# .NET – Definindo a arquitetura de um projeto de software

Fonte: <http://imasters.com.br/framework/dotnet/net-definindo-a-arquitetura-de-um-projeto-de-software/>

Se você trabalha com desenvolvimento de software, por experiência própria, deve saber que a única constante no processo de desenvolvimento é a mudança.

Mas por que isso ocorre? Com o desenvolvimento tecnológico obtidos nos últimos tempos a área da tecnologia da informação foi impactada por dois fatores importantes:

1. O aumento do tamanho e da complexidade dos sistemas de software;
2. A redução do tempo e custo de desenvolvimento e manutenção do software.

Assim, os sistemas ficaram mais complexos e maiores e o tempo e o orçamento para desenvolver tais sistemas diminuiu. E o negócio não pode parar…

Um software é um produto que foi criado para atender uma necessidade de negócio, e os requisitos de negócio mudam a todo instante com base nas prioridades do mesmo, logo o software tem que ser alterado para dar suporte a tais mudanças. Dessa forma, um software que não foi bem projetado para dar suporte às constantes mudanças irá se tornar obsoleto e não vai mais cumprir o seu objetivo afetando os negócios.

Então ‘o negócio’ é projetar um software que seja fácil de ajustar e barato de manter. Parece simples não é mesmo? Tudo começa com a definição da arquitetura de software a ser adotada. Uma das tarefas fundamentais no processo de criação de um software é a definição da arquitetura do projeto de software pois uma arquitetura consistente e bem definida se torna fundamental para que o projeto seja implementado com eficiência.

A arquitetura de software desempenha um papel fundamental para gerenciar a complexidade inerente ao software a ser criado. Uma boa arquitetura possibilita que um sistema satisfaça às exigências principais de um projeto, tais como: desempenho, confiabilidade, portabilidade, manutenibilidade, interoperabilidade e etc.

Uma má arquitetura simplesmente fará com que o software seja um fracasso.

## O que é arquitetura de software ?

Existem muitas definições de arquitetura de software:

*“Uma arquitetura de software envolve a descrição de elementos arquiteturais dos quais os sistemas serão construídos, interações entre esses elementos, padrões que guiam suas composições e restrições sobre estes padrões”. GARLAN, 2000*

*“A arquitetura de software define o que é o sistema em termos de componentes computacionais e os relacionamentos entre estes componentes”. VAROTO, 2002*

*“A arquitetura de software de um sistema consiste na definição dos componentes de software, suas propriedades externas, e seus relacionamentos com outros softwares. O termo também se refere à documentação da arquitetura de software do sistema. A documentação da arquitetura do software facilita: a comunicação entre os stakeholders, registra as decisões iniciais acerca do projeto de alto-nível, e permite o reuso do projeto dos componentes e padrões entre projetos”.*[*http://pt.wikipedia.org/wiki/Arquitetura\_de\_software*](http://pt.wikipedia.org/wiki/Arquitetura_de_software)*, acessado em 2012*

**Obs**: Stakeholder é qualquer pessoa ou organização que tenha interesse, ou seja afetado pelo projeto. (Stake: interesse, participação, risco / Holder: aquele que possui)

Assim, a arquitetura de um sistema deve definir os elementos que irão compor o software. Tais elementos definem como o software é particionado em pedaços menores e, assim, definem como o software é entendido.

## Como definir uma boa arquitetura?

Segundo o RUP – Rational Unified Process, a arquitetura de um software envolve o conjunto de decisões que definem a organização do sistema e devem cumprir os seguintes objetivos:

* Definir os elementos estruturais e suas interfaces de modo a estabelecer a composição do sistema;
* Estabelecer o comportamento pela colaboração entre estes elementos;
* Compor estes elementos estruturais e comportamentais em subsistemas (agregação).

Diversos fatores influenciam a definição da arquitetura como por exemplo:

* Arquitetura do hardware;
* Sistema operacional utilizado;
* Sistema Gerenciador de Banco de dados adotado;
* Protocolos de rede;
* A linguagem de programação;
* O ambiente de interface gráfica;
* As bibliotecas de funções disponíveis;
* Os sistemas legados envolvidos;
* As necessidades de desempenho, portabilidade, usabilidade, disponibilidade, etc;
* Documentação.

Levando em conta esses fatores e sendo bem objetivo e prático, como podemos definir uma arquitetura básica que possa ser implementada em sistemas de pequeno é médio porte?

## Por onde começar?

Comece pelo começo…

### 1. Defina os requisitos do sistema

Definir de forma sistemática os requisitos do sistema envolvendo todos os stackholders (interessados) no projeto de forma a ter o maior número possível de requisitos definidos de forma clara e objetiva;

Os requisitos expressam as características e restrições do produto de software do ponto de vista de satisfação das necessidades do usuário, e, em geral independem da tecnologia empregada na construção da solução sendo a parte mais crítica e propensa a erros no desenvolvimento de software.

Requisitos são objetivos ou restrições estabelecidas por clientes e usuários do sistema que definem as diversas propriedades do sistema. Os requisitos  de software são, obviamente, aqueles dentre os requisitos de sistema que dizem respeito a propriedades do software.

A necessidade de se estabelecer os requisitos de maneira de forma precisa é crítica na medida que o tamanho e a complexidade do software aumentam. Os requisitos exercem influência uns sobre os outros. Por exemplo, o requisito de que o software de ter grande portabilidade (por exemplo, ser implementado em Java) pode implicar em que o requisito desempenho não seja satisfeito (programas em Java são, em geral, mais lentos).

Uma boa especificação de requisitos deve ser:

* Clara e não ambígua;
* Completa;
* Correta;
* Compreensível;
* Consistente;
* Concisa;
* Confiável;.De acordo com [Farley](http://www.dimap.ufrn.br/~jair/ES/home.html#biblio), um documento de especificação de requisitos deve possui as seguintes seções:
* Visão geral do produto e sumário;
* Ambientes de desenvolvimento, operação e manutenção;
* Interfaces externas e fluxo de dados;
* Requisitos funcionais;
* Requisitos de desempenho;
* Tratamento de exceções;
* Prioridades de implementação;
* Antecipação de mudanças e extensões;
* Dicas e diretrizes de design;
* Critérios de aceitação;
* Índice remissivo;
* Glossário.

### 2. Definir uma ferramenta ORM

De forma geral, você pode escolher uma ferramenta ORM para ajudá-lo a criar os objetos com base no seu domínio e a partir deles gerar o banco de dados usado pela aplicação.

Existem dezenas de ferramentas ORMs disponíveis no mercado. Abaixo uma relação das mais importantes para a plataforma .NET:  
(fonte: <http://en.wikipedia.org/wiki/List_of_object-relational_mapping_software>)

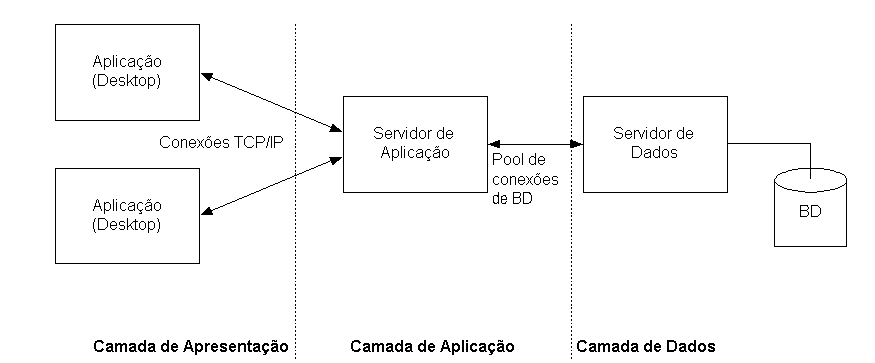
* [ADO.NET Entity Framework](http://en.wikipedia.org/wiki/ADO.NET_Entity_Framework), included in .NET Framework 3.5 SP1 and above
* [AgileFx](http://en.wikipedia.org/wiki/AgileFx), open source
* [Base One Foundation Component Library](http://en.wikipedia.org/wiki/Base_One_Foundation_Component_Library), free or commercial
* [Devart](http://en.wikipedia.org/wiki/Devart_%28company%29) LinqConnect, commercial, an ORM solution for [Oracle](http://en.wikipedia.org/wiki/Oracle_Database), [MySQL](http://en.wikipedia.org/wiki/MySQL), [PostgreSQL](http://en.wikipedia.org/wiki/PostgreSQL), and [SQLite](http://en.wikipedia.org/wiki/SQLite)
* [Castle ActiveRecord](http://en.wikipedia.org/wiki/Castle_ActiveRecord), ActiveRecord for .NET, open source
* [DatabaseObjects](http://en.wikipedia.org/wiki/DatabaseObjects) .NET, open source
* [DataObjects.NET](http://en.wikipedia.org/wiki/DataObjects.NET), commercial
* [ECO](http://en.wikipedia.org/wiki/ECO_%28Domain_Driven_Design%29), commercial but free use for up to 12 classes
* [EntitySpaces](http://en.wikipedia.org/wiki/EntitySpaces), commercial
* [Habanero](http://en.wikipedia.org/wiki/Habanero.NET), free open source enterprise application framework with a free code generating tool
* [MyBatis](http://en.wikipedia.org/wiki/MyBatis), free open source, formerly named iBATIS
* [iBATIS](http://en.wikipedia.org/wiki/IBATIS), free open source, maintained by [ASF](http://en.wikipedia.org/wiki/Apache_Software_Foundation) but now inactive.
* [LINQ to SQL](http://en.wikipedia.org/wiki/Language_Integrated_Query#LINQ_to_SQL), included in .NET Framework 3.5
* [LLBLGen Pro](http://en.wikipedia.org/wiki/LLBLGen_Pro), commercial
* [Neo](http://en.wikipedia.org/wiki/Neo_%28object-relational_toolset%29), open source but now inactive.
* [NHibernate](http://en.wikipedia.org/wiki/NHibernate), open source
* [nHydrate](http://en.wikipedia.org/wiki/NHydrate), open source
* [OpenAccess ORM](http://en.wikipedia.org/wiki/OpenAccess_ORM), by Telerik free or commercial
* [Persistor.NET](http://en.wikipedia.org/wiki/Persistor.NET), free or commercial
* [Quick Objects](http://en.wikipedia.org/wiki/Quick_Objects), free or commercial
* [Signum Framework](http://en.wikipedia.org/wiki/Signum_Framework), open source
* [SubSonic](http://en.wikipedia.org/wiki/SubSonic_%28software%29), open source
* [Symbiotic](http://en.wikipedia.org/wiki/Symbiotic_%28software%29), by Frozen Elephant Inc.
* [XPO](http://en.wikipedia.org/wiki/XPO), commercial, by DevExpress

### 3. Aplique a separação das responsabilidades ao seu projeto de software

A definição da separação em camadas estimula a organização da arquitetura do sistema em um conjunto de camadas coesas com fraco acoplamento entre elas onde cada camada possui um propósito bem definido.

Principais camadas:

* UI(camada de apresentação): agrega as classes do sistema com as quais os usuários interagem;
* Negócio: mantém as classes do sistema responsáveis pelos serviços e regras do negócio;
* Dados: camada responsável pelo armazenamento e recuperação dos dados persistentes do sistema;
* Comunicação: responsável pela distribuição do sistema em várias máquinas.

[](http://imasters.com.br/wp-content/uploads/2013/01/net_arq4.gif)

Considerações:

* Seja explícito sobre como as camadas se comunicam entre si – Permitir que todas as camadas em um aplicativo se comuniquem ou tenham dependências sobre todas as outras camadas resultará em uma solução que é mais difícil de entender e gerenciar;
* Use a abstração para implementar o acoplamento fraco entre as camadas – Isto pode ser conseguido através da definição de componentes de interface, tais como uma Facade com entradas e saídas bem conhecidos que traduzem os pedidos para um formato compreendido por componentes no interior da camada;
* Não misture tipos diferentes de componentes na mesma camada lógica – Comece por identificar diferentes áreas de interesse e, em seguida agrupe os componentes associados a cada área de interesse em camadas lógicas. Por exemplo, a camada de interface do usuário não deve conter componentes de processamento de negócios, mas deve conter componentes usados para manipular a entrada do usuário e solicitações dos usuários do processo;
* Mantenha o formato de dados consistente dentro de uma camada ou componente – Misturar formatos de dados irá tornar a aplicação mais difícil de implementar, ampliar e manter. Toda vez que você precisar converter dados de um formato para outro, você é obrigado a implementar o código de tradução para executar a operação e incorrendo em uma sobrecarga de processamento.

### 4. Utilize padrões de projeto de forma a obter um código robusto e obter a reutilização de código

* Mantenha os padrões de projeto consistentes dentro de cada camada – Dentro de uma camada lógica, sempre que possível, o design de componentes deve ser consistente para uma determinada operação;
* Não duplique a funcionalidade de um aplicativo – Deve haver apenas um componente proporcionando uma funcionalidade e esta funcionalidade específica não deve ser repetida em qualquer outro componente. Isso faz com que seus componentes coesos e torna mais fácil otimizar os componentes se uma determinada função ou funcionalidade sofrer alterações;
* Estabeleça uma convenção de estilo de codificação e nomeação para o desenvolvimento – Verifique se a organização estabeleceu o estilo de codificação e padrões de nomenclatura.

### 5. Adote princípios básicos de projeto de software

* Separação de Responsabilidades – Divida sua aplicação em recursos distintos com a menor sobreposição de funcionalidade possível. O fator importante é a minimização dos pontos de interação para alcançar alta coesão e baixo acoplamento;
* Princípio da responsabilidade individual- Cada componente ou módulo deve ser responsável por apenas uma característica específica ou funcionalidade, ou agregação de funcionalidade coesa;
* Princípio da menor conhecimento – (também conhecida como Lei de Deméter ou LoD). Um componente ou objeto não deve saber sobre detalhes internos de outros componentes ou objetos;
* Não repita você mesmo (DRY) – Você só precisa especificar a intenção em um só lugar. Por exemplo, em termos de projeto da aplicação, a funcionalidade específica deve ser implementada em apenas um componente; a funcionalidade não deve ser repetida em qualquer outro componente;
* Minimizar projeto inicial – Projete apenas o necessário. Em alguns casos, você pode requerer um projeto inicial abrangente com realização de testes para verificar se o custo de desenvolvimento ou uma falha no projeto será muito alto. Em outros casos, especialmente para desenvolvimento ágil, você pode evitar antecipadamente um projeto grande. Se os requisitos da aplicação não são claros, ou se existe a possibilidade do projeto evoluir ao longo do tempo, evite fazer um esforço grande prematuramente. Este princípio é conhecido como YAGNI (“Você não vai precisar dele” – You ain’t gonna need it);
* Considere a operação de sua aplicação – Determine quais métricas e dados operacionais são exigidos pela infraestrutura de TI para garantir a implantação e operação eficiente de sua aplicação.

### 6. Determinar o tipo de aplicação

Escolher o tipo de aplicação adequada é a peça chave do processo de concepção de uma aplicação. Sua escolha é governado por suas necessidades e limitações específicas de infraestrutura. Muitas aplicações devem suportar vários tipos de cliente, e podem fazer uso de mais do que um dos arquétipos básicos:

* As aplicações concebidas para dispositivos móveis;
* Aplicações rich client projetadas para funcionar primeiramente em um PC cliente;
* Aplicações ricas da Internet projetadas para serem implantadas a partir da Internet, que suportam interface rica e cenários de mídia;
* Aplicações de serviços projetados para suportar a comunicação entre os componentes fracamente acoplados;
* Aplicativos da Web concebidos para serem executados principalmente no servidor em cenários totalmente conectados.

### 7. Determinar a estratégia de distribuição

O aplicativo pode ser implantado em uma variedade de ambientes, cada um com seu próprio conjunto específico de restrições, como a separação física dos componentes em diferentes servidores, uma limitação em protocolos de rede, configurações de firewall e roteador e muito mais.

Existem vários padrões comuns de implantação, que descrevem os benefícios e as considerações para uma série de cenários distribuídos e não distribuídos. Você deve equilibrar as necessidades da aplicação com os padrões apropriados que hardware pode suportar, e as restrições que o ambiente exerce sobre as opções de implantação. Estes fatores irão influenciar o seu projeto de arquitetura.

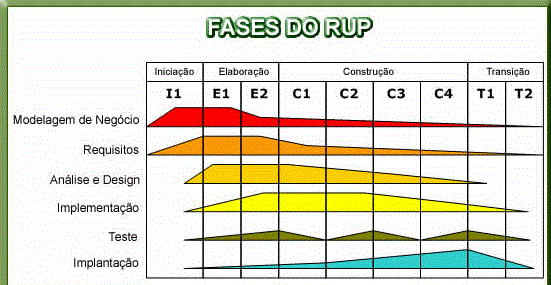
### 8. Determinar os atributos de qualidade desejados

Os atributos de qualidade, tais como segurança, desempenho e usabilidade podem ser usados para focar o seu pensamento sobre os problemas críticos que seu projeto deve resolver. Dependendo de suas necessidades, você pode ter que considerar todos os atributos de qualidade, ou talvez você só precise considerar um subconjunto. Por exemplo, cada projeto de software deve considerar sempre a segurança e o desempenho, mas o projeto talvez não precise considerar os atributos interoperabilidade e escalabilidade.

### 9. Determine um processo bem definido de desenvolvimento de software

Um exemplo muito usado a ser considerado é o Processo Unificado da Rational (RUP) que foi construído envolvendo as seis melhores práticas para fornecer um processo bem definido e foi baseado nas boas práticas de desenvolvimento de software.

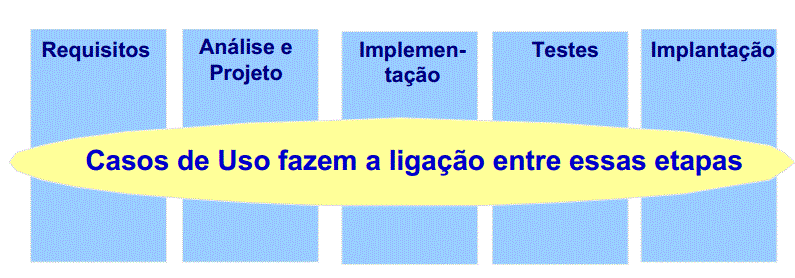
O RUP é um processo de desenvolvimento de software que assegura a produção de software de qualidade de forma repetida e previsível, com isso, se ganha tempo e facilita no desenvolvimento do projeto, visto que as tarefas são divididas de formas iterativas.

[](http://imasters.com.br/wp-content/uploads/2013/01/net_arq1.gif)[](http://imasters.com.br/wp-content/uploads/2013/01/net_arq2.gif)

O modelo do RUP é interativo e incremental onde temos que em cada iteração:

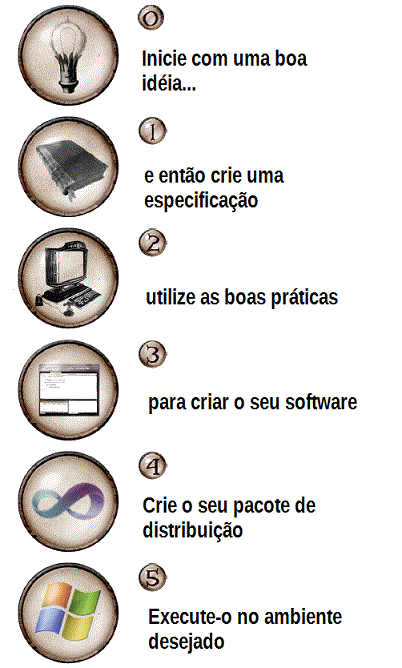
* São identificados e especificados os casos de uso mais relevantes;
* E feita a análise e projeto dos casos de uso, usando-se a arquitetura como guia;
* São implementados componentes que realizam o que foi projetado;
* Verifica-se se os componentes satisfazem os casos de uso escolhidos.

Os casos de uso são usados no planejamento e acompanhamento das iterações:

[](http://imasters.com.br/wp-content/uploads/2013/01/net_arq3.gif)

A arquitetura serve para organizar o desenvolvimento, estruturar a solução e identificar oportunidades de reuso enquanto que os casos de uso dizem o que deve ser feito.

Dessa forma, podemos resumir o processo de desenvolvimento de software em cinco etapas:

[](http://imasters.com.br/wp-content/uploads/2013/01/net_arq5.gif)