Perceived Self-Efficacy and APIs

John M. Daughtry III

Applied Research Laboratory
The Pennsylvania State University
daughtry@psu.edu

John M. Carroll

College of Information Sciences and Tech.
The Pennsylvania State University
jcarroll@ist.psu.edu

Keywords: POP-II. B. Program Comprehension, Scenario-Based Design; POP-IV. A. Object-Oriented Design; POP-I.A. Group Dynamics; POP-V.B. Research Methodology

Abstract

Application program interface (API) use and design is critical, non-optional, and cross-cutting in the construction of modern software systems. However, only recently has the explicit study of API design process and API designs been initiated from the perspective of usability, and little is known with respect to how various forms of information about an API aids programmers in the use of an API. In this paper, we present findings from our exploration of perceived self-efficacy (PSE) for API use. First, we describe the development of a novel PSE instrument that focuses on the task of using an API. Second, we evaluate the validity and sensitivity of the instrument with respect to changes in the information given professional programmers about an API. To accomplish this goal, we articulate and utilize two complementary forms of API documentation grounded in scenario-based design. Through this work, we demonstrate the validity of the evaluation tool and raise questions about the perceived value developers place on information about an API and its intended use.

1. Introduction

Modular programming is ubiquitous in modern software application construction. The benefits we seek of modularity are technical, psychological, and organizational in nature (Parnas 1972). Using modules affords reuse, which in turn reduces duplication. Thus, there is less code to maintain. Modularity also reduces code into small pieces that can be more easily understood by a developer given humanity's constrained capabilities for cognition. One can imagine the difficulties in trying to read and edit the source code of any modern software system were it contained in a single structure. Finally, modularity affords the breakdown of work in teams by allowing groups and individuals to focus on discrete pieces of a system.

Because of the prominence of modularity, the use of any popular modern programming language necessitates the use of APIs (Stylos and Myers 2007). The centrality of APIs within the context of programming is of growing importance (Stylos 2009). For example, at a minimum, Java programmers make use of the Java platform SDK (JDK), and C# programmers make use of the .NET Framework. Even to output "hello world" requires utilization of the APIs provided with the language. Note that we use the term API broadly, reflecting the programming vernacular (e.g., de Souza, Redmiles, Cheng, Millen, and Patterson 2004; Bloch 2005). This definition implies that every module has an API (the interface through which the module is invoked), which is in contrast to stricter definitions that define APIs in terms of interchanges between applications (e.g., Software Engineering Institute 2008). However, it more closely aligns with practice, as reflected in Bloch's argument that "… if you program, you are an API designer, whether you know it or not, because good code is modular, and those inter-modular boundaries are effectively APIs" (2005).

In the wild, API design perfection is unattainable (Bloch 2005). Indeed, interface design is about navigating a design space of trade-offs as opposed to finding the optimal design (Carroll 2000). For example, as de Souza and colleagues describe, information hiding has significant organizational communication drawbacks, despite the technical benefits (2004).

Given the centrality and rising importance of APIs in professional programming, our research agenda is to expose the psychological aspects of API design and use as a leverage point for impacting the practice of programming via empirical data that supports interaction design for APIs themselves and the tools used by API consumers.

1.1. API Usability

Extant work in API usability analysis focuses on traditional usability laboratory studies (e.g., Clarke 2004). In essence, the researcher sets out to analyze the use of a particular API to uncover flaws in that API. Another approach is to seek out general guidelines that can be applied to APIs. Stylos and Clarke (2007) used this approach to study the usability implications of constructor parameters versus setters in the general case. Most recently, researchers have sought out more effective methods for API usability analysis, since the lab study approach is extremely time consuming and resource intensive (Farooq and Zirkler 2010).

Each of these approaches holds value and has provided unique empirical understanding of API design and use. Yet, in large part, these approaches fail to deliver a richer theoretical understanding of the psychology of programming. Rather than explaining the rich social, improvisational, and psychological aspects of API use, they explain API designs, decisions, and their trade-offs.

Rather than focussing wholly on the artefact itself, the scientific exploration of API design and usability needs to reach beyond the artefact and explain the relationship between the social psychology of programming and the API, as well as the relationship between the computational sciences and the API. How does the API and information about the API shape, constrain, limit, enable, or undermine the psychological aspects of programming? How does the underlying technology (e.g., programming languages, paradigms, algorithms) shape, constrain, limit, enable, or undermine the API?

When thinking about API evaluation, one must be careful not to lose sight of the larger picture of reuse within the design process. We are not only designing an API that is easy to use, but also supporting the reuse of existing code in the larger design of a system. This is particularly challenging for a usability lab study, where programmers are studied working on small- to medium-sized singular tasks for which many API issues are set a priori. Of course, real programmers do not work alone and can often choose to reformulate their problem space, for example, rejecting one API for another. These decisions are not purely technical in nature. Cockburn argues that software development is in part an economic endeavour (2004). Indeed, one may choose to use the Google Web Toolkit (GWT) instead of the Dojo Framework not because it is better, but because it is perceived as being a more marketable experience to have on one's résumé. While we must not overly focus on such aspects of API design at the expense of usability, the social nature of software engineering is extremely salient.

1.2. Perceived Self-Efficacy

Perceived Self-Efficacy (PSE) is a construct from Social Cognitive Theory (Bandura 1997). It "is the belief in one's capabilities to organize and execute the courses of action required to manage prospective situations" (Zimmerman, 1995, p. 203). Examples of self-efficacy would be one's perceived ability to lift weights or one's perceived ability to pass a math test. PSE has value because it is found to correlate highly with actual task performance across a wide range of domains (Bandura, 1997).

The basic premise of PSE is that it is based on stable and grounded beliefs about one's capabilities. Over time, people develop and refine PSE with respect to the tasks in which they engage. For example, one may over time develop a PSE with respect to writing academic research papers. PSE is the stable and grounded construct that represents the overarching belief about one's ability to write such papers.

Bandura described a technique for developing psychological instruments exposing this underlying construct that has been utilized in many fields such as education and computer use (1997). Because PSE is highly correlated with performance, we believe that it may have value in the study of API design and usability. Unlike lab studies and peer reviews, PSE places small time requirements on the participants. In an API lab study, participants have to use the API. In a peer review, participants have to take the time to discuss an API with peers. PSE, however, is a psychological instrument that only requires the participant to reflect on an API and fill out a questionnaire containing Likert scale items. Further, PSE exposes a deep-rooted belief as opposed to focussing solely on the pragmatic outcomes. Thus, in addition to providing guidance, it also describes the impact of interventions at a psychological level.

Given that PSE is based on stable and grounded beliefs, after one is able to expose PSE for a given activity, the degree to which the construct is impacted becomes an important research question. For example, do you believe you can survive a bear attack in the woods? If you have a gun, do you believe you can survive the attack? What about if you have a bazooka, a gun, and some time to read a book on killing bears? Our perceived self-efficacy for accomplishing specific goals is moderated by the tools we have at our disposal. Certainly, in cases such as this toy example, our PSE is heavily impacted by the tools and environment. However, cases of PSE grounded in everyday activities are less clear. Does having a calculator impact PSE for passing a math test? Does having access to an IDE (vs. command-line) impact a programmer's PSE for debugging a problem in a large system? In these real-world cases, the degree to which PSE is impacted by tools is less clear. If we can expose programmers' PSE for API use, to what extent is the construct impacted by the tools and information at their disposal?

1.3. API Documentation

Given that no API can be perfect (or at least most), API documentation is critical to the success of projects (Bloch 2005, pg. 18). Indeed, more research into code documentation has been explicitly called for from within the software engineering community (e.g., Parnas 1994; 1998).

Existing work in API documentation focuses on documentation tools as opposed to information needs. Apatite and Jadeite are two recent examples of such tools. Apatite refocuses the structure of information around the notions of importance (i.e. – which parts of the API are most often used) and the relationship between API elements in the context of use (Eisenberg, Stylos, and Myers 2010). The interaction design in Apatite contributes to API documentation tools on two fronts. First, it shows the efficacy of using font size to indicate importance of API elements based on usage data. If you open the `java.io` package in Java, for example, `File` will be shown as an important class within that package. This helps programmers focus their attention on the most often used elements of the API. It also shows the utility of navigating by association. For example, if you open the `java.io` package, you find that `read` is an important method. By selecting that method, you find that `FileInputStream` is an important class with respect to that method.

Jadeite incorporates elements of Apatite, such as basing font size on usage, but adds in placeholder and object instantiation capabilities (Stylos, Faulring, Yang, and Myers 2009). Placeholders provide Java developers with a mechanism for communicating about API elements they expect to see as opposed to just what is there. It also scours existing source code to extract how classes are instantiated as objects. Thus, when a user uses Jadeite documentation for any given class, it tries to give you an example of how that class it created. The researchers found that developers performed three times faster with Jadeite than with traditional Java documentation. Calcite (Mooty, Faulring, Stylos, and Myers 2010) is an integrated development environment (IDE) plug-in related to Jadeite that pulls the object instantiation code directly into your code when you start to use an object (while Jadeite only puts that information in the documentation).

The approach used to evaluate the utility of Jadeite to show a significant performance improvement when using the tool is limited because it imposes significant artificial scaffolding for the programmer. Specifically, many API users have to begin with the selection of which API to use. For example, if you want to implement logging in Java, you can utilize Java logging, Log4J, or Commons Logging. One can also utilize Simple Logging Façade for Java (SLF4J) for dependency injection to separate the logging calls from the specific implementation. These are complex decisions that cannot simply be navigated away via documentation structure. However, these aspects of the problem-space are important notions when it comes to the formulation and development of a programmer's PSE.

1.4. Research Questions

To explore the role of PSE in API use, we focus on the following research questions. If we cannot operationalize the notion of a programmer's PSE for using an API, then the idea has little (if any) pragmatic value. Further, before we can make use of the construct, we need to develop a firm understanding of how the construct is impacted by the tools and information at hand. Note that in this paper we focus on the information at hand as opposed to the tools at hand.

RQ1: Can we identify, extract, and elucidate a professional programmer's PSE for using an API?

RQ2: To what extent does the information provided to professional programmers about an API impact their PSE for using that API?

2. Scale Development

Developing a perceived self-efficacy scale requires three steps (Bandura 1997). First, one must define the task, in particular the various aspects of the task. Having an analysis of the activity helps you identify the critical capabilities one would need in order to be able to do the activity. Second, one must construct statements about the task in the form of "you can do X". Third, one must incorporate obstacles to the task such as "even when you are tired". When presented with obstacles, questions become more salient, resulting in an accurate assessment of one's own capability. For example, if you ask someone if he can stick to a diet, even when he is tired, he will give a more accurate response than if you only ask him about his ability to stick to a diet.

To construct our scale, we iteratively built and refined an analysis of the core activities of API use, drawing upon Daughtry's work experience as a professional software engineer as well as extant literature on software design. We began with Schneiderman's programming task breakdown (1980). The tasks articulated by Schneiderman relevant to API use are learning, designing, composing (writing the code), comprehending, testing, and debugging. In order to use an API, programmers must learn how to use the API, design the system to use the API, write the code that calls the API, comprehend the code that uses the API, test the usage of the API, and debug the code that uses the API. Similarly, Fischer, Henninger, and Redmiles described reuse as being a cyclical process of location, comprehension, and creation (1991, p. 319). However, these task breakdowns fail to take into account the social side of programming activities.

The field of design rationale offers a particularly useful glimpse into the social process of design. From this perspective, design is a cyclical process of task analysis and artefact envisionment (Carroll and Rosson 2003). Namely, designers make claims about a design, reason around those claims, and then build the system. These claims may be implicit or explicit. The field of design rationale has sought to leverage explicit rationale; but with respect to our purpose, both forms of rationale are relevant. When designers go through the process of task analysis (whether it is grounded in formal task analysis methods or simply the thought process of what might work best), they are chiefly concerned with what Alexander, Ishikawa, Silverstein, Jacobson, Fiksdahl-King, and Angel refer to as "goodness of fit" (1977). Specifically, does this design fit our problem, and how does it compare to other alternatives?

However, design is embedded in a socio-political process (Cockburn 2004). Thus, rationale must be defensible not only to rational team members, but also to irrational team members and time-constrained, budget-conscious managers. Further, one may make decisions based on self-interest as opposed to what is best for the system, other team members, or the customer.

Using these conceptualizations of design, along with reviews from professional programmers and other researchers, we developed the PSE instrument given in **Error! Reference source not found.**.

This instrument takes into account the design, implementation, testing, and debugging of API uses. At the same time, it also

No.	Scale Item
1	I can determine if this component is a good fit for use in a system design, even if the system requirements are not completely clear.
2	I can identify 5 or more well-suited applications for this component even if I were only given 1 minute to identify them.
3	I can recognize when this component has been wrongly chosen for use in a system design, even without knowledge of the entire system architecture.
4	I can identify an application where this component would fit (but be a poor fit) even if I had to identify such an application immediately.
5	I can defend my choice to use this component in my solution to other software developers, even if they are sceptical.
6	I can defend my choice to use this component in my solution to higher-level management, even if there is pressure to go with another component that already exists in the company product line.
7	I can make a strong case to a development team that the component should not be used, even if the team wants to use it because it is a hot technology (i.e., a buzzword that looks good on a résumé).
8	I can defend someone else's choice not to use this component in a solution to a technologically oriented customer, even if I had not discussed the decision with that developer.
9	I can use this component in building a system without having access to anyone who has used it before.
10	I can use this component in building a system without having access to any documentation other than the API (e.g., using no examples posted on the internet).
11	I can build a prototype system to demonstrate the power of the component to management in fewer than 30 minutes.
12	I can successfully augment an existing system I wrote to use this component instead of another mechanism even if it requires architectural refactoring in the system.
13	I can write coded tests against uses of this component that ensure the implementation will exceed performance needs of the entire system, even if I was extremely tired.
14	I can write coded tests against uses of this component that ensure that a system installation is configured correctly, even if the configuration is slightly different for each installation.
15	I can write system thread-level test procedures for a system that uses this component that validates every functional behavior of this component, even if a unit test procedure is not available.
16	I can write test plans for negatively testing behaviors of this component, even if the source code is not available.
17	Given a bug in the use of this component in a system I wrote, and knowing the effects of the bug, I could isolate the root cause of the problem even if the only things available were the source code and a running system. A debugger is not available, and there is no way to add new logging calls such as `System.out.println("test line 1: did it make it here?");`.
18	Given a known bug in the use of this component in a system, where the root cause has been isolated, I could fix the problem, even if I didn't completely understand the context of the component being used.
19	Having just used this component to build a larger system, I could identify most of the bugs caused by incorrect usage of this component before releasing the code to a testing group, even under extremely heavy scheduling constraints.
20	I could successfully fix a problem in the usage of this component within a larger system I didn't write, even if I had to make the fix on-site without access to a test group.

Table 1 - Perceived Self-Efficacy Scale for API Use

takes into account the social nature of programming. Because of space limitations, we do not include the Likert scales associated with each item. Each item has an associated Likert scale ranging from 1 to 9. Item 1 was labelled "Cannot do at all"; item 5 was labelled "Moderately certain can do"; and item 9 was labelled "Certain can do".

Questions 1-8 cover the design activity (determining goodness of fit), taking into account the social aspects such as collaboration with team members, overcoming self-interest, and hierarchy. Items 9-12 deal with the use of the API within code. Item 9 takes into account the social component of implementation (having access to others who already know how to use the API). Item 12 takes into account the problem of real-world programming, whereby code is not always written from scratch. Items 13-16 deal with testing an API. This portion takes into account low-level unit testing through system-level testing, while at the same time taking into account the notion of propriety (source code may not be available) and the real-world hardships of testing (systems can be configured in many different ways). Items 17-20 address the debugging task, taking into account problems one is likely to encounter on real-world systems such as a limited schedule.

Each of the 20 items in our PSE scale includes obstacles, as previously discussed. When creating a PSE scale, it is important that the obstacles are realistic and challenging. Specifically, they should be something that one might actually encounter on the job and indeed pose a challenge for the programmer. After defining our scale, we conducted a sandbox pilot study where professional developers completed the scale in a think-aloud manner. The scale presented in **Error! Reference source not found.** is the final instrument, after revisions were made when issues surfaced during the sandbox pilot.

3. Code Vignettes and Claims Analysis

Having designed a scale for measuring a programmer's PSE for using an API, we needed a mechanism for evaluating the validity of the scale (RQ1) and evaluating the impact information could have on the exposed construct (RQ2). The validation of a PSE scale involves the use of internal validity and factor analysis. We concluded that the same study could be used to address both research questions. However, we needed varying forms of API information in order to do this evaluation.

Scenarios and claims are the two fundamental components of scenario-based design (Carroll 2000). Scenarios are stories about people and their activities, while claims are an enumeration of the causal factors and relations that are left implicit in scenario narratives. With respect to programming, scenarios are reified as code examples with notes detailing the purpose of the example code. For the balance of this paper, we refer to such examples as code vignettes.

Example code and statements about the design are a natural representation when talking about code and software design. Indeed, one can find examples of this form in almost every practitioner-oriented discussion on API design (e.g., Bloch 2001; Bloch 2005; Cwalina and Abrams 2008; Tulach 2008; Pugh 2007) and other texts (e.g., Gamma, Helm, Johnson, and Vlissides 1994). Thus, we need to clarify what we mean by code vignettes and claims and distinguish it from other forms of example code and rationale. In each of the sources listed above, we find example code with lightweight design rationale expressed in the form of claims. However, these uses often differ in significant ways from what we mean by code vignettes and claims. With respect to the example code, they often show example *design* code as opposed to *uses* of that design. And, with respect to claims, they often describe the design *intention* of the design as opposed to describing the ramifications seen in a usage.

Figure 1 gives an example of design and intention information as opposed to what we mean by code vignettes and claims. First, let us consider the design against the code vignette. While the design conveys information about the interface, from which a programmer can derive the use, the code vignette is explicit in describing a use of the interface. This is beneficial because it affords a quicker translation to the activity of using the interface. Indeed, a programmer can copy-paste his way to using the interface with the code vignette. We know that programmers are opportunistic, debugging code into existence (e.g., Rosson and Carroll 1996; Brandt, Guo, Lewenstein, Dontcheva, and Klemmer 2009). Rather than using an API directly, they seek out uses of the API and adapt it to their needs. Turning to the justification for the design, the design example does not provide the salience the vignette example provides. Specifically, the analysis is loosely grounded on broad statements about the design as opposed to being lightweight analytical evidence. The design example can only make broad generalizations such as the interface being easy to code against while the vignette example has the grounding to make specific statements about why the interface is easy to code against. Within the publications mentioned above, and in other corpus (in print and online), code and rationale takes many forms that reflect vignettes and claims to varying degrees.

Design and Intention Information	**Code Vignette and Claims Information**
`public interface Collection {`	A programmer seeks to iterate over a collection.
`  …`	
`  public Iterable iterator();`	`for (Iterator i = c.iterator(); i.hasNext();)`
`  public boolean hasNext ();`	`  System.out.println(i.next());`
`  public Object next();`	
`}`	
This design is violates the bean custom of `get` and `set`. However, it makes for code that is easier to write and easier to read.	+ The programmer can express a loop on one line, even if the Collection name is a long expression. - The programmer cannot rely on the bean getter and setter convention when learning the API.

Figure 1 - A design and its intent vs. vignettes and claims

(Design and rationale adapted from Sun 1997)

Returning to Carroll's articulation of scenarios and claims, we find reasoning as to why object models and scenarios are particularly compatible (2000, p. 232). Known tasks are defined (at times explicitly) and translated to envisionments of use. For example, one may envision an API and its use via unit tests (if utilizing test-driven development). Alternatively, one may envision an API and its use via the Unified Modelling Language (if using model-driven development). This activity, and the associated reasoning via use (i.e., claims) helps to evoke, identify, and refine the API design.

We believe that code vignettes and claims are valuable information for programmers. As discussed above, vignettes support the opportunistic behaviour many programmers exhibit. Specific vignettes have been utilized in API documentation before. For example, some designers of API's have incorporated these into their API documentation. As discussed above, Jadeite and other tools have supported specific vignettes for object instantiation. Thus, it seems that the inclusion of vignettes should have significant value for programmers. We also believe that claims about an API use should have value to an API user by clarifying the designer's reasoning and thought process. For example, the designers of the Collections API for Java were compelled to document the rationale behind many decisions they made in the form of a design rationale document (Sun Microsystems 1997). Presumably, they went to such effort in order to fend off change requests, the questioning of their decisions by those who may not be able to infer the rationale from the design, and to aid other API designers by providing explicit reasoning about a particular API. Since all programmers are API designers and consumers, these are valid justifications for the documentation of rationale in all API designs. We do not know why the Sun designers chose to write a separate document instead of including the rationale in the API documentation itself. It makes it less accessible, locatable, and maintainable. Indeed, it was Sun that popularized the use of API documentation within source code via JavaDoc.

4. Methodology

Having contrasting forms of API information, we derive the following three hypotheses from our overarching research questions:

H1: Our new instrument effectively measures programmers' PSE with respect to API use.

H2: Inclusion of scenarios of use in an object-oriented API specification increases the user's PSE for using the API.

H3: Inclusion of claims in an object-oriented API specification can increase the user's PSE for using the API.

To test our hypotheses, we used the experimental design given in Table 2. To test H1, we needed a number of professional programmers to evaluate an API with our scale. To test H2, we needed a control group to evaluate an API with basic documentation (no vignettes or claims). We also needed a test group to evaluate an API with documentation that included vignettes. To test H3, we needed an additional test group to evaluate an API with documentation that included claims.

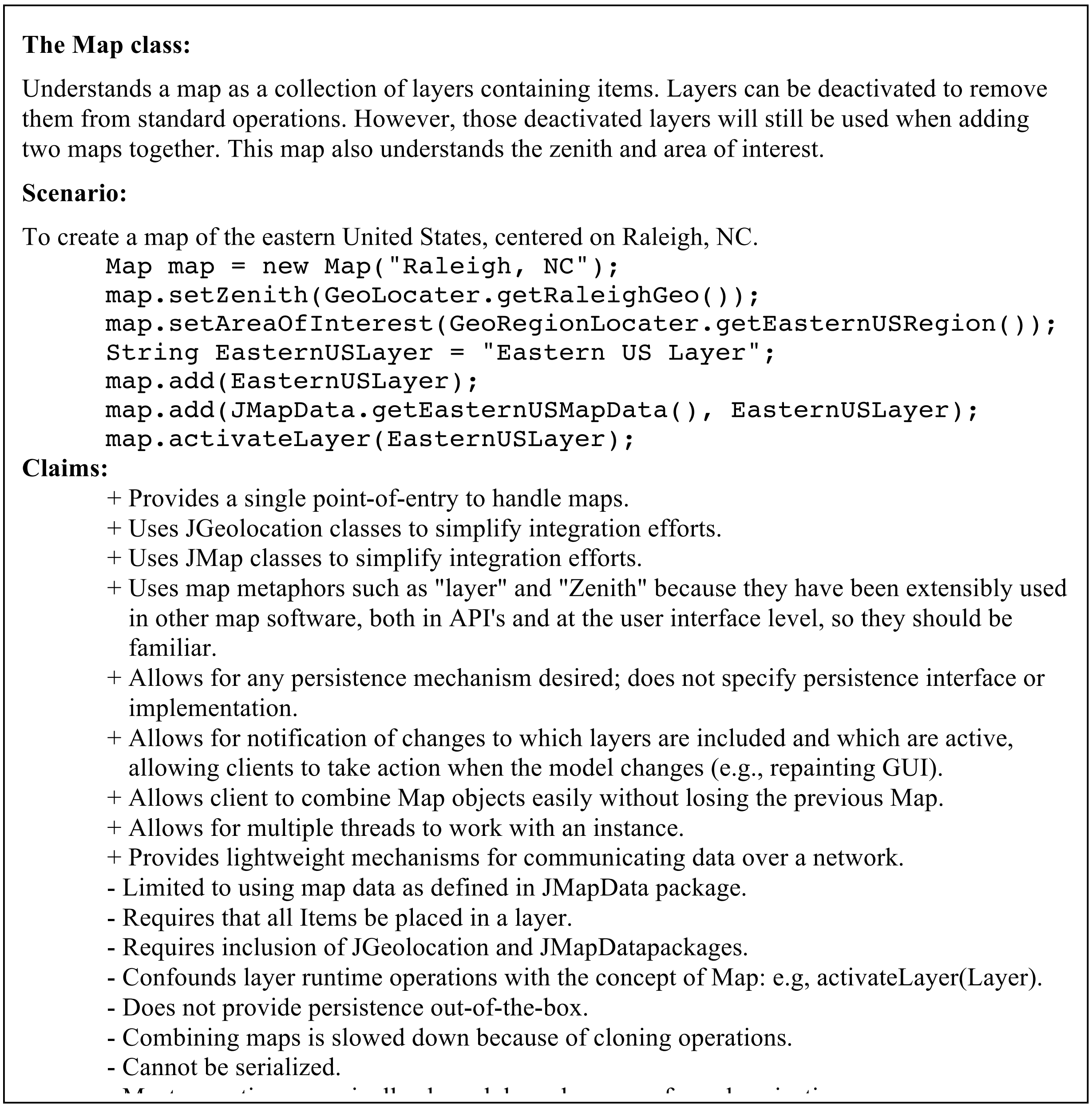
	Initial Data	**Assignment**	**Treatment**	**Observation**
Group 1	Reported number of years experience designing and developing software	Quasi-randomly assigned to groups based on experience	Presented API with basic documentation	Completed questionnaire (perceived self-efficacy)
Group 2			Presented API with basic documentation and the scenario	
Group 3			Presented API with basic documentation, the scenario, and claims	

Table 2 - Experimental Design

For subjects, we needed people who would know how to develop software and read an API. Therefore, we had two options for subjects: undergraduates or experienced professionals. While using undergraduates would allow us to obtain subjects more easily, we chose to use experienced professionals because we preferred ecological validity over power. Therefore, we chose a target of 20 subjects per group. To obtain experienced professional software developers as subjects, we leveraged existing professional contacts. In addition, we sought to keep the study as short as possible. With a limited and distributed sample pool from which to draw, we needed a very high participation and completion rate. The need for a short study was limiting in that we could not use multiple APIs.

Given that the potential subject pool would consist of primarily Java developers, we utilized JavaDoc as the documentation format. Figure 2 gives the top-level API documentation as given to group 3. We stripped the claims for group 2. We stripped the claims and scenario for group 1. One may assume from the depicted documentation that there was a substantial difference between the amount of documentation given to each group. Certainly, some groups were given more information than other groups. However, there was other basic information included in the API specification that is not included in the figure because of space limitations. The complete specification for each group (were it printed out from online) is 5.5 pages for the control group, 6 pages for the first test group, and 6.25 pages for the second test group.

Subjects were recruited via email and sessions were conducted over the web. They reported their experience and were assigned to a group using an algorithm that controlled for experience. Once assigned to a group, they were all given the same instructions. Specifically, they were told to imagine that they had just been placed on a development team tasked with building a large system. Their first task was to evaluate potential software components and frameworks for use within the system. They were then given the API and asked to complete the questions at the bottom of the page with respect to that API. There was no time limit imposed by the study. These questions were the PSE scale.



The Map class:

Understands a map as a collection of layers containing items. Layers can be deactivated to remove them from standard operations. However, those deactivated layers will still be used when adding two maps together. This map also understands the zenith and area of interest.

Scenario:

To create a map of the eastern United States, centered on Raleigh, NC.

```
      Map map = new Map("Raleigh, NC");
      map.setZenith(GeoLocater.getRaleighGeo());
      map.setAreaOfInterest(GeoRegionLocater.getEasternUSRegion());
      String EasternUSLayer = "Eastern US Layer";
      map.add(EasternUSLayer);
      map.add(JMapData.getEasternUSMapData(), EasternUSLayer);
      map.activateLayer(EasternUSLayer);
```

Claims:

+ Provides a single point-of-entry to handle maps.
+ Uses JGeolocation classes to simplify integration efforts.
+ Uses JMap classes to simplify integration efforts.
+ Uses map metaphors such as "layer" and "Zenith" because they have been extensibly used in other map software, both in API's and at the user interface level, so they should be familiar.
+ Allows for any persistence mechanism desired; does not specify persistence interface or implementation.
+ Allows for notification of changes to which layers are included and which are active, allowing clients to take action when the model changes (e.g., repainting GUI).
+ Allows client to combine Map objects easily without losing the previous Map.
+ Allows for multiple threads to work with an instance.
+ Provides lightweight mechanisms for communicating data over a network.

- Limited to using map data as defined in JMapData package.
- Requires that all Items be placed in a layer.
- Requires inclusion of JGeolocation and JMapDatapackages.
- Confounds layer runtime operations with the concept of Map: e.g, activateLayer(Layer).
- Does not provide persistence out-of-the-box.
- Combining maps is slowed down because of cloning operations.
- Cannot be serialized.

Figure 2 - API Documentation

5. Results

We achieved our target of 20 subjects per group with a mean experience level of 8.6 years and a standard deviation of 5.8. We were able to control for experience successfully, with each group having participants ranging from 3 to 20 years experience as programmers and nearly identical means.

Every item on the scale had a variance greater than 1, indicating that there was some variance in the results. Factor analysis was performed on the 20 items. A scree plot (Figure 3) showed that the eigenvalues decreased more gradually after the first factor. Therefore, we ran a factor analysis for one factor. We found that one factor accounted for 37.4% of the variance. Adding more factors did not significantly change the explained variance; adding a second and third factor changed the explained total variance to 48.6% and 55.8%, respectively. In addition, removing any one item from the scale did not significantly change the variation explained by one factor, with the highest result being 38.8%

and the lowest being 36.5%. The Cronbach's alpha for the results was 0.91, which is high, and shows that the scale was internally consistent. Excluding items resulted in an alpha within the range of 0.90 – 0.91, so we kept all of the scale items.

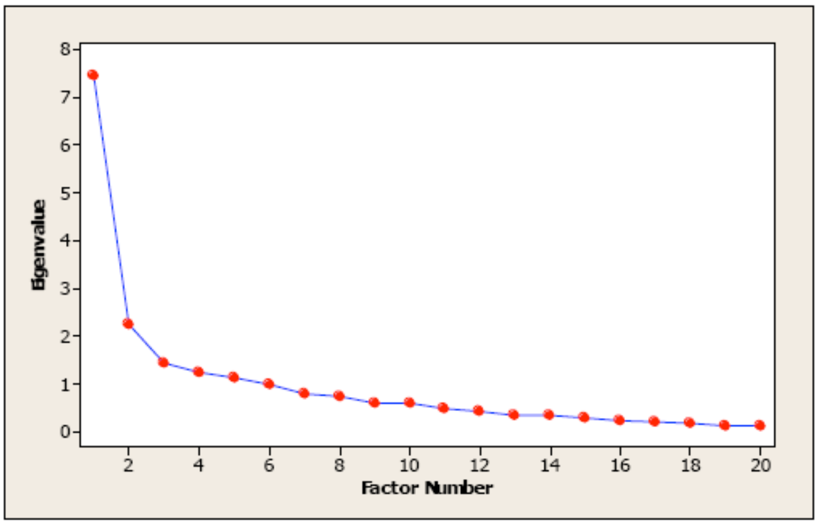


Figure 3 - Scree plot for the self-efficacy scale

We examined the probability plot of each item on the scale, and they all showed normal or nearly normal distribution. This also held for each group and there were no outliers.

We checked for equal variance and used a one-way ANOVA to compare the means of the three groups. Our F was 1.78 (which is a safe value for 3 treatments and 20 subjects). With a p-value of 0.320, there is not a statistically significant difference between the groups.

The results did not show significance, but we chose our sample size because of limitations in available subjects. Although student programmers are relatively easy to recruit, we would rather have participants that more accurately reflect the profession. This limited our sample size, so we did a power analysis. With three groups, an alpha of 0.95, a maximum difference between group means of 0.605 (between the basic documentation group and the scenario with claims group), and an overall standard deviation of 1.269, 113 subjects would be needed. This means that if the distributions we obtained in this study hold true, 113 subjects per group would be needed for the results to be statistically significant.

Finally, as data exploration, we examined the relationship between experience and self-efficacy. We computed Pearson's product-moment correlation coefficient for the entire data set and within each group. We found no evidence of a significant correlation between experience and PSE. In each group, either the correlation was too weak to be significant (e.g., -0.026 for the control group) or the correlation was not statistically significant (e.g., 0.128 for the vignettes group).

6. Discussion

In this work we had two research questions. First, we sought to develop an instrument for exposing programmers' PSE for using an API. Notably, in exposing this construct, we incorporated the social dynamics of API use within real-world systems. Second, we sought to evaluate the extent to which information (we used code vignettes and claims) impacts the exposed construct.

We found that the PSE scale we developed was able to capture programmers' PSE effectively for using an API. The very significant Cronbach's alpha (0.91) shows that the scale is internally consistent. One factor accounts for 37.4% of the variance between participants, and no other factor is significant. Thus, the construct being measured is a simple and unitary construct.

Intuitively, one might expect that over time, programmers have a higher PSE for API use. However, we saw that there was not a statistically significant correlation between self-efficacy and experience.

In the strongest case, we are only 87.2% sure there is a correlation coefficient of -0.352 between years experience and self-efficacy when scenarios are in the documentation. However, this is a weak case for a marginally weak correlation. If the scale was more dependent on experience than the API design itself, then we would have to conclude that the scale measured a programmer's general PSE for API use as opposed to measuring the programmer's PSE for a particular API. However, if we make the assumption that a programmer's PSE for API use in general goes up over time, then we must conclude that our scale is indeed measuring PSE for the API in question.

Looking only at the means of the groups, the results suggest (though insignificantly) that code vignettes may actually decrease the self-efficacy of the subjects and that claims did so to an even greater degree. Our initial presumption was that vignettes and claims would increase the usability of the API. However, upon reflection, the real target outcome of vignettes and claims is that the programmers can make more accurate judgements about the API. Thus, it is possible that our API had poor design elements, and the programmers were able to assess the API more accurately based on the additional information.

We found that we could not significantly alter the PSE results with significant documentation changes. Thus, we also know that the PSE being measured is fairly robust. Unfortunately, this robustness also means that we cannot use PSE as a way to evaluate small design decisions such as the inclusion of vignettes and claims in documentation.

Given that the exposed construct is robust, it may hold value with respect to endeavours that require a more robust construct. First, API usage performance in general may be related to our construct. Evaluating programmer performance is a long-sought goal in the study of programming. Unfortunately, a PSE instrument is easily manipulated if programmers are aware that they are being evaluated against their responses. However, if a relationship does exist, it would still provide a significant contribution to the psychology of programming. Further, its utility with respect to evaluating the development of novice over time may hold value. Although experience was not correlated with PSE in our study, we focussed on professional programmers.

Second, since it is not impacted by small design decisions, the PSE construct might be useful at a higher granularity with respect to technology. For example, could it have shown the Java platform design team that their Calendar API needed significantly more work, while the Collections API satisficed? At present, the only mechanism we have for making such decisions is a subjective analysis of importance with respect to how often the API will be used (Stylos and Myers 2007).

We only examined the relationship between PSE and the information given a programmer about an API. We did not examine the relationship between changes in an API itself and PSE or changes in the tools leveraged in API use (e.g.. Integrated Development Environments). It is possible that these may still significantly impact PSE.

We studied professional programmers who, over time, have developed expectations of API documentation. Some or all may have developed distrust for documentation, thus ignoring everything except the structure of the API (e.g.. class name, method names, and method return types). This would have led to scenarios and claims having no impact. Even if they did trust the documentation, the inclusion of the scenarios and claims may have thrown them off. This is information they are not used to seeing in documentation. Second, engineers are presumably extremely analytic. They may have over-analyzed the questions presented in the self-efficacy scale in ways that did not surface in the sandbox pilot. It is also possible that some APIs would benefit from scenarios and claims, while others do not. This study only examined one API due to constraints in study length. The API selected may have not been an API that benefits from scenarios or claims. Finally, the scenarios and claims included in the API could have been different. Perhaps we did not construct scenarios and claims that significantly contributed to the use of the artefact. Again, we were constrained due to study length.

7. Conclusion

Our analysis supported H1. Interestingly, we had no support for H2 and H3. But, our results do translate to the following significant results:

> **Result 1:** Our perceived self-efficacy scale appears to measure a single construct, which we presume to be the perceived self-efficacy of a developer in using an API.
>
> **Result 2:** Programmer's PSE for using an API is robust, seemingly unaffected by changes to documentation.
>
> **Result 3:** Experience does not play a significant role in developers' perceived self-efficacy for using APIs.

Given the robustness of the scale developed, several research opportunities arise from this work. First, the scale needs to be applied against other APIs so that we can assess whether it has utility in answering other important questions about programming. For example, one could compare the results for a notoriously poor API design (e.g., something like the Java Calendar API) and a good API design (e.g., something like the Java Collections API) to evaluate the effect that different APIs have on a programmer's PSE. Second, given the robustness of the scale, similar scales could be developed that are at a lower level. Indeed, Bandura warns that the granularity of PSE scales can heavily impact their utility. Specifically, an API use debugging PSE scale would be much more sensitive to small design changes than the broad API usability PSE scale we developed.

Finally, and perhaps most importantly, we need to develop a more thorough understanding of the role documentation plays in programming. We did not include such a test in our experiment, but it is safe to assume that had we included an API with no documentation (to include meaningful package, module, and member names), it would have received a very low score by programmers. How can one even begin to use such an API, much less ascertain things like its goodness of fit and testability? Although we have all been told (and we tell our students) that documentation is important and names should have meaning, we don't have an understanding of how these API elements support the information needs of API users.

Currently, we are seeking to develop a better understanding of the API design decision space described by Stylos and Myers (2007). In order to explore the relationship between psychological constructs and API design decisions effectively, we must be able to understand the dimensions of API design and use with respect to particular API features.

8. Acknowledgements

We would like to thank Mary Beth Rosson, Steve Haynes, Umer Farooq, Matt Peters, and Josh Gross for their advice and support in the design and execution of this work. John Carroll would also like to thank Edward M. Frymoyer for his financial support. We would also like to thank Dwight Berry for his comments on a prior draft.

9. References

Alexander C., Ishikawa S., Silverstein M., Jacobson M., Fiksdahl-King I., and Angel S. (1977). *A Pattern Language*. New York, NY: Oxford University Press.

Bandura, A. (1997). *Self-efficacy: The exercise of control*. W.H. Freeman and Company.

Bloch, J. (2001). *Effective Java*. Second Edition. Upper Saddle River, NJ: Addison-Wesley.

Bloch, J. (2005). How to Design a Good API and Why it Matters. keynote in Library-Centric Software Design. Retrieved July 1, 2006 from Library-Centric Software Design Web Site: http://lcsd05.cs.tamu.edu/slides/keynote.pdf.

Brandt, J., Guo, P.J., Lewenstein, J., Dontcheva, M., and Klemmer, S.R. (2009). Opportunistic Programming: Writing Code to Prototype, Ideate, and Discover. *IEEE Software*, 26(5), pp. 18-24.

Carroll, J. M. (2000). *Making Use: Scenario-Based Design of Human-Computer Interaction*. Cambridge, MA: MIT Press.

Carroll, J. M. and Rosson, M. B. (2003). Design Rationale as Theory, in J. M. Carroll (Ed.) *HCI Models, Theories, and Frameworks* (pp. 431-461), San Fransisco, CA: Morgan Kaufmann.

Clarke 2004. Measuring API Usability. *Dr. Dobb's Journal*. May 2004. pp.

Cockburn, A. (2004). *The End of Software Engineering and The Start of Economic-Cooperative Gaming*. Retrieved January 28, 2006, from Alistair Cockburn Web Site: http://alaistair.cockburn.us/crystal/articles/teoseatsoecg/theendofsoftwareengineering.htm.

Cwalina, K. and Abrams, B. (2005). *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*. Upper Saddle River, NJ: Addison-Wesley.

de Souza, C.R.B., Redmiles, D., Cheng, L., Millen, D., and Patterson, J. (2004). Sometimes You Need to See Through Walls – A Field Study of Application Programming Interfaces. *Proceedings of the 2004 ACM Conference on Computer Supported Collaborative Work (CSCW 2004)*. Chicago, IL: ACM Press, pp. 63-71.

Eisenberg, D.S., Stylos, J., Myers, B.A. (2010). Apatite: A New Interface for Exploring APIs. *Proceedings of the International Conference of Human Factors in Computing Systems (CHI 2010)*. Atlanta, GA: ACM Press. To appear.

Farooq, U. and Zirkler, D. (2010). API Peer Reviews: A method for evaluating usability of Application Programming Interfaces. *Proceedings of the 2010 ACM Conference on Computer Supported Collaborative Work (CSCW 2010)*. Savannah, GA: ACM Press, pp. 207-210.

Gamma, E., Helm, R., Johnson, R., Vlissides, J.M. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Upper Saddle River, NJ: Addison-Wesley.

Fischer, G., Henninger, S. and Redmiles, D. (1991). Cognitive Tools for Locating and Comprehending Software Objects for Reuse. *Proceedings of 13th International Conference on Software Engineering (ICSE'91)*. Austin, TX: IEEE Computer Society, pp. 318–328.

Mooty, M., Faulring, A., Stylos, J. and Myers, B.A. (2010). Calcite: Completing Code Completion for Constructors using Crowds. *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC 2010)*. Madrid, Spain: IEEE Computer Society, to appear.

Parnas, D.L. (1972). On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*. 15(12), 1972, pp. 1053 – 1058.

Parnas, D.L. (1994). Software Aging. *Proceedings of the 16th International Conference on Software Engineering*, 1994, pp. 279-287.

Parnas, D.L. (1998). Successful Software Engineering Research. *ACM SIGSOFT Software Engineering Notes*, 23(3), pp. 64-68.

Rosson, M.B. and Carroll, J.M. (1996). The Reuse of Uses in Smalltalk Programming. *ACM Transactions on Computer-Human Interaction*, 3(3), 219-253.

Scheiderman, B. (1980). *Software Psychology: Human Factors in Computer and Information Systems*. Cambridge, MA: Winthrop Publishers.

Software Engineering Institute (2008). Carnegie Mellon Software Engineering Institute Software Technology Roadmap: Application Programming Interface. Retrieved December 15, 2008 from: http://www.sei.cmu.edu/str/str.pdf. SEI 2008.

Stylos, J. (2009). *Making APIs More Usable with Improved API Designs, Documentation, and Tools*. Doctoral Dissertation. Pittsburgh, PA: Carnegie Mellon University.

Stylos, J. and Clarke, S. (2007). Usability Implications of Requiring Parameters in Objects' Constructors. *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007).* ACM Press, pp. 529-539.

Stylos, J., Faulring, A., Yang, Z., Myers, B.A. (2009). Improving API Documentation Using API Usage Information. *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC 2009)*. Corvallis, OR: IEEE Computer Society, pp. 119-126.

Stylos, J. and Myers, B.A. (2007). Mapping the Space of API Design Decisions. *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC 2007).* Coeur d'Alène, ID: IEEE Press, pp. 50-57.

Sun Microsystems (1997). *Java Collections API Design FAQ*. Retrieved March 2, 2010 from the Java 1.4 documentation: http://java.sun.com/j2se/1.4.2/docs/guide/collections/designfaq.html.

Zimmerman, B. J. (1995). Self-Efficacy and Educational Development, in A. Bandura (Ed.) *Self-Efficacy in Changing Societies* (pp. 202-231). Cambridge, MA: Cambridge University Press.