICSE Companion*. IEEE, 2007.

[8] D. Hou and H. J. Hoover. Using SCL to specify and check design intent in source code. *IEEE Transactions on Software Engineering*, 32(6):404–423, 2006.

[9] J. Koskinen, M. Kettunen, and T. Systa. Profile-based approach to support comprehension of software behavior. *ICPC'06*.

[10] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion*, 2000.

[11] C. D. Roover, I. Michiels, K. Gybels, K. Gybels, and T. D'Hondt. An approach to high-level behavioral program documentation allowing lightweight verification. *ICPC'06*.