The Journal of Systems and Software 126 (2017) 17–33

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Design annotations to improve API discoverability

André L. Santos[a,*], Brad A. Myers[b]

[a] Instituto Universitário de Lisboa (ISCTE–IUL), ISTAR–IUL, Portugal
[b] Carnegie Mellon University, Human-Computer Interaction Institute, USA

ARTICLE INFO

Article history:
Received 25 January 2016
Revised 17 October 2016
Accepted 29 December 2016
Available online 4 January 2017

Keywords:
API usability
Annotations
Code completion
IDE
Eclipse

ABSTRACT

User studies have revealed that programmers face several obstacles when learning application programming interfaces (APIs). A considerable part of such difficulties relate to discovery of API elements and the relationships among them. To address discoverability problems, we show how to complement APIs with design annotations, which document design decisions in a program-processable form for types, methods, and parameters. The information provided by the annotations is consumed by the integrated development environment (IDE) in order to assist API users with useful code completion proposals regarding object creation and manipulation, which facilitate API exploration and learning. As a proof of concept, we developed *Dacite*, a tool which comprises a set of Java annotations and an accompanying plugin for the Eclipse IDE. A user study revealed that Dacite is usable and effective, and Dacite's proposals enable programmers to be more successful in solving programming tasks involving unfamiliar APIs.

1. Introduction

Writing programs and using application programming interfaces (APIs) are inseparable activities in modern software engineering; it being nearly infeasible to write a program without resorting to third-party APIs, for instance addressing data types, GUIs, or networking (Myers and Stylos, 2016). In this work, we focus on object-oriented APIs provided by libraries or frameworks. APIs are developed by *API designers*, who represent a small proportion of the programmers' universe and typically have strong technical skills, whereas *API users* program against the APIs and represent nearly everyone who writes code, many of whom have a different background than computer science (Scaffidi et al., 2005).

Software practitioners and researchers have raised awareness as to the importance of good API design (Bloch, 2006, 2008; Cwalina and Abrams, 2008; Tulach, 2012; Myers and Stylos, 2016), and metrics have been proposed to automatically measure API usability (e.g., Scheller and Kühn, 2015; Rama and Kak, 2015). As the complexity of software increases, both due to intrinsic domain complexity and non-functional requirements (e.g., adaptability and reusability), APIs can become more complex, and consequently, more difficult to use. Such requirements lead API designers to adopt certain design decisions which may hinder API usability from an API user's viewpoint. For instance, object factories (Gamma et al., 1995) have a significant impact on API usability (Ellis et al., 2007), as do decisions regarding method placement in different classes (Stylos and Myers, 2008). Design patterns (Gamma et al., 1995) play a significant role in API design, since many design decisions relate to a pattern. Although the solutions offered by design patterns are needed, they often introduce design complexity.

Although design patterns are pervasive in software engineering, it may not be apparent to API users when they are being used. For instance, certain objects of an API might need to be created through a *static factory* (Bloch, 2008) instead of using a constructor, because underlying the design could be an instance of the *singleton* or *flyweight* patterns (Gamma et al., 1995), of which the API user might be totally unaware. Previous research showed that even if programmers were aware of certain patterns, they did not necessarily expect them when trying to learn an API (Ellis et al., 2007). Moreover, it has been shown that in contrast to simpler solutions, the presence of design patterns may hinder understandability (Prechelt et al., 2001). We argue that a design pattern whose participant classes are exposed in the API potentially introduces an additional difficulty to API users. For example, given that in certain situations users will not be able create API objects through the most "natural" way (that is, using constructors), they must discover factory methods which are potentially located in different classes that will enable them to create the desired objects. As another example, essential functionality related to a certain object might be available in methods contained in helper classes, which also might not be easy to find. User studies revealed that these *discovery* barriers have a considerable impact on API learning (Ellis et al., 2007; Stylos and Myers, 2008; Duala-Ekoko and Robillard, 2012).

* Corresponding author.
E-mail address: andre.santos@iscte.pt (A.L. Santos).

http://dx.doi.org/10.1016/j.jss.2016.12.036
0164-1212/

In this paper, we provide techniques to improve API discoverability by means of complementing APIs with **design annotations**,[1] which explicitly and formally contain information regarding design decisions. The main purpose and advantage of having these annotations is to enable integrated development environments (IDEs) to assist API users with useful code completion proposals based on the information embodied in the annotations. A further advantage is that empirical experiments have shown that documenting design patterns in the implementation is beneficial for system maintenance (Prechelt et al., 2002). Therefore, as a side effect, our design annotations also play the role of documentation artifacts in the source code, in addition to conventional documentation. Even though it has been demonstrated that it is sometimes possible to automatically detect design patterns in source code (Tsantalis et al., 2006), this kind of approach has not been targeted at API usage assistance but rather at program comprehension in general.

We refer to the whole set of artifacts that we developed as *Dacite*.[2] We designed a small set of Java annotations and an annotation processor that can be used by API designers to apply our approach in their APIs. The annotations are fairly simple for the API designer to enter, given that in most cases they are merely formalizing (i.e. in a program-processable form) information that is otherwise typically present in informal documentation (i.e. free-form text). The annotations have the advantage of being validated by the processor against well-formedness rules. We developed a plugin for the Eclipse integrated development environment (IDE) that enables API users to gain leverage from design annotations, in a form that integrates seamlessly with existing code completion mechanisms. We address API designs that may cause a wide variety of discoverability issues, involving common patterns and idioms (Gamma et al., 1995; Bloch, 2008), namely *static factories, factory methods, object builders, helper methods, decorators*, and *composite objects*.

Dacite provides a unified mechanism to address API discoverability based on enriching the API with code annotations. We integrate in a systematic way support for assisting with overcoming difficulties that have previously been described (e.g., factory methods (Mooty et al., 2010)), as well as new mechanisms targeting difficulties that have not been addressed before, namely object composition based on decorator and composite patterns. Although previous tools have augmented code completion proposals of IDEs in similar ways to ours, in those approaches the intent of API designers is not captured by the API implementation, a characteristic that is unique to our approach. In this way, API developers gain control over the code completion proposals pertaining to API discoverability recommended by IDEs, since solutions based on automated detection and mining of API design aspects are always subject to precision/recall issues to some extent.

Fig. 1 presents two usage scenarios of our approach, illustrated with Java's Collections API. The class `java.util.Collections` contains several static helper methods to manipulate collections, such as for sorting and to create immutable collection views. The upper part of the figure contains a snippet of two of those methods with design annotations. In the first case, the method `emptyList()` is annotated with `@StaticFactory`, denoting that it consists of a static factory to create a `List` object (the return type). In the second case, the method `sort` has a parameter annotated with `@Helper`, indicating that `sort` is a helper method that can be used on a `List` object (the parameter type). The figure also presents two screenshots of the code completion

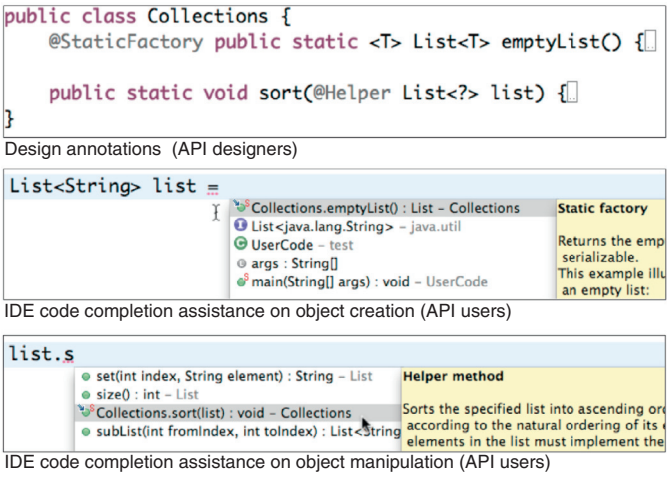

Fig. 1. With Dacite, API designers complement APIs with design annotations, from which code completion proposals related to API discoverability are provided to API users.

menus of Eclipse, enhanced with proposals that are automatically derived from the design annotations.

We envision that our design annotations would ideally be applied by the API's developers, who are in control of its implementation and should be the most well-informed people for this purpose, as they are also typically in charge of writing the API documentation. Design annotations can be applied to existing APIs without breaking client code. However, it may also be desirable to annotate an API one does not own (e.g., for an open-source component), and therefore, we also provide a way to annotate an API externally by a third-party. In the context of large-scale enterprise software development, in-house components are reused through their API across a number of other components developed by different teams, and here API discoverability issues are also relevant. In these cases, given that all the components are in control of the same organization, it would be relatively easy to enforce that the API code has to be annotated, as with other coding rules and conventions.

We conducted a user study to evaluate the effectiveness of the code completion proposals of Dacite from an API user's perspective. The study was based on programming tasks using unfamiliar APIs under time constraints. The results showed that programmers using Dacite were up to twice as successful in accomplishing the given tasks within the given time, through the use of our proposals. This provides evidence that Dacite's code completion mechanisms are usable and effectively help programmers to discover needed information about the APIs.

The main contributions of this research are: (1) a unified approach based on design annotations to address both previously identified and other API design patterns that can hinder API discoverability, (2) identifying *additional* design patterns (such as the decorator and composite patterns) that can benefit from this kind of help, (3) enhancements to an IDE that can take advantage of the design annotations to assist API users, and (4) empirical evidence that the provided IDE enhancements are usable and enable programmers to be more successful in solving programming tasks involving unfamiliar APIs.

2. API discoverability

An exploratory study of API usability (Duala-Ekoko and Robillard, 2012) identified types of questions programmers ask when working with unfamiliar APIs. The study collected data from think-aloud protocols, screen captures and interviews, involving 20 participants working on two programming tasks on different

[1] *Annotations* is Java's terminology; they are called *attributes* in C#.

[2] Dacite is a kind of rock, and here stands for: Design Annotations for Complementing Interfaces Targeting Effectiveness. A prototype implementation is available at github.com/andre-santos-pt/dacite.

APIs. Some of the observed questions were characterized as being more difficult to answer than others. The level of difficulty was measured based on the sequences of actions taken by the study participants that reflected a lack of progress in obtaining the answer to the questions. Among the five questions identified as difficult, three are directly related to the discovery of API elements (Duala-Ekoko and Robillard, 2012):

(A) *"How do I create an object of a given type without a public constructor?"*
(B) *"Does the API provide a helper-type for manipulating objects of a given type?"*
(C) *"How is the type X related to the type Y?"*

All of these questions registered a high ratio between the number of times they were observed and the difficulty of answering them each time. The remaining two questions that were considered difficult pertain to other kinds of difficulties that we do not address in our approach, namely high-level documentation content ("Which keywords best describe a functionality provided by the API?") and operation-level semantics ("How do I determine the outcome of a method call?").

In another study not specific to API usability (Sillito et al., 2008), the authors cataloged types of questions programmers ask during software maintenance tasks. Some of these questions strongly relate to discoverability. Within a category of questions related to relationships between entities (*"understanding a subgraph"*), the authors identified the questions *"How are instances of these types created and assembled?"* (related to question A above) and *"How are these types or objects related?"* (related to B and C).

The questions listed above relate to two different issues related to objects — creation (A) and manipulation (B and C). However, they have in common the problem that what the user wants to achieve using an object type is not apparent just from the information in the type that originates the question. (Although this is not necessarily true for question A, given that a static factory method might be present in the same type, often that is not the case, such as when having an object factory or an object builder).

A previous API usability study (Ellis et al., 2007), which strongly relates to question A, investigated the effect of having the factory and abstract factory patterns used in an API. The study revealed that factories are detrimental to API usability, given that users required significantly more time when using factories in contrast to constructors. The reasons relate to the difficulty in finding and using the factory types and methods. Another previous API usability study (Stylos and Myers, 2008), which strongly relates to questions B and C, investigated the implications of method placement in API learnability. Users often start their exploration of an API from certain types they believe to be relevant to their goals. The problem arises when the functionality that one wants to reach is present externally in other types. The study revealed that when the relevant API types are not accessible from the starting types, the time to discover them tends to be significantly higher. These difficulties pertaining to discovering relations between API types were confirmed by another empirical study (Piccioni et al., 2013), which also concluded that accurate documentation is crucial for good API usability.

3. Related work

The difficulties we detailed from the aforementioned user studies can be classified as *discovery* problems, given that it is not obvious to API users how to find the required API elements. The survey by Robillard et al. (2013) overviews a wide range of approaches for automated API property inference techniques addressing various goals, some of which related to discoverability, namely API documentation, understanding, navigation, and recommendations. Our approach also targets these goals. However, although we provide automated support for assisting API usage, our approach is not an inference technique, but rather a technique to augment APIs with the necessary information to achieve the same outcomes. Several approaches to assist API users have been previously proposed, which can be divided into two broad categories: alternative forms of API documentation, and intelligent code completion in an IDE. Within these categories, the different approaches essentially rely either on existing API usage examples to propose recommendations, or additional artifacts that complement the API in order to assist their usage.

The Jadeite tool (Stylos et al., 2009) provides documentation in the style of Javadoc that takes advantage of existing code corpora to facilitate API learning and navigation through the documentation, featuring font size differentiation of API elements according to frequency of usage found in the corpora, "placeholders" that are manually inserted in the documentation for API operations that a user would expect to be there but are not, and Javadoc extensions for including the most common ways of instantiating the API types, using examples that were found most frequently in the corpora. Another alternative form of documentation is provided by Apatite (Eisenberg et al., 2010), which supports visualizing and browsing API documentation through the associations among elements.

The Calcite tool (Mooty et al., 2010) is closest to Dacite with respect to the code completion proposals presented to API users. However, the two approaches achieve the end goal very differently, given that Calcite relies on existing API usage examples for assisting object creation and manipulation, capitalizing on Jadeite's database of discovered information. Calcite provides code completion proposals based on the most common ways of creating an object of a certain type, and method completion proposals based on "method placeholders" compiled manually, providing the user hints about how to achieve a certain goal, but in this case, without supplying working code. Calcite improved the success rate of API users by 40% on a comparative user study where participants had to complete programming tasks using APIs, providing evidence that discovery mechanisms combined with code completion may help API users significantly. The suggestions shown to API users in Dacite are modeled on Calcite's, but Dacite relies *only* on the API designer to annotate the API, while it also covers more design patterns. With respect to the code completion proposals supported by Calcite, Dacite's proposals can be considered equivalent from a user viewpoint.

The Graphite tool (Omar et al., 2012) provides developers with active code completion by means of specialized palettes, which provide alternative forms of instantiating specific types. For example, developers can use a color palette to generate code for instantiating `Color` objects. Although the Graphite mechanism may be used to instantiate objects of an API, its scope is limited to particular types, and it was not designed to provide assistance involving inter-type relations.

The API Explorer tool (Duala-Ekoko and Robillard, 2011) derives code completion proposals automatically from structural relationships between API types. Despite some differences in the way code completion proposals are presented and programmers are assisted, the main advantage of this approach over ours is that it is automated, requiring no enhancements to the API itself. On the other hand, in our approach, the intent of the API designer is made explicit, resulting in the API designer staying in control of what is recommended to API user. Regarding object manipulation (helper methods), our code completion proposals resemble the ones of API Explorer, although the latter also exploits synonym search to find other methods that might be of interest, which has an added risk of false positives which Dacite avoids. With respect to object creation (factories), our code completion proposals are significantly different than API Explorer's. Our factory proposals

follow a stepwise goal completion by directing the programmer to the factory types, whereas API Explorer inserts the required code all at once. Both approaches have trade-offs. Whereas the latter may drive programmers to a solution in a faster way, the former enables a stepwise path to the solution where the developer decides at every step the direction to follow, becoming more aware of the impact that those decisions have on the solution.

Several approaches take into account the context in which the user is writing code in order to provide API usage recommendations. Keyword programming in Java (Little and Miller, 2007) is a technique that can be used to obtain type-correct code expressions that use an API from keywords typed by the API user, reducing the burden of having to remember the exact names of identifiers and forms of composition. The Strathcona tool (Holmes and Murphy, 2005) is capable of recommending source code examples from a repository. The Prospector tool (Mandelin et al., 2005) synthesizes code fragments using both the code examples of a repository and API structural information. MAPO (Zhong et al., 2009) is a tool to mine API usage patterns that helps users to locate useful code examples. PARSEWeb (Thummalapenta and Xie, 2007) is a tool that also relies on searches on code repositories, where users may write queries that provide a source and destination type in order to obtain relevant method-invocation sequences.

In Bruch et al. (2009), the authors propose to modify how code completion menus order the proposals, based on a previous automated API usage learning process from source code repositories. The operations that are most likely to be used given the current context have higher priority (appearing first in the list of proposals). APISTA (Santos et al., 0000) is a tool for code completion capable of recommending subsequent calls to complete API sentences, based on training n-gram language models built from source code repositories. The focus in that system is to predict the next instruction that a programmer is likely to write given the immediate context (instructions written before the code completion is requested).

The main drawback of the approaches based on source code examples is that they are dependent on the existence of such a corpus and on its quality. Therefore, for APIs that were recently created or for which there is a limited collection of examples (coverage and amount), these approaches will not be effective. Another aspect that differentiates our approach is that Dacite reveals *all* the forms that API designers intended for an API task to be accomplished, instead of only the most frequently used ones. Approaches for recommending related functions (e.g., Saul et al., 2007; Long et al., 2009) generate associations between functions based on static analysis of the API implementation, not relying on source code corpora or additional documentation artifacts. However, these approaches were designed for procedural programming (C language), without taking into account the characteristics of object-oriented APIs. On the other hand, (Robillard, 2005) proposes automatic suggestion of object-oriented program elements of potential interest for developers. Although this approach could be used for API discovery, it was designed for program maintenance in general.

The eMoose tool (Dekel and Herbsleb, 2009) adds *directives* about API usage (i.e. rules or caveats about certain API operations). Directives are written by API developers in free-form documentation text that is not strongly linked to the source code. IDEs can then decorate in the code editor the operation calls that have associated directives, raising API users' awareness of their presence. However, directives do not aid API users with discoverability.

The *extension method* mechanism of C# partially addresses the difficulty related to the discovery of helper methods. Extension methods enable external classes to define static methods that can be used as if they were instance methods of another type, facilitating exploration and code completion. However, the possibility of using the extension methods in that way is only activated once the API user writes the import declaration[3] of the package in which the extension methods are declared. Therefore, the problem of having to know which package to import, which may be unknown to the API user, still results in a discovery barrier, given that without knowing about the package beforehand this mechanism cannot help the user to discover the helper methods. In contrast, the code completion proposals of Dacite do not require the programmer to write the relevant import statements previously.

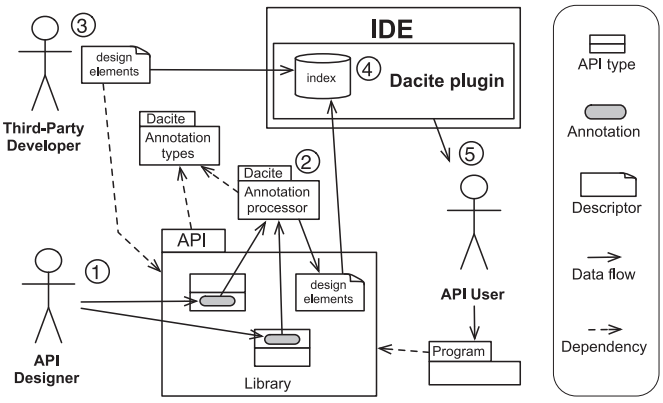

Fig. 2. Dacite overview: (1) API designers annotate elements of a library; (2) the annotation processor generates a descriptor of design elements to package with the library; (3) third-party developers may write descriptors to define external design annotations for an API; (4) the IDE plugin loads the design elements information of all the available descriptors into an in-memory index; (5) API users program against the library using Dacite's code completion proposals.

The IDE IntelliJ IDEA (JetBrains, 2014) provides code completion proposals on assignment by searching on the projects' build path for static methods that return a compatible type. Although these proposals cover static factories, no support is given regarding factory methods, helper methods, or object composition, which are supported by Dacite.

4. Approach overview

We address the API discovery difficulties identified in the studies detailed previously, providing an innovative approach that consists of annotating APIs with additional information that is processed by IDEs, which in turn, provide code completion proposals that help to discover API elements.

Fig. 2 presents an overview of our approach, for which we have implemented a proof of concept for Java and the Eclipse IDE. Dacite annotations are used by API designers on libraries they own, inserting the annotations into their source code (a concrete example is shown in the upper part of Fig. 1). The *annotation processor* validates annotation usage, producing compile-time errors when they are used incorrectly. Validations are an important issue, given that a design annotation used incorrectly would significantly confuse the API user by providing misleading or even incorrect recommendations. The annotation processor also extracts information from the annotations, outputting *descriptors* that conform to a schema for describing design elements. Each descriptor is bundled together with the library package. In some cases it will be useful to annotate an API *externally* (as opposed to *internally*), such as when the API cannot be modified due to not having its ownership. Third-party developers may define external annotations by writing descriptors of design elements. Both internal and external descriptors of the available APIs are gathered in an in-memory *index* loaded by the IDE. The index contains entries that encode associations (*structural element* $\mapsto$ *design element*), for instance, repre-

[3] In C#, this corresponds to the `using` directive.

senting that a given class (structural element of the API) embodies a factory (design element). The Dacite IDE *plugin* provides code completion proposals to API users based on the index information.

Annotations are a form of metadata for enriching source code with additional information. In the presence of non-obvious relations between API types, API designers annotate relevant members (types, methods, constructors, and parameters), explicitly documenting the design with specific annotation types for each type of design decision. For instance, the API designer may annotate an API type as representing a factory. Section 5 presents in detail the different annotations of Dacite, how they can be applied by API designers, and which code completion proposals are presented to API users.

The number of design annotations that an API designer must write is directly related to the API size and the number of design decisions that involve using the API in non-obvious ways. Given that the number of designers of an API is likely to be very small proportional to the number of its users, the additional work involving the design annotations has a potentially large impact, because it will bring benefits to the large population of API users. Several APIs with design annotations can be used simultaneously by users. This does not impose scalability issues regarding the code completion proposals, because the number of proposals that are presented is directly related to the number of annotated design decisions that involve the specific API type with which the user is working.

4.1. Implementation

In Java, annotations are treated like types (syntactically, they are preceded by an "@"), and hence, they are type-checked by the compiler. Further, annotations may have parameters whose arguments hold values. When defining annotation types, it is only possible to specify on which kind of member they may be applied (e.g., method, constructor, parameter, etc). However, Dacite's design annotations require more complex validations when applying them that we check. For example, the annotation `@StaticFactory` can only be applied to a public static method returning a reference type. These validations are performed by the annotation processor, emitting compiler error messages when violations are detected. Annotations may have different retention policies, which imply that they are encoded in the binary `.class` files or not. Given that the information embodied in the annotations is stored in the descriptors of design elements (recall Fig. 2), annotations are not required to be present at runtime[4] (all the relevant information in gathered in the index). We developed the annotation processor in Java through the provided standard infrastructure for extending the compiler with respect to annotation handling.

We developed a prototype plugin for the Eclipse IDE using its code completion extensibility mechanisms, allowing several APIs supplied with design annotations to be used simultaneously. The IDE code completion proposals are based on the in-memory index. The index is populated with the information provided in the descriptors of design elements, which are XML files that encode the annotations on the structural members of the API. Fig. 3 presents an example excerpt of a descriptor to annotate a static factory. So far, we have not developed a specialized editor for external third parties to use to annotate APIs externally, but this would be easy to achieve in the future.

For internal API annotations, the descriptors that are produced by the annotation processor are packaged in the JAR file of the library. In order to populate the index, the IDE looks up for descriptors in all the JAR files available in the build path. With respect to external annotations, descriptors are provided separately as Dacite plugins (no code has to be provided), which are loaded at runtime by the IDE and are also used to populate the index.

```
<type type = "javax.net.SocketFactory">
    <StaticFactory name = "getDefault"/>
</type>
```

Fig. 3. Example of design element descriptor representing a static factory annotation.

4.2. Interface for API users

From a user perspective, there could be Dacite plugins for different IDEs, which make use of the design annotations in the same manner given that the annotation types are independent from the IDE plugin. API users are provided with the additional code completion proposals in the form of existing interactive mechanisms (as shown in Fig. 1), integrating seamlessly with existing IDE facilities for code completion. API users do not need to be aware of the design annotations in order to benefit from the code completion proposals. These proposals take into account the code editing context in which the API user solicits proposals. There are two types of context, namely when a certain type is expected (e.g., on assignment) and when a compatible operation is expected (e.g., on invocation):

```
List list = ___        (expected type)
list.___               (expected operation)
```

In the case of an *expected type*, it is appropriate that the IDE recommends expression proposals which are compatible with the expected type, i.e. that evaluate to the same type or a subtype of it (covariant). For instance, in the assignment above (considering the Java API), the expression after the equals sign could, for example, be an instantiation of `ArrayList` or `LinkedList` (both implement the interface `List`). On the other hand, in the presence of an *expected operation* it is appropriate that the IDE recommends proposals of operations available for the type, including the available operations of its supertypes. In the example above, not only the operations of `List` are appropriate to recommend, but also the operations of `Collection` and `Iterable` (supertypes).

5. Design annotations

This section describes the design annotations we have implemented so far. Each annotation is explained in terms of its purpose, how it can be used, and how the information it embodies can assist API usage. We use diversified examples from existing APIs that are widely used, such as Java's standard libraries, Eclipse's Standard Widget Toolkit[5] (for developing graphical user interfaces), Google's Guava libraries[6] (that offer additional data structures and utilities to complement Java's standard libraries), and JFreeChart[7] (for drawing various kinds of charts in a Java Swing application; used in our Section 7 and previous (Duala-Ekoko and Robillard, 2012) user studies). We present the original type names and signatures (possibly omitting irrelevant parameters), as well as excerpts of the source code comments of their Javadoc documentation when relevant. Occasionally, we underline parts of the documentation text to emphasize the relation to our annotations.

[4] In Java, this is equivalent to an annotation retention policy of either `SOURCE` or `CLASS`.

[5] www.eclipse.org/swt.

[6] github.com/google/guava.

[7] www.jfree.org/jfreechart.

Validations: `@StaticFactory` is used on a public static method, contained in a public class, that returns a public reference type.
Proposals: When an expression of type t is expected, every annotated static factory that returns an object compatible with t gives rise to a proposal to call the static method as the assignment expression.
Example usage:

```
public class SocketFactory {
  /**
   * Returns a copy of the environment default socket factory
   */
  @StaticFactory
  public static SocketFactory getDefault() {...}
  ...
}
```

Fig. 4. `@StaticFactory`: validations, proposals, and example usage.

5.1. Static factories

The normal and simplest way to create objects of a certain class is to use an available constructor of that class. However, the solution to certain design problems may disallow having public constructors in favor of *static factories* (Bloch, 2008). These are static methods that create or obtain objects of a certain type, enabling developers to control class instance creation in a more flexible way. For instance, a static factory method may be used to implement a design solution based on the *flyweight* or *singleton* patterns (Gamma et al., 1995). Static factories may be located in the class whose type matches their return type, but often they are located externally in another class. As examples of static factories, Java's `SocketFactory` has a static method `getDefault()` for obtaining the default socket factory, JFreeChart has a class `ChartFactory` that contains several static factories for creating the different kinds of charts.

In order to enable API designers to annotate static factories, we provide the annotation `@StaticFactory`. Fig. 4 describes this annotation in terms of the validations regarding where it can be applied, the code completion proposals that are implied by using the annotation, and an example of using the annotation on Java's `SocketFactory` class.

The information pertaining to static factories can be used by the IDE whenever code completion is requested on any location where the created object is compatible with the expected type (e.g., variable assignment `SocketFactory sf = ␣`). These recommendations aid the API user in discovering static factories, whose location might not be obvious.

Given that often a class may contain several static factories that return objects of the same type (or subtypes), we also provide an annotation `@StaticFactories` that can be used to annotate a class to denote that several static methods in that class are static factories, avoiding having to annotate each method. In this case, one has to specify in an annotation parameter the types that should be considered to match the static factories.

5.2. Factory methods

The design patterns *factory method* and *abstract factory* (Gamma et al., 1995) are appropriate when there is a need to abstract the creation of particular object types, enabling client code not to depend on the concrete classes of the created objects. These solutions offer reuse and flexibility because new object types can seamlessly be integrated in a system without changes to the factory clients. A factory method is an abstract method that is meant to be implemented in concrete methods, which possibly return objects of a covariant type. An abstract factory groups several factory methods, providing an interface for creating related objects without specifying their concrete classes. As an example, Java's `SocketFactory` class consists of an abstract factory, which contains several factory methods for creating network sockets.

In order to enable API designers to annotate (non-static) factories and their methods, we provide the annotations `@Factory` and `@FactoryMethod`. The former is used to annotate an abstract class or an interface, meaning that the corresponding type represents a factory. The latter is used to annotate the factory methods of a factory type. Fig. 5 describes these annotations, using Java's `SocketFactory` as an example. Although a class could be considered a factory simply if it contains at least one method annotated with `@FactoryMethod`, we decided to enforce using the `@Factory` annotation on the type because it corresponds to a well-defined role of the Abstract Factory pattern (Gamma et al., 1995), and hence, marking the type as such is useful for documentation purposes.

As with static factories, the IDE can make use of the information provided by these annotations in order to recommend the usage of a factory. However, the case is slightly more complex, because instead of a direct call (static) to obtain the desired object, a factory object itself has to be obtained first, so that one of its factory methods can then be invoked on it. In this way, when for example code completion on assignment is requested (e.g., `Socket s = ␣`), the proposals consist of obtaining a reference to a factory object that is able to create the desired object (`SocketFactory` in this example). However, in accordance to the abstract factory pattern, the factory type will most likely be an abstract type, while several concrete factories are available. Therefore, a reference to the factory object may be obtained using different mechanisms (e.g., a constructor call or a static factory), and this step may consist of another discoverability hurdle. For instance, in the given example, a `SocketFactory` may be obtained through two static factories (one shown in the previous example and another in a different class, `SSLSocketFactory`). This is a motivation for why the IDE only fills in one recommendation at a time, as illustrated in Fig. 10.

Similarly to the case of static factories, abstract factories can alternatively be specified using an optional parameter on the `@Factory` annotation. This parameter indicates which types should be considered to match factory methods, in order not to require each factory method to be annotated individually. For instance, in `SocketFactory` there are actually 5 factory methods whose return type is Socket.

Validations: `@Factory` is used on a public class or interface that contains at least one of its methods annotated with `@FactoryMethod`. The latter is used on public instance methods that return a public reference type.
Proposals: When an expression of type t is expected, every annotated factory class f that holds at least one factory method that returns an object compatible with t, gives rise to a proposal for obtaining a reference of type f (prior to the assignment, Figure 11 illustrates this in practice).
Example usage:

```
/**
 * This class creates sockets. It may be subclassed by other factories,
 * which create particular subclasses of sockets. (...)
 */
@Factory
public abstract class SocketFactory {
  /**
   * Creates an unconnected socket.
   */
  @FactoryMethod
  public Socket createSocket(...) {...}

  @FactoryMethod
  public Socket createSocket(..., ...) {...}
  ...
}
```

Fig. 5. `@Factory` and `@FactoryMethod`: validations, proposals, and example usage.

5.3. Builder objects

Both standard constructors and (static) factory methods do not scale well when facing the need of having a large number of parameters, some of which may be optional. The problems stem from requiring several constructor variants, which can easily be confused, both by implementers and consumers, especially due to the absence of named parameters. A *builder* (Bloch, 2008) is an object whose purpose is to assist the construction of another, complex object (we are not addressing the Builder pattern exactly as described in Gamma et al. (1995), but a simpler and frequent variation of it). One may use a builder to address the described parameters problem, or to guarantee safe creation of immutable objects. For instance, in Google's Guava libraries, several immutable collection types (e.g., `ImmutableList`) have static member classes implementing builders. A builder class always has a method typically named "build" or "create" that returns the object that it constructs.

As with factories, using object builders requires that objects cannot be created in the most obvious way, i.e., using a constructor of their class. Moreover, most likely the builder classes are not accessible from the class of the objects they create. Due to these reasons, the discovery of builders may also be a barrier for API users. The "build" method resembles a factory method, but given that the typical way object builders are used in client code is significantly different than factories, we decided to have a dedicated annotation for these. Therefore, we provide the annotation `@Builder` to mark object builders. This is done by annotating the "build" method of a builder class, denoting that such class is a builder of objects of the type equal to its return type. Fig. 6 describes this annotation using Guava's `ImmutableList` builder as an example.

As with factories, the IDE can recommend the use of a builder on expected type code completion requests. For instance, on assignment to a variable of type `List` (e.g., `List l = __`), recommendations include a proposal to instantiate a `ImmutableList.Builder`.

5.4. Helper methods

Often API designers place *helper methods* (also referred to as *utility methods)* that are essential or useful to objects of a certain type on classes other than the ones that implement that type. Such design options have their advantages, which may be related to information hiding or reuse. However, as found by previous studies (Stylos and Myers, 2008), placing the methods external to the types where they are useful hinders API learnability and discoverability because API users typically use the types they are working with as starting points.

We provide the annotation `@Helper` to associate helper methods with other types. The annotation is used on a single parameter of the helper method, denoting that the owner method is a helper for the annotated parameter's type. We decided to use a parameter annotation rather than a method annotation because, in the presence of several parameters, it would not possible to infer which parameter would represent the type to which the method is a helper (without resorting to a convention that would compromise flexibility). Fig. 7 describes this annotation using as an example JFreeChart's `ChartUtilities` class to denote a helper method which saves the chart as a PNG image.

Since users will often use autocomplete to try to discover operations on a given variable, the IDE can include the helper methods based on the information provided by the annotations, along with the available instance operations. Using this example, if we have a variable compatible with the type `JFreeChart`, when requesting code completion on expected operation (`chart.__`), the proposals include the static methods that are applicable according to the annotated type.

Given that it is common to find classes that contain several helper methods for a same type, we provide the annotation `@HelperClass` to annotate a class containing helper methods. The annotation requires a parameter to indicate which types will be considered as targets of the helper methods contained in the class. For instance, in the JFreeChartAPI there is

Validations: `@Builder` is used on a single instance method of a public class (if an inner class, then it has to be static). The method returns a public reference type and the class has at least one public constructor.
Proposals: When an expression of type t is expected, each public constructor c of every builder class that holds a builder method b whose return type is compatible with t, gives rise to a proposal to invoke c chained with the invocation of b.
Example usage:

```
public class ImmutableList<E> implements ... {
  /**
   * A builder for creating immutable list instances, (...)
   */
  public static class Builder<ImmutableList<E>> {
    /**
     * Returns a newly-created ImmutableList based on the contents
     * of the Builder.
     */
    @Builder
    public ImmutableList<E> build() {...}
    ...
  }
  ...
}
```

Fig. 6. `@Builder`: validations, proposals, and example usage.

Validations: `@Helper` is used on a parameter whose type is a public reference type and is owned by a public static method contained in a public class.
Proposals: When listing the available operations for a reference r of type t, each helper method that has an annotated parameter p compatible with t type gives rise to a proposal to invoke the method using r as the argument for p.
Example usage:

```
/**
 * A collection of utility methods for JFreeChart.
 * Includes methods for converting charts to image formats (PNG and JPEG)
 * plus creating simple HTML image maps. (...)
 */
public class ChartUtilities {
  /**
   * Saves a chart to a file in PNG format.
   */
  public static void saveChartAsPNG(..., @Helper JFreeChart chart, ...){...}
  ...
}
```

Fig. 7. `@Helper`: validations, proposals, and example usage.

Validations: `@Parent` is used on a single parameter of a public constructor of a public class.
Proposals: When listing the available operations for a reference r of type t, each constructor c with a annotated parameter p whose type is compatible with t gives rise to a proposal to invoke c using r as the argument for p.
Example usage:

```
public class Button extends Control {
  /**
   * Constructs a new instance of this class given its parent and a style value
   * describing its behavior and appearance. (...)
   */
  public Button(@Parent Composite p, ...) {...}
  ...
}
```

Fig. 8. `@Parent`: validations, proposals, and example usage.

Validations: `@Decorate` is used either on: a parameter of type t of a public constructor of a public class c, given that t is compatible with the type of c; or a parameter of type t of a public static method of a public class whose return type is compatible with t.
Proposals: When listing the available operations for a reference r of type t, each constructor or static method m with an annotated parameter p whose type is is compatible with t, gives rise to a proposal to invoke m using r as the argument for p.
Example usage:

```
public class Collections {
  /**
  * Returns an unmodifiable view of the specified set. (...)
  */
  public static Set<T> unmodifiableSet(@Decorate Set<? extends T> set) {...}
  ...
}
```

Fig. 9. `@Decorate`: validations, proposals, and example usage.

the `ChartUtilities` class that contains more than 15 helper methods for `JFreeChart` objects. As another example, the `java.util.Collections` class also contains several helper methods for `List` and `Collection` objects.

5.5. Object composites

When designing solutions based on the *composite* pattern (Gamma et al., 1995), hierarchical tree structures are formed, where parent objects hold child objects of different classes through a uniform interface. The concrete child objects' types are unknown to the parent object, and therefore, despite the fact that they all have a common supertype, it is not obvious which child types a composite object may hold. For instance, in SWT's widgets, there is the widget type `Composite`, which may hold child widgets of type `Button`, `Label`, `Text`, etc., and `Composite` itself. However, the operations of `Composite` do not enable users to add children directly (that is, there is no "add" method on the `Composite`). Instead each child must be included in the parent composite by passing a reference to the parent when the child is constructed. Given that the parent types have no direct dependencies to their child types, it might not be obvious to API users which types of children a certain composite type may have or how to add them.

We provide the annotation `@Parent` for making explicit in the child role the relationship to its parent object. Fig. 8 describes this annotation using the `Button` class of the hierarchy of SWT's widgets. In this example if code completion is requested on a variable of type `Composite` (`comp.`␣), the proposals include the creation of a `Button` object using the variable as the parent object (`new Button(comp, ...)`).

5.6. Object decorators

Also as a form of object composition, the *decorator* pattern (Gamma et al., 1995) enables an object to be adapted at runtime by wrapping it in a decorator object. As with composites, decorator types are often not accessible using the documentation or IDE from the types they are able to decorate. Since a decorator is capable of modifying the behavior of an object of a certain type, we argue that references of that type are natural starting points to discover the functionality offered by the decorators. We provide the annotation `@Decorate` for making explicit in the decorator role the relationship to the decorated object. Fig. 9 describes this annotation using a method of Java's `Collections` class to decorate a `Set` so that it becomes wrapped in an immutable view. In this example, if code completion is requested on a variable of type `Set` (`set.`␣), the proposals include the invocation the decorator instantiation operation using the variable as the decorated object (`Collections.unmodifiableSet(set)`).

Table 1
Design annotations and their relation to API discoverability questions (detailed in Section 2).

Question	Annotations
A (object creation)	`@StaticFactory`, `@Factory`, `@Builder`
B (helpers)	`@Helper`
C (type relations)	`@Parent`, `@Decorate`

5.7. Summary

The annotations presented in this section consist of structured information that documents design decisions. The information can be consumed by an IDE in order to assist API users with discoverability difficulties (see Section 6). In Section 2 we described questions related to API discoverability hurdles that were identified in empirical experiments. Table 1 summarizes the relation between the provided annotations and those discoverability issues.

6. IDE integration

This section explains how IDEs may gain leverage from design annotations. We illustrate the code completion mechanisms with the Dacite plugin that we developed for Eclipse. As explained in Section 4, the plugin makes use of the design annotations to assist API users, capitalizing on two existing code completion mechanisms that operate on the contexts mentioned earlier – *expected type* and *expected operation*. In this way, code completion proposals that originate from the design annotations are seamlessly integrated with the existing IDE facilities, which are already familiar to users.

6.1. Expected type proposals

In Eclipse, expected type proposals can be requested from the IDE when writing an assignment statement by typing control-space. In order to illustrate expected type proposals,

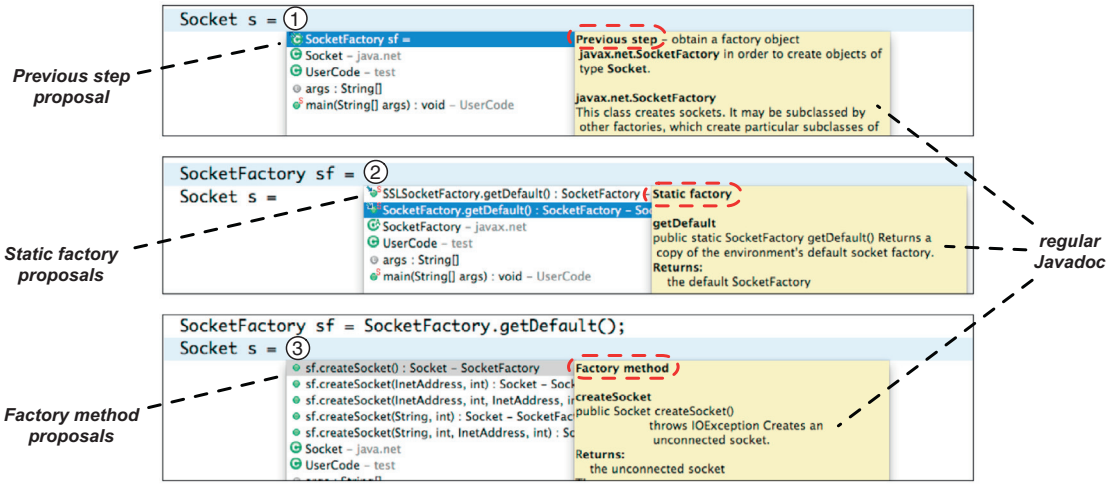

Fig. 10. API usage assistance in an expected type context. Illustration based on the examples introduced in Sections 5.1 and 5.2. (1) Proposal to obtain a `SocketFactory` to create `Socket` objects; (2) proposals to obtain the `SocketFactory` via static factories; (3) proposals to use the factory methods to create the socket.

we present a user interaction example in Fig. 10, involving Java's `SocketFactory` class which we mentioned earlier. Assume that the static factory and factory methods related to this part of the API are annotated as in the code snippets of Sections 5.1 and 5.2.

As a starting point, in step 1 of Fig. 10, the user is writing an assignment to a variable of type `Socket`, whose instances can be created through a factory (`SocketFactory`). When requesting code completion assistance, the user is presented with a proposal for obtaining a reference to an object of type `SocketFactory`. This proposal is derived from the fact that this type has factory methods (`@FactoryMethod`) that return references that are compatible with the expected type. This proposal leads the user to the accomplishment of a previous step (obtaining the factory). When accepting the proposal, the reference initialization instruction for the factory type is inserted *before* the instruction where the request was triggered, and the required import declaration is inserted in the class header if not present. A complete instruction is not inserted, as there might exist more than one way to obtain a compatible factory object. In step 2, the user again requests code completion assistance, this time for obtaining a `SocketFactory` reference. Given that there are static factories (`@StaticFactory`) on two classes (`SocketFactory` and `SSLSocketFactory`), the system provides two proposals, one for each. After selecting one of the proposals, the code is inserted on the right-hand side of the assignment. At this point, the user has completed the recommended previous step of acquiring a reference to a relevant factory. In step 3, the user requests code completion assistance as in step 1 once again, returning to the initial goal of creating a `Socket` object. At this point, given that there is an available reference for an appropriate factory, the proposals consist of using that reference to create the desired object through the factory methods which are available. Note that the Dacite code completion proposals change depending on the context with respect to other code that is already above the invocation point.

The design element involved in each proposal appears in the documentation tooltip, accompanied by the regular Javadoc contents for the API element. As in conventional settings, these tooltips assist the user in making decisions regarding which elements to use. An alternative to the adopted stepwise approach (suggesting previous steps) would be to suggest the factory obtainment and factory method usage as a single step (e.g., as in Calcite (Mooty et al., 2010) and API Explorer (Duala-Ekoko and Robillard, 2011)). We decided to decompose the process as explained here, given that in cases where several factory types and methods exist, a large number of proposals would be presented to the user at once. Calcite does not have this problem since it provides only the most popular operations to create objects, rather than the comprehensive technique here, which provides users with *all* of the relevant operations provided by the API developers.

6.2. Expected operation proposals

In Eclipse, when manipulating a reference type variable, users may request code completion assistance for manipulating that variable to see all of the available operations. The fact that the user wrote the variable name can be interpreted to mean that he or she wants to do something with it. Based on this assumption, our system adds additional code completion proposals where the variable can be used as a parameter by annotated helper methods and object compositions.

Regarding the helper methods (see Fig. 11), the Dacite plugin adds additional proposals interleaved with the regular type operation proposals. The proposals appear according to alphabetical order, considering the name of the helper method. In the example, we can see that the user wrote an "s" after the dot for invoking an operation on `chart`, and that the two helper methods (that start with an "s") are being recommended. When accepting one of these proposals, the line of code that the user is writing is "tweaked" in order to form a method call instruction that uses the variable which the user is working with. The user is then responsible for filling in the remaining parameters. Although it is somewhat unusual for a code completion action to "tweak" the code that has already been written, the alternative is to simply add comments as in Calcite (Mooty et al., 2010), but that requires the API user to perform edits manually that the IDE is able to do automatically.

With respect to object composition proposals (see Fig. 12), the Dacite plugin introduces object creation proposals on the upper part of the code completion menu. This group of proposals is sorted alphabetically according to the name of the class involved the proposal. In the example, we can see that for the variable `composite`, the plugin recommends the creation of widgets (`Button`, `Group`, etc.) having the composite object as parent, as well a proposal recommending the creation of a `ControlDecoration` object. As with helper methods, when accepting a proposal of this kind, the line of code that the user is writing is "tweaked" in order to form an instruction that invokes

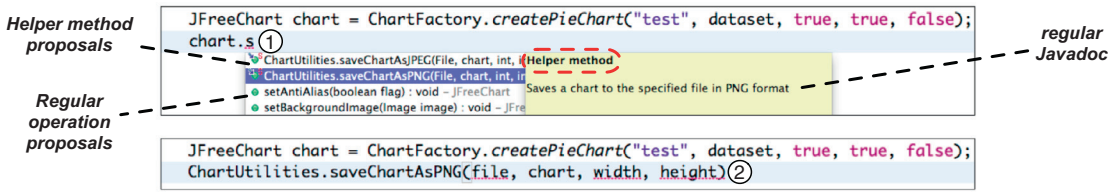

Fig. 11. API usage assistance regarding helper methods in an expected operation context, illustrated with the example of Section 5.4. (1) Proposals of helper methods for the `JFreeChart` type; (2) code insertion of the selected proposal.

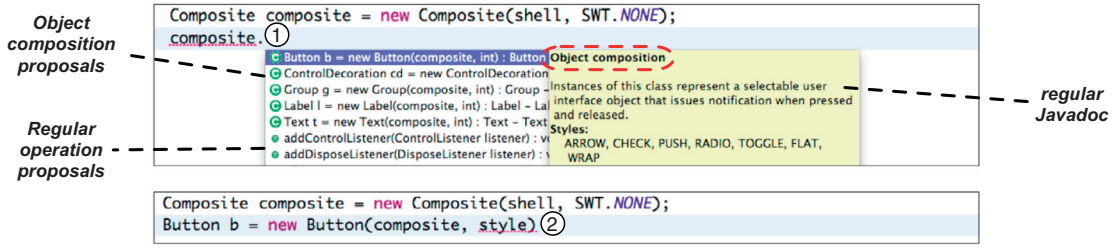

Fig. 12. API usage assistance regarding object composition in an expected operation context, illustrated with the example of Section 5.5. (1) Proposals of object compositions for the `Composite` type; (2) code insertion of the selected proposal.

a constructor of the recommended class using the variable as a parameter.

7. Evaluation

Section 6 provides evidence that our features would be useful in helping developers answer discovery questions, but it remains to be shown that our design is usable. Therefore, we conducted a user study where programmers were asked to accomplish programming tasks on unfamiliar APIs. Our hypothesis was that Dacite's mechanisms would help programmers discover the required information about the APIs and therefore accomplish the given tasks more successfully.

7.1. Method

The study was composed of two programming tasks, each on a different API that was unfamiliar to the participants. We used a between-subjects design for each task, with a control group using regular Eclipse and an experimental group using Eclipse with the Dacite plugin.

We recruited participants from both institutions to which the authors of this paper are affiliated (Carnegie Mellon University and University Institute of Lisbon). Even though conducted at universities, we only accepted participants who were experienced programmers. Study participants were recruited at our institutions through mailing-list posts and direct invitation or recommendation from previous participants. In a pre-study questionnaire, participants indicated they had on average 4 years of programming experience with Java. Using a 7-point Likert scale (1–very little, 7–very much), we asked participants how familiar they were with Eclipse (median response 5) and how often they use code completion menus (median response 6). Given that previous experiments revealed that the answers given by participants in questionnaires is not always reliable (Siegmund et al., 2014), before performing the actual study tasks, participants were screened on-site to ensure that they had reasonable programming experience in Java by requiring them to accomplish a small warm-up task. Three study candidates were excluded after failing or taking excessive time to accomplish the task (i.e. more than 10 min, given that the expected time was 2 min). This resulted in 16 actual participants (10 males, 6 females) with ages ranging 21–35. The academic background of participants was distributed among Computer Science and Engineering, Information Systems, and Telecommunications and Networks. There were 10 participants enrolled or recently graduated from an MSc program, and 6 participants who were PhD students. Participants were compensated for their participation in the study with 15 American Dollars (USA) or 10 Euro (Portugal).

The study sessions took place on both sites using the same procedure. The tasks were performed on a Mac computer with a 21-inch screen, using Eclipse 4.2. Study sessions were screen-recorded for detailed analysis. Each participant performed one task in the control condition and the other task in the experimental condition. (We classify this as a "between subjects" experiment because in the results below, we compare only by task, not by participant – that is, we compare the 8 control and 8 experimental results for the JFreeChart task, and separately the 8 control and 8 experimental results for the JAXP task. Since the tasks were not of identical difficulty, it would not have been valid to compare a single participant's results from the two tasks.) The assignment of participants to the two tasks and different conditions was randomized, considering that each task would have 8 participants in each condition, 4 of which performing first in the control condition and the other 4 participants performing first in the experimental condition, counter-balancing the condition order to minimize possible learning effects.

Before carrying out the task in the experimental condition, participants were given a 3–5 min walk-through tutorial in order to demonstrate the types of code completion proposals that Dacite could suggest. The tutorial was carried out by participants themselves and the examples were based on classes from the standard Java API (e.g., collections, sockets), which do not relate to the classes used in the study. Therefore, the study tasks were not blind experiments in that the participants knew which condition they were in. We felt that it was necessary to inform them about the new kinds of code completion so that they would not be surprised when facing them during the tasks. However, although this may have biased the participant's reported opinions about the tool, we do not feel it impacted the performance measures. After the study session, each participant filled in a post-study questionnaire containing subjective questions related to the usability and desirability of Dacite's code completion proposals and open questions concerning feedback and suggestions for improvement.

```
DefaultPieDataset data = new DefaultPieDataset();
data.insertValue(0, "Undergrads", .45);
...
JFreeChart chart = ChartFactory.createPieChart("..", data);
ChartPanel cp = new ChartPanel(chart);
ChartUtilities.saveChartAsPNG(file, chart, 500, 500);
window.setContentPane(cp); // given as window.setContentPane(null)
```

Fig. 13. Possible solution for the JFreeChart task.

```
SchemaFactory factory = SchemaFactory.newInstance(lang);
Schema schema = factory.newSchema(schemaFile);
Validator validator = schema.newValidator();
StreamSource source = new StreamSource(dataFile);
validator.validate(source);
```

Fig. 14. Possible solution for the JAXP task.

7.2. Study tasks

We used the same tasks as in the exploratory study (Duala-Ekoko and Robillard, 2012) mentioned in Section 2, given that many of the API discoverability issues that we address with Dacite were identified in the context of those tasks. The tasks required using two different APIs, which we refer to as "JFreeChart" and "JAXP".

The tasks were described on a sheet of paper. A task description walk-through was given by the session supervisor in order to make sure that participants understood the task goals correctly. While performing the tasks, participants could not use the Web, but were provided offline with the Javadoc for the APIs. This ensured that the subjects only used the official standard documentation for the APIs. For each task, the main API package of interest was indicated to the participants, so we could focus on the discoverability of classes and methods within the package. We have been asked why we did not allow the participants to use search engines on the Internet during the study. For both tasks, the correct answer is readily available as a top search result, so this would not have measured the participant's ability to understand and use Dacite, which is the goal of the study.

Participants were given a maximum of 30 min to accomplish each task. If the participants indicated they thought they were done before the time expired, the session supervisor checked if the task was correctly completed, and if not, the participant was asked to continue working on it.

7.2.1. JFreeChart

This API renders charts in Java Swing windows. The programming task consisted of rendering a pie chart with a given data in a window and saving that same chart into a PNG image file. The code snippet in Fig. 13 presents one of the possible solutions for the task, omitting the code skeleton that was provided for launching the graphical application.

This task had 3 main discoverability hurdles where Dacite proposals could be helpful, namely with respect to static factories (`createPieChart(...)`), object composition (`ChartPanel`), and helper methods (`saveChartAsPNG(...)`). Although we did not anticipate that obtaining the dataset (first instructions) would consist of a barrier, since there was no "hidden" information with respect to dependencies, we noticed during the study sessions that this step did pose a barrier for some of the participants.

7.2.2. JAXP

This API addresses XML processing (construction, parsing, validation, etc.). The programming task required validating a given XML file against a given XML Schema. The code snippet in Fig. 14 presents one of the possible solutions for the task, also omitting the given code skeleton, which included instantiated `File` objects with the paths to the necessary files and the XML Schema language version identifier.

This task also had 3 main discoverability hurdles where Dacite proposals could be helpful, namely one regarding static factories (`newInstance(...)`) and two related to factory methods (`newSchema(...)` and `newValidator(...)`). As with the JFreeChart task, we did not anticipate that obtaining the `StreamSource` object would be consist of a barrier, for the same reason that there was no "hidden" dependencies. Again, we found that this step turned out to be a barrier to some participants. However, in this case there was an object composition proposal to obtain a `StreamSource` object from a file.

Table 2
Success rate of task completion.

Task	Control	Experimental
JFreeChart	3 / 8 (37.5%)	7 / 8 (87.5%)
JAXP	5 / 8 (62.5%)	8 / 8 (100%)

7.3. Results

We scored each participant task as successful or unsuccessful. We considered a task as successful if the task goal was correctly achieved within the 30 min. We were flexible and did not require that the solution was done the same way that we were expecting, given that in both tasks there were different ways to achieve the goals.

We found that the groups in the experimental condition were considerably more successful in both tasks. Table 2 summarizes the results for both tasks under the two conditions. Across both tasks, almost twice as many tasks were finished successfully by the experimental groups using Dacite (15 out of 16 vs. 8 out of 16). Only one participant failed both tasks (under the two conditions), whereas all the other participants succeeded in at least one task. All the participants that could not complete one of the tasks, failed the task on the control condition.

We analyzed our results by means of a model to predict task success obtained through mixed effects logistic regression (fixed and random effects)[8]. We considered a *participant id* variable representing each of the study participants as a random effect, whereas the following variables were treated as predictor factors: *task* (JFreeChart or JAXP), *condition* (control or experimental), *order*

[8] We used the `melogit` package of the statistical environment R.

Table 3
Usage of Dacite code completion proposals.

Task	Static Factory	Factory Method	Helper Method	Object Composition
JFreeChart	6 / 8	–	6 / 8	3 / 8
JAXP	7 / 8	7 / 8	–	1 / 8

(performing first or second under the experimental condition). We also considered an additional interaction variable *task* × *condition* to verify if the effect of being under the experimental condition was not reliably different among tasks.

The regression model revealed that being under the experimental condition was a strong predictor of being successful in the task (p-value = 0.0252). Therefore, we conclude that our tool contributed to the success of accomplishing the tasks, namely on overcoming the discovery barriers. This is backed up by the number of participants under the experimental condition that used the different types of proposals offered by Dacite (when applicable). This information was gathered from analyzing the recorded sessions and we summarize it on Table 3. On the JFreeChart task, the static factories and helper method proposals were used by most of the participants (6 out of 8), whereas the object composition proposals were less used (3 out of 8). On the JAXP task, both the static factories and the factory method proposals were used by almost all participants (7 out of 8), whereas the object composition proposal was used by a single participant.

The regression model also revealed that: (a) the greater success of accomplishing the JAXP task was marginally significant (p-value = 0.0533), indicating that this task was somewhat easier; (b) the order in which participants performed on the two conditions had no impact on the results (p-value = 0.9014); and (c) the effect of being under the experimental condition was not reliably different between the two tasks (p-value = 0.1612).

7.4. Observations

By analyzing the sessions' recordings, we collected a number of observations that either explain some of the obtained results or provide further insights regarding difficulties or participants' behaviors.

We observed that with the JFreeChart task when using Dacite, there were two participants that made use of a single code completion proposal, whereas regarding the JAXP task using Dacite there was only one participant that did not use any code completion proposals. By taking a closer look at the recordings, we found that these participants spent most of their time navigating through the Javadoc API documentation in the browser. These participants did not come across most of the relevant proposals simply because they requested code completion rarely, even though they were explicitly told during the tutorial that additional proposals from Dacite would be available. One of these three participants could not complete the JFreeChart, whereas the other two were comfortably successful (< 15 min.).

We conclude that the object composition proposals were less used because in JFreeChart some participants found by themselves the relevant class that Dacite would recommend when browsing the documentation (still, 3 out of 8 used Dacite's proposals successfully). In the case of JAXP, the object composition case was less critical for accomplishing the task and participants tended not to request code completion proposals for it. Therefore, we believe the non-use of Dacite's features in these cases does not reflect badly on Dacite's usability with respect to object composition proposals.

We surprisingly found that many of the participants had difficulty and spent considerable time in dealing with the situation where they were faced with API calls involving parameters whose type was an API interface (cases mentioned in Section 7.2). Having to browse the documentation to find an appropriate implementation of the interface was generally time-consuming, and in some cases revealed some participant disorientation while trying to overcome this difficulty. This issue is not about hidden dependencies, since all the necessary navigation links were available in the Javadoc, but perhaps due to a lack industrial experience involving aspects of object-orientation pertaining to abstract types.

7.5. Post-study questionnaire

In the post-study questionnaire, participants were mostly highly positive towards the code completion proposals provided by Dacite. Table 4 summarizes the results of the questions pertaining to the usability and usefulness of Dacite. Answers were given on a 7-point Likert scale, and the table presents median values and median absolute deviation (*MAD*).

The questionnaire included an open question for participants to express what they liked the most regarding the code completion proposals. Four participants mentioned issues related to the possibility of being able to discover API methods not accessible through the variable that is being manipulated, and two participants mentioned that the type inference and adaptation to the code that the user is writing was efficient.

The questionnaire also included an open question for participants to indicate drawbacks and suggest possible improvements to the code completion mechanisms. Three participants expressed the desire to have code completion aids to help choose a particular interface implementation, possibly taking into account the nearby variables. Two participants expressed the desire to have a more obvious visual distinction between the regular proposals and Dacite's ones, such as through visual elements such as icons and background colors.

In informal post-study conversations we became aware of two situations where participants successfully applied a Dacite code completion without noticing it, both pertaining to helper methods. This may be considered as anecdotal evidence that the code completion "tweaks" are so smooth that sometimes they are not even noticed.

7.6. Discussion

The results indicate that the Dacite mechanisms are usable and effective for helping participants overcome the targeted discoverability problems. Further, the presence of the Dacite mechanisms significantly improved the success of the participants with respect to task accomplishment. Participants in the experimental condition were able to successfully use the Dacite proposals in several situations as evidenced by data given on Table 3. Especially the code completion proposals pertaining to static factories, factory methods, and helper methods were clearly useful and usable, given that most of the participants using Dacite made use of them. Therefore, we conclude that the greater success rate of the experimental group was due to the aid provided by Dacite's proposals.

The high scores obtained in the subjective evaluation (Table 4) back up the study results, and we conclude that participants appreciated the added value that our code completion proposals could represent in real settings.

7.7. Threats to validity

7.7.1. Construct validity

We decided to evaluate our approach under controlled settings while having tasks that reflect real scenarios, i.e. a programmer wants to achieve a certain goal using a certain API. We replicated

Table 4
Post-study questionnaire results on Dacite's usability and usefulness ($n = 16$).

Question	Median ± MAD
How helpful were the completion proposals related to object creation? (1–very unhelpful, 7–very helpful)	7 ± 0
How helpful were the completion proposals related to object manipulation? (1–very unhelpful, 7–very helpful)	6 ± 1
How confusing do you find the code completion proposals that perform small changes in the code previously written by the user? (1–very confusing, 7–not confusing at all)	6 ± 1
Would you like to have the new type of completion proposals along with the standard Eclipse completions for your actual programming? (1–definitely not, 7–definitely)	7 ± 0

the study tasks of a previous experiment (Duala-Ekoko and Robillard, 2012), from which the discoverability hurdles were identified (Section 2). Since the tasks were previously successfully used in an experiment, this gave us some confidence that they would be adequate in our case. Further, the clearly identified discoverability hurdles stemmed from those tasks, and therefore, these would be good candidates to evaluate if our approach would help on overcoming the associated difficulties.

The relatively low number of study participants could give rise to uneven control/experimental groups with respect to Java programming skills. This led us to alternate between roles, so that each participant would fulfil one task under each condition. Informally, differences in Java skills seemed evenly distributed across conditions.

We previously mentioned the unexpected difficulty involving interfaces that some participants faced in the tasks. Participants in both conditions faced this difficulty, and therefore this issue had no significant impact on the results.

7.7.2. Internal validity

The selection of study participants may face a possible bias, since some of them (roughly half) were carrying out their work in the same or related research institute as the authors. We believe that this fact has no impact in terms of better or worse performance in the tasks. In order to minimize the possibility of participants being influenced by working nearby, we made sure that they did not have any sort of prior knowledge regarding the project.

The performance of participants could have been different if they were allowed to browse the Web, namely as a quest for examples that resemble or relate to the task goal. As mentioned above, we decided not allow participants to access the Web since this would not be testing Dacite's usability or effectiveness. Furthermore, using the Web would add an independent variable that is difficult to control. For instance, search hits of search engine such as Google vary from day to day, and participants would potentially obtain different search hits for the same query. Moreover, the original study (Duala-Ekoko and Robillard, 2012) that we replicated ran the tasks in two groups, one allowed to use the Web and the other not, and their results concluded that there was no significant difference in the performance of participants (Duala-Ekoko, 2012). However, since our study was carried out 2–3 years later, the results could be different since the search hits could potentially be significantly different now.

Furthermore, although we tested APIs that are used widely, there are other APIs that are proprietary (and hence, examples are not found in the Web) and that are less used (and hence, examples might be difficult to find). Our focus was to evaluate the usability of Dacite and the intrinsic API difficulty, which stems from the API's design, and therefore we minimized the presence of extrinsic factors such as the existence of code snippets from the Web and their search hit ranks.

7.7.3. External validity

The generalizability of our results could be threatened by the fact that the study only involved two APIs, despite that their domain was substantially different and that they were authored by different people. However, given that most of the discoverability hurdles stem from the exposure of certain design patterns in the API, and given that patterns are pervasive in software development, many other APIs will manifest those issues under similar contexts, and there is no apparent reason that our code completion proposals would not be useful there as well.

Another threat pertains to the programming experience, and mostly code design skills, of the participants. We believe that a developer that has implemented solutions using the design patterns will likely overcome the discoverability hurdles faster, due to the intuition gained with experience in developing similar solutions. Although we have no available data, we argue that it is reasonable to assume that the "average" programmer may not be experienced in applying design patterns. Therefore, the participants are likely to be representative in terms of software development skills, within the realm of junior programmers, given that most of them were graduate students.

8. Limitations and future work

The API designs that we addressed in Dacite were partially driven by the difficulties that were identified in previous studies (Ellis et al., 2007; Stylos and Myers, 2008; Duala-Ekoko and Robillard, 2012). The current state of our work does not cover other possible designs that are not captured in the API design patterns we have addressed. Nevertheless, with respect to object creation, we argue that we achieved a broad coverage, given that all the creation patterns described in the design patterns book (Gamma et al., 1995) were addressed. (Although we did not explicitly address the *prototype* pattern, it can be handled the same way as the *abstract factory* pattern, while Java libraries already provide some support for it through the `java.lang.Cloneable` interface and object clones.)

Dacite can be extended with additional annotations by defining new annotation types and extending the annotation processors. Given that there is no obvious reason for new design patterns to interfere with existing ones, the extensions would be purely incremental. Additional support for other design elements would result in having other types code completion proposals that would coexist with the current ones. However, in our current implementation we did not consider a plugin-based solution as a structured and black-box infrastructure for third-party extensibility.

Adopting design annotations in API development practice comes at the expense that API designers must write the annotations. The task of including the relevant annotations is essentially a form of structured and program-processable documentation. However, in contrast to regular documentation, documenting in this manner is safer given that the annotation processor performs verifications that reduce the chance of inconsistencies that are

more prone to occur with regular textual documentation. We argue that the annotations we describe here constitute a lightweight mechanism that does not require significant development effort or skills. The main drawback regarding maintenance is the risk of inconsistency with the underlying code (e.g., a missing annotation on a factory), a problem that is generally present in documentation maintenance as well.

The structured information provided by the annotations is useful for automating processes that inspect the API (such as code completion, which is the focus of this work), or other inference processes for different purposes, such as generating documentation containing more detail with respect to the API design. The enrichment of the API with this form of precise documentation adds little overhead, while it opens possibilities for automated processes to be carried out externally. A field study of API learning concluded that the main obstacles faced by users pertain to documentation (Robillard and Deline, 2011). Currently, we have only integrated design annotations with IDE code completion mechanisms, using existing forms of documentation in the proposals' description tooltips. However, design annotations can be easily integrated into documentation generation tools (e.g., Javadoc), to insure that the resulting documentation includes the information embodied in the annotations. Although we did not implement such a documentation generator, this could be achieved in a straightforward way.

9. Conclusions

The approach taken in this work improves API discoverability with a novel technique based on augmenting the API implementation code with design annotations. The main advantage of what we provide over existing approaches is to enable API users to be assisted in the IDE with respect to the discovery of API elements, without relying on code examples, corpora or alternative forms of documentation exploration. In our approach, API designers are in control of making explicit the relations between API elements. The Dacite implementation stands as a proof of concept for the feasibility of our approach, and our user study provides evidence that it is usable and effective from the API user's standpoint.

Acknowledgments

This work was partly carried out while the first author was a visiting faculty at Carnegie Mellon University, sponsored by the Carnegie Mellon Portugal Program. Funding for this research comes also from NSF grants IIS-1116724 and IIS-1314356. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of Carnegie Mellon Portugal Program or the National Science Foundation. We would like to thank Martin Robillard for providing us the study materials used in Duala-Ekoko and Robillard (2012), Robert Kraut for his advice regarding our statistical analysis, and Andrew Faulring, YoungSeok Yoon, and Stephen Oney, for their help with the Dacite pilot studies. Finally, we thank the anonymous reviewers for their valuable comments and suggestions to improve earlier versions of this paper.

References

Bloch, J., 2006. 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. In: OOPSLA '06, pp. 506–507. doi:10.1145/1176617.1176622. ACM, New York, NY, USA

Bloch, J., 2008. Effective Java (2nd Edition) (The Java Series). Prentice Hall PTR.

Bruch, M., Monperrus, M., Mezini, M., 2009. Learning from examples to improve code completion systems. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. In: ESEC/FSE '09, pp. 213–222. doi:10.1145/1595696.1595728.

Cwalina, K., Abrams, B., 2008. Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd edition Addison-Wesley Professional.

Dekel, U., Herbsleb, J.D., 2009. Improving API documentation usability with knowledge pushing. In: Proceedings of the 31st International Conference on Software Engineering. In: ICSE '09, pp. 320–330.

Duala-Ekoko, E., 2012. Using structural relationships to facilitate API learning Ph.d. thesis. McGill University, Montreal, Que., Canada, Canada.

Duala-Ekoko, E., Robillard, M.P., 2011. Using structure-based recommendations to facilitate discoverability in APIs. In: Proceedings of the 25th European Conference on Object-oriented Programming. In: ECOOP'11, pp. 79–104. http://dl.acm.org/citation.cfm?id=2032497.2032505. Berlin, Heidelberg

Duala-Ekoko, E., Robillard, M.P., 2012. Asking and answering questions about unfamiliar APIs: An exploratory study. In: Proceedings of the 34th International Conference on Software Engineering. In: ICSE '12, pp. 266–276. URL http://dl.acm.org/citation.cfm?id=2337223.2337255.

Eisenberg, D.S., Stylos, J., Faulring, A., Myers, B.A., 2010. Using association metrics to help users navigate API documentation. In: Proceedings of the IEEE Symposium of Visual Languages and Human-Centric Computing. In: VL/HCC '10, pp. 23–30.

Ellis, B., Stylos, J., Myers, B., 2007. The factory pattern in API design: A usability evaluation. In: Proceedings of the 29th International Conference on Software Engineering. In: ICSE '07, pp. 302–312. doi:10.1109/ICSE.2007.85.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Holmes, R., Murphy, G.C., 2005. Using structural context to recommend source code examples. In: Proceedings of the 27th International Conference on Software Engineering. In: ICSE '05, pp. 117–125. doi:10.1145/1062455.1062491.

JetBrains, 2014. Intellij IDEA IDE. http://www.jetbrains.com/idea.

Little, G., Miller, R.C., 2007. Keyword programming in java. In: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering. In: ASE '07. ACM, New York, NY, USA, pp. 84–93. doi:10.1145/1321631.1321646.

Long, F., Wang, X., Cai, Y., 2009. API hyperlinking via structural overlap. In: Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. In: ESEC/FSE '09, pp. 203–212. doi:10.1145/1595696.1595727.

Mandelin, D., Xu, L., Bodík, R., Kimelman, D., 2005. Jungloid mining: Helping to navigate the API jungle. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. In: PLDI '05, pp. 48–61. doi:10.1145/1065010.1065018.

Mooty, M., Faulring, A., Stylos, J., Myers, B.A., 2010. Calcite: Completing code completion for constructors using crowds. In: Proceedings of the IEEE Symposium of Visual Languages and Human-Centric Computing. In: VL/HCC '10, pp. 15–22.

Myers, B.A., Stylos, J., 2016. Improving API usability. Commun. ACM 59 (6), 62–69. doi:10.1145/2896587.

Omar, C., Yoon, Y., LaToza, T.D., Myers, B.A., 2012. Active code completion. In: Proceedings of the 34th International Conference on Software Engineering. In: ICSE '12, pp. 859–869. URL http://dl.acm.org/citation.cfm?id=2337223.2337324.

Piccioni, M., Furia, C.A., Meyer, B., 2013. An empirical study of API usability. In: Proceedings of the ACM/IEEE International Symposium on Empireical Software Engineering and Measurement, pp. 5–14.

Prechelt, L., Unger, B., Tichy, W.F., Brossler, P., Votta, L.G., 2001. A controlled experiment in maintenance: comparing design patterns to simpler solutions,. IEEE Trans. Softw. Eng. 27 (12), 1134–1144.

Prechelt, L., Unger-Lamprecht, B., Philippsen, M., Tichy, W.F., 2002. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance,. IEEE Trans. Softw. Eng. 28 (6), 595–606.

Rama, G.M., Kak, A., 2015. Some structural measures of API usability. Software 45 (1), 75–110. doi:10.1002/spe.2215.

Robillard, M.P., 2005. Automatic generation of suggestions for program investigation. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering. In: ESEC/FSE-13, pp. 11–20. doi:10.1145/1081706.1081711.

Robillard, M.P., Bodden, E., Kawrykow, D., Mezini, M., Ratchford, T., 2013. Automated API property inference techniques. IEEE Trans. Softw. Eng. 39 (5), 613–637. doi:10.1109/TSE.2012.63.

Robillard, M.P., Deline, R., 2011. A field study of API learning obstacles. Empir. Softw. Eng. 16 (6), 703–732. doi:10.1007/s10664-010-9150-8.

Santos, A. L., Prendi, G., Sousa, H., Ribeiro, R.,. Stepwise API usage assistance using n-gram language models. Journal of Systems and Software (to appear). doi:http://dx.doi.org/10.1016/j.jss.2016.06.063.

Saul, Z.M., Filkov, V., Devanbu, P., Bird, C., 2007. Recommending random walks. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. In: ESEC-FSE '07, pp. 15–24. doi:10.1145/1287624.1287629.

Scaffidi, C., Shaw, M., Myers, B., 2005. Estimating the numbers of end users and end user programmers. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing. In: VL/HCC '05, pp. 207–214. doi:10.1109/VLHCC.2005.34.

Scheller, T., Kühn, E., 2015. Automated measurement of API usability: the API concepts framework. Inf. Softw. Technol. 61, 145–162. http://dx.doi.org/10.1016/j.infsof.2015.01.009. http://www.sciencedirect.com/science/article/pii/S0950584915000178.

Siegmund, J., Kästner, C., Liebig, J., Apel, S., Hanenberg, S., 2014. Measuring and modeling programming experience. Empir. Softw. Eng. 19 (5), 1299–1334. doi:10.1007/s10664-013-9286-4.

Sillito, J., Murphy, G.C., De Volder, K., 2008. Asking and answering questions during a programming change task. IEEE Trans. Softw. Eng. 34 (4), 434–451. doi:10.1109/TSE.2008.26.

Stylos, J., Faulring, A., Yang, Z., Myers, B.A., 2009. Improving API documentation using API usage information. In: Proceedings of the IEEE Symposium of Visual Languages and Human-Centric Computing. In: VL/HCC '09, pp. 119–126.

Stylos, J., Myers, B.A., 2008. The implications of method placement on API learnability. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. In: SIGSOFT '08/FSE-16, pp. 105–112. doi:10.1145/1453101.1453117.

Thummalapenta, S., Xie, T., 2007. PARSEWeb: A programmer assistant for reusing open source code on the web. In: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering. In: ASE '07. ACM, New York, NY, USA, pp. 204–213. doi:10.1145/1321631.1321663.

Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S., 2006. Design pattern detection using similarity scoring,. IEEE Trans. Softw. Eng. 32 (11), 896–909. doi:10.1109/TSE.2006.112.

Tulach, J., 2012. Practical API Design: Confessions of a Java Framework Architect. Apress.

Zhong, H., Xie, T., Zhang, L., Pei, J., Mei, H., 2009. MAPO: Mining and recommending API usage patterns. In: Proceedings of the 23rd European Conference on Object-Oriented Programming. In: ECOOP '09, pp. 318–343. doi:10.1007/978-3-642-03013-0_15.

André L. Santos is an Assistant Professor at the Department of Information Science and Technology, University Institute of Lisbon (ISCTE-IUL). He received a PhD in Informatics in 2009 from the University of Lisbon. His research interests are related to program comprehension, API usability, object-oriented frameworks, domain-specific languages, and integrated development environments.

Brad A. Myers is a professor in the Human-Computer Interaction Institute in Carnegie Mellon University's School of Computer Science. His research interests include programming environments, programming-language design, and user interfaces. Myers received a PhD in computer science from the University of Toronto. He is a Fellow of IEEE and ACM and belongs to the IEEE Computer Society and the ACM Special Interest Group on Computer-Human Interaction.