LESSONS LEARNED THROUGH SIX YEARS OF COMPONENT-BASED DEVELOPMENT

Adjusting to new processes and techniques is an important aspect of project success.

MICHAEL SPARLING

There must be some truth to the theory that software development in "Internet time," much like a dog's life, actually compresses seven years of experience, effort, and activity into a single year. It has been six years since my organization, Castek, embarked upon our voyage of component-based application development. In that time we have successfully delivered a series of large, enterprise-class applications using component-based development and related techniques. These systems have encompassed many parts of the financial services application domain including bill presentation and reconciliation, debt management, transportation, and insurance policy administration/claims processing. We have used pre-existing components in each of these projects and have seen our practices, development approaches, and tool utilization improve with each project. What follows is a set of lessons we've learned over the years.

The terms component and component-based development have become overloaded—some might say contentious—during the past few years. A quick Internet search reveals a variety of definitions including the following:

- A chip, package or other object to which wires must be routed [2];
- A subsystem not bound to any specific application, producing a high-quality product that can be customized and used in several places [6];
- A unit of packaging, distribution, or delivery that provides services within a data integrity or encapsulation boundary [4].

Since "component" means different things to different people, for clarity let me quickly define how I am using the term for the duration of this article:

A component is a language neutral, independently implemented package of software services, delivered in an encapsulated and replaceable container, accessed via one or more published interfaces. While a component may have the ability to modify a database, it should not be expected to maintain state information. A component is not platform-constrained nor is it application-bound. [5]

We have adopted a formal specification approach that applies a clear distinction between the specified view of the behavior (*what* we plan to do) as a separate form from the implementation of the specification (*how* we plan to do it). Thus, we say "a given component specification may be realized in one or more implementation technologies," meaning that components implemented in Java or Visual Basic are functionally equivalent if they implement the same specification. This statement has been reasonably simple to make given our historical development tools, but it has become increasingly difficult to adhere to as we have adopted newer implementation tools and approaches.

Historically, we built solutions with 4GL-based CASE tools. Our primary development tool has had minimal to nonexistent support for components. Through the use of naming standards, alternative process approaches, a large amount of ingenuity, and an organizational desire to succeed with components, we have been successful in practicing component-based development (CBD). The past year has seen us transitioning from our traditional 4GL tools to Enterprise JavaBeans (EJBs) and COM+. Throughout this change in technologies we've benefited immensely from the following lessons we've learned in the creation and management of component-based systems. These lessons apply to component-based development in general, whether using development tools and programming languages that treat components and interfaces as first-class elements or not.

Challenges are Associated with the Adoption of CBD

The adoption of component-based development brings with it many changes that touch beliefs and ideas considered core to most organizations. These adjustments, and the approaches taken to resolve contention, can often be the difference between succeeding and failing in a component-based undertaking.

Depending on the team size and the project timelines, incremental development can be difficult. While detached subsystems may be analyzed, designed, and developed incrementally you need a clear picture of the end state before beginning to work on the components, user interfaces, and collaborations used to meet these goals. Some of the activities in a traditional waterfall approach are difficult to execute, especially if you are employing parallel development.

Lesson 1 is the importance of a component reference model, which serves as a guide through analysis and development; Lesson 2 concerns things to watch for in parallel development.

Component-based development changes the way your project teams behave. There is a natural tendency in most software developers to want to develop software—that's what they believe they're paid for. The component-based approach requires that developers accept the importance of working with the components as encapsulated black boxes and not attempt to repeatedly rebuild them. Conversely, the developers building components have to balance the desire to create reusable components with the realities of the application requirements available at the time the component is developed.

Lesson 3 examines the pros and cons of reuse, while Lesson 4 discusses immutability as it relates to components.

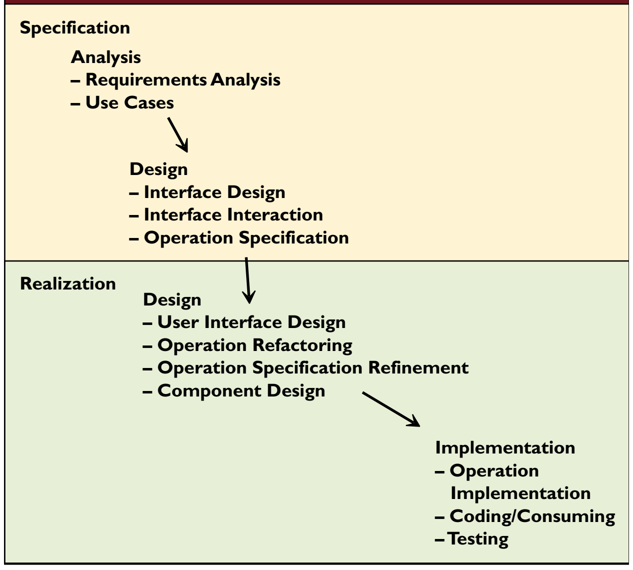
Database administrators can be some of the hardest people to sway to a component-based approach. When you propose an application should not be composed of a single integrated database but as a collection of components, where each component may implement an isolated data store, we've experienced considerable pushback. Plan to spend extra time working with the data management group to educate them on the value of CBD and how a component must be encapsulated at all levels, including any data it may be required to persist in a database.

Lesson 5 looks at how prototypes are an important part of selling CBD to an organization and succeeding with CBD projects. Often these prototypes are instrumental in demonstrating how the partitioned databases would look when the system is deployed and why they are important.

Network architecture and hardware assumptions may also be challenged. To be successful with CBD we've found a company needs to think like an Internet corporation rather than a traditional information technology organization. For example, the Web site Google.com runs its entire operation on a network of 4,000 servers, consisting of no-name PCs running the Linux operating system. Google's rationale for this decision is that whatever it loses in functionality or robustness can be made up in cost savings, programming flexibility, economy of space, and ease of maintenance [7]. In a component-based system, scalability and reliability predominantly come from clusters of commodity hardware, not simply from hand-optimized code. There is a limit to the payback companies will get from working to engineer better code, when that time could be spent designing the solution. Incorporating new ways of thinking about computer hardware and application architecture is necessary to succeed with CBD.

Lesson 6 describes our experience with error and exception handling and how they may be different in a networked solution, while the final lesson we've learned, Lesson 7, looks at the changes to testing strategies brought about by CBD.

Conceptual steps in a CBD/e project [3].



Traditional versus component life cycles [3].

Basic Project Process	Component-based Approach
Understand the requirements	Understand the requirements
Complete analysis and design	Specify the components and their interactions
Code	Parallel design
Test	Parallel coding
Fix bugs	Test each component in isolation
Retest	Integrate components with application workflow
Deploy	Application testing Deployment

Lesson 1. Base your development on a component reference model. Requirements and scope changes in an information technology project, when managed correctly, are positive if they result in a solution that better satisfies the customer's needs. Change in the deployment targets or the technical standards once the project is under way should be avoided. We learned this lesson during our first full-scale CBD project. This endeavor nearly failed, in part because we kept evolving the principles of component-based development as the project progressed. Refining component modeling standards and incorporating new analysis techniques pushed the project deadlines and frustrated the project team.

An enterprise scale project—one that utilizes the talents of many people—requires a stable set of principles for the project. One of those principles is what we call a component reference model. Our approach to CBD involves a fairly rigorous specification and design stage, as shown in the figure appearing here.

We believe this approach provides us with the ability to create multiple realizations of a given specification, affording us a level of isolation from platform and technology dependencies. This approach also allows us to utilize the specification repeatedly, both in the creation of component implementations and in the assembly of solutions based on components. To achieve this requires an accepted component vocabulary and a set of design standards, which form key deliverables of the component reference model. In addition, we use an abstract method of describing dependencies on the services provided by an EJB container or a COM+ server within the implementation technology. We do this so that our component specifications are not bound to a particular code deployment environment.

Our component reference model is composed of:

- Descriptions of the goals of CBD;
- A set of design principles and modeling standards;
- A standard set of analysis, design, development and testing tools; and
- A uniform set of documentation standards.

The component reference model has been our most valuable resource across each of our successful projects. Our initial CBD project was saved from disaster by great project and departmental management, yet its near failure taught us that the component reference model was the key to success with CBD.

We continue to update our reference model, even when working with the "state of the art" development technologies available today. While projects based on CBD give us a certain degree of freedom that traditional design approaches do not, we now know we must establish the guiding principles for the project before work begins. This is communicated through the component reference model, and then the project teams resist change in techniques at all cost.

Lesson 2. Parallel development. One of the goals we felt CBD would help us achieve was the ability to split our project teams into multiple groups to work in parallel on a given project. Our assumption was that if the system works based on encapsulated components, so can our development teams. Differences between a traditional project life cycle and a component-based approach are shown in the table here, showing how parallel development may be used.

When using CBD, our expectation was that because the entire solution is specified in the beginning, each parallel task would then be able to deliver a completed, tested, and robust component meeting the specification. Likewise, the team responsible for the application user interface, collaborations, and workflow could build to the component specifications, and thus should be able to develop and deliver a complete solution without extensive integration testing. Our early estimates were that this approach could save as much as 25% of the development effort by streamlining the process. If only it was that simple.

The good news is parallel development, facilitated by component-based development, works. However, it becomes increasingly important to ensure the work conducted in parallel is valuable to the project at integration time. Each team must understand their role and contribution to the entire project and how important it is they conform to the specifications. In short, once the architecture and specifications are set, there is not much room for unexpected innovation or project scope creep.

The architecture group has most of the responsibility for ensuring parallel development occurs. After designing the solution, based on the user requirements, they follow through as the custodians of the project direction. They are instrumental in developing the project testing plans, since they understand how the application is expected to operate. As well, technical architects must become the educators and cheerleaders for each of the development teams working in parallel. As educators they ensure team members understand the importance of implementing the specification. They become cheerleaders because as a large part of the project team loses contact with the users and business subject matter experts, the architects must provide a coherent picture of the overall project.

Another challenge parallel development imposes is in the changing roles people play during the development process. A parallel development stream means that many jobs, such as developing code, user interfaces, or collaborations, are based on specifications prepared by a team separate from the developers. We found this process was a different experience for most of the staff, familiar with the traditional waterfall approach, and created a feeling of disconnect from the project goals. One solution we tried was to rotate most people through different roles, while maintaining a project structure that could ensure success. As a result, a business analyst started in the architecture group, moved to the testing team, and finished in the assembly group. Likewise, a programmer gained better project perspective when he was transferred to the testing team. Another solution was to try and shift more of the design work down to the respective teams and away from a central group. While this created a greater feeling of project participation within the parallel development groups, it created the danger of designed specification incompatibility, which required significant attention from the application architects during the integration phase.

Ultimately we expect that as our repository of components increases in size, we will experience less new component development and will instead spend more of the project time engineering user interfaces and component collaborations. This effort can occur much closer to the users, granting the project teams more exposure to the business experts.

Lesson 3. Pros and cons of reuse. While reuse of software assets is only one of the goals of a component-based approach, it is often a key selling point. The idea that an organization could utilize existing software when creating a new solution is understandably appealing. However, the "designing for reuse" syndrome, the creation of disincentives to reusing preexisting code, deters many organizations from successfully reusing enterprise class components. I have found that most developers view building a component as much more glamorous than using one which already exists. The traditional "lines of code" measure of productivity reinforces the syndrome. Prolific programmers can write 10,000 lines of code to develop a new component, while it may require less than 10 lines of code to incorporate a reusable component. One developer is achieving a more reliable result faster, while the other is generating more code.

Another popular reuse fallacy is that every component should be engineered to be reusable, regardless of whether or not an organization can identify what the future requirements of the component will be. Suddenly a great deal of effort is directed into the design and development of components that have no business driver to validate the effort. In these cases, the 80/20 rule often applies: 80% of the utilization of a component is based on 20% of the component's operations. Thus it is important to pay attention to any time where you are tempted to spend extra effort designing and constructing additional functionality, beyond the current application's needs. That extra effort probably falls victim to the same ratio, resulting in an even less appealing return on the investment.

To ensure that appropriate reuse occurs, a component has to be locatable, consumable, and extensible. As a result, a component must have a complete specification, combined with some assurances that it complies with the specification. Ideally, an existing component will come with some metrics and design criteria, so a designer can get an idea of the component's applicability to a task. This type of supporting documentation becomes important when the component is sourced from beyond the current project team.

Lesson 4. Immutability and components. Immutability is one of the most contentious topics I have encountered. The issue can be stated as: "When is a component or interface considered published and, thus, must be considered immutable and when can we allow it to be change without versioning the component or interface?"

At some point a component implementation is released. Provided the implementation has been tested and certified to implement the specification, a consumer should be assured that neither the specification nor the implementation would change without some form of notification. If a change is made to a component operation, modifying its behavior, it is difficult to argue against versioning the interface. We've faced a question though when we've extended an interface, through the addition of operations. Should you version the interface? Versioning would prevent a consumer of the previous interface from receiving an interface different to their expectation. But managing multiple versions of an interface is complex and time-consuming.

We've settled on a managed process that offers a project freedom when working on new components, or extensions to existing components that will result in new versions. At a point determined by the project manager, a statement to the effect that "this specification is frozen" is made, and from that point forward changes that impact an interface require a new version to be created. Any project dependent upon the existing implementation is notified when a new version is made available, and it is up to the project to choose to accept the new version. If the notification is for a revision, the project team is expected to initiate a regression test of their work to ensure that the new revision remains compatible with their needs. We call this approach "modify and notify."

Current implementation technologies such as COM+ and EJB, while simplifying the marshalling and data passing, have done little to address the danger of misinterpretation of the specification. These misinterpretations are what continuously drew us into complex debugging sessions. Our solution was to adopt checksums into the signatures, not to ensure the data marshalling was correct, but to guarantee the provider/sender contract was valid and enforced.

Lesson 5. Prototype early and often. If you look at most current application development tools, you see how visual components have helped user interface designers. You can open Visual Basic, drag components from the tool palette onto a form, and construct a basic user interface. This example is one of my favorite demonstrations of the power of CBD and prototyping. Each of those visual controls is a language-independent package of software services: a component.

Our experience to date has shown us that prototyping, especially the sort of active prototypes that can be built when you have a collection of components at your disposal, helps to ease the adoption of a component-based approach. Since the specification and realization steps are formal, developing the ideas in an experimental stage helps set the requirements for development. Prototypes have many advantages, including the following:

- Establishing the application's goals and component boundaries before we have committed a large numbers of developers.
- Acting as powerful demonstration tools, especially when selling CBD to a new organization.
- They show how distributed components help solve some traditional application problems, such as load balancing, data distribution, and replication.
- They often point out potential problems in the component collaborations.

It is worth noting that not all prototypes exist as programs or program shells. Some of our best successes with prototypes have come through storyboards. When you can storyboard the flow of the application, and its related information, on a whiteboard or a collection of index cards, then you probably have a realistic chance of implementing these same collaborations in software.

Lesson 6. Err on the side of too much information. All components eventually have to communicate something other than a successful result back to a consumer. When considering an approach to error

PROTOTYPING HELPS TO EASE THE ADOPTION OF A COMPONENT-BASED APPROACH.

handling for component-based systems, we've found it has been important to consider more than just the needs of a single application.

There is a delicate balance between errors that relate to the component's business logic, such as the nonexistence of a customer record, and errors that relate to the infrastructure hosting the component. As a result we've had a running debate about the merits of separating these two types of error messages and more contentiously the use of the native exception handling model for the environment and infrastructure errors, while using a generic error message passing scheme for the business logic errors.

We've found that through the establishment of a two-tier error message structure, using a return code to indicate the cause of the error and a reason code to refine the generic message of the return code, we were able to assign every condition—be it an error, warning, or confirmation—to a set of basic return codes. These return codes gave us the ability to broadly define how any component reacts to a given scenario. The component designer augments the error return code with additional information to make it more appropriate and recoverable for the components consumers. This further information takes the form of a reason code, context string, and message text, all as optional parameters of the error.

We have debated the merits of encouraging a consumer to test and make decisions based on only the return code. Doing so guarantees any component that can return the list of general error return codes we have defined will be compatible. Using the information found in the reason code further helps refine a programmatic response to the error. We've used the additional information—the reason code, context string and message text—simply as supplemental information placed into an error message dialogue or program log to aid a developer in debugging the problem.

The advantage we gain through this separation of environment from business logic is a simplified distinction between the types of errors. We're able to trap environmental exceptions using the native exception handling model—typically a try/catch/finally approach—while writing code within the protected block to handle any conditions that may be raised by the component. While this multiplicity of error handling may seem redundant, it has assisted us in creating a simpler approach to the development of multi-application components, though not without leaving a lot of nagging questions about the approach. While our components can operate without regard to the implementation technology they are operating in, be it COM+, J2EE, or our 4GL CASE tool, they have this portability at a cost, the application errors are not being handled in the native format, which causes a problem for many developers. This remains an open question.

Lesson 7. Testing strategies change. When developing a component-based solution, traditional testing strategies are altered to support the changes that CBD makes to the project life cycle. Testing of a component-based solution is best viewed as two distinct activities: the testing of the components, and the testing of the assembled solution.

Because a component is an independently implemented module of software, this suggests it can be tested as a standalone unit. This assumption led to a problem on our earlier projects. We spent as much time building a user interface to test the component—often called a test-harness—as we did developing the component. Unlike the semiconductor industry, we lacked a test bench that allowed a test plan to be executed against a nonvisual component. We struggled to find a way to test these server-side components in the absence of a user interface. We wound up performing minimal testing on the component, and instead concentrated more effort on testing the application. Each component was tested based on the roles it was performing within the application. There was significant overhead in this approach, especially if a component implementation was changed during development. Any component used in multiple roles within the application required that each usage had to be tested to ensure that a change in a components implementation was not going to cause dependency problems within the applications. This is not what we would have hoped for.

Today we use both discrete component-level testing, based on test plans prepared by the component architects, and regressive application testing, based on scenarios prepared by the application architects. The rise in recent years of more robust component implementation models has brought with it a richer, more integrated circuit-like approach to testing. Now software-testing products deliver a "virtual test bench" where components and interfaces may be exercised in the absence of a user interface. These types of tools will help us with our future applications and component delivery projects, as testing is the least appealing task for most developers and yet is one of the most important activities for a successful project.

Conclusion

I hope this article does not give the impression that we have had bad experiences with CBD. In fact, the exact opposite is true. Castek has achieved milestones through the use of CBD that would otherwise have been impossible using traditional software development techniques. Our track record speaks for itself: on-time and within-budget delivery, with scope that exceeded the original customer expectations. Our success is due to the gains in quality, productivity, and functionality we have experienced with components. With each project, and through the lessons outlined here, we have improved our behaviors and process to get better with each pass, and we know we can continue that cycle. C

References

1. Burg et al. Exploring a Comprehensive CBD method: Use of CBD/e in practice; www.sei.cmu.edu/cbs/cbse2000/papers/03/03.html
2. Darnauer, J. Component definitions; www.cse.ucsc.edu/research/surf/users-manual/node94.html
3. McInnis, K. An introduction to CBD/e; www.cbd-hq.com/articles/1999/991115km_overviewcbde.asp
4. Microsoft Corporation. Definition of the term component; msdn.microsoft.com/repository/OIM/resdkdefinitionoftheterm-component.asp
5. Sparling, M. Is there a component market?; www.cbd-hq.com/articles/2000/000606ms_cmarket.asp
6. Sutherland, J. Healthcare technologies futures; www.jeffsutherland.org/papers/MSHUG_hctf.htm
7. Wagner, M. Google bets farm on Linux; www.techweb.com

Michael Sparling (mikesparling@acm.org) is Chief Technology Officer at Castek in Toronto, Canada.

The University of Texas at Austin

Available Online!

Software Project Management Certificate Program

Now taking registrations for the October 2000 class!

Software Quality Institute
www.sqi.utexas.edu/swpm
a unit of the
Center for Lifelong Engineering Education