Web Services as the Foundation for Learning Complex Software System Development

Marty Humphrey
Department of Computer Science
University of Virginia
Charlottesville, VA 22904
humphrey@cs.virginia.edu

ABSTRACT

A significant challenge for Computer Science departments is how best to get new graduate students involved with their chosen research projects. Ideally, the incoming graduate students will as a whole have both a solid understanding of computing principles behind large-scale software development and a broad "skill set", e.g., for conducting systems-oriented research. This rarely occurs, due to the diversity of backgrounds of incoming students–even the most qualified applicant can have deficiencies. To address this problem, we have developed a first-year graduate course that balances and integrates practical considerations with basic principles of complex software system development. To make the discussions of designing, implementing, and evaluating complex software systems more concrete, we situate the core of the class in the context of Web Services. We are currently teaching this class for the second time, and, while there will always be open issues given the nature and scope of this class, we have received positive feedback from the students and the other members of the department. Students recognize the practicality of Web Services. Students also appreciate the research possibilities that arise while evaluating the state of the art with regard to Web Services.

Categories and Subject Descriptors

K.3 [**Computers & Education**]: Computer & Information Science Education – Computer Science Education

General Terms

Management, Design, Experimentation, Security

Keywords

Teaching, Computer Science systems research, Distributed systems

1. INTRODUCTION

The learning curve for new graduate students as they start their research careers can often be steep. When a student joins an existing research group, the student can, understandably, spend most of his/her time learning and appreciating the significance of the research issues being addressed by the group. However, a student can often spend a significant amount of time learning their group's established techniques, methodologies, and tools. Because graduate students' productivity is a crucial determinant of the research output of the department, it is increasingly important that this time spent learning "background material" be minimized, so that the time can be spent investigating the core research issues. We refer not only to the basic tools of UNIX-based software development (such as *gcc*, *emacs*, *gdb*, *purify*, etc.) and Windows-based software development but also to the "skill set" of large-scale software development, such as the ability to design and implement software that *others* can use, read and/or modify, write a concise yet comprehensive software design document, design a comprehensive test and evaluation plan, etc. The prototypical new graduate student has ever only written short programs that don't persist beyond the scope of a single semester or quarter and are usually tested with the professor's input files. Incoming graduate students usually have little or no experience in designing, implementing, and evaluating large-scale software systems for complex, dynamic, and heterogeneous environments.

The traditional approach to this problem is for the graduate student's advisor, over time, to identify the weaknesses and send the student off to self-learn the material. Learning a single tool such as *gdb* in this manner can work in the right situation (although this sometimes results in an incomplete, superficial understanding, requiring further hands-on instruction). However, there is generally not a suitable tutorial for learning some of the less tool-oriented of this background material (such as performance evaluation, which can often be very fined tuned to the subtleties of the particular software being evaluated). Arguably, an instructor in a structured class setting can best facilitate the learning of basic principles of large-scale software development while rounding out the students' tool set.

We have created a course for designing, building, and evaluating complex software systems. Because the research projects in our department are increasingly focusing on these large software systems (as opposed to the more theoretical aspects of computer science), it is a required course for all first-year graduate students. The goal of this paper is to describe the approach, issues, and lessons learned in designing and teaching this course. One of the biggest challenges was deciding what *not* to include in the class! The core of the class, and the core of the discussion in this paper, is the use of Web Services as the foundation for learning complex software system development. While it might appear at first glance that Web Services might not be the best match for an introductory course that teaches core software principles—we should wait until the "Web Services landscape" (including standards, tools, and interoperability issues) becomes more stable! However, we argue that this actually *further justifies* its use in this type of course, as students are exposed to both the practical aspects of complex software development and the positive/negative aspects of creating or using such cutting-edge technology (much of what will define "Web Services" still resides in R&D labs and in the standards-building organizations). Overall, we have found that students recognize the practicality of Web Services. Students also appreciate the research possibilities that arise while evaluating the state of the art with regard to Web Services.

The remainder of this paper is organized as follows. Section 2 describes the course goals, challenges and decisions made regarding the course content and structure. Section 3 details the outline of the class and the goals of each subsection of the class. Section 4 describes the use of Web Services in the classroom, both from a practical viewpoint and as a more theoretical enabler of distributed systems research. Section 5 contains a discussion of the successes and failures of the class. Section 6 contains the conclusions.

2. COURSE GOALS, ISSUES, AND DECISIONS

The creation of the class originated a few years ago with the near-universal agreement within the department's graduate curriculum committee that, while incoming graduate students met and surpassed the high standards of admission, nearly all of them could benefit from revisiting or re-learning some basic computer science background material and/or set of skills. These skills, in particular, were viewed as being important for most of the existing research projects in the department. Such skills include OS concepts and UNIX programming (e.g., process creation and manipulation, synchronization, threads), basic software engineering principles, computer security, and distributed systems. More broadly, it was perceived that incoming students could greatly benefit from a systematic discussion of how to design software that persists longer than their typical semester projects, how to navigate large existing software projects, how to design experiments and evaluate software artifacts, etc.

But while everyone agreed that this class could be very effective for the research productivity as a whole, the design of the class presented a number of challenges:

Course Content. While most professors agreed that this class should exist, there was little consensus on *exactly what material* should be contained in the class. Roughly speaking, the proposed content ranged from being very hands-on to being very theoretical. The proposed course topics also ranged from an emphasis on UNIX programming to an emphasis on software engineering.

Leveling vs. Learning. Students come from all over the US and the world. Therefore, there will probably be a point in the class that every person would see material of which they are already quite aware (hopefully, not all at the same time!). This required that the course be highly dynamic to the particular makeup of the class. Above all, the class was *not* to be a "utility course" that merely leveled all students but rather a opportunity to learn core concepts for designing, building, and evaluating software systems (with discussions of "tooling" where appropriate).

Lack of Textbook and/or Example Courses. We could not identify a textbook that could be used as a good supplement for the entirety of this class. Similarly, while we found course at other universities that covered parts of this proposed course, we could not find a close match upon which to model ours. While we ultimately recommended a few textbooks in various parts of the class (e.g., Silberschatz' "Operating Systems Concepts" [10] for a few weeks in the beginning, Szyperski's "Component Software - Beyond Object-Oriented Programming" [12] for a few weeks toward the middle, etc.), ultimately we just pointed the students at *google.com*, which is a necessary tool for research anyway.

Instructor. The identification of the proper person to teach such as class was not trivial. That is, this class was arguably unique in that it had to span many traditional computer science disciplines, such as operating systems, networks, programming languages, compilers, and distributed systems! While many professors *understand* the principles of these subjects, much fewer are prepared to *teach* across this broad material. We settled for a single instructor for the majority of the class, with two other instructors coming in to teach the core software engineering section and the performance evaluation section.

Our department's approach to the major issue, which was course content, was to let the particular instructor make many of the finer-level decisions while staying in the general guidelines and intent of the course. The instructor chose to take an "historical approach" to the hands-on material in the class, by beginning with single-machine (UNIX) programming, and then building into approaches of higher-level functionality for building distributed systems. The distributed systems portion of the class covered (in order) sockets, (SUN)RPC, CORBA, and Web Services/.NET. The approach was to focus on programming assignments as an enabler for both the tools development and for discussions into the broader aspects of software development. For example, a sockets assignment both covered the compile-execute-debug cycle on UNIX (*emacs*, *gcc*, *gdb*, perhaps *cvs* and *purify*) and covered experimental design for ensuring that the server listening on the socket was complete and correct. All programming assignments required a thorough design document (approximately 5 pages) in addition to the code itself.

The decision to emphasize Web Services/.NET in a mandatory first-year graduate class was based on many observations. It is *important*, in that Microsoft is placing its future on the success of Web Services/.NET. When a company of Microsoft's impact on the world produces and endorses an approach, it is undoubtedly important. It is *modern*, reflecting the latest in language design and software development methodologies. It is *practical*. And it is *emerging*, implying that that it is not a finished product. As we describe in Section 4, there are a number of issues in .NET that are being actively researched by Microsoft and the Web Services community. We used the emerging aspect of .NET to reinforce to the students that they were learning not only something that could be of practical importance, but it could also excite their desire for research (e.g., by identifying potential topics). The *breadth and scope* of .NET allowed students the opportunity to pursue their own particular interests, be it programming language design, distributed system design, compilers (e.g., the Shared Source Common Language Infrastructure, SSCLI [6]), security (e.g., sandboxing techniques, distributed system security, etc.), mobile computing via the .NET Compact Framework, etc. Finally, Web Services fits perfectly with complex, dynamic, and heterogeneous environments.

3. COURSE OUTLINE

The course lasted 16 weeks. The topics (and their durations) are as follows:

Classic systems papers (1 week). To set the proper context, we began the class by covering five classic systems-building papers: "The Scientific Method" [14], "An Evaluation of the Ninth SOSP Submissions" [4], "The Task of the Referee"[11], "Hints for Computer System Design"[3], and "End-to-End Arguments in System Design"[9]. These papers are very important and do not require much context in order to understand many of the main points.

UNIX programming (2 weeks). Systems research does not *require* UNIX, although the ability to utilize the UNIX environment is extremely valuable in many situations. The first programming assignment was to implement a simple shell (to both start off the semester with a reasonably straight-forward project and to reinforce the concept of OS as service provider). The second programming assignment required the students to implement a remote shell. The goal of this second assignment was to have the students gain hands-on experience with programming distributed systems–first, via the low level socket abstraction and then via the higher-level interface provided by SUNRPC.

CORBA (1 week). We continued with a move to higher-level middleware via a discussion of CORBA. This included the history of CORBA and the Object Management Architecture (OMA), including the Object Request Broker (ORB), CORBAservices, and CORBAfacilities. There was not sufficient time to have a programming assignment.

Component-based development (2 weeks). The topic next introduced was components, which are independent pieces of executable code that can be dynamically incorporated into programs, allowing the programs to add and change functionality as they execute. Students are introduced to the basic architectures of component systems, learning how to develop simple components, and use component architectures to develop small programs. We contrasted objects with components and covered interfaces, IDLs, common type systems, metadata information, and execution semantics. We used [12] as a reference for this material.

Basic software engineering (2 weeks). This section covered the software development process, basic design in software, and source code standards. Many topics were covered, including the potential cost of designing software *incorrectly*, ethics, modularity, design methods (e.g., top-down functional decomposition), documenting the software design, and best practices in source code (e.g., inheritance, copy constructors, memory management, constants, and public inheritance).

Web Services/.NET (5 weeks). The purpose of this section was to continue to investigate the challenges of creating large, complex software systems via higher-level services, in particular Web Services and .NET. In the rest of this paper, we focus on the use and evaluation of Web Services as the basis for learning complex software development.

Performance evaluation (3 weeks). After completing the discussion of how to design and build software, we spent the remaining three weeks discussing how to evaluate our software artifacts. This topic was touched upon throughout the entire class in an attempt to raise its level of importance in the minds of the students; we now give a more formal treatment to it to wrap up the semester. Topics included the proper use of metrics, and how to write and evaluate simulations.

4. WEB SERVICES IN THE CLASSROOM

The structure of this portion of the class is to begin with the core technologies of Web Services, discuss and evaluate the future of Web Services, and have hands-on programming assignments to reinforce the technologies. The core technologies presented and discussed, in order, were HTTP, XML, SOAP, WSDL, and UDDI. At this point, we had some in-class demonstrations for how to create and consume simple Web Services (respectively, Visual Studio .NET and Internet Explorer). Next, we discussed the architecture of .NET and went in-depth into the run-time structure of the Common Language Infrastructure (CLI). We then covered ASP.NET and ADO.NET, and finally C#.

In order to give the students a little more depth in Web Services, and avoid too much breadth, we chose to focus on Web Services security and .NET security. This started with a "crypto primer", the XML Digital Signature standard, XML encryption, WS-Security [8], and finally the joint IBM-MS Web Services Security roadmap [2] (in this context, we also investigated how the Liberty Alliance [5] is or is not consistent with the roadmap). It was this section of the class, in particular, that we discussed the open research issues in Web Services.

We wanted to get into mobile computing and Web Services, so we also spent some class time and demonstrations on the .NET Compact Framework.

In class, in addition to the technical aspects of Web Services, we discussed the political/business aspects of Web Services and the standardization process, focusing on the W3C [13], OASIS [7] and the Web Services Interoperability organization (WS-I [15]). The W3C and OASIS are standards bodies in which many of the relevant Web Services standards are homed; the mission of WS-I is to get its constituents to agree on *profiles*, which are a subset of the standards, to ensure interoperability.

The first programming assignment was designed to give the students hands-on experience with designing, implementing, and evaluating the *server* side of client-server architectures, particularly in the context Web Services. Thus the first assignment was to create an "Auction Service". The nature of the auction is that, at any time, the Auction Server has zero, one, or more items available for clients to bid on. Sporadically, the auction server will put an item up for bid, where the object has a declared value and a fixed time at which the auction for that item expires. Bidding for each item starts at $0. The current highest bid is not a secret; anyone can ask the Auction Server for the current highest bid for every item. The students were told that their next assignment would be to create the client (hence, they were really designing both the client and the server). To facilitate the *design* process, the assignment itself was underconstrained in terms of absolute requirements. In addition, in this and all programming assignments, a significant portion of the grade involved the *design* of the experimental evaluation of the code. Windows and Visual Studio.NET were recommended but not required (ultimately, every student chose to use Visual Studio .NET). Fault tolerance was a requirement to a certain degree. That is, the students had to at least identify potential faults and the consider the ramifications of such potential problems. Approaches to handling these faults were required to be identified and at least intuitively assess the cost of implementation.

The second programming assignment was to design, implement, and evaluate the Auction Client that interacted with the Auction Service (the professor chose the "best" auction server from the students and used that as the single Auction Server for the entire class). The Auction Client was a standalone entity that requires no human intervention in order to make a bid. However, the Auction Client was required to convey to a human watching the program what the

Auction Client was doing. The nature of the auction as the same as the previous assignment, except that the server does not declare the value of the items. Instead, the client should interact with the Amazon.com Web Service interface [1]. (Amazon.com supported and encouraged us in this endeavor.) Using Amazon.com allowed the students to consume both the Web Service designed for this class and also a pre-existing, commercial Web Service. To make the assignment more exciting, we decided that 10 points of the grade on this assignment would be based on the client's performance in a 24-hour period toward the end of class, where performance was defined as the amount of cash the client had left + the price (from Amazon.com) of the items that were obtained in auction. Students all started with the same amount of hypothetical cash and competed with their fellow classmates, so strategy was important.

5. DISCUSSION

We knew from the original design of the class that we faced a number of issues with the use of Web Services in the classroom for this type of class. We made sure to emphasize that we did not see Web Services as a proprietary technology and that it has broad buy-in. However, perhaps because we only focused on Microsoft tools and platforms out of practical considerations, some students still wrote in the course evaluation that Web Services were, in fact, a Microsoft-only technology.

We had to make careful decisions regarding the material to not really cover, both for the existing tooling for Web Services and the emerging standards. For example, we only briefly covered Web Services in Java (i.e., JAXRPC, EJB, and J2EE) mainly because we did not have the IDEs and SDKs installed/available for pursuing both .NET and a Java-based solution. We also did not get a chance to go in-depth on Web Services on devices (i.e., .NET Compact Framework), although there was clear interest in devices and mobile computing. Regarding the emerging standards, we focused on security because of its obvious, immediate importance. But even within the scope of security, there are a number of competing standards. Without a crystal ball, it is impossible to focus on only those that will emerge. Instead, the approach taken was to present the technical specifications specifically as works-in-progress and subject to change or even disappear completely! Again, in this course in particular, this was reasonable (arguably, some undergraduate courses might not be able to present such "less concrete" ideas so easily).

The two programming assignments as a whole were effective, as determined via discussions with the students and the student course evaluations:

Auction Server The Auction Server assignment was a relatively easy and effective start with Web Services. Many of the students had no experience developing on Windows, so this was their first exposure to Visual Studio (which they enjoyed). We had a shared IIS server that seemed to work well–students would develop from their desktop and push out to the share (a SAMBA mount). The design documents that the students handed in were clearly reflective of the discussion of software engineering principles immediately prior to this assignment. The resulting code was generally good–most students learned some C# (which they were very excited about), and some students learned SQL and/or MS Access. However, some students had not generated sufficient server functionality (e.g., server could crash on bad input, security such as the need for nonrepudiation was not considered, and certain designs were not scalable). In addition, some students still did not think in terms of XML (for example, choosing to return simply a -1 on error or not returning a complex data structure as an XML document). The assignment was designed specifically to be open-ended and underconstrained to facilitate actual design on the part of the students. To many students this was their first real experience with an underconstrained problem in the classroom setting. Again, this was a goal of the class, so this was good—most real-world problems are underconstrained. However, students sometimes had difficulty determining whether to *design* or *implement.* That is, there was simply not enough time to design *and* implement the best Auction Server. Some students questioned whether they should have merely designed everything out and implemented nothing. The proper balance in the assignment requirements must be more precisely defined in the future.

Auction Client The Auction Client assignment gave students additional practical experiences with a new technology and a new methodology (for example, many students developed clients based on the Windows Forms support in Visual Studio .NET). Students also enjoyed designing and experimenting with game-playing algorithms. Unfortunately, because of logistics, the final 24-hour experiment could not be performed (the professor ended up giving each of the students the 10 points). However, in hindsight, this did not doom the assignment because this only impacted the evaluation, not the design of the Auction Client itself.

Overall, the use of Web Services in this type of course was overwhelmingly positive. Occasionally, we would hear students voice concerns that Web Services were "too practical" and "of no importance" to their research. Our response generally boiled down to two points:

1. The course was not about Web Services per se. Rather, the course was about basic principles of complex software development.

2. Even if you never directly use Web Services for your research, it is crucial to keep an eye on the current trends in the community as a whole. Without a doubt, *the* current trend is Web Services.

While we believe that this class has made an impact on last year's incoming students, we can only assess its near-term impact via anecdotal tales from the students and the advisors. The purpose of this class is to prepare a student for their entire graduate career (and beyond). As such, it will take a few more years before we can properly assess its long-term impact.

In the next class, which is being taught this semester, there are small variations we are going to try. We are going to change the sockets assignment to use SSL (because students have shown that they are familiar with sockets programming, are familiar with SSL, but have never used SSL programmatically). We are going to add a programming assignment on basic OS synchronization (synchronization is historically a difficult topic to master). We are going to allow students to develop Web Services in either .NET or a java-based approach. It will be interesting to see how many students choose each. The landscape of Web Services standards have changed considerably, so the course content will be updated (e.g., BPEL). Also, programmatic support for some of the emerging Web Services has started to appear, so these standards may appear in the Web Services assignments (e.g., WS-Security in a java SDK and correspondingly in Microsoft's Web Service Enhancements v2.0). It is not clear at this point if the same Auction paradigm will be used. We are currently working on a “community-based” assignment in which each student produces a service, registers it with a UDDI server (this was missing in the previous incarnation of the Web Services programming assignments), and consumes other UDDI-registered services.

6. CONCLUSIONS

While incoming graduate students are undoubtedly as highly qualified as ever, acclimation to the research environment remains a challenge. There are so many basic skills and methodologies that a student needs throughout a research career that incoming students cannot be expected to possess them all. Recognizing this need, we have designed a course for all first-year graduate students that attempts to fill their basic skill set with a fundamental understanding of how to design, implement, and evaluate software for large, complex environments. The focus of this paper was the use of Web Services as the main “application domain” upon which to investigate common approaches and best practices for software development. The near-term impact of the class has been anecdotally successful and we believe that this experience can ultimately prove to be one of the most important classes taken during a graduate student’s career.

7. ACKNOWLEDGMENTS

The material in this paper greatly benefited from discussions with my colleagues in the Department of Computer Science at the University of Virginia, in particular Jack Davidson, John Knight, Worthy Martin, John A. Stankovic, and Kevin Sullivan. This work is supported in part by the National Science Foundation under a grant from the Next Generation Software program (ACI-0203960) and a grant from the NSF Middleware Initiative program (ANI-0222571). This work is also supported in part from a grant from Microsoft Research.

8. REFERENCES

[1] Amazon.Com. Web services interface and sdk to amazon.com. www.amazon.com/webservices/.

[2] IBM Corporation and Microsoft Corporation. Security in a web services world: A proposed architecture and roadmap, April 2002. http://www-106.ibm.com/developerworks/webservices/library/ws-secmap/.

[3] B. W. Lampson. Hints for computer system design. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP)*, volume 17, pages 33–48, 1983.

[4] R. Levin and D. D. Redell. An evaluation of the ninth sosp submissions or how (and how not) to write a good systems paper. *Operating Systems Review*, 17(3):35–40, July 1983.

[5] Liberty Alliance Project. Open, interoperable standards for federated network identity, 2003. http://www.projectliberty.org/.

[6] Microsoft Coorporation. Shared source common language infrastructure 1.0 release, 2003. http://msdn.microsoft.com/net/sscli/.

[7] OASIS. Organization for the advancement of structured information standards, 2003. http://www.oasis-open.org/.

[8] OASIS Web Services Security Technical Committee. Web services security: Soap message security, 2003. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss.

[9] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, Nov. 1984.

[10] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating Systems Concepts – Sixth Edition*. John Wiley and Sons, Inc., 2002.

[11] A. J. Smith. The task of the referee. *IEEE Computer*, 23(4):65–71, 1990.

[12] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 1998.

[13] W3C. World wide web consortium, 2003. http://www.w3c.org/.

[14] F. Wolfs. Introduction to the scientific method. http://teacher.nsrl.rochester.edu/phy_labs/AppendixE/AppendixE.html.

[15] WS-I. Web services interoperability organization, 2003. http://www.ws-i.org/.