The Evolution of Software Evolvability

Chris Lüer David S. Rosenblum André van der Hoek
Institute for Software Research
University of California, Irvine
Irvine CA 92697-3425, USA
{chl,dsr,andre}@ics.uci.edu

ABSTRACT
We analyze two trends that have influenced the evolvability of component-based applications: increase of component exchangeability and increase of component distance. Exchangeability mechanisms can be classified either as code reuse or as service reuse. Component distance can vary from file scope to Internet scope. We discuss the various stages of evolvability in these dimensions, describe the state of the art, and speculate on future developments.

Keywords
Evolvability, exchangeability, code reuse, service reuse.

1. INTRODUCTION
Maintenance, or evolution, is the longest and most expensive phase of the software product lifecycle. Once released, software has to be corrected and updated. Evolvability is the property of programs that can easily be updated to fulfill new requirements; software that is evolvable will cost less to maintain. A component-based application is evolvable if it is easily possible to exchange individual components without changing others.

Component reuse exacerbates the problem of maintenance [9]. An application that consists of a large number of independently bought components will be much harder to update than a traditional, monolithic application, since each component will have individual updates from its manufacturer, and manufacturers will be independent from each other and located all over the world.

In this paper, we take a look at two historical trends that have made applications more evolvable. We identify several stages in each of these trends, including past stages that are often considered as outdated, and stages that many consider to be state of the art, but that are not widely used yet.

2. CLASSIFICATION
Two trends have influenced the development of evolvability technologies: an increase of component exchangeability and an increase of component distance (see Figure 1 for an overview). Component exchangeability means that components can easily be exchanged for other components, or updated with newer versions. Ideally, it should not be necessary to change other components to do this, or to change the architecture of the application. Component distance means the physical distribution of components over networks.

A consequence of these trends is the shift from design-time evolvability techniques to deployment-time evolvability techniques [7]. Design-time approaches to evolution require the source code to be accessible, but do not increase the evolvability of the compiled system. Deployment-time approaches, on the other hand, allow evolution to be managed by a component user without source access.

2.1 Component Exchangeability
The means to increase component exchangeability has been to introduce additional levels of indirection between components, i. e., to decouple them by making their connections more dynamic. The typical trade-off of this is a performance overhead. Additionally, application complexity increases, while component complexity decreases. Applications become harder to understand through the various indirection and dynamism techniques used; but individual components become more decoupled from each other, more independent and easier to identify, and thus they become cognitively less complex.

Increase of component exchangeability has happened in two contexts: code reuse and service reuse. We distinguish three

Exchangeability (Code Reuse)	(1) Copy and Paste	(2) Static Linking	(3) Dynamic Linking		
Exchangeability (Service Reuse)			(1) RPC	(2) Messaging	(3) Broadcasting
Distance	File		File System	Internet	

Figure 1. Comparison of the evolution of component exchangeability and component distance. Time moves from left to right; corresponding stages are shown on top of each other.

stages of exchangeability in code reuse, and three stages of exchangeability in service reuse. While service reuse is a different approach, it provides a higher level of exchangeability than code reuse and can be regarded as an extension of code reuse techniques.

2.1.1 Code Reuse

Code reuse means that an application reuses a component by accessing its actual code (whether in source or in compiled form), loading it into memory, and then executing it. The application controls where, when and how the component is executed.

The first stage of component exchangeability in code reuse is no exchangeability at all. This is the consequence of reusing code by copy-and-paste—an arbitrary piece of source code from the old project is copied and then pasted into the source of the new project. No connection is established between the original and the copy, and the copied code is not delimited in any way as to be recognizable as having been copied. There is nothing to keep the developer from changing the reused code once it has been pasted; this means that it may not be possible to identify the code as reused even if the source files are compared line by line later on. As a result, maintenance effort multiplies: each copy of the code will have to be maintained separately; the copies will evolve into separate directions and become more and more dissimilar over time. The advantage of this form reuse is that it requires only a minimum of tool support and does not increase program complexity or lower performance.

The second stage of exchangeability of components is exemplified by statically linked libraries, as they are usually used in the language C, for example. Libraries are distinct, well-defined units, or modules, but they are copied into each executable file. To exchange or update them, the application has to be relinked, which requires access to the compiler output files and the configuration of the application (as embodied in a make script, for example). Maintenance has become easier, but still requires rebuilding of the whole program whenever a library is modified. If a library is used by several applications, each of these has to be rebuilt when the library is updated. The advantage over copy-and-paste is that modules are clearly identified (at least on the source level), and that there is at most one copy of each reused code piece in each program. The trade-off is the need for a more complicated programming system.

Dynamic linking [2] constitutes the third stage of exchangeability; Java is an example. Each module is stored in its own file and exists only once per file system, and is accessed by all programs that need it. To update a module, one only has to exchange the corresponding file; it is not necessary to touch the actual application. Alternatively, it may be possible to update an environment variable (e. g., the Java class path) to point to the new version of a module instead of the old version. Besides added complexity, dynamic linking entails a performance trade-off: each module has to be linked to the application at run-time before it can be used.

Dynamic linking as used in Java avoids redundant copies of the same module in the scope of the file system. In a networked or distributed system, one may still have to cope with multiple copies at the various locations. It is possible, however, to extend dynamic linking to work on an Internet scale [5]. While the performance overhead becomes large, it can be reduced significantly through caching and event notification: the local system keeps a copy of the module, and is notified by the server that owns the original copy of the module whenever it is updated. In this way, the module has to be downloaded through the network only once after each update. The advantage of dynamic linking on an Internet scale is that only one master copy of each module needs to exist worldwide. Once the master copy is updated, the update is automatically promoted to all systems that use the module.

2.1.2 Service Reuse

The same historical evolution towards increasing exchangeability of components as with code reuse exists with service reuse mechanisms. Service reuse [4] means that the application is not granted access to the reused code, and thus cannot link to it, but instead it has to communicate with an independently running instance of it. We distinguish three stages in the development of service reuse technologies.

The first stage of exchangeability in service reuse is procedure call communication. Components are accessed when needed, for example through a remote procedure call. When the call is finished, the connection to the component is severed. For each subsequent call, a new connection has to be established. With dynamic linking, components can be updated independently from the applications that they are used in, but the update might not be effective unless the application has been restarted. Each component is loaded into memory after it is needed for the first time, and stays loaded until the application is shut down. With service access, whichever component is installed when the service call is made will be used. The performance overhead that is incurred is that of a remote procedure call. If a given service is used rarely, the overhead will be lower than with dynamic linking, but if it is used often, the overhead can be significantly higher.

Message based communication is the second stage of service reuse. Service reuse through procedure calls avoids linking the components while they are not communicating, but over the duration of the call, the components cannot be exchanged. Messaging makes components exchangeable at all times [8]. If an appropriate messaging infrastructure is provided, messages can be stored and resent in the case that the receiver is temporarily unavailable or does not respond. The performance overhead of message passing is significantly larger than of procedure calls. Also, it increases program complexity, because mechanisms to handle asynchronously arriving messages have to be present; programs cannot rely on messages arriving in a given order.

The third stage of exchangeability in service reuse is broadcasting, or implicit invocation [6]. Whereas messaging as in stage 2 is point-to-point communication with a limited number of receivers, broadcasting means that the application that requires a service sends this request as an event to all other applications. If one of them is able to fulfill the request, it returns a reply. The effect of broadcasting is that the number and availability of service providers is completely transparent to the requesting application. A publish- and subscribe-mechanism can make this more efficient, but the overhead still includes all the overhead of message passing plus the overhead that is created through the

potential multiplication of service providers; i. e., each service may be provided by more applications than necessary.

2.1.3 Discussion

Service reuse provides a higher level of exchangeability than code reuse, but its use is limited. It can provide data, or the results of computations, but it has only limited facilities to provide new data types or new behavior, as code reuse can. Service reuse is instance-oriented, whereas code reuse is type-oriented. As a consequence, service reuse is only practical for rarely used services that return results with a simple structure. It requires all data types to be converted to those data types that are known to the common platform of both communicating components (character strings are typically used); since no code is exchanged, custom data types cannot be used. Often-used services or services with complexly structured results will have to be integrated with the application as code. The tight coupling between components that service reuse avoids is traded off with a tight coupling between the components and their common platform. At the same time, a certain amount of functionality has to be replicated, since it has to be available at both communication partners, which makes maintenance harder. For these reason the required commonalties between the platforms should be kept as small as possible.

Java Remote Method Invocation [10] is an interesting combination of code reuse and service reuse. It is a language-specific remote procedure call mechanism and it can automatically load code that is not available at the destination of a call, but is necessary to execute the call. This typically happens when the call has parameters with polymorphic types. As with RPCs, the time during which the application is connected to the called component is limited to the duration of the call; as with dynamic linking, complex data types can be used for communication. The disadvantage of RMI compared to RPCs is that it does not support interoperability; both communication partners have to be Java programs. Its disadvantage versus dynamic linking is that it still requires objects to be marshalled; apart from the overhead, this means that object identity is lost.

2.2 Increase of Component Distance

Historically, the physical distance between components has increased. Increased distance usually causes looser coupling, because communication costs increase with distance. Thus, increase in component distance leads to an increase in application evolvability.

The trend of increasing component distance is linked to the increase in component exchangeability that was described above. Generally, the more distant two components are, the more exchangeable they are. This is caused by the fact that geographically distant systems are often administered by different people, making it necessary to be able to exchange or update components independently. We distinguish three stages of component distance: file, file system, and network.

In the first stage, all components of an application are contained inside one file. This corresponds to stages one and two of code reuse. Service reuse at file scope is simple procedure calls between modules in the same file; since this is not possible without code reuse of stage one or two, we do not consider this a service reuse stage of its own. To exchange a component, the file has to be rebuilt. The distance between components is zero.

In the second stage, components are spread out over a file system, which can either be local or distributed. Here, components can be exchanged by file system operations (such as moving, copying and deleting files), which are typically much more accessible and usable than the various functions of compilers, linkers, and similar tools that are needed to rebuild files in stage one.

The third stage of distance is the network, i. e., a system of multiple file systems that are owned and administered by different organizations. Components can be anywhere in a local or wide-area network. Reuse through networks that are not spanned by a file system is still rare or experimental. Applets and mobile agents are examples of code reuse here; various Internet protocols provide service reuse on a wide-area scale. In this stage, applications have to deal with high communication cost, potential network failures, and potential unavailability of components, so that coupling between components is typically low. Because of the global nature of the Internet, network communication protocols are typically highly standardized, so that individual components of a distributed application can easily be exchanged.

3. FUTURE DIRECTIONS

The current state of the art in code reuse is dynamic linking on a file-system scale. The discussion above shows a path to the future direction of evolution: dynamic linking on an Internet scale, as described above. This will move installation and maintenance effort from the local system administrator to the manufacturer of the component. The development of "self-installing" components fits well into another trend of software technology, the trend to put more and more information into components. Components contain not only code, but also assertions, documentation, and other forms of self-description. In the same way, components will be able to install themselves through the network. Complex applications composed out of independent components will be hard to maintain; dynamic linking on an Internet scale will automate most of the maintenance tasks.

WREN [5] is a prototypical component-based development environment developed by us that supports component self-description and Internet-wide dynamic linking. WREN allows an application developer to search for components in remote repositories, select components, compose them into an application, and execute the application. It maintains a logical connection to the master copy of a component in its repository, so that it can retrieve updates automatically.

Internet-wide dynamic linking as implemented in WREN is similar to the way Web pages are accessed. There is only one logical copy of each Web page. Pages are not copied to local systems; instead, clients always access the original page through the network. Pages can be cached to improve performance, but this happens internally and is completely transparent to all parties.

The current state of service reuse is procedure call or message passing; scaleable broadcast systems are still experimental [1]. They are, however, a prerequisite of dynamic linking on an Internet scale, since the linker has to be notified of component

updates. For many systems, code reuse will not be possible, either because the amount of code and data is too large to be transferred or because code and data change too quickly. This is often the case with services that are provided through the Internet; for example, search engines, library catalogs, or weather forecasts. In these cases, service reuse is the only possibility. Since broadcasting is the form of service reuse that provides the highest degree of evolvability, we believe that systems will evolve in this direction.

It is obvious that the trend is going towards systems that are more and more distributed. The increasing availability of services on the Internet, wireless computing, and ubiquitous computing all work in this direction.

Strong mobile code [3] may turn out to be the fourth stage in code reuse. Strong mobile code is code that can change its own location in a network while it is executing, and the execution state is moved together with the code. While dynamic linking establishes a connection to the reused component for all of the execution time of the application, mobile code technologies might make it possible to reduce the connection time. It might be possible to combine strong mobility and broadcasting (the highest identified stage of service reuse) in the same way that RMI combines dynamic linking and remote procedure calls.

4. CONCLUSION

We believe that reusable software components are a promising technology. But to make reuse happen, composition mechanisms must provide for application evolvability. Off-the-shelf components that cannot be maintained will not be used.

The trends we described show the direction into which the construction of evolvable applications is evolving. Dynamic linking, event broadcasting, and Internet-wide distributed programs have been recognized as the next good things before, but only putting them into a historical framework shows the driving factors behind this development.

Further, this framework helps to identify areas of future research. It seems to be promising to work on identifying the next stages in this evolution.

5. ACKNOWLEDGMENTS

This effort was sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061; by the National Science Foundation under grant number CCR-9701973; and by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0599. The U. S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research, the Defense Advanced Research Projects Agency, the Air Force Research Laboratory, or the U. S. Government.

REFERENCES

[1] Carzaniga, A. *Architectures for an Event Notification Service Scalable to Wide-Area Networks.* PhD Thesis. Politecnico di Milano, Milan, 1998.

[2] Franz, M. Dynamic Linking of Software Components. *Computer* 30, 3 (1997), 74-81.

[3] Fuggetta, A., Picco, G. P., and Vigna, G. Understanding Code Mobility. *IEEE Transactions on Software Engineering* 24, 5 (1998), 342-361.

[4] Harrison, C. G., and Stern, E. H. Alpha Services—An Experiment in Developing Enterprise Applications. Accessed June 2001 at http://www.isr.uci.edu/events/twist/twist2000/statements/harrison-stern.doc. 2000.

[5] Lüer, C., and Rosenblum, D. S. Wren—An Environment for Component-Based Development. In *Joint 8th European Software Engineering Conference (ESEC) and 9th ACM Sigsoft International Symposium on the Foundations of Software Engineering (FSE-9).* Vienna, 2001, to appear.

[6] Notkin, D., Garlan, D., Griswold, W. G., and Sullivan, K. Adding Implicit Invocation to Languages: Three Approaches. In *Object Technologies for Advanced Software.* Springer, Berlin, 1993, 489-510.

[7] Oreizy, P. Decentralized Software Evolution. In *Proceedings of the International Conference on the Principles of Software Evolution (IWPSE 1).* Kyoto, 1998.

[8] Oreizy, P., Gorlick, M. M., Taylor, R. N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D. S., and Wolf, A. L. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems* 14, 3 (1999), 54-62.

[9] Voas, J. Maintaining Component-Based Systems. *IEEE Software* 15, 4 (1998), 22-27.

[10] Waldo, J. Remote Procedure Calls and Java Remote Method Invocation. *IEEE Concurrency* 6, 3 (1998), 5-7.