Understanding Service-Oriented Software

Nicolas Gold and Andrew Mohan, *UMIST*
Claire Knight, *Volantis Systems*
Malcolm Munro, *University of Durham*

Many hail service-oriented software as the next revolution in software development. Web services' capabilities are constantly expanding from simple message passing toward the construction of full-fledged applications such as those envisaged by the UK's Pennine Group in their Software as a Service (SaaS) framework.[1]

These new, service-oriented approaches appear to many to solve the significant issue of software inflexibility that arises during maintenance and evolution. While they address some aspects of the problem, however, understanding the software still poses some difficulty. This shift toward service orientation compels us to consider its implications for software understanding, which is potentially the primary cost in software engineering.[2]

Using an example of on-the-fly software services construction, we discuss the problems software engineers still face when working with service-oriented software. We also introduce some new issues that they must consider, including how to address service provision difficulties and failures.

Service-oriented software lets organizations create new software applications dynamically to meet rapidly changing business needs. As its construction becomes automated, however, software understanding will become more difficult.

The service-oriented vision

Software evolution still poses a significant problem for many organizations despite new development methods that promise to enable flexibility and simplify systems' evolution as business needs change. Among the largest costs is the time software engineers spend trying to understand existing software, either to fix bugs or add functionality. We use the term *software understanding* to mean the application of techniques and processes that facilitate understanding of the software. We need this understanding to ensure the software evolves through the application of various maintenance activities.

The SaaS framework, advanced as a solution to the evolution issue,[3] automatically discovers fine-grained software services, negotiates to acquire them, and composes, binds, executes, and unbinds them. This process potentially occurs for every execution of the software, and would thus alleviate evolution problems because there would be no system to maintain—it would be created from a composition of services to meet particular requirements at a given time. The SaaS approach includes elements of *outsourcing* (providing business functions at a given price under a service-level agreement) and *application service provision* (renting complete software applications from another organization). However, it goes further than both ideas.

On the surface, although SaaS appears similar to ASP, it differs in the provision granularity and supply network size. SaaS coordinates the composition of fine-grained, customized services as opposed to the ASP approach's larger-grained, more standardized applications. Also, whereas the ASP supply network typically pairs one customer with one supplier, the SaaS approach deploys a far larger supplier network that aggregates services into increasingly larger units until it delivers the top-level functionality requested.

We envisage a micropayment approach based on service invocation that lets customers pay only for what they need and when, with price reflecting marketplace supply and demand. The marketplace would need to integrate payment mechanisms.

Current Web services technology (see the "Service-Oriented Technology" sidebar) can support some of this vision's lower-level aspects, and new initiatives to define workflow and composition languages will be capable of supporting some of the higher-level elements. Nonetheless, many problems remain to be solved—for example, negotiation to obtain a service and trust in a particular service or supplier.

Any service supply chain depends on establishing trust between the parties involved. When relationships first form between organizations, contract warranty and redress terms compensate for any lack of trust. As the relationship matures, trust accrues and future contracts become easier to negotiate. Since such issues occur in traditional outsourcing, we can reasonably expect them also in a service-oriented architecture.

Managing trust within the automated procurement process SaaS proposes will be more difficult, however. Automatic methods for negotiating such nonnumeric and human-oriented concepts will require further research before they're sufficiently mature to be incorporated into everyday business practice. Also, any framework of warranty and redress must be legally enforceable, another significant challenge for an automated and global solution. Selecting a legal framework within which to form contracts could thus be both crucial and difficult.

Although the trust issue might prove difficult to resolve for an automated service acquisition's initial instance, both system and user experiences with a particular service provider can inform subsequent negotiation. The nego-

Service-Oriented Technology

Many views exist of the technologies that can be used for service-oriented software. These range from all-inclusive applications sold via application service providers to specific pieces of code or components. The latter description usually means that the term is being used as a replacement for Web services. The W3C (www.w3c.org) provides a good generic definition of a service-oriented architecture (SOA) as a set of components that can be invoked and whose interface descriptions can be published and discovered. A Web service is a specific instance of a component (or components) that has a public interface defined and described in XML and that other systems can discover and use by passing messages transported via existing Internet protocols.

The term *service-oriented* has now been applied to the older technologies of DCOM and CORBA, more recently to J2EE and .NET deployments, and of course to Web services. There's no reason why the technology has to be a discriminating feature in a SOA. Standards such as SOAP for Web services help to ensure that heterogeneity of solutions poses no problems.

Layering of architectures is also possible. Many J2EE applications can interface on the company side with the legacy software systems still in use. In turn, the J2EE application can be partially exposed for B2B transactions using Web services to remove the need for both sides to use Java technology.

Versioning helps ensure that organizations can use different versions of services over time in a compatible manner. For example, the .NET framework uses versions of assemblies (collections of classes) (in C#). These can then be used in code with various patterns matching the files required. This permits numerous approaches to version specification such as a specific version of a service, versions having the same major numbers, or a given version or newer. Differently versioned service instances can coexist, and differentiation can be made at runtime and the appropriate assembly used.[1]

The ability to layer solutions and support heterogeneity allows for gradual migration to service-based solutions. The development of XML-based languages for defining and enforcing service-level agreements, workflow, and service composition is supporting the gradual change of business processes, envisioned as part of the growth of Software as a Service.

Reference

1. B. Wagner, "Implementing Versioning in .NET," *Visual Studio Magazine*, vol. 12, no. 3, Mar. 2003, www.fawcette.com/vsm/2002_03/magazine/columns/desktopdeveloper/default_pf.aspx.

tiating agent (such as an automated broker) can use such factors as a service's promised versus actual processing time, user satisfaction ratings, or price comparisons to inform its negotiation strategy and update its profiles of service providers and their offerings. Because the organization must trust the negotiating agent to negotiate on its behalf, this functionality will likely remain in-house, to preserve organizational control over it.

The issues we've raised here are common where business functions are contracted out to another organization. Adding automation might introduce complexity, but we see feasible solutions for the automated domain. Market segmentation along national lines, for example, would facilitate legal-framework solutions. These and other SaaS-related issues are discussed elsewhere.[1,3,4]

In short, the SaaS approach will require both new (though not radically new) business models and new technologies to be successful. Migration to this approach will not be a "big bang" process but rather gradual, with organizations wrapping their existing offerings as services and gradually decomposing them when market opportunities appear for value-added functionality both within and outside their organization. General Motors has adopted such an approach toward build-to-order manufacturing.[5] Internally, the opportunity exists to increase organizational information systems' flexibility and adaptability—the internal market will likely develop first because the complexity of automatic contract negotiation is less important. Externally, the opportunity exists to generate revenue from existing software and to flexibly and rapidly obtain new software without the burden of ownership. Starting with existing systems increases the potential return on investment and decreases the migration risk.

The SaaS approach's relevant key concepts include

- An open marketplace for services
- Dynamic provision of software in response to changing requests
- The potential for one-time execution followed by unbinding
- A services supply network where service providers may subcontract to provide their services
- Delivery transparency to software users, whose interest lies in its use

A scenario

To illustrate some of the problems the software comprehender faces in a SaaS world, we use the fictional example of a large company, Bizness plc, which operates in several countries and thus must produce its quarterly reports in several languages.[4] Bizness plc has its own in-house IT department.

John, a Bizness plc executive, wrote the latest quarterly report in English and wants to submit it for automatic translation. He requests automated translation services for French, German, Italian, and Spanish from Bizness plc's automated broker. The broker searches the marketplace for suitable service compositions that meet John's needs (the composition description doesn't, however, bind the request to actual services). Once it has procured one, the broker searches the marketplace for organizations offering suitable services, negotiates the supply of these services using Bizness plc's predefined policies for negotiation, and binds the contracted services together for John to execute.

Figure 1 shows the supply network formed. John won't necessarily know which companies comprise it, because he only interfaces with his automated broker, and the broker itself might only see the suppliers it contracts with directly. As Figure 1 shows, providers F, G, I, and S will fulfill John's request. G and S provide their complete service in-house without having to subcontract further, whereas F and I have subcontracted for grammar and dictionary information to providers FG, and ID and IG, respectively. However, John doesn't (nor should he need to) know this.

John submits his document to the translation service provided. When he receives the results, he finds that the Italian translation hasn't taken place. He needs to understand why this happened and what changes to make to ensure it works now and for future service requests.

Initially, Bizness plc would seek an explanation from the failed service's supplier. In an automated domain, one of numerous predefined responses might provide sufficient information or appropriate action. Determining the explanation's veracity could prove tricky (although not unique to this domain), particularly if it didn't require much reparation from the supplier.

If the supplier in question provided no explanation, Bizness plc (and its suppliers in the supply chain) could take legal action to en-

The SaaS approach will require both new (though not radically new) business models and new technologies to be successful.

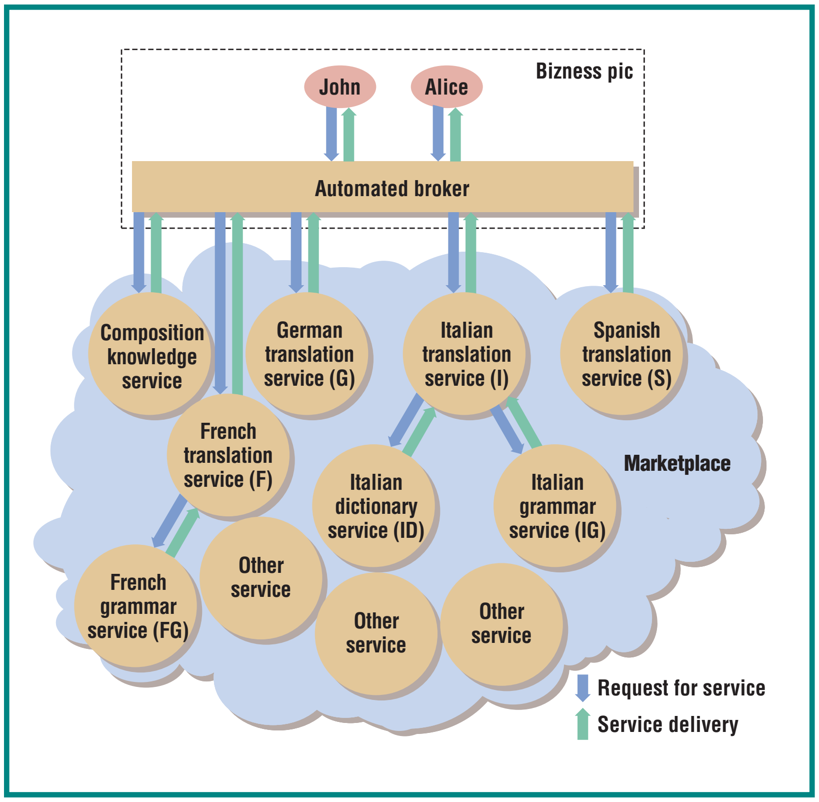
Bizness plc
John
Alice
Automated broker
Composition knowledge service
German translation service (G)
Italian translation service (I)
Spanish translation service (S)
French translation service (F)
Italian dictionary service (ID)
Italian grammar service (IG)
Marketplace
French grammar service (FG)
Other service
Other service
Other service
Request for service
Service delivery

Figure 1. The supply network formed in response to John's request.

force their contracts with the other service providers. However, such action might cost more (in time or money) than the original service cost warranted, given the micropayment model envisaged for SaaS. To facilitate contract enforcement in the automated domain, the parties involved could employ a third party that holds payment (in escrow, for example) and only releases it when all parties are happy with the service's execution. Ultimate dispute resolution might be necessary through arbitration or court action. For the SaaS approach to succeed, any automated dispute resolver should avoid this final recourse in most cases. In the event of failure, we can reasonably expect that Bizness plc's broker would substantially decrease the rating of the supplier concerned (if not remove it from the set of potential partners altogether).

Another (possibly less expensive) alternative is for John to try to diagnose the problem and its location—either to direct the legal action more specifically or to fix it for future invocations. Or he could simply reexecute the service, explicitly stating to the broker that it shouldn't use the Italian service previously employed and find an alternative. However, he must weigh the potential additional cost of changing his requirements and perhaps procuring a more expensive service against that of trying to fix the current problem (after all, provider I might have been let down by another service below it in the supply network and have already taken corrective action).

This situation becomes more complex if the service failure is partial—that is, the Italian service executed successfully but returned the document partially or completely untranslated. In addition to determining where the error occurred, John would need a mechanism to demonstrate that the results didn't match those promised in the service description.

Let's assume that John decides to try to fix the problem (in this case, a complete failure of the Italian translation service).

Understanding failures

John might try to understand the software himself, but we think he'll more likely call on an expert, Alice, to help diagnose and fix the problem. Alice is a software engineer in the Bizness plc IT department.

First, Alice will gather information about the failure. Service-oriented software, however, might provide very little information, with what little there is fragmented and hard to obtain. Alice needs to understand the software's behavior after it has executed but has no means of exactly reproducing the relevant processing. Bizness plc doesn't own the service provided to John but simply contracted with others to deliver the functionality for a given price. We can see possible candidates for the failure (I, ID, IG) in Figure 1, but Alice can only see the top level of service providers (F, G, I, S). Therefore, she knows only that the Italian translation service failed, but can't see the details of who might be supplying subservices to the Italian translator.

So what other information does Alice have available initially? She has the requirements John provided, information the broker provided about the top-level service composition, and information about the service providers with which the broker contracted.

She might submit the request again and try to trace the service's behavior during its execution. This could help her develop a behavioral model of the overall service provided, but it comes at a cost. The providers contracted to provide supporting services will charge for a new execution (unless contract terms indicate otherwise), and this cost must be weighed against the benefit of the information gained. Also, even if Alice can examine the data flowing between Bizness plc and the top-level providers, they might not be able to release information about providers further down the supply network to her, leaving an incomplete picture.

The supply network presents one of the major obstacles to effective software understanding in a service-oriented context. Organizations in the network have a vested interest in protecting the details of their implementations—this is the added value the service consumers pay for. Whether they contract out to produce a composite service or implement the functionality themselves, this knowledge is their prime asset. This problem might thus require nontechnical solutions such as business alliances or proactive supply chain management to increase trust between organizations and promote information sharing.

Alice also faces the possibility that the particular set of services contracted and subcontracted will differ from the original set when she reexecutes the service request. Their coordination might also differ because of the wholly dynamic and negotiated nature of SaaS. Consequently, the way the software is provided could have changed even though the requirements haven't. This is a strength of SaaS from an evolutionary viewpoint but a real problem in the event of a failure. Even if the same services are contracted, the providers might have updated the functionality in the interim. Current technologies exist to address this through versioning (see the "Service-Oriented Technology" sidebar), but any application relying on other services (particularly those external to the organization) faces this risk.

Understanding the software

Whichever strategy Alice adopts, a service-oriented approach requires that she understand various artifacts and their relationships, some quite traditional (although perhaps having a different role) and others not normally considered in software understanding.

Traditionally, Alice would build a mental model of the system and analyze the point of failure.[6] This would require her to understand the architecture, data flow, and control flow, perhaps using tools like program slicers (such as CodeSurfer[7]) or object browsers (such as NetBeans[8]). However, as we've seen, much of this information might remain hidden and unavailable in a service context. Alice must therefore shift her focus from understanding a system to understanding the relationships between composed services, which will require her to be conversant with composition languages and rules and understand their implications. This differs from traditional system understanding in granularity. Services will typically have a larger granularity than the source code statements traditionally used for understanding.

Alice must also understand the requirements which, in some ways, might be less rigorously specified than in traditional software systems to be useful to John because he's an end user who must be able to express his needs easily and quickly. The resulting requirements statement must, however, be formal enough to enable the automated broker to understand what John needs.

The most radically different area for Alice to deal with is the broker itself, which likely has "intelligence" to let it negotiate with providers for their services on Bizness plc's behalf. Bizness plc policies (such as upper limits on service costs or collaboration agreements between departments and organizations) will guide this negotiation. If Alice tries to reproduce an exact copy of the procured services, she should understand the implications of the broker's policies and strategies to ensure she obtains the same services (assuming they still exist in the marketplace).

Understanding changes

One of the major advantages claimed for service-oriented software is the ease of making changes. If John now needs to provide his quarterly reports in an additional language because Bizness plc has expanded into Russia, he simply changes his requirements to include the extra language. This change is much simpler than its equivalent in traditional software because it doesn't require implementing such features in code but only procuring them for the length of time needed to translate the document. The cost of changing the requirements should be minimal, but John might want to use Alice's knowledge of procurement strategies to make the most appropriate change to his request. Service-market conditions could affect whether John needs Alice or not. A buoyant market will likely have suppliers to meet John's needs, but if the market goes into recession he'll probably need Alice's expertise,

One of the major advantages claimed for service-oriented software is the ease of making changes.

Service-oriented software presents an inherent uncertainty because of its distributed and negotiated nature.

either as a procurer or to create some small in-house services to meet his needs. Creating in-house services, however, starts to erode the advantages of a fully service-based software development approach.

Adding the Russian translation service would previously have been known as *perfective software maintenance* and, by definition, changes the software requirements. In SaaS the effort required of John comes down to nothing more (at least not visibly to Bizness plc and John) than this requirements change.

However, should a problem arise with the change (for example, the procurement fails), it becomes not perfective but *corrective* maintenance. Considering service-oriented software in terms of the staged software lifecycle model,[9] initial development and evolution should be relatively simple because they involve merely a statement or restatement of requirements. The model implies that servicing is a relatively easy (albeit perhaps lengthy) phase, but in service-oriented software and from an understanding viewpoint this will likely be difficult because, as we've shown, defect repairs are costly to manage. The phase-out stage occurs with every execution of the software during unbinding (the particular "application" is phased out). Close-down simply involves throwing away requirements. We conclude, then, that service-oriented software requires a completely new maintenance model and even a redefinition of the different types of maintenance.

Potential solutions

We see several possible solutions to some of the problems the software engineer faces when trying to comprehend service-oriented software. Although some of the provision activity is technical, many solutions to the understanding problems are nontechnical (as with many SaaS issues).

Problem: Knowledge boundaries between organizations

The knowledge Alice needs is locked up in the service providers. Alice must therefore negotiate with the providers for the information she needs, accounting for their need to protect their assets. Forming industry supplier networks could reduce interorganizational distrust and make information more readily available. This active management of the supply network will prove important for quick problem resolution. Organizations could also tackle this issue using technical means such as a preexisting agreement to exchange technical information, perhaps through linked code-browsing tools.

Problem: Partial visibility of the supply network

Alice could try to solve this issue by negotiation. A more technical solution would be to have a service that could "see" the whole supply network and, perhaps for a fee, release this information to Alice (so she would discover that ID and IG exist in our example). The fee would provide compensation to those providers who have lost their privacy (and would require their agreement).

Problem: Understanding the state of the software

Some traditional approaches to distributed understanding look at state information,[10] which could help Alice find a failure's source. An overall view could prove difficult to achieve, however, because of the supply network's partial visibility and the limited flow of information between service providers due to confidentiality concerns.

Problem: Uncertain software construction

Service-oriented software presents an inherent uncertainty because of its distributed and negotiated nature. Also, the delivered solution might include several levels of granularity. Runtime tracing of the service invocation seems the most promising approach to gathering as much information as possible about the services and supply network. This would let Alice retrace the service execution to the failure point (assuming suppliers can release the required information). This is an issue for the framework within which the software is constructed. Managing runtime tracing at the framework level would alleviate many of the difficulties raised here but would require a spirit of openness between suppliers and consumers.

Problem: Inappropriate tools

Service-oriented software's construction means traditional understanding support tools won't work. We need tools that automatically collect as much information as possible for Alice (perhaps using prenegotiated agreements between providers) so that she begins her work with as complete a picture as possible.

Such tools' construction and provision, however, will likely be more complex from a nontechnical than a technical viewpoint.

Alice's role

Alice's role differs from that of the traditional in-house software engineer primarily in that she must understand fewer low-level technical software details but be skilled in negotiation and communication with clients and service providers. She must comprehend business policies pertaining to service procurement and understand the activities of Bizness plc's broker. Her activities focus much more on obtaining and organizing information from contracted service providers than on building code. The only stage at which she might be involved in creating a new system is as an advisor on requirements definition. Although coding skills are perhaps less important, Alice clearly needs some knowledge of software construction to successfully process the information she receives about failures.

Many barriers to successful understanding of service-oriented software arise from its distributed and dynamic nature. The flexibility that gives this approach the potential to ease the evolution problem creates new difficulties in software understanding, many of which will be primarily nontechnical. Examining the processes involved in this kind of understanding in terms of both corrective and perfective maintenance suggests possible solutions to these problems, including tailoring the process for service understanding rather than program understanding.

Acknowledgments

The authors acknowledge the work of the UK Pennine Group and their colleagues at the Universities of Durham and Keele and UMIST on the service-oriented model of software and for helpful discussion of some of these ideas and the example. We also thank participants at the working session "Comprehension Needs in Highly Agile Software" at the IEEE International Workshop on Program Comprehension, Paris, 2002, and at the 1st Pennine Research Forum, Manchester, 2002, for additional useful discussions in this area. The reviewers also provided many detailed and helpful comments that substantially improved this manuscript. This work is partly supported by the UK Engineering and Physical Sciences Research Council under grant GR/R71733.

About the Authors

Nicolas Gold is a lecturer in the Department of Computation at UMIST (the University of Manchester Institute of Science and Technology). His main research interests include software comprehension, software evolution, and software maintenance. He received his PhD in computer science from the University of Durham, UK. He is a member of the IEEE and the Institute for Learning and Teaching in Higher Education. Contact him at the Information Systems Group, Dept. of Computation, UMIST, PO Box 88, Sackville St., Manchester, M60 1QD, UK; n.e.gold@co.umist.ac.uk.

Claire Knight is a development engineer at Volantis Systems Ltd., Guildford, UK. Her main research interests include software visualization; program comprehension; Java; grid and Web services; and Java, Ant, XML, and PHP development. She received her PhD in computer science for research on software visualization from the University of Durham. Contact her at Volantis Systems Ltd., 1 Chancellor Court, Occam Road, Surrey Research Park, Guildford, Surrey, GU2 7YT, UK; claire.knight@volantis.com.

Andrew Mohan is a doctoral candidate at UMIST, UK. He received a BSc (Hons) in computer science from the University of Durham and has worked for several years in legacy systems support. His main research interests include software maintenance and evolution, program comprehension, and software quality. He is a member of the British Computer Society and is a Chartered Engineer. Contact him at Information Systems Research Group, Dept. of Computation, UMIST, PO Box 88, Sackville St., Manchester M60 1QD, UK; a.mohan@postgrad.umist.ac.uk.

Malcolm Munro is a professor of software engineering in the Department of Computer Science at the University of Durham, UK. His main research focus is software visualization, software maintenance and evolution, and program comprehension. He is also involved in research in Software as a Service and the application of Bayesian networks to software testing and program comprehension. Contact him at the Dept. of Computer Science, Univ. of Durham, Science Laboratories, South Rd., Durham DH1 3LE, UK; malcolm.munro@durham.ac.uk.

References

1. K.H. Bennett et al., "An Architectural Model for Service-Based Software with Ultra Rapid Evolution," *Proc. IEEE Int'l Conf. Software Maintenance* (ICSM 01), IEEE CS Press, 2001, pp. 292–300.
2. T.A. Standish, "An Essay on Software Reuse," *IEEE Trans. Software Eng.*, vol. SE-10, no. 5, Sept. 1984, pp. 494–497.
3. K.H. Bennett et al., "Prototype Implementations of an Architectural Model for Service-Based Flexible Software," *Proc. 35th Hawaii Int'l Conf. System Sciences* (HICSS 02), IEEE CS Press, 2002, p. 76b.
4. M. Turner, D. Budgen, and P. Brereton, "Turning Software into a Service," *Computer*, vol. 36, no. 10, Oct. 2003, pp. 38–44.
5. "Beyond the Hype of Web Services—What Is It and How Can It Help Enterprises Become Agile," *EDS*, www.eds.com/about_eds/homepage/home_page_lehmann.shtml.
6. A. Von Mayrhauser and A.M. Vans, "Program Comprehension During Software Maintenance and Evolution," *Computer*, vol. 28, no. 8, Aug. 1995, pp. 44–55.
7. Grammatech, 2004, www.grammatech.com.
8. Netbeans.org, 2004, www.netbeans.org.
9. V.T. Rajlich and K.H. Bennett, "A Staged Model for the Software Life Cycle," *Computer*, vol. 33, no. 7, July 2000, pp. 66–71.
10. J. Moe and D.A. Carr, "Understanding Distributed Systems via Execution Trace Data," *Proc. IEEE Int'l Workshop Program Comprehension* (IWPC 01), IEEE CS Press, 2001, pp. 60–67.