A Case Study of API Redesign for Improved Usability

Jeffrey Stylos[1], Benjamin Graf[2], Daniela K. Busse[3], Carsten Ziegler[2], Ralf Ehret[2], Jan Karstens[2]
[1]: *Carnegie Mellon University* [2]: *SAP AG* [3]: *SAP Labs, Inc.*
jsstylos@cs.cmu.edu, {benjamin.graf, daniela.busse, c.ziegler, ralf.ehret, jan.karstens}@sap.com

Abstract

As software grows more complex, software developers' productivity is increasingly defined by their ability to effectively reuse code. Even APIs (application programming interfaces) and other code explicitly intended for reuse are often difficult and time consuming for developers to use. This paper describes the user-centered design and evaluation process we evolved in redesigning SAP's BRFplus - a business rules engine, whose API was created for platform development, but which is now also increasingly being used by application developers – even though it was not initially designed with their specific needs in mind. Our API redesign attempts to take both the initial as well as the new emergent user requirements into account. A usability evaluation of our proposed changes to the API suggests that our user-centered design process was successful in helping to create an API that significantly improved users' productivity and better matches the different users' needs.

1. Introduction

As a market leader in business software, SAP employs roughly eighteen thousand software developers. One of the key elements of their productivity is being able to quickly and effectively reuse code that colleagues have already written. APIs explicitly allow for code to be reused by other developers; however previous work [8] and experience at SAP has shown that using APIs that do not meet the developers' specific requirements can often be tedious, difficult, or even impossible. As a result developers often spend a long time trying to make existing APIs work for them, and might end up writing code from scratch rather than using a difficult-to-use API.

To better understand this problem and explore possible solutions, we examined a specific API — the SAP "Business Rules Framework Plus" API (BRFplus) implemented in the ABAP programming language. We used this API as the focus of a case study in which we developed and applied a process for studying and improving the usability of APIs. We chose to study the BRFplus API because of its importance to SAP (BRFplus recently replaced all 26 previous rules engines to provide one common and standardized business rules framework to all of SAP), and because some users had reported difficulty using it.

By encapsulating business rule functionality, the BRFplus API allows additional functionality to be specified by business users, who know the details of what the software should do, instead of software developers (see inset on next page). This provides the potential for a powerful customization of software at the hands of the business user, enabled by BRFplus. By making BRFplus easier to use, we hope to bring it to more areas of SAP, providing more flexibility and value to SAP's customers.

We did a user-centric design of an API wrapper (using input from stakeholder interviews, user requirements gathering sessions, and a pseudo-code study) and then performed a usability evaluation to assess its value in terms of helping application-level developers address their specific use cases more easily.

Our overall goals stretch beyond the BRFplus API. We hope demonstrate within SAP the value in focusing on API usability. SAP has developed and applied user research for user interfaces and other areas, but — like most companies — has not previously examined the usability of APIs with a systematic user experience effort. Doing this will help SAP create APIs that are easier to use, improving the productivity of SAP's developers and customers. In doing so, we hope to advance the state of art in API usability and contribute our insights, results, and techniques back to the research community.

2. Identifying User Requirements

The BRFplus API we examined allows for the creation and editing of business rules. The original API was primarily intended for internal SAP developers for creating platform level code – i.e. code that hundreds of SAP applications and solutions would be building

1-4244-2528-0/08/$25.00 ©2008 IEEE

on. This requires the API to give the developer a significant amount of control and transparency in using the API. Initial interviews with stakeholders and domain experts at SAP brought to light, however, that also more and more application developers were using the API directly for their coding tasks – and that these users were increasingly running into difficulties using the API, as it was not designed to meet their specific (less granular and less flexible) needs. We received consistent feedback from our initial stakeholder interviews that, while they had designed a powerful and flexible API (as required by platform users), many current users struggled to implement their simpler use-cases..

In terms of the cognitive dimensions [5], which provide a framework for describing usability issues and have been adapted to describe API usability [2], the interviews led us to suspect that there was a mismatch of abstraction level. The API was seen as providing low-level functionality (and was thus very flexible) while the specific class of application developer users had higher-level goals, prioritizing simplicity of use over control and transparency – their use cases were not as complex as to require significant amounts of granularity or flexibility, and the time spent in understanding and debugging the API's use was very limited compared to platform developers.

The difference in user roles can also be expressed by previously identified developer personas [3] that describe different styles of programming with different strategies and goals. We used these personas to help understand different developer strategies in our research. The system level developers at SAP, who help write the platform that others rely on, tend to exhibit traits of the systematic programming persona, cautiously understanding the implications of each line of code. The application level developers tend to act more pragmatically, wanting to understand and tweak the code but also be productive. Consultants, under the most time pressure, often exhibit traits of the opportunistic programmers.

Business Rules

Business rules [7] allow end users (non-programmers) to specify software behaviors that would normally be specified at software development time by a developer. This allows software to be customized by users after the software is deployed. For example, business rules might be used to specify how much tax is computed on different items, since this varies by location. An end user might do this by using a graphical user interface (GUI) to specify that for "food items" the total tax is 5% of the base price, and "other items" are taxed at 7%, potentially providing more detailed specifications for food and other. This differs from "traditional" programming, where a software developer would write a function with an "if" or "case" statement to compute the tax, and the code would have to be recompiled when the logic changed.

Business rules can similarly be used to compute how large each employee's end-of-year bonus should be, or how to specify rules used during the approval process of expense reports. After being specified by a user, these rules can be automatically executed, like software. In this sense a business rules API and accompanying GUI provide an environment for end user programming [3].

In order to validate these initial hypotheses based on our stakeholder interviews, we ran six 60-80 minute individual requirements gathering sessions with existing BRFplus users. In these sessions, we asked the developers to first explain the overall purpose of their code that used business rules, and then to explain which parts of the BRFplus API they used, and how. We also discussed how they might have wanted to use BRFplus but were not able to.

In our interviews the API users we talked to did indeed have relatively simple use cases, compared with the flexibility of BRFplus.

For example, one developer used BRFplus for Manufacturing Execution. The automatic generation of production orders as well as the guidance of material lots through a shop floor was to be governed by rules. For instance, a rule might specify that scheduling of certain work packages on the shop floor (e.g. those concerning Product X, or from Supplier Y) should follow a specific scheduling strategy – with those strategies themselves being sets of rules, or a formula). However, rather than describing the full rules set using BRFplus (which would be possible using a nested construct of tables and formula rules – one of the more complex features in BRFplus), the developer instead used BRFplus only to return a "strategy-code" that referred to a rule that he hard-coded outside of BRFplus. This made the business rules easy for business users to execute, with a loss of generality that was seen as acceptable for this use case. But it also limited the ability of the business user to specify rule changes at run-time, having to rely on the hard-coded values predefined by the developer.

This is an example of what we saw with a number of uses of the BRFplus API: While the overall scenarios often included computing values from formulas and triggering actions based on rule results, the users we interviewed moved this functionality outside of the rule system for simplicity, having a developer specify the formula or action rather than an having an end-user specify it in the rule itself.

3. Redesign Strategies

In previous API usability research [1][8][6], the results of API usability studies have been used to improve the API before its final release. The API we examined had already been (internally) released and used, and so the team had to continue to support it. We could not simply create a new version and ignore the old API, however we had several different options about what to do. For example we could adapt the BRFplus API in backwards-compatible ways (which would limit the changes we could make) or we could create a completely new API (at the cost of having to support both of them). Because of the specific abstraction level problem we had identified, we chose to design a "wrapper API," a higher-level API implemented on top of the original API. This would also enable programmers to choose which level of granularity they wanted to interact with the API on, the wrapper, or what was underneath. Other scenarios might well call for other types of solutions, such as adding additional classes at the same level or even lower levels of abstraction.

4. Pseudocode to Reveal Expectations

Before designing a wrapper API we first wanted to learn how users thought, what (if any) mental models they had and what terminology they used. To do this we designed a study in which existing and prospective users would write pseudocode against an imaginary business rule API using a simple text editor. We adapted and extended this technique from our previous work examining API design choices [8][4].

We contacted several users of the BRFplus as well as other developers who were knowledgeable about business rules, and conducted another round of six different one hour sessions. Because the volunteer participants willing to participant were mostly remote colleagues in different locations, we conducted all of our sessions remotely via telephone and Windows NetMeeting to see their screen. This setup allowed us to discuss privately by muting the speakerphone.

We emailed participants study instructions immediately before the study that briefly described our project and gave them a scenario to write pseudocode for. Participants were told they could write ABAP or Java-like pseudocode. During the study we asked participants to think aloud.

We asked participants to write code for the task of defining the rental car price for customers based on their age and the rental duration. We chose our example to be representative of the application-level use cases we saw in our interviews that would be easily understandable.

The pseudocode that the six participants came up with showed many similarities to each other. Some of the results were:

- Participants wrote code at a high level of abstraction, reinforcing what we had learned in earlier interviews. Participants wrote in a dozen lines what would take a hundred or so with the BRFplus.
- Participants tended to separate the *structure* of rules from the *data* in their code. For example, users would first specify that a number, "price", should be associated with an age and a duration, and then set the specific values. In the current BRFplus model, there is no separation between rule structure and data.
- Most participants used tables to model the structure of their rules. This can possibly be put down to the fact that the most efficient data structure for handling sets in ABAP are internal tables.
- Participants omitted details such as locking data structures for thread safety, versioning and activation, and explicitly saving rules. These details were required to support more complicated scenarios, but could be handled automatically in the simpler cases.

5. Usability Evaluation

After we had designed and implemented a prototype version of the wrapper API we designed a think-aloud study to evaluate the wrapper design. Our goals were to find areas of improvement and to see if participants would be able to use the API at all without much documentation. We chose to provide almost no documentation because we wanted to avoid hiding unusable aspects of our design behind good documentation.

We based our API evaluation study design on the previous work on evaluating early API designs [1][8]. In the study, participants wrote code that used the wrapper API to implement a series of up to three tasks, as time allowed. Task one involved creating new rules for how many vacation days different employees in different countries should get (e.g. German, full time employees get 30 days vacation a year). Task two involved storing these rules and then loading them to calculate the vacation days a new employee should get. Task three involved creating an additional rule involving ranges (e.g. German, full time employees who have worked less than one year get 20 days vacation). These tasks were a simplified version of a real use-case.

We then performed a third round of 60-90 minute sessions, this time to study the usability of the simplified wrapper API we proposed. Three of these sessions took place in a usability lab with a one-way mirror separating observer and participant rooms. The other

sessions took place remotely, using a phone and Netmeeting for screen sharing. We maintained a connection between the participant's computer and the observer's computer and used screen-capturing software to record the session on the observer's computer.

To better understand participants' behavior, we asked them to think out loud as they worked. A consequence of this is that the time they spent to complete a task could have been affected by speaking out loud. This was especially likely since most of our participants were non-native English speakers and we asked that they speak their thoughts in English (the only common language among the project members).

We gave participants brief written study instructions and documentation giving a one-to-two sentence description of each class in the wrapper API.

The high level results of the study were that five out of the six participants were able to finish the first task in 90 minutes, four out of six were able to finish the first two tasks, and three were able to finish all three tasks within 90 minutes. Because of the limited documentation and time, we felt that having most participants finish at least one task was a positive reflection of usability of the API. We also observed some common difficulties using our API; fixing these might allow more participants to finish all of the tasks.

We had initially considered performing a comparative study between the wrapper API and the original API. However, the BRFplus felt that the results of the wrapper API evaluation were so strong that using it was clearly faster and it was not worth having participants perform the same tasks with the original API, which the BRFplus team did not feel would be practical in 90 minutes. To get an upper bound on the usability of the original API, we had a developer of the BRFplus team with intimate knowledge of the API perform the same study tasks using the BRFplus instead of the wrapper API. The BRFplus developer was able to solve all of the tasks, but required 120 minutes, because of the additional details required to keep track of in the BRFplus as opposed to the wrapper API.

6. Design Iteration

Overall participants were able to use the wrapper API easily. However our study revealed several, relatively minor, usability problems with the wrapper API design. Based on these observations, we created a revised wrapper API. If time had permitted, we would have then run more study participants with the revised API to ensure that these changes solved the problems we observed and did not introduce other problems.

We had assumed that participants would be familiar with the "range" object in ABAP programming language. These are used to create rules with ranges, for example that employees who have worked between 1 and 2 years should get a certain number of days vacation. However, we found that while most participants had heard of it, few knew how to use this construct. Based on this, our revised API included a convenience function for adding rules ranges.

In our initial design we used two separate objects for creating rules and for using the rules to compute a value based on some input. We chose this because rules can contain ranges, while the concrete values used for rule instance processing cannot. However, participants had some difficulty understanding the distinction between these classes, so our revised wrapper uses the same class for both of these operations.

7. Conclusions

We did a user-centric design of an API wrapper and then performed a usability evaluation to assess its value in terms of helping application-level developers address their specific use cases more easily.

Beyond the implications for our target API, we hope that this project will help demonstrate the importance and viability of API usability. Our project showed that, while not trivial, an API can be investigated, designed and evaluated (in prototype form) in three months with a handful of people and the help of current and prospective users.

8. References

[1] Clarke, S. Measuring API Usability. *Dr. Dobbs Journal*, May 2004, pp S6-S9. 2004.

[2] Clarke, S. Describing and Measuring API Usability with the Cognitive Dimensions. *Cognitive Dimensions Workshop*. 2006.

[3] Clarke, S. "What is an End User Software Engineer?", *End User Software Engineering, Dagstuhl Seminar Proceedings*, Dagstuhl, Germany, 2007.

[4] . Ellis, B., Stylos, J. and Myers, B. The Factory Pattern in API Design: A Usability Evaluation. *International Conference on Software Engineering*. 2007

[5] Green, T.R.G., and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework", *Journal of Visual Languages and Computing*, 1996, pp 131-174.

[6] McLellan, S.G., Roesler, A.W., Tempest,, J.T. and C.I. Spinuzzi, "Building More Usable APIs," *IEEE Software*, vol. 15, no. 3, May/Jun 1998, pp. 78-86.

[7] Ross, R. G. *Principles of the Business Rules Approach*. Addison-Wesley Longman, Amsterdam, February 2003.

[8] Stylos, J. and Clarke, S. Usability Implications of Requiring Parameters in Objects' Constructors. *International Conference on Software Engineering*. 2007.