~ *By* Ahmed Elfatatry

DEALING WITH CHANGE: COMPONENTS VERSUS SERVICES

Understanding the philosophy behind components and services is necessary before choosing a solution to deal with change in software design.

Underlying almost every software design is a philosophy of perception, abstraction, and decomposition. Such a philosophy claims that adopting a specific abstraction and decomposition techniques will lead to a better design. In the context of dealing with change, we contrast two design philosophies: the component approach and the service approach to software development and analyze the differences between them in addressing change.

The Problem: Change

Business Perspective. To businesses, dealing with change is a fact of everyday life that must be exploited and enabled. Mergers, acquisitions, and the introduction of new technologies are examples of drivers for change in business environments. Business agility refers to the ability of an enterprise to thrive in a continuously changing and unpredictable environment [3, 10].

Knowing what aspects are more likely to change and what

aspects are not is vital to dealing with change. While many things change in business, some elements tend to remain constant. The core competencies of a business are relatively stable over the medium term; changes may affect how businesses operate as a result of changing business procedures or the introduction of new technologies. Over the long term, almost every aspect of the business is subject to change.

Technical Perspective. To programmers, change has been seen as an evil to be evaded, rather than an opportunity to be embraced as it has a negative effect on the goal of completing the software system. The ability to make significant changes to design and behavior throughout the evolution of a system is a major differentiator of software development from other engineering disciplines [12]. Generally, software flexibility is the term that describes the ability of a software system to adapt change. Design time flexibility is the ease of changing software (which includes but is not limited to code change) with minimal cost in terms of time and effort. Runtime flexibility addresses the capability of a software system to change requirements on the fly, with no need to change code.

While it is almost a fact of software design that business systems requirements always change, not all software development methodologies account for such a fact. Different approaches have different philosophical beliefs with respect to how a system should be decomposed in order to tackle change.

In the 1970s, structured analysis evolved as a response to dealing with complex systems that are developed cooperatively by a number of programmers. Structured analysis was mainly based on functional decomposition [6]. A top-down functional decomposition starts with a top-level description of a system and then refines this view step by step. With each refinement, the system is decomposed into lower-level and smaller modules. Top-down decomposition requires identifying the major higher-level system requirements and functions, and then breaking them down throughout the successive steps until function-specific modules can be designed.

While functional decomposition has been successful for more stable types of systems, it has been less effective in dealing with business changes and consequently with system maintenance. To change a data structure it is often necessary to change all the functions related to that structure. Thus, the system easily becomes unstable as a slight modification can generate major knock-on consequences.

The object-oriented paradigm addressed issues of reuse and maintenance by encapsulating data and its corresponding operations within a class. The concept of objects in the problem domain had a higher chance of being stable than data structures and functions; hence the overall architecture of the system will normally be stable [4]. Moreover, due to the very foundation of the object-oriented paradigm, changes of internal details do not spread into the system architecture.

Software development methodologies can be broadly classified into two main categories: requirement anticipation and requirement adaptation methodologies. The first assumes that it is possible to identify and solve almost all problems before coding. The latter adopts a more pragmatic approach that believes business system development is an incremental process and changes are an inescapable aspect of software design and are expected to occur in every stage.

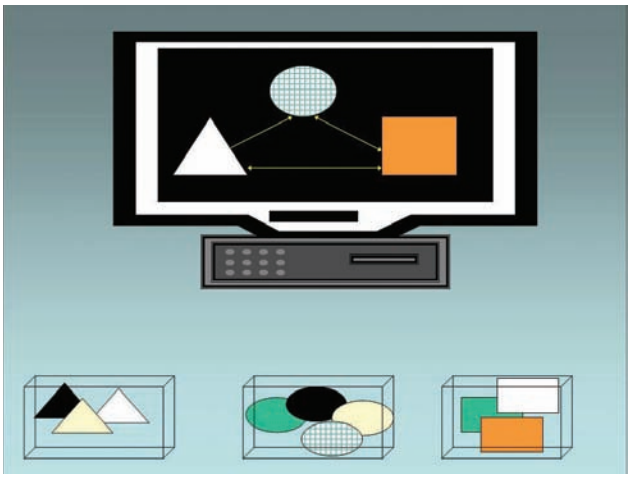
To meet constantly changing business requirements software systems must be in constant evolution. Consequently, the separation between the process of software development and software maintenance is becoming less significant. Here, we contrast two design approaches that claim to support continuous software evolution: component-based development and service-based development.

To meet constantly changing business requirements
software systems must be in constant evolution.

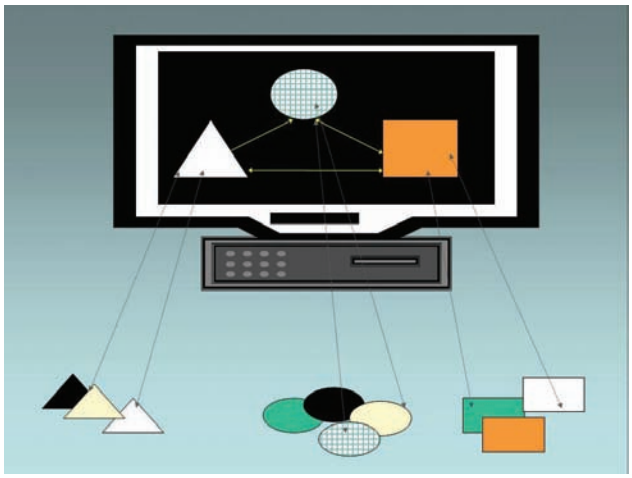
Accommodating Change: Components vs. Services

Producing flexible software is a result of a combination of factors, both technical and non-technical. The difference between components and services in dealing with change is influenced by the factors discussed here.

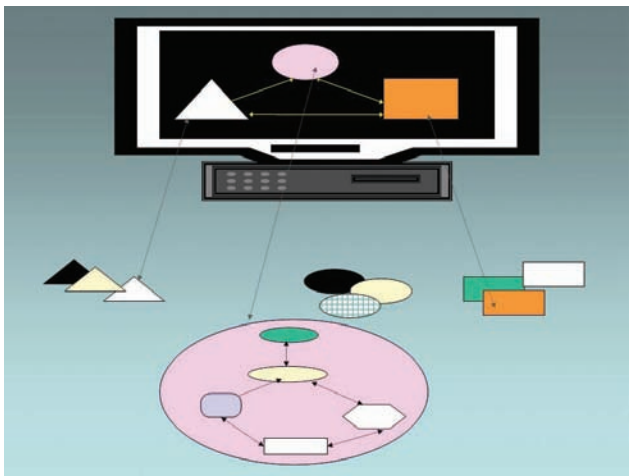
Philosophy. The idea of component-based development is to industrialize the software development process by producing software applications by assembling prefabricated software components. In response to change and evolving requirements, two basic ideas



1.



2.



3.

(top to bottom) Figure 1. Component-based development. Figure 2. Service-based model. Figure 3. Composite service.

underlie component-based development. Firstly, that software development can be significantly improved if applications can be quickly assembled from prefabricated software components. Secondly, that an increasingly large collection of interoperable software components will be made available to developers in both general and specialist catalogues.

Logical separation of need from the need-fulfillment mechanism is at the heart of the service model. For example, when a customer reserves a train ticket from place A to place B, he neither controls the operation of the train nor chooses the crew. In such a case, the customer is only interested in the outcome and has no control over the mechanism by which the outcome is achieved. A service is defined as: "Any act or performance that one party can offer to another that is essentially intangible and does not result in the ownership of anything. Its production may or may not be tied to a physical product." In software, this is known as "loose coupling" [9]. A software service is a coarse-grained, discoverable entity that exists as single instance and interacts with applications and other services. The idea of a service differs from the concept of a component by the fact a service does not define any structural constraints but the interface.

Binding. Although the service-oriented model of software and component-based development share common characteristics, there are also major differences. The characteristic they share is that parts of a software system can be developed separately and then added to the system later (bound). However, their approach to binding is substantially different. The main component-based software assumes early binding of components; that is, the caller unit knows exactly which component to contact before runtime.

Service-based development adopts a more flexible approach where the binding is deferred to runtime, enabling the change of the source of provision each time. The service approach not only allows flexible change in providers but also accommodates the change in the quality of the requirements over time [1, 2]. The difference between the two models is explained further by Figures 1–3.

In Figure 1, different shapes represent different functionalities, illustrating that in component-based development, software components are "taken out of the box" and then plugged into the system, probably with the addition of some "glue" code. In such a case, the exact source of the required functionality is determined before runtime.

Figure 2 shows the dynamics of a service-based application. An application may be composed of a number of services (shapes: square, circle, or triangle). For each service, there may exist a number of providers who offer the same service but with different combinations of quality characteristics (colors). Each time a service is invoked, a different provider may be chosen to negotiate terms and conditions, and then the service is finally bound. The dotted arrows

indicate a case of loose coupling between the provider and the consumer of the service.

Figure 3 depicts a case of a composite service. Here, the service is composed of a number of different services combined to provide a certain outcome. However, such composition is transparent to the service consumer.

Abstraction and Granularity. An influencing factor on the mechanisms of software change is the granularity of the change. Granularity refers to the scale of the artifacts to be changed and can range from very coarse to medium to a very fine degree of granularity [5]. Granularity is a relative concept that can be precisely defined only in a specific context. For instance, if a service implements all functions of a banking system, then it can be considered coarse grained. If it supports just credit-balance checking, it is considered fine-grained.

After the object revolution of the early 1990s, it was obvious that object-oriented technologies were not enough to cope with the rapidly changing requirements of real-world software systems. While object-oriented methodologies provided rich models to describe problem domains, this was not enough to adapt changing requirements. Specifically, objects were too fine-grained and did not make a clear separation between computational and compositional aspects. Components were then proposed to encapsulate the computational details of a set of objects.

Services should be published at a level of abstraction that corresponds to a real-world activity or recognizable business function. The appropriate level of granularity for a service and its methods is relatively coarse. A service generally supports a single distinct business concept or process. It contains software that implements the business concept so it can be reused in a similar context.

Delivery mechanism. The difference in the delivery mechanism between components and services is probably the revolutionary concept of the whole story. In the main, software engineering has concentrated on the provision of technical and management support for the production of software as a product-oriented concept. Components are product-oriented, where software is delivered on CDs or other media. However, the proliferation of Internet-based computing has brought about new concepts, opportunities, and challenges, not only in terms of a wide range of general service provisions, but also an opportunity to rethink the methods and modes for delivering software. The key benefits of delivering software as a service include the potential for increased business agility through loose coupling and the ability to evolve as business requirements change.

In a service-oriented model, software functionality is delivered as a service, where functionality is required each time, service elements are identified, terms and conditions are negotiated, executed, and then "discarded." This allows flexibility for change even at the level of the smallest unit of functionality. In addition to the differences in the technical model, delivering software as a service brings about new business models built on the opportunities made available by such vision. Examples include business models for billing software services, rules for service negotiation, and trust assessment and provision.

Architecture. A component architecture is a specification of a set of interfaces and rules of interaction that govern the communication among components. Most component architectures represent a case of tight coupling [8]. For instance, in CORBA (a component-based architecture), there is a tight coupling between the client and the server as both must share the same interface with a stub on the client-side and the corresponding skeleton on the server side. Also, most implementations of component-based architecture have been closed systems in the sense they could only handle proprietary technology.

Service-oriented architecture (SOA) is a way of designing a software system to provide services to either end-user applications or other services through published and automatically discoverable interfaces. Service consumers are decoupled from service providers by a broker. A SOA adds a layer of abstraction on top of existing IT environments. Typically, a service layer can be added over a component infrastructure.

Challenges

Achieving software flexibility through components or services involves both technical and nontechnical challenges. Such challenges must be addressed before a solution becomes a commercial reality.

Trust. In a software context, trust is the confidence that a component or service will provide its functional and nonfunctional obligations as promised in the description associated with it [7]. Testing a component by examining the source code is not a practical solution. However, trusting a component from an unknown source may be partially solved by testing it several times before usage. In addition, any changes made to the source code may invalidate the componentís contractual specification [11].

The trust issue is much more complicated in the

case of service-based development as it is very difficult to predict the compliance of a provider to the agreed service level. When software is delivered as a service, the service-level agreement (SLA) must be monitored for compliance. This problem becomes even more complicated in the case of a service composed from other services. In such a case, the final quality of the service will depend on the service quality of the constituent services.

Composition management. Composing a system from a number of components is relatively controlled compared to dynamic service composition. As more service providers expose their services in a large distributed system, it becomes infeasible for humans to manage and compose them manually; this process must be fully automated. Associated with such an open environment are the problems of managing rollbacks, billing, licensing, and transactional semantics.

Adaptation vs. advanced finding. Component selection is a design time activity, which may subsequently need some kind of adaptation. Such adaptation is sometimes referred to as glue code. In service-based development, service discovery and selection take place at runtime; that is, when the source of provision is determined. This makes testing a service before usage almost impractical, as the source of the service as well as the conditions for usage may vary between two consecutive invocations.

Specification. The promise of automatic discovery in service-based development is the most significant advancement over its predecessor, component-based development.

A major limitation of building flexible software using components has been the way components are specified. Proprietary standards and implementation-dependant specification of components have hindered component-based development from achieving its primary goal facilitating reuse.

The linking point in a SOA is the service specification and not the implementation. This provides implementation transparency and minimizes the impact of change on software systems.

Implementation efficiency. The crucial concept of late binding is inherent in the SOA. While implementing such a concept facilitates flexibility, it results in execution overhead, especially if service discovery and matching will be done each time a functionality is invoked.

Conclusion

Evolution is critical in the life cycle of many software systems, particularly those serving highly volatile business domains. Components and services, although similar, are not the same; they have different philosophies and abstractions. Evolution is supported by both component- and service-based development. The differences in philosophy and level of abstraction make services a better solution to change.

Suggesting that all future software will be service-based is closer to hype than it is to practicality. Indeed, the idea of services is suitable for systems with frequently changing requirements that can tolerate some kind of implementation inefficiency. While components are a good way to implement services, an ideal component-based system does not necessarily yield an ideal service-oriented system. Hence, services will not completely replace components, but rather complement them. C

References

1. Bennett, K., Layzell, P., Budgen, D., Brereton, P., Macaulay, L. and Munro, M. Service-based software: The future for flexible software. In *Proceedings of the 7th Asia-Pacific Software Engineering Conference*, (Singapore, 2000). IEEE.
2. Budgen, D., Brereton, P. and Turner, M. Codifying a service architectural style. In *Proceedings of the 28th Annual International Computer Software and Applications Conference*, (Hong Kong, 2004). IEEE, 16–22.
3. Conboy, K. and Fitzgerald, B. Toward a conceptual framework of agile methods: A study of agility in different disciplines. In *Proceedings of the ACM Workshop on Interdisciplinary Software Engineering Research*, (Newport Beach, CA, 2004). ACM Press, NY, 37–44.
4. Costagliola, G., Ferrucci, F., Tortora, G. and Vitiello, G. Class point: An approach for the size estimation of object-oriented systems. *IEEE Trans. Software Engineering 31*, 1, 52–74.
5. Costagliola, G., Francese, R., Risi, M., Scanniello, G., and Lucia, A.D. A component-based visual environment development process. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*. ACM Press, NY, 2002, 327–334.
6. DeMarco, T. *Structured Analysis and System Specification*. Prentice-Hall, Englewood Cliffs, NJ, 1979.
7. Elfatatry, A. and Layzell, P. Negotiating in service-oriented environments. *Commun. ACM 47*, 8 (Aug. 2004), 103–108.
8. Gore, P., Pyarali, I., Gill, C.D. and Schmidt, D.C. The design and performance of a real-time notification service. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, (Toronto, Canada, 2004), IEEE.
9. Jobber, D. *Principles and Practice of Marketing*. McGraw-Hill, NY, 1998.
10. Padmanabhuni, S., Ganesh, J. and Moitra, D. Web Services, grid computing, and business process management: Exploiting complementarities for business agility. In *Proceedings of the IEEE International Conference on Web Services*, (San Diego, CA, June 6–9, 2004). IEEE.
11. Srivatsa, M., Xiong, L. and Liu, L. TrustGuard: Countering vulnerabilities in reputation management for decentralized overlay networks. In *Proceedings of the 14th International Conference on World Wide Web*, (Chiba, Japan, 2005). ACM Press, NY, 422–431.
12. Zimmermann, T., Weisgerber, P., Diehl, S. and Zeller, A. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, Washington, DC, 2004, 563–572.

Ahmed Elfatatry (ahmed@katwa.net) is a lecturer of software engineering in the Department of Information Technology at the University of Alexandria, Egypt.

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

© 2007 ACM 0001-0782/07/0800 $5.00