NexWave

Component-Based Software Engineering for Consumer Electronics

Abstract

Component-based software engineering (CBSE) has become widely practiced in the desktop and enterprise computing arenas. Yet despite its applicability, the many benefits of CBSE have remained largely unavailable to many embedded software developers.

Meanwhile, the amount of software contained in modern Consumer Electronics (CE) devices over the past decade has exploded, due largely to the market's move from analogue to digital devices (e.g., VCR to DVD). Modern CE companies have had to change almost overnight from hardware suppliers who provide small amounts of software on their devices, to software suppliers who provide generic digital computing devices (with some specialized I/O peripherals) on which their software is run. Many CE vendors are struggling to cope with this profound transformation, and stand to benefit enormously from the advances brought by CBSE.

Most CE companies now have significant code bases, much of which it is difficult for them to reuse effectively. The challenge is to find a way to reuse effectively this existing code, as well as new code that is written over time. Is this paper we argue that CBSE is the best way to achieve this, but explain why we believe currently available component models do not suit the very specific needs of the CE industry.

In this paper, we present our experiences in industry of the applicability of CBSE to Consumer Electronics software, and the problems that so far have prevented its widespread adoption in this field.

1 Introduction

Programmers have known for a long time the benefits of constructing their programs in a modular fashion [1]. As programs have become increasingly complicated, so the need for their structure and modularity has increased. Component-based software engineering (CBSE) [2] has emerged as a powerful tool to help programmers cope with the increasingly discouraging task of managing their programs' exploding complexity. CBSE is increasingly being deployed in the world of desktop and enterprise computing. Technologies such as Java [3], COM [4] and .NET [5] all offer real advantages to software development in these higher-end arenas. Indeed, use of these technologies is now so prevalent that one could argue that, at some levels, CBSE has become more an accepted assumption than a "hot new technology". However, while this may be true for desktop and enterprise software, it is not the case for embedded software, in particular for consumer electronics (CE). It is our experience that much of CE software development would benefit enormously from CBSE.

The amount of software contained in these devices has exploded in recent years. From none or a few kilobytes a decade ago, to megabytes of complicated

code in today's products (e.g., DVD recorders and digital televisions). Embedded/systems software has been largely left behind by the advent of CBSE. While there are embedded implementations of component technologies such as Java [6][7] and .NET [8], these remain exclusively at the application layer (see Section 5.2). While useful for things like an Embedded Program Guide on a set-top box, these technologies are not so useful for the "grunt" CE software, which tends to be comprised of lower-level software, such as device drivers and codecs. Furthermore, even these highly optimized implementations still impose overheads that often are unacceptable for CE. It is ironic that for varying reasons and with varying degrees, existing popular CBSE technologies are not applicable to the embedded/systems software space; yet this arena has as much—if not more—potential to benefit from CBSE than do desktop and enterprise arenas.

2 What is CBSE?

CBSE is the technique of breaking software down into reusable components, and then composing different combinations of components to form different software systems. Developing software in this fashion brings many advantages (explored in Section 1).

One widely used definition is Szyperski's:

> A software component is a unit of composition with contextually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition [9].

A component's interface defines the syntax and semantics of how the implementation behind it is accessed. Implementations are separated from interfaces, in that implementations on either side of an interface know and care only about the interface, not the other implementation.

Components are independently deployed, meaning that components with different authors can be assembled by a third-party, without any explicit cooperation between any of the parties, merely adherence by each to the component model and interfaces.

We define a component model as a set of rules that defines how components and interfaces are represented, defines a set of basic elements from which interfaces are composed (i.e., a type-system), and defines the mechanism whereby components may interact over those interfaces.

Deciding whether or not a software system is classified as component-based is rarely without controversy. We consider the existence of a component model a useful predicate that can be used to define whether or not a system may be considered component-based. Although an explicit definition of the component model is preferred, the component model may also be defined

implicitly (i.e., by its usage). For example, consider UNIX [10] processes communicating using pipes: a simple, informal, yet effective component model [11]. The UNIX shared libraries and the Windows DLL mechanisms are also component models.

Our categorizing of processes and shared libraries as components may be surprising. In a sense, CBSE is such a good idea that it has been practiced for years, often without acknowledgment. However, these systems suffer several drawbacks as component models. As an example, let us examine UNIX. A UNIX deployment consists of many parts, including the kernel, configuration files such as /etc/passwd and /etc/fstab, the login and init processes, various daemon processes, the X-windows system, etc. These different components look and work very differently, and cannot uniformly interact (files are very different from the kernel, which is different from a process). In addition, the boundaries between components are not obvious (for example, one might consider device drivers as part of the kernel component, as separate entities, or as subcomponents of the kernel). Components such as processes are relatively heavyweight, meaning that the overhead involved in splitting an application into separate processes can be prohibitive.

We have identified five properties that we believe a good component model should possess:

Obvious

Components are clearly separated and well isolated, with strict, well-defined interfaces. This enables one easily to identify the components that comprise a system, and how they are related.

Orthogonal and universal

The nature of a component is independent from the task it performs. System-level components (e.g., a device driver) look no different from user-level ones (e.g,. an image viewer). This means that all components in a system may belong to the same component model, thus extending the benefits brought by components.

Fine-grained

Components are lightweight; the model imposes no or little overhead. This allows systems to be decomposed from an architectural perspective, rather than decomposition being dictated by performance considerations.

Encapsulated

Components can be easily encapsulated and then delivered as a single entity. If done properly, this can prevents "missing pieces" such as header files when components are delivered.

Typed interfaces

Interfaces can be expressed in an Interface Definition Language, which explicitly identifies the syntax of the interface. The component model should guarantee that components will be permitted to communicate only across interfaces where both sides agree on the types.

3 What CBSE isn't

This section introduces two well-known styles of programming that are commonly confused with CBSE: namely modular programming [12] and object-oriented programming (OOP) [13]. While they are both related to CBSE (and to each other), there are important differences that are highlighted here. However, it should be noted that it is impossible to define CBSE, modular programming or OOP without controversy—there are many differing definitions of each, and many of these definitions overlap and contradict one another. That said, what is typically and generally meant when these different techniques are referred to is quite different, and while there are contentious areas where the disciplines overlap, for the most part they can be considered different things. The following two subsections analyze what is typically meant by modular programming and OOP, and contrast them with what is typically meant by CBSE.

3.1 Modular programming

The concept of modular programming has been around for a long time. Modular programming means breaking down software system into its constituent parts, as does CBSE. CBSE, however, takes this notion considerably further. That is, modular programming can be seen as a subset of CBSE—or in other words, CBSE is a form of modular programming, but with additional qualities.

One of the most striking differences between a module in what is traditionally meant by "modular programming" and a component in CBSE, is that components are independently deployed. Components are truly independent pieces of software, which can be brought together with other components at system build time and/or at runtime. Components are designed without the programmer knowing with which other components they will operate—only the interfaces they will use. Typically, when a modular program is compiled and linked, the boundaries between the modules disappear. It is not possible to tell from the executable whether a program was written in a modular way or as a monolithic block. Typically, in component-based systems the boundaries between components are preserved even at runtime.

3.2 Object-oriented programming

OOP is in many ways an extension of modular programming, just as CBSE is. However, OOP offers a different (although overlapping) set of features than does CBSE.

OOP extends modular programming by adding the "pillars of OOP": encapsulation, inheritance, and polymorphism. Concretely, these pillars are essentially features of the programming language's scoping and type-system. For the most part OOP consists of syntactic and type-system extensions to a programming language that augment modular programming (although there is also a philosophy applied).

It can be argued that OOP can be practiced without an object-oriented language. However, here the programmer is still using the scoping and typing rules implied by the use of OOP, just without support from the programming language. In other words, regardless of what programming language is used, OOP is still about applying certain rules to the scoping and typing of a computer program's source code. In contrast, CBSE is a way of working that is independent of the components' source code.

Note that OOP and CBSE are by no means mutually exclusive. It is perfectly possible to write components using OOP, and some component models provide for interaction between components in an OOP style (some component models even require use of object-oriented techniques between components).

3.3 APIs

A set of APIs should not be confused with component-based programming. As an example, an operating system will have a set of APIs, but typically this is not thought of as CBSE. However, it should be noted that CBSE offers a practical and useful way to implement and deliver libraries based on such APIs.

4 Why component-based software engineering?

There are many advantages brought by CBSE. The ones that we consider most important to CE manufacturers are listed in the following sub-sections. Not all component models will provide all of these features, and the list is not exhaustive. An in-depth discussion of each of these advantages warrants a separate paper for each; this paper dedicates just a few sentences to each. The items are listed in the order we believe is most applicable for CE software, starting with the most advantageous.

4.1 Better code reuse between projects

Developing a single software system in terms of distinct and reusable components may take more effort than would developing an equivalent system in an add-hoc, throwaway manner. But as reusable components are added to the population, greater component reuse is facilitated across different projects, and developers see increasing returns.

Effective code reuse manifests itself in three ways. The obvious is the cost of development. Software engineers do not come cheap, and studies have shown that even the most talented engineers are able to write only 10 lines of production-quality code per day [14]. For many CE organizations, time to market is much more important than the cost of development-for the same reason that reused software is cheaper, it can also be produced more quickly than rewritten software. Finally, related to time-to-market is quality. Achieving the quality standards required by customers is often the most common cause of delays in software projects (i.e., many late software projects are late due to protracted bug cleaning phases). Reusable software that has been tested in a multitude of environments is likely to be of a higher quality than software written from scratch.

4.2 More easily configured and customized software deployments

As well as making it easier to add components to a system, CBSE makes it easier to remove components. This means that it is relatively easy to select only the components required for a particular system, omitting unnecessary services. This leads not only to reduced memory requirements, but also arguably to more stable systems, since less code typically means less bugs (just because a particular piece of code is never executed in normal operation does not mean that it is not a liability—such code is the source of many failures which occur in deployed installations). In addition, rarely executed code is frequently targeted in security attacks. The process of auditing highly-dependable systems is also clearly helped by the ability easily to remove unneeded code.

In addition, component-based systems are more easily customized by the replacement of certain components. This is particularly relevant when the system integrator does not have access to the source code, which may be the case when libraries are purchased from independent software vendors.

4.3 OS/Hardware Independence

Many component models allow pre-compiled components to be executed seamlessly on different operating systems, provided the interfaces required by the components can be made available. Some component models (such as Java) allow components to be executed without modification on different CPU

architectures. Others require a recompilation, but even here in our experience it is generally easier to make small components portable than larger monolithic pieces of code.

4.4 Transparent communication between components

Some component models permit components that reside on different computers to communicate across the network[1], so that it appears to the programmer that distributed components reside on the same machine. This is known as location transparency.

This is useful for CE devices that contain several heterogeneous processors, joined by some interconnect (i.e., asymmetrical or NUMA multiprocessor systems). Programming the communication of these processors can be time-consuming and error-prone; a component model that offers location transparency can be very useful in such cases.

Furthermore, the trend in CE is widely anticipated to be towards interoperation of devices, facilitated by wireless communication technologies, such as Bluetooth [15]. Location-transparency clearly has a role to play here.

4.5 Deployment of new services in the field

Many component models provide for loading new components into a deployed system. For this to work, the system must have some medium over which components can be loaded (for example by inserting a DVD, or downloading over an Internet or private connection). If such a medium is in place, this can be a useful way to provide new services on products already deployed, establishing a potential ongoing revenue-stream between the end-user to the device vendor or service provider.

In a similar way, it is more practical to add components (and thus add features) relatively late in the design cycle, compared to doing so with a monolithic application. In fact, this benefit is more a feature of modular programming than component-based software engineering, but since CBSE enforces modular programming, it is possible to have more confidence in the practicability of this approach.

4.6 Dynamic on-line replacement of individual modules

Some component models permit a component's implementation to be replaced on a live system, and without requiring a reboot. This can be very useful when fixing bugs on systems that must be highly available, such as network routers. Some models require a reboot after downloading a replacement

1. "Network" in this context may mean a fixed interconnect, such as PCI bus, or a more traditional open network, such as Ethernet or Bluetooth.

implementation, which suffices in most instances (although it should be noted that CBSE is not the only way to achieve such updates [16].)

4.7 Language independence

Some component models are language independent, or at least permit components to be implemented in a variety of languages. This means that programmers are not forced to use a certain language, and software modules written in different languages can be reused within a single project.

4.8 Improved software delivery processes

Supplying/delivering software as components is much more practical than delivering large monolithic blocks and/or arbitrary collections of source files. A good component model will enable components to be completely self-contained, allowing delivery of a part of a software system or a library as a single file. This can mitigate problems such as a developer not having access to a particular compilation environment when trying to build his system.

4.9 Isolation/containment of faults

Some component models provide for fault containment, whereby an individual component or set of components cannot adversely affect the rest of a system in the case that the component is either buggy or malicious. This protection [17] might be provided by hardware, software or both [18][19][20][21]. Fault isolation is useful to increase a product's robustness, and necessary to support safe execution of code downloaded from untrusted sources.

4.10 Maintainable software architectures

Component technologies make the division of work more obvious and clean-cut than is usually the case (even when modular programming is used). In addition, regardless of the quality and rigor with which a large software system is decomposed when it is designed, over time the modularity of the system often decays as the software evolves. All too often, `quick hacks' are made which progressively impair the system's modularity. A good component model will make these quick hacks less likely since it will be easier for the programmer to observe the systems' modularity (see Section 3.1). Thus CBSE can help to ensure not only a clearer architecture when the product is initially designed, but also help to ensure that the architecture persists during the maintenance of the code base.

4.11 Why not CBSE?

While we would like to think that CBSE represents only Good Things, like anything there are good sides and bad sides. In this section we attempt to list some of these disadvantages in the most honest and objective way we can. Of course, we believe that these disadvantages are outweighed by the advantages of CBSE, and so in this section we also give reasons why we believe the disadvantages are relatively minor.

4.11.1 It takes longer to write reusable code than throwaway code

Writing throwaway code is often easier and quicker than writing reusable code [14][22]. The quality requirements on the latter are typically a lot stricter: reusable code often must behave correctly in a wider range of scenarios and a wider range of input data than must throwaway code. In addition, more care must be taken to providing generic interfaces to ensure that software components can be used in future scenarios, not necessarily anticipated at the time of writing. The savings that are promised by CBSE must be offset against this cost.

4.11.2 High-quality documentation is necessary to make reuse effective

Reuse is not practical unless interfaces are well documented. Computer programmers (and their managers) are traditionally very bad at documenting software at a level of detail and completeness that is sufficient for other programmers to use their components. Thus, component-based-software engineers must learn and practice writing higher quality documentation than they are perhaps used to.

However, it should be noted that writing quality documentation is widely acknowledged as a Good Thing, even for code that is not intended to be reused. Software has a habit of living much longer than the programmer expects, and more often than not the source code at some point needs to be read and understood by other programmers, frequently after the programmer of the original code has left the organization that owns that code.

Moreover, the process merely of writing down what a piece of code does and how other pieces of software can interact it with it, often reveals to the programmer weaknesses in the code's design, allowing the quality of code to be improved.

Therefore, rather than arguing CBSE requiring a higher-standard of documentation as a disadvantage, one could argue that yet another of CBSE's advantages is that it encourages programmers to write good documentation!

4.11.3 Quality processes are necessary to make reuse effective

Effective code reuse is as much (or arguably more) a question of process as it is of tooling. Anecdotal evidence suggests that simply using CBSE makes it more likely that a programmer will at least look for a suitable component already available. But still there must be an effective way for the programmer to find the components that exist. Component cataloging tools [23] can be helpful, but no amount of tooling will substitute good processes.

Software development process and engineering-paradigms/tools are largely orthogonal issues, and both are critical to effectively managing and reusing large amounts of software. In other words, deploying CBSE tools is only part of the answer to solving the problem of code reuse: necessary, but not sufficient [24].

4.11.4 Error handling and propagation is difficult

Error handling is one of the more difficult parts of computer programming in general, and inappropriate or non-existent error handling is frequently the cause of bugs. Error handling is made more difficult in modular programming (and CBSE by implication), because the client components are not necessarily aware of exactly what errors their servers might generate. The problem is that a server's implementation is "hidden" behind an interface. There may be many different implementations, and implementations are frequently developed after the interface has been defined, and often after the client code is written and deployed. Listing the full set of errors that a function may generate is difficult, and doing so will likely constrain any future implementation. This means that a client must anticipate that a function call on a server may result in any kind of error. A client may be able to deal directly with certain kinds of error, but in most cases errors are simply propagated up the call chain to be dealt with by the client's client. When a server returns an error that is unknown to the client, the client's only choices are to propagate or to ignore the error (ignoring unexpected errors is typically not a good idea!).

One of the most common ways to deal with this problem is to have functions indicate errors through their return value (typically an integer error code, although not always; e.g., Plan9 [25] uses strings). If the client is able to deal with the error directly it does so, otherwise the error is simply propagated up the call chain by the client function, itself returning that error. This idiom has the attractions of being proven, simple and well understood. But it is cumbersome for the programmer, and it is easy for the programmer to make mistakes (such as failing to check for returned error conditions, or propagating errors without properly freeing allocated resources). It can also lead to inefficiencies in the code; this technique imposes at least an extra test and branch per function call, even when no error is generated (approximately 5% of typical machine instructions are procedure calls [26]).

To overcome these problems, many modern programming languages support the notion of exceptions [27]. Exceptions effectively automate this error-propagation process, and so remove from the programmer the scope for a common source of errors. In addition, if implemented efficiently, exceptions can eliminate the extra test-and-branch from function calls, leading to more efficient code in the case that errors are not generated. Sadly however, exceptions are usually a feature of the programming language, making them difficult to support well in a language-independent component model. It is difficult to propagate exceptions between different programming languages, particularly languages that don't support exceptions.

In summary, for error handling with CBSE one typically has a choice between integer error codes which are cumbersome, and exceptions which limit the implementation language. Moreover, one is forced to make this choice for the component model as a whole—it cannot practically be made on a per-component basis, since all servers must be able to communicate errors to their clients[2].

4.11.5 Some optimizations become more difficult

Even if the implementation of the component model itself imposes no overhead, there are optimizations that can be made with a statically-linked program that are difficult to make with component-based systems. For example, small functions or functions that are called frequently from few call-sites can be "inlined" with monolithic, statically-linked programs. This common technique (and others) is more difficult with CBSE, because components are compiled and linked without knowledge of how they will be used. (However, that such optimizations can still be performed on an intra-component basis, just not on functions that are called between components.)

4.12 Debunking some CBSE myths

There are several myths surrounding CBSE that we list here and attempt to dispel.

4.12.1 People have been trying this "code reuse" thing for ages, it just doesn't work in practice

The previous decade has seen real advances in the use of components and in code reuse more generally in some areas. Desktop Java programmers think nothing of going to the Internet to look for Java classes that they can reuse before writing their own. Games developers now rarely implement their own 3D libraries, and business programs rarely their own printer drivers, or spell-

2. One can imagine a system whereby servers that generate exceptions can be used only by clients that can handle them, but such a scheme is likely to be difficult to use and restrictive.

checkers. On an MS-Windows system, Excel spreadsheets can be embedded inside a Word document, but Word itself does not duplicate the Excel code; Word and the Outlook mail-client share a common spell-check component. UNIX systems have been employing effectively CBSE techniques for years [28] (see Section 2).

However, not all programmers are able easily to use CBSE. In our experience, embedded programmers still struggle to reuse code effectively, partly because of culture, and partly because of lack of availability of appropriate tools.

4.12.2 Component-based systems are inefficient

It is true that certain optimizations are more difficult to make with component-based code (see Section 4.11.5), but this rarely has significant consequences. Some component models do impose significant inefficiencies, but this is a feature of a particular component model or implementation of that model, rather than a facet of CBSE in general. It is certainly not the case that all component models introduce runtime overhead between inter-component function calls, or expensive virtualization of the underlying hardware.

5 Components for CE software

In Section 4, we outlined general advantages brought by CBSE (albeit with a slant towards CE software). In this section we present the case specifically that consumer electronics has a particularly pressing need for CBSE, especially the software reuse it brings. We then go on to examine why we believe none of the existing component models in common use today is suitable for CE.

5.1 CE's need for components

In our experience, CE manufacturers are increasingly suffering under the huge burden imposed by the development of the software that controls their devices. As well as moving to digital media (e.g., DVD, MP3 players, digital television, digital cameras, etc.), consumers are demanding increasingly advanced functionality. CE vendors have turned to "digitalization" to provide this, with an increasing proportion of functionality being provided by software (as opposed to hardware). The result is that CE companies' role has shifted to being principally provision of software rather than hardware. Many CE companies are struggling to cope with this dramatic change in their business, and stand to gain enormously from the advantages brought by CBSE, particularly code reuse.

The CE space is demonstrating considerable convergence of devices. It is becoming difficult to buy a mobile phone without an inbuilt camera, or to buy a digital camera that can't be used as a generic portable storage device. The range of CE products available today is a diverse population, where each

differs only a little from its closest cousins.

More precisely, the CE market comprises a large array of products, where the software required to drive them is a multidimensional continuum. For example, consider televisions: from small analogue televisions, through more advanced models and integrated digital sets, to integrated systems including DVD or hard-disk playback and recording. There is perhaps little or no opportunity for shared software between opposite ends of the continuum (e.g., a low-budget portable television and a high-end digital plasma screen), but we can expect most of the software requirements to be shared between products that are adjacent on this continuum. The continuum is also becoming increasingly multidimensional: consider most DVD players' ability to play back audio CDs—a TV/DVD combo product joins the continuum of all TVs, DVD players and stereos. Today the entire CE product-space is covered by this single continuum. A "path" can be drawn between any two products consisting entirely of small steps.

Ten years ago it was practical to write from scratch the software for each CE product (or at least each product family). This has become increasingly less desirable and is beginning to become infeasible. Instead, CE manufacturers must find a way to share software across their product range.

Furthermore, it is becoming increasingly desirable that CE companies are able to share software between each other. Different manufacturers rewrite from scratch large amounts of software between themselves, regardless of how effectively individual manufacturers manage code reuse within their organization. Much of this software offers no product differentiation or competitive advantage to the CE manufacturers, and thus would be more effectively shared between them. (CE manufacturers have told us that up to 80% of their code is "non-differentiating"—i.e., code that is broadly the same for all manufacturers, and does not influence the end consumers' experience, or their choice of which product to buy.)

Note that whether such reused code is written by third-party software vendors, or by the CE companies themselves is of little consequence. The code still needs to be reused across a wide range of products in different settings, and we believe that components are the only practical way to achieve this. (For the record, we believe that the CE companies are best placed to provide this code since it is they who have the expertise and the existing code base.)

Of course, one way to reuse code is simply to ship source code. However, this is often a less convenient way to deliver (and to receive!) code than components. Indeed: a component model is precisely about providing a framework to ease the combining of independently developed pieces of code. Moreover, using CBSE and shipping source code is in no way mutually exclusive: there is nothing preventing delivery of components along with their associated source code, and component models such as Koala [29] even work

at the source-code level. (That said, customers have told us that they prefer to distribute code in binary form to protect their IPR (intellectual property rights) and to avoid certain legal liabilities.)

In our experience, not only do CE manufacturers have a lot to gain through CBSE, many are starting from a particularly low point in terms of the maturity of their software development processes. This is largely because until relatively recently, software has played a minor role in a CE products; within the last few years the opposite has become true. It should not be surprising therefore that many have struggled to adapt to this change. We contend that CBSE offers the only practical way for CE manufacturers to climb out of their current hole.

5.2 CE's need for a new component model

There are several component models currently available and in wide use, but none is well suited to CE software (or embedded software generally). The more well-known component technologies include CORBA, Java, COM and .NET. This section gives a very brief overview of each, and explains why they are not well suited to the CE domain.

5.2.1 CORBA

According to our definition of CBSE, CORBA is not actually a component model. Critically, CORBA objects cannot be independently deployed. CORBA's primary focus is in providing an object-based RPC mechanism, along with some standard interfaces. In other words, CORBA is about distributed object communication, not about componentization and reuse.

To address this, the latest version of CORBA (version 3), includes the CCM: CORBA Component Model [30]. However, this is targeted towards enterprise software and too heavyweight for embedded/CE applications. A "Lightweight CCM" standard has been proposed [31], but remains unratified by the OMG, and there are no implementations of it outside the research community.

5.2.2 Microsoft's COM and .NET

Microsoft has two proprietary component models—the older COM/DCOM/COM+, and its successor, .NET. .NET is similar to Java, with components compiled to an intermediary code and executed on a virtual machine[3]. .NET therefore suffers the problem that it is more difficult to write lower-level systems software due to the virtualization (for example, there is no support for placing inline assembly code in .NET; something that is trivial[4] with C). Even

3. At least, this is the model. In reality a JIT is used to translate the intermediate code into machine code. However, this is merely an optimization of the implementation: conceptually the .NET runtime provides a Virtual Machine used to execute .NET's intermediate code.

worse, .NET requires all existing software be either rewritten in C#, or extensively modified to be used with MC++.

COM components do not suffer these disadvantages of virtualization: COM components may contain native machine code, and can execute directly on the hardware. They can be written in existing languages, principally the C language[5]. As such, we consider COM better suited to lower-level CE software than is .NET. However, Microsoft has shifted its emphasis from COM to .NET; Microsoft and others already consider COM to be a "legacy technology".

5.2.3 Java

Conventionally Java is thought of as an object-oriented language, rather than a component model. However, according to our definitions, the JVM and Java object model constitutes a component model.

One of the most frequently cited criticisms of Java is that the JVMs often impose performance and footprint overheads that are too high for lower-end CE devices. It is indeed our experience that many CE products remain so resource-constrained that even the Java solutions that are optimized for use in embedded devices [6] impose runtime and footprint overheads that are prohibitive.

However, due partly to Moore's Law and partly to advances in Java technology, Java's performance is steadily becoming acceptable for more and more CE devices. Java is increasingly used in CE software, particularly at the application-level for higher-end devices (for example, the address-book functionality or downloadable game of a high-end mobile phone using a service such as Vodafone Live! [32] or NTT DoCoMo [33].

Java is less well suited to low-level programming than .NET[6] (and both are less well-suited than C). Java does have some support for lower-level programming, notably "native methods". However, it is notable that native methods are pieces of code that were too low-level to be written in Java. Also relevant here is that Sun (the owner of the Java standard) explicitly discourages use of native methods [34]. That is, a Java program is deliberately far removed from the underlying hardware: great for portability, but not so useful when writing code that requires direct access to the hardware!

Most significant of all however, is that the Java component model requires use of the Java language. This immediately renders it suitable for use with much of the industry's existing code base, most of which is written in C/C++ or

4. Albeit highly dependent on the compiler used.
5. In fact other, higher-level languages are used to write COM components too (such as Visual Basic and Delphi). However, such languages are better suited to desktop than to embedded software, and so outside the scope of this paper.
6. .NET's has support for arbitrary memory access through use of its unsafe modifier [36], has support for "managed C++ (MC++)".

assembler. Throwing away your existing investment and rewriting all your code in Java is a curious way to go about obtaining code reuse!

In summary, we see Java as an increasingly important technology in CE for the higher-level application software on higher-end devices. The rest of CE software, however, remains relatively poorly addressed by Java and other software component technologies.

5.2.4 Other component models

Obviously, the above list of component models is far from exhaustive. Models not covered include research systems such as Darwin [35] and proprietary component models for a certain company's internal use, such as Philips' Koala system [29]. However, these models do not meet the criteria listed above of "in wide use", or "available", and so do not justify further discussion here.

6 Conclusion

The best way to get high quality software is to not write it at all, but to reuse it. Software components are an excellent way to enable effective code reuse, both within and across organizations. In addition, software components bring many other advantages, including location transparent communication, language independence, improved architectures, and more.

Over the past decade, the desktop and enterprise software markets have seen a silent revolution, whereby component technologies and code reuse have become the norm. This has been enabled by component technologies such as Java, COM/DCOM/COM+, .NET, etc., and the advent of the World Wide Web—application developers today think nothing of searching the Internet for a Java class file to do what they need, before they write one themselves.

Also over the past decade, the amount of software present in CE devices has exploded. CE manufacturers have had to change almost overnight from hardware manufacturers who need to embed small amounts of software in their devices, to software houses that ship a generic computing platform with a little specialized hardware on which to run their applications. This sea change has been brought about through the move from analogue to digital devices, and the ever-increasing demands for features from customers. The wide range of CE devices may appear very different at first glance, but inside they all work the same way, and there is a huge amount of shared behavior across the range.

At the same time, software engineering tool providers, particularly component technology providers, have largely ignored the CE industry. Combined with the inexperience and relative immaturity of the CE industry with respect to software development, this means that despite the huge perceived scope for the sharing and reuse of code across these disparate CE devices, practically

very little is realized. Consequently, the CE industry has a real need for software component technologies that it can use in order to reuse their existing software, and new software that they will write in the future.

References

[1] D. L. Parnas. On the criteria to be used in decomposing systems into modules. Communications of the ACM, 5(12):1052-58, December 1972.

[2] Paul C. Clements. From subroutines to subsystems: Component-based software development. In Alan W. Brown, editor, Component-Based Software Engineering: Selected Papers from the Software Engineering Institute, pages 3-6. IEEE Computer Society Press, 1996.

[3] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java Language Specification Second Edition. Addison-Wesley, Boston, Mass., 2000.

[4] Sara Williams and Charlie Kindel. The component object model: A technical overview. http://msdn.microsoft.com/library/en-us/dncomg/html/msdn_comppr.asp, Oct 1994.

[5] Microsoft Corporation. .NET homepage. http://microsoft.com/net/.

[6] Esmertec. www.esmertec.com.

[7] Gatzka Th Geithner. The Kertasarie VM. http://www.netobjectdays.org/pdf/03/papers/node/285.pdf.

[8] Microsoft Corporation. .NET Compact Framework homepage. http://microsoft.com/netcf/.

[9] Clemens Szysperski. Component Software: Beyond Object-Oriented Programming. Addison Wesley, 2002.

[10] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. Communications of the ACM, 17(7):365-375, July 1974.

[11] Dr. Gregory V. Wilson. Extensible programming for the 21st century. http://www.thirdbit.com/~gvwilson/xmlprog.html.

[12] David Collopy. Introduction to C Programming, a Modular Approach. Prentice Hall, 2002.

[13] G. Booch. Object-Oriented Design with Applications. Benjamin Cummings, Redwood City, California, 1991.

[14] Frederic Brooks. The Mythical Man Month: Essays on Software Engineering, Anniversary Edition. Addison-Wesley, 1995.

[15] J. Haartsen, M. Naghshineh, J. Inouye, O. Joeressen, and W. Allen. Bluetooth: Vision, goals, and architecture, 1998.

[16] Bitfone corporation. WWW home page. http://www.bitfone.com.

[17] Butler Lampson. Protection. In Proceedings of the 5th Annual Princeton Conference on Information Sciences and Systems, pages 437-443, Princeton University, 1971.

[18] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, David Becker, Marc Fiuczynski, and Emin Gün Sirer. Protection is a software issue. In Proceedings of the 5th Workshop on Hot Topics in Operating Systems, pages 62-65, Orcas Island, WA, May 1995.

[19] George C. Necula. Proof-carrying code. In Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 106-119, Paris, France, January 1997.

[20] Brian N. Bershad, Craig Chambers, Susan J. Eggers, Chris Maeda, Dylan McNamee, Przemyslaw Pardyak, Stefan Savage, and Emin Gun Sirer. SPIN - an extensible microkernel for application-specific operating system services. In ACM SIGOPS European Workshop, pages 68-71, 1994.

[21] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. ACM SIGOPS Operating Systems Review, 27(5):203-216, December 1993.

[22] Steve McConnell. Code Complete. Microsoft Press, 1993.

[23] Roger Smith and Rosaclaire Baisinger. Component Management Tools. Software Development, September 1998.

[24] Maurizio Morisio, Michel Ezran, and Colin Tully. Success and failure factors in software reuse. IEEE transactions on computer systems, 28(4), April 2002.

[25] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. Computing Systems, 8(3):221-254, Summer 1995.

[26] John L Hennessey and David A Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann, 2002.

[27] P. Rutter. Uniform Handling of Exceptions in a Stack-Based Language. ACM SIGPLAN Notices, 12(9):71-76, September 1977.

[28] Mike Gancarz. The UNIX Philosophy. Digital Press, 1995.

[29] R. van Ommering, F. van der Linden, and J. Kramer. The Koala component model for consumer electronics software. IEEE Computer, 33(3), March 2000.

[30] The Object Management Group. CORBA Components. http://www.omg.org/docs/formal/02-06-65.pdf, June 2002.

[31] The Object Management Group. Light Weight CORBA Component Model (revised submission). http://www.omg.org/docs/realtime/03-05-05.pdf, 2004.

[32] Vodafone Live! http://vodafone.com/live.

[33] NTT DoCoMo. http://nttdocomo.com.

[34] Sun Microsystems. 100% Pure Java Cookbook -- Guidelines for achieving the 100% Pure Java Standard. http://java.sun.com/products/archive/100percent/4.1.1/100PercentPureJavaCookbook-4_1_1.pdf.

[35] A. Garg, M. Critchlow, P. Chen, C. van der Westhuizen, and A. van der Hoek. An environment for managing evolving product line architectures. In International Conference on Software Maintenance, pages 358-367, September 2003.

[36] Microsoft Corporation. C# programmer's reference: unsafe. http://msdn.microsoft.com/library/en-us/csref/html/vclrfUnsafe.asp