# Enhancing Component interactivity and manageability using delegates

**By**
**Fady S. FADDOUL**

A thesis
submitted for partial fulfillment
of the requirements for the degree of
Masters in Computer Information System

Department of Computer Science
Faculty of Natural and Applied Sciences
Notre Dame University

March 2004

# Enhancing Component interactivity and manageability using delegates

By
Fady S. FADDOUL

**Approved**

[illegible]

**Khaldoun El-Khaldi: Assistant professor of Computer Science.**
**Advisor**

[illegible]

**Omar Rifi: Assistant professor of Computer Science.**
**Member of committee.**

[illegible]

**Ahmad Shahin: Assistant professor of Computer Science.**
**Member of committee.**

[illegible]

**Ziad Rached: Assistant professor of Mathematics.**
**Member of committee.**

**Date of Thesis Defense: March 24, 2004**

## ***Acknowledgements***

Conducting a research requires a lot of guidance and supervision from academic people and experts. Not quite that many helped me with this thesis , but I certainly couldn't have done it alone. If I have inadvertently left someone out below , I am really sorry and I thank him too.

For his helpful suggestions, attention and precious remarks throughout the thesis, I would like to give a special thanks to *DR Khaldoun Khaldi* whose expertise in component field was the corner stone behind the completion of this thesis.

A special thanks to my parents who always supplied me and still supplying me with all the needed resources to pursue my education. Without them I wouldn't have been given the opportunity to defend my thesis in front of the jury.

A special thanks to NDU and USJ for supplying me with the scientific background necessary to conduct a Masters Degree research. They are doubtlessly 2 of the best universities in the Middle East.

## ***Abstract***

Software components enable practical reuse of software parts and amortization of investments over multiple applications. Software components are binary units of independent production, acquisition that are combined together to form a functioning system

Building solution by combining bought and made components improves quality and supports rapid development, leading to short time to market. For these reasons, component technology is expected by many to be the corner stone of software in the year to come.

Even though component software is a very promising way of building applications, it is very naïve to think that several components can be picked from a catalog and wired together to form the final product. In real life adopting a component approach poses several problems and rise several important challenges. The aim of this thesis is to identify some of these challenges and problems and to propose an adequate solution to reach a successful final software.

# Table of Contents:

## List of figures:

# Chapter I:

## 1.1 Component introduction:

### Components are for composition:

One thing can be stated with certainty: components are for composition. Composition enables prefabricated things to be reused by rearranging them in every new composite. Beyond this trivial observation, much is unclear. Are most current software abstraction not designed for composition as well? What about reusable part of designs and architectures? Is reuse not the driving factor behind most of compositional abstract? Reuse is a very broad term covering the general concept of reusable asset. To become a reusable asset it is not enough to start with a monolithic design of a complete solution and then partition it into fragment. Instead, descriptions have to be generic to allow for reuse in sufficiently many different contexts. Over generalized has to be avoided to keep the description nimble for practical reasons. Software components are binary units of independent production, acquisition and deployment that interact to form a functional system [1]. Composite systems composed of software components are called component software. The requirement of independence and binary form rules out many software abstractions, such as type declarations, C++ templates. Other abstractions such as procedures, classes, modules or even entire applications could form components as long as they remain in a binary form that remains composable. Indeed, procedural libraries are the oldest example of software components. Insisting on independence is essential to allow for multiple independent vendors, for independent development and for robust integration. What is the motive for producing, distributing buying or using software components? What are the benefits of component software? The simplest answer is: components are the way to go because all other engineering disciplines introduced components as they became mature - and still use them [1] [5].

## ***The nature of software and deployable entities:***

Software components were initially considered to be analogous to hardware components in general and to integrated circuit in particular. Thus the term software IC became fashionable. Also popular is the analogy between software components and components of stereo equipment. More far fetched are analogies with the field of mechanical and civil engineering: with gears, nuts and bolts. However, comparison did not stop at engineering discipline and continued on into areas as extreme as the world of toys, the Lego block model of objects technology was conceived and marketed. These analogies helped to sell the idea of software components by referring to other disciplines and areas in which components technology has been is use for some time and had begun to fulfill its promises [1] [5]. Software is different from products in all other disciplines, rather than delivering a final product, delivery of software means delivering the blue prints for products. Computers can be seen as fully automatic factories that accept such blueprints and instantiate them. It is important to distinguish between software and its instances, as it is to distinguish between blueprints and products. Between plans and building etc... The corresponding distinction between class and object is frequently omitted, although there is occasional clarification of something as an 'object instance' or an 'object class'. The established practice of not distinguishing between objects and classes leads to a number of some questionable publications such as 'The port class has 1024 virtual-circuit classes'. What the author meant was the port has 1024 virtual circuit objects. In this case a port should not be seen as a class but as a group and the virtual port not as a subclass but as set of elements. From a purely formal point of view, there is nothing that could be done with components that could not be done without them. The differences are concepts such as reuse, time to market, quality and viability [1].

***<u>Market versus technology:</u>***

Components are reusable assets. Compared with specific solutions to specific problems, components need to be carefully generalized to enable reuse in a variety of contexts. Solving a general problem rather than a specific one takes more work. In addition because of the variety of deployment contexts, the creation of a proper documentation, test suites, tutorial, online help texts, and so one is more demanding for components than for a specialized solution. Components are thus viable only if the investment in their creation is returned as a result of their deployment [1]. If components are developed for in-house use, such return on investment can be indirect via benefits of using components rather than monolithic solutions. Such benefits are typically a reduction in the time to market and increased manageability, maintainability, configurability, flexibility and so on. Of course, return on investment can also be sought by selling components. The direct sale of components to deploying customers is one way, but it is not the only one. Another way is the coupling of components and services: while the components may be cheap or free, their effective use may require significant expertise that is offered as a service. It is important to understand that software components as all components in all areas need to be understood in a market embedding. Components will exist only where components vendors and components clients join forces to reach a critical mass. For the existence of software components, software component technology needs to meet with proper market. The technology can't evolve without a sufficiently strong market. Let us consider a simple example: An engineering company works on a contract basis for many clients in tuning software for engine control. Its clients have varied requirements and traditionally each project started from scratch. Then the company realized that most of its jobs had a lot in common, and so it decided to extract and generalize a set of generic components [1]. This effort sat firmly on the experience obtained with the concrete specific projects that it has run before. Thus the new component set was slim, the efficiency of the company increased quickly amortizing the initial investment in the component development. In the meantime the company opened a subsidiary selling the components to other engineering companies serving different regions. The story of the above-mentioned company's success reached a start-up company; its engineers knew everything about componentware and decided to start by designing the ultimate component collection before even approaching a single client. As first projects came in most of the generic facilities of the components were not used at all. Also, the solutions it eventually delivered required excessive amount of processing power and computing resources. It turned out to be a fatal mistake to generalize before solving some specific problems These 2 stories describe briefly and clearly the 2 different sides of Component market 'coin'. It also shows that component technology requires experience before it is successfully implemented on the market. Burning steps can be a fatal mistake [1] [5].

## ***The importance of standards:***

For a component to find any reasonable number of clients, it needs to have requirements that can be expected to be widely supported. It also needs to provide service can be expected be widely. How wide? The answer depends on the domain addressed by a component. A component needs to hold a significant portion of a market specific to its domain. If that market is truly large, a small portion of it may be enough for economic viability. If the market is only small, even the market in only small, then even a total monopoly may not be enough to justify the investment. If a component viably addresses a market segment covering a small number of clients, the component vendor may exactly understand the individual client's needs and deployment environments. The vendor then makes sure that the component will function as required in these environments [1] [5]. As the number of potential uses and clients grow, the chance that any component could possibly address all need while being deployable in all environments decrease rapidly. The unavoidable middle ground that both client and vendors need to seek is based on environment standards. It is totally irrelevant whether such a standard has been approved by a regular standardization body. The most successful standards have been created where and when needed by those parties who needed them. For software components, the need for standards was recognized a long time ago. One approach is to build working markets first, followed by the formulation and publications of standards. In the software component world, Microsoft (COM, OLE, ActiveX, and COM+) is one player following this approach. Another approach is to build standards first then to build the market; the prime player in this arena is Object Management Group (OMG). Where markets are created first, working products needs to come before establishing standards. Products using Adhoc solution may be sold before fully understanding of all ramifications is reached. To upgrade to better solutions without losing the established based of customers, products need to be evolved carefully. Almost all successful standards emerged this way [1].

## ***Component standards:***

Who will win the standardization race in component technology? Interestingly, there is no need for a single winner. As long as market shares remain large enough, multiple standards can coexist and compete. Even in highly mature engineering discipline there are a number of alternative standards for a given situation. With a slowly developing maturity of software components comes a slow liberation from poor objects [1]. It is too simplistic to assume that components are simply selected from a catalog, wired together and magic happens. In reality, the disciplined interplay of component is one of the hardest problems of software engineering today. Question arise about how can the abstract interaction of components be described, or how can performance be guaranteed in an heterogeneous system composed of different parts coming from different places. A particular powerful approach is beginning to take shape – that of component frameworks. A component framework is a set of interfaces and rules of interactions that govern how components plugged into the framework may interact. Microsoft .Net component framework is an example of how components interact efficiently together for an optimum performance [1].

***<u>More about components:</u>***

The characteristics properties of a component are:

A component is a unit of independent deployment
A component is a unit of third-party composition
A component has no persistent state [1] [5].

These properties have several implications: For a component to be independently deployable, the component needs to be very well separated from its environment and from other components: component software therefore encapsulates its constituent features. Also it is a unit of deployment; a component will never be deployed partially. In this context, a third party is one that cannot be expected to have access to the constructions details of all the components involved [1]. For a component to be composable with other components by such a third party, it needs to be sufficiently self-contained. Also it needs to come with clear specifications of what it requires and what it provides. In other words a component needs to encapsulate its implementation and interact with its environment through well-defined interface. Finally for a component not to have a persistent state, it is required that the component cannot be distinguished from copies of its own. Possible exceptions for this rule are attributes not contributing to the component functionality, such as serial numbers used for accounting. Not having state a component can be loaded into an activated into a particular system [1] [5]. But is make very little sense to have multiple copies. In other words, in any given process, there will be at most one copy of a particular component. In many current approaches, components are heavy weights units with exactly one instance in a system. For example, a database server could be a component; the database server together with the database might be seen as a module with global state. According to the above definition, this module is not a component. Instead the database server is and it supports a single instance: the database object [1].

## ***Objects:***

The notion of instantiation leads to the notion of objects. The characteristics properties of objects are:

An object is a unit of instantiation; it has a unique identity
An object has a state
An object encapsulates its state and behavior [1] [5].

A number of objects properties directly follow. Because an object is a unit of instantiation, object can't be partially instantiated. Since an object has an individual state, it also has a unique identity that identifies the object despite state changes for its entire lifetime. As objects get instantiate, there needs to be a plan that describes the initial state and behavior. Such plan may be explicitly available and it is called *class*. The newly created object needs to be set to an initial state. The initial state needs to be valid state of the constructed object. But it may also depend on parameters specified by the client asking for a new object. The code required to control object creation and instantiation can be a static procedure usually called a constructor [1].

## ***Components and objects:***

A component is likely to come to life through objects and therefore would normally consist of one or more classes. However there is no need for a component to contain classes only, or even to contain classes at all. Instead a component could contain traditional procedures or even static variables and declarations. A component may contain multiple classes, but a class is necessarily confined to be part of a single component. Partial deployment of a class wouldn't normally make sense. The superclass of a class do not necessarily need to reside in the same component that the class itself [1]. Where a class has a superclass in another component, the inheritance relation between these 2 classes crosses components boundaries. Whether or not inheritance across component is a good thing is the focus of a heated debate between different schools of thought [1] [5].

## ***WhiteBox versus blackbox:***

Blackbox and whitebox abstraction refer to the visibility of an implementation behind its interface. In an ideal blackbox abstraction, no details beyond the interface are known by the client. In a whitebox abstraction, the interface may still enforce encapsulation and limit what clients can do, however the implementation of a whitebox is fully available and can be studied by the client. Grayboxes are those that reveal a controlled part of their implementation [1]. Blackbox reuse refers to the concept of reusing implementation without relying on anything but their interfaces. In contrast, whitebox reuse refers to using a software fragment while relying on the understanding gained from studying the actual implementation. Most class libraries and frameworks and delivered in source form, and applications developers study the classes implementation to understand what a subclass can or has to do [1].

## ***Interfaces:***

An interface may be defined as a component access point. It allows the client to access the service provided by the component. Normally a component will have multiple interfaces corresponding to different access points. Each access point may provide a different service, catering for a different client needs. While designing an interface the economy of scale has to be kept in mind. A component can have multiple interfaces, each representing a service that the component offers. Some of the offered services may be less popular than others, but if none is popular and the particular combination of offered service is not popular either, the component has no market [5].

## ***Explicit content dependencies:***

Beside the specification of provided interfaces, the above definition of components also requires components to specify their needs. In other words, the definition requires specifications of what the deployment environment will need to provide the functionality of the component. These needs are called context dependencies. In reality there are several component words that partially coexist and partially compete: For example today there are 3 majors' component worlds emerging: OMG's CORBA, Sun's Java, and Microsoft's Com and recently the .NET framework. Just as the markets have so far tolerated a surprising multitude of operating systems, there will be room for multiple component words. CORBA and Java component are system independent; they can run of various operating systems. Microsoft COM and COM+ can only run in a window environment. . NET components require .NET framework to be installed as part of their dependency system [1].

## ***Horizontal versus vertical markets:***

When aiming for the formation of standards that covers all areas that represents sufficiently large markets it is useful to distinguish standards for horizontal and vertical markets. A horizontal market sector cuts through all or many of different market domains. ; It affects all or most clients and providers. A vertical market sector is specific to a particular domain and thus addresses a much smaller number of clients and providers [1]. For example, the Internet and World Wide Web standards are both addressing horizontal market sectors. In contrast, standards for the medical radiology sector address a narrow horizontal vertical market. Standardization is hard in horizontal market sectors. If a service is relevant to almost everyone, the length of the wish list tends to be excessive. Surprisingly, standardization in vertical sectors is just as difficult as it is in horizontal market sectors, but for different reasons. To justify the investment the majority of the vertical market should purchase the component [1] [5].

## ***Interfaces as contracts:***

A useful way to view interface is a contracts between a client of an interface and the provider of the interface. The contract states what the client needs to do to use the interface. It also states what the provider has to implement to meet the service promised by the interface. The 2 sides of the contract can be captured by specifying pre and post conditions for the operation, the client has to establish the pre condition before calling the operation and the provider can rely on the precondition being met whenever the operation is called. The provider has to establish the postcondition before returning to the client and the client can rely on the postcondition being met whenever the call to the operation returns [1].

## ***Contracts and non-functional requirements:***

As long as an implementation respects its contracts, revisions pass unnoticed by clients. It is worth noting that typical contemporary contracts often exclude precise performance requirements. However, even for simple procedural libraries a new release adhering to the original contract but changing performance can break clients. Consider a math library that is used by an animation package. Next, the math library is improved to deliver more accurate results, but at lower average speed. This improvement turns out to break the animation package, which now fails to deliver the required number of frames per second [1]. Take an example from the current practice that Swiss banks use when subcontracting a component to a third party. The contractual specifications consist of the functional aspects: input – output parameters, pre and post conditions – but also a so-called service level. The service level covers guarantees regarding availability, mean time between failures, mean time to repair, throughput latency etc... Failure to treat the service level is treated on the same grounds as wrong results: the component broke its contract. It can be expected that this practice of including non-functional specifications into a contract and monitoring them strictly will become more widely popular in the future [1].

***Component usefulness:***

A component may be viewed as:

***Unit of abstraction:*** Abstraction is a very powerful tool available to a software engineer. Abstraction aims at reducing details making the thing that has been subjected to abstraction simpler to handle. The main benefit of an abstraction is the design expertise embodied in it, ready for reuse. However from a software point of view, the hardest design problem is how abstractions such as objects should interact [1].

***Unit of accounting:*** In large systems, as are typical for enterprise solutions, the actual cost incurred by individual parts of a system and their use may need to be monitored. It thus becomes important to partition a system into units of accounting. As components are units of deployment, it makes sense to also make them units of accounting. In this way it becomes possible to link costs and benefits to acquired components [1].

***Unit of compilation:*** compilation is a quite fundamental aspect in computing. Incremental compilation can speed up the edit – compile – link – run cycle considerably. In a component software world, complete application no longer exist and thus the application as a unit of compilation is not feasible. To enable global optimizations, compilation units should be as large as practically feasible; components are the best upper limit [1].

***Unit of delivery:*** Today, applications and sometimes components are the typical units of delivery. Individual objects are rarely worth the administrative effort and cost of delivery. Individual classes are also rarely sufficiently self-contained to allow for separate deployment. In a system in which classes are the only structuring facilities, it becomes very difficult to extract and package a suitable subset of classes; component packaging seems to be a better option [1].

***Units of dispute:*** If a system composed of several components fails, component vendors tend to blame each other for the problem. To minimize this undesirable effect, it is vital that error remains contained in an individual component. This means that they should not endanger the system as a whole (bug containment). Languages either prevent errors or allow the component that caused the error that occurred to be pinpointed exactly. In a system composed of independently developed, these help clients to find out which vendor's software has failed. If a component identity could not be clearly determined, it would become very difficult if not impossible to find out which vendor's software is culprit [1].

***Unit of extension:*** A component may not provide completely new functionality, but instead extend existing functionality, the coupling between the objects forming an extension is tighter that between extending and extended components [1].

## ***1.2 Problem to Solve:***

Components are a very hot topic in today's business computer. They are important in both technical and management fields since they promise to cut expenses, distribute the right piece of work to the right people and considerably increase the success rate of a product. This is the bright side of the story because in reality implementing a component approach poses several problems and challenges. The purpose of this thesis is to discuss some of these problems and to find a satisfying implementation to solve them. My proposed solution can be used as a template to build component oriented application in various fields.

Components are units of code written and compiled by several third parties and assembled in the main product. It is obvious that from such fact will derive coordination problems especially in a multi thread environment such as Windows, UNIX, LINUX etc. To solve this problem a technique should be found to enable the component to periodically send information from the component side to the client side.

The relation between the component and its container is dynamic and therefore can't be determined during the design phase of the components. For a component builder, it is impossible to determine when the container will require a service in advance, or a modification of the internal status of his object. Such information can only be known during runtime mode. To solve this problem a technique should be found to enable the client to dynamically interfere with the contained object and to change its internal status on demand.

Extending a component requires that the language used to build the original component be the same that the language that will be used for extending it.

This handicap reduces considerably software flexibility and binds the component to a single language. By using the cross platform technique recently introduced to the market a solution to this problem can be found.

Introducing a new version of a Com component sometimes causes confusion because applications relying on the previous version will fails to access the new version. This problem can be solved by using the features of the global assembly cache recently introduced to the market. It allows several versions of the same components to coexist with no conflict at all.

## ***1.3 Thesis Background:***

### ***1.3.1 Component framework:***

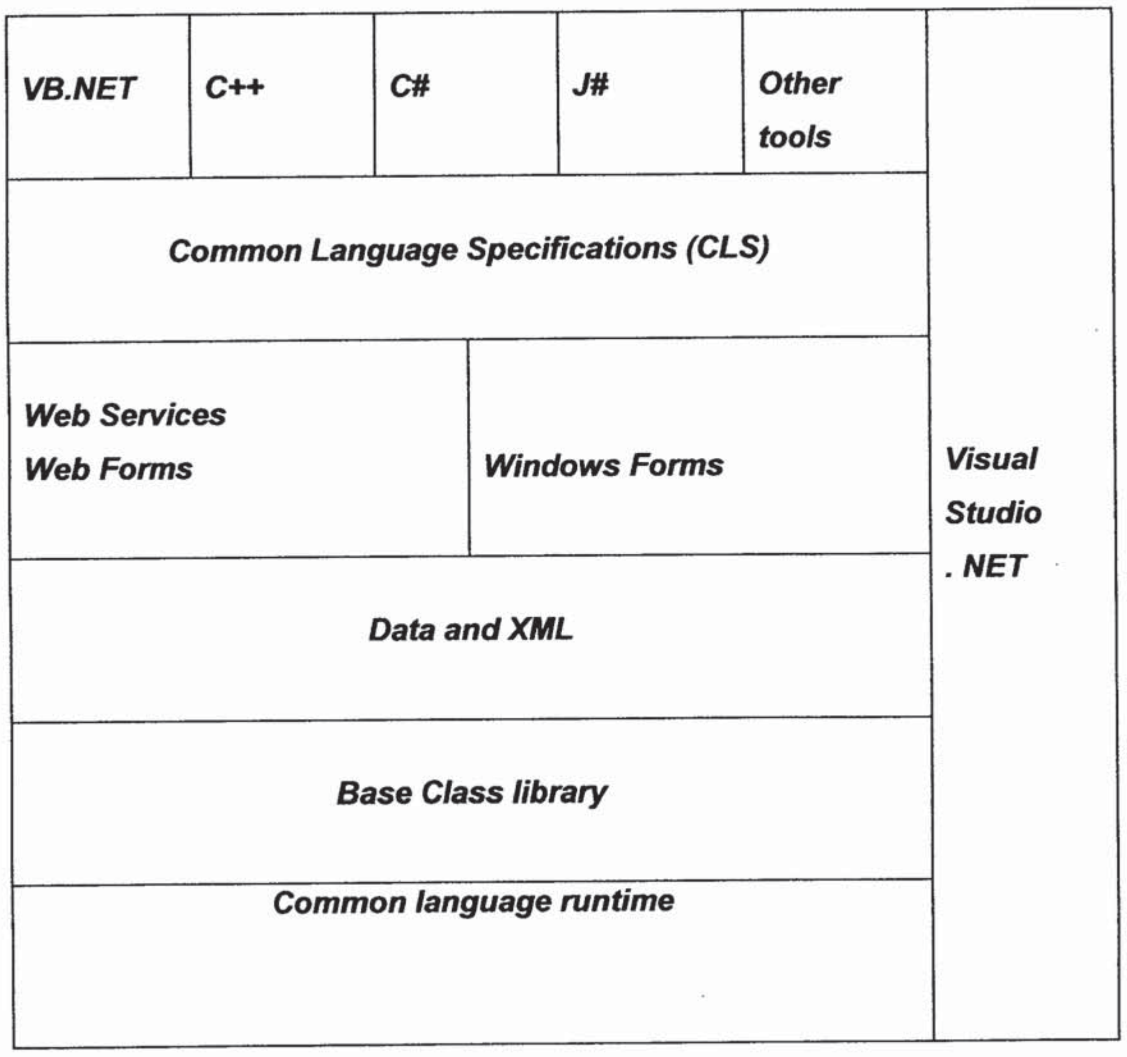
Component frameworks are the most important step to lift component software off the ground. Most current emphasis has been on the construction of individual components and on the basic of wiring support of component. It is thus highly unlikely that components developed independently under such conditions are able to cooperate usefully. The primary goal of components technology, independent deployment and assembly of component is not achieved [2]. A component framework is a software entity that supports components conforming to certain standards and allows instances of these components to be plugged into the component framework. The component framework establishes environmental conditions for the component instance and regulates the interaction between component instances. Component frameworks can come alone, or they can cooperate with other components or component frameworks. In essence Microsoft .NET framework is a collection of classes and types that encapsulates and extend Win32 API in an object oriented context. This hierarchy is defined by inheritance: Simple reusable classes such as components are provided and then used as a base from which more sophisticated classes is derived. The following table gives an overview about the core classes of Microsoft .NET [2] [4].

### ***1.3.2 The .NET platform introduction:***

The .NET platform is the foundation upon which the next generation of software will be built. Microsoft has invested a lot of capital in its development, and it putting considerable weight behind its adaptation as a new standard. The .NET platform is more than a new language, software development kit (SDK) or even an operating system. It offers powerful new services: a new processor independent binary format, new managed language, extension to existing languages, and the list goes on. The percept behind the .NET platform is that the world of computing is changing from on of PC's connected to servers through networks such as the Internet, to one where all manners of smart devices, computers and services work together to provide a richer user experience. Looking at the overall architecture, .NET consists of three primary components [2] [4].

***The .NET framework***: A completely new application development platform. It can be divided into 3 parts: the CLR responsible of the execution environment such as memory allocation, error trapping and interacting with the operating system; The base class library an extensive collection of programming components and application program interfaces and finally 2 top-level development targets: one for web application (ASP.NET) and another for regular windows applications [2] [4].

***Several .NET products:*** Various applications from Microsoft based on the .NET framework, including new version of exchange and SQL Server, which are extensible markup language (XML) enabled and integrated into the .NET platform. The advantages offered by the .NET framework include shorter development cycle (code reuse, fewer programming surprises, support for multiple programming languages), easier deployment and a garbage collector. The .NET platform is the next generation of what was called windows DNA. Windows DNA was a technical specification that focused on building software based on Microsoft server products utilizing numerous technology and languages (ASP, HTML, JavaScript, COM and so on). The following diagram shows the .NET platform architecture. Essentially the .NET family of languages [2] [4].



***Features of the .NET platform:***

***Multilanguage development***: Because many languages target the .NET Common language runtime, it is now much easier to implement portions of the application using the language that's best suited for it. The .NET platform allows languages to be integrated with one another through the use of MSIL. Although it contains instructions that appear similar to assembly code such as pushing and popping values, moving variables in and out of registers, it also contains instructions for managing objects and invoking their methods, manipulating arrays and raising and catching exceptions [2] [4]. The Microsoft Common Language Specifications describes what other development tools must do in order for their compilers to output IL code that will allow them to integrate well with each other. Cross language inheritance is another feature made possible with by the use of IL. Per example it is possible to create a class in C++ that derives from a class implemented in VB.NET. One of the great challenges for developing application under the Windows DNA specifications was in debugging applications developed in a variety of languages. Thanks to the unified development environment of Visual Studio.NET and the use of IL as output of all .NET languages, cross-language debugging is possible without resorting to assembly language. The .NET common language runtime fully supports debugging applications that cross language boundaries. The runtime also provides built-in stack walking facilities, making it much easier to locate bugs and errors [2] [4].

***Platform and processor independence:*** Once written and built, a .NET application can execute on any platform that supports the .NET Common language Runtime. Although at the time of this writing, .NET applications run only on windows platforms, on June 27th 2001, Microsoft announced that it had reached an agreement with Corel to develop a version of a C# compiler and the .NET framework version for UNIX [2] [4].

***Automatic memory management:*** Developers coming from visual basic or COM backgrounds are familiar with the reference counting technique. This technique recovers the memory used by an object when no other object has a reference to it. Essentially when it is no longer needed. Although this sounds perfect in theory, in practice it has a few problems [2] [4]. One of the most common is a circular reference problem where on object contains a reference to another object which itself contains a reference back to the first object. When the memory manager looks for objects that are not in use, these objects will always have a reference count greater than zero, so unless they are implicitly deconstructed, their memory will never be recovered [2] [4].

***Versioning support:*** In COM technology when a customer installs a software package that uses one of the same DLLs as an installed application. However this application used version 1.0 of this DLL, and the new software replaces it with version 1.1. The new DLL makes the application exhibit some strange problems or perhaps stop working. The .NET architecture guarantees that if the application runs after application it is going always to run, regardless the version of any coming version in the future [2] [4].

***Support of open standards:*** In today's world, not every device necessary works under Microsoft OS or using Intel CPU. Realizing this, the architecture of .NET is relying on XML. In fact the entire .NET framework in build around XML [2] [4].

***Interoperability with unmanaged code:*** Unmanaged code is code that isn't managed by the .NET CLR. However, this code is still running in the CLR environment, it just can't get the advantages that it offers such as automatic memory management. COM component interoperate today with the .NET framework component through a layer that handles all the work required when translating messages that pass back and forward between the managed the managed runtime and the COM component operating as unmanaged code
[2] [4].

***Performance and scalability:*** The .NET framework gives a tool to make it easier to design better performing software. One big gain for web development will come from ASP.NET's improved support for keeping code, data and presentation separate. The .NET base class library has an enormous set of functionality, which means that less basic code is required giving the programmer a much more margin to refine his application. New version of Microsoft software rotating around .NET offer improved performance over earlier versions. SQL Server.NET offers quite an enhancement over earlier versions of the database engine and other server products offer enhanced scalability as well [2] [4].

***Garbage collection:*** Memory management is one of those housekeeping duties that take a lot of programming time away from developing new code while tracking down memory status. . NET hopes to do away with all of that within the managed environment with the garbage collection system. Garbage collection works when an application is apparently out of free memory. When an application requests more memory and the memory allocator reports that there is no more memory on the managed heap, garbage collection is called. It starts by assuming everything in memory is trash that can be freed. It then walks through the application's memory, building a graph of all referenced memory. One completed it compact the heap by removing all the memory in use together at the start of the free memory heap [2] [4]. After this is complete, it moves the pointer that the memory allocator uses to determine where to start allocating memory. It also updates all of the application's references to point to their new locations in memory. This approach is known as mark and sweep implementation. As one can see, the garbage collection involves a lot of work and it does take some time. A number of performance optimization involved in the .NET garbage collection mechanism makes it much more than the simple collection given here. Normally programmers just let the CLR take care of running garbage collection when it is required. However at a given time the garbage collection may be forced to run by just calling GC.Collect () [2] [4].

### ***1.3.3 .NET assemblies:***

Application in the .NET always consists of one or more assemblies. An assembly is:

A functional unit of sharing, versioning in the .NET framework can be shared across .NET applications.

In the simplest case, an application can consist of one assembly that contains one module with all the code and resources for the application. In most scenario however, an application has multiple assemblies and each assembly may have multiple files. There are 2 kinds of .NET assemblies [2] [4].

***Private assemblies:*** a private assembly is deployed with and used exclusively by a single application. It is referenced by its simple name. When the application is installed, the private assembly is also installed in the same root directory of the application. That appears under the root directory of the application [2] [4].

***Strong name assembly:*** the .NET framework uses strong name to provide a way to identify assembly uniquely, allowing applications to run with the versions of the strong named assemblies. Strong-named assemblies consist of the assembly identity, which is:

The simple text name of the assembly
The version number of the assembly
The culture information (optional)
A public key for the client.

Strong name guarantees name uniqueness by relying on unique key. Strong name assemblies can reside in:

The application folder
Any folder on the local computer
Any folder on a remote computer
A URL
The global assembly cache.

Assemblies shared by multiple applications should be installed in the global assembly cache. . NET client can access the same copy of the assembly, which is signed and installed in the global assembly cache. If an assembly is not going to be shared, then the assembly should be installed in the application directory. Once a strong named assembly has been installed on the global assembly cache, it is referred to as a shared assembly. The following table lists the main differences between strong name assemblies and private assemblies [2] [4].

| ***Private assemblies*** | ***Public assemblies*** |
|---|---|
| -Can only be installed in an application's directory structure<br>-Are reference only by their simple name<br>-Can have version information, but the runtime does not use it<br>-Are not installed in the global assembly cache and therefore the runtime will not look in the global assembly cache when probing for the private assembly | - Can be installed in a number of different locations.<br>- Are referenced by their simple name culture, version and public key.<br>- Contain version information that the runtime checks when loading the assembly.<br>- Can be installed in the global assembly cache and therefore the runtime will look in the global assembly cache as part of the process.<br>- Can have multiple versions deployed in a side-by-side manner in the global assembly cache. |

### ***1.3.4 .Assembly Cache:***

The assembly cache is a directory normally found in the \WinNT\assembly directory. When an assembly is installed of the machine, it can be merged into the assembly cache. The assembly cache has 2 separate caches: a global assembly cache and a transient assembly cache. When assemblies are downloaded to the local machine using Internet explorer, the assembly is automatically installed in the transient assembly cache [2] [4]. Keeping these assemblies separated prevent a downloaded component from impacting the operation of an installed application. What might be a great feature of the assembly cache is its capability to hold multiple versions of an assembly. As an example, we'll say we have installed versions 1.0 and 1.1 of MyComponents.dll on a system. If an application was built and tested using version 1.0 of MyComponent.dll, even though a later version a later version of an assembly exists in the cash. The application will continue to work normally because the code that is executing is the same code that it is executing is the same code that was built and tested with. Once registered .NET takes a copy from the component and places it inside the global cash assembly to make it available to various different applications running on the machine [2] [4].

### ***<u>1.3.5 Procedure for creating a strong assembly key:</u>***

The command line strong name tool (sn.exe) that comes with the .NET framework is used to build strong name key files. For example the following command will build a strong-name key file named CalcKey.snk: sn – k. In the assembly file of the application 2 directives must be added [assembly: AssemblyKeyFile (<FileName>)], [assemblyVersion (<version number>)] (C# syntax) before the application is recompiled to produce the strong named assembly [4]. Once an assembly is strong named, it is ready to be installed in the global assembly cache to be shared among different .NET application running on the machine. Gacutil.exe is a command line utility that allows viewing and manipulating the contents of the global assembly cache. Gacutil.exe options includes:

/i or –i installs a strong-named assembly into the global assembly cache.

/l or –l lists the contents of the global assembly cache.

/u or –u removes one or more assemblies from the global assembly cache.

NB: During development, Gacutil.exe can be used to install the assembly in the global cache for testing purpose. This tool is available for convenience only and should be used for production deployment. For production environments, assemblies should be installed in the global assembly cache by using windows installer included in .NET platform [4].

## ***1.3.6 Main differences between COM and .net objects:***

| Characteristic | COM | .NET framework |
|---|---|---|
| Identity | Globally unique identifiers: (GUID) identifies a specific unmanaged type | Strong keys are used to uniquely identifies the component |
| Object lifetime management | Reference counting. Client of COM objects manage lifetime by means of reference counting | Garbage collection: The CLR runtimes manages the lifetime of objects by means of garbage collection |
| Registration mode | Are registered in the operating system registry. | Stored in the global assembly cache. |
| Interface – GUID relation | COM interface are immutable. If the name is changed, the GUID should also changed | Component can evolve retaining the same strong name |
| Versioning | The registry of the OS can only store on unique version of a COM | Several Versions of an assembly can coexists in the global assembly cache. |

# ***Chapter II:***

## ***2.1 Proposed solution template and implementation:***

This thesis proposes to build applications by using a component-oriented approach. The thesis solution will finally lead to an easy-to-update-and deploy product composed of several parts build by different software houses and wired together to behave in harmony and coordination as if they were a unique block of software controlled by a unique runtime. This template can be used to build component-based applications in fields that usually require the combination of effort of several software houses such as:

Image processing applications: The component side will be responsible of pixel blurring. Each time a certain number of pixels are processed a notification containing the partially processed picture, the number of blurred pixels and a basic type is sent to the client. The latter can use the returned data to inform the user about the state of the operation. After receiving a notification from the component side the client can stop the process by setting the basic type sent to STOPVALUE.

Directory service applications: A component side will contain a recursive procedure capable of browsing all the files and directories contained in a given location, each time a file is read a notification containing the filename and path is sent to the client. The latter can subject the returned data to any kind of service: virus scanning, image comparison, grammar spelling and checking etc … . This service can be embedded on the client side or inside another component.

Search Applications: A component side will contain a recursive function capable of searching all the tree nodes. Each time a node is expanded, a notification containing the node value and a basic type is sent to the client. The latter send the returned value to another component capable of checking if it corresponds to the goal value, in case it does the client stop the process by setting the basic type to STOPVALUE

Security Systems applications: A component side will contain functions capable of filming what is happening in a bank entrance. Each time a person enters, a notification containing the picture is sent to another component capable of accessing a database to check if this client has an image inside the "WANTED" database table. In case it exists the component alarms the security team.

### ***2.1.1 Procedure for achieving the thesis goal:***

#### ***2.1.1.1 Using a unique framework:***

Since components in absolute are heterogeneous pieces of software written and compiled by third parties. The first step toward achieving the interactivity between the different pieces of a system is to guarantee a high coupling between different involved parties. Such coupling can only be obtained by using classes and libraries belonging to the same framework where all the elements are built using the same technology and can be easily wired together [1].

#### ***2.1.1.2 Creating a notification system:***

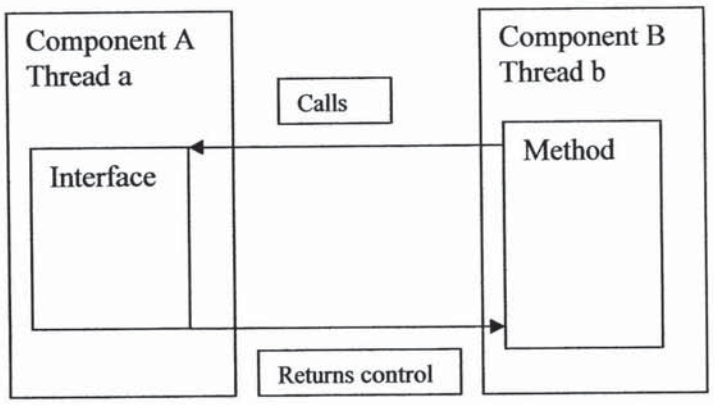
Because each component runs in its own thread and acts independently from the rest of the application the next step towards obtaining homogenous final software is the creation of a notification system between different involved parts: the proposed mechanism should be capable of transferring control from one thread to another and passing information between different components.

The thesis solution proposes to create a C# a notification system by instantiating a delegate having the same signature that the event to be fired [2] [4].

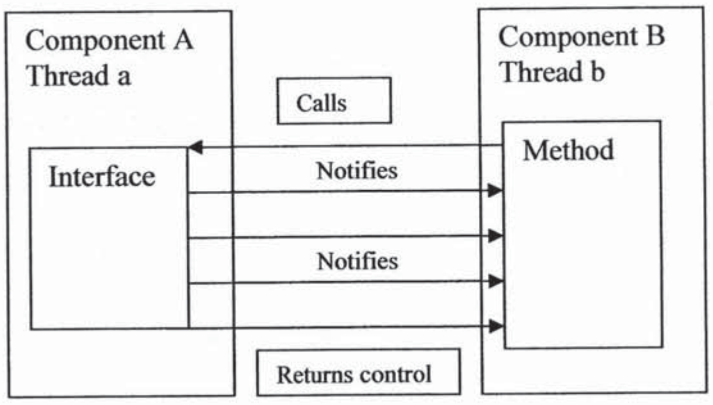
Creating a delegate in C# is similar to a function pointer in C or C++. Using a delegate allows to encapsulate a reference to a method inside a delegate object. The delegate object can then be passed to code, which can call the referenced method, without having to know at compile time which method will be invoked. A delegate declaration defines a type that encapsulates a method with a particular set of arguments and return type. An interesting and useful property of a delegate is that it does not know or care about the class of the objects that it references. Any object will do; all that matters is that the method's arguments type and return type match the delegate's. This makes delegates perfectly suited for anonymous invocation [2] [4].

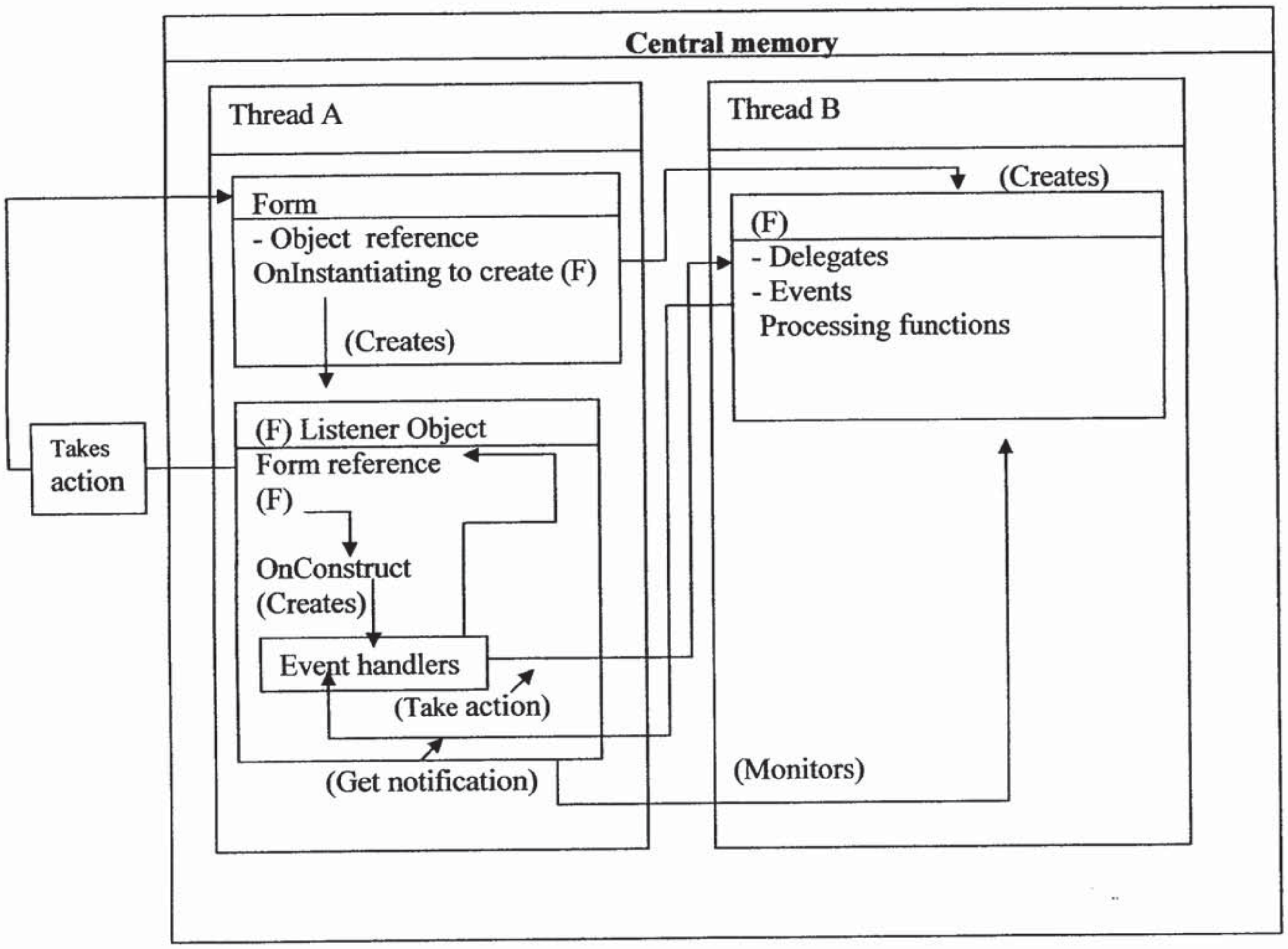
An event in C# is a way for a class to provide notifications to the clients of that class when some interesting things happen to an object. Events provide a generally useful way for objects to signal state changes that might be useful to client of that object. Events are an important building block for creating classes that can be reused in a large number of different programs. In C#, events may be declared by using delegates [2] [4].

**Isolated Components**



**Communicating components**




Central memory
Thread A
Thread B
Form
- Object reference
OnInstantiating to create (F)
(Creates)
(F) Listener Object
Form reference
(F)
OnConstruct
(Creates)
Event handlers
(Take action)
(Get notification)
Takes action
(Creates)
(F)
- Delegates
- Events
Processing functions
(Monitors)

#### ***2.1.1.2.1 Remotely accessing assemblies:***

Using private components:

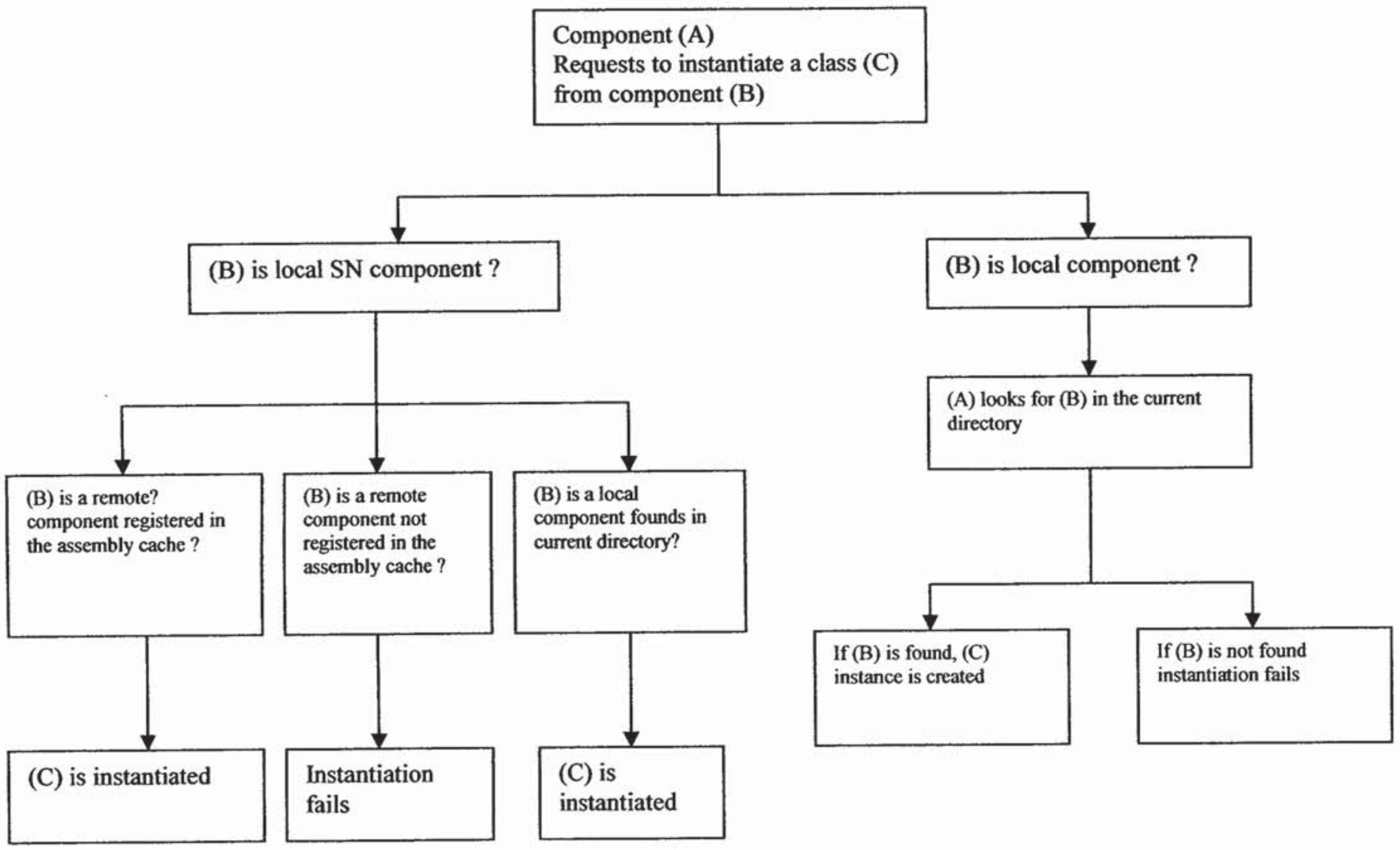
The ExtendedImageProcessing Component that is part of the Image Processing application that accompanies this thesis has been packaged using installer 2.0 and setup on a network server. On the other side the client has been placed on another network location and modified to point the needed components. After recompiling the client, it has been noticed that the latter has failed to access and instantiate component's objects unless a copy of the component was locally placed inside the current directory of the executable. The only way to create interaction between the different components was to place them in the same directory and on the same machine.

Using strong name components:

This experience has been remade but instead of using private components, strong name components has been created and registered in the assembly cache using the process described in a previous section of the thesis. It has been noticed that even though the application is physically distributed on several computers' network the interaction between different components took place successfully as if the whole system was installed on the same machine: used delegates, events, events listeners successfully cooperated together.

Conclusions:

In case a component is especially built to support a unique application, using local name assemblies sounds like a good option because when deployed all components are placed inside the same directory which make them easy to handle and maintain. In case a component is build to support several applications and to guarantee location transparency, strong named assemblies should be used.


Component (A) Requests to instantiate a class (C) from component (B)
(B) is local SN component ?
(B) is local component ?
(A) looks for (B) in the current directory
(B) is a remote? component registered in the assembly cache ?
(B) is a remote component not registered in the assembly cache ?
(B) is a local component founds in current directory?
If (B) is found, (C) instance is created
If (B) is not found instantiation fails
(C) is instantiated
Instantiation fails
(C) is instantiated

### *2.1.1.3 Overcoming language boundary:*

A language boundary is probably one of the most important handicaps faced when updating a component. The thesis solution proposes to use .NET platform to overcome the language problem. Any component written by using .NET can be extended by using C#.net, C++.net, VB.net, J#.net. Implementing this feature in a software component increases its flexibility and makes it more ready to evolve and be enhanced.

### *2.1.1.4 Overcoming versioning problem:*

The operation system registry can't accommodate more than one component version. In real cases, installing a new version of a component causes that software relaying on the previous component fails to run. The thesis solution proposes to use strong assembly components registered in the global assembly cache. The procedure for creating a strong assembly is described in detailed in a previous part of this document.

### 2.1.2 Implementation example:

The Image processing application that accompanies this thesis is an implementation of the proposed solution and can be used as a template to build interactive-object-oriented application. It has the following characteristics.

While processing an image, the component send data about the status of the operation, this information is going to be used by the client to update the screen and to supply the user with information. Each time 10 pixels are processed, the component triggers a notification containing:

| ***Parameter*** | ***Description*** |
|---|---|
| TotalNbrPixel | A variable containing the total number of pixels contained inside the picture sent for embossing. |
| e | An instance of ImageServiceArgs. It allows the client to interact with the component |
| PixelProcessed | A variable containing the number of processed pixels |
| Pic | The partially processed bitmap object |

To following delegate is used as a template that can be instantiated to any function capable of handling the event. The signature of the delegate and the instanciated function should have the same parameter type.

```
public delegate void PixelProcessedDelegate
(int PixelProcessed ,
ImageServiceArgs e , double TotalNbrPixel
, System.Drawing.Bitmap Pic);
```

The following code is a function compatible (same data type) with the above-described delegate.

```
public static void EventHandler(int PixelProcessed ,
ImageServiceArgs e , double TotalNbrPixel ,
System.Drawing.Bitmap Pic) {}
```

The following code instantiates the delegate to create the event.

```
public event PixelProcessedDelegate
PixelProcessedEvent
= new PixelProcessedDelegate(EventHandler);
```

Each time 10 pixels are processed the event is triggered and the above-described information are passed to the client, the latter can set the value of e to true the stop the object.

```
if (EmbossProgress % 10 == 0)
{
      PixelProcessedEvent(EmbossProgress , e ,
      this.InitialPicture.Width *
      this.InitialPicture.Height ,temp);
}
```

After notifying the client the component checks the value of e to take the corresponding action

```
if (e.Cancel == true)
{
      EndEmbossOperation = DateTime.Now;
      return(temp);
}
```

In order to make the interaction between the client side and the component side possible, it is important that a basic type is periodically sent from the component side to the client side. When the callback is over, the component checks the value the sent variable, if the value is true the process is aborted, if not the process continues. Under .Net, parameters are sent by value which means that changes made on them do not persist. To make the client modifications persistent, the basic type should be placed inside a class and a reference of that class sent as parameter.

When an image processing is constructed, a listener object is directly created. += sign appearing in the code is used to say that the following statement is a handler creator. On the other hand -= can be used to destroy an object event handler. The segment of code placed within the body of Srv_pixelSmoothedEvent is triggered whenever the component throws a notification. At each time the component processes 10 pixels the client uses the sent information to update some display elements: The title of the form is always modified to show the process progress and the initial bitmap is always updated by the intermediary processed bitmap. At any given time the client is capable of interrupting the component process. We can see in the above example that whenever the number of processed picture is >= 60000 or whenever the client choose to explicitly interrupt the component the process is aborted.

```
public class ImageProcessingEventListener
{
        PictureForm FormInst;
        private bool stop = false;
        public void Stop()
        {
                stop = true;
        }
        public ImageProcessingEventListener
                (PictureForm Inst,
                ExtendedImageProcessing.ImageService Srv)

        {
         FormInst = Inst;
         Srv.PixelProcessedEvent+=
         new DllCSharpImageProcessing.
         ImageService.PixelProcessedDelegate
         (Srv_PixelProcessedEvent);
         Srv.PixelSmoothedEvent+=new
         ExtendedImageProcessing.ImageService
        .PixelSmoothedEventEventHandler(Srv_PixelSmoothedEvent);
        }

        private void Srv_PixelProcessedEvent // Embossing event handling
        (int PixelProcessed,
         DllCSharpImageProcessing.ImageServiceArgs e,
         double TotalNbrPixel, Bitmap Pic)
        {
                FormInst.Text = "Progress:" +
                Convert.ToString(PixelProcessed) + "/"
                + Convert.ToString(TotalNbrPixel);
                FormInst.pictureBox1.Image = Pic;
                FormInst.pictureBox1.Refresh();
                if (stop == true) e.Cancel = true;
                //if (PixelProcessed >= 60000) e.Cancel = true;
        }

        private void Srv_PixelSmoothedEvent(int PixelProcessed,
                DllCSharpImageProcessing.ImageServiceArgs e,
                double TotalNbrPixel,
                Bitmap Pic) // Smoothing event handling
                {
```

```
                FormInst.Text = "Progress:" +
                Convert.ToString(PixelProcessed) + "/"
                + Convert.ToString(TotalNbrPixel);
                FormInst.pictureBox1.Image = Pic;
                FormInst.pictureBox1.Refresh();
                if (stop == true) e.Cancel = true;
                     //if (PixelProcessed >= 60000) e.Cancel = true;
               }
        }
}
```

This application also solves to some extend the problem of language dependency since it can be extended an enriched using any language supported by the .NET platform. The basic class DllCSharpImageProcessing is originally written using C#, later it was extended using using VB.NET and finally the wiring of components takes place inside a C# application.

The involved components are registered in the global assembly cache in order to create a unique copy available for any application running on the machine. The process of registering an assembly inside the global cache is explained in a previous part of this thesis.

| *Assembly Name* | *Public Key* | *Version* | *Used Language* |
|---|---|---|---|
| *DllCsharpImageProcessing* | *648164a407ff915a* | *1.0.0.0* | *C#* |
| *ExtendedImageProcessing* | *286c5b671ce09698* | *1.0.0.0* | *VB.NET* |

In case one of these assemblies has been upgraded, the new version is not going to replace the current one; instead they both are going to coexist in the cache under the same assembly name and public key but under different version numbers. This way will be guaranteed that whenever an application works properly in the development environment it is going to work properly after deployment. For a full implementation of this solution check the code that accompanies the thesis.

## ***2.1.3 Conclusions and results:***

Components oriented applications are by nature client server applications, if no messaging system is implemented no intermediate result could be shown on the client side, to obtain information the user should wait till the control is returned back to him. To stop the process before it reaches its end the user should use the hard way by forcing the operating system to kill the application.

In the image-processing example, if notifications were not part of the structure of the application, it would have been impossible to show how the blurring is progressing. The client side would have to wait till the end of the process to see the result; this means that the user can only see 2 object states, the initial one and the final one.

Without implementing a notification system, the image processing application couldn't have been build as a component oriented application. The only means to proceed would have been to implement it as a unique bloc and sacrificing all the flexibility that object oriented programming offers.

The image processing application has been built using 2 components: a first one playing the role of the super class and written by a software house specialized in C# and a second playing the role of the sub class and developed by a software house specialized in VB.net. This special and unique feature increases considerably the flexibility of components and is a first step toward creating a "universal component"

This application would continue to function even if a new version of the component has been installed on the system. This is doubtlessly a solution to old dlls versioning problem.

## ***2.3 Future Work:***

This thesis found some solutions to several common component problems and proposed a .NET implementation template. But Java is also an important component standard available on the market: EJB technology is used by a very important number of software builders due to the flexibility it offers.

### Entity java beans overview:

*It is permanent:* Standard Java objects come into existence when they are created in a program. When the program terminates, the object is lost. But an entity bean stays around until it is deleted. A program can create an entity bean, then the program can be stopped and restarted. The entity bean will continue to exist. After being restarted, the program can again find the entity bean it was working with, and continue using the same entity bean [6].

*It is network based:* Standard Java objects are used only by one program. But an entity bean can be used by any program on the network. The program just needs to be able to find the entity bean, in order to use it [6].

*It is executed remotely:* Methods of an entity bean run on a "server" machine. When you call an entity bean's method, your program's thread stops executing and control passes over to the server. When the method returns from the server, the local thread resumes executing [6].

*It is identified by a primary key:* Entity Beans must have a primary key. The primary key is unique -- each entity bean is uniquely identified by its primary key. For example, an "employee" entity bean may have Social Security numbers as primary keys. You can only use entity beans when your objects have a unique identifier field, or when you can add such a field [6].

### Session beans overview:

Session beans can be used to distribute and isolate processing tasks, somewhat analogous to the way each Java class can be used to encapsulate a type of related processing. Each session bean can be used to perform a certain task on behalf of its client. The tasks can be distributed on different machines [6].

Another way session beans can be thought of, is like how browsers and web-servers operate. A web-server is located in a particular location, but multiple browsers can connect to it and get it to perform services (such as delivering HTML pages) on their behalf. Each server performs a specialized unique task (for instance, ejbtut.com performs the specialized task of providing EJB tutorial material!) The clients can connect to any of a number of servers, depending upon their needs [6].

*Finding a way using EJB technology to overcome the above mentioning problems to reach an easy to manage component-oriented solution constitute a future extension of this thesis.*

## References:

[1] Component Software Beyond Object-Oriented Programming, Clemens Szyperski , Addison-Wesley

[2] C#.NET Web Developer's Guide, Adrian Turtchi, Jason Werry, Greg Hack , Joseph Albahari , Syngress

[3] Mastering VB.NET Evangelos Petroutsos , Sybex edition.

[4] http://msdn.microsoft.com/library/.

[5] http://www.objs.com/survey/ComponentwareGlossary.htm.

[6] http://www.ejbtut.com/Overview.jsp.

# Appendix: Fragment of the source code:

**DllChsarpImageprocessing component:**

```
namespace DllCSharpImageProcessing
{
        public class ImageServiceArgs //Argument Class
        {
                public bool Cancel = false;
        }


        public class ImageService
        {
                public override string ToString()
                {
                        return "CSharpImageProcessing" +
                                "ImageService Class : NDU MS Thesis";
                }

                protected System.Drawing.Bitmap InitialPicture;

                public delegate void PixelProcessedDelegate
                        (int PixelProcessed ,
                        ImageServiceArgs e     , double TotalNbrPixel
                        , System.Drawing.Bitmap Pic);

                public event PixelProcessedDelegate
                        PixelProcessedEvent
                        = new PixelProcessedDelegate(EventHandler);

                private DateTime StartEmbossOperation;
                private DateTime EndEmbossOperation;



                public int GetEmbossOperationTime
                {
                        get
                        {
                                TimeSpan dur =
                                        EndEmbossOperation.Subtract
                                        (StartEmbossOperation);
                                return (dur.Milliseconds +
                                        dur.Minutes * (60000)
                                        + dur.Seconds * 1000);
                        }
                }

                public static void EventHandler(int PixelProcessed ,
                        ImageServiceArgs e , double TotalNbrPixel
                        , System.Drawing.Bitmap Pic) {}

                public ImageService(System.Drawing.Bitmap img)
                {
                        InitialPicture = img;
                }

                public System.Drawing.Bitmap Emboss()
                {
                        System.Drawing.Bitmap temp =
                        this.InitialPicture;
                        int EmbossProgress  = 0;
                        ImageServiceArgs e = new ImageServiceArgs();

                        StartEmbossOperation = DateTime.Now;

                        for (int y=0 ;y <= this.InitialPicture.Height - 1;y++)
                        {
```

```
                        for (int x=0;x <= this.InitialPicture.Width -2;x++)
                        {
                                System.Windows.Forms.Application.DoEvents();

                                int red =
                                        Math.Max
                                        (Math.Min
                                        (this.InitialPicture.GetPixel(x+1,y).R -
                                        this.InitialPicture.GetPixel
                                        (x,y).R + 128 , 255),0);

                                int green =
                                        Math.Max(Math.Min(this.InitialPicture
                                        .GetPixel(x+1,y).G -
                                        this.InitialPicture.GetPixel
                                        (x,y).G + 128 , 255),0);

                                int blue =
                                        Math.Max(Math.Min(
                                        this.InitialPicture.GetPixel
                                        (x+1,y).B -
                                        this.InitialPicture.GetPixel
                                        (x,y).B + 128
                                        , 255),0);


                                temp.SetPixel(x,y,System.Drawing.Color.FromArgb
                                        (red,green,blue));
                                EmbossProgress++;

                                if (EmbossProgress % 10 == 0)
                                {
                                        PixelProcessedEvent(EmbossProgress , e ,
                                        this.InitialPicture.Width *
                                        this.InitialPicture.Height temp);}

                                if (e.Cancel == true)
                                {
                                        EndEmbossOperation =
                                        DateTime.Now;return(temp);}

                        }

                }
                PixelProcessedEvent(EmbossProgress , e ,
                        this.InitialPicture.Width *
                        this.InitialPicture.Height ,temp);
                EndEmbossOperation = DateTime.Now;
                return(temp);

        }
    }
}
```

**ExtendedImageProcessing component:**

```
Public Class ImageService
    Inherits DllCSharpImageProcessing.ImageService
    Event PixelSmoothedEvent(ByVal PixelProcessed As Integer _
    , ByVal e As DllCSharpImageProcessing.ImageServiceArgs _
    , ByVal TotalNbrPixel As Double, ByVal Pic As Bitmap)
    Private StartSmoothingOperation As Date
    Private EndSmoothingOperation As Date


    Public Sub New(ByVal Img As Bitmap)
        MyBase.New(Img)
    End Sub

    Public ReadOnly Property GetSmoothingOperationTime() As Integer
    Get
        Dim dur As TimeSpan = _
        EndSmoothingOperation.Subtract _
        (StartSmoothingOperation)
        Return (dur.Seconds * 1000 + dur.Minutes * _
        60000 + dur.Milliseconds)
    End Get

End Property

Public Function Smoothing() As Bitmap

    Dim temp As Bitmap = MyBase.InitialPicture
    Dim e As New DllCSharpImageProcessing.ImageServiceArgs
    StartSmoothingOperation = DateTime.Now()

    Dim x As Integer = 0
    Dim y As Integer = 0
    Dim SmoothingProgess As Integer = 0

    For y = 1 To temp.Height - 2

        For x = 1 To temp.Width - 2

            Application.DoEvents()

            Dim red, green, blue As Integer

            red = (CInt(Me.InitialPicture.GetPixel(x - 1, y - 1).R) + _
                  CInt(Me.InitialPicture.GetPixel(x, y - 1).R) + _
                  CInt(Me.InitialPicture.GetPixel(x + 1, y - 1).R) + _
                  CInt(Me.InitialPicture.GetPixel(x - 1, y + 1).R) + _
                  CInt(Me.InitialPicture.GetPixel(x, y + 1).R) + _
                  CInt(Me.InitialPicture.GetPixel(x + 1, y + 1).R) + _
                  CInt(Me.InitialPicture.GetPixel(x - 1, y).R) + _
                  CInt(Me.InitialPicture.GetPixel(x + 1, y).R)) / 8

            green = (CInt(Me.InitialPicture.GetPixel(x - 1, y - 1).G) + _
                  CInt(Me.InitialPicture.GetPixel(x, y - 1).G) + _
                  CInt(Me.InitialPicture.GetPixel(x + 1, y - 1).G) + _
                  CInt(Me.InitialPicture.GetPixel(x - 1, y + 1).G) + _
                  CInt(Me.InitialPicture.GetPixel(x, y + 1).G) + _
                  CInt(Me.InitialPicture.GetPixel(x + 1, y + 1).G) + _
                  CInt(Me.InitialPicture.GetPixel(x - 1, y).G) + _
                  CInt(Me.InitialPicture.GetPixel(x + 1, y).G)) / 8

            blue = (CInt(Me.InitialPicture.GetPixel(x - 1, y - 1).B) + _
                   CInt(Me.InitialPicture.GetPixel(x, y - 1).B) + _
                   CInt(Me.InitialPicture.GetPixel(x + 1, y - 1).B) + _
                   CInt(Me.InitialPicture.GetPixel(x - 1, y + 1).B) + _
                   CInt(Me.InitialPicture.GetPixel(x, y + 1).B) + _
                   CInt(Me.InitialPicture.GetPixel(x + 1, y + 1).B) + _
                   CInt(Me.InitialPicture.GetPixel(x - 1, y).B) + _
                   CInt(Me.InitialPicture.GetPixel(x + 1, y).B)) / 8

            temp.SetPixel(x, y, Color.FromArgb(red, green, blue))

            SmoothingProgess = SmoothingProgess + 1

            If (SmoothingProgess Mod 10 = 0) Then

                RaiseEvent PixelSmoothedEvent(SmoothingProgess, e, _
                 Me.InitialPicture.Width * Me.InitialPicture.Height, temp)
            End If

            If (e.Cancel = True) Then
                EndSmoothingOperation = _
                DateTime.Now
                Return (temp)
            End If
            Next

        Next

        EndSmoothingOperation = Now()
        Return (temp)

    End Function


End Class
```

**Client event listener:**

```
public class ImageProcessingEventListener
{
        PictureForm FormInst;
        private bool stop = false;
        public void Stop()
        {
                stop = true;
        }
        public ImageProcessingEventListener
                (PictureForm Inst,
                ExtendedImageProcessing.ImageService Srv)

        {
         FormInst = Inst;
         Srv.PixelProcessedEvent+=
         new DllCSharpImageProcessing.
         ImageService.PixelProcessedDelegate
         (Srv_PixelProcessedEvent);
         Srv.PixelSmoothedEvent+=new
         ExtendedImageProcessing.ImageService
        .PixelSmoothedEventEventHandler(Srv_PixelSmoothedEvent);
        }

        private void Srv_PixelProcessedEvent // Embossing event handling
        (int PixelProcessed,
         DllCSharpImageProcessing.ImageServiceArgs e,
         double TotalNbrPixel, Bitmap Pic)
        {
                FormInst.Text = "Progress:" +
                Convert.ToString(PixelProcessed) + "/"
                + Convert.ToString(TotalNbrPixel);
                FormInst.pictureBox1.Image = Pic;
                FormInst.pictureBox1.Refresh();
                if (stop == true) e.Cancel = true;
                //if (PixelProcessed >= 60000) e.Cancel = true;
        }

        private void Srv_PixelSmoothedEvent(int PixelProcessed,
                DllCSharpImageProcessing.ImageServiceArgs e,
                double TotalNbrPixel,
                Bitmap Pic) // Smoothing event handling
                {
                 FormInst.Text = "Progress:" +
                 Convert.ToString(PixelProcessed) + "/"
                 + Convert.ToString(TotalNbrPixel);
                 FormInst.pictureBox1.Image = Pic;
                 FormInst.pictureBox1.Refresh();
                 if (stop == true) e.Cancel = true;
                        //if (PixelProcessed >= 60000) e.Cancel = true;
                }
        }

  }
```