Evaluating application programming interfaces as communication artefacts

Luiz Marques Afonso[1], Renato F. de G. Cerqueira[1,2], and Clarisse Sieckenius de Souza[1]

[1] Departamento de Informática, PUC-Rio
{lafonso,rcerq,clarisse}@inf.puc-rio.br
[2] IBM Research Brazil
rcerq@br.ibm.com

Abstract. Application programming interfaces (APIs) allow the reuse of software artefacts by providing abstractions to other software layers, and their design is critical to enable the effective use of the underlying software and avoid programming errors. As such, the role of an API designer should be strengthened in any software project that has reuse among its goals. Also, we should be able to evaluate the effectiveness of an API in communicating its design to programmers and identify the tools and techniques that help the designers to accomplish this task, so that APIs may be easier to understand and use. This paper describes a work in progress that proposes the use of a combined semiotic and cognitive method to evaluate APIs as an artefact mediating the communication process between designers and programmers, and also aims to investigate some possibilities of enhancing this communication.
Keywords: POP-I.B Barriers to programming; POP-II.B Program comprehension; POP-III.C Cognitive dimensions; POP-V.B Research methodology

1 Introduction

Abstraction is one of the central concepts in Computer Science [30], permeating all the activities related to software construction and use. It is a cognitive resource that allows us to remove details in order to simplify things and focus attention on the core properties of a complex object [24]. As such, it seems intuitive that, although executed by a machine, a software artefact has its creation process deeply based on human interpretation.

Reuse is one of the main goals of Software Engineering. To be achieved, software reuse relies on abstractions in the form of libraries, components, objects, and other artefacts. At each level, software interfaces allow a programmer to construct new abstractions to be provided to another layer, exposing new concepts and design, and hiding internal details as needed.

Reusable software components are specified and implemented by programmers to be used by other programmers in order to construct new software. The reuse of a software artefact is achieved via its interfaces, which allow new software to call the available operations and create new functionalities on top of the existing abstractions. Programmers need to realise the concepts and the design behind the interfaces available in order to use them effectively. From a human-centric perspective, we can consider that a communication process takes place between programmers, mediated by the software artefacts involved.

We refer to software interfaces, or APIs (application programming interfaces), as any set of semantically related operations and data, usually associated with a specific domain. Software components, modules, libraries and frameworks usually provide APIs that expose their functionality to other software elements. This definition is similar to the one used in the work by de Souza et al. regarding the study of APIs in the context of cooperative work [13]. APIs play a central role in modern development environments and languages, since even the most simple programs depend on the provided library and framework interfaces [23].

Software projects are known to be a difficult endeavour, and defects are usually expected. Although it is not easy to estimate the percentage of defects related to the incorrect use of APIs or to misinterpretation of its design, probably every seasoned programmer has already experienced difficulties and errors when writing code involving a complex API.

The use of software interfaces may impose a considerable amount of cognitive load on the programmer, depending on the abstractions involved and the design of the artefacts provided. The higher this load is, the higher is the intellectual effort needed, which may increase the error-proneness of the activity of software development.

To illustrate how API design and communication of intent may have an impact on the work of developers, we present a simple example from the Java API, based on the book from Bloch and Gafter [5]. The example shows that even well known concepts like date and time, used in almost all application domains, can be a source of problems. The Java class *Calendar* implements some of the functionalities regarding date and time in the language API, and it contains many *set* methods that allow the programmer to change an object's internal fields. One variant of these *set* methods is defined as below:

```
void set(int year, int month, int date)
```

At first, the method's short description in the documentation looks pretty obvious: *"Sets the values for the calendar fields YEAR, MONTH, and DAY_OF_MONTH"*. Although it may seem straightforward, a very simple program reveals what is behind the implementation:

```
Calendar c = Calendar.getInstance();
c.set(2012, 8, 31);
System.out.println( c.getTime() );
```

Surprisingly, the output of a Java program containing this code snippet is similar to the following:

```
Mon Oct 01 13:21:27 BRT 2012
```

Most people would expect something like "Aug 31". The explanation comes from the fact that the *Calendar* class handles months in a zero-based integer representation (e.g. 0=Jan). So, the parameters in the example mean "Sep 31", which is itself an invalid date. In this case, the *Calendar* class silently "corrects" the date overflow to the next day, which is "Oct 1".

Despite the fact that this simple example may be considered as a bad design decision, it illustrates how the designer intentions behind an API may differ from the first user interpretation of its meaning. And if this may happen with a well known concept like date and time, more complex domains can be a real challenge to the designer in order to represent the concepts, meanings and behaviour behind an API, and also to the programmer, who has to interpret the designer's message and use the software artefact as originally intended.

This paper describes a work in progress in which we aim to analyse software interface specifications from a human-centric perspective, trying to identify how its effectiveness can be evaluated and what can be done to enhance the communication of a software artefact design to programmers. We intend to use a combined semiotic and cognitive inspection method in this investigation.

The remainder of this paper is organised as follows: section 2 discusses some aspects related to the view of API design from a communication perspective that motivate this work. Section 3 analyses methods for evaluating APIs from a semiotic and cognitive perspective. Section 4 arguments about the use of some techniques that may influence the communication of design intent in the context of programming interfaces. Section 5 presents a sample scenario for the application of the referred methods and possible findings for the type of experiment proposed. Finally, we describe related work in section 6, and conclude with our final remarks and future work in section 7.

2 APIs: communicating design

In this section, we look at APIs as a design artefact and analyse the communication aspects involved in their representation of the designer's intent to programmers.

Programming is a hard mental work, and a developer usually has to deal with a great amount of information to write functional code. Problem domain, requirements, specifications, algorithms, programming language, tools and frameworks are among the kind of knowledge that is demanded from a programmer to perform his tasks. APIs are used most of the time when programming in modern languages, especially when dealing with distributed systems and enterprise frameworks.

API design is critical because it is intended to be written once and used many times, and later changes may impact users due to compatibility issues. Complex APIs may be daunting, and difficult usage may discourage adoption, as it increases the demand for experienced programmers that may be able to use it effectively and efficiently. An example of the consequences of a overly complex API can be found in Henning's work [19]. Another article by the same author discusses the importance of API design [18].

An interesting aspect related to the design of APIs is that they can have many different and specific goals, which generally make them unique. There can be subtleties behind it that make some decisions critical to its completeness, usability and flexibility. The design process should anticipate some crucial aspects about API usage, and this may influence the underlying system design. For example, the work by Ierusalimschy et al. [22] provides interesting insights about how the design of the API for embedding Lua scripts in C programs influenced the design of the Lua language, and vice-versa.

When developing software, a programmer should have a good mental model of the software artefacts being reused in order to correctly apply these models to his own design, and call the available operations and services accordingly. If there is not a good understanding about the abstractions being provided, this can lead to subtle errors that may appear later in the software life cycle. In a research work dedicated to identifying common software defect causes and characteristics [20], the authors report that "API misuse is the single most prevalent cause" of bug patterns detected. Although this work is specific to the Java language, it is a good illustration of how poor API design may have a strong impact on software quality.

Designing an API is about describing abstractions through type and interface specifications, and it is usually work assigned to development team members without specific expertise or training in this design task. Although this may provide good results depending on the team's talent, more attention should be paid to the role of an API designer, given the impact that this activity may have on the overall project outcome. At least, the most experienced members of a team should be involved, as they have probably seen more of badly designed APIs and may know better what should be avoided [18].

If we consider that an API represents abstractions created by its designers that need to be understood by programmers in order to be used effectively, this may be viewed as a communication process taking place between these two parties, mediated by the software artefacts involved (specifications, documentation, code, binaries, messages, etc.). More precisely, these artefacts communicate to the programmers how they should interact with the API, which is itself another piece of communication. So, in this sense, API design may be regarded as a metacommunication process taking place between designers and programmers.

In their work [27], Robillard and Deline describe qualitative findings regarding API learning obstacles and conclude that documentation of intent is one of the most important factors that impacted learner's experience. This stresses the importance of effectively communicating to programmers the API designer's intentions when developing its abstractions and concepts, and thus supports our argumentation. Also, they state that "the responsibility from documenting an API cannot be cleanly separated from the responsibility for designing the API, even though different skills are involved". This reinforces the need for a specific role of API designer in the development team.

The idea of studying API design as a communication process between designer and programmer builds on the discipline of Semiotic Engineering [11], which provides a semiotic theory for

HCI. Semiotic Engineering has been successfully developed and applied in the last two decades to study interactive computer systems as "one-shot messages sent from designers to users", taking into account the meaning-related and signification processes that occur in the design and construction of software artefacts.

Although the concepts and methods in this discipline have been traditionally applied to the analysis of interactive systems in the context of graphical user interfaces, we aim to develop them further to evaluate the interaction of programmers with APIs, adapting to the differences of this type of system (a programming interface).

When dealing with interactive computer systems, the designer may convey his message to the user by means of choosing proper graphical elements in the interface, screen layout, dynamic system behaviour, and other representations, most of them visual. User interfaces have been the focus of much attention in the last decade with the availability of new technologies that allow different forms of interaction, especially touch and voice-based. This gives the designer a rich toolset to communicate his intent to the user during interaction time.

As to programming languages and APIs, the tools available to the designer usually do not offer as many possibilities to represent his intentions at interaction time, when compared to graphical user interfaces for computer systems. The most common resources for communicating API design are the syntactical representation of the interfaces, and the textual documentation explaining its concepts and behaviour. The syntactical structure of an API comprehends the names of the interfaces, operations and data structures, as well as their types. Good name choices are very important at this level, because they are almost the only representation of intent available to the designer. They should also be consistent throughout the various elements.

To overcome the limitations of syntactical specification, textual documentation is the most common medium to complement the representation of the designer's intentions. Natural language documentation is commonly used, but sometimes it may be ambiguous or incomplete. Documentation may also be augmented with formal descriptions of the operations using a rigorous mathematical notation, in an attempt to reduce or eliminate these ambiguities. On the other hand, this type of notation can be more difficult to be understood by programmers with less formal background.

In order to give a more dynamic approach to textual documentation, some integrated development environments (IDEs) try to bring them closer to the code writing activity, suggesting operations, parameters and showing their description to the programmer. Although this can be of great help when writing code, it does not eliminate the need for a more complete description of the concepts and abstractions behind the API design, which are fundamental to its adequate use.

Code examples are another important technique to represent the intent behind the design of an API, as they present use cases from the perspective of the designer's interpretation of the abstractions involved, which can be different from the user's, at least while there is not a clear comprehension about the message being conveyed. Good examples can eliminate possible gaps between syntactical and textual description of the API, and its actual use, as they usually make the concepts and the intent behind the design more concrete. The downside of code examples is that they can be a source of "copy/paste programming" without a real understanding of the concepts and the dynamics of the API, but this type of practice is up to the programmer to be avoided.

Differently from graphical software, a programmer's interaction with an API occurs when a program is written, compiled, executed and debugged. The programmer reads the documentation, writes code to perform a certain task, evaluates the return codes, handles exceptions, runs the program, analyses the outcome, reads messages, output traces and logs, and so on. This type of interaction inherently limits the designer options to be "present" at interaction time when compared to more visual and dynamic system interfaces. One of the research questions that arises when investigating this subject is how the use of different tools and techniques, other than syntactical specification and natural language documentation, can help the designer perform a more effective communication to the programmers of how interaction with the API is expected to be done.

As already mentioned, programming with different APIs demands a high cognitive load from the user perspective. So, the expected result of making the communication of API design more effective is to lower the hurdle for the programmer to accomplish his task. If we intend to analyse the representation of API design from the Semiotic Engineering perspective, it seems natural to combine this analysis with a view of the cognitive impact on the programmers tasks.

In this context, the Cognitive Dimensions of Notations framework (CDN) [2] has been successfully employed in previous works related to the evaluation of programming-related tasks [8, 25], and it can provide an interesting counterpart to the semiotic view of API design. In the next section, we describe this combination of semiotic and cognitive inspection methods in the context of API design evaluation.

3 Evaluation of APIs

This section refers to existing methods for the evaluation of APIs and proposes the use of a combined semiotic-cognitive inspection method that can be used in a technical context, to evaluate a particular API, and in a scientific context, to generate new knowledge in HCI and Software Engineering.

The discipline of API design and implementation has been extensively studied in the past decade, as it is a main concern for software development companies that publish APIs to a large client base. Some of the most representative work in this field come from large software companies like Microsoft or Google [8, 3]. Their concern originates from the fact that getting it right before publishing is mandatory, because post-release fixes are costly and may break legacy code.

Many recent studies in API design have been developed under a usability or learnability context ([27, 28, 9, 7]), putting stronger emphasis on the user side. Although they are of great value, it is also important to study this communication process from the designer perspective, analysing how tools an techniques can be used and improved to make it more effective. We believe that the combination of semiotic and cognitive methods can be a powerful resource to help us understand the human-related aspects of API design and software reuse.

Building on related work concerning the validity of new knowledge generated by inspection methods [12], we intend to adapt the Semiotic Inspection Method (SIM) to the evaluation of programming interfaces, analysing the communicability of these software artefacts. We expect that this type of qualitative research may produce interesting insights regarding the effect of using different approaches to API design, which can serve as input to a more specific quantitative research in the same subject.

In Semiotic Engineering, signs play a central role in the investigation of the message being conveyed from the designer to the user. Their analysis relies on a classification scheme according to the interactive conditions that express their representation. They are divided in three classes: static, dynamic and metalinguistic.

From the definitions in [12], static signs are "those whose representation is motionless and persistent when no interaction is taking place". Dynamic signs are the ones "whose representation is in motion regardless of users' actions or whose representation unfolds and transforms itself in response to an interactive turn". And, finally, metalinguistic signs "represent other static, dynamic, or metalinguistic signs".

The designer's message conveyed to the users is strongly influenced by the choices made when combining those different types of signs. This message can be described by instantiating the metacommunication template [10] :

"Here is my understanding of who you are, what I've learned you want or need to do, in which preferred ways, and why. This is the system that I have therefore designed for you, and

this is the way you can or should use it in order to fulfil a range of purposes that fall within this vision'.'

The signs that compose the software artefacts under analysis should be able to represent the implicit message described by the instantiation of the metacommunication template, and SIM's goal is to evaluate the communicability of these artefacts from the perspective of the designer. (i.e. the communication sender). The full description of the SIM method is beyond the scope of this paper, but it can be summarised as a sequence of five steps, where the first three *deconstruct* the designer's message by performing a segmented analysis of the different classes of signs, and the last two *reconstruct* it by integrating and interpreting the deconstructed signs. The result is a characterisation of the designer's message structure in terms of signs and meanings.

Also in the semiotic context, the work by Tanaka-Ishii [29] provides an interesting account of the use of signs in programming, stressing the role of identifiers in programs and providing a semantic classification for them in three levels:

– Hardware: an identifier represents a memory address that stores a pattern of bits
– Programming language: an identifier represents the definition or use of a variable, routine, module, etc.
– Natural language: an identifier represents a "message" from the programer who writes the code to another programmer who reads it

In the context of evaluating APIs as a communication artefact, the natural language level can be viewed as the most relevant, as it represents the designer intents when creating a software artefact. At this level, the appropriate choice of metaphors is an important resource for the effectiveness of an API design.

The inspiration for a combined use of qualitative research methods to evaluate APIs comes from a recent work regarding the investigation of visual programming environments and computational thinking acquisition [16], which proposes the use of discourse analysis and inspections based on Semiotic Engineering methods and the CDN framework. Although the nature of the object of analysis is different (visual programming environments vs. APIs), we intend to adapt the method used in order to obtain qualitative findings which are expected to be relevant to the field of API design.

The basic idea behind the combination of these research methods is to provide different perspectives for the same experiment and obtain a more complete cycle of analysis of the findings. SIM offers insights about sign systems and notations used by the designer to communicate his vision. CDN provides the basis for a cognitive analysis of the experiments, which is relevant due to the fact that the use of a new API implies a learning experience for programmers. These two methods may be complemented by discourse analysis by providing additional evidence to support or contradict other findings.

To the best of our knowledge, there is no report of a similar experiment using the combination of these methods. This means that there is no previous evidence that this approach is effective. On the other hand, there is a high expectancy of obtaining relevant results based on the application of these methods to the evaluation of visual programming environments.

4 Enhancing the communication of API design

In the previous sections, we presented how the design of an API may be regarded as a communication process, an how it can be evaluated. In this section, we discuss some concepts and techniques regarding API specification that may influence this communication by offering more expressiveness to the designer in order to convey his vision of the system to the programmers.

In our current work, the main expected contribution is the evaluation of API design from a communication perspective and the cognitive impact of the designer's choices by using a novel combination of research methods. This may serve as the basis for the comparison of different API specification techniques and tools, as a secondary contribution from this work.

The most common form of API specification is the combination of its syntactic elements written in a formal language and a textual description in natural language of the semantics of its operations and parameters, as well as its behaviour. The designer has a few options to represent his vision of the software artefact to the users, namely:

– names for operations and parameters
– types for parameters and return values
– textual description of semantics and behaviour
– exceptions and error codes to indicate misuse of API or unexpected conditions

The sole use of syntactical constructs in interface specifications limits the designer's options to be "present" at interaction time to provide more dynamic information to the programmer. In a programming environment that offers mechanisms for behavioural specification and runtime monitoring, the designer is able to include a more formal description of its intents that may give a richer interactive experience to the programmer when dealing with an API.

For instance, contracts [26] are a lightweight formal specification technique based on the use of preconditions, postconditions and invariants to describe the behaviour of software artefacts. This form of specification provides more powerful tools to the designer in order to describe how the interface is expected to be used, and which are the results of the operations depending on the calling context. This type of specification may interfere on the programmer's activity by giving feedback while code is written (static analysis), as well as when the program is executed (dynamic analysis), since the violation of any assertion may give more information about the cause and the nature of the problem.

From a cognitive perspective, the use of contracts may also have an impact on the programmer's work, since they provide a more precise description of the API behaviour than textual documentation, helping the programmer to understand the causes of possible errors by giving immediate feedback related to API misuses. Putting in CDN terms, contracts may have, for instance, a higher closeness of mapping to the API behaviour than textual documentation, and also make hidden dependencies between operations in the interfaces more explicit.

The use of formal behavioural specification languages [17] provide an even higher expressiveness to describe a software artefact, and allows the use of tools like model checkers to validate the specification. Although they can be a very powerful specification resource, they may also impose a higher demand for abstractions on the user, specially in mathematical terms, which can possibly have a negative impact on learnability. These are also interesting aspects to be investigated.

Beyond behavioural specifications, there are further levels that can be approached to describe a software artefact. In [1], the authors provide a classification of contracts in four levels: syntactic, behavioural, synchronisation, and quality of service. Most contract systems support the second level, described in terms of invariants, pre and post conditions, which consist of logical assertions to describe the system state before and after operation calls, supporting modular reasoning and runtime verification of these conditions.

The specification of synchronisation contracts can be a valuable resource in expressing the designer's intents, as they offer a formal definition of the allowed sequence of operations, which can be enforced at runtime to prevent an unexpected call pattern, informing the user the precise reason to why the API does not work as expected in that particular scenario.

Quality of service contracts open the possibility of specifying non-functional aspects of a software artefact that are more related to the execution environment or the preciseness of the results of the computation being carried out. Although they usually do not represent a correctness constraint, they offer the designer the opportunity to specify the limitations or requirements of an API in terms of its execution environment.

Runtime monitoring of these API constraints may allow a richer interactive experience to the programmer, as there is a constant verification of the designer's assumptions and intentions, with a corresponding “alert” in case of violation of these conditions. This signal may come in the form of a runtime exception, log or trace message, routine call, and so on. These mechanisms may pinpoint the source of API misuses and provide the programmer with a “higher level” message indicating what is wrong, which may also have a positive impact from the cognitive perspective.

We intend to investigate the aspects described in this section using the combination of methods proposed in section 3. We are currently selecting the appropriate tools and elaborating the experimental scenario in order to capture the most representative aspects in the context of API design and use, and we will build on our previous experience regarding user experiments, as well as on the reports from similar research, in an effort to make the most of the results.

5 Example of application scenario

In this section, we illustrate the concepts presented throughout the previous sections with an example scenario of application of the inspection methods, and the issues that may arise when investigating the design of a particular API from a semiotic and cognitive perspective. Once again, the example comes from the Java language, as it is a popular language that may be familiar to many readers.

The designers of the Java language and the core API created a single rooted class hierarchy based on the *Object* class. This class defines common operations to all Java objects. In particular, the method *Object.equals()* provides logical equality comparison, and its default implementation is as restrictive as possible, since it compares the object address in memory, which makes any object different from all objects but itself.

Another method in the *Object* class is *hashCode()*, which calculates a hash value for an object. The API designers created it in anticipation of the need of a standard way of allowing any object to be part of a hash-based collection such as *HashMap*. The result of the *hashCode* method determines the distribution of the objects in this kind of collection, which has a direct impact on the performance of the searching algorithm.

Although the *Object* class provides a default implementation for these methods, it is often necessary to override them. If the programmer wishes to provide logical equality for different instances of the same class, the *equals()* method should be overridden. For example, a programmer might consider two objects of a *Car* class equivalent if they have the same registration number. In this case, it would be necessary to provide a specific implementation for the *equals()* method. The method’s documentation specifies that any implementation should satisfy the requirements of an equivalence relation, which means that it should be reflexive, symmetric and transitive. Also, it should be consistent, returning always the same value (true or false) when called, provided the internal state of the objects being compared do not change between calls.

There is a strong relationship between these two methods that is commonly overlooked or ignored by Java beginners. The “rule of thumb” says that, if one overrides one method, the other should also be overridden, in order to maintain the class consistence, since objects that are considered “equal” must return the same hash code. One typical consequence of not getting this right is that an object inserted into a hash-based container may never be recovered.

A quick search for terms like “java hashcode equals” in Google provides many results regarding tutorials, articles and discussions in development forums about this topic. The “Effective Java” book [4] dedicates almost 20 pages to the discussion of these methods, which shows that it is not a trivial subject. Due to the inherent complexity of this feature, the misinterpretation of the designer’s intent behind the API specification may cause subtle defects that can be difficult to trace [21].

One particular issue related to this characteristic of the Java language can be found in the Java API itself. The class *java.net.URL* overrides the *equals()* method, based on the assumption that two URLs are equivalent if the name of their host components resolve to the same IP address.

The use of domain name resolution inside an equality comparison of URL objects is, by itself, a bad idea. It makes the *URL.equals()* operation dependent on the network status, which means that it may fail or take a long time. Also, virtual hosting allows two different websites to share the same IP address, which breaks the assumption that two URLs pointing to the same host may be considered equal.

Apart from being a bad design example, the URL class does not comply with the *Object.equals()* contract, since it breaks the consistency requirement. As it depends on factors that are external to the object state (i.e. network), the method may return different results between calls. Also, the timing issues related to name resolution makes the use of URL objects in hash-based collections impractical. For example, if a *HashMap* contains URL objects as keys, a *get* operation will compare the requested key with the collection elements, calling *URL.equals()* repeatedly, and each call will perform a host name resolution in the network.

The URL class example illustrates that the communication of API design intent may involve three different roles: the designer of the API, the implementor of the API, and the client programmer. In this case, the developer of the *URL* class misinterpreted the design of the *Object* class, and provided a broken implementation of the original intended artefact. The application programmer, at the end of the chain, has to deal with the burden of interpreting different “messages”, one coming from the original *Object* class designer, and the other from the *URL* class developer.

In the context of this work, some of the research questions that arise regarding this example are: how does the Java API “communicate” these design decisions to programmers ? Could it be more effective ? What are the cognitive aspects involved, and what tools or resources could be used to help the programmers get things right, in the first place, especially for beginners ?

From a Semiotic Engineering perspective, the main signs used by the API designers in order to send their message to the users are the method signatures with names and parameters (static signs), the return values for these methods and other related operations like inserting and removing from collections (dynamic signs), and the textual description in the Java API documentation (metalinguistic signs). A detailed inspection might reveal if the signs are appropriate, and what changes or additions to the API could make this communication more effective. For instance, there could be better code examples in the documentation, methods to test a class implementation for consistence regarding these methods, formal specifications (e.g. contracts) that could be enforced statically or dynamically, and so on.

From a cognitive perspective, a CDN based inspection may provide interesting insights regarding this particular design. For instance, the issue described may be considered a hidden dependency between classes in the API, as it is not obvious at first, especially to a novice Java programmer, that inserting the class into a container may not work if the methods are not overridden. Also, any change in the internal structure of an object may impact its *hashCode()* or *equals()* implementation, which can be another example of hidden dependency, or even viscosity. Premature commitments may also arise when a programmer creates a new class, as it is necessary to anticipate if it will be used in a hash-based container or need a logical equality comparison.

The scenario described in this section can serve as the basis for a user experiment concerning programming tasks carefully selected to provide qualitative findings regarding a particular API design, combining the semiotic and cognitive approach, complemented with discourse analysis. It can also be used in the evaluation of the effectiveness of advanced specification techniques and tools for the communication of design intent of APIs.

6 Related work

This section presents a brief description of related work concerning the evaluation of API design and programming activities from a semiotic or cognitive perspective that inspire or influence our current research.

Clarke and Becker's work [8] is one of the first cognitive approaches to API design evaluation based on the CDN framework. They created a modified version of CDN in order to assess the usability of an object-oriented library, and conducted empirical studies based on a set of development tasks which had to be implemented by the participants during videotaped sessions. The results were analysed to extract patterns of behaviour from the participants that might help to identify problems in the library's design.

Maia et al. [25] present a qualitative method to evaluate the flexibility of middleware implementations based on a CDN inspection of representative adaptation tasks to be performed on the middleware platforms under analysis. Although this work is not specific to API design, it presents a good example of CDN instantiation to evaluate cognitive aspects of programming tasks, which is closely related to our objectives.

The work by Farooq et al. [15] describes API usability peer reviews, an inspection method conducted as a group-based walkthrough of the source code. They contrast this method to usability tests, arguing that both methods can be used in conjunction and complement each other, because peer reviews have a lower cost and shorter execution time, but identifies less usability defects than usability tests.

The work by Dubochet [14] evaluates programming languages as a medium for human communication, based on an experiment using an eye-tracking device and distributed cognition. Although the goals of this work differ from ours, it provides interesting insights about the programmer's cognitive experience when dealing with two different programming languages.

Cataldo et al. [6] performed a quantitative study of the impact of interface complexity on the error-proneness of the source code of two large systems. In their findings, they concluded that the increase in interface complexity leads to more bugs in the source code files that use these interfaces. This may be a good reference for future quantitative research based on our qualitative results, and reinforces the importance of evaluating the effectiveness of API design communication, specially when dealing with complex ones.

7 Final remarks and future work

In this paper, we discussed the importance of API design in the context of software reuse, and presented the motivation for the evaluation of software artefacts from a communication and human-centric perspective. Also, we proposed the use of a combined semiotic and cognitive method to perform this kind of evaluation, describing a typical scenario of application, as well as possible contributions.

We are currently refining the application of the methods to the API evaluation context, based on the previous experiments concerning visual languages, and selecting scenarios for a user study to maximise the relevance of qualitative findings. One possible scenario would be the evaluation of the Java API features described in section 5 by performing an experiment with undergraduate students in Computer Science. After performing the user experiments based on the described methods, we intend to analyse the results and report the most relevant findings in a future work.

We also intend to apply the combined semiotic and cognitive inspection methods to evaluate APIs which have been previously analysed in related works. This can be an interesting opportunity to experiment the methods in a context of API evaluation and compare the findings with the previous results, and also to improve the methodology itself.

In the long term, we expect to achieve the more general objective of providing a practical and effective approach to API design evaluation, in order to support software projects in which APIs are considered a critical asset.

Acknowledgements

Luiz Marques Afonso is partially supported by CAPES (PGCI-CAPES-013/10).

References

1. Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–45, July 1999.
2. Alan Blackwell and Thomas R. Green. Notational systems – the Cognitive Dimensions of Notations framework. In John M. Carroll, editor, *HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science*, Interactive Technologies, chapter 5, pages 103+. Morgan Kaufmann, San Francisco, CA, USA, 2003.
3. Joshua Bloch. How to design a good API and why it matters. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 506–507, New York, NY, USA, 2006. ACM.
4. Joshua Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008.
5. Joshua Bloch and Neal Gafter. *Java Puzzlers: Traps, Pitfalls, and Corner Cases*. Addison-Wesley Professional, 2005.
6. M. Cataldo, C. R. B. de Souza, D. L. Bentolila, T. C. Miranda, and S. Nambiar. The impact of interface complexity on failures: An empirical analysis and implications for tool design. *Technical Report CMU-ISR-10-101, School of Computer Science, Carnegie Mellon University*, 2010.
7. Steven Clarke. Measuring API usability. *Dr. Dobb's Journal*, 29:S6–S9, 2004.
8. Steven Clarke and Curtis Becker. Using the Cognitive Dimensions Framework to evaluate the usability of a class library. In M. Petre and B. Budgen, editors, *Proc. Joint Conf. EASE & PPIG*, pages 359–366, April 2003.
9. Aniket Dahotre, Vasanth Krishnamoorthy, Matt Corley, and Christopher Scaffidi. Using intelligent tutors to enhance student learning of application programming interfaces. *J. Comput. Sci. Coll.*, 27(1):195–201, October 2011.
10. Clarisse S. De Souza. *The Semiotic Engineering of Human-Computer Interaction*. The MIT Press, 2005.
11. Clarisse Sieckenius de Souza. *Semiotics: and Human-Computer Interaction*. The Interaction-Design.org Foundation, Aarhus, Denmark, 2012.
12. Clarisse Sieckenius de Souza, Carla Faria Leitão, Raquel Oliveira Prates, Sílvia Amélia Bim, and Elton José da Silva. Can inspection methods generate valid new knowledge in HCI? the case of semiotic inspection. *Int. J. Hum.-Comput. Stud.*, 68(1-2):22–40, January 2010.
13. Cleidson R. B. de Souza, David Redmiles, Li-Te Cheng, David Millen, and John Patterson. Sometimes you need to see through walls: a field study of application programming interfaces. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, CSCW '04, pages 63–71, New York, NY, USA, 2004. ACM.
14. Gilles Dubochet. Computer code as a medium for human communication : Are programming languages improving ? *Psychology of Programming Workshop (PPIG 2009)*, pages 174–187, 2009.
15. Umer Farooq and Dieter Zirkler. API peer reviews: a method for evaluating usability of application programming interfaces. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, CSCW '10, pages 207–210, New York, NY, USA, 2010. ACM.
16. J.J. Ferreira, C.S. de Souza, L.C.C. Salgado, C. Slaviero, C.F Leitão, and F.F. Moreira. Combining cognitive, semiotic and discourse analysis to explore the power of notations in visual programming. *To appear in the Proceedings of VL-HCC'2012 - IEEE Symposium on Visual Languages and Human-Centric Computing.*, 2012.
17. John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16:1–16:58, June 2012.
18. Michi Henning. API design matters. *Queue*, 5(4):24–36, May 2007.
19. Michi Henning. The rise and fall of CORBA. *Commun. ACM*, 51(8):52–57, August 2008.
20. David H. Hovemeyer. *Simple and effective static analysis to find bugs*. PhD thesis, College Park, MD, USA, 2005. AAI3184274.
21. J. Howell. What's the deal with Java equals() and hashcode() ? http://www.summa-tech.com/blog/2010/01/26/what's-the-deal-with-java-equals-and-hashcode/.
22. Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. Passing a language through the eye of a needle. *Queue*, 9(5):20:20–20:29, May 2011.
23. John M. Daughtry Iii and John M. Carroll. Perceived self-efficacy and APIs. *Psychology of Programming Workshop (PPIG 2010)*, 2010.
24. Jeff Kramer. Is abstraction the key to computing? *Commun. ACM*, 50(4):36–42, April 2007.
25. Renato Maia, Renato Cerqueira, Clarisse de Souza, and Tomás Guisasola-Gorham. A qualitative human-centric evaluation of flexibility in middleware implementations. *Empirical Software Engineering*, 17:166–199, 2012. 10.1007/s10664-011-9167-7.
26. Bertrand Meyer. Applying design by contract. *Computer*, 25(10):40–51, 1992.
27. Martin P. Robillard and Robert Deline. A field study of API learning obstacles. *Empirical Softw. Engg.*, 16(6):703–732, December 2011.
28. Jeffrey Stylos and Steven Clarke. Usability implications of requiring parameters in objects' constructors. In *ICSE*, pages 529–539. IEEE Computer Society, 2007.

29. Kumiko Tanaka-Ishii. *Semiotics of Programming.* Cambridge University Press, New York, NY, USA, 1st edition, 2010.
30. Jeannette M. Wing. Computational thinking and thinking about computing. *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, 366(1881):3717–3725, October 2008.