Situated Software:
Concepts, Motivation, Technology, and the Future

Sriram Balasubramaniam, Grace A. Lewis, Soumya Simanta, and Dennis B. Smith, Software Engineering Institute, Carnegie Mellon University

Opportunistic software systems are created when a user combines two or more software systems to satisfy specific needs that weren't foreseen when the individual systems were developed. These software systems usually aren't complete or comprehensive because they're meant to address specific situations or use cases. They're often lightweight composite applications that might be hardened over time to support additional growth, features, and capabilities. Because the individual software systems usually aren't built for use in different contexts, integration is challenging.

Opportunistic software development is nothing new. Shell scripts and Excel spreadsheets that let users quickly automate tasks specific to their needs have been around for a while. Over time, one or more users identify a pressing need for specific functionality that the software applications their enterprises use don’t provide. Even if the functionality exists, it’s not present in the way the users want to use it. So, they go about implementing that functionality using a tool such as Microsoft Excel.

Situated software, a term made popular by Clay Shirky, is a form of opportunistic software that a small subset of users create to fulfill a specific purpose. In Shirky’s words, “Situated software is designed in and for a particular social situation or context.” In contrast to standard software development, situated software usually isn’t constructed by a team of traditional software developers. Instead, it’s created by users who desire specific functionality that mainstream software doesn’t provide. Typically, it’s personalized, localized software that has evolved organically and has been created by the community that uses it. Bonnie Nardi reports that users frequently share customization files, and that this often applies to situated software, too. Users who customize software to suit their specific needs often end up sharing their customization efforts with a like-minded community.

In this article, we focus on situated software as an example of opportunistic software and as a potential means for helping users significantly improve their ability to use software and data to address pressing business needs. We provide examples of situated software’s use, trace how the Internet contributed to its rapid evolution, outline its role, describe situations where it’s appropriate to develop software opportunistically, discuss its limitations, and outline enablers for its adoption.

The Context for Situated Software
The advent of the Internet and the recent explosion in Semantic Web applications’ availability has made it easier for users to develop situated software. The increasing availability of XML (www.w3.org/TR/xml) data feeds (using protocols such as RSS [www.rssboard.org/rss-specification] and Atom...
[www.atomenabled.org/developers/syndication/atom-format-spec.php] has made it easier for small groups of people to consume these data feeds in ways that were never envisioned before. This has led to new classes of user-generated applications that are examples of situated software. The situated-software model departs from the more traditional and established application-centric computing model that was, and continues to be, the norm.

In a sense, it’s as if data has been liberated from the confines of the IT department. Users are becoming more aware of the power of flexible data use to serve business needs in diverse ways. This new inflection point also spans multiple mediums, as evidenced by the use of an increasing number of mobile, wireless, and set-top devices to both build and access situated-software applications.

Another term related to situated software is Web 2.0,4 which refers to “changing trends in the use of World Wide Web technology and Web design that aim to enhance creativity, information sharing, collaboration, and functionality of the Web … [and] have led to the development and evolution of Web-based communities and hosted services, such as social-networking sites, video-sharing sites, wikis, blogs, and folksonomies” (http://en.wikipedia.org/wiki/Web_2.0). While this definition of Web 2.0 focuses on the user and social perspective, note that Web 2.0 also exhibits the notion of the “software as a platform” paradigm by providing data feeds and application programming interfaces (APIs), thus enabling developers and users to extend and customize Web 2.0 applications. A Web 2.0 application typically consists of one or more of the following capabilities:

- provides and facilitates customization, personalization, collaboration, and data sharing between users;
- encourages the use of data feeds, which are typically provided in XML format;
- provides multiple APIs in a variety of programming technologies, thus allowing software developers to choose the technology and programming language;
- promotes the extraction of partial functionality and data;
- provides a rich set of interaction widgets, which serve as user-interface building blocks for developing and personalizing the user experience; and
- often provides source code, which lets software developers modify sample applications to suit their needs and also extend the software to support additional functionality.

These technologies provide tools that let users combine subsets of functionality from different applications to build entirely new applications. Online applications such as Dapper (www.dapper.net) and Data Mashups (www.datamashups.com) are examples of this new phenomenon. These Web sites let users quickly combine data, user interface widgets, and APIs from multiple Web sites to create and host a new composite Web application. Such applications are called mashup applications. Mashups have gained traction because of the availability of tools and APIs for extracting and manipulating data in software applications. Because of this ability, software applications transcend their authors’ original intentions by letting developers and users use the data from their applications in new and different ways.

This capability to promote the use of application data in new and different ways lets users create opportunistic software applications because they can extend and enhance the software’s functionality. Software applications’ ability to provide extensibility, customization, APIs, and data feeds is called the “software as a platform” paradigm and has been instrumental in users adapting software for their contextual needs.

Richard Veryard identifies the need for such types of situated applications and suggests that an enterprise needs to use lightweight methods to more effectively serve the users’ needs.5 He specifically proposes mashup technology as one way of generating flexible solutions to user problems. Other researchers have claimed that because mashups are created without access to the engineers who created the services, they can reflect the social and pragmatic nature of user demands that focus on the actual use that’s required in practice.6

**Mashups: An Example of Situated Software**

According to Wikipedia, “A mashup is a Web application that combines data from more than one source into a single integrated tool; an example is the use of cartographic data from Google Maps to add location information to real-estate data, thereby creating a new and distinct Web service that was not originally provided by either source” (http://en.wikipedia.org/wiki/Mashup_web_application_hybrid).

An example of a mashup created on ProgrammableWeb (www.Programmableweb.com) is the AP National News and Google Maps application (www.81nassau.com/apnews). This application combines the latest news feeds from the Associated Press and plots the locations of the cities referenced
by the news articles onto a Google Map, as Figure 1 shows.

Figure 2 shows a hypothetical mashup application. In this example, three independent applications provide mapping services, real estate listings, and school district information. The combination of these three distinct sets of applications can result in a mashup application that displays homes for sale and school district data on a single map. This enables flexible user programming in response to new needs or demands, and it illustrates an example of “power to the edge,” in which decision-making ability is placed as close to the ultimate user as possible.

Types of Situated Software Users
We can typically classify users who participate in situated-software efforts as belonging to one or more of the following categories:

- Nontechnical business users who are early adopters from a business perspective rather than a technical perspective. They require new features and additions to the software’s capabilities to solve immediate business needs, and they can’t afford to wait through a traditional software development cycle. An example would be an executive who puts together a self-service dashboard application for spotting sales trends in a way that makes sense to the executive. Business users of successful dashboard applications are usually enthusiastic supporters of situated software and welcome additional features and customization capabilities.

- “Lego-like” situated-software assemblers, who mix and match multiple applications and data feeds to produce situated-software applications. They use lightweight tooling and open standards and tools such as Yahoo Pipes (www.yahoo.com/pipes), ProgrammableWeb, or even Microsoft Excel to design and subsequently reuse a series of macros that fulfill a specific business context. Programmers who use mashup technologies to build situated software are examples of this user category. These programmers combine the capabilities of one or more applications by leveraging APIs, data feeds, and mashup tools to produce situated software. Overlaying the addresses of potential customers on a Google Map is an example of how a customer relationship management (CRM) application’s functionality is combined with mapping software’s functionality to develop software that’s context specific.

- Traditional software developers, who, apart from building and maintaining situated-software components and interfaces, are power users of situated software. One example is a software developer who uses a mashup tool such as Yahoo Pipes in conjunction with Java code to extract, transform, and render data from multiple Web applications to develop an application for monitoring and displaying news items and articles on technology topics.

Especially in the first category, the important thing to note is the organic evolution of the software creation process. The users creating the software might not have a background in software development. They’re not building the software on the basis of classic enterprise software development requirements such as security, scalability, and fault tolerance. What they’re seeking to do is implement functionality that they need using a tool that they understand. We can think of this phenomenon of users creating software that fulfills a specific need as the long-tail effect in action—the use of a large number of situation-specific applications compared to the use of the main enterprise applications.

(Long tail refers to statistical distributions in which a high-frequency or high-amplitude population is followed by a low-frequency or low-amplitude population that gradually “tails off” asymptotically. The events at the far end of the tail have a very low probability of occurrence [http://en.wikipedia.org/wiki/The_Long_Tail].)

Situated software can start out as a series of dis-
ruptive efforts that end up serving unserved consumers. These unserved consumers get the benefits of flexibility and quick access to new functionality. They become advocates for situated software and understand the need for such software to exist and thrive. This is the context for the real benefits that can come from situated software. But, it also has limitations in both scale and generality, as we explain in the following sections.

**Situated Software’s Uses and Limitations**

We could argue that situated software’s success lies in its ability to accommodate change and remain flexible and adaptive, as opposed to traditional software, which isn’t very forgiving and which operates in a well-defined boundary, is costly, takes significant effort to develop and maintain, and can be difficult to modify.

However, although the motivation and some of the technology is available to construct software opportunistically, it’s not necessarily a universal approach. Flexibility comes at the expense of quality attributes such as performance, security, and availability. As an example, most reuse efforts and techniques focus on reuse at design time. In the case of approaches such as service-oriented architecture (SOA) or free and open source software (FOSS), developers at design time discover services and FOSS products, respectively, to provide portions of functionality. Software product lines are based on a rigorous architecture and a disciplined process to build reusable core assets as well as product variants.

We could view situated software as reuse at runtime, where there’s more flexibility at the expense of the rigor required for reuse at design time. Typically, there’s little governance overhead associated with situated applications because they’re created (at least initially) by a small community of users to meet a specific demand, and those users don’t have to interact with multiple teams or negotiate service-level agreements (SLAs). This type of approach to software development therefore favors situational software applications that are very data-centric and nontransactional in nature, as reflected by the situated-software applications that exist today.

The following are examples in which situated software is appropriate:

- **Quick rollout to users.** Situated-software applications are valuable in scenarios that require quick rollout of some unanticipated functionality. Typically, situated-software applications consume services and data in an opportunistic fashion and on a need-to-use basis. They wouldn’t likely have formal SLAs with service providers. Lack of such formal agreements between the providers and the consumers might raise concerns regarding trust, availability, security, and reliability. For example, a mashup that uses multiple free services may not function properly if some of the underlying services are unavailable. Therefore, an opportunistic approach that provides functionality quickly to users might not be appropriate for developing mission-critical software applications in which qualities such as availability and reliability are important.

- **Short-lived and transient applications.** Many transient problems exist where it’s unlikely that an organization would invest in enterprise-scale applications. Situated-software applications are one solution to such usually short-lived problems. As Shirky says, “By relying on existing social fabric, situated software is guaranteed not to work at the scale [regular] apps do, but for the same reason, it can work in ways [regular] software can’t.” Rarely does situated software grow and evolve like traditional software. But if it does, it’s appropriate to invest in more robust and scalable applications that provide this reoccurring functionality.

- **Data-centric applications.** The majority of technologies to support situated software are targeted toward the amalgamation of data from several diverse data sources. Providers are usually unaware of the various contexts in which their software will be used. If the situated software is built to perform other operations on data, flexibility is lost and back-end applications must be hardened to deal with users who, as we mentioned earlier, might not have software development backgrounds.

- **Prototyping of potential new functionality.** The hands-on participation of users and quick roll-out time of these applications enable an early demonstration of feasibility by avoiding the
Situated software's true success is in users' desire and willingness to create situated software.

Enablers for Situated Software Development

An opportunistic software development approach brings an enterprise many benefits. On the Internet, the creators of situated-software applications are technically savvy users who want to explore new ways of solving their problems in specific contexts. The advances in technology, publicly available Web services, and infrastructure tools have provided an environment that lets users create such applications. (The most common form of SOA is that of Web services, in which service interfaces are described using the Web Services Description Language [WSDL], payload is transmitted using SOAP over HTTP, and, optionally, UDDI is used as the directory service.) Situated-software applications are more suitable in an environment that’s open and provides choices, suitable tools, and infrastructure to application creators to facilitate situated-software development. Situated software has several key enablers, and we describe them in the next sections.

Web-Based and Service-Oriented Environments

Situated-software applications are easier to develop in an open environment that supports service orientation and is Web enabled. Most relevant examples of situational-software applications have come from Web 2.0 and are based on Web services with open interfaces. Currently, no universal standards exist for these services on the Internet. However, these data services are restricted to a fairly small set of popular standards—such as XML, SOAP, REST (Representational State Transfer), and RSS—that are practical and easily implementable with the proper tool support. Service orientation abstracts users from the underlying complexities and details of how the service provider implements a particular functionality. For example, a user creating a mashup using Google Maps isn’t aware of any underlying complexities. If enterprise software applications expose their data as usable APIs and services, this will let users view and integrate data from various sources to meet their specific requirements.

End-User Computing Tools

Toolkits, languages, and techniques that make it easier for nonprogrammers to create these applications are important for the successful proliferation of situated software inside an enterprise. Tools such as Marmite, MashMaker, and Anthracite (http://metafy.com/products/anthracite) are a good starting point in this direction.

Infrastructure That Supports Reuse

Users often face similar problems, even if each problem requires a slightly different solution. The ability to learn and reuse existing software is valuable, saving time and effort. For example, a tool similar to Yahoo Pipes or Google Mashup Editor (http://editor.googlemashups.com) lets users copy and modify existing software that others have created.

Discovery Mechanisms

Such mechanisms, such as a searchable shared service repository, let situated-software creators find services and APIs. These discovery mechanisms should be not only easily accessible but also intelligent enough to support discovery relevant to the user’s context. Given the agility expected of this environment, if the users developing the situated-software applications can’t find the desired services quickly, they’ll either abandon their ideas or try to recreate the missing functionality. The discovery mechanism should only allow providers to register their services while also letting users share their composite services and applications.

Culture, Adoption, and Training

Situated software’s true success is in users’ desire and willingness to create situated software. All the enablers we discussed earlier are important. But unless the users see real value in situated software, they won’t adopt it. This requires a cultural shift. Training programs and other incentives can be a good starting point. Software platforms for creating situational software are getting easier. However, that doesn’t mean that users are creating and using more situational software. We need to encourage and reward the development and use of situational software.

Despite the benefits we’ve described here, situated software has significant limitations. It’s not appropriate for classic enterprise software that has quality attribute requirements for security, availability, or performance. It’s also not appropriate for large communities of diverse users or for situations that involve complex validations or significant transaction processing. It would be a costly mistake to take a specific situational-software solution and assume that you
could harden it quickly into an enterprise application. Companies such as Google and Amazon offer programmable platforms for building situated software on a large scale. Companies such as Salesforce.com that offer hosted applications using the “software as a service” (SaaS) model provide robust up-time and availability. The key differentiator between public companies and enterprise IT departments offering programmable software platforms might be the ability to control and customize the software. IT departments can be more responsive and malleable than public companies. Enterprise users can also influence the situated-software platform’s direction and future capabilities by interacting with an internal department. This isn’t the case with the offering from a public company, which follows its own product update cycles and can’t let every user influence changes to technology and capability.

Significant research questions need to be addressed in order to understand situated software’s use. These include expansion and refinement of the categories of users and types of applications that we’ve presented here, analysis of new types of applications in this rapidly expanding field, development of case studies across a broad set of user types and organizations, and development of generalizations backed by empirical data. It’s especially appropriate to focus on areas in which situated software has proven to be successful, as well as on its limitations. Failures and partial successes can contribute as much as or more than success stories can to our understanding of situated software’s uses and limitations.

References


For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.