

ADVANCED INFORMATION



WHITE PAPER

DigiSlice Corporation

Software Components and Services

or, Objects Considered Harmful

Introduction

Progress in computer software engineering has occasionally required abandoning older accepted practices. A famous example of this is documented in Dijkstra's seminal letter, "GOTO Considered Harmful", which ushered in the era of structured programming by discouraging the use of the then-popular GOTO statement. Another example of this comes from *The Elements of Programming Style* by Kernighan and Plauger, which promoted good programming technique by showing examples of bad programming; all of which were taken from popular programming textbooks of the day.

Software is again at a point where it needs to throw away some accepted practices in order to progress. The desperate need for change should not be a surprise to anyone. Today, over half of software projects fail (either are cancelled or do not meet their requirements). Some reports put the failure rate as high as 70%. And software projects that do not fail produce results that contain levels of reliability (bugs), security (susceptibility to viruses), and difficulty of use that would be unacceptable in other disciplines. Large projects are especially difficult and risky.

Computer software development looks particularly unhappy if you compare it to computer hardware. Back in 1965, Gordon Moore observed that the number of transistors in integrated circuits was doubling every year. This trend has pretty much held true ever since, while costs have gone down. These exponential advances in digital hardware have made a wide range of technical marvels possible, from today's desktop personal computer that runs faster and has more storage capacity than yesterday's most advanced supercomputer at a tiny fraction of the space and cost, to digital cellular phones, notebook computers, PDAs, digital media including CDs, DVDs and MP3s, and on and on.

One can only try to imagine what wonderful software applications could be produced if we could double the number of lines of code in our software every year or two, without increasing costs. It is a telling symptom of the grave problems facing software development that programmers often shudder at the mere suggestion of doubling the lines in their applications that often. Is the construction of large software systems so fundamentally different from assembling hundreds of millions of transistors in an IC that software is forever doomed? I want to think not.

Goals

Our goal is simply stated: we want a way to *double* the size and complexity of the largest software systems that can be built *every one to two years*, without increasing costs or time. Attaining this goal will require unprecedented increases in software productivity. It will also require throwing away some of today's widely held beliefs.

In order to achieve this goal we need to study other similar fields of endeavor to see what has worked and what hasn't. We need to figure out what strategies will apply to software

development, and how to adapt strategies from other fields to make them applicable to software development.

Our work, combining scalable *components* (including Web services) with new technology for *integration and customization*, has shown that we can achieve factor-of-ten decreases in time and money on a range of projects [reference case studies]. Our experiences have convinced us that we can achieve the above-stated goal, however, it will take a massive cooperative effort on a scale never before seen in software engineering (but commonplace in other engineering disciplines).

Software Parts

Even though the title of this paper refers to components and services, it is in fact about changing the way that software is designed and built. Unfortunately, the word “component” is so overused in software engineering today that it has become all but meaningless. And while Web services are a current hot topic, services are nothing more than separately deployed components. So let us forget everything we know (or think we know) about components and services and start from scratch.

The obvious first step is that we must change software development from a *craftsman technology*, where software is handmade a line at a time, to an industry that uses *standard, reusable, scalable* parts. This is the software equivalent of the industrial revolution. Before the industrial revolution, machines were built one-at-a-time, by hand. There were no standard parts, and no part distributors or hardware stores that sold standard parts. If you needed a bolt to hold something together, you had to construct the bolt yourself. Building complex machines was an art (much like programming is today).

Building software from parts is nothing new: components, services, objects, and even subroutines can be described as software parts. What is different here are the qualifiers *standard, reusable, and scalable*. Our parts must follow standards (just like hardware parts) so we know how to integrate them together. And they must be reusable, so they can be used in a number of different applications (this is one way we get our required productivity gains). It is not enough, however, that our parts are reusable. After all, individual transistors are reusable, and computer instructions are reusable. In order to achieve the productivity and complexity gains we desire, it must be possible to marshal these simple reusable parts into *increasingly complex* (scalable) parts.

Computer hardware started with individual transistors, but it progressed to small-scale integration (SSI), which used transistors to build integrated logic gates. After this point, designers did not need to be concerned about individual transistors; they designed at the gate level. This level of abstraction made it easier to create more complex structures, such as shift registers and counters, which were called medium-scale integration (MSI). Again, the abstraction went up a level, and MSI parts were combined to build arithmetic and logical units, memory, and other more complex parts, called large-scale integration (LSI). The next level combined LSI into complete computer processors, memory subsystems, I/O controllers, and so on. Running out of acronyms, this level and beyond was simply called VLSI (very large scale integration). What we want is *VLSI for software*.

Imagine what computer hardware would be like if computers were still built out of individual transistors. Modern computers and other digital devices would simply not be

ADVANCED INFORMATION

possible — using discrete transistors, today's personal computer would be around a mile in diameter. It would be slow, unwieldy, difficult to maintain, and consume huge amounts of resources. Sound familiar?

One would think that software would be a natural candidate to be built out of parts, since there is no physical cost associated with software. In hardware, there is an engineering cost to design a good, reusable integrated circuit (IC), but there is also a cost associated with manufacturing physical instances of that IC, a cost associated with distributing it (building and keeping a physical inventory, shipping, etc.), and a cost associated with throwing away old ICs when new versions are developed. The physical costs associated with software are close to zero, so one would think that reusing an existing software part would be almost infinitely cheaper than building a new part.

Unfortunately, the physical cost of a part is only a fraction of the total cost of using it. There are also costs associated with finding an appropriate part, testing it, and integrating it into your system (whether software or hardware). In the hardware world, there is an existing industry infrastructure to support reusable parts, including catalogs of ICs and standards to help with testing and integration. There is no such infrastructure in the software world.

Integration is particularly messy in the software world — integrating the subparts of a project is the most expensive and riskiest part of a large software project. Most software tools are craftsman tools — designed to help with writing software line by line — and are little help with integration on any significant scale.

Software has resisted being built from parts mainly because it is so easy to whip off small programs from scratch. Even if the cost of a reusable part is zero, the cost to find the proper part and integrate it into your software system must be less than the cost to build the part from scratch before there is any incentive to use parts.

Hardware engineers had no choice but to use parts — it would be beyond the means and ability of most hardware engineers to whip up a new integrated circuit (even a small one), but any software engineer has the means to throw together a small program. Similarly, a hardware engineer cannot take an existing IC and modify it to meet new requirements, while any software engineer can (far too) easily make changes to existing source programs.

The lack of physical cost associated with software has in some ways actually *hindered* the development of reusable software parts, since the need to purchase an IC is what pays for its development. There needs to be an economic reward for the development of software parts to pay for the necessary infrastructure.

There is no doubt that if software could be built from scalable parts then we could easily double software productivity every one or two years. The important issue is that it must be the complexity of the parts that increase, not the complexity of the software that integrates those parts into a final project. After all, even though the complexity and power of personal computers has increased millions of times, the complexity of a computer motherboard has stayed about the same. It is the complexity of the parts (ICs) that has increased. The complexity of the top-level design process (to integrate the components together) has stayed about the same over time.

ADVANCED INFORMATION

There are other benefits from the use of software parts. For example, a previously tested and widely used software part is invariably more reliable than a part built from scratch. Security concerns make software reuse even more important, since the cost of making a component secure can be amortized over many uses.

In order to achieve the goal of software parts, we need an infrastructure for finding and testing these parts. We also need new software methodologies that support the construction of software out of parts, and new tools to help with integrating these software parts together, both into applications and into new, more complex, parts.

Asynchronous Control

An equally important requirement to achieve our goal of dramatic increases in software productivity and complexity is to give up the concept of single locus of control. Software must change from being *synchronous*, to being *asynchronous*. Synchronous software is left over from the earliest days of computing, when computers did only one thing at a time.

Synchronous software is too restrictive for building large, complex systems. It creates control dependencies between separate parts. In order for software parts to be reusable, they must be independent — dependencies between parts, especially control dependencies, cause massive problems during integration. It does no good to have powerful software parts if they can't work together on a task.

Synchronous software is like a single-person company. That person can work efficiently, but they only can do one thing at a time. There is a limit as to how much work can be done by one person working alone. In order to get more work done, you have to hire additional employees. Having people work together introduces overhead (meetings, bureaucracy, etc.) but overall more work can be accomplished. Whether these employees are assembly line workers or middle-level managers, they work independently and only synchronize when necessary. It is this kind of cooperative asynchronous behavior that allows massive increases in productivity.

If we want similar increases in software productivity and complexity, we must do the same thing. Large software systems are by necessity asynchronous. They are always multi-threaded and typically distributed. It is necessary that software support this kind of system directly.

Component Architectures

Most people assume that the motivation behind existing component architectures (e.g., COM, DCOM, JavaBeans, EJB, or CORBA) is to enable reusable software parts, but the primary motivation has actually been to deal with asynchronous control.

While traditional software is mostly synchronous, the rest of the world is not:

- Humans are not synchronous, which is why they prefer user-driven graphical user interfaces (GUIs) to synchronous, command line interfaces (like DOS). Even systems that still feature a command line interface (like UNIX) now provide multiple windows with multiple command prompts, so the user can do more than one thing at a time.

ADVANCED INFORMATION

- Networks of computers are not synchronous — many different things are going on at the same time. Most enterprise applications involve multiple computers that coordinate over a network to solve a single problem.

It is no coincidence that just as there are two main ways in which computers must behave in an asynchronous manner — to deal with humans and to deal with multiple computers — there are two main kinds of component architectures:

1. Component architectures that were developed for dealing with user interfaces, including Sun's JavaBeans, and Microsoft's Component Object Model (COM and COM+, also known by a number of other names including ActiveX controls).
2. Component architectures for dealing with distributed computing, including Sun's Enterprise JavaBeans (EJB), the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA), and Microsoft's Distributed Component Object Model (DCOM), now part of Microsoft .NET.

These two kinds of component architectures seem to be very different, but what they have in common is that they both are designed to deal with an asynchronous world.

Both kinds of component architectures provide system services that help deal with asynchronous program behavior. For example, the JavaBeans component architecture provides a framework for dealing with asynchronous user events, such as mouse clicks, keyboard presses, and screen repaints. The Enterprise JavaBeans component architecture provides a framework for distributed computing that includes facilities for communicating with remote asynchronous components over a network (remote interfaces), transactions, asynchronous events, and message queues (especially with message-driven beans, in EJB 2.0).

While existing component architectures do an acceptable job of dealing with asynchronous control, what they don't do very well at all is support reusable — let alone scalable — software parts.

Software Components

The word “component” has two meanings. This is just like the word “object”, which has both a common usage and a specific usage in object-oriented programming parlance. In common usage, a component can be any piece of an application. For example, a database or a web server might be a “component” of an enterprise application. Before we can talk about components, we need a specific definition.

Unfortunately, there is no industry consensus on just what a software component is, or what makes a good component. The term “component” has become so overused as to become almost meaningless, other than as a marketing buzzword. There is confusion about the differences between components and other software concepts like objects and services. There are no agreed upon methodologies for building components. There are few languages facilities or tools to help build good component-based systems. Component integration remains a nightmare.

Many of these problems are due to the fact that components are a new concept, but we also believe that the virtual stranglehold that object programming has on the study and

ADVANCED INFORMATION

practice of software engineering (especially in the US) has severely hindered the development of good component-based practices.

I will use the terms **component** and **component-based design (CBD)** specifically to describe software technologies that directly support asynchronous control and the creation of standard, reusable, and scalable software parts. The goal, of course, is to achieve massive increases in software productivity and complexity — a doubling every year or two. Something can only be called a component if it supports these technologies and goals.

Note that under these definitions, services, especially Web services, are components if they are standard, reusable, and scalable. In practice, Web services — since they follow Web standards and are used by multiple third parties — tend to be better components than many components from traditional component architectures like EJB or DCOM.

Components and Objects

Our notion of components and CBD are quite different from some definitions for these terms. A common definition of component is “an object supported by a published component architecture, such as JavaBeans, Enterprise JavaBeans, COM, DCOM, or CORBA”. Such definitions do not distinguish components from objects, or CBD from object programming (OP). We have found that many software technologies that claim to support components are actually about objects, and many systems that claim to be component architectures are really object-oriented frameworks (that just provide services to objects). In general, any definition of component (or CBD) where you can remove the word “component” and replace it with “object” is not about components at all, it is about objects.

The OP industry is so firmly entrenched in the software world that it is easy to recast components as just a new form of object. This is one of the major contributors to the current confusion about what a component is. As the old saying goes, *when all you have is a hammer, everything tends to look like a nail.*

Of course, a component is not the same thing as an object. If a component were the same as an object, why would we even need new terms like component or CBD? Is “component” just a buzzword, to make old ideas seem new? In an industry used to excessive levels of hype, this may be what most people expect. From an engineering perspective, however, if we are going to use a new term like “component”, then the supporting practices need to be new.

In order to understand the differences between components and objects, we must first understand what objects are. While OP is fairly complex, the concepts behind OP are quite simple. They are mainly concepts taken from primitive types — abstract data-types, data hiding, encapsulation, and type polymorphism — extended to user-defined data structures.

OP was a *huge* success: it revolutionized the way software was designed and developed; spawned hundreds of new computer languages including Smalltalk, C++, and Java, and dozens of new journals and conferences; and generally revitalized the science of computer science (which had fallen into something of a slump after everyone got bored with studying fascinating subjects like the theoretical speed of all the different ways to

ADVANCED INFORMATION

sort a list of numbers). OP promised — and delivered — more reliable software, with fewer bugs. It also promised huge gains in programmer productivity, largely based on the ability to construct *reusable* objects.

However, OP did not live up to its hype. Controlled experiments showed that productivity gains were around 1.3X (30%) — nothing to sneeze at, but not even close to the 2X to 10X gains that had been promised — and even those modest gains tended to vanish on large applications. The reusability of objects was pretty much a complete bust. [\[add references\]](#)

We sometimes hear software teams say things like “we tried CBD, but it didn’t work.” But when you dig a little deeper, you often find that what they really mean is “we tried software reuse, but it didn’t work.” But what were they trying to reuse? Since all they knew how to build were objects, they were trying to reuse objects. Since hardly anyone has been able to successfully reuse objects in any significant way, it no surprise that they failed.

In their book *Business Component Factory*, Peter Herzum and Oliver Sims sum it up: “Object-oriented approaches seemed so powerful, but after a while it became clear that something was missing. The expected gain in productivity, high levels of reuse, and ability to master complex development did not materialize.”

The problem is that object programming is a craftsman technology. It is like giving a craftsman improved tools, so they can build something slightly faster and better, but still by hand. Object programming still builds software one line at a time. It does not support software parts, and it does not support the construction of large, asynchronous, complex systems.

As a craftsman technology, OP is mainly about programming (down in the dirt, cowboy-style coding). CBD covers every phase of the software lifecycle: design, development, deployment, maintenance, customization, and so on. In particular, CBD is not just a development approach, but also a **deployment** approach, which leads to new ways to create, deploy, market, and buy software solutions.

Levels of Reuse

Software reuse is nothing new — even before programming languages supported reuse, programmers could cut and paste code from one program to another.

The first explicit support for reuse was the invention of subroutine calls with formal arguments. This not only allows procedures to be reused, it encouraged the development of **standard libraries**. The user (i.e., programmer) uses standard subroutine libraries supplied by the language or operating system, rather than rewriting common functionality from scratch. A good example of this is standard random number generators. Programmers wrote (and rewrote) many not-so random number generators before standard ones were developed. Standard libraries are also used to supply functionality that could not be written by the user, such as I/O routines, and to isolate the programmer from differences between operating systems, encouraging code reuse.

The next level of reuse was the development of **frameworks**. Unlike a standard library, which supplies reusable subroutines, a framework has the user supply subroutines to be

ADVANCED INFORMATION

called by the framework at the appropriate time. Frameworks became popular about the same time as OP, and OP features such as inheritance are useful in frameworks, so many frameworks are object oriented (OO), but they do not have to be. The standard C library `sort` (or `qsort`) subroutine, where the user supplies a callback subroutine that is called repeatedly when `sort` needs to compare two objects, can be considered to be an early example of a (very simple) non-object-oriented framework.

In both standard libraries and frameworks, reuse is largely limited to functionality provided by the system supplier. There is limited support for reusing functionality written by the user. The primary difference between an object-oriented framework and a component framework is that, in an OO framework, the framework itself is reused, but not typically the objects called by the framework (written by the user). A component framework encourages the components written by the user to be reused.

So, what are examples of good component frameworks? Unfortunately, GUI component frameworks (like Visual Basic and JavaBeans) are just about the only success stories for component frameworks. Application server (middleware) frameworks, like CORBA, DCOM and EJB, are more OO frameworks than component frameworks. The “components” (EJBs, DCOM objects) are more objects than they are components, and are consequently difficult to reuse. Indeed, Web services are getting so much attention because they provide more of the reusability benefits of a component framework than existing application server frameworks.

Some authors [\[add references\]](#) have pointed out that operating systems themselves are early examples of component frameworks, where the components are applications. This is especially true of the UNIX operating system, where pipes are used to connect programs together using streams.

For my first encounter with the UNIX operating system I was asked to add some features to a program that kept track of papers submitted to a large software conference while they were being reviewed. The existing application was written in Fortran (this was a *long* time ago). The source printout was an inch thick and had taken several months to write. Since I had just started learning UNIX a few days beforehand, I decided to try to rewrite the entire application the UNIX way. By combining `awk`, `sed`, `sort`, `tbl`, and `nroff`, using a half-page long shell script, the result had more functionality than the existing application and even ran faster. Including the time to learn all those utilities, it took a week to write. It was a good introduction to the power of software reuse.

Methodology for Components

How does one go about building components (scalable, reusable software parts)? In most texts on the subject of components, this is where things get muddy. Advice on how to build components is scarce and sometimes contradictory. For example, the best advice found in most articles and books on CBD is about the need for good, well-designed interfaces. These authors appear to ignore the fact that OP already promotes good, well-designed interfaces, but never delivered significant software reuse* .

* This kind of behavior is called a *cargo cult*, after a religious cult that sprung up on a South Pacific island during World War 2. During the war, soldiers arrived and built landing strips and soon airplanes appeared

ADVANCED INFORMATION

This is not to say that good interfaces are not important. Good interfaces are definitely *necessary* for software reuse, but they are not *sufficient*. And what constitutes a good interface for an object may be (and, as we have found, often is) a bad interface for a component. To see why this is so, let us look at the role that good interfaces play in OP, and how that differs from the role that good interfaces play in CBD.

In OP, good interfaces are developed through a methodology called **top-down decomposition** (also called *divide and conquer*). In top-down decomposition, you begin with the requirements for the application and then you start carving it up into smaller pieces, decomposing the problem to be solved into pieces that are (hopefully) easier to solve. Eventually, you have divided up the problem into pieces that can be easily solved — these pieces become your objects. The interfaces between objects are defined by the relationships between the objects, called the contracts between objects. A **contract** is an agreement that says that if the caller of an object ensures that a set of pre-conditions are met, then the called object will ensure that a set of post-conditions will exist. These contracts, in aggregate, define the contract for the entire application, which (if your methodology is *sound*) fulfills the requirements for the application.

For software reuse, however, you want a methodology that is based on assembling applications out of independent, reusable components — a bottom-up approach. In OP, you want the objects to be as simple as possible. In CBD, you want the relationships between components to be as simple as possible, so that the components are reusable. In CBD it is less important that the components are simple, since, if they are indeed reusable, you only have to write them once.

Good CBD design methodologies are more similar to hardware design methodologies than they are to existing OP-based software design methodologies.

Of course, OP design techniques remain fully applicable *inside* of a component. A good component will often contain many objects, and top-down decomposition is a very good way to implement individual components.

Independence

In good CBD design, the components should be **independent**: the goal is to be able to pull out a single arbitrary component from one application and use it in another application. If you pull out a single component, and find that you also need to pull out other components (or other objects and data structures) in order to use that component, then it is not a particularly well-designed component. Unfortunately, it is not unusual to see applications that claim to be component based, but where it is impossible to pull out any single component without dragging along the rest of the application.

and landed with huge bounties of cargo. Once the soldiers left, the previously isolated natives developed rituals to attract planes, including building elaborate ritual landing strips lit by torches. No planes came, but the rituals just got more and more complex. It didn't matter that the rituals didn't give the desired result — the members of the cult kept on trying even harder. In the same way, proponents of OP technology appear to act like cult members when they keep trying harder and harder to deliver reusable objects, in the face of repeated failure.

ADVANCED INFORMATION

How do you make a component as independent as possible? One common problem is that if an object is used as an argument in a method call on a component, then this introduces an undesirable dependency between that object and the component. To solve this problem, we stipulate that the methods (function calls) on a *good* component should take only arguments and return values that are **primitive** or **standard** types. Primitive types are the built-in language types like `integer`, `float`, `character`, and `string`. By **standard types**, we mean types (or objects) that are defined in a standard library for the language and are always available, such as Java's `Point` — as opposed to types that are defined by the application programmer or by an optional library.

An object (other than a standard type) should almost never be passed as an argument to or returned from a component. If you find such a dependency, it probably means that the object should itself be converted into a component. Does this mean that it is acceptable to pass a component as an argument to a method call (or return a component as the result of the method)? Passing a component to another component creates a dependency between these components, and our goal is to make our components as independent as possible. If a method on a component takes a component argument (or returns a component), then you cannot take the first component out of the application and reuse it elsewhere without taking the other component with it. Such dependencies should be confined to the glue code between components. In the section on glue code, we shall see how to do this.

Another issue is **control dependencies** between components. Control dependencies are often found in the pre-conditions in the contracts between objects. A pre-condition will often require that one object execute before another object can execute. In order to be completely independent, there should be no implicit control dependencies between components. Any control dependencies should be made explicit, so that a signal is sent to a component when its preconditions are met and it is ok to execute.

Functional Completeness

The goal of OP is to make the objects as simple as possible. This goal is often stated as *a good object does only one thing*. In CBD, however, a component should be as **functionally complete** as possible so that it can perform the specified function without depending on other components or objects.

For example, a component to perform a credit card transaction should provide all the functionality required to perform that transaction. The information needed to perform the transaction — the credit card number, transaction amount, etc. — should be passed in as primitive or well-known types (often strings). The component should test that the credit card number is valid, connect to the bank's server that will perform the transaction, and perform the transaction. In good OP design, separate objects perform these different functions. In good CBD design, one component can perform them all.

Note that a component internally will normally consist of several components and/or objects. To the user (programmer), however, the component appears as a single piece of functionality. A good component is mostly an issue of good packaging.

Grain Size

Grain size refers to the chunkiness of an application built out of components — an application containing lots of small components is *fine grained*, while an application containing a few large components is *large grained*. The goal of functional completeness means that components often have a larger grain size than do objects; although it is certainly possible to have a simple (but functionally complete) component that is as fine grained as any object.

Note that early component architectures had a huge amount of overhead, largely due to the need to *marshal* data passed between components*. This overhead meant that the grain size of components had to be very large (on the order of separate applications), or else the resulting component-based applications would run unacceptably slow. For example, originally COM was mainly used to do things like place an Excel spreadsheet or an Adobe Illustrator drawing into a Word document — very large grained components.

With new languages like Java this overhead is significantly reduced, to the point where the JavaBeans component architecture has virtually no overhead and components can be as small as desired without any performance concerns. Distributed component architectures (including EJB) still have the overhead of sending data between computers or between separate processes on a single computer, so there are still some performance concerns that affect grain size; for now these concerns are unavoidable.

Another issue that affects grain size is reuse. It is relatively pointless to make something into a component unless you plan to reuse it. This is in contrast to OP, where you want to make everything into an object. An example of this is a car engine. The engine itself is like a component and can be reused, but the engine itself is made up of lots of pieces. Some of these pieces are also reusable and so are like components, such as oil filters and spark plugs, but other pieces do not need to be reusable. Those non-reusable pieces are like objects.

Glue Code

One common problem that gets in the way of software reuse is not keeping a clear separation between the reusable components and the glue code used to integrate components together. Consider a component for charging credit cards in an e-commerce application, designed and implemented using standard object practices.

* Marshalling is used to transfer data between computers or languages that have different representations for the same type. For example, in C++, the `integer` type may be defined to be 16 bits, 32 bits, or even 64 bits long on different computers (or the same computer with different operating systems), and different languages often have very different representations for the string type. Passing even simple data between components written in these languages requires considerable overhead.

ADVANCED INFORMATION

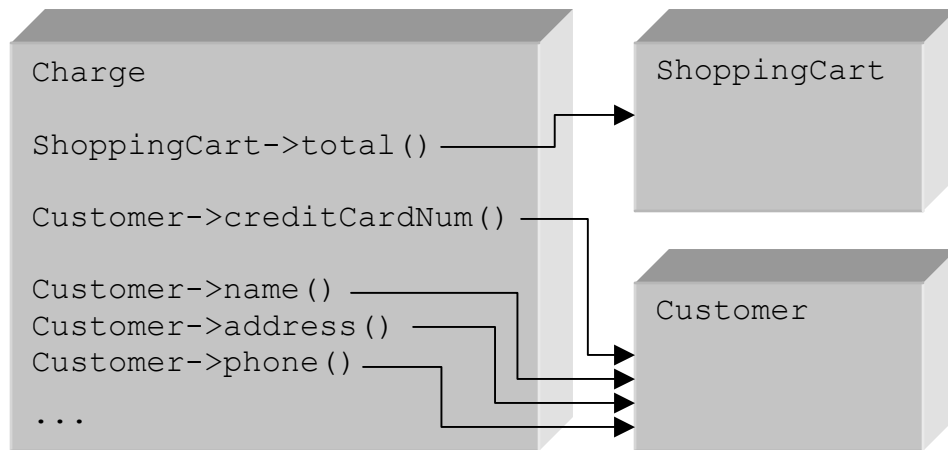


Figure 1 — Non-reusable component.

As shown in Figure 1, the `Charge` component first calls a method on the `ShoppingCart` component to find out the dollar amount to be charged. It then determines the customer's credit card number. Finally, to process the charge, it must obtain information about the customer to verify the charge to the bank.

Now, the `Charge` component seems generally useful, and we would like to be able to reuse it in other applications. Unfortunately, we have to make quite a few changes to the `Charge` component to make it reusable. Here are some of the problems:

- The `Charge` component calls the `ShoppingCart` component explicitly. What if we want to use the `Charge` component in an application that doesn't use a shopping cart? We would have to modify the `Charge` component to (explicitly) call some other component to get the amount to charge.
- Likewise, the `Charge` component calls the `Customer` component repeatedly, to get several pieces of information. But what if we want to use the `Charge` component in a different application, where customer information is stored in a different format? For example, let's say that the `Customer` component doesn't have a `name` method, but instead has two methods, called `firstName` and `lastName`. Then we would have to add some glue code to the `Charge` component to change name formats.

Even this simple `Charge` component requires many changes to reuse it in a different application. To be truly reusable, all code that is dependent on other components must be removed from each component, and placed in separate integration (glue) code. This isolates the code that must be changed for each application from the reusable components, as shown in Figure 2.

ADVANCED INFORMATION

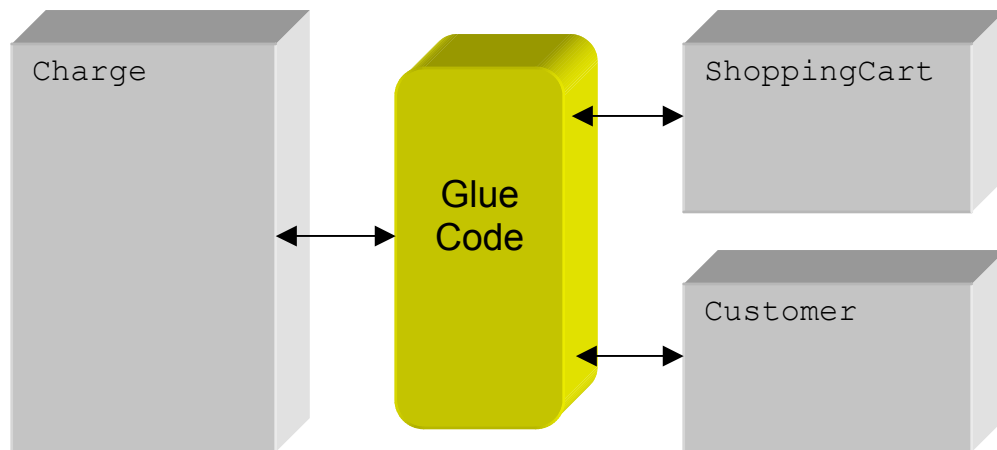


Figure 2 — Reusable component with glue code.

In this case, the glue code would look something like the following:

```
Charge->setTotal (ShoppingCart->total () )
Charge->setCreditCardNum (Customer->creditCardNum () )
Charge->setName (Customer->name () )
Charge->setAddress (Customer->address () )
Charge->setPhone (Customer->phone () )
```

Of course it is not this simple. In a large application, having all the glue code in one place would make the glue code itself large, complicated, and prone to error. We need mechanisms for organizing glue code, facilities for showing dependencies between components, and ways to find the glue code for each component.

Caller Interfaces

One goal of OP is to build objects with well-defined interfaces. Method calls — with formal arguments and returned values — provide an interface *into* an object. Providing a good interface to a method is a good thing, but to enable reuse it is only half the job. For good components, we also need an interface for the *caller* of a method.

In top-down decomposition, we are only concerned about the interface into an object, which is necessary so that we can change the internal implementation of an object without affecting anything that calls this object. But in CBD, it is vitally important to simplify the dependencies between components as much as possible, so it is equally important to be concerned with the interface *out from* a component, not just the interface *into* the component. OP allows calls to other objects to be buried deep inside of an object, with no defined interface at all.

ADVANCED INFORMATION

To highlight the necessity for good interfaces (both into and out from a component), consider a distributed computing application where we have three components that communicate with each other over a network. One common way to implement this application is through the use of input and output statements, which, instead of reading and writing information to disk, communicate data through some form of inter-process communication (IPC) mechanism, typically **sockets**. This is illustrated in Figure 3.

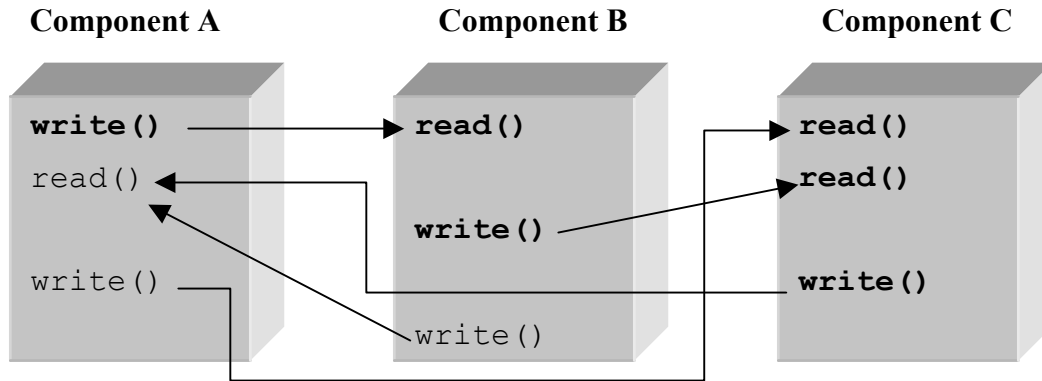


Figure 3 — Distributed components using sockets.

Using `read` and `write` statements to pass control between components is the distributed equivalent of using `GOTO` statements in a sequential program, and is just as bad. Control and data can leave a component at any point and enter another component at any point, not just through the component's interface. This causes a number of problems:

- Changes in one component can have unintended effects in other components because there is no clear interface between the two.
- It is very difficult to remove any one component from this application and reuse it in another application, without dragging along the other two components, because of all the direct dependencies between the two.

A better way to implement this application is through the use of **remote procedure call (RPC)**. In the same way that regular procedure calls solve some of the problems of `GOTO` statements, RPC solves some of the problems of IPC using sockets. RPC works like a regular procedure call, except it calls a procedure on a (possibly) remote computer (e.g., over a network). This provides a good, well-defined interface *into* the remote procedure, as shown in Figure 4.

ADVANCED INFORMATION

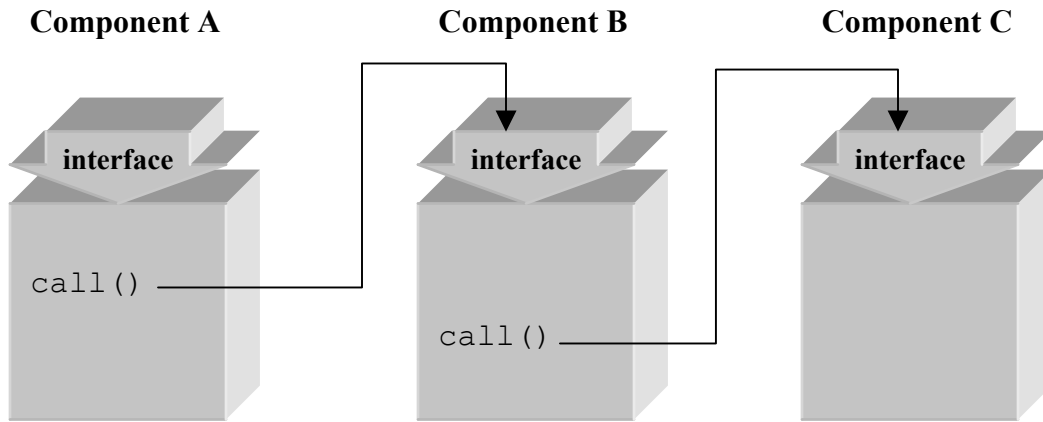


Figure 4 — Distributed components using RPC.

RPC uses the abstraction of a procedure call to enforce a separation between two components. It also prohibits jumps into the middle of a component.

RPC benefits distributed computing, since we can now change the implementation of a component without affecting other components that call it, but there are still problems.

- Control can still pass out of a procedure at any point, without going through an interface. If we pull component A or B out of this application, we will run into hidden dependencies on other components. For example, component B has a call to component C buried in it, so pulling out component B drags component C along with it. Pulling out component A drags the entire application along with it.
- Procedure calls are synchronous. RPC must be asynchronous so that more than one procedure can be executing at the same time, but this means that RPC calls cannot return values like normal procedure calls. It also destroys most of the semantics of a traditional procedure call.
- Procedure calls form a hierarchy. It is no coincidence that this hierarchy mirrors the hierarchy of top-down decomposition. Communicating distributed components are more like co-routines than subroutines. This can cause a host of other problems, in particular during debugging (what does a stack trace mean in a system that uses RPC?)

What we really want is an interface both into and out from the component, so that *any* dependencies from a component go through an interface, as shown in Figure 5.

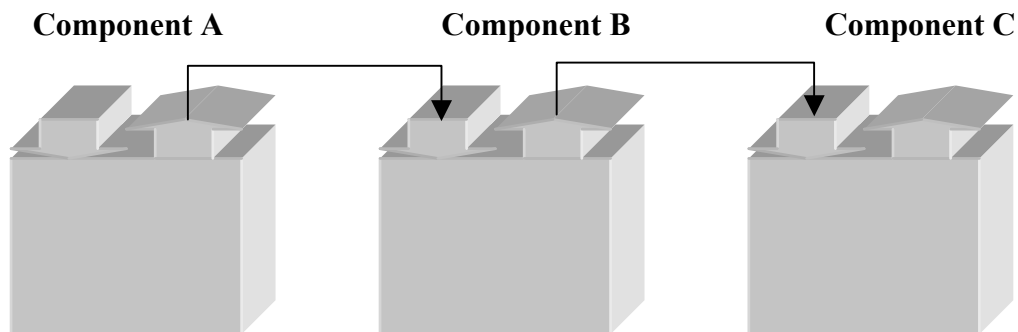


Figure 5 — Distributed components using caller interfaces.

ADVANCED INFORMATION

With good interfaces both into and out from a component, we can readily pull any component from an application and reuse it in another application, as long as the new application meets the requirements of the component's interface. The dependencies between components are confined to the **glue code** between components. The component's interface is separate from its implementation, so we can make changes to the implementation (or even replace the component with an entirely new one) without affecting other components — including both components that *call* this component or are *called by* this component.

This new component interface also changes the relationship between components. Using RPC, the relationship between components forms a hierarchy, like the hierarchical decomposition used in OP. Using caller interfaces, the components are more like cooperating equals (like co-routines). This parallel decomposition is far more component-like.

At this point, you (the reader) might point out that while computer languages provide support for an interface *to* the called object (via procedure calls), they do not provide support for an interface *from* the calling object. But there is a well-known mechanism that supports a caller interface. This mechanism even supports asynchronous execution, and (not surprisingly) it is a part of many component architectures. This mechanism is **events** (sometimes called a messaging interface).

Event Mechanisms

Event mechanisms were originally developed as a way for a procedure to be called in response to a user input event, such as a mouse click or a keyboard press. In this situation a normal procedure call cannot be used because the caller — the mouse or keyboard driver — has no way to know what procedure to call, and in procedure calls, the caller must statically know the identity of the procedure to be called.

Events are backwards from a normal procedure call. In a procedure call the caller specifies whom to call, while with an event the called object specifies who should call it. For example, a routine to move a cursor on a screen can specify that it wants to receive mouse motion events. This backward nature is exactly what is needed to provide a caller interface for components.

Early event mechanisms often used a single, centralized event queue, and used inefficient broadcast mechanisms. But newer event systems, like the one used by the JavaBeans component architecture, use a mechanism whereby a procedure **subscribes** to a type of event. When an event is generated, it iterates through the list of all procedures that are subscribed to that event and sends the event to each one, using a procedure call. Thus sending an event is roughly as efficient as a regular procedure call. This event mechanism also allows an arbitrary number of event queues, one for each kind of event, and so is very amenable to being used for distributed computing.

Event mechanisms were originally developed for component architectures designed for building user interfaces, and are almost universally part of component architectures of this type. Unfortunately, not all component architectures designed for distributed computing have event mechanisms. It is interesting to note that in at least one case, the lack of an event mechanism was enough of a problem that one was added to an existing

ADVANCED INFORMATION

component architecture. Even though the JavaBeans component architecture had an event mechanism from the beginning, Enterprise JavaBeans (which were designed after JavaBeans had been around for a while) did not have an event mechanism until the Java Messaging Service (JMS) was added in version 2 of the EJB specification.*

With the use of events, the interface to a component consists of:

1. The procedures (methods) that can be called — the interface into the component.
2. The events that the component can generate — the interface out from the component.

Note that most component architectures also allow components to have settable **properties**. Properties are implemented using normal procedure calls (these are typically called *set* and *get* methods). All three parts of the interface to a component — methods, events, and properties — are actually implemented using normal procedure calls, but to the component user they appear separate.

When a component needs to communicate with another component, rather than make a procedure call directly to that component, it should generate an event through its interface. This provides the same valuable abstraction that procedure calls provided over GOTO statements. Every connection to other components — both incoming and outgoing — goes through the component's interface. Dependencies are explicitly contained in (and limited to) the glue code between components.

Entity-based Computing

Another difference between OP and CBD is that while most OP is **class based**, CBD tends to be **entity based**. Entity-based computing supports both software reuse and asynchronous computing, and has other benefits as well.

An **entity** is merely an instance of a component, including its persistent state. In entity-based computing, component entities are treated as persistent, statically existing objects. It is as if entities were physical objects, like integrated circuits on a circuit board. With physical circuits, even when the circuit board is turned off, the integrated circuits still exist, as does the wiring between the integrated circuits. In entity-based computing, you treat the entities (the instances of components) and the “wiring” (the connections) between entities as if they always exist, even when the program is not running.

Contrast this with OP, which is class based. Objects — instances of classes — are created, initialized, and connected together (wired) when the program starts, and even more instances are created (and destroyed) as the program executes. Objects are transient, as are the connections between objects. Attempts to make objects persistent, such as in object-oriented database management systems (OODBMS), largely failed because of the mismatch between transient objects and persistent storage.

The difference between class-based and entity-based systems is highlighted by the facilities each uses to deal with common functionality. In class-based computing, if two

* If the discipline of component architectures had received as much study and public debate as the discipline of object programming did when it was new, such omissions probably could have been avoided. Unfortunately, the design of existing component architectures has been, at best, ad hoc. One purpose of this paper is to encourage such study and public debate.

ADVANCED INFORMATION

classes share functionality, good practice says that you should create a common **superclass** containing the common functionality. The two original classes become subclasses of a common superclass, using a mechanism called inheritance. In entity-based computing, common functionality is handled using **delegation**. An entity can *delegate* functionality to another entity. COM was designed to use delegation extensively through the `iQuery` interface.

Similarly, if you have an existing object or component representing a `Customer`, and you want to add new functionality to it, say to keep track of whether or not that customer wants to receive a monthly newsletter. In a class-based system, you would normally derive a new subclass from the original `Customer` class. The subclass implements the new functionality and inherits the remaining functionality from the original `Customer` class. In an entity-based system, you would create a new component, and delegate all the old functionality to the original `Customer` component. If your system doesn't support delegation, you could instead use **aggregation** and build a new component containing both the original `Customer` component and the new functionality.

In CBD, components tend to be persistent, in that they retain their state even when the applications that use them are not running. For example, JavaBeans and EJBs are serializable, so their persistent state (including properties) can be stored on disk, or can be transmitted over a network. This allows a *running* instance of a Java component to be saved and then later restored, or moved to another computer. Serialization saves the state of the component by writing out enough information so the component can be recreated in another time or place. The recreated component is essentially identical to the original component.

In OP, a program owns the object instances; the instances are created when the program starts and cease to exist when the program stops. Component entities (particularly in distributed computing) are independent of any particular program. Many different programs may access the same component entity over time.

Because entities exist statically, entities and objects tend to differ in the way they deal with collections (sets, lists, or arrays) of things. Consider a database query to find all the potential suppliers of a required part. In OP, the query would normally return a collection (array) of results. Many objects are created, one for each supplier, but objects are transient, so it is normal to create lots of them. In CBD, you would instead have a single entity that represents a supplier, which iterates over time through all the values returned by the query. This kind of program is more in keeping with the idea that components are static, like physical parts.

If you are an experienced object-oriented programmer, you might argue that the OP way of returning a collection of objects is better, but in CBD our goals are different. The CBD way of iterating over time promotes reusability. At the very least, the OP way requires an additional object (the collection object) that introduces an additional dependency and additional wiring. In CBD, the functionality for iterating over a collection of things is built into the component itself (for *functional completeness*) rather than being performed by a separate object.

ADVANCED INFORMATION

Unfortunately, since most of the designers of current component architectures come from the OP community, some component architectures embody more of the OP way of dealing with collections, rather than the CBD way. For example, in the EJB standard, you find an instance of an entity bean using a *find method*. These find methods violate good component practice in two ways: first, they take an object (`PrimaryKey`) as an argument, and second, they return an enumeration as a result.

As previously noted, the entity approach is similar to the way that real, physical objects behave (like integrated circuits on a circuit board). Perhaps coincidentally, the entity approach is also closer to how problems like this are solved by humans. For example, to find a supplier for a needed part, a human would usually go through a phone book sequentially, calling suppliers one at a time until they find one that has the required part.

Another difference between an object and an entity is how they deal with type information. In OP objects are transient, but type information is static, so the type of an object is specified by a separate, static object: the object's class (this is true even in those OO languages where classes are not themselves objects). In OP, the type of an object is completely determined by its class. In CBD, entities are persistent, so their type does not need to be entirely static.

As a simple example, consider an object-oriented graphics system that has a `Triangle` class and a separate `Polygon` class. The latter class has a property called `numSides` that specifies the number of sides in the polygon. Since the class of each object specifies the type information, if you ask both an instance of a `Triangle` and a `Polygon` if they represent a triangle, only the `Triangle` instance can say yes. A `Polygon` instance, even where `numSides` is equal to 3, will still say no. Furthermore, if you ask if there is any class that can represent a quadrilateral, no class will say yes, even though a `Polygon` (with `numSides` equal to 4) could easily do it. In CBD, any component that can fulfill the interface requirements of a triangle should be treated as a triangle, to promote reusability. So if you ask an entity if it is a triangle, it should say yes even if it is really a polygon with 3 sides. Thus, the type of a component can be computed by the component (it is not a static attribute of the component).

Persistence

Entity-based computing helps bridge the conceptual gap between transient programs and persistent memory. In traditional computer programs, the programmer has to explicitly save state to persistent memory. Often, the programmer passes this task on to the user. For example, when using most word processing programs, the state of your document is transient. You have to perform a save operation to make your document persistent. Using entity-based programming, a document can be represented by an entity: a persistent instance of a component.

Reading and writing of data files is distinctly non-Object-like. Reading and writing objects to persistent storage is not very file-like. Traditional techniques for increasing efficiency, such as clustering and sequential access, are difficult to utilize in a purely object-oriented system. This is one of the main reasons that large programs written in a pure OP fashion can be notoriously slow to the point of being unusable. This problem occurs because objects are transient, so they must be explicitly saved. In contrast,

ADVANCED INFORMATION

components are inherently persistent — they are not attached to the execution of any particular program.

In traditional development, the end result of the development process is one or more programs to be deployed and executed. In CBD, the end result will also usually contain some entities (e.g., EJBs or JavaBeans in `.jar` files), representing *stateful* instances of components. Again, it is as if these components always existed. This is in contrast to traditional development, where you can start everything over simply by restarting the program (or at worse, rebooting the computer).

Not only do we need facilities for deploying stateful entities, but we also need facilities for *partial deployment*, since we will often need to redeploy an application without disturbing the state of its entities. Also, the entities that are a part of an application will often change over time. To support partial deployment, each component must be independently deployable.

Partial deployment is especially important in mission-critical applications. Some applications need to be running 24 hours a day, 365 days a year. Even the shortest downtime may cost thousands of dollars and adversely affect customers. In this case we need the ability to redeploy an application *while it is running*. Redeploying some components while the rest of the system continues to run performs this feat.

Web Services

Services, like those used in client-server systems, have been around for a long time, but **Web services** have recently been receiving much attention. There is some confusion about just what constitutes a Web service, but in general a Web service is just a service that uses Web technologies. For example, many Web services communicate with their clients using the HTTP protocol, and the data communicated is in XML format.

The primary goal of using Web services is to divide a large enterprise system into components. These (service-providing) components can be used by multiple other systems (they are *reusable!*). And like components, a good Web service must have a well-defined interface (defined using schemas, with the data passed as an XML string), and must deal with an asynchronous world.

At this point you might be asking yourself what is the difference between a Web service and a component. If you are still stuck in the camp that considers a component to be just like an object, you might think that there is a huge difference, but using our definition, there really isn't any. A Web service is just a component that uses Web technologies to communicate with other components. The same principals and methodologies that are used to build good components can be used to build good Web services.

As stated earlier, CBD covers the complete software lifecycle, including deployment. One thing that distinguishes web services is that they are deployed independently. Components can be deployed together, or they can be deployed independently (or any combination of the two). Note that since objects are always deployed together, there is a big difference between an object and a service.

Historically, Web protocols are (mostly) stateless — each use of the service is independent and the server doesn't save any client information between accesses.

ADVANCED INFORMATION

Various techniques (including passing information in URLs using CGI, and **cookies**) have been developed to allow the server to utilize client state information. Even so, a Web service must be able to hold state information for multiple clients simultaneously, so the service can be accessed by multiple clients at the same time without them interfering with each other. These same concerns apply to components.

Integration and Customization

CBD consists of two tasks:

1. creating good components, and
2. integrating (connecting) the components together using glue code, including customizing them to meet specific needs

Traditional (OP-based) tools work fine for building the components (assuming that you use good CBD methodologies), but integration is a different matter.

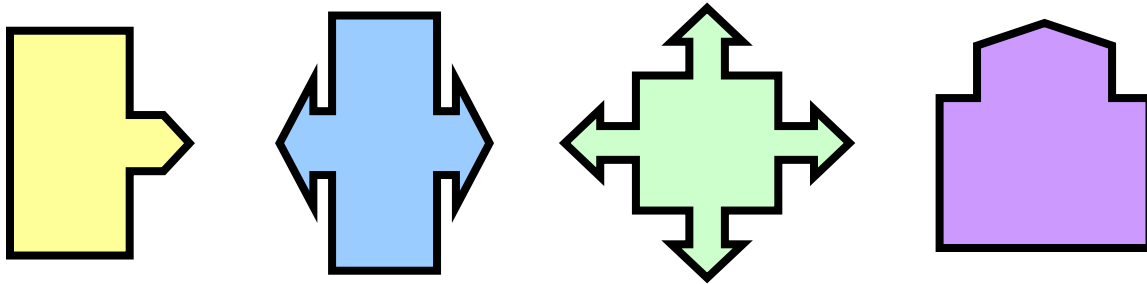


Figure 6 — Components with different interfaces

In Figure 6, each box represents a component, and each component has its own interface. Glue code must be used to connect the component interfaces together. In many cases, the glue code can be more complex than the components themselves, and integration bugs can be extremely difficult to track down and eliminate. Integration is the riskiest part of building enterprise software; this is where the majority of enterprise software projects fail. Even when the project succeeds, it is difficult to estimate the time required, which frequently results in schedule slippages and cost overruns.

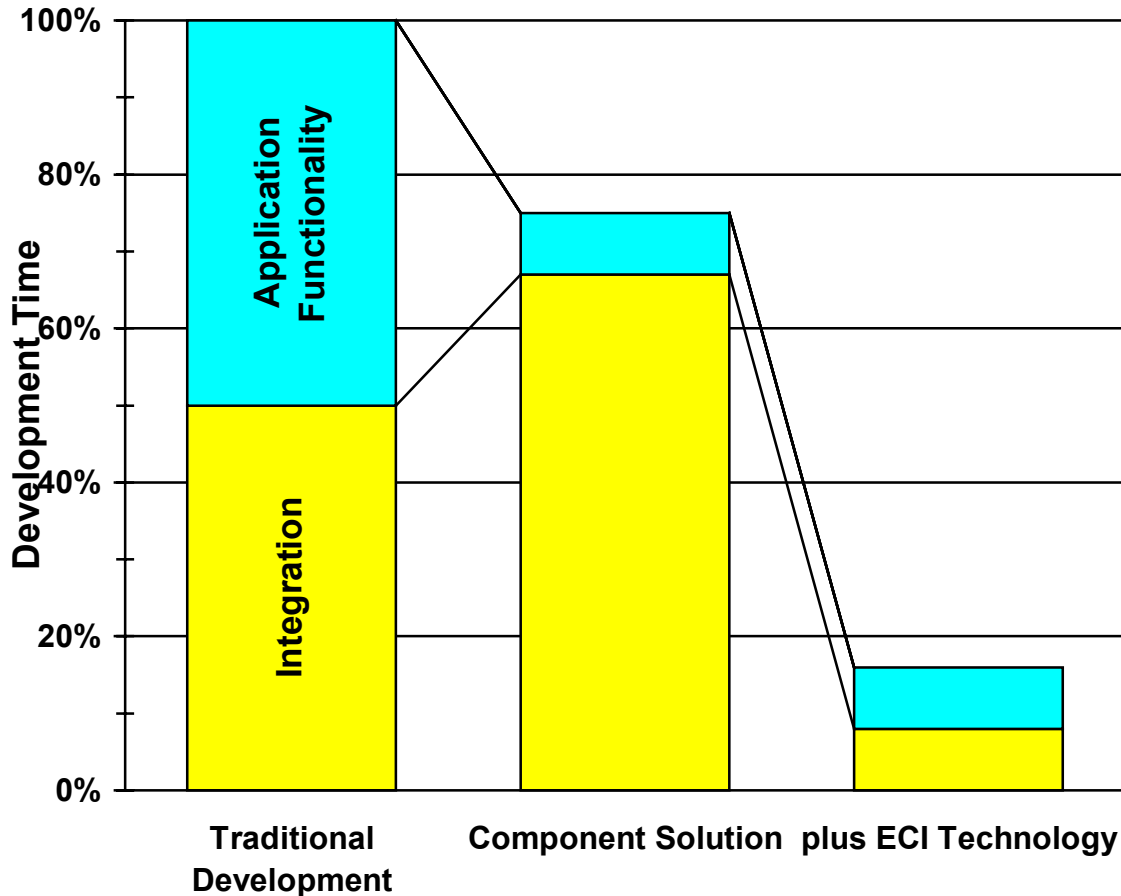


Figure 7 — Development time for different methodologies

Using a component methodology can actually make integration worse. As shown by the first bar in Figure 7, the time it takes to develop a typical enterprise application using traditional (OP) methods is roughly split between programming the application functionality (blue) and integrating the system (yellow).

Moving to the use of component technology (the second bar in Figure 7) can significantly reduce the time spent building application functionality, since you can reuse components you have previously built or purchase components from third parties. But the time for application integration actually increases, significantly reducing the productivity gains.

The integration time increases because we are no longer integrating pieces that were specifically written to work together in this application. Instead, we are often working with disparate components borrowed from other projects or purchased from outside vendors, and the interfaces to the components rarely match. The integration problem is made worse by the fact that, in traditional CBD, the glue code is not reusable.

Existing programming methodologies were designed for building application functionality, and have little to say about integration issues. It is not surprising that integration is such a problem. Likewise, integration today is typically done using programming tools, but programming tools are designed for writing application functionality, not for performing integration tasks.

Note that traditional (OP-based) tools and methodologies are still important — if anything even more important — for implementing the components themselves. If a component is to be reused in hundreds or even thousands of applications it is very important that it be as reliable and robust as possible. The cost of a bug in such a component is very high, so every reasonable step should be taken to eliminate all possible bugs, including careful design and testing. We have also found that it is vitally important that a test harness and suite is shipped with each component, so the user can do their own testing.

Enterprise Component Integration

Recognizing that building enterprise applications by integrating components together is a fundamentally new kind of activity, we coined the term Enterprise Component Integration (ECI) to cover technologies specifically aimed at making such integration and customization tasks faster, easier, and more risk free.

Possibly the most successful integration tool was Microsoft's Visual Basic. Visual Basic was highly successful at building client-server applications from components (OCX controls). At one point, Visual Basic was the most popular programming language in use. Unfortunately, Basic is more of a scripting language than a programming language, so it is not a good language for building mission critical applications (for a number of reasons, including a lack of strong type checking). But Visual Basic demonstrated that visual tools are more appropriate for integration tasks than traditional programming tools.

What technologies are needed to support application integration from components? The following is a partial list of facilities that are needed for integration of enterprise applications:

- Facilities for writing (and reusing) glue code for connecting components together.
- Support for strongly typed programming languages, to prevent errors.
- Facilities for discovering, exploring, and testing component interfaces.
- Facilities to support caller interfaces (events).
- Extensive debug and testing facilities, preferably so that debugging can be done at the same time as editing, so that changes can be tested as they are made.
- Facilities for controlling complexity, so a large system does not become a rat's nest of connections between components.

For methodologies, the primary requirement is flexibility and rapid prototyping. Thus, for assembling components, methodologies like Rapid Application Development (RAD) and Extreme Programming (XP) are preferable to traditional (waterfall) methodologies.

For integration support, we have built an application integration framework called AppComposer. AppComposer has several new technologies that make integration and customization easier. For example, **behaviors** are used to build the glue code to connect components together, and the behaviors are themselves reusable, speeding up integration. A technology called **capsules** is used to control complexity of large systems by organizing glue code hierarchically. Capsules are themselves components, so they

ADVANCED INFORMATION

promote reuse at multiple levels, including the reuse of glue code. Such reuse is vital for the creation of increasingly complex software components.

A significant feature of AppComposer is **live editing**, which allows a large system to be modified while it is running. This allows the impact of changes can be seen immediately. Previously, live editing like this was limited to scripting languages (like Visual Basic), but AppComposer provides live editing to a full, statically typed programming language (currently Java, but the same technology is applicable to other languages, such as C#). And like Visual Basic, AppComposer is visual.

Our experience has shown that use of the proper tools and methodologies can reduce the time (and risk) associated with integration to the same size as that for building application functionality (the third bar in Figure 7), enabling significant productivity gains. Because the glue code is also reusable, the resulting systems tend to be more reliable. And by using a visual RAD tool like AppComposer, the resulting applications are easier to maintain and change over time.

Enterprise Application Integration

ECI is primarily concerned with integration between components, but there is a similar term used to describe a superficially similar integration task — integrating separate applications together so that they work as part of a larger system. Enterprise Application Integration (EAI) is a major problem in the modern software world. Analysts estimate that 75% or more of the time and money spent on enterprise applications is spent on integration. An entire industry exists to solve EAI problems.

EAI is a big problem for several reasons. Probably foremost, most existing applications were not designed to work together. Many do not even have application programming interfaces (APIs) at all. For example, it is estimated that more than half of all business data is still stored in IBM's CICS (Customer Information Control System) whose only interface was through alphanumeric terminals. To get at data buried in CICS systems, companies had to write “screen scraping” programs that emulated a CICS alphanumeric terminal, passing queries as if a user had typed them in, and (painfully) extracting the answer from the resulting display screen (as if scraping the data from the terminal screen).

A secondary problem is that applications are not designed with good interfaces, so when the application changes, usually the interface to the application changes dramatically. If a company is trying to build an e-commerce system that accesses data in half a dozen legacy systems (inventory, shipping, accounts receivable, etc.) then if any of the legacy systems are upgraded, it will likely break the entire system. And it might break the system in ways that only become apparent after the entire system is deployed (i.e., right during the height of your busiest shopping season).

CBD can help solve some of the problems of EAI. The same issues that affect integrating components when building applications also affect integrating the resulting applications together. If applications are built using appropriate CBD methodologies, then the resulting applications are also easier to integrate together, and tools that are good for integrating components together are naturally useful for integrating (component-based) applications together.

ADVANCED INFORMATION

Even if you need to connect to legacy systems that were not built using CBD, **connector components** can be used to isolate changes in the legacy applications. There are already a number of companies that develop and sell connector components for a wide range of legacy applications [\[add references\]](#).

For a well designed and implemented component-based system, however, the term *application* starts to lose its meaning. You might have already noticed this in your desktop applications. For example, Microsoft (and others) used its COM component architecture to provide *publish and subscribe* services to its productivity applications. Publish and subscribe can be used, for example, to insert a spreadsheet into a word processing document or a presentation. So, when you are editing a spreadsheet inside of a text document, which application are you using? Likewise, rather than having each desktop application supply its own spell checker with its own dictionary, it makes more sense to have a component that provides spell checking services to other components.

Modern enterprise applications are also better designed as a large system of components, rather than as separate applications. Indeed, one of the main purposes of ERP (Enterprise Resource Planning) and BPM (Business Process Management) systems is to blur traditional application boundaries. SCM (Supply Chain Management) extends this concept even further so that components in different companies can work and communicate together.

Note that there will likely always be a need for EAI solutions to integrate with legacy applications that were not built using CBD methodologies. But the right way to do this is to provide components to interface with each kind of legacy system, and then integrate these EAI components using ECI tools.

Conclusion

We believe it can be possible to double the complexity and power of software every year or two without increasing costs, but it will require a dramatic change from the traditional craftsman style of programming to a robust industry based on reusable, scalable components. To achieve this, we need support for finding, testing, and integrating components. We need to shift from OP methodologies and synchronous programming to component methodologies and asynchronous programming that support scalable, reusable components.

These changes will necessarily require giving up some of our currently accepted software practices, but the rewards are worth it.