Pragmatic and Opportunistic Reuse in Innovative Start-up Companies

Slinger Jansen, Sjaak Brinkkemper, Ivo Hunink, and Cetin Demir,
Utrecht University

Two case studies show how start-ups generally select third-party functionality pragmatically and how they deal with the problems encountered with opportunistic reuse.

Both practitioners and academics often frown on pragmatic and opportunistic reuse.[1] Large organizations report that structured reuse methods[2] and software product lines are often the way to go when it comes to efficient software reuse.[3] However, opportunistic—that is, nonstructured—reuse has proven profitable for small to medium-sized organizations.[4,5] Here, we describe two start-ups that have opportunistically and pragmatically developed their products, reusing functionality from others that they could never have built independently. We demonstrate that opportunistic and pragmatic reuse is necessary to rapidly develop innovative software products.

When software vendors are developing an innovative product or service, its success depends largely on the time to market. We often hear companies say, "It may be buggy, but we were first!" Opportunistic reuse is one way to speed up product or service development. Research has shown that the cost of purchasing a component or service is between 1 and 19 percent of the cost of developing that component or service.[5]

We define *pragmatic reuse* as extending software with functionality from a third-party software supplier that was found without a formal search-and-procurement process and might not have been built with reuse in mind. Our definition differs from that of Amir Tomer and his colleagues[6] in that we generalize the term to comprise reusing both services and components. We define *opportunistic reuse* as extending software with functionality from a third-party software supplier that wasn't originally intended to be integrated and reused.

Software Functionality Extension Mechanisms

Several mechanisms extend software functionality with third-party functionality: component calls, service calls, source code inclusion, and shared data objects (see Table 1 and Figure 1).

Component calls can be direct to a component or indirect through a component bus or another (glue) component. Components are dormant until invoked, as opposed to services, which must be live to be used. Components can run independently (think of a "hello world" application) or can require a component framework or virtual machine for instantiation.

Services run locally (such as MySql running on the same server as a PHP program) or re-

0740-7459/08/$25.00 © 2008 IEEE

Table 1

Software functionality extension mechanisms

Unit of inclusion	Call type	Figure	Interaction method	Example
Component	Direct	1a	Pipe and filter	Call runnable component and feed it data
		1b	Component library reuse	Method calls and class use, dynamic linked libraries (DLLs), libraries, jars, and so on
	Indirect	1c	Glue code	Glue code is written between the application and the extending component
		1d	Shared data object	Call runnable component on data source, report when finished
		1e	Component bus	Component invocation bus that's active like a service, Corba call
		1f	Plug-in architecture	Java plug-in architecture, Eclipse plug-ins
Service	Direct	1g	Service framework	Specific integration, forward form submits, Web page inclusion, SOAP call, MySql database service, online currency converter, and so on
	Indirect	1h	Enterprise service bus	Service call that's evaluated and directed to three different services, of which one is found most suitable to handle the request (for example, OWL-S)

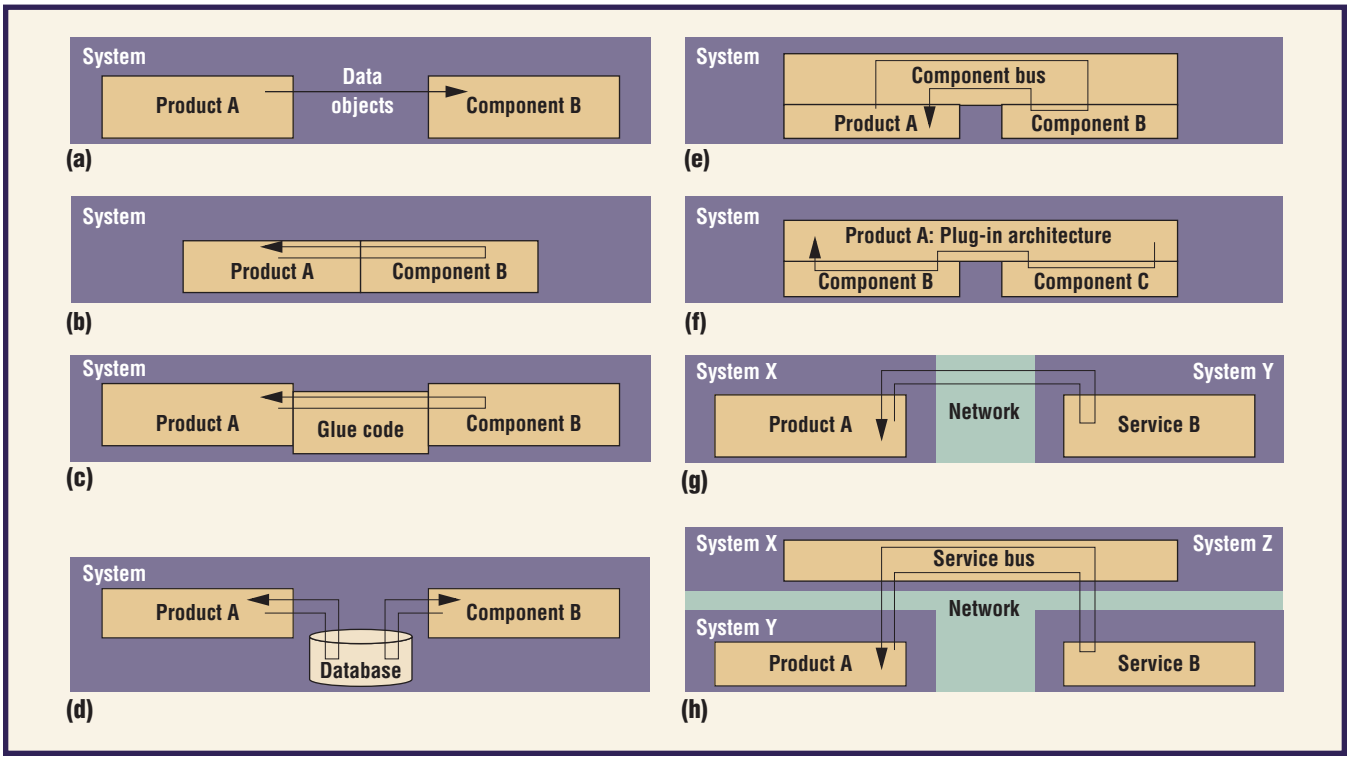


Figure 1. Graphical representation of software extension mechanisms. Here, we see mechanisms in which (a) one component sends its data to another; (b) one component calls on another in a library; (c) one component calls another through a customized interface; (d) two components communicate using shared data objects; (e) components communicate using a bus; (f) components plug into another component and communicate through this component; (g) services communicate with each other while sticking to previously established conventions; and (h) services communicate with each other through a bus.

motely (for example, a currency-conversion service) when called on. Services can be called on directly through, for instance, a SOAP call, or indirectly through a service bus. We can't avoid the difference here between services and components, in that services are tied less to a location than components are. Furthermore, facilitating dynamic composition and updating of service configurations is easier for services than for component configurations.[7]

Netherware

Netherware is a software incubator Utrecht University founded that provides young student entrepreneurs with cheap office space, courses, development tools, and advice from experienced entrepreneurs and venture capitalists. Students develop product prototypes with a business plan, in addition to performing staff roles such as marketing, bookkeeping, human-resources management, or business management in the incubator. Since its start in 2002, Netherware has already delivered upward of 20 successful start-ups in the Netherlands.[1]

Reference

1. J.H. ten Berge et al., "Teaching Entrepreneurship in the Software Branch: The Design of an Authentic Task in Higher Education," to be published in *Education and Information Technologies*, 2008.

Unlike Tomer and his colleagues,[6] we explicitly rule out source code inclusion from the list of software functionality extension mechanisms because it has serious drawbacks: Simply copying a method, class, or full package into a component configuration establishes tight coupling—with all its downsides, such as that updating the third-party source requires another copy-paste action.

Finally, data object sharing, although popular, requires a "smart" database (at least locking and transactions are required) and software objects can't independently evolve the data model.

We classify the different mechanisms for third-party functionality inclusion for two reasons. First, we can use the classification to see whether organizations drive business differently when they use different inclusion types. Second, when studying opportunistic COTS reuse, we can establish whether the products that are integrating with these COTS require a specific software architecture. The mechanisms we present have been inspired by comprehensive overviews of reuse mechanisms, including software reuse,[8] components,[9] and architectural connections.[10]

Two Start-up Cases

Both cases of opportunistic reuse are examples of newly developed solutions (ideas developed in 2006 and 2007 and first releases planned in 2008), and both start-ups still need to prove themselves in the field. We selected these cases from among 10 start-up software businesses from the Netherlands (see the "Netherware" sidebar). Our main selection criteria for the two cases included the number of integrated components and services the product had, the business's viability, and the team's dedication. We designated them FirstStartUp and SecondStartUp for uniformity and to protect the business ideas. We gathered empirical data in two case-study databases and collected it by interviewing four developers, reviewing development and design artifacts, testing prototypes, and inspecting software code.

For both case studies, we describe each integrated functionality (see Table 2), including the extension mechanisms used, the functionality's cost, the integration effort, the architectural impact, feature criticality, and coupling. The integration effort refers to the number of hours developers spent integrating functionality. The architectural impact refers to the scale on which the software's architecture and interaction model needed adjustment to accommodate new functionality. Feature criticality refers to how important the feature is for the solution's key selling point. Finally, coupling refers to how easily developers could replace a component with a substitute, independent of that substitute's availability.

FirstStartUp: Online Workshop-Planning Application

In 2007, FirstStartUp generated the idea to develop a workshop-scheduling application for large groups of people. Similar applications exist, but this application is directed at a specific customer target group. The application facilitates planning workshop series in which participants must indicate which workshop they wish to attend. Normally, developing such an application requires a lot of time and specific knowledge on time-tabling. In this case, however, FirstStartUp decided to reuse common components to solve this problem. The developers integrated three components into their solution, Solution 1.

Spreadsheet service. To plan a series of workshops, the application required a workshop list and a participant list. This application can be successful online mainly because it reduces paperwork for the organizer. It uses a smart method for an organizer to enter data, which the developers wanted to be simple, especially because workshop organizers will already have a list or even a spreadsheet of participants and sessions. To enable copying and pasting such a list to the application, the developers decided they needed a spreadsheet application that they could integrate. Zoho Sheet (http://sheet.zoho.com) provides this functionality as an online service. Third parties can reuse it in their own applications. Zoho Sheet interacts with Solution 1 by including a Zoho spreadsheet page on the solution Web page and

by making SOAP calls to the Zoho server to read locally stored data.

- Interaction method: Web page inclusion and SOAP calls
- Cost: free
- Integration effort: ♦
- Architectural impact: ♦♦♦
- Feature criticality: ♦♦
- Coupling: ♦♦♦

Payment service. To enable payment in the Netherlands, where FirstStartUp will first launch the application, developers used the iDeal (www.ideal.nl) component. Most banks in the Netherlands provide this component to let online vendors receive quick electronic payments. iDeal is especially popular because the money is transferred from bank account to bank account immediately, reducing credit problems and the like. The iDeal component is accessed remotely from a button on a Web site (which posts the payment data to the iDeal site). After users have finished entering their credentials and the payment is completed, the iDeal site redirects the user back to the original Web site.

- Interaction method: forward form submit
- Cost: service charge of €0.75 to €1.50 per transaction
- Integration effort: ♦♦♦
- Architectural impact: ♦
- Feature criticality: ♦♦
- Coupling: ♦

Time-tabling component. Once the organizer has provided all the data and workshop participants have entered their preferences, the application must create and publish a schedule. Schedule creation is a complex problem that has been studied since the earliest days of computer science. To avoid having to reinvent the wheel, FirstStartUp's developers looked for planning applications that provided the required features. Tablix (www.tablix.org) is a time-tabling application that uses genetic algorithms. It's available under the GNU General Public License and includes, among other features, a GUI. Tablix proved to be a good fit for Solution 1 because it has a command-line interface and can import and export data in myriad formats (XML and so on). Tablix is integrated with a pipe and filter, such that when Solution 1 needs to create a schedule, Tablix starts up and receives the data. Tablix then creates a solution and stores it locally in XML for Solution 1 to read.

Table 2

Integration impact characteristics

Integration effort	Time spent integrating functionality
♦	Less than a week
♦♦	Between one and two weeks
♦♦♦	At least two weeks
Architectural impact	**Impact on the host architecture**
♦	No adjustments required
♦♦	Some adjustments required (such as glue code)
♦♦♦	Complete restructuring of interaction model and product architecture required
Feature criticality	**Feature's importance to the software's key selling point**
♦	Nice to have
♦♦	Essential but can be substituted
♦♦♦	Critical to the software
Coupling	**Effort required to replace a functionality with a substitute**
♦	Loose—easy to replace with a substitute
♦♦	Medium—possible to replace but with considerable difficulty
♦♦♦	Tight—intertwined with the software, almost impossible to replace

- Interaction method: shared data source and pipe and filter
- Cost: free
- Integration effort: ♦♦♦
- Architectural impact: ♦
- Feature criticality: ♦♦♦
- Coupling: ♦♦♦

SecondStartUp: Online Direct Marketing via Third-Party Social Network

In 2006, SecondStartUp began developing an automatic marketing service via an existing social network. The marketing service lets customer companies communicate with their clients in an automated way but with a human-like feel (that is, it simulates a real-life operator). Simultaneously, the service gathers valuable, client-specific information, such as preferences and characteristics relevant for marketing. Solution 2 can be used to directly sell products. Normally, automated human–computer communication requires linguistic algorithms and extensive knowledge in linguistics, which are expensive to develop and gain. Furthermore, using a third-party network to communicate requires a network-specific implementation to approach the social network's users. As in Solution 1, SecondStartUp decided to reuse common functionality to solve these problems.

Component and service selection was driven largely by information available on specific API and integration Web sites.

Third-party social-network library component. SecondStartUp chose a specific implementation and social network, as well as a library that supports full interaction with that network. To make the application as extendable as possible, SecondStartUp uses a layer of abstraction.

- Interaction method: direct component call and component library reuse
- Cost: free
- Integration effort: ♦
- Architectural impact: ♦♦♦
- Feature criticality: ♦♦♦
- Coupling: ♦♦

Advanced language interpreter component. To automatically communicate with customers, Solution 2 needed an advanced language interpreter that could recognize language structure (grammar) and meaning (semantics). Developing such a component requires extensive knowledge of linguistics and artificial intelligence. Basically, the solution requires a component that lets customers chat with the service using natural language. The Artificial Intelligence Markup Language (AIML) is an XML dialect for creating natural-language software agents. The AIMLbot2 (http://aimlbot.sourceforge.net) component uses linguistic algorithms that can interpret user input and automatically respond with relevant, predefined information. This information is stored in AIML scripts. To save time, SecondStartUp used configurable scripts that were freely available. An event occurs, for example, when the service receives a message from a customer. When Solution 2 invokes this event, the AIMLbot2 component executes, interpreting the message and responding to the client with appropriate information.

- Interaction method: direct component call and component library reuse
- Cost: free
- Integration effort: ♦
- Architectural impact: ♦♦
- Feature criticality: ♦♦♦
- Coupling: ♦♦♦

Micropayment service. To enable customers to easily pay small fees, Solution 2 uses a text-message service. This service lets online and offline vendors quickly charge fees between €0.00 and €10.00 (micropayments) through a client's cell phone account or from a prepaid balance.

- Interaction method: direct service call through form submission
- Cost: service charge per transaction
- Integration effort: ♦♦♦
- Architectural impact: ♦
- Feature criticality: ♦♦
- Coupling: ♦

Component and Service Integration

Let's look at some difficulties the two start-ups encountered during component and service integration. The empirical evidence will show us what challenges lie ahead for component providers and integrators and how all this relates to current research.

Component and Service Selection

Both products were created by two-person entrepreneur teams that weren't engaged in the projects full time. This let the developers take certain risks in the selection process.

For Solution 1, the spreadsheet selection process was challenging. The team needed to provide at least a spreadsheet interface to users. The two services on the short list were from Google Docs (http://docs.google.com) and Zoho (http://spreadsheets.zoho.com). Furthermore, Solution 1 needed to be able to access any data users created. During the first evaluation, the team discovered that Google Docs didn't allow anonymous login to its service. This presented another problem because users would now need to explicitly share sheets with Solution 1. Zoho was thus the better-fit component for Solution 1, even though the developers needed to overcome many integration problems.

For Solution 2, the third-party networking component was most critical. Similar to FirstStartUp, SecondStartUp was looking for components that were free, to keep costs low. Several free and commercial products that support the chosen network were available in a variety of environments, such as PHP, .NET, and J2EE. SecondStartUp preferred the .NET component because it had experience with .NET. Moreover, SecondStartUp had personal contact with the developers of the library it chose, and selected it even though it's no longer formally supported.

Component and service selection for both start-ups was driven largely by information available on specific API and integration Web sites. Both the payment service and the spreadsheet service have sites that provide comprehensive examples of integrating services into applications. The time-tabling component has extensive manual pages that describe exactly how the component works and how developers can reuse and integrate it with other applications.

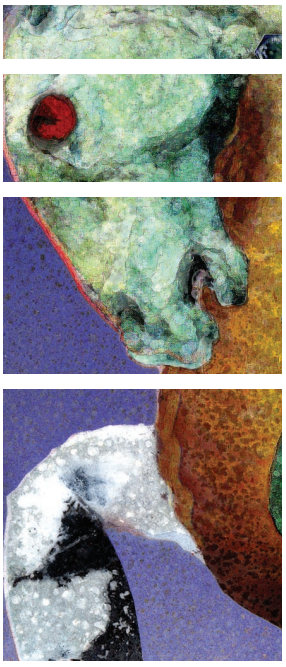
Component and Service Integration and Interaction

The method used to integrate and interact with components and services determines how a product or service's software architecture looks. When developers are opportunistically reusing functionality, severe changes to the architecture and complex glue code might be necessary to achieve correct interaction and integration.

For Solution 1, the time-tabling component was relatively simple to integrate. Because the component runs each time the application needs to create a schedule, it was fairly loosely coupled to Solution 1. FirstStartUp integrated the time-tabling component using simple steps. First, it determined the component's workflow (table data in, schedule data out). Then, it implemented the data types in Solution 1 so that Solution 1 could readily supply the correct data in the correct XML format for the time-tabling component. Solution 1 now checks regularly (every minute) to see whether the time-tabling component has finished. Once the component finishes, Solution 1 imports the schedule (output in XML) and sends an email to the customer. The component is still tightly coupled, mostly because no abstract data layer is implemented in Solution 1; switching to another component introduces substantial work. One issue with the time-tabling component is that when the application gives it a scheduling problem with no solution, it runs eternally. A time-out checks whether the component can find a solution or not.

Integrating the spreadsheet service was painful. Solution 1 required different or "varying" features from the service: the spreadsheet functionality to provide a user interface and the storage functionality to enable storing the sheet on the FirstStartUp Web server. The spreadsheet functionality is designed for integration with a sheet that's stored on the third-party spreadsheet server. When a user opens the spreadsheet service, it comes from the same location as all the spreadsheet service HTML/Ajax/JavaScript code. Solution 1, however, requires that the file be opened not from the third-party server but from the Solution 1 server, leading browsers to believe that a "cross-site scripting hack" had occurred. The developers solved this using a hidden form post, which made it seem as though the user uploaded the spreadsheet to the spreadsheet service. Solution 1 also requires that users navigate between different sheets. Between steps, any changes to these sheets need to be saved. Currently, to save a sheet, the user must click the "save" button in the spreadsheet service, meaning that changes to the sheet are lost if a user clicks "next" to go to the next sheet. FirstStartUp solved this problem pragmatically by graying out the "next" button using JavaScript until the user clicked "save." Finally, to get data from a stored sheet, the application uses SOAP calls.

SecondStartUp chose to make the application as flexible as possible using a layer of abstraction for third-party functionality. This approach has several downsides. First, errors are difficult to handle in a standardized way. Second, implementing an abstraction layer that handles the essential features requires much effort. The advantage is that this approach creates loose coupling between the solution and the third-party functionality. Consequently, the abstraction layer enables easily switching to another component or updating the same component.



Dealing with Upgrades

When a system depends on third-party functionality, upgrades can affect the offered functionality considerably. If the interaction with a component or service changes, for example, the system will break or behave unpredictably. On the other hand, if the upgrade offers new features, third parties will want to use these features as soon as possible. Software integrators can avoid problems and discover opportunities earlier when they are frequently informed about changes.

The time-tabling component in Solution 1 is updated only when the developers wish to upgrade it, because it's installed locally. With regard to the spreadsheet service and the payment service, the developers are members of mailing lists so that they stay up-to-date when problems arise and potentially profitable changes occur. With regard to upgrades, services have one advantage, in that they can be updated without interference from the service integrator. This involves a certain amount of trust, such that the service supplier preserves both functionality and interface when upgrading. Solution 1's developers trust the service suppliers to not change the service radically without an announcement.

Practical Considerations

Even though the integration of third-party functionality can be painless, we must take into account some practical considerations. Solution 1's developers were well on their way to integrating the spreadsheet service when they realized that they were integrating the product in English and that they hadn't thought about localization. Luckily, the spreadsheet services are available in many different languages, among which Dutch is an option.

The SecondStartUp developers were less fortunate—the free scripts they used for the advanced

Both teams prefer pragmatism and agility over formality and rigidity.

language interpreter were available solely in English, whereas their customer companies and their clients preferred Dutch. So, to support Dutch, they needed to translate more than 50,000 sentences. To overcome this problem, SecondStartUp is planning a distributed translation approach in which they ask friends, family, and colleagues to translate a few sentences using a simple (and easy-to-develop) Web application.

Lessons Learned

From interviews with the two start-ups we derived several lessons. These lessons show that software reuse isn't just a technical problem, and they support Will Tracz, who aptly chose "software reuse is a technical problem" as the first myth of software reuse.[11]

Components and services are selected on the basis of critical functionality. Both presented solutions integrate third-party functionality. Interestingly, the fact that all third-party functionality suppliers offered functionality as either a component or a service only remotely influenced both selection processes, showing that pragmatism trumps architectural considerations when developers are deciding between components and services. According to the development teams, however, significant differences in impact exist between reusing a service or a component. One advantage of services over components is that a service's users and integrators are encouraged to use new functionality when it's released. The development teams reported that a more continuous release cycle speeds up the implementation of new innovations in products. Developers prefer reusing components over services when these components provide critical functionality. This is because the component's functionality remains stable, the component doesn't have to run remotely, and performance is guaranteed and independent of network capacity. Another factor that influences the choice between a component or a service is where the most valuable data is located. The planning application, which hopes to acquire customer data through a spreadsheet interface, sees its core value in the customer data. FirstStartUp found outsourcing data storage at Google Docs, for instance, to be too risky.

Documentation availability is critical to component and service reusability. The selection process for both solutions was influenced by the knowledge network surrounding the third-party functionality. For Solution 1, the developers relied heavily on forums, Web sites, source code repositories, API documentation, and so on. Solution 2, as we mentioned, decided to use one specific library because SecondStartUp knew the library developer. Both cases show that, should a third-party functionality supplier wish to have its product adopted more frequently, a knowledge infrastructure is required to support the integrating party. Developers deemed some knowledge sources essential to integration, such as API documentation, an integration example, a demo version for a commercial product, and an active development community in the shape of a mailing list or forum. A formal standard for knowledge infrastructures concerning COTS and services could greatly assist both suppliers and integrators.

COTS selection methods are hard to apply in practice. Both FirstStartUp and SecondStartUp place a critical note with regard to formal behavioral models, COTS selection methods, and formal development procedures. These models, methods, and procedures either aren't adopted by wide ranges of third-party functionality suppliers or are much too formal and lengthy to be useful. Both teams prefer pragmatism and agility over formality and rigidity. The teams do wonder, however, how long they can stick to these software engineering morals.

COTS and services don't interact sufficiently. For all six reused components and services, we determined that interaction with their solutions was very crude. Neither the time-tabling component nor the spreadsheet service used exit codes, return codes, and so on, making testing and debugging intensive processes. We identify this as one main disadvantage to opportunistically reusing third-party functionality that wasn't originally intended for intensive reuse.

Reusing COTS and services enforces abstraction. Opportunistically reusing components forced both FirstStartUp and SecondStartUp to invest extra time in modularizing and abstracting parts of their software because they knew beforehand that integration might be troublesome and that components might evolve in the future. Both teams suspect that their solutions are more stable owing to modularization and abstraction.

Opportunistic reuse enables rapid development. The two start-ups demonstrate that you can develop innovative products more rapidly via opportunistic and pragmatic reuse. Initial releases might not be the highest-possible quality with extensive formal

COTS selection and reuse methods, but at least the products are released sooner.

FirstStartUp and SecondStartUp must still prove themselves in the field, and we don't know whether results from this research will hold in the long run. In its defense, we encountered similar functionality-extension mechanisms in earlier case studies of product software vendors in the Netherlands.[12] Furthermore, the members of FirstStartUp and SecondStartUp all have industrial experience with regard to software engineering, software architecture, and COTS reuse.

Although practitioners and academics often look down on opportunistic reuse, we suspect that this pragmatic approach to reuse is day-to-day practice in innovative industries. The two presented cases show that pragmatic reuse can be highly profitable and cost-effective for rapid development. They show that if third-party functionality providers wish their products to be reused, they must surround those products with a sufficient knowledge infrastructure. Furthermore, business requirements are much more crucial than technical requirements to the choice of services or components. Finally, standards are required for knowledge infrastructures and for service and component interactions to promote reuse.

Acknowledgments

We thank fellow entrepreneurs Peter Duijnstee and Wilco van Duinkerken for their inspiring ideas that contributed to this work. Furthermore, we couldn't have conducted this research without the software incubator Netherware (see the sidebar on p. 44).

References

1. R. van Ommering, "Software Reuse in Product Populations," *IEEE Trans. Software Eng.*, vol. 31, no. 7, 2005, pp. 537–550.
2. M.F. Dunn and J.C. Knight, "Software Reuse in an Industrial Setting: A Case Study," *Proc. 13th Int'l Conf. Software Eng.*, IEEE CS Press, 1991, pp. 329–338.
3. E. Henry and B. Faller, "Large-Scale Industrial Reuse to Reduce Cost and Cycle Time," *IEEE Software*, vol. 12, no. 5, 1995, pp. 47–53.
4. S. Morad and T. Kuflik, "Conventional and Open Source Software Reuse at Orbotech—an Industrial Experience," *Proc. Int'l Conf. Software—Science, Technology, and Eng.* (SwSTE 05), IEEE CS Press, 2005, pp. 110–117.
5. S. McConnell, *Rapid Development: Taming Wild Software Schedules*, Microsoft Press, 1996.
6. A. Tomer et al., "Evaluation Software Reuse Alternatives: A Model and Its Application to an Industrial Case Study," *IEEE Trans. Software Eng.*, vol. 30, no. 9, 2004, pp. 601–612.
7. S. Ajmani, B. Liskov, and L. Shrira, "Modular Software Upgrades for Distributed Systems," *Proc. European Conf. Object-Oriented Programming* (ECOOP 06), IEEE Press, 2006, pp. 452–476.
8. C.W. Krueger, "Software Reuse," *ACM Computing Surveys*, vol. 24, no. 2, 1992, pp. 131–181.
9. C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley Professional, 1997.
10. M. Shaw, R. DeLine, and G. Zelesnik, "Abstractions and Implementations for Architectural Connections," *Proc 3rd Int'l Conf. Configurable Distributed Systems*, IEEE Press, 1996, pp. 2–10.
11. W. Tracz, "Software Reuse Myths Revisited," *Proc. 16th Int'l Conf. Software Eng.* (ICSE 94), IEEE CS Press, 1994, pp. 271–272.
12. S. Jansen et al., "Integrated Development and Maintenance for the Release, Delivery, Deployment, and Customization of Product Software: A Case Study in Mass-Market ERP Software," *J. Software Maintenance*, vol. 18, no. 2, 2006, pp. 133–151.

About the Authors

Slinger Jansen is an assistant professor in the Department of Information and Computer Science at Utrecht University. His research focuses on software product management and software supply networks, with a strong entrepreneurial component. Jansen received his PhD in computer science from Utrecht University. Contact him at s.jansen@cs.uu.nl.

Sjaak Brinkkemper is a full professor of organization and information in Utrecht University's Department of Information and Computer Science, where he leads a group of roughly 30 researchers specializing in product software entrepreneurship. The group's main research themes are the methodology of product software development, implementation and adoption, and entrepreneurship in the product software industry. Brinkkemper received his PhD in computer science from the University of Nijmegen, the Netherlands. Contact him at s.brinkkemper@cs.uu.nl.

Ivo Hunink is a master's student in information science at Utrecht University. His research interests include social networking, software development, and entrepreneurship. Hunink received his bachelor's degree in gaming, virtual media, and security at Saxion University of Applied Sciences. Contact him at ihunink@cs.uu.nl.

Cetin Demir is a master's student in game and media technology at Utrecht University. His research interests include serious gaming, software development, and entrepreneurship. Demir received his bachelor degree in computer science at Saxion University of Applied Sciences. Contact him at cdemir@cs.uu.nl.