



Chasing Silver Bullets: Realities of Software Factories and Component Based Development

[Rev 1.1, Sept 14, 2005]

Cogitance Whitepaper Series

. . . the Industrial Revolution of software is finally upon us. Specialization of resources, standards for interchangeable parts, and streamlined assembly tools have been used in other industries for hundreds of years to speed the development of highly complex products. Despite their ubiquity, application of these concepts to the modern software industry is just beginning.

Bill Gates, Chairman, Microsoft, 1997

To understand the .NET runtime, you must understand this: The .NET runtime is designed to give first-class support for modern component-based programming - directly in the runtime. In other words, it's all about components. If you understand this, understanding why the .NET runtime is designed as it is becomes much easier.

Dr. GUI, 2003

Abstract

Are we on the cusp of a manufacturing revolution in software development, the assembly of applications in software factories using interchangeable parts? Or is component based software development and “software factories” just another silver bullet? Developers are continually separating fact from hype, learning to integrate new tools within existing frameworks, and when proven effective leverage new processes along the way. The goal of this paper is to elucidate what component based development really means for developers. Specifically, ***what are the risks, rewards, and responsibilities of using components***, how does it relate to our existing design and development models and tools, and how can we as developers use components to our competitive advantage.

Overview

In 1798 an American inventor named Eli Whitney built a firearms factory that employed standardized, interchangeable parts. Whitney's idea was simple: you could manufacture products much faster by assembling them from pre-made parts rather than hand-crafting each one. This idea of mass production revolutionized manufacturing.

Is the software industry poised for such a revolution? Software today is currently hand-crafted in such a manner that at best defies predictable estimation and at worst becomes a ruinous death march. Instead of handcrafting, why not assemble software applications? The benefits boggle the mind:

- Lowered cost
- Higher output
- Improved quality
- Lowered skill requirements
- Higher maintainability
- Greater flexibility
- Faster time to market
- Reduced impact of change

There are 2 basic obstacles to realizing this vision: **Factories** and **Parts**.

What would a software factory look like? In the software industry it is construction, not production, that is the issue. A software factory must be a place where software construction is assembly-lined. A software application built on such an assembly line needs parts. These parts need to fit, not in one application, but unlimited applications.

Software parts and factories are much more complex than the metaphor provides, and the software industry as a whole has been grappling with that fact since its inception. Are software factories and parts viable? If so, where are we with respect to achieving them?

Factories

The software factory is not a new idea. Japanese companies in particular have employed factory-like approaches to software construction. During the 1980s, Japan's larger computer manufacturers, including Hitachi, NEC, Toshiba, Fujitsu, and Mitsubishi all were engaged in extensive "software factory" approaches to software development [CUS03]. These companies defined software factories as applying factory-style methods and philosophies to improve software development. These methods included:

- Centralized project management
- Standardized tools and methods
- Reuse of system components

These approaches have proven successful in improving software construction in situations where the problem at hand is exceedingly well defined and constrained, which is not a place most software project find themselves.

The Microsoft Way

Microsoft creates the tools more developers use to build software than anyone else. In 2005, Microsoft defined a software factory as "a development environment configured to support the rapid development of a specific type of application." [Gre04]. This is a markedly different view from the philosophical ideas of the Japanese software factories. To begin with, it is a pragmatic definition, not a philosophical one. As Microsoft explains in their .NET overview:

To understand the .NET runtime, you must understand this: The .NET runtime is designed to give first-class support for modern component-based programming - directly in the runtime. In other words, it's all about components. If you understand this, understanding why the .NET runtime is designed as it is becomes much easier. [MS03]

This strong statement reflects a consistent evolutionary path of Microsoft tools and technologies. Without a doubt, Microsoft is, and has been, very firmly committed to component based development at the **programming level**, which is significantly different than at either the process, design, or philosophical levels.

The "Software Factory" however is a controversial metaphor. Proponents say it represents an enduring vision for software development, following in the path of the great industrialized efforts, from clothes to foods to machines and now to software. Critics argue that the metaphor oversimplifies the development process, doing more harm than good. The critic's typical contention is twofold:

1. **Development versus Production.** The purpose of an assembly line is production. However, in software, strictly speaking there is no production. The primary issue in software is development, and development is not akin to production. If you want to compare apples to apples, they say, software design and construction should be compared to **factory design and construction**.

| Microsoft's Component Roadmap | |
|-------------------------------|---|
| 1981 | ■ Static libraries (LIB) |
| 1985 | ■ Dynamic libraries (DLL) |
| 1990 | ■ DDE |
| 1992 | ■ DLL with extensions ■ Export C++ classes (MFC) |
| 1993 | ■ OLE 1.0 |
| 1994 | ■ OLE 2.0 (COM) |
| 1995 | ■ DCOM |
| 2002 | ■ .NET |

Figure 1: Component roadmap

2. Software is a malleable digital product, not a physical good, and therefore subject to different dynamics. For example, software does not wear out. In addition, unlike physical products, the customer expects their software product to change after receipt, in terms of patches, upgrades, etc.

Clemens Szyperski, a component proponent and software architect with Microsoft Research has suggested that we might consider a software factory as a special kind of factory, where the product being assembled is a software factory [Szy03]. Consider the automobile assembly line. On these lines autos are typically assembled piece by piece by a series of robotic machines. In our factory metaphor, the product the developer is building is these robotic machines.

For a small to medium complexity application this metaphor can be overly complex. For example, we typically only need to develop one application, not one thousand. Where is the benefit to building a robot that builds the application for us? Can it build other applications (application domains)? Can it adapt to evolving requirement sets in real-time?

The software factory, like an assembly line, presupposes a waterfall methodology whereby most or all requirements and details have been worked out up-front. Japan's experience with software factories reinforces this. Michael Cusumano, in his book *The Business of Software* notes: "...software factories seemed designed and best suited for companies building product lines, similar systems, or well defined products with domain constraints" [Cus03].

But what about the rest of us? Can't we just ignore factories and use the parts instead? Indeed, the greatest risk with software factories may be that their inherent complexity causes us to lose sight of the original value Eli Whitney recognized: **Assemble complex products from pre-built parts.**

Parts

Software components are like standard reusable parts – and as such, they could spur an industrial revolution in software, shifting the emphasis from hand-crafting to assembly.”

Clemens Szyperski & David G. Messerschmitt

If the ultimate evolution of software production is to follow other manufacturing models, it is the interchangeable part (component) that is most lacking today and the greatest obstacle towards realizing this vision.

But what exactly is a software component? Bertrand Meyer offers possibly the best real-world criteria for what a component needs to support in order to be useable as an interchangeable software part [Mey00]:

1. May be used by other software elements (clients)
2. May be used by clients without the intervention of the components developers
3. Includes a specification of all dependencies
4. Includes a precise specification of the functionalities it offers
5. Is usable on the sole basis of that specification
6. Is composable with other components
7. Can be integrated into a system quickly and smoothly

To support this criteria, a component's structure should be:

| | |
|---------------------|--|
| Encapsulated | It protects its implementation from us, and vice versa |
| Descriptive | Both us and the framework can discover it's services |
| Replaceable | We can easily swap in new versions |

In addition, a component should contain:

| | |
|-----------------------|---|
| Interface | The contract that specifies what the component will do |
| Implementation | The component's underlying code |
| Package | The physical component file (in .NET this is the DLL or “assembly”) |

Apply these criteria to the alternator in a typical car's engine:

| | |
|----------------|--|
| Encapsulated | Yes. Client does not typically modify the inside of an alternator |
| Descriptive | Yes. A schematic is typically published in alternator's manual. |
| Replaceable | Yes. Author has personally done it. |
| Interface | Yes. Typically 3 electrical terminals. |
| Implementation | Yes. Sub-components inside the metal casing make an electric charge. |
| Package | Yes. There is a metal case that bolts onto the car's engine. |

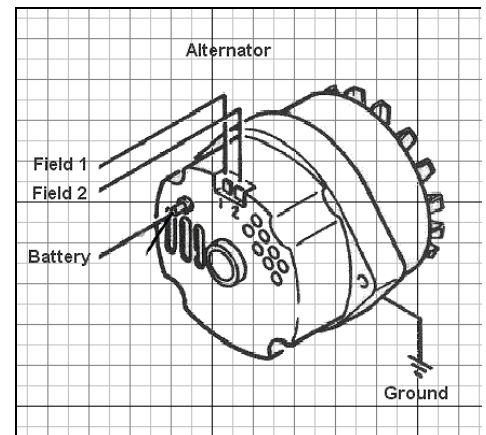


Figure 2 : Alternator as component

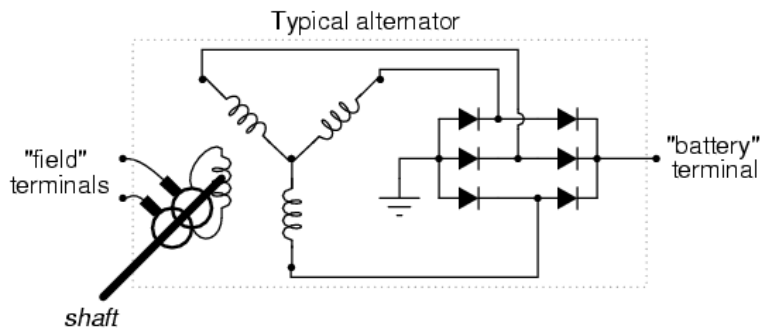


Figure 3: Typical alternator schematic

There is one significant aspect missing: **an alternator is not extensible**. One does not typically upgrade or patch an alternator. When it is replaced, it is replaced with an exact duplicate of itself, often made by a different supplier. This is perhaps the most significant distinction between physical components and software components: **Customers do not expect the functionality of a hardware component to change over time.**

An alternator is also a collection of parts itself. This principle of reusable parts used to make more complex reusable parts is the building block of complexity and in itself reveals the true strength of decomposition by component.

Component Definition

Having discussed the requirements for a software element to act as an interchangeable part in a software application, we can now define a software component.

Software Component : A Software Component is a reusable, self-describing piece of software that can be independently deployed and integrated with other software parts without modification.

Component Based Development

During the web hysteria of the late 20th century, component based development practices (CBD) were hyped as the next “silver bullet” to solve the software complexity issue. Most of the web companies are gone, but what about the hype?

The software industry is wary but not immune to the lure of silver bullets, which underscores the strong understanding and desire among many to improve predictable software development. Developers, however, are at worst quick to dismiss anything containing the word “methodology” and at best simply pragmatic. Since developers construct software, let's look at CBD from a developer's perspective.

From a pragmatic viewpoint, components are a good thing. We know this because we know software is hard to write, and if can reuse another piece of code, especially if it is code we don't want to write, we will. Software factories, however, and we're not so sure. The question then is, How can we benefit from components without incurring the overhead of the “software factory”? More importantly, how do components influence how we currently build software? Where exactly do components fit into our existing methodologies? For this paper we will borrow the term Component Based Development to simply mean that collection of tools and processes (practice) that help us answers these questions.

Component Based Development (CBD) : Component Based Development is the practice of developing software by assembling software components.

To develop software, basic questions about the software project must be answered. At a minimum we must know:

- **What are we building?**
- **How are we going to build it?**

To answer these two basic questions, the software industry has produced numerous **methodologies**. Most methodologies involve some form of the following activities:

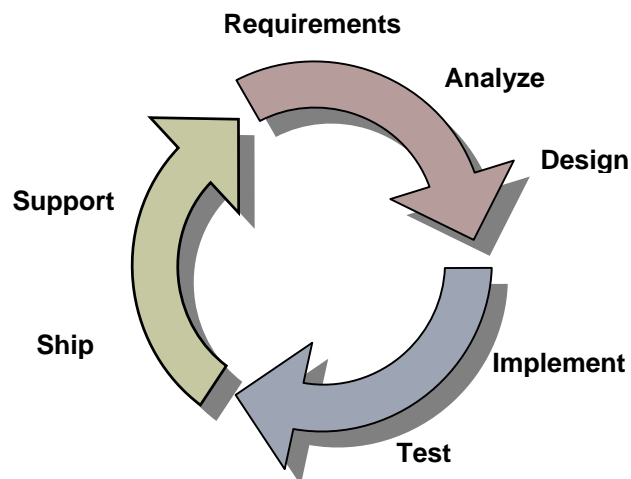


Figure 4: The software lifecycle

Over the past 50 years this software lifecycle has changed precious little. We have, however, established ways of doing it. In particular, we have established:

- Development Paradigms
- Design Paradigms
- Programming Paradigms

The following diagram relates software lifecycle activities to these paradigms:

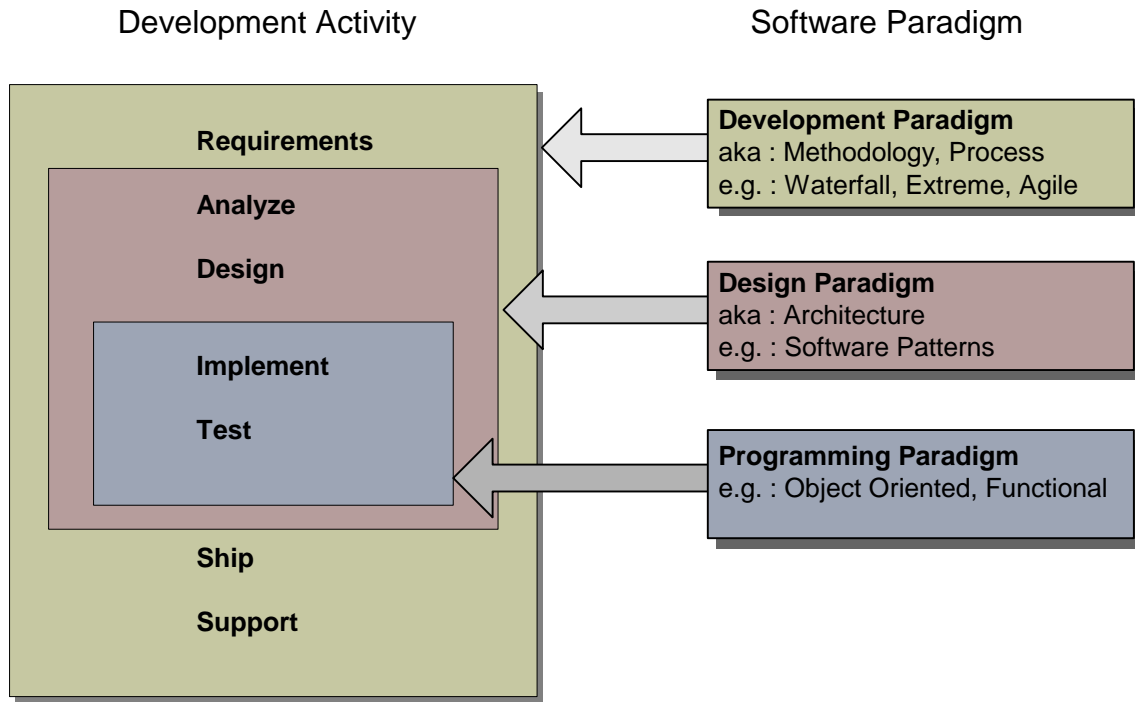


Figure 5: Paradigms applied to the software lifecycle.

The development paradigm applies to all stages of the lifecycle. It is a meta-paradigm in that it seeks to model the software lifecycle itself. The Design and Programming paradigms are more often used by developers, and included specific tools and structures for performing specific programming related activities. Component based development impacts all of these paradigms.

CBD and the Development Paradigm

Some software groups use methodologies (Extreme, Agile, RUP, Waterfall) and some do not. How does CBD work with these existing methodologies? Are we better off without one? There are certain development paradigms that are simply not well suited for component based development. Evolutionary prototyping, for example, presumes a software project's features cannot be known up front, but are evolved iteratively through interactions with the customer. Components, by definition have their features declared up front. Unless we can evolve the component's feature set, evolutionary prototyping will not work.

The following table is a brief summary of suitability issues between CBD and existing software development methodologies.

| Methodology | CBD Suitability | Pros | Cons |
|--------------------------|-----------------|--|--|
| None | High | Provides stability and structure in an otherwise ambiguous process | Difficult to justify expense High risk of components not interoperating |
| Waterfall | High | Up front spec allows selection of components | |
| Evolutionary Prototyping | Mixed | Components can provide instant prototype | Cannot evolve components |
| Spiral / RUP | Mixed | Working software significantly reduces risk | Can depend on length of iterations and "specability" of features |
| Extreme / Agile | Low | | Cannot refactor components |

Figure 6: Suitability of component based development (CBD) for common methodologies

The following is a simple example of a component based lifecycle as applied to the waterfall method.

Component Based Development Tasks During The Waterfall Process

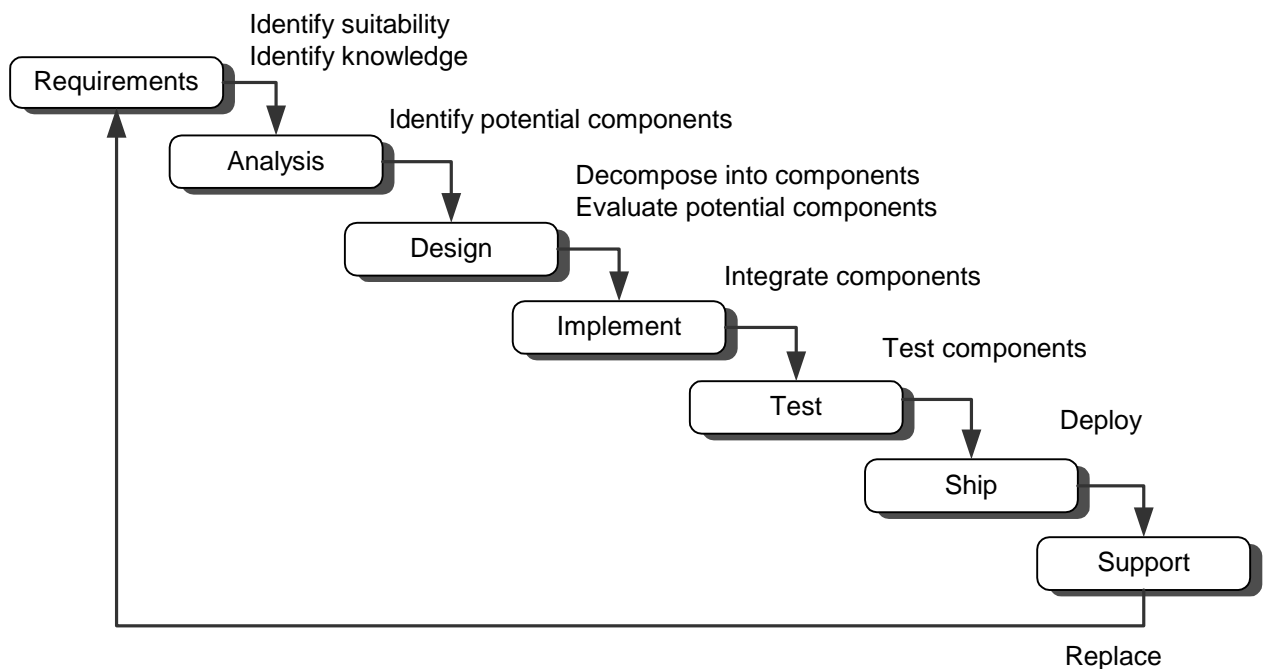


Figure 8: CBD alongside the waterfall

The preceding diagram suggests that most of the additional work required when doing component based development is at the front end, where specification and design lead to component selection. While this maps neatly to waterfall processes, there is no reason why these considerations can't be applied to any particular methodology.

CBD and the Design Paradigm

To be clear, when we talk about the design paradigm we are talking about architecture, which is to say, how are we going to build this system, specifically? At this level, what requirements are unique to component based architectures? Components typically

1. Require a model or framework (e.g. CORBA, COM, .NET) that specifies assemblage
2. Implement specific “design patterns”
3. Relate strongly to architectural decomposition
4. Make design assumptions that may not be readily apparent

The Microsoft .NET platform has significantly enabled the practice of CBD because of the first and most significant item : a component framework. The .NET framework is large, robust, backed by Microsoft, and exceedingly easy to use for novice and experienced developers alike. These factors support Microsoft’s claims of revolutionizing software development through what they term software factories.

What the .NET platform does not provide for is the remaining three. Specifically, while there is a component platform and an increasingly growing number of components available to developers, there is no factory whereby these components can be rapidly assembled into meaningful applications. While there are a growing number of components in the toolbox, what is not present is a guide to architectural decomposition along component boundaries. This is where application domains and frameworks come into play and where we begin to find a more pragmatic role for software factories. For the developer, the need for application frameworks and software factories is thus directly related to the ability to architecturally decompose their software project, and whether this architectural decomposition follows a known domain or not.

Perhaps a more serious implication for the developer at the design level however is the design of the components themselves. Design assumptions made by a component developer may not be readily apparent to the component consumer, but can become serious problems when issues of scalability or performance are addressed later in the project. One needs look no further than the aftermarket for GUI components for the .NET framework. Many of these commercial components are enhancements of components that ship for free in the .NET toolbox. When considering a component during the design stage, it is as important to know what the component can’t do as well as what it can do.

CBD and the Programming Paradigm

Components by nature are typically encapsulated, and as such map well to object oriented decompositions of specific tasks. This relationship of component oriented development to object oriented development is an evolutionary extension, and reflects one of the basic tenets of building solid, reusable code: **tight cohesion, loose coupling**.

This tenet is at the cornerstone of modern object oriented programming (OOP), which has offered significant advances in software modularization, primarily through encapsulation. Proponents of OOP promised a revolution of code re-use at the time. This revolution did not occur. So how is a component so much different from a well written object that it can promise what OOP did not deliver? What, in short, is the difference between Object Oriented design and development and Component Oriented design and development?

Packaging, for starters. While packaging in itself is neither technically complex nor sufficient, the powerful combination of run-time packaging and a framework (e.g. Microsoft's .NET) makes it easy to both create packages (easier than COM) and use packages (VS.NET and reflection, metadata, designer support). In addition, the VS.NET toolbox ships with a considerable number of ready to use, drag-and-drop, software components. From a pragmatic perspective, packaging in this way has significantly improved developer productivity in the real world.

Components are, in reality, the natural evolution of object oriented design. There really is no significant break between the two. There is no OO versus CO – this is the stuff for magazine articles and panel discussions for people who don't write software. Developers know there is very little discontinuous innovation in software. Software evolves. Most components are classes wrapped in packages with self-describing metadata that a framework tool like VS.NET can use to expose them in a friendly way to component consumers.

Still, **packaging is everything**.

In addition to packaging, there are subtle difference between OO and CO. For example, implementation inheritance in object oriented programming led to tighter coupling, which in turn made re-use more difficult. Often times objects were tied to parents which could not easily be extracted for re-use in other code.

The following table lists the major differences between components and objects. You can decide whether they are revolutionary or evolutionary.

| Objects | Components |
|--|---|
| Bound to OO language (C++, Smalltalk) | Can be written in any language |
| Tight couplings (implementation inheritance) | More loosely coupled than objects (Interface inheritance) Functional Aggregation |
| Fine granularity in object decomposition | Larger granularity |
| Limited forms for accessing (methods) | Interface oriented |
| | Mechanism for runtime discovery |
| | Designed to obey rules of a common framework (.NET, CORBA, COM) |
| | Has a binary specification allowing run-time discovery and reuse |

Figure 9: How components differ from objects

Moving Beyond the Hype

"The Industrial Revolution of software is finally upon us" makes good copy but not much sense. It undermines the evolutionary progress of software development advancements, and provides a metaphor that at best has no pragmatic value to working developers and at worst confuses everyone. The reality is the software factory depends on a more complex understanding of application domains and frameworks than we currently have.

However, this does not mean we should throw the baby out with the bathwater. If developers did that with every over-hyped, under-promised technology we'd still be using punch cards. Building software from components is something we can do today with real productivity gains. Component based development is not a software methodology. It is not a prescriptive plan for how to go through the software lifecycle. It is the acknowledgement that software is an inherently complex product. Like other complex products, a proven approach for managing this complexity is to break it down into constituent parts. While CBD can coexist with current methodologies, what is needed is a set of best practices for building software from components. These practices serve two purposes: First, to help developers construct with components wisely and productively today. Second, to build a set of requirements a software factory would need to uphold should such a factory exist.

While a set of best practices for component based development is beyond the scope of this paper, what we can do is call out some of the risks, rewards, and responsibilities developers and managers alike can expect from CBD. We call it developing with eyes wide open.

Real Risks

Outsourced Knowledge

Components are outsourcing. When assembling applications developers must be careful to avoid outsourcing knowledge development that yields greater benefits than just the code being constructed. This typically comes into play when technical support is required. Relying on a component vendor for technical support is perhaps preferred for ongoing in-house software, but can be potentially disastrous when the customer is a large, blue-chip client that expects the organization to have product expertise.

Black Boxes

Components are black boxes. This can make it difficult to know what design assumptions about the component have been made, which in turn can make it difficult to predict how the component will perform. Load testing is a good example. For what kind of throughput has the component been designed?

Dependencies

Components have dependencies. These dependencies become the client application's dependencies.

Quality

Components vary in quality. Vendors have different ideas of quality. There is no independent certification for software components. There is a prevailing presumption that because a component is used by several if not hundreds of clients, all of the component's bugs have been shaken out. That is, it has undergone real-world testing, and that this is a unique property of components. While this may be true, the fact remains different vendors and customers alike have different standards of quality.

Interoperability

Components must work together. How? Components come in many different packages. Even on the Microsoft Windows platform we find:

- Source code (e.g. C libraries)
- Static libraries
- Dynamic libraries (DLL)
- COM and ActiveX controls
- .NET Controls and Assemblies

Managing these interoperations is the role of the framework. For example, the .NET framework provides three basic services to handle interoperability:

- COM Interop
- Platform Invoke (P/Invoke)
- C++ Interop (or IJW "It Just Works")

Evolution

Software changes. What is the vendor's update cycle for the component? What features will be included? How do you synch application development with the component update cycle? How can you ensure application critical features will be included?

To reduce the long term risk of component updates, the developer typically has three options:

1. Motivate the vendor to add the required functionality.
2. Modify the component in-house.
3. Replace the component with another vendor's component that has the desired functionality.

These options are often not available. To mitigate long term risk developers should establish specific change request procedures with the component vendor, or else code for replaceability.

Glue Code

Most components will require application specific integration code (glue). This code will need to be maintained. This is also where we code for replaceability.

CBD and Software Processes

What requirements exist for doing component oriented development? Can these requirements be integrated into an organization's development paradigm? For example, in current agile or XP evolutionary style paradigms, designs are not always specified up front. In component based

development it is not always possible to add features to components as necessary. To mitigate risk, required component functionality needs to be specified during the evaluation phase.

Architecture

Within the design paradigm, a system architect, even on a team of one, will need to understand how to decompose user requirements into particular components. Does the proposed project decompose along well defined component boundaries? What metaphors can help us in the decomposition?

Real Rewards

We have seen the following benefits typically attributed to components:

- Cost savings
- Time to market
- Code reuse
- Higher quality
- Tool Flexibility
- Reduced risks
- Developer focus

The goal of this section is to evaluate the pragmatic value of these claims.

Cost Savings

Likely, but how much?. Let's do a rough estimate:

Components typically cost less than US\$1,000.00. This would buy on average three (3) days work for an in-house developer based in the United States to possibly fifteen (15) days work for a developer in the global marketplace, for an average of nine (9) days. Typically only small, focused feature/function points of software can be designed, implemented, tested and debugged in nine days. For functionality of the typical software component, the estimated time is significantly greater. On the surface the savings appear to agree with what intuitively seems correct – it is cheaper to buy than to build. But how much cheaper?

Software cost estimation is notoriously difficult. A simple approach is to measure the lines of code used in the purchased component and compare that to the average time required to write those lines of code in-house. Multiply this by the rate the developer receives to establish a base price structure:

Cost to Build = lines of code × hours per line of code × developer cost per hour

The cost to buy is also tricky to establish. In addition to the developer license fee, there may be royalty or “run-time” license fees that are charged on a per seat or application basis. There may also be support or maintenance fees. More significantly, there is the cost to evaluate components for suitability, and the cost to actually integrate the component, which may be up to as much as 20% of the cost to develop [Bro02]:

Cost to Buy = license fees + runtimes + support + cost to evaluate + cost to integrate

The reality is cost savings can be quite large depending upon the complexity of the component. However, when purchasing components developers need to evaluate components against their

total cost structure, not just the license fee. In addition, the cost of lost expertise (that is, that cost of not having the expertise in-house) must be factored in by the company, specifically with regards to application support.

Time to Market

Likely, except in the unfortunate situation where time to integrate is longer than time to develop. To avoid this, developers should allow sufficient time when evaluating the target component in terms of ease of integration with existing software as well as compatibility with the deployment platform. This last point can often go overlooked if the integrator is working on a sub-system of the main application and is not aware of the main application's possible distribution constraints. For example, with .NET components, are components required to be installed into the global assembly cache?

Code Reuse

Likely. A well packaged and well documented component is designed specifically for re-use. The ability to create in-house code libraries offers perhaps the greatest reward both from an application developer's perspective and a business cost savings perspective.

Higher Quality

Likely. Quality is relative in a software organization. Certain companies prefer "good enough" software practices that allow them to ship and fix. Other organizations require strict adherence to established quality procedures, such as ISO 9000 or CMM level standards.

The assertion of higher quality of components is generally based upon the notion that components have been installed and used by many developers. This process, over time, has detected and eliminated most known bugs from the component. This is a solid argument, as there is simply no replacement for real world testing. Excepting shrink wrapped software companies, few development houses have dedicated test staff for software projects.

In the future we may see software component certification. The likelihood that components vendors would embrace a certification process depends on how much cost and trouble the process involves weighed against the perceived value and/or requirement by the customer.

For the foreseeable future, the likelihood of higher quality is thus dependent upon the number of users of the current component and the responsiveness of the component vendor in resolving discovered product defects.

Reduced Risks

True and False. Short term risk is mitigated by proof of working, functional software. Assuming the developer proves it meets his initial software needs, risk is substantially reduced.

Long term risk, however, grows as the application software evolves new requirements for the component. For an in-house piece of software, code updates can be made immediately. The changeability of the internal workings of a COTS component however is typically dependent upon the component vendor, who may be unwilling or insufficiently motivated to make changes within a timely manner.

Developer Focused

There exists the idea that components free developers from working on mundane code that has already been written. This claim is both true and false. Part of component oriented development is the notion of **specialization**. Specialization in all industries has occurred rapidly through the last half of the 20th century. While software development is a comparatively young industry, it too has seen a great deal of specialization. However, this specialization typically takes the form of languages and tools. As the computer is the general purpose tool, computer programmers have been the general purpose tool makers. Domain specific knowledge typically exists at the business end, with domain rules handed to a general purpose programmer for encoding.

Components suggest a new level of specialization, whereby programmers produce parts within a particular domain. This ability for domain knowledge encapsulation in reusable code rather than in business rules allows for application integrators to assemble domain knowledge where necessary. This suggests a more generalization of developers rather than a specialization, as developers become more like integrators of specialized parts rather than constructors of specialized parts.

What is not true is the implication that programmers will simply assemble domain encoded parts into applications. Application development of any novelty will always entail programming details. In addition to writing application specific code modules, application integrators will need to become expert component integrators, familiar with component frameworks (such as Microsoft's .NET), interface based programming for isolating components that may be placed, integration code for integrating a component into the application, and application infrastructure for holding these components together.

Real Responsibilities

We can improve our chances of successful component based development on software projects if we acknowledge and identify key responsibilities in the relationship between component supplier and component consumer.

| Supplier | Consumer |
|--|---|
| Support some form of the minimum component properties: Encapsulated, Descriptive, Replaceable. | Have a Process. Develop a component based development strategy that works with your existing software development. Realize no methodology is still a methodology. |
| Provide a clear specification for component interoperability and integration. | Allocate time for evaluating components. It's possible the time taken to evaluate will be longer than the time taken to integrate. This is OK. If components really do save time and money, you can afford to pick the right one. When evaluating, consider how both functional and non-functional system and system building requirements map to a components requirements, including: <ol style="list-style-type: none"> 1. Documentation 2. Total product cost 3. Licensing 4. Quality 5. Support 6. Component dependencies (including the framework architecture) 7. Vendor update cycle |
| Provide the component in a standard packaging and deployment model. | Test the selected component for compatibility. Since it is difficult to test the component completely without integrating it completely, Are there other applications with a similar context using this component successfully? One strategy is to use context comparison for component breadth testing, and specific tests for non-functional testing, such as component performance under anticipated load, which would vary from context to context. |
| Provide a mechanism by which the client can predictably replace components with new versions. | When integrating the component, code for replaceability to minimize risk if and when the component needs to be swapped out |
| Provide a clear mechanism by which clients can add features to the component. Provide a predictable release schedule that allows clients to synchronize their development schedules with component version releases. | Plan for evolution. Coding for replaceability is a defensive measure. In addition, developers can pro-actively seek to clarify a maintenance relationship with the component supplier that provides for unanticipated feature changes, especially if using an evolutionary/agile/XP development methodology. |

One key uncertainty in the role of the component supplier is customizability. To what extent does the component supplier provide a means by which the client can extend or customize the component? This is a key departure from the notion of standard, interchangeable parts we have been discussing, and is the result both of software's malleable form and our need as client to modify it.

Summary

Any activity becomes creative when the doer cares about doing it right, or doing it better.

- John Updike

Software development is creative. Software is not a manufactured product. It is still predominantly a creative, problem solving process done by creative people for specific problem sets within changing environments. Any attempts to automate the software development process must account for this.

Software is digital. The economic models of a software parts industry do not match traditional products industries. A metaphor like "software factories" helps us to understand where we have been and where we may be going. Taken too literally, however, and the metaphor does more harm than good.

Software development is evolving. Like its predecessors, component based software underwent its period of "revolutionary" status. It's 15-minutes of fame in the software world. The reality is in our current software development market we find mature frameworks and robust components, a quietly developed component marketplace, and a tacit acceptance that "buy or build" is a natural design question. In fact, assembling applications from pre-built and in-house components is simply considered good software practice, implementing healthy modular principles core to computer programming and separation of implementation and interface. Many developers today would probably claim they've been doing component based software all along. Such is the true status of component based software today.

Finally, software is complex. The risk of failure for any software project of moderate size is high. To minimize this risk, as well as maintain a competitive edge in a global business environment, project managers, architects, and developers must decompose the complexity of the project into understandable chunks. During decomposition, decisions can then be made about which parts of a software project can and should be outsourced, and if so which form this outsourcing should take. Component based development helps us do that now.

If and when a true software factory emerges, it will be built on the foundations of today's software engineering principles. Those interested in quick, discontinuous "revolutions" in software development are better off chasing silver bullets.

About Cogitance

Cogitance specializes in component based solutions for emerging media applications. Cogitance provides next generation software parts that combine low-level native multimedia code with the high-level rapid application development environment of .NET. Cogitance also help clients refactor legacy code into component based managed code, preserving intellectual assets that provide rapid redeployment for rapidly changing markets. More information on Cogitance can be found at www.cogitance.com.

References

- [Bro02] Brooke, Chris. "The Return on Investment on Commercial off-the shelf (COTS) Software Components." *Component Source*. August 2002.
- [Cus03] Cusumano, Michael. *The Business of Software*. Free Press/Simon & Schuster, New York, NY, 2004.
- [Gre04] Greenfield, Jack. "Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools" *MSDN Library*, November 2004, <<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/softfact3.asp>> (10 June 2005).
- [Mey00] Meyer, Bertrand. "What To Compose." *Software Development* 8, 3 (March 2000).
- [MS03] Microsoft, "Dr GUI .NET 1.1 #0: Introduction to .NET, Hello World, and a Quick Look Inside the .NET Runtime." *MSDN Library*, 24 April 2003, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnguiet/html/drguiet0_update.asp> (10 June 2005).
- [SM04] Szyperski, Clemens and Messerschmitt, David. "The Flexible Factory." *Software Development* 11, 12 (December 2003).

