Editor: Martin Fowler ■ ThoughtWorks ■ fowler@acm.org

Components and the World of Chaos

Rebecca Parsons

Many writers espouse the idea that commercial software development will soon consist of tribes of monkeys assembling commercially available components, resulting in faster development, significantly reduced costs, and more reliable software. Although we all dream that complex applications development will eventually become faster, cheaper, and better, realizing this dream with components as currently conceived has some fundamental flaws.

Components

As is often the case with trends in the IT industry, the term *component* has too many meanings. For the purposes of this discussion, a component

- Has a simple, well-defined interface
- Is an object, meaning that the data and methods are combined as a unit
- Exhibits a degree of specialization of functionality (often obtained through configuration), with the appropriate range of lifecycle methods to support the desired functionality
- Is designed with the expectation of reuse, although the reuse context is unpredictable

Additionally, a system built with components achieves complex functionality through the interaction of various components.

Sounds ideal. According to this, each simple component should be easy to specify, describe, create, and test. Encapsulation should ensure that components don't interact in ways other than through the interfaces or assembly framework. Also, tests should be able to demonstrate that the component respects its interface, and no part of a system should be terribly complex. Additionally, suitably designed components should be more amenable to reuse than monolithic systems.

The complex development effort required for building component systems involves both the creation of these components and their aggregation into systems. In this view, the hard work of software development moves from building complete systems to designing and creating these individual components, but even this is easier given the boundaries on components. The difficulty in software development's assembly phase involves identifying the appropriate components to use rather than creating complex software. Some even envision global repositories of components that existing systems monitor to find new or improved components that perform desired functions. You could then incorporate these new components dynamically into an existing system.

Problems

However, problems exist, even without considering the impact of systems dynamically changing their component configuration and use. Assembling components requires

- Initially identifying the appropriate components to implement the desired functionality
- Determining and resolving gaps between the desired functionality and the components' functionality
- Specifying the component interactions

0740-7459/03/$17.00 © 2003 IEEE

This requires a language that can sufficiently describe the components' semantics so that the component assembler can identify the desired components and see how to connect them to achieve the desired systems functionality. The assembler must also understand how the component assembly framework interacts with the components. Neither these activities nor the tools that support them, nor the skills required to use them, are trivial.

Another problem with components concerns their nature. One of the component model's primary strengths is that it separates large systems into simpler components interacting through simple interfaces. Other types of systems share this strength. Work in the nonlinear systems field has shown that the characteristics of simple interfaces and the interaction of simple entities through those interfaces can give rise to emergent behavior.

Emergent computation

Emergent behavior, or emergent computation, describes the appearance of global computation or problem solving from a system of distributed, discrete computational systems. It is a computational ability that is not specifically programmed but rather emerges at a high level when simpler components interact. Researchers in this field are often startled at the level of complexity that arises from very simple computational structures. In systems with emergent characteristics, the system's global behavior differs from what designers anticipated. These failed expectations are often in the behavior's specifics (the particular solutions generated, for example); however, these surprises can also deviate from expectations in more substantial ways.

It is easy to dismiss this notion as something akin to the latest science fiction thriller. Clearly, the components used in software systems may not interact in arbitrary ways. They can't alter their internal behavior and act autonomously, and systems using components are not designed to learn new behaviors. The components still exist in the particular framework originally specified, so these systems should still behave tractably.

We don't need direct parallels to emergent computation to see that constructing component systems is difficult. Despite significant investment in tools and formalisms, delivered systems still contain significant defects, even when measured against expected behavior. Defining expected behavior at the component level is not sufficient to ensure correct behavior at the global level, even when systems are constructed using components. Specifying the component solution's expected behavior requires specifying the results of the components' interactions with each other. The component framework communicates with the components and external environment.

Emergent computation in similar settings should be enough to convince us that describing the expected behavior of such a complex of interacting components is challenging at best. The most rigorous component-level testing will not tell us enough at the system level to provide any assurances that we've achieved the desired result. So, we must resolve the issue of understanding the global behavior. We must also face how the computational power inherent in components' interacting systems affects our ability to construct systems.

Constructing applications

Application development in the component world involves three categories of development: designing and developing component implementation and constructing applications. We focus on the latter—constructing application components and applications using a component framework and a suite of existing components. Designing a good component involves considering several factors, including appropriate encapsulation, clean interfaces, appropriate granularity of functionality, appropriate balance of specificity and reusability, and completeness of functional coverage. The developers responsible for creating the component must understand the component framework as well as the required component behavior. The test design of the individual components is the most straightforward of these activities. If the component is designed properly, the interfaces should be simple and completely specified and the behavior should be properly encapsulated. (I've already described the difficulty of constructing and testing applications created using components.)

The most rigorous component-level testing will not tell us enough at the system level to provide any assurances that we've achieved the desired result.

Although components have potentially altered the desired balance of skills for developers, they have neither radically simplified the task nor radically altered the basic skills and training needed. The skills for component designers and developers parallel those needed for large-scale object-oriented design and development. Good objects and components share many design characteristics, such as clean interfaces, functional cohesion, adaptability, and completeness. Similarly, developing these systems requires skills in performance, readability, and extensibility.

Designing and developing good components requires even more insight and vision into the application domain's needs. It might seem that global systems understanding becomes less necessary in the component world. However, the issues with emergent behavior demonstrate the need for both a global understanding of the component interactions and a local understanding of individual components' behavior. This level of understanding is more

common in developers who understand the behavior of heterogeneous distributed and parallel systems. We are not looking at a significant reduction in the skills necessary to develop systems in this model. In the world of components, components are not easy to create properly, and applications are not easy to create, even when using well-designed components.

The ideal world of systems development by simple-minded composition of existing components doesn't exist. Application development is still a difficult undertaking. We can't resolve the issues with testing and ensuring correct behavior by simply invoking the power of components. The power of simple component interaction contributes to the component development process's complexity, requiring greater developer skills. However, this same complexity and power provides intriguing possibilities for harnessing the potential emergent properties of components in the development process itself. But that's another story.

Rebecca Parsons is a technology principle and senior architect at ThoughtWorks. Contact her at rjparson@thoughtworks.com.

SOFTWARE ENGINEERING GLOSSARY

Richard H. Thayer ■ California State University, Sacramento ■ thayer@csus.edu

Requirements engineering domain
(cont'd from inside back cover)

functional requirement: A system or software requirement that specifies a function that a system/software system or system/software component must be capable of performing. These are software requirements that define system behavior—that is, the fundamental process or transformation that the system's software and hardware components perform on inputs to produce outputs. Contrast with *nonfunctional requirement*.

performance requirement: A system or software requirement specifying a performance characteristic that a system/software system or system/software component must possess—for example, speed, accuracy, and frequency.

external interface requirement: A system or software requirement that specifies a hardware, software, or database element with which a system/software system or system/software component must interface, or that sets forth constraints on formats, timing, or other factors caused by such an interface.

design constraint (requirements): A software requirement that impacts or constrains the design of a software system or software system component. Examples of design constraints are physical requirements, performance requirements, software development standards, and software quality assurance (SQA) standards.

quality attribute (requirement): A requirement that specifies the degree of an attribute that affects the quality that the system or software must possess—for example, reliability, maintainability, or usability. See also software quality attribute (requirement).

requirements specification: In system/software engineering, a document that states the functions that software must perform, the required level of performance (speed, accuracy, and so on), the nature of the required interfaces between the software product and its environment, the type and severity of constraints on design, and the quality of the final product. Synonymous with *external specification*. See also software requirements specification.

nonfunctional requirement: A software requirement that describes not what the software will do but how the software will do it—for exam-ple, software performance requirements, software external interface requirements, software design constraints, and software quality attributes. Nonfunctional requirements are sometimes difficult to test, so they are usually evaluated subjectively. Contrast with *functional requirement*. Sometimes referred to as *design constraints*.

software specification review (SSR): In software system engineering, a joint acquirer–supplier review conducted to finalize software configuration item (SCI) requirements so that the software developer can initiate the next step in the software development process. The SSR is conducted when SCI requirements have been sufficiently defined to evaluate the developer's responsiveness to and interpretation of the system- or segment-level technical requirements. A successful SSR is predicated on the developer's determination that the software requirements specification and interface specifications form a satisfactory basis for proceeding to the preliminary design phase. [Military Std. 1521B-1985]

software design specification: A document that specifies the design of a system or component. Typical contents include algorithms, system or component architecture, control logic, data structures, I/O formats, and interface descriptions. Also called *software design description*, *internal specifications*. Contrast with *software requirements specification*, *external specifications*. [ANSI/IEEE Std. 610.12-1990]

software requirements phase: The software development lifecycle phase during which the requirements for a software product, such as functional and performance capabilities, are defined, documented, and reviewed.

requirements traceability: The identification and documentation of the derivation path (upward) and allocation/flow-down path (downward) of requirements in the requirements hierarchy. See also traceability.

—Continued on p. 88