THE ANNALS OF "DUNĂREA DE JOS" UNIVERSITY OF GALAŢI
FASCICLE I - 2006, Economics and Applied Informatics, Year XII, ISSN 1584-0409

ARE COMPONENTS THE FUTURE OF WEB–APPLICATION DEVELOPMENT?

Adrian LUPAŞC
Ioana LUPAŞC

University "Dunărea de Jos", Galaţi
alupasc@ugal.ro,
ioanalupasc22@yahoo.com

The software industry is still creating much of its product in a "monolithic" fashion. The products may be more modular and configurable than they used to be, but most projects cannot be said to be truly component based. Even some projects being built with component-enabled technologies are not taking full advantage of the component model. It is quite possible to misuse component capabilities and as a result, to forfeit many of their benefits. Many organizations are becoming aware of the advantages and are getting their developers trained in the new technologies and the proper way to use them. It takes time for an organization to adopt such a significant change in their current practices. Some of the trade magazines would have us believe that the industry is years ahead of where it truly is – those of us in the trenches know that the reaction time is a little longer in the real world. The change to component-based development has begun, however.

Keywords*: component-based development, frameworks, language, market, technology.*

1. Introduction

In this paper I discuss the why of components and frameworks and the rationale behind their use. I also talk about the current state of *Web*–application development and where the industry perceives it is going to establish a firm foundation and justification for the use and development of components and frameworks. Also, I look briefly at *Java*'s suitability for component-based development and for the development of application frameworks, as well as for the specialized features of the extended Java platform and associated *API*'s that make them ideal to this task, including JavaBeans, Enterprise JavaBeans and Reflection.

2. The Market

We are living in an era of unprecedented change in the software industry. From the great boom in the late 1990s to the slump in today, two factors that have remained constant in the software industry are change and growth. Even as some software companies fall on hard times, others prosper – the difference is usually in the way they perceive the software marketplace. Software is no longer entirely an art form or a process of creating one unique masterpiece at a time. It has evolved into more of an engineering discipline, one driven by the real–world economics of what works and what does not. Simply stated, components and component-based development are one of the things that work.

The signals from the software market are clear: products that exhibit poor quality, inflexibility, and significant schedule overruns are increasingly being rejected in favor of the new breed of component–based systems. As the infrastructure to support components becomes more mature and standardized, and as cross–architecture integration tools such as *SOAP* (Simple Object Access Protocol) become widely available, the component marketplace will likely grow even more rapidly.

This paper was recommended for publication by Assoc. Prof. Vasile MAZILESCU, PhD

Components provide a known quantity, a building block of established functionality and quality that can be used to assemble applications in a way more akin to assembly of a television than traditional development of an application. This is not to say that the assembly is necessarily simple, by any means. After all, can you put together a television from a collection of parts?

Component–based development is still an exacting process, but it is a more rapid, reliable, and predictable process. It is easier to say how long it will take, what will be needed, and how the finished product will work – all very desirable attributes in any engineering discipline, including software engineering.

The history of software engineering has been, by and large, the history of a battle against ever-increasing complexity. Object–oriented design and development was one of the big guns in this battle, and components and frameworks, correctly applied, are the biggest yet. Components provide one of the most potent tools to overcome this crisis, addressing the underlying concerns of productivity, reuse, reliability, and quality.

3. Why Projects Fail

The reasons for software project failures are varied, but they often fall into one of a few categories:

⇨ *Schedule:* One of the key reasons for project failure is the inability to achieve scheduled milestones. Companies often simply cannot continue spending time on a project. Sometimes it is a matter of unrealistic goals having been imposed on a development team – this is not a software problem, this is a management problem. No manager should ever set a schedule without consulting in detail with the people who will actually make it happen – the developers themselves – but they do. Because component–based development timeframes are much more predictable, the developer can give better defined estimates to management, and management can rely on them with more confidence. Even tasks that are common to component–based and non-component–based developments, such as capturing user requirements, analysis, and design, can benefit from pre-built structures in which such capture and analysis can be made.

⇨ *Specification changes during development:* another common reason for failure is that a project's goals change so radically while it is under development that the current project can no longer be adapted to serve the newly defined purpose. The greater flexibility of component–based systems can help avoid this in the first place, and the shorter development time gives less opportunity for significant "spec creep" to occur. When specification changes occur during a project, despite the faster development, the fact that components are usually, by their nature, more configurable and flexible than custom–created system elements gives a further advantage: they can simply be reorganized and reconfigured in many instances to adapt to the change in specification. If the change is substantial, then new components to provide the additional functionality can be added to the existing set more easily than in traditional development.

⇨ *Project management failure:* Perhaps the most common reason of all is a failure in the project management. It may be that the specification did not change, but that the project team understands of it was never complete, and they did not have access to the customer for clarification. Also, it may be that the schedule was being adhered to by the project team, but the customer was under a different impression as to what that schedule was. These are not technical problems; they are again management issues – primarily communication issues, in fact. A closer connection between the project leadership and management and the development team, of course, is the first step toward avoiding such problems. The ability of component–based development to shorten project life cycles, to make elements of the project more predictable, and to provide prototypes very early in the process helps avoid the communication gaps that result from management problems. Component–based development cannot do anything to solve bad project management, of course, and component–based projects can fail through bad management just like any other project. However, the component–based process encourages good practices that facilitate management of the project, making

component–based projects a little easier to handle than projects that do not use the technique.

In the software market, fear of failure of a project is very real and well justified. An embarrassingly large percentage of major software projects fail completely or fail to meet their overall goals, schedule, and budget. Information technology (*IT*) spending is no longer driven by the technology, if it ever was; it is now almost exclusively driven by the need to fulfill business requirements. Companies are seeing component–based technologies as the most cost–effective way to meet these requirements, with the lowest risk of failure. Where the market demand goes, development follows. Projects happening on "Internet time" simply are not allowed the luxury of time to develop entirely new architectures and low–level capabilities from scratch – the schedules simply demand reuse, and components fill this demand. The unit of purchase used to be the application – an entire solution providing full capability in a particular area of business, say purchasing or Customer Relationship Management (*CRM*) – but the unit of purchase is shifting to the component or the service, providing a single unit of service that is then combined with others to provide full capability.

Components bring advantages to the entire process of development. During design, finding components with the right kind of interface is an essential part of the process. If components need to be developed, the external interface can be defined, and then development can proceed in parallel. This is the "black box" approach to components. We do not look inside; we simply deal with the component as a unit that performs its contracted function without concern for how it does it. By assembling components, we are able to deal with larger programs than in a monolithic one-piece design. Components are insulated from one another, and the development of one component team is independent of the developments of any other, reducing bugs and unexpected interactions. The design process becomes mostly concerned with decomposing the application into components, as opposed to being oriented around either procuring or creating these components.

In the development process of the components themselves, quality is aided by defining the interface and the contract the component will fulfill early in the process. Then the component can be tested during development to ensure correctness by checking whether it fulfills this contract. Indeed, some methodologies advocate creating the unit test that verifies the contract as the first step, before the component itself is procured or created.

Once the components are assembled into the application and we determine that each component and the container and interactions are bug-free, then we can be confident that the overall application is of high quality.

Advantages of Components and Frameworks:

⇨ *Time to market* – most of us is in the software business to make money. The industry moves quickly and often the first to market has opportunities that others do not. Sometimes it is the opportunity to be the first to go belly–up, if applications are built quickly and without regard to quality – but if the job is done right, the first to market can establish them as a leader early on, and this advantage often lasts. Components help with this; they let the job of building applications precede much more quickly, without sacrificing quality.

Building with components goes much faster because the detailed logic to perform each individual service is already complete – it is just a matter of wiring up the pieces. The time saved by not having to design and create sophisticated infrastructure is also important. Even when custom logic is required, creating it is faster when we start with existing component architecture.

The time to respond to changes in the market is often shortened by components as well, allowing a company to not only be the first out of the gate, but also faster and more nimble in the twists and turns involved in changing their product to keep up with changes in the market demands and in technology over time.

⇨ *Quality* – what do we mean by "quality"? Why is it important? Quality describes the abilities of a product to meet

stated or implied needs - in other words, to get the job done. It is more than this bare definition, however; quality affects the entire life cycle of a software product, from design to long–term maintenance.
The old view that some defects are inevitable, and that to maximize profitability a certain percentage of defects must still reach the customer, is being seriously challenged today. Particularly by using components and frameworks, we have the ability to build quality from the outset, rather than correcting defects later in the develop–test–debug cycle.

Why quality? Simply put, quality software makes good business sense. The true cost of creating, selling, and supporting high-quality software is overall far lower than the cost of low–quality software. One of the least expensive ways to improve software development *productivity* is to improve software *quality.* Quality, therefore, is cost effective – it retains customers, and has become a key competitive issue in software development.

As the practice of application development continues to mature, becoming less of an "art" and more of an engineering discipline, the lessons of quality learned in the manufacturing and engineering world can be applied to it more readily.

To increase software quality, the people involved and the processes they work with are even more important than the tools they use. Frameworks and component technology provide more than tools; they provide a philosophy of development that is quality oriented.

One of the advantages of a reusable component is that it has been, well, reused. This means that it has been tried and tested in other applications and is known to perform its particular function correctly. All other things being equal, a known quantity is better than an unknown one when you are building an application.

Components themselves demand higher quality: in once–off software development, a particular function of an application may be rarely used, or rarely used in a certain way. A defect in the rarely used area might never be noticed, or might not be perceived as a significant problem even if it is. With components, however, the component developer does not necessarily know how the finished component will be used when it is assembled into an entire application. A function he thought was of little importance might be the cornerstone of an application, and defects in that function would be completely unacceptable. As a result, it is essential that every part of a component be of as high quality as possible. Because of the reuse of components, any defect is likely to be replicated into many applications, creating a bigger problem. On the other hand, the benefits of extra effort in developing quality software in the first place are also replicated. This makes the effort all the more worthwhile, the more the component is reused.

Because the components in a framework do not have any knowledge of our specific application, they are more independent of a particular application – this usually makes them more reliable overall. After all, if we are building something to be used once, quality is not as essential as if you are building something to be used again and again. As software life spans very often far exceed their initial estimates, it is always better to assume a long life than a short one, and build accordingly.

- *Quality environment* – the first step in achieving a quality application is the environment in which it is developed. By this we mean all of the factors around the application, including the corporate infrastructure, the physical environment, the corporate culture, and more.

In a corporate environment where development is thought of as a necessary evil, it will be difficult to achieve quality. If management is only concerned about getting it done, and spending as little time and money as possible, quality is hard to achieve. If the corporate culture is such that developers are an afterthought in the *IT* infrastructure, and are not given the tools to do a job well, chances are that the job will not get done properly. This includes everything from a quiet, isolated workspace to proper software tools, and sufficient time

to do the job right. Unrealistic schedules do not inspire faster work – quality software development, like quality in any engineering discipline, takes careful planning and sufficient time.

Proper tools for a job are essential. This starts from the hardware up all the way to the Integrated Development Environment (*IDE*) and the documentation tools. Some *IT* managers have, for instance, the absurd notion that developers should have the slowest machines, and are under the impression that this will somehow make sure they develop faster code. The same managers then skimp on a test environment, denying the opportunity to test in a realistic deployment scenario. Such misconceptions do not permit quality development.

Although a framework does not necessarily influence any of these things directly, its use does imply certain attitudes about development. Using a framework represents a commitment to a training cycle. It represents a decision on the part of the organization to produce an application that is better equipped than something that could be developed independently, and this in itself gives an indication of the attitude of the organization toward development – that it should be done right, and with the right tools.

1. **Design quality:** as I have said before, a framework is not just a collection of components and abstract classes. It is also a set of design patterns, and defined ways for these patterns to interact. In the process of developing a framework, these abstractions are created, usually from existing applications in the problem domain. A well–designed framework represents the distillation of many different application design principals into a cohesive and usable whole, and therefore provides a substantial jump start on the design process for the application overall.

2. **Meeting of stated and implied requirements:** by definition, quality includes the ability of the finished application to meet the stated and implied requirements. *Stated* requirements are not too bad – generally there is some consensus between the developer and the user on what these are at least. *Implied* requirements are bit harder. If operate with a somewhat clouded crystal ball, may discover that the user or customer was under the impression that much more was implied than we were led to believe. A framework assists substantially in this area as well, as it supports the need for the application to change or even expand during the development process by providing pre-built functionality, even if this functionality was not seen as part of the requirements for the project initially.

3. **Prevention of defects – proper planning:** an important part of preventing defects early in the development process is to have a proper project plan for the development. Correct and realistic scheduling is an essential part of this process. Nothing creates the opportunity for defects better than late specification changes – they cause new functionality to be "shoe–horned" into a design that was not intended for it. A framework-based project has the advantage here as well. Rather than leaving the development process open to "surprises" later on, a framework gives you an almost "paint–by–numbers" level of control over the development. You know what the capabilities of the framework are, and what is provided. You are assembling parts, rather than embarking on journeys of discovery. We will not have a "surprise" half–way through the development when you discover that a feature planned for development takes much longer time than anticipated, or that a driver does not work with the database you have been planning to use. An existing framework with functioning examples takes the mystery out – you know applications in the same problem domain as yours have been developed using this tool–set. You can plan better by working with the known components, and this makes for a better overall project plan, and enhanced quality.

4. **Avoid coding defects:** the best way to avoid coding errors is to write less code. By using a framework, the total number of lines of code that need to be developed for any given application is an order of magnitude less than it would otherwise be, and the preexisting code of the framework is likely far better tested than any single application's code. This leads to much lower incidence of coding errors, and at the same time provides more time for the code that *does* need to be written, increasing its quality as well. Real–world experience has indicated that use and reuse of components can reduce

the occurrence of defects from five to ten *times* compare to once-off code.

5. **Coding standards:** by establishing coding standards and sticking with them, coding defects can be reduced further, and existing problems can be more readily discovered. Coding standards also make it much easier for teams of developers to read and understand each other's code, thereby promoting team development.

6. **Detection of defects and testing:** if we have a hundred identical parts lying on the table in front of you, it is easy to scan them all for defects: you look for what stands out, what is different and not in conformance with the rest. It is much harder if you have got a hundred different parts – you cannot scan for patterns, you must examine each one individually.

7. **Support quality:** quality of support and training for an application is also an important factor. The best application is not meeting its stated and implied requirements, after all, if no one can understand how to use it. This is another area where the consistency of development created by the use of a framework comes into play.

8. **Maintainability:** to continue providing a quality solution to its users, any application must be maintained over time. New features will be added, integration with other applications will be made, user-interfaces may be enhanced, and so forth. Using components and frameworks makes a significant difference here, because by their very nature, reusable components are intended for many different purposes. Therefore, as the purpose of the application is extended and changes over time, we are more likely to be able to adapt components to these changes than to adapt custom–built code. In addition, the clean encapsulation of components means that individual components can have their implementations replaced easily – without disrupting the remainder of the application. For example, an application may originally be written to use flat files for its data storage. A later upgrade then requires a relational database – the component relating to data storage can be upgraded to one that is database-capable, whereas the remainder of the application remains unchanged. If, instead, custom file access code was used throughout the application, it may not have been viable to upgrade at all.

Although it is hard to measure the ability to maintain software, most experts agree that component–based systems are much easier to maintain than their monolithic counterparts. Therefore, as we have seen, component–based development can make a significant difference in the most serious problem with software development in today's industry: quality. Not only can components address this issue, but they can do it while at the same time increasing developer productivity, decreasing the time to market, and with greater adaptability than any other approach.

⇨ *Cost* – partly as a result of the issues discussed earlier, the overall cost of building with components, despite the extra work of achieving reuse and the other qualities required by a good component, is lower than once-off development. This becomes even more the case when the components can simply be assembled, rather than created. Time is literally money in the software business, so the time saved in development by assembling components contributes to the cost savings.

Even commercial components that have a direct cost associated with their use can still sometimes make the overall project cost lower. Not only is the cost often lower for components than for single-purpose-built software, but, almost more importantly, it is *predictable*. We know the cost of using a component ahead of time; this means one less variable in the complex equation for forecasting a project's overall budget.

Decisions that make overall architectural changes to a project are often those that incur the maximum cost, and the most risk. Designing a new infrastructure is not easy or quick, and a mistake at this level can be disastrous to a project. Standardizing on a tried and true component model eliminates this risk, further contributing to control of cost.

⇨ *Adaptability – change by reconfiguring, not rewriting* – components, by their nature as reusable pieces, tend to be more configurable and adaptable than code that is written as a one–of–a–kind. This flexibility is then available when the

specifications of the original development change. It is usually possible to change the components by simply reconfiguring them, as opposed to having to open the black box and start tinkering inside. This means the adaptability is gained without a loss of quality. This has a direct impact on the maintainability of the overall project.

Like most techniques, however, use of configurable components can have its down-side if applied incorrectly. Some components are not good candidates for being made highly configurable – performance can suffer, and the component complexity can increase substantially. Sometimes separate implementations are a better approach than a component with a huge number of configurable options. Finding a balance in this trade–off is part of the skill of an expert component developer, and when it is done well, it becomes another reason to prefer well-built components over single–use software elements.

As most projects have over 80 percent of the development effort they absorb spent on maintenance, this adaptability is a major benefit in terms of cost.

⇨ *Scalability* – components, if built correctly in the first place, can often provide a basis for better scalability than custom code. Because the scope of the component is known ahead of time, its place in the architecture is better understood in advance, giving an opportunity for the component to be made distributed and clusterable, opening the door to enhanced scalability.

⇨ *Integration* – by design, components are intended to plug into other things – they are not a complete application on their own, so integration is expected. This tends to make components easier to integrate with just about any other code elements, and raises the integration capabilities of the whole application.

This means components and component-based systems are ideal for connection to legacy systems – indeed, components are often created to "wrap" legacy applications, allowing them to be used just like any other component.

The increasing interest in components, due to the advantages they offer, has in turn spawned an urgent interest in the technologies that connect them together and support them. The two major leaders in the race to provide such connections and infrastructure are Java, and in particular the *J2EE* platform, and Microsoft, who arrived late to the game, but are aiming their *.NET* platform at the same space. Interestingly, the business requirements becoming the driving force in the market has pushed vendor preferences lower on the importance scale than it used to be: people are less interested in whom they are doing business with than the quality of the results to their business. This drives the requirement for multi-vendor solutions to integrate more frequently, as *IT* departments become "multi–vendor" shops. One of the emerging technologies in this area that has great promise is Web services, and in particular the *SOAP* protocol.

Beyond *E–Commerce*: components at work

The newer the sector of the *IT* industry, the more you will find components there. When something works, organizations tend to leave it alone – many legacy systems that predate component technology are still out there, ticking away, performing their functions. New developments, however, have a better chance of being created with component development, and e-commerce is one such development.

Before the Internet, there was no such animal: e-commerce has come about with the rise of the Internet – or more specifically, the *WWW*. *E–commerce* is generally thought of as the process of offering products or services for sale via the Web. The condensed time frame of trends on the Internet has prompted companies trying to get on the rapidly moving bandwagon to try out components and other rapid development technologies, such as scripting languages.

The scripting route, using technologies such as *Javascript*, *ASP*, *CGI*, and so forth, was faster, at least initially. But many such solutions were all scenery with little substance. They looked good, but did not have the back-end power to connect to essential business systems, as necessary when weaving e–commerce into a cohesive

part of an overall company strategy. The solutions built with components tend to fare a little better – although some of these were all flash and no bang.
E-commerce, however, was only the beginning – the first ripple of a much larger wave. The second phase was *e–business*. This is where not just a company's interaction with customers and potential customers is carried out in part over the Web, but where a company also interacts with its peers and business partners via the medium of the Internet. Suppliers, distributors, and wholesalers – the volume and critical nature of the Web interactions was suddenly much higher than when the Web was just one more means of reaching customers.

Now the Web was essential to business, and significant competitive advantages could be gained by leveraging it correctly. Component–based development became the technology of choice for creating these new business–to–business Internet applications. Its standard and easily connected attributes, and its ability to adapt quickly, became essential facilities.

Business–to–Business has already become a more important form of e-business than *Business–to–Consumer*, and the growth is still rapid. As more organizations begin to take advantage of these new opportunities, they will increasingly find component and framework development an essential tool.

Conclusion
We have seen why all of the interest in components and the frameworks that support and foster them, has come about. To entirely mangle a famous saying, however, a rose by any other name is just a red flower with sharp thorns. Calling something a component does not make it so, and definitely does not magically convey the advantages we have discussed on the product so named. Marketing and hype being what they are in the software industry, it is easy to make a product buzzword compliant, but not as easy to actually do the engineering necessary to make it worthy of the name *component*. Searching the net, it is easy to believe that the whole software world has jumped on the bandwagon of components, and that virtually any product is built of these wonderful, configurable, and reusable pieces. In reality, the battle for components has only just gotten underway, and the big guns have just started firing recently. So beware of any solution that promises to be the end-all and be-all of component development. Ask the questions that can determine just which of the many advantages of components the solution is providing, and how.

The great wave of component-based development is so far just a ripple. It is a great concept, but the groundwork that had to be laid is only just propagating to companies *IT* departments. Once the enablers are in place though, dramatic strides forward in quality, productivity, and interoperability will be seen. This is not theory – many companies are there already, and are showing the way to the future.

References:

- *Fernandes, C., Web Application Development - A Guide to Success, Upper Saddle River, 2003.*
- *Korotkiy, M., Top, J.L.: On the Effect of Ontologies on Quality of Web Applications, (under review), 2005.*
- *Reifer, D.J., Estimating Web Development Costs: There Are Differences. The Journal of Defense Software Engineering, 2004.*
- *Trauring, A., Python: Language of choice for EAI, EAI Journal, p. 43–45, 2004.*
- *Vinoski, S., Dark matter revisited. IEEE Internet Computing 8,2004, p 81–84.*
- *http://se.ethz.ch/~meyer/publications/computer/component_development.pdf.*
- *http://www.artificia.co.uk/.*
- *http://www.cs.ucl.ac.uk/staff/A.Finkelstein/papers/uwa.pdf.*
- *http://www.dig64.org/specifications/IA64_Java2.pdf.*
- *http://www.elet.polimi.it/conferences/wq04/final/paper02.pdf.*
- *http://www.iist.unu.edu/newrh/III/1/docs/techreports/report284/paper8.pdf.*
- *http://www.lcc.uma.es/~av/mdwe2005/camera-ready/8-MDWE2005-portlets.pdf.*
- *http://www.mrtc.mdh.se/publications/0953.pdf.*
- *http://www.sybase.com/content/1048029/Sybase_WADWS_L02894-111606-wp.pdf*