The Long Awaited Promise of Component Reuse is About to be Achieved

Ivar Jacobson and Martin L Griss

The technology and standards are now largely in place to support the long-promised "golden age" of component-based development. The time has come to work together to take advantage of this situation. With more and more of the world's activities being carried on by software, we can no longer afford to build all the critical applications totally from scratch. We must systematically reuse components!

In our opinion, economically significant reuse has three prerequisites:

1. A widely understandable modeling language to do for software what a standard format for engineering blueprints has done for the construction and mechanical disciplines.
2. A process for developing software, again drawn from the example of the earlier engineering disciplines.
3. Software tools to support the entire development process, covering both the detailed lifecycle activities and key cross-lifecycle activities.

The work products of software engineering will be described by this blueprinting language; they will be developed with this standard process; they will be transformed and managed by these tools. Building on the structure provided by language, process, and tools, it is technically possible to make great strides toward component reuse.

Learning from older industries

"If we want to become as industrial in software as we are in the older industries," we have been saying for years, "we need to apply techniques similar to those of the other disciplines, but--and it is a big "but"--we must adapt them to our specific problem space."[1]

The other engineering disciplines have held out for us the vision of where the software industry should go. As inspiration, we first used the hardware industry. The vision was one of software components and interfaces comparable to the pluggable boards and boxes connected with busses and cables of the hardware domain. Later we drew examples from the construction industry, manufacturing industries, and chemical engineering.

[1] This vision is very different from what the proponents of programmer empowered methods suggest. They are like the Luddites who worked against the industrial revolution, who felt that the rise of industry would de-humanize and exploit workers. They use a very similar language, talking about the "rights of the programmer" and the social aspects of programming. This community does not want software to become "industrialized".

In each of these industries, engineers and architects know that they can reduce time and improve quality by employing standard components and proven best practices, often codified as processes. Early in their careers engineers are trained in the selection of components and their assembly into larger systems, using standard processes. They learn to use catalogs, handbooks, and standard notations. They do not start each new project from scratch.

That has been the metaphor that shaped our joint thinking. It has guided us in developing blueprinting languages and development processes. In the last few years the Object Management Group has standardized the languages as the Unified Modeling Language, and the Rational Unified Process has emerged as a de-facto standard for developing component-based systems. This language, this process, and the tools that support the language and process are making reuse an industrial-strength best practice.

Today we see substantial reuse of complete commercial applications and other packages (such as DBMSs, communications infrastructures, operating systems, accounting packages, and GUI frameworks). They often use a process of "best fit" - they ask "which commercial packages can I use as the base for this system that gets me as close to my requirements as possible?". They then try to change their application requirements to be closer to what these commercial packages offer.

We also see some amount of *ad hoc* reuse practiced by individuals and organizations[i]. Many development teams regularly create their own libraries of code, share designs, re-apply templates and frameworks, etc. But this tends to be uncontrolled, inspired by individuals, and rather fragile.

Neither of these current reuse practices has significantly transformed the software industry. What we envision and advocate is a systematic engineering practice, a controlled, planned and architected reuse program that is based on large-scale managed components. Quality and productivity due to reuse should be controlled, measurable, nor managed.

The three prerequisites are finally here

Achieving the long-promised benefits of large-scale component-based development has been hampered because the three essentials that support it were missing until very recently. Their absence made it almost impossible to systematically create and find software pieces suitable for reuse. Because there was no effective, standard way of describing the pieces, it was hard for potential users to find what was available, and to decide if the piece was suitable.

1. We didn't have a standard way of communicating about software.[ii] We have now: the Unified Modeling Language, UML.[iii]

To get reuse, we need to understand what we will reuse. We need to understand the critical concepts, structures and design that underlay effective engineering. Up until a few years ago we had no standard for communicating what a component could do. The UML is the blueprinting language that has been adopted by the whole software industry. We travel all around the world and we dare say that nearly every university teaching software engineering (or computer science) teaches UML. It is widely used in requirements engineering, architecture, analysis, design and implementation courses. It is used to define patterns, processes, protocols, workflows and even organizations. It is used to specify systems of all sizes.

Thus, the UML provides the means to communicate between people, both those doing the same kind of work (analysts with analysts) and those in different roles (analysts with designers). It provides communication not only between people working within the same project, but also people in different projects. Its communication capabilities extend to people working in different parts of the world and even in different companies. UML is used for all the kinds of diagrams it takes to develop software. UML even extends upstream from software development per se to model the business processes that developers are to implement in software.

Simply put, the function of this modeling language is to describe the many different views of components and complete applications at various stages of development. Different constructs are used in different models. For example, use cases in the analysis model or classes in the design model. Each of these models evolves as the system is refined. Some of the models can be used to describe what a component does, while others describe how it does it, and how it interacts with other components in doing it. In addition to using some form of component catalog to search for components, these models allow a designer to understand what a reusable component will do without looking at the design or implementation. The language is sufficiently precise so that he should not need to examine the source code to understand exactly what the component might be able to do.

This blueprinting language makes it possible for us to unambiguously capture, share and standardize critical domain-specific and infrastructure concepts and design knowledge. It is the communication and understanding of this knowledge that makes possible the effective engineering of the complex systems we must build.

2. We didn't have an effective and standard process for developing software. We do now, RUP.

For these past 30 years the software industry has employed a great variety of technologies (methods, languages and tools) to develop software systems. This variety extends from the waterfall process model of 1970 to the computer-aided software engineering tools of the 1980s, to the more recent design with objects and components. In the past few years, fortunately, we have made further progress.

How do we get a whole team to develop software together? That is what we mean by *process*. They all need to know what to do, when to do it, and how to do it. The process helps by telling what artifacts need to be produced, by whom and what activities need to be carried out at each step. By helping you to know what you should think of, and when you need to, it frees you to focus on developing the right solution.

Thus given a useful set of high-quality components and the UML to describe them precisely, the next step was a well-defined development process to create and reuse components. With contributions from thought-leaders all over the world, we agreed on a process to develop software systematically. That way became institutionalized as the Rational Unified Process (RUP). [iv] Now adopted as a standard by thousands of companies, it answers the need for a systematic software development process. [v]
RUP can be applied both to developing complete applications and to developing reusable components. RUP satisfies this need because it enables people to work together effectively, making software development into a kind of "team sport."

Because the needs of software development are as broad as the world itself, the process was designed to be specializable and scalable in many ways, based on skills, role, application areas, and organizational constraints. For simpler developments in smaller organizations, a light-weight subset of RUP is used—RUP can be made as light-weight as you wish. As the scale of the work and the organization grows, RUP provides the basis for all the processes followed by the whole organization. RUP provides an extensible framework in which we can easily incorporate key aspects of the flexible community oriented processes that underlie OpenSource and pair programming aspects of Extreme Programming.

However, RUP alone does not cover all of the processes needed to fully support reuse. As we showed in our reuse book, reuse requires a suite of distinct processes targeted to architecting families of systems, developing components, and putting together application systems[i]. We build on the RUP foundation to produce a scalable family of component reuse-oriented processes that cover system architecture, component construction, and reuse of components to build other systems. Likewise, RUP by itself does not automatically supply all of the guidelines needed to support the specific design patterns, scalability and robustness needs of, for example, a specific class of e-commerce systems based on Microsoft .NET or J2EE - but these can easily be provided as domain-specific checklists and guidelines in the RUP style.

3. We didn't have the tools to support the entire software development lifecycle. Now we do, with tools for UML and RUP, and all cross-lifecycle activities

"A process not tightly supported by an integrated set of tools is just an academic idea," is what we use to say. Fortunately, both UML and RUP are supported by suites of tools. These tools support many aspects of both the specific software lifecycle stages like requirements, analysis, design, implementation, test and also the key cross lifecycle activities like change management, version control, configuration management, defect tracking, documentation, project management, and process automation. Tools like

Requisite Pro, Rose, SODA, ClearQuest, TestManager, etc. are well integrated, customizable to different users of RUP and UML. They support all the important artifacts.

Towards a well understood engineering discipline and industry

In a nutshell, we want to get to a software industry with many active players, each proficient in some aspect of the development, distribution, or use of reusable components. No longer can lone individuals invent and develop by themselves from scratch all the pieces of software that a large-scale software system comprises.

There are useful parallels between the construction industry and software development. Building architects and engineers (such as structural and lighting engineers) work with diagrams and computer models to produce detailed design blueprints and specifications. Later masons, carpenters, electricians and plumbers "install" these designs on site by assembling prefabricated physical components or building pieces of a building from scratch.

Likewise in software development we have a variety of analogous skills and specialties. Software architects, analysts and designers produce detailed designs in the form of a set of artifacts (specifications, models, etc.) utilizing UML. Later, other developers such as programmers skilled in different languages, database experts, GUI specialists, and testers complete the implementation. The architects, testers, etc. make use of large catalogs of components.

Reusable components are large-scale

While class libraries and other forms of small scale reuse have been quite useful, they have not transformed the software industry in the way we envision.
Components must be worth the effort of being sold, bought and maintained.
We will call the large-scale reusable assets *components* in the broad sense of "substantial chunks of functionality designed for reuse." [2] The reusable assets we are talking about are at the size of subsystems or frameworks.

Components have significant measurable value

These big *components* are represented by a standard set of artifacts, to use the terminology of RUP and our reuse work. For instance, these components have a set of models: use case, analysis, design, implementation, and test. In practice, one of these components may support several use cases. It may include many subsystems, each with a number of classes and their implementation in terms of source code and executable code. These components have a strict configuration and version-control system in place. With each such component there follow ready-made software models that can be incorporated

[2] In this sense *component* is a much wider term than the component construct employed in the UML. There it just refers to a software element that may be quite small.

in tools supporting the UML and RUP. With each component also follow guidelines for its reuse.

These components are used to build applications. Reusing a component means that each of its different models will be reused in the course of building the application. Thus, when you make the use cases for the application, it reuses the use cases of the component; when you move to the design of the application, you reuse the design model of the component, and therefore its code, its tests, and so forth. Such a component accelerates the application development for two reasons: lower costs and faster time to market. Both are achieved by starting not from scratch but from quality components.

A software component fits an architecture setting

A software component is not usually as easy to use or connect to other software components as building construction components are. Some construction components, such as doors, windows and cabinets can just be connected to other things by means of a simple seam, such as a line of nails or screws. Other construction components are more complex and have to be consistent with many fine-grain and coarse-grain architectural constraints and standards. For example, electrical components often have to distinguish 220 vs. 110 voltage, 50hz vs. 60hz, English vs. metric measurements, etc. Adherence to such standards and use of compatible prefabricated components (such as doors, cabinets, windows, plumbing fixtures, roofing trusses, and wall panels) has greatly improved the efficiency of the construction industry.

Software is still more complex and moreover, less tangible. Therefore, strong and effective architectural standards are absolutely essential if we are to have a viable software components industry. While some small software systems are simpler than some buildings, most large software systems are much more complex than most large buildings.[3] Large software systems have many components that must fit into the often very complex setting defined by the underlying architecture: system software (operating system, database management system, the programming environment). Components must interface with other components through many complex interaction points.

This complexity makes it hard to produce software components that are usable in several architectures. We must usually be content to have a component capable of playing a role only in the architecture for which it was designed. For example, a component in the middleware layer is dependent on the system software it runs on top of, as well as the other components it works with in the same layer. You cannot easily move a component to work in another architectural setting, such as moving it from Microsoft.NET to J2EE. (Here you may have to provide two configurations of the component.) You cannot easily enable a component to work with components it was not designed to interface with. (Here

[3] One way to measure this added complexity is to compare the number of developers that are needed to design (architect, engineer – not actually build) the building and the number needed to develop (analyze, design, implement, and test) the software.

you may have to put a wrapper around it or a wrapper around the environment to get the interaction to work out.)

A component in a higher layer is dependent on the middleware and the system software layers. It will not immediately work on top of other such layers from a different architecture. (We may try to insulate the higher layer from the underlying layers by using a layering wrapper. This wrapper, on top of the lower layers, makes the top layers much more independent of the underlying platform.)

Software components are understandable

Developers, that is people, create the source code for the components and their interfaces. That source code will be translated mechanically into executing code intended for interpretation by a machine. Of course, we can reuse this code by shipping it to many places, installing it, and letting it execute. However, developers trying to reuse a component need a lot more information than they will ever get from executing code or even from source code. They will need models and documentation that describe what a component does, how to use it, how it interacts with other components, and what architectural, standards and infrastructure constraints it must abide by. Furthermore, they may need guidance on how to specialize or adapt the component if they will not use it "as is". Thus, software components must be understandable.

Should the component documentation and released models include source code? Many software components are not, and will not be, sold with complete UML models and source code, just as detailed chip layout diagrams are not always included with hardware components. Often all the details used to implement and maintain the component are not needed to understand how to use it or do the most important customizations. So, just like a chip, in many cases binary code and a partial UML model with some textual elaboration is the practice for most components sold today.

We do not want to confuse the discussion with too a simplistic presentation regarding our position that complete source code should not be part of the component release. There are several deep philosophic implications of this approach to large-scale component reuse. By not providing all (or any) source code we appear to stand in opposition to the popular OpenSource movement. Many powerful and widely used systems such as Linux, Apache and others have been developed to high quality by this approach of a large, open community of developers. Perhaps as the economics and processes of OpenSource get better understood, some form of OpenSource will be a key part of developing some components. But certainly only a small number of commercial software components can or will be developed this way. See also our later discussion about unauthorized change.

In other cases, to make the "how to reuse this component" clearer, UML models are focused at describing the façade of the component. The façade is a model element that specifies the information application builders need to reuse a component. In some cases, some of the source code may be needed. For example, when reusers expect to do

significant modification, their own maintenance, or possible recompilation of a component, more will be needed, including a lot of source code and a lot of the models (use cases to test models).

This reliance on UML will happen because

- more people will be trained to read UML,
- UML will be extended with 'profiles' to more effectively address specific problem areas,
- more components will be sold with effective UML descriptions, and
- new customization mechanisms will make it easier to extend code without modifying or recompiling the source code.

We expect that developers will come to understand components modeled in UML.

However, understanding UML and developing with RUP are not enough. We need more to create this industry. That *more* is a standard that enables people to understand how to become producers and consumers of reusable components through a global marketplace.

As we will describe in more detail below concerning the creation of a standard Reusable Asset Specification (RAS), we need consistent component specification standards if we want a much wider-spread component reuse market. Ad hoc component specifications will work so long as components are produced and reused within a single company or small group of companies. But as soon as we want to scale up to a broadly distributed market, much stronger standards are needed. It will be very hard (if not impossible) to effectively reuse components produced by different companies if they are modeled, specified and documented poorly and inconsistently.

The players in the new industry

One of the requirements for a robust and efficient market is that there be many participants with a diversity of viewpoints and objectives. To understand this emerging market, let us focus on the four most important players: the component factories, the application builders, the tool makers, and the component brokers.

Component factories

Component factories will be the vendors that have developed components. Usually a component will be targeted to a particular vertical niche, such as insurance components or banking components. However, there will be components at all layers including middleware and system software. With the prominence of the Internet, components will be built specifically to accommodate the requirements of e-business.

Many of the companies that will become the component factories of the future have today developed components, or something that can be turned into components, for their own business needs. Of course, each of these companies has had to develop its own techniques, such as its own proprietary reuse-oriented processes, modeling languages,

component specifications, frameworks and tools. These unique technologies make it very hard for anyone else to reuse the components; some will have to be replaced by widely used standards. In many cases, these vendors have been depending on these differences as barriers to competition. They will have to change some of their business models and approaches to competition in a world of more open standards,

The component factories will come from five areas:

- *Service organizations* like TCS, Anderson/Accenture, or EDS
- *IT departments* in the larger companies--insurance, banking or other specialized fields in particular.
- *Package vendors* like SAP. They will have significant challenges in standardizing their proprietary solutions, adapting to new component standards, or getting their established customers to change technology.
- *Infrastructure vendors,* the developers of operating systems, networking and middleware products, such as SUN Microsystems, Microsoft, Hewlett Packard, and BEA Systems .
- *Current and emerging vendors* of smaller sets of components, like class libraries (e.g., RogueWave) and brokers such as ComponentSource, Component Planet or Flashline.

The potential components that some of these companies have today cannot simply be factored out and sold. They are usually integrated within complete solutions from each of those companies and rely on a mixture of proprietary and standard technologies and ad hoc specifications. The organization that developed the component usually is the only one that can reuse it. They will have to do substantial re-packaging, and some significant reengineering, before these components are ready for the open market. They will also need to change some of their business models. For example, service companies must turn the component or customizable solution part of their business into a component product business.

Given the state of today's marketplace, a potential component vendor has few effective ways to let others know what it has to sell. Then, even if it finds a purchaser, it has no agreed-upon methodology with which to instruct the buyer in the employment of the component. They are left on their own. This is the infrastructure that the component factories of tomorrow will need to provide or help build for the industry.

In this future, component factories will take on the role of a product company, delivering reusable assets to a world market. The difficulties they will face fall into several categories. They all need to move to a standard component specification and development process, such as UML, RUP and RAS provide. Also, they will need to become compatible with one of the major component architecture standards (such as COM, .NET, CORBA, or J2EE) The package vendors need to transition from their home-brew technology into these standards.

Application builders

Application builders will build applications or end-user products – that is, what we usually call systems. They will do it by selecting and customizing components from vendors, adding their own components and additional software to integrate these components into complete solutions. Thus they will add application specific functions on top of more generic reusable components to form systems for their clients.

Several kinds of companies like this exist today, but their roles will become more specific as a real component market develops. In some cases they build applications for sale, in other cases they build applications for use by their own or customer organizations. They include:

- *Solution builders or system integrators,* such as EDS, CSC, which provide software to others.

- *e-Business service companies,* that build by reusing packages and dynamic e-services.

- *Large product-line builders,* that build a suite of applications (a product-line) around a common product-line architecture. For example, telecommunication systems, printers or instruments.

Companies that are the potential application builders of the future are solutions-centric. They have a vertical organization--their customer relations or marketing organization is familiar with the business needs of the end users. We foresee a future in which companies of this kind take a stand and decide on their direction. Either they will become competitive application builders or they will become component factories.

Of course, a few companies will attempt to play both roles, though, in our opinion, it will be hard to do both. Application builders and component factories have many differences which can best be summarized as: application builders are solutions oriented and their customers are other businesses, while component factories are technology oriented and their customers are other software companies.

Tool makers

Tools will enable component factories to build reusable components more rapidly to higher quality standards. They will enable application builders to get systems into use more swiftly. Without powerful tools, developers would find it very hard to build large-scale components, adaptable to reuse. Similarly, application builders would find it equally hard to incorporate these reusable components into new systems.

For the interface between the component factories and the application builders, the tools needed boil down to two principal functions: importing a set of models to the application builder and allowing him/her to build on those models. The tools for importing should prevent the reusers (the developers from the application builder) from actually *changing* what is reused, but just *specializing* what is reused. This is why we think that with rich UML models and effective descriptions the reusers should normally not need source

code. The intent of this limitation is to lodge *change control* of the reusable component firmly in the component factory. (Refer to later section describing the component builders' responsibility for *change control, and the discussion about OpenSource philosophy*).

Also, since the application builder may reuse many underlying components from different component factories, the tools must permit him/her to build on top of several underlying sets of models.

Component brokers

Given that we now have the technology to develop components and to use them, we need to make sure that component factories and applications builders can find one another. This is a much simpler problem today than it was in the past. In the past getting together was hard. We didn't know what a component should look like. We didn't have standards like UML and RUP. We didn't have supporting tools. These lacks alone prevented progress. As if they were not enough--

- We didn't have the Web and search engines,
- We didn't have standard architectures and frameworks,
- We didn't have XML to transfer components and interact with them.

So, until this point, the business of brokering components has not been very interesting or feasible.

Now it is. There is a growing understanding of the economics of make-vs.-buy, both among developers and among managers. It is worth advertising and marketing components. It is worth creating a marketplace for components. The operators of these component marketplaces will be the component brokers.

Standards underwrite the new industry

In the marketplaces of antiquity, the products were inconsistent from one unit to the next and their sizes were haphazard. Merchants had to weigh the gold or silver pieces offered in exchange to avoid being "cheated." Even the weights of the items they bought and sold were not enforced by some external authority superior to the players. These inconsistencies forced market players into time-consuming one-on-one bargaining and bartering.

In other words, nothing about these old marketplaces was *standardized*. That was inefficient! Now we have the experience of a long practice of marketplaces with standard sizes, defined measurement units, and fixed prices. Our task is to transfer this experience to the component marketplace.

In a similar way, we need a standard for how some software vendors can create and sell reusable components and how others can buy and reuse those components. The component factories and the application builders must know exactly how to play the game. Standards will tell them how.

Think about a modern game, such as soccer. It has a rulebook that tells you how to play. It has what we might call a set of tools--a ball, a type of shoe, goal posts, even a marked off field. It has a referee to help interpret and enforce those rules.

Armed with these rules and tools, teams from the far ends of the Earth can meet and play successfully. Without these "standard" rulebooks strange teams could meet in a meadow, but they would be reduced to kicking at tufts of grass. To play a game, you must have standard rules, tools (football, goals, field, boots) and institutions (referees and associations).

The same prescription applies to the new game we are creating--the marketplace for software components. A key goal of this marketplace is to establish standards for quality and performance, as well as for functionality and interoperability. It is important to both sellers and buyers to ensure quality, to reduce costs, and to increase trust in the components.

- We must have a rulebook, the standard for what it means to play the game. It will be non-proprietary, of course. It will specify, for instance, what kinds of models need to be created for each component. It will explain how to be sure that a component has been appropriately tested or certified to meet its specification.
- We must have tools to play this new game. Tool vendors will design and produce the tools for building the models necessary to follow the standard.
- We must have associations that establish and enhance the standards and referees that validate their use. Some companies will set their own standards. They will have their own acceptance testing and certification laboratories for their application builders. Others will participate in open standards organizations and use 3rd party testers. Some may depend on the supplier to do a good job and market forces to weed out "bad" components.

The standard will specify, as, in fact, RUP does now, what models component factories deliver for each component. It will specify the details about these models, including their contents and how to describe variation points. RUP and RAS already specify what should be modeled and UML defines the legal syntax and semantics of the things being modeled.

In order for application builders to reuse components effectively, they must be well-specified, well-documented, and easy to apply. By *well-specified*, we mean that the set of UML artifacts provided makes clear what the component does and how it interacts with other components. In particular, its variability and its interfaces must be carefully spelled out.

By well-documented, we mean there must be artifacts in addition to the UML diagrams to provide the rationale for features, classification, testing status, relationship with other components, who to call for support, evolution plans, and so on.

By easy to reuse, we mean the set of artifacts must be amenable to direct importation into the models being built for the new application. Direct importation supports quick modeling. It is part of the automation of the development process.

Creating the marketplace standards

The first step in creating the appropriate standards for the set of artifacts (models and documents) is already being taken today. An initiative is underway to create an initial Reusable Asset Specification (RAS) and a consortium to evolve and use this specification. This initiative takes a broad perspective on which UML and textual artifacts should always be provided and which are optional. The specification will show how each artifact provided with a component should be structured. Furthermore, it will specify how the set of artifacts should be related to each other. The RAS consortium has a goal of creating industry consensus and widespread use. RAS is part of Rational's "e-Development Accelerators" initiative, combining standards, reusable assets and automation. (See http://www.rational.com/eda/index.jsp). RAS is based on the concepts defined in the UML, RUP and specific architectural views. (See eWeek, 9/11/00 http://www.zdnet.com/eweek/stories/general/0,11011,2624688,00.html)

As RAS gains acceptance and as architected large-scale component reuse becomes more widespread, the next step will be to focus on the reuse process - how an application is best built from reusable components. This will determine which artifacts are needed, and in what order. By also standardizing these key reuse-process steps and their related UML artifacts and associated documents, we ensure that component builders provide the right set of artifacts. We ensure clear relationships between them. We enable application builders to use each in a well-defined way at an appropriate step in the reuser-process.

We believe that the current RAS specification, and the later refinements that come from the reuser-process-based view will address the evolving needs of component reuse. We expect that the reuser-process-based approach will ultimately lead to a more coherent set of standard artifacts and, hence, to better results. .

Component factories bear prime responsibility for *change*

The component factory that delivers a component must take long-term responsibility for it. That factory should maintain and further develop the component in accordance with the terms of a strict release plan.

For its part the organization reusing a component should take a conscious decision--and promulgate it to its troops--not to change a component. In part, the reason for not providing all the source code, or all the internal design models, is to prevent reusers from changing the source code.

The possibility of unauthorized change is, in fact one of the key challenges facing the OpenSource movement. While the appeal of a "thousand eyes" to catch all defects is compelling, the ability of almost anyone to make changes at any time makes it hard to be sure one has a stable, "guaranteed" version for reuse in other systems. The responsibility for making changes within such a component really belongs to the component factory producing it, and not arbitrary reusers.

Of course, there will always be component factories that go out of business. Ideally, support for its components will be assumed by another component factory. Inevitably, there will be instances where a reuser has to assume responsibility for an "orphan" component. Alternatively, some reuser organizations require source code to be placed in escrow to insure against such problems.

To carry out this policy, tool builders need to devise ways to assure that developers in the reusing organization cannot easily make changes within the component. In the normal course of development, they can only specialize what the component maker makes available to them, such as parameters, tables, or other variation points. With good specialization techniques such as those provided with UML, a reuser can go far in making a component useful for his or her needs.

The point is: if any user can readily change the content of a component, the component maker cannot effectively discharge the responsibility it properly assumes for the future evolution of the component. That future evolution, of course, is kept under control by configuration control management, enabling all concerned to know what they are actually dealing with.

The standard will specify the rules for actual reuse. For instance, we think it is very important that the component factory that delivers a component take long-term responsibility for it. That factory should maintain and further develop the component in accordance with the terms of a strict release plan.

Let our imagination roam the future

Thus far in the software half-century, we have reached some reuse on a small scale. Some companies have reached a larger scale, achieving reuse over a line of related products. A few have reached a still larger scale, reuse over different lines. We are now ready to take the next step. That is to create a component market that will go still further.

In this marketplace there will be different kinds of participants:

- Vendors developing reusable assets,
- System builders using these assets,
- Tool vendors,
- Component brokers,
- Standards organizations.

With the agreement on a standard set of component models and widely available processes and tools to construct and reuse components, the whole world can build teams. It can let these teams compete with one another. They compete, not only pair-wise, but all at once. There is no rest in such a marketplace. But in such a turmoil the goals of the marketplace as a whole advance.

The newly emerging "e-services" trend promises to lead to a new generation of large-grain "components" and expand the market opportunities. Many companies are participating in the standardizing of several XML- and Web-based technologies for the next generation of flexible e-commerce applications. The emerging Internet (HTTP) and XML standards for e-commerce set by the W3C, CommerceOne, UDDI, etc. are beginning to stabilize enough to enable a vast array of "Chapter 1" Internet based e-commerce systems.

New standards and web-based infrastructure established by Microsoft .NET (which includes SOAP and BizTalk), CommerceOne, Ariba, eCo, RosettaNet, IBM, HP Bluestone and NetAction, OBI etc., have potential to expand what we mean by component. These next generation of "components" will not be statically composed traditional components. Instead, they will be web-based e-service components, with well-defined interfaces described in XML, that can be dynamically found using registries and "brokers" and dynamically composed into complete applications using workflow-like glue languages. With the advent of these new standards, internet middleware and e-commerce frameworks, we are on the verge of a much more dynamic "Chapter 2" Internet e-commerce world.

This marketplace will mean business at a level of tens (hundreds?) of billions of dollars worldwide within a decade. It is a means of reducing the cost of building software by a factor of 10. This cost reduction need never result in the unemployment of software people. Instead, as people come free, they can use their power to develop new kinds of software and new kinds of technologies that would not be possible otherwise. And we will see things that we could not even dream of in the past.

SIDEBAR

In this sidebar, we collect and define several key terms used in the main text

- Component - A large-scale reusable asset, in the broadest sense of "substantial chunks of functionality designed for reuse.", such as subsystems or frameworks.
- Façade - A model element that specifies the information application builders need to reuse a component. This element is a packaging of several public UML elements available for imported reuse. The façade effectively hides the inner details of a modeled system. A component can have multiple facades, each presenting a useful and coherent subset of public UML elements available to support some reusable functionality provided by the component.

- Rational Unified Process (RUP) - A widely used defacto standard development process (framework) that defines artifacts, roles and process steps to be followed in architected, incremental-interative development
- Reusable Asset Specification (RAS) - A proposed specification of the needed artifacts and their format under development by a consortium of vendors, sponsored by Rational.
- Unified Modeling Language (UML) - A widely used software blueprinting language standardardized under the auspices of the OMG.

---

[i] Ivar Jacobson, Martin Griss, Patrik Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success,* Addison-Wesley, Reading, MA, 1997.
[ii] Grady Booch, James Rumbaugh, and Ivar Jacobson, *The Unified Modeling Language User Guide,* Addison-Wesley, Reading, MA, 1998.
[iii] James Rumbaugh, Ivar Jacobson, and Grady Booch, *The Unified Modeling Language Reference Manual,* Addison-Wesley, Reading, MA, 1998.
[iv] Ivar Jacobson, Grady Booch, and James Rumbaugh, *The Unified Software Development Process,* Addison-Wesley, Reading, MA, 1998.
[v] Philippe Kruchten, *The Rational Unified Process – An Introduction,* Addison-Wesley, Reading, MA, 1998.