

Software for Everyone by Everyone

Tevfik Bultan
Computer Science Department
University of California
Santa Barbara, CA 93106, USA
bultan@cs.ucsb.edu

ABSTRACT

Given the dizzying pace of change in computer science, trying to look too far into the future of software engineering is hard. However, it might be possible to predict the future of software for the next decade based on the current trends. And based on the predictions on future of software, it might be possible to speculate about the future of software engineering. I try to do such a prediction in this position paper. I predict that the future of software will be applications that will be accessible everywhere, such as web applications and mobile applications. I also predict that increasingly more applications will be developed by non-computer-scientists. The challenges and the opportunities for software engineering research will be in providing tools and techniques that will enable non-programmers to become programmers for everywhere-accessible-software.

Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques—*Modules and Interfaces*; D.2.4 [Software Engineering]: Software/Program Verification—*Model Checking*; H.3.5 [Information Storage and Retrieval]: Online Information Services—*web-based services*

General Terms

Design, verification, human factors

Keywords

Web-based applications, mobile applications, end-user programming, model-driven development, modularity, automated verification

1. INTRODUCTION

Let me state the obvious first. The future of software engineering will be tied to the future of software. So, the first question to answer is what will be the future of software?

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.
Copyright 2010 ACM 978-1-4503-0427-6/10/11 ...\$10.00.

What type of software will software developers write in the future? Second question has to be: Who will be programming in the future? What skills, background, knowledge will the future programmers have? The answers to these questions should guide the software engineering techniques of the future. Based on the future of software and the future programmer, we can try to predict the future software engineering challenges. And, based on these challenges we can make predictions about the future of software engineering.

2. FUTURE SOFTWARE

I suggest the following equation for characterizing the future of software in the next decade:

$$\text{future software} = \text{cloud computing} \times (\text{web apps} + \text{mobile apps})$$

Although the above equation may seem like a random combination of arithmetic symbols and buzzwords, what I mean by it is the following: In the next decade, the majority of the developed software will be either web applications or mobile applications. Cloud computing will be a multiplying factor in increasing the effectiveness, efficiency and availability of these applications, bringing an end to the era of standalone desktop applications that has dominated the software industry since the introduction of the personal computer.

Let's make a simple comparison between a desktop application and a web application. A typical desktop application runs on a single machine and it does not need a network connection to run. It reads and stores its data locally on the machine it runs on. In contrast, a web application runs on a server over the network. It is accessed via internet by a client machine running a browser. A typical web application stores its data in a backend database and stores a minimal amount of data (if any) on the client side.

It should be clear from this simple description that writing a web application is more challenging than writing a standalone desktop application. A web application is a distributed system that involves interaction among multiple components running on multiple computers (the client machine, the web server, the backend database). The complexity of writing such an application should be greater than writing a standalone application that stores and accesses data locally. However, the advantages of web applications compensate for the challenges they create, resulting in a continuing shift towards them in the software industry.

What are the advantages of web applications? A web application is accessible from any machine with an internet connection and a browser. Storing the data in a backend database saves the users the hassle of copying files from

one machine to another machine. When combined with the cloud computing platforms, web applications provide reliable, efficient and scalable access to data from anywhere in the world. Moreover, the users do not have to deal with installation, configuration management, software upgrades and security patches. Software-as-service paradigm enables the developers to seamlessly upgrade and modify the applications without bothering the users.

Software-as-service paradigm becomes even more effective when an application is not only accessible from a desktop computer but also from a smartphone. Increasing use of smartphones such as iPhone and Android phones means that nowadays users have network access even without having access to a computer. Hence, a software that is accessible over the network becomes even more valuable. Although web applications must run on any browser on any computer (since a typical user uses multiple computers), mobile applications typically have customized front-ends (since a typical user uses a single smartphone and since small screens of mobile devices make browser based interfaces hard to navigate). However, these mobile applications with customized front-ends can still access a server over the network and access non-local data.

I predict that in the future most software applications will be accessible through the internet and most applications will be accessible both from a computer and a smartphone. Most applications will use the three-tier architecture [20] consisting of: 1) A client: This is the presentation tier that is responsible for the user interface. It sends user requests to the logic tier and presents the responses back to the user. 2) A server: This is the logic tier that serves the requests from the client by interacting with the data tier, and 3) A backend datastore: This is the data tier that stores the data and interacts with the logic tier to serve the user requests. The client will either be a browser running on a computer or a mobile application running on a smartphone. The logic and the data tiers for most applications will be hosted in a compute-cloud. Since I could not find a better term, I will call these everywhere-accessible-software-applications *everyware* applications.

3. FUTURE PROGRAMMERS

The reason Apple can claim that “there is an app for anything” is not because Apple is developing iPhone applications for everything. The availability of thousands of iPhone applications at the Apple App Store is due to the fact that Apple provides the iPhone software development kit (SDK) [1] to anyone interested in developing an iPhone application and helps them in marketing their software at the Apple App Store. Android SDK and Android Market [9] do the same for mobile applications for the Android platform. There are thousands of developers who are writing software applications for these platforms since barrier to entry to these software markets is very low.

A similar trend also holds for social networking applications targeting the Facebook social network. Software developers can write Facebook applications based on the publicly provided Facebook API [7] and their application would be accessible to Facebook users.

At the same time, it is becoming easier to write and launch web applications due to availability of cloud computing technologies. A web application developer does not have to worry about where to store a web server, how to

maintain it, how to handle high volume traffic, etc. Cloud computing platforms provide servers that can be allocated, used and maintained easily. For example, Google App Engine [10] enables software developers to write and run web applications on Google’s infrastructure.

The low barrier to entry for writing everywhere applications will encourage an increasing number of people to try to develop their own applications. It is likely that a substantial number of these people will try to implement the applications based on their ideas themselves even if they do not have a computer science background.

Combination of these factors will encourage many non-programmers to become programmers. And, eventually such programmers will significantly outnumber programmers with a computer science background. In fact this is already true [23] if one considers end-user programming for spreadsheet systems and databases. Everywhere application development will just accelerate the rate and level of end-user programming, increasing the need for software engineering for end-user programming [3].

Let me digress a little, I do not think it is unreasonable to predict that *everyone* will be a programmer in the future. Programming computers should be a natural activity for humans since we are naturally equipped with computing machines called brains. However, unlike many fields of knowledge, education in computing is mainly provided at the college level. The lack of education of basic computing concepts in K-12 education is probably an anomaly due to the youth of computing field. In the long term, one can reasonably expect that every high school graduate will know basic concepts of computing and will be literate in computer programming. However, even before this long term transition, I expect that we will see a continuous increase in programming by non-computer-scientists.

4. FUTURE CHALLENGES

The two trends I mentioned above, increasing development of everywhere applications with increasingly less experienced developers, will amplify a large stumbling-block that is faced by software developers today: Everywhere applications are not dependable. For example, web applications are known to consistently mishandle unexpected user actions caused by unanticipated use of a browser’s back-button or multiple browser windows [13]. Web applications are also notorious for security vulnerabilities that can be exploited by malicious users [19]. Given their increasing importance and ubiquity, the lack of dependability in everywhere applications will be the most significant challenge faced by software engineering researchers in the future. Below, I will elaborate on several features of everywhere applications that make establishing their dependability extremely challenging.

Security.

Since they are accessible from everywhere, security of everywhere applications is extremely important. Ease of access is an advantage when it is considered from the perspective of the legitimate user. However, it also means that everywhere applications can be targeted by malicious users from all around the world. Even today web applications are used for mission-critical tasks and frequently handle sensitive user data. In the near future, web applications will play a significant role in improving the efficiency of national infrastructures in many critical areas such as healthcare [12], na-

tional security, and the power grid [11]. Hence, security is an extremely important concern for these applications. If increasingly inexperienced programmers develop everywhere applications in the future, then we can expect the security vulnerabilities to become even more significant in the future.

Interaction.

Everyware applications involve many types of interactions. As I mentioned above, a typical web application is implemented using a three-tier architecture, where the application consists of (a) client-side code that executes at the user machine through the browser, (b) server-side code that executes at the server machine and handles the user requests, and (c) a backend database server that stores the persistent data. Moreover, each of these tiers consists of a system stack made up of layers of abstractions (in hardware and software) that are built on top of each other. They include the system hardware, operating system, virtual machines and runtime systems, browsers, database engines, and application components. There are multiple interactions among the tiers and the components within the tiers, and multiple interactions between different layers of abstractions on the system stack. Understanding and managing these interactions presents a significant challenge to any programmer.

Diversity.

Everyware applications will be developed using a diverse set of languages and technologies. For example, each tier of a three-tier web application can contain one or more languages including Java, Perl, PHP, Python and Ruby on the server side, HTML, XML, and JavaScript at the client side, and SQL and XQuery on the backend database (which itself is typically implemented in Java or C++). Moreover, it is common to have interactions among different applications written using different development frameworks. This diversity is not a transitional state that will eventually converge to a single dominant language. Diversity is an essential feature of everyware applications that is likely to increase in the future given the benefits of different languages and frameworks for different uses, and the increasing diversity in backgrounds and preferences of software developers. Increasing use of mobile applications will add to this diversity even more.

Changeability.

Everyware applications are updated frequently since the code resides on the server side and can be updated without user participation or knowledge. This is a very convenient feature for application developers. When a bug is found and fixed, the fix can be immediately uploaded to the server and every user who uses the application after that point will use the new version of the software. In fact the use of frequent, incremental updates is now ingrained in the software process models used by the web application developers. The agile processes [2, 4] that have been widely adopted for web application development encourage frequent and incremental updates. Hence, dependability of everyware applications must be investigated assuming frequent updates and many versions.

5. FUTURE OF SOFTWARE ENGINEERING

In this section, I would like to speculate on future research

directions for software engineering for addressing the challenges I discussed earlier.

Models.

Model driven software development has been investigated in the software engineering community for quite a while [18]. In the web application development domain, WebML provides a good example for model driven development [5]. In the future, I expect that the model driven development will become the mainstream approach used in software engineering research. Model driven development will be extended to model driven programming where the model is embedded in the code and becomes the code itself.

Model driven development will provide two important benefits: 1) Models will enable analysis to improve dependability, 2) Models will help inexperienced programmers to specify the behavior at a higher level of abstraction, minimizing the errors that are due to misunderstanding and misuse of low level language primitives.

Eventually, model based development may become a programming approach that is accessible to everyone, mirroring the success of languages such as Scratch [21] for introducing children to programming. In the long term, computer scientists will need to figure out how to teach basic computing and programming skills to everyone. Software engineering research can and should take a leading role in this area.

Automation.

I predict that automation will become increasingly important for software engineering. Automation will be essential in many contexts such as automated code generation from high level models, automated testing and verification, automated debugging, etc. Automation is crucial for accommodating developers without much programming experience. There may be tasks that cannot be automated, such as identifying the requirements for an application, designing the data model or the navigation flow of a web application. However, many tasks, such as mapping the navigation steps to scripts in a scripting language might be automated for many applications. The interesting research question is separating automatable tasks from the ones that require user involvement.

Modularity.

Given the complexity of everyware applications, it is essential to have modularity in design and development. For example, in a typical web application, a user interacts with a sequence of forms. Each form has an associated server-side script, usually written in a dynamic scripting language such as PHP, Python, or Ruby. While serving the user request, a script can generate queries for the backend database in a query language such as SQL. When it completes serving the request, the script generates an HTML page to be sent back to the user. Moreover, on the client side there could be JavaScript code executing and interacting with the server side scripts. It is not feasible to develop an application with such a set of complex interactions without using some kind of modularity.

Most web applications use a script-oriented programming style. Building a large application as a collection of scripts is difficult. In recent years, an increasing number of web applications are being developed based on the Model-View-Controller (MVC) design pattern [16, 8], using frameworks

such as Zend for PHP [25], Django for Python [6], Spring for J2EE [24], and Ruby on Rails [22]. The MVC design pattern brings structure to the collection of scripts that constitutes a web application by separating the parts that are responsible for the control flow logic (the controllers), the business logic and the data model (the model) and the user interface logic (the views). Controllers consist of actions (i.e., scripts) that process the incoming requests by querying and modifying the data model. When the processing is done, the next page is rendered using the next view.

The main advantage of using MVC-frameworks is modularity of the architecture. Properties related to navigation are handled mainly by the controllers, and properties related to the data are handled by the model, and the user interface is handled by the views. In addition to making software development easier, this separation of concerns also presents an opportunity for effective automated verification and analysis. The research question is: How can modularity at the design level be better integrated with the verification techniques that depend on it?

Verification.

Both concerns about security of everywhere applications and increasing programming by inexperienced programmers will make automated verification and analysis crucial areas for future of software engineering.

A key part of effective verification is the ability to exploit the modular structure of the software. If a software developer divides a software system to modules without restricting the possible interactions between them, this type of modularity will not be useful during verification since a sound verification tool has to consider *all* possible interactions. Actually, while developing a module, a software developer makes assumptions about the behaviors of the other modules. The problem is, these assumptions may not be expressed in any part of the code because 1) it takes significant effort to specify such assumptions precisely, 2) programming languages may not support specification of such assumptions.

I believe that the key to solving this problem will be 1) providing effective tools and techniques for specification and analysis of interfaces, 2) isolating the behavior of interest based on these interface specifications in order to achieve scalable modular verification that exploits the inherent modularity in everywhere applications. These interface specifications will be tightly integrated with the models used for characterizing the behavior of the application.

For example, with an appropriate modeling language and appropriate interface specifications, it might be possible to automatically extract a data model from a web application and analyze it. One can use bounded verification techniques to check properties of the data model within a certain bound [15]. It might also be possible to extract a navigation model from a web application [17] and analyze its properties using state space exploration techniques [14]. However, an important goal has to be integration of the verification task to the software development process. Verification should not be a separate step that requires reverse engineering of the artifacts produced during development. It should be part of the design process to eliminate the errors as early as possible.

6. CONCLUSIONS

Future software applications will be accessible from browsers and smartphones. They will reside on the cloud and will function as globally accessible services. Low barrier to entry to application development will create many software developers without computer science background. The future of software engineering research must address the challenges that will be created in such a future. I believe that model-driven-programming combined with automation, modularity and verification techniques can address those challenges.

Although there are notable exceptions, so far in its 40-plus years of history, software engineering research has mostly been either reactionary (for example reacting to developments in other areas such as programming languages, operating systems and networking) or too academic and hard to adopt in practice (for example a lot of the research on formal methods). The future of software provides a unique opportunity for the software engineering research to take a leadership role in the field of computer science. Facilitating the massive amount of programming for the everywhere applications of the future by inexperienced programmers will require contributions from all areas of software engineering research, including research on software design, modeling and analysis, verification, testing and debugging, formal methods and empirical studies. The future of software engineering will be exciting.

Acknowledgments. I would like to thank Chandra Krintz, Ben Hardekopf and Timothy Sherwood for their input on the challenges in building Web applications, and I would like to thank anonymous reviewers for identifying some of the related work.

7. REFERENCES

- [1] Apple. iphone software development kit (sdk). <http://developer.apple.com/iphone/index.action>.
- [2] K. Beck. Embracing change with extreme programming. *IEEE Computer*, 32(10):70–77, 1999.
- [3] M. M. Burnett. What is end-user software engineering and why does it matter? In *2nd International Symposium on End-User Development*, pages 15–28, 2009.
- [4] by Ken Schwaber and M. Beedle. *Agile Software Development with Scrum*. Prentice Hall, 2001.
- [5] S. Ceri, P. Fraternali, and A. Bongio. Web modeling language (webml): a modeling language for designing web sites. *Computer Networks*, 33(1-6):137–157, 2000.
- [6] Django framework. <http://www.djangoproject.com/>.
- [7] Facebook. Facebook apis. <http://developers.facebook.com/>.
- [8] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP 93)*, pages 406–431, 1993.
- [9] Google. Android. <http://www.android.com/>.
- [10] Google. Google app engine. <http://code.google.com/appengine/>.
- [11] Google powermeter. <http://www.google.org/powermeter/>.
- [12] Google health. <http://health.google.com/>.

- [13] P. T. Graunke, R. B. Findler, S. Krishnamurthi, and M. Felleisen. Modeling web interactions. In *Proceedings of the 12th European Symposium on Programming (ESOP 03)*, pages 238–252, 2003.
- [14] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Eng.*, 23(5):279–295, May 1997.
- [15] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.
- [16] G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.
- [17] A. Kubo, H. Washizaki, and Y. Fukazawa. Automatic extraction and verification of page transitions in a web application. In *Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC 2007)*, pages 350–357, 2007.
- [18] S. J. Mellor, A. N. Clark, and T. Futagami. Guest editors’ introduction: Model-driven development. *IEEE Software*, 20(5):14–18, 2003.
- [19] OWASP. Top ten project. <http://www.owasp.org/>, May 2007.
- [20] A. O. Ramirez. Three-tier architecture. *Linux Journal*, 2000(75), 2000.
- [21] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. S. Silver, B. Silverman, and Y. B. Kafai. Scratch: programming for all. *Commun. ACM*, 52(11):60–67, 2009.
- [22] Ruby on rails. <http://www.rubyonrails.org/>.
- [23] C. Scaffidi, M. Shaw, and B. A. Myers. Estimating the numbers of end users and end user programmers. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2005)*, pages 207–214, 2005.
- [24] Spring framework. <http://www.springsource.org/>.
- [25] Zend framework. <http://framework.zend.com/>.