

Usability Challenges for Enterprise Service-Oriented Architecture APIs

Jack Beaton, Sae Young Jeong, Yingyu Xie, Jeffrey Stylos, Brad A. Myers
School of Computer Science, Carnegie Mellon University
jackbeaton@cmu.edu

Abstract

An important part of many programming tasks is the use of libraries and other forms of Application Programming Interfaces (APIs). Programming via web services using a Service-Oriented Architecture (SOA) is an emerging form of API usage. Web services in a business context (called enterprise SOA or E-SOA) add additional complexity in terms of the number of the services, the variety of internal data structures, and service interdependencies. After altering existing Human-Computer Interaction (HCI) methodologies to address the unique context of software development for SOA, we evaluated a large E-SOA API and identified many usability challenges. Prominent results include difficulties developers encountered while assembling data structures in web service parameters, cycles of errors due to unclear control parameters within data structures, and difficulties with understanding long identifier names. We recommend a tolerance for unspecified objects in automatically-generated web service proxy code, consistent data structures in parameters across services, and encoding optional namespace schemes into WSDL files.

1. Introduction

The informed design of Application Programming Interfaces (APIs) is important to ensuring the effectiveness of developers. Design choices can affect the levels of efficiency and rates of adoption of APIs, but without objective and empirical data to justify design decisions, API usability is a difficult goal.

“Service-Oriented Architecture” (SOA) is a way to divide software into services that communicate over a network with the client and with other services, and which can be combined into composite applications. Enterprise SOA (E-SOA) is focused specifically on supporting processes throughout an entire organization, such as through Enterprise Resource Planning [9].

E-SOA, as supplied by many companies through web services technology, is a new development paradigm whose usability aspects have not yet been studied. Complex business needs cause a tremendous increase in complexity, in addition to the existing usability issues with web service technology. Prior research has

shown that programming using object-oriented APIs has interesting and unique challenges with respect to usability and API design [2, 3, 8], but the usability implications of large SOA API frameworks for developers are interesting and unstudied.

We focused on the usability challenges of coding a composite application using a large E-SOA API framework provided by a cooperating company. This paper summarizes the usability problems we found, and discusses recommendations for future E-SOA projects.

2. E-SOA Web Service Model of Use

Web services are provided by a *service provider*, and used by developers to create an application that provides the service to end users. We call the developer a “consumer” of the web service.

The typical steps when consuming web services using the common SOAP protocol involve first finding the appropriate service in the online documentation, and downloading the requisite Web Service Description Language (WSDL) file for that service from a service registry. The WSDL file is entered into a *stub generator* which creates a code reference “stub”, and finally the developer writes the application code against the stub. The stub code supports the service by sending and receiving SOAP XML messages, and makes the service appear like familiar “offline” code in the developer’s Interactive Development Environment (IDE). The WSDL file describes the supported messages and their parameters. The consuming application can be written in any language that supports SOAP, including Java, C#, C++, Ruby, etc.

API designers providing web services are limited in their choices by the lack of support for richness in data structures. The services in the API we studied, rather than taking values as a series of parameters, primarily take one data structure as a single parameter that must be constructed and filled with values, many of which are themselves data structures. The required data structures are fairly large and complex. Furthermore, various fields serve as control parameters that select one of multiple possible uses of each service. For each use, some fields are required to be filled in, some fields are required *not* to be filled in, and other fields are option-

al. Patterns of values and fields that do not meet one of the templates recognized by the service will result in errors. We were working in C#, in which each data structure is implemented as an object that users create, but these objects have no methods relevant to consuming the service.

The API naming scheme needed to ensure that each of the tens of objects used within each of the hundreds of services would remain uniquely named if any two services were used in the same composite application. As a result, the names of the objects were very long, often over 100 characters.

3. Related Work

Other research in this field has helped inform our studies, but does not focus directly on the unique topic of SOA API usability.

SOA API designers have debated the Documentation-style and Remote Procedural Call (RPC) WSDL standards. RPC WSDL files allow the specification of API design patterns consistent with certain target languages (such as naming specifications for languages supporting namespaces) at the expense of compatibility with other languages, restricting their popularity [1, 6].

Existing research on API usability focuses on non-SOA development. Ko described learning barriers associated with programming, including the phenomenon of misconceptions by developers leading to further misconceptions [5]. Stylos developed methodologies for studying APIs [3, 8], which we have adapted for the current research.

Providing the correct “granularity” of services has consequences for service usability. Jones listed anecdotal SOA “anti-patterns”, or common mistakes made when developing SOA architectures. These include problems caused by service hierarchies that are either too fine-grained or too coarse-grained [4].

In the case of E-SOA, a fine-grained design with too many services makes finding the correct service difficult. On the other hand, providing a coarse-grained design of a few multipurpose services makes them difficult to understand, because they have many parameters controlling variations of behavior. Large-framework E-SOA vendors must supply functionality for a vast variety of business needs, and so they simultaneously face both the challenges associated with fine-grained services and those associated with coarse-grained services. Jones’ solutions in both cases are to move towards the opposite extreme [4], which companies with large business application frameworks, already arguably in the middle, cannot do without removing functionality and alienating customers.

4. Methodology

To better understand the challenges of consuming E-SOA web services, we performed a user study focusing on how well developers could code a composite application using a large-framework web service API.

Six participants were recruited for the coding study. All were recruited from university classes on web services and were familiar with consuming web services from class work. Of the six participants, two had substantial experience with enterprise applications and business processes.

We identified services relating to the generation of new sales orders using customer and product information as likely to be heavily used. The scenario chosen was a sales order creation application. Participants were given product, supplier, and buyer string names, and three services to translate each into the primary key IDs required by a fourth service to create a sales order.

Participants used the Visual Studio .NET environment due to its current popularity among existing E-SOA customers of the tested API. We provided the coding study participants with a pre-configured C# project, including code stubs generated from WSDL files and a user interface of a sales order creation utility for which the participants were asked to create working code. Participants were given the service documentation and asked to consume services in this order: “Find Customer”, “Find Supplier”, “Find Product”, and “Create Sales Order”. Testing then proceeded using the Think Aloud protocol.

5. Study Results

All six participants in the study implemented the first service, “Find Customer”, but only with significant advice from the moderator. Only two of the six also implemented the “Find Supplier” service within the time limit, one of whom had business application experience. None implemented the “Find Product” or “Create Sales Order” services successfully.

We observed eight barriers that the participants experienced, and classified them by the errors that caused them. Three of these were given informal names, because they were very disruptive and quickly became familiar:

Assembly Error (“Some Assembly Required”): This occurred when the developer failed to create and combine some or all of the required message tree in the correct way. For example, for the “Find Customer” service, a Customer object must be sent that contains a Common object that contains a Name object. Absences caused unhelpful null reference exceptions, and often occurred because, in the generated stub code, instantiation of an object does not imply automatic instantiation

of objects lower in the tree hierarchy. All users were surprised to discover that they needed to create every one of the objects themselves, instead of just the top-level object used in the method parameter.

Exclusion Error (“Batteries Not Included”): In addition to the data-carrying object, users must include miscellaneous objects as parameters of the service method, such as a log object. Unlike the main data structure, they do not need to be combined or assigned values, but they must be supplied. Some of these were undocumented, and were often left out by participants.

Specification Error (“Do Not Eat Silicon Gel”): Certain objects should *not* be created or given values to accomplish the task, because doing so was interpreted as an incorrect, incomplete new branch of the tree. Users filled an invalid combination of fields in the data structure, usually because they thought they needed an object that signified a different service functionality, but this caused other unspecified fields to become mandatory. This was the most time-consuming error to overcome.

Structure Error: All users expressed confusion about the basic structure of the object combinations that must be created. Users were unsure of what the fields meant, and what data should go where.

Default Error: Three users were unsure if default values must be supplied or an object must be created, and if so, what they should be. All three tried to guess values, which led them to make Specification Errors.

Foundation Error: Web service providers cannot know what API a third-party client-side stub generator will create, and so documentation is vague and code examples are missing. All users were confused by ambiguous descriptions of the generated code stubs in the online API documentation.

Naming Error: All six users either misread names or were confused by them. All complained about the length of the names and all made accidental Specification Errors by mixing up two long object names that are different only because of one word in the middle, which may be a “blind spot” in long names.

Consistency Error: Before the think aloud test, all six users asserted that the “Find Customer”, “Find Supplier”, and “Find Product” services would share similar API design patterns because they shared a similar purpose. Of the two users who implemented the first two tasks, one deliberately copied and pasted the “Find Customer” code and tried to manually replace each word “Customer” with “Supplier” before realizing the correct code was quite different. This is because the data structures of the “Find Customer” and “Find Supplier” services carried similar information using a different pattern of branches.

6. Discussion

Coding errors due to invalid values were observed leading to other errors, which steadily increased in severity until the moderator had to intervene.

Of the three users who exhibited the Default Error symptom, all three tried to guess field values and made Specification Errors, and none implemented more than one service. One attempted to set every field that could be found until stopped by the moderator, one attempted to avoid specifying bad values by setting them all to NULL (which was also a bad value), and another user refused to continue with the task until he was told whether or not to specify the value of an incorrect field after expressing the intuition that he ought to do so.

These appear to be symptoms of *cascading programmer error*, which we hypothesize is a condition in which violated expectations cause the developer to question their every assumption about the API and so begin exploring a wider space of possible actions, resulting in more errors that would not otherwise have occurred and a vicious cycle. Ko found evidence of developer misconceptions leading to further misconceptions [5]. During an earlier study on API design patterns [8], it was observed that if users experienced errors due to several unexpected requirements, they would specify values they did not understand to try to avoid another error message.

Structure, Foundation, and Consistency Errors preceded Assembly and Inclusion Errors, making the participant suspect the documentation. To avoid further error messages, the developer began specifying all available fields. Without examples or descriptions, the developer does not know which values belong where (Structure Error); because they do not know if defaults are set automatically they assume that the null reference errors experienced during Assembly will continue until every possible field is specified (Default Error). Because the values entered are incorrect, Specification Errors cause new errors, creating a cycle that is self-sustaining until all possibilities, or the developer’s patience, are exhausted.

Repeated Specification Errors were the most time-consuming, because unless given outside help or working example code, developers could continue trying random combinations indefinitely. The complexity of available fields makes identifying one of the modes of a multipurpose service by guesswork a monumental task, resulting in loss of productivity and a general sense of despair and intimidation.

Long names were used to uniquely identify the vast number of objects in the large framework of services, to avoid naming collisions when services are combined into a composite application. Since the WSDL design-

ers could not count on having a namespace feature in the target programming language, the front of the name was used for the service name. The end was used to communicate the object's position in the data structure. This makes it more likely that other critical information may occupy a "blind spot" in the middle of the resulting long names.

Long names are also harder to mentally process and often cause code to extend horizontally off the screen, requiring horizontal scrolling, a well-known problem during normal text navigation [7]. In our study, the lack of visibility of the surrounding code while scrolling to the right was seen to interrupt and frustrate the developer's concentration. On the other hand, names that were too short to be descriptive were observed to be misinterpreted in our study, causing Structure Errors.

Other results from our study, which we do not have room to discuss in depth here, include that business domain knowledge is valuable when navigating for appropriate services, that users are principally searching for input and output parameters to better link up the modular services, and users deliberately turn away from areas of a documentation site with an inconsistent visual design because they believe they have left the site.

7. Recommendations

To prevent Assembly Errors entirely, generated stub code could arrange for all fields of an object to be automatically instantiated in the constructor of the top-level object, using default values that signify the object is uninitialized. Provided services should also tolerate objects supplied as parameters with fields that are uninitialized or empty. This requires the cooperation of stub generator tool designers and SOA API designers. Such coordination may require substantial effort, but could make the largest difference in the usability of web service APIs and the long-term adoption of SOA.

A more easily achievable goal by individual service providers is to standardize services so that the code for each is consistent. Although the service provider may know that the backend for each service is very different, if the consuming developer perceives the services as similar, they will expect a similar interface. Services sharing the same business object or appearing to perform similar operations should "inherit" the same comprehensible data structure. This will preserve developer expectations by presenting greater consistency across a large API framework.

Current naming may result in long, difficult names in large-framework SOA APIs because all objects in all services must be unique, so multiple services can be used in a composite application. Documentation-style

WSDL files could encode an optional namespace schema for stub generators for languages using namespaces, like RPC WSDL files can. Research is needed to determine how multiple web services could be combined into the same namespace structure. If name lengths cannot be reduced, research is needed to quantify the effects of name length on usability.

8. Conclusions

We found that users have significant challenges in consuming enterprise SOA web services that can potentially be overcome. There is a large space for future research into the benefits of SOA API usability, and we hope to start an ongoing dialogue based on empirical data about current problems and feasible solutions. We hope our research will help create improved APIs, clearer documentation for the APIs, and better tools to support the use of the APIs.

9. References

- [1] Akram, A. and Meredith, D., "Approaches and Best Practices in Web Service Style, Data Binding, and Validation," in *Securing web services: practical usage of standards and specifications*, P. Periorellis, Editor 2007, Idea Group Inc. pp. chap. 13., pp. 7.
<http://epubs.cclrc.ac.uk/bitstream/1542/AsifAkramDaveMeredithChapter.pdf>.
- [2] Clarke, S., "Measuring API Usability." *Dr. Dobbs Journal*, May, 2004. pp. S6-S9.
- [3] Ellis, B., Stylos, J., and Myers, B. "The Factory Pattern in API Design: A Usability Evaluation," in *International Conference on Software Engineering (ICSE'2007)*. May 20-26, 2007. Minneapolis, MN: pp. 302-312.
- [4] Jones, S., "SOA Anti-Patterns," Jun 19, 2006. C4Media Inc.: InfoQ.com. <http://www.infoq.com/articles/SOA-anti-patterns>.
- [5] Ko, A.J. and Myers, B.A. "Development and Evaluation of a Model of Programming Errors," in *IEEE Symposium on End-User and Domain-Specific Programming (EUP'03), part of the IEEE Symposia on Human-Centric Computing Languages and Environments*. October 28-31, 2003. Auckland, New Zealand: pp. 7-14.
- [6] Loughran, S. and Smith, E. "Rethinking the Java SOAP Stack," in *IEEE International Conference on Web Services (ICWS)*. 12-15 July, 2005. Orlando, FL: <http://www.hpl.hp.co.uk/techreports/2005/HPL-2005-83.pdf>.
- [7] Nielsen, J., "Scrolling and Scrollbars. Useit Alertbox," July 11, 2005. NielsenNorman Group: <http://www.useit.com/alertbox/20050711.html>.
- [8] Stylos, J. and Clarke, S. "Usability Implications of Requiring Parameters in Objects' Constructors," in *International Conference on Software Engineering (ICSE'2007)*. May 20-26, 2007. Minneapolis, MN: pp. 529-539.
- [9] Woods, D. and Mattern, T. 2006 *Enterprise SOA: Designing it for Business Innovation*. O'Reilly Media, Inc.