2
Conceitos da Engenharia de Software

> Engenharia de Software é a criação e a utilização de sólidos princípios de engenharia a fim de obter software de maneira econômica, que seja confiável e que trabalhe eficientemente em máquinas reais.
>
> Friedrich Ludwig Bauer

Os sistemas com base em computadores têm se tornado cada vez mais complexos. Atualmente, grande parte dos serviços e dos produtos faz uso ou incorpora computadores ou sistemas, seja para controle ou para produção e, nestes, o software representa um grande e crescente percentual do custo total. Assim, a economia mundial vem se tornando fortemente atenta à razão custo-benefício dos sistemas de software.

A Engenharia de Software é uma disciplina que busca maximizar a relação custo-benefício no desenvolvimento de sistemas de software. O fato de o software ser abstrato e intangível torna seu desenvolvimento simples, já que ele não precisa respeitar leis da física, por exemplo. Por outro lado, essa característica pode deixá-lo muito complexo e difícil de ser compreendido.

Os primeiros conceitos de Engenharia de Software datam de 1968, na chamada "crise do software", quando a terceira geração de computadores ficou disponível e seu poder de processamento era, naquela época, inimaginável. Os sistemas subseqüentes se transformaram em grandes sistemas de informação e a abordagem até então utilizada para construí-los se mostrou ineficaz (SOMMERVILLE, 2004).

2.1.
Requisitos de Software

Os desafios enfrentados ao se desenvolver um software podem ser extremamente complexos. Para compreender o universo do problema e para estabelecer o que um sistema precisa fazer é necessário descrever todas as suas funções e res-

trições. O processo de descobrir, analisar, documentar e verificar essas funções e restrições é denominado Engenharia de Requisitos.

Um requisito é, ao mesmo tempo, uma declaração abstrata e uma definição detalhada de uma funcionalidade ou restrição que o sistema deve possuir, (SOMMERVILLE, 2004). Um exemplo que facilita entender esse paradoxo é o de uma empresa que precisa estabelecer um contrato para o desenvolvimento de um grande projeto de software. Para isso, ela deve definir suas necessidades de maneira suficientemente abstrata, para que uma solução não seja predefinida. Entretanto, os requisitos devem ser rígidos de forma que os diversos fornecedores apresentem suas propostas, podendo até oferecer diversas maneiras de atender às necessidades do cliente. A partir do contrato, o fornecedor precisa preparar uma definição de sistema com mais detalhes para que o cliente compreenda e valide o software a ser feito. Esse dois documentos são chamados de Documentos de Requisitos do Sistema (DAVIS, 1993).

Os requisitos de sistema são classificados, normalmente, como funcionais, não funcionais e de domínio. Os requisitos funcionais definem **o que** o sistema deve ou não deve fazer e **como** o sistema deve reagir às diferentes situações de uso. Os requisitos não funcionais são restrições que o sistema deve respeitar. Por exemplo, restrições de desempenho, de padrões, de segurança, etc. Os requisitos de domínio se referem ao domínio de aplicação do sistema e podem ser funcionais ou não funcionais (SOMMERVILLE, 2004).

2.2. Arquitetura de Software

Normalmente, um sistema de médio ou grande porte necessita ser subdividido para que seu desenvolvimento seja simplificado. O processo de identificar os subsistemas, produtos dessa divisão, e estabelecer um arcabouço para o controle e a comunicação entre eles é chamado de Projeto de Arquitetura. A saída desse processo é uma descrição da arquitetura do software.

O Projeto de Arquitetura é o primeiro estágio do desenvolvimento de um software e possui uma forte ligação com a Engenharia de Requisitos descrita na seção anterior. A decomposição da arquitetura em subsistemas ajuda a estruturar e organizar a especificação dos requisitos do software em questão. A Figura 1 apre-

senta as primeiras etapas do processo de desenvolvimento de um software, onde as especificações abstratas são as especificações de cada um dos subsistemas. Nessa figura, nota-se que o Projeto de Arquitetura gera a Arquitetura do Sistema e a Especificação Abstrata gera a Especificação mais formal do software.

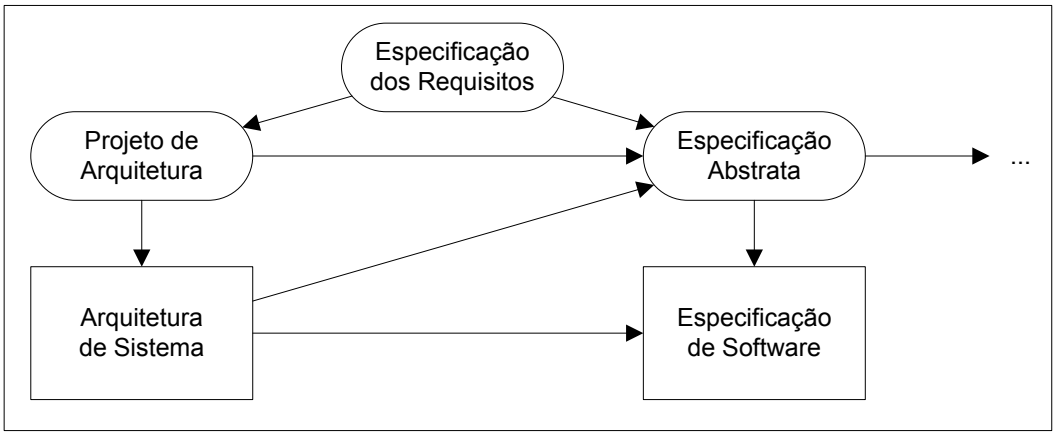

Figura 1 – Início do processo de desenvolvimento de um software.

O projeto da arquitetura precisa estabelecer um arcabouço básico do sistema e envolve a identificação dos seus componentes principais e do processo de comunicação entre esses componentes (BASS, CLEMENTS e KAZMAN, 2003). A documentação dessa arquitetura facilita a comunicação com os *stakeholders*, porque apresenta o sistema em alto nível, e facilita a análise do sistema quanto, por exemplo, ao desempenho, à confiabilidade, etc. Tal documentação ainda possibilita a reutilização em larga escala do projeto no desenvolvimento de outros sistemas com requisitos similares (GAMMA, HELM, *et al.*, 1994).

Os documentos de arquitetura incluem as representações gráficas dos modelos do sistema e um texto descrevendo esses modelos. Além disso, eles devem apresentar como o sistema é dividido em subsistemas e como esses são quebrados em módulos. Alguns modelos gráficos freqüentemente utilizados são:

- Modelo estrutural estático: apresenta os subsistemas ou componentes a serem desenvolvidos;
- Modelo do processo dinâmico: mostra a organização dos processos do sistema em tempo de execução;
- Modelo de interface: define os serviços públicos oferecidos pelos subsistemas;

- Modelos de relacionamento: apresenta o fluxo de dados entre os subsistemas.

Esses modelos podem ser desenvolvidos utilizando-se linguagens de descrição de arquitetura, conhecidas como ADL – *Architectural Description Languages*, que são formadas por componentes e conectores, além de incluírem regras e diretrizes para arquiteturas. Entretanto, tais linguagens não são muito comuns, porque são conhecidas apenas pelos arquitetos de software e isso dificulta a análise dos especialistas de domínio. Por isso, notações informais, como a UML – *Unified Modeling Language*, são mais utilizadas. No desenvolvimento deste trabalho, foi utilizada a UML nos documentos de arquitetura.

2.2.1. Decomposição em Módulos

Depois de montar uma arquitetura da estrutura do sistema, o próximo passo é decompor os subsistemas em módulos. Existem dois modelos principais que podem ser utilizados nessa decomposição: (a) o modelo orientado a objetos e (b) o modelo de fluxo de dados. Ambos podem ser implementados como componentes ou como processos.

No modelo orientado a objetos, os módulos se transformam em conjuntos de objetos minimamente acoplados, mas com interfaces bem definidas, e esses objetos possuem serviços que são invocados por outros objetos. A Figura 2 apresenta uma parte do diagrama de classes de objetos da ANNCOM.

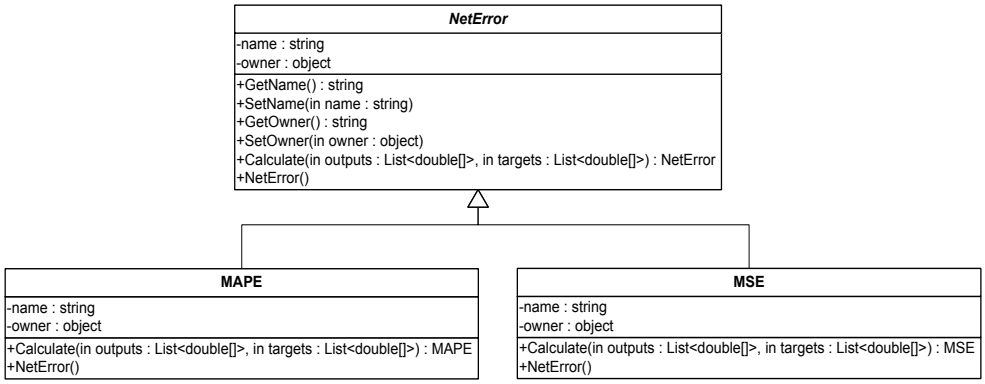

Figura 2 – Parte do diagrama de classes de objetos da ANNCOM em UML.

No diagrama de classes da Figura 2 foi utilizada a linguagem UML, onde as classes têm nomes e um conjunto de atributos e operações. As ligações com seta entre as classes representam uma hierarquia de pai e filho. A saída do processo de decomposição, usando a abordagem orientada a objetos, é um software cujos objetos são criados a partir das classes e com algum modelo de controle para coordenar suas operações.

Algumas vezes, para impedir ambigüidade entre itens que possuem o mesmo nome, os objetos podem ser organizados em Pacotes ou Espaços de Nomes. Esses pacotes podem ser definidos como contêineres abstratos que estabelecem um contexto para os objetos que armazena. Tendo em vista que cada contêiner trata de um contexto distinto, o significado de um nome pode variar de acordo com o Espaço de Nomes ao qual ele pertence.

A abordagem discutida acima apresenta várias vantagens, entre elas a facilidade de manutenção e alteração de partes do programa sem alteração do resto do software, uma vez que os objetos devem ser minimamente acoplados. Além disso, é muito mais fácil compreender um sistema se este é modelado com entidades que correspondem às do mundo real, e que neste caso são representadas por objetos.

Por outro lado, a orientação a objeto introduz alguns problemas. Por exemplo, se for necessário alterar a interface de alguma classe, deverá ser avaliado o impacto disso nas outras classes do sistema. Outro fato que deve ser levado em consideração é a dificuldade de mapear entidades complexas do mundo real em classes.

O outro modelo citado, que é utilizado para decomposição dos subsistemas em módulos, é o de fluxo de dados. Nesse, as transformações funcionais processam suas entradas e produzem saídas. Os dados vão sendo transformados à medida que vão passando de uma função para outra do sistema. Cada etapa do processo é desenvolvida na forma de uma transformação. Essas transformações podem ser efetuadas em série ou em paralelo.

Algumas vantagens do modelo de fluxo de dados são: (a) reuso das transformações; (b) é intuitivo, já que muitas pessoas imaginam seu trabalho na forma de processamento de entradas e saídas; (c) a adição de novas transformações é muito simples de ser projetada e desenvolvida; (d) é, geralmente, mais simples de ser implementado se comparado à orientação a objetos.

A grande desvantagem desse modelo vem da necessidade de se ter um formato comum entre as transformações. Por exemplo, o Unix utiliza um método chamado *pipeline*, onde uma seqüência de caracteres é trocada entre as funções e estas devem analisar suas entradas para gerar a saída. Essa análise cria um tempo extra (*overhead*) no sistema e pode significar que é impossível integrar transformações com entradas e saídas incompatíveis (SOMMERVILLE, 2004).

As duas abordagens apresentadas para decompor um sistema em módulos são exemplos de modelagens já conhecidas para resolver determinados tipos de problemas. Existem na literatura diversas abordagens que auxiliam na solução de problemas, algumas das quais serão descritas a seguir.

2.3. Padrões de Projeto

Atualmente, a maioria dos sistemas é decomposta utilizando-se um modelo orientado a objetos. Como visto anteriormente, essa abordagem possui diversas vantagens mas, em algumas situações, pode impor dificuldades na modelagem de entidades complexas. Mais do que isso, o projetista deve criar um modelo específico para o problema em questão, mas suficientemente abstrato para endereçar problemas correlatos futuros e evitar a remodelagem de sistemas semelhantes.

Para contornar esses problemas, os projetistas utilizam, em projetos novos, soluções e princípios utilizados em projetos anteriores; as boas soluções são reutilizadas sempre que possível. É comum encontrar padrões de classes e de comunicação entre objetos nos diferentes sistemas. Esses padrões resolvem problemas específicos de arquitetura e tornam a orientação a objetos mais flexível, elegante e extremamente reutilizável. Além disso, se um projetista estiver familiarizado com esses padrões, ele poderá aplicá-los rapidamente sem ter que redescobri-los (GAMMA, HELM, *et al.*, 1994).

De acordo com o arquiteto civil Christopher Alexander, um padrão de projeto descreve um problema que ocorre repetidas vezes em um ambiente, e descreve o núcleo da solução para este problema de forma que esta possa ser usada muitas vezes sem precisar ser refeita. Isso também é verdade quando se trata de padrões de projeto de software (GAMMA, HELM, *et al.*, 1994).

Em (GAMMA, HELM, *et al.*, 1994), considera-se que os padrões de projeto são constituídos por quatro partes principais:

- Nome;
- Problema;
- Solução;
- Conseqüências

O nome do padrão resume o problema, suas soluções e conseqüências em uma ou duas palavras. Esse vocabulário de padrões pode ser usado para aumentar o nível de abstração, facilitar a comunicação entre projetistas e a documentação do sistema.

O item Problema indica **quando** o padrão deve ser usado, explicando o problema em questão e seu contexto. Nessa parte, os algoritmos são apresentados como classes de objetos e estruturas que são inflexíveis. Além disso, nesse item podem ser apresentados os pré-requisitos para a aplicação do padrão.

No item Solução são descritos os elementos da arquitetura e seus relacionamentos, suas responsabilidades e colaboradores. Nesse item não é apresentada nenhuma implementação ou arquitetura concreta do sistema, porque os padrões não são específicos para um determinado problema, mas para muitos problemas em diversas situações. Entretanto, a solução provê uma descrição abstrata de um problema arquitetural e como as classes e os objetos o resolvem.

A Conseqüência descreve os resultados e o "preço" pago por se usar o determinado padrão. Uma análise de custo-benefício deve ser feita antes de se utilizar o padrão, de modo que alternativas possam ser estudadas. Normalmente, essas conseqüências se referem a tempo de processamento e espaço em memória, mas podem endereçar detalhes de implementação e linguagem. Como o reuso é uma das principais vantagens da orientação a objetos, as conseqüências incluem impactos na flexibilidade, na portabilidade e na extensibilidade do sistema (GAMMA, HELM, *et al.*, 1994).

A seguir serão apresentados os padrões de projeto utilizados para o desenvolvimento da ANNCOM e uma explicação resumida de cada um.

2.3.1.
Composição (*Composite*)

O objetivo desse padrão é compor objetos em estruturas de árvore. Com isso, objetos individuais ou composições desses objetos podem ser tratados de forma igual. No caso da ANNCOM, a complexa relação entre camadas, neurônios e sinapses foi montada de forma que esses objetos mais simples formassem uma rede neural artificial completa. Um detalhe existente no projeto da biblioteca é a inexistência da classe abstrata para representar todos os elementos, pois a classe *NeuralNet*, que será apresentada no capítulo 4, já faz o papel de compor tudo e distinguir o tratamento de cada "sub" classe facilitando a interface com o usuário. A Figura 3 mostra a estrutura recursiva da ANNCOM.

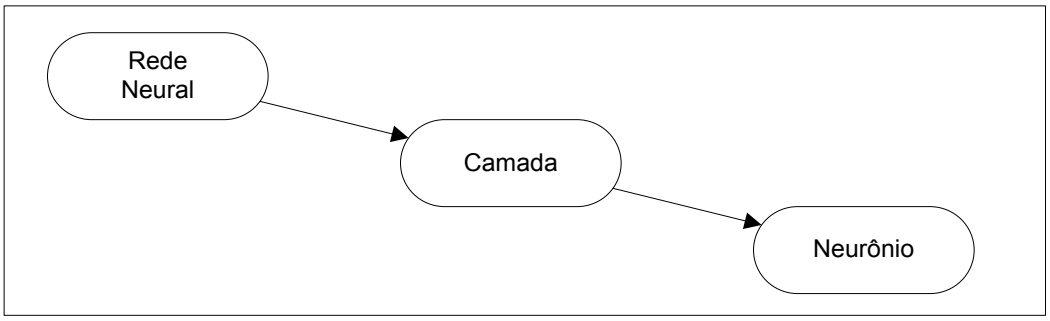

Figura 3 – Estrutura recursiva da ANNCOM.

Os participantes deste padrão de projeto são:

- Componente: os objetos *NeuralNet* e *Synapse* declaram os objetos das "sub" classes, endereçam o acesso a todos os objetos contidos nele, definem o comportamento do grupo e a estrutura recursiva na forma de "pai e filho";
- Folha: os objetos *Layer*, *Neuron* e *InputSynapse* não possuem filhos e definem seus respectivos comportamentos na composição;
- Composição: este representa a rede neural e define o comportamento dos componentes, armazena os objetos que fazem parte da composição e implementa as operações de todo o conjunto;
- Cliente: o aplicativo que será apresentado no capítulo 4, chamado *Clinn*, é o cliente e manipula toda a estrutura.

A primeira conseqüência da Composição é o modelo hierárquico da biblioteca. Além disso, vale destacar: a facilidade de manipulação por parte do cliente, pois este não precisa se preocupar em como a estrutura é formada; a simplicidade

para adição de novas "sub" classes, uma vez que sua estrutura interna deve ser muito parecida com a de outra "sub" classe irmã; e a generalidade do modelo. Esta última característica é extremamente importante no caso da ANNCOM, porque permite a criação de diversos tipos de redes neurais utilizando os mesmos componentes.

2.3.2. Cadeia de Responsabilidade (*Chain of Responsibility*)

A Cadeia de Responsabilidade tem como objetivo evitar o acoplamento entre o objeto solicitante de um pedido e o objeto que foi solicitado, criando uma cadeia onde vários objetos têm a possibilidade de manipular a requisição. Na ANNCOM, um exemplo de uma cadeia é o pedido de propagação de um sinal feito pelo cliente à rede neural. Essa solicitação pode ser tratada pela rede, camadas ou neurônios. Um problema dessa abordagem é que o objeto solucionador não tem informação sobre o objeto que foi requisitado. Entretanto, no nosso caso, essa informação não é importante. A Figura 4 mostra como o objeto *NeuralNet* passa o pedido de propagação para os objetos subseqüentes.

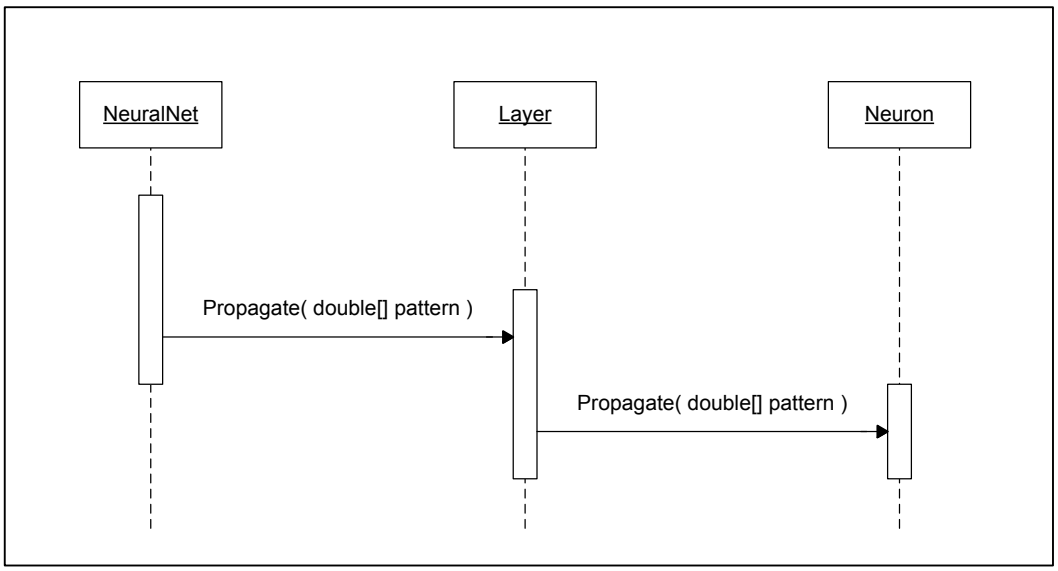

Figura 4 – Cadeia de responsabilidade do método *propagate*.

Para que o pedido seja passado por toda a cadeia, é necessário que cada objeto dentro dela compartilhe uma interface comum para atender ao pedido e passá-

lo ao seu sucessor. Deve-se perceber que os métodos na Figura 4 possuem a mesma interface.

No caso da cadeia de responsabilidade os participantes estão listados com suas respectivas descrições abaixo:

- Manipulador: o objeto *NeuralNet* define a interface para o cliente iniciar o processo de propagação;
- Manipulador concreto: os objetos *Layer* e *Neuron* tratam os pedidos pelos quais são responsáveis ou encaminham para o sucessor;
- Cliente: como no exemplo do padrão anterior, o *Clinn* é um cliente, pois inicia o processo.

Ao se utilizar a Cadeia de Responsabilidade é possível obter uma redução de acoplamento entre os objetos, uma vez que o objeto está livre da responsabilidade de saber quem irá tratar determinada solicitação; deve simplesmente saber que alguém irá tratar. Outro ponto positivo é a flexibilidade na delegação de responsabilidades para os objetos. Por exemplo, na ANNCOM, ao se propagar um sinal pela rede, as responsabilidades são distribuídas pelas camadas e, conseqüentemente, pelos neurônios que são os mais especializados. Uma conseqüência negativa da Cadeia de Responsabilidade é a não garantia de tratamento da solicitação. No caso da ANNCOM, a camada recebe o pedido de propagação e a repassa para os neurônios, mas sem saber se este será ou não tratado.

2.3.3. Estratégia (*Strategy/Policy*)

O padrão denominado Estratégia surge da necessidade de se definir uma família de algoritmos. Por meio dessa abordagem, é possível encapsulá-los e torná-los intercambiáveis facilitando a independência de uso para os clientes.

Ao se treinar uma rede neural na ANNCOM, é possível escolher vários tipos de treinamento, que são encapsulados por um único treinador. Se o padrão não tivesse sido usado, cada vez que se quisesse treinar um determinado tipo de rede neural, seria necessário adicionar o algoritmo de treinamento específico dentro do cliente e essa adição consumiria mais recursos, pois o cliente teria que ser modificado. Mais do que isso, nem sempre todos os algoritmos são utilizados e não pre-

cisam estar carregados junto com o resto do sistema. A Figura 5 mostra como é possível encapsular os diferentes tipos de algoritmo.

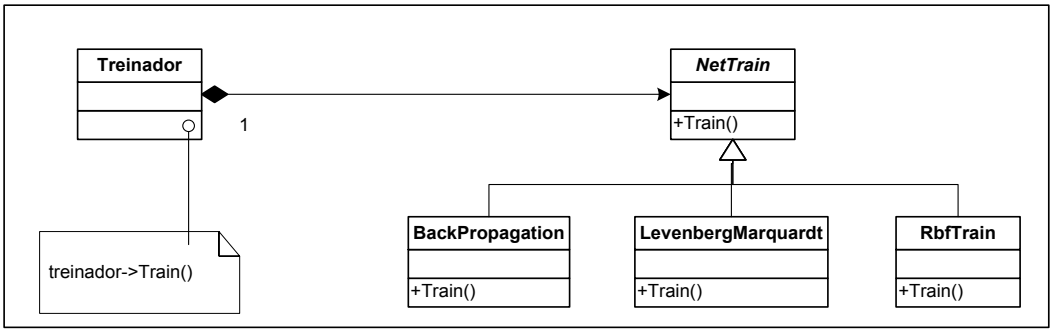

Figura 5 – Algoritmos de treinamento da ANNCOM encapsulados.

Os participantes do padrão Estratégia estão listados abaixo:

- Estratégia: este é o *NetTrain*, que declara a interface comum para todos os tipos de treinamento;
- Estratégia concreta: implementa o algoritmo usando a interface da Estratégia, por exemplo, o *BackPropagation* e *LevenbergMarquardt*.
- Contexto: o treinador *NetTrain* é um exemplo deste participante. Ele configura um tipo de treinamento, referencia-o e fornece à Estratégia o acesso aos dados.

O padrão Estratégia apresenta como principal benefício a definição de famílias de algoritmos, comportamentos e contextos relacionados para reuso. Um segundo benefício é a eliminação da necessidade de declarações condicionais encapsulando os diferentes contextos em classes de Estratégia separadas. Outro ponto positivo é a possibilidade de escolha de várias implementações para um mesmo comportamento. As conseqüências negativas são: aumento no número de objetos, o que pode ser minimizado ao se utilizar o padrão *Flyweight* (GAMMA, HELM, *et al.*, 1994); excesso de comunicação entre estratégias e contextos; e a necessidade dos clientes entenderem como cada estratégia funciona para poder escolher a mais adequada.

Existem outros vários padrões apresentados em (GAMMA, HELM, *et al.*, 1994) e muitos outros foram desenvolvidos depois da publicação do livro. Entre eles destacam-se o MVC e o DAO.

2.4. .NET Framework

Até agora foram discutidas várias abordagens para se melhorar a qualidade de sistemas de software e minimizar seus custos de desenvolvimento. Nesta seção será apresentada a base sobre a qual a ANNCOM foi montada; o *.NET framework* – o arcabouço da Microsoft.

O arcabouço da Microsoft provê suporte à programação orientada a objetos, ambiente para execução, que minimiza custos com implantação e controle de versão. Além de promover a execução segura de sistemas e permitir programação dinâmica. Essas características são possíveis, em grande parte, por causa da Linguagem Comum em Tempo de Execução (CLR – *Common Language Runtime*), que será explicada na próxima seção. Pode-se entender o componente responsável pela CLR como um agente que gerencia o código em tempo de execução provendo serviços centrais como gerência de memória, tratamento de *threads*, acesso remoto e etc. Os sistemas que são compilados para a CLR são chamados de gerenciáveis enquanto que os outros sistemas são chamados não gerenciáveis.

Os sistemas podem ser criados com código gerenciável e com código não gerenciável, possibilitando que o software usufrua das características dos dois "mundos". Por exemplo, o navegador *Web* da Microsoft foi desenvolvido em código não gerenciável, mas executa Formulários Windows e componentes gerenciáveis em documentos de Internet.

Outra característica do arcabouço da Microsoft são os modos de execução (*runtimes*). Entre eles, talvez o mais usado seja o ASP.NET, que provê execução gerenciada e escalar em servidores *Web*, permitindo o funcionamento de aplicações e serviços, ambos *Web*. A Figura 6 ilustra o relacionamento da CLR e das bibliotecas de classes com as aplicações e o resto do sistema. Nela podem-se ver as bibliotecas de classes e de componentes sendo executadas sobre o *runtime* básico e as aplicações *Web*, sendo executadas sobre o *runtime* ASP.NET (MICROSOFT, 2009).

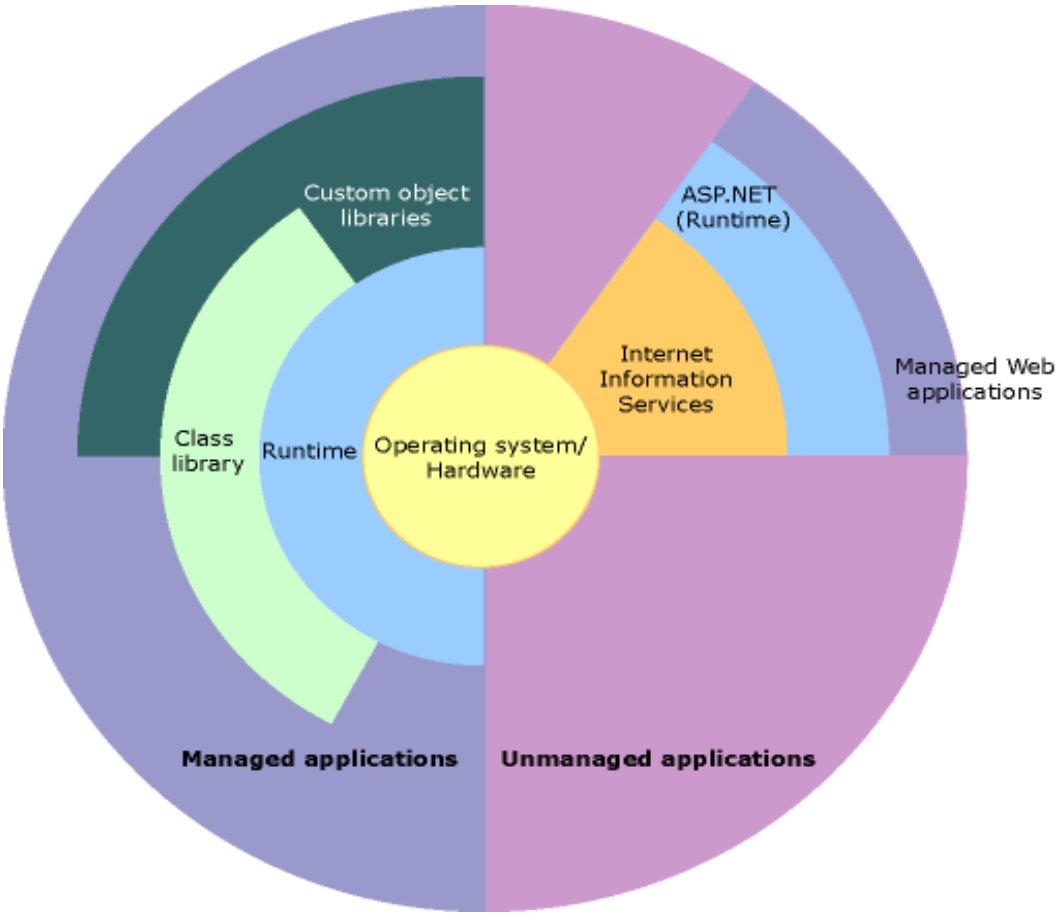

Figura 6 – O *.NET Framework* e o resto do sistema (MICROSOFT, 2009).

2.4.1. Linguagem Comum em Tempo de Execução (CLR)

Os compiladores e as ferramentas da CLR expõem as funcionalidades de cada *runtime* e permitem que o desenvolvedor escreva código usufruindo dos benefícios desses ambientes gerenciáveis de execução. O código gerenciável apresenta, além dos benefícios apresentados acima, integração entre linguagens, manipulação de exceção, segurança, controle de versões e suporte de implantação, um modelo simplificado para interação de componentes, depuração e verificação de desempenho (*profiling*). Para permitir que o *runtime* forneça esses benefícios, os compiladores CLR devem gerar um dicionário de dados (metadados), que descreve os tipos, membros e referências no código. Esses dicionários são armazenados com o código e todo componente executável contém um. O *runtime* usa os metadados para localizar e carregar classes, estabelecer instâncias na memória, resolver invocações de método, gerar código nativo, reforçar a segurança, e definir limites de contexto.

O *runtime* automaticamente trata o formato dos objetos e gerencia suas referências, liberando-os quando eles não estão mais sendo usados. Objetos cujas instâncias são geridas dessa forma são chamados dados gerenciados. O coletor de lixo (*Garbage Collector*) elimina vazamentos de memória, bem como alguns outros erros comuns de programação. Se um código é gerenciado, é possível usar dados gerenciados, dados não gerenciados, ou ambos em seu sistema. Como compiladores CLR resolvem seus próprios tipos, como os tipos primitivos, não existe a necessidade de saber se os dados estão sendo gerenciados. Além disso, a CLR facilita a criação de componentes e aplicativos cujos objetos interagem entre linguagens. Objetos escritos em linguagens diferentes podem se comunicar uns com os outros, e seus comportamentos podem ser fortemente integrados. Além disso, é possível passar uma instância de uma classe para um método de outra em linguagem diferente. Essa integração entre linguagens é possível porque os compiladores CLR e ferramentas, que usam o mesmo *runtime*, possuem seus tipos definidos por esse *runtime* e seguem suas regras para a definição de novos tipos, bem como para a criação, uso e persistência dos tipos.

Como parte de seus metadados, todos os componentes gerenciados carregam as informações sobre os objetos e recursos que foram construídos. O *runtime* usa essas informações para garantir que o sistema manterá versões específicas de tudo o que precisa, o que torna o código menos propenso a erro por causa de alguma dependência faltante. Os registros com informações e estados dos dados, como não são armazenados no Registro, são fáceis de estabelecer e manter. As informações sobre os tipos definidos, e suas dependências, são armazenadas com o código na forma de metadados. Isso torna as tarefas de replicação e remoção de componentes muito menos custosas.

Para concluir, é possível encarar os compiladores CLR e as ferramentas como facilitadores no desenvolvimento de sistemas. Entretanto, algumas características podem ser mais visíveis em um ambiente do que em outro. Na Tabela 1, são destacadas algumas das principais características das três linguagens suportadas pela versão 3.5 do arcabouço da Microsoft.

Linguagem *Visual Basic*	Melhorias de desempenho.
	Capacidade de usar com facilidade os componentes desenvolvidos em outras linguagens.
	Tipos extensíveis fornecidos por uma biblioteca de classes.

	Novos recursos da linguagem, como herança, interfaces e sobrecarga de programação orientada a objeto; suporte para *threading*, que permite a criação de vários *threads*, aplicações escaláveis; suporte para manipulação de exceção estruturada e atributos personalizados.
Linguagem C++	Integração entre linguagens, especialmente herança entre linguagens.
	Coletor de lixo, que gere a vida útil dos objetos para que a contagem de referência seja desnecessária.
	Objetos que se descrevem, torna desnecessário o uso da Linguagem de Definição de Interface (IDL).
	A capacidade de compilar uma vez e rodar em qualquer CPU e sistema operacional que suporta o *runtime*.
Linguagem C#	Orientação a objetos completa.
	Muita segurança no controle dos tipos fortemente tipados.
	Uma boa mistura entre a simplicidade do *Visual Basic* e o poder do C++.
	Coletor de lixo.
	Sintaxe e palavras-chave semelhantes ao C e C++.
	Uso de delegados, em vez de ponteiros de função, para maior segurança de tipo e de segurança. Ponteiros de função estão disponíveis através do uso da palavra-chave *unsafe* a opção */unsafe* do compilador C# (Csc.exe) para o código não gerenciado e os dados.

Tabela 1 – Principais características das linguagens *.NET*.