



# Princípios de Engenharia de Software

*Descriptivo Conceitual*

# Sumário

<b>1.</b>	<b><u>ENGENHARIA DE SOFTWARE X PRÍNCIPIOS</u></b>	<b>3</b>
1.1.	ENGENHARIA DE SOFTWARE.....	3
1.2.	DIMENSÕES DA ENGENHARIA DE SOFTWARE .....	4
<b>2.</b>	<b><u>PRÍNCIPIOS EXTERNOS</u></b>	<b>6</b>
2.1.	CORRETUDE.....	7
2.2.	CONFIABILIDADE .....	7
2.3.	ROBUSTEZ.....	7
2.4.	DESEMPENHO .....	8
2.5.	ERGONOMIA -FACILIDADE DE USO.....	8
2.6.	VERIFICABILIDADE .....	8
2.7.	FACILIDADE DE MANUTENÇÃO.....	9
1.7.1	SOFTWARE REPARÁVEL .....	10
1.7.2	SOFTWARE EVOLUTIVO.....	10
2.8.	REUTILIZAÇÃO.....	10
2.9.	PORTABILIDADE .....	11
2.10.	COMPREENSIBILIDADE .....	11
2.11.	INTEROPERABILIDADE.....	11
2.12.	PRODUTIVIDADE.....	12
2.13.	DESvio DE PLANEJAMENTO .....	12
2.14.	VISIBILIDADE.....	13
2.15.	SEGURANÇA.....	13
<b>3.</b>	<b><u>PRÍNCIPIOS INTERNOS</u></b>	<b>14</b>
3.1.	RIGORISMO.....	15
3.2.	SEPARAÇÃO DE ASPECTOS .....	15
	DIMENSÃO: ATÔMICA .....	16
	DIMENSÃO: PRIMITIVA .....	17
	DIMENSÃO: NEGÓCIO .....	17
	DIMENSÃO: CONDICIONANTE.....	17
	DIMENSÃO: EXISTENCIAL.....	18
3.3.	MODULARIDADE .....	18
3.4.	ABSTRAÇÃO.....	19
3.5.	GENERALIDADE.....	19
3.6.	ANTECIPAÇÃO A MUDANÇAS .....	20
3.7.	REFINAMENTO.....	20
3.8.	CLASSIFICAÇÃO .....	21

# 1. Engenharia de Software x Princípios

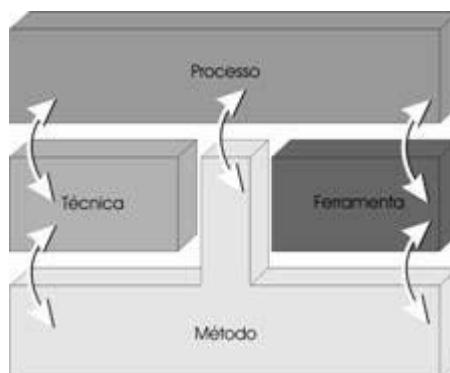
## 1.1. Engenharia de Software

A engenharia de software define um conjunto de princípios que fundamentam a qualidade e as técnicas de construção de um software. Portanto, quando estes princípios são corretamente atendidos acreditamos que o software estará corretamente construído e com a qualidade adequada.

Estes princípios fundamentam, ou pelos menos deveriam fundamentar, todas as técnicas de desenvolvimento de software.

Iremos nos ater a dois pontos fundamentais em qualquer Engenharia, o **produto** e as **técnicas** utilizadas para desenvolvê-lo. As técnicas podem ser aplicadas artesanalmente ou industrialmente. Para este salto nas técnicas precisamos de um **processo** de produção.

Paralelamente às técnicas, temos os **métodos**. O método está orientado para o rigor formal , sendo a abstração e a corretude de suas formulações sua meta final, independente se podermos na prática produzir ou não algum produto usando estes métodos.



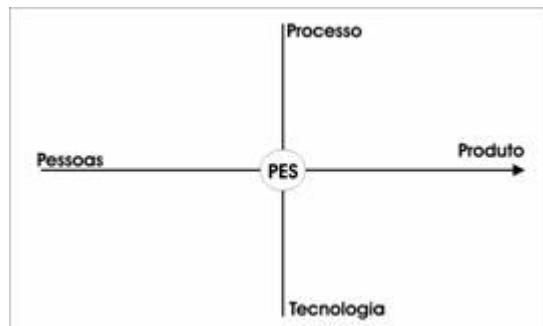
**Figura 1 – Processo de Engenharia de Software**

**Engenharia de Software:** Conjunto de **técnicas**, **métodos**, **processos** e **ferramentas** utilizadas na especificação, construção, implantação e manutenção de um software visando garantir a gerência, o controle e a qualidade dos **artefatos** gerados através de **recursos humanos**

## 1.2. Dimensões da Engenharia de Software

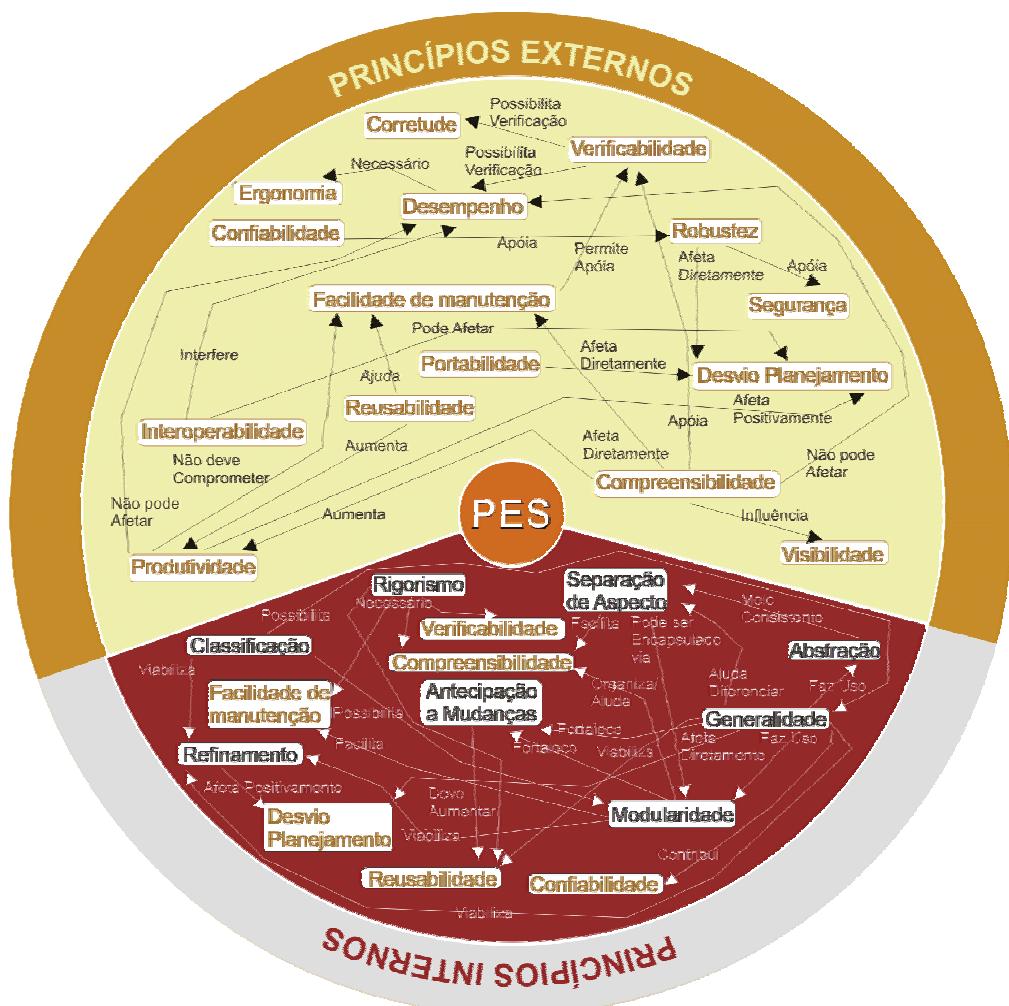
A criação de um produto, na maioria das vezes, exige um conjunto de **ferramentas** apropriadas. Um software só poderá ser produzido através de **recursos humanos**. O intelecto humano está continuamente defendendo-se do ambiente e de seus instintos. Portanto, a correta gerência sobre estes aspectos é condição essencial para o sucesso de um software.

Estes seis fatores de Engenharia de Software dão origem a quatro dimensões de aplicabilidade de nossos princípios, para os quais são desenvolvidos padrões internacionais de qualidade.



**Figura 2 - Dimensões de Engenharia de Software**

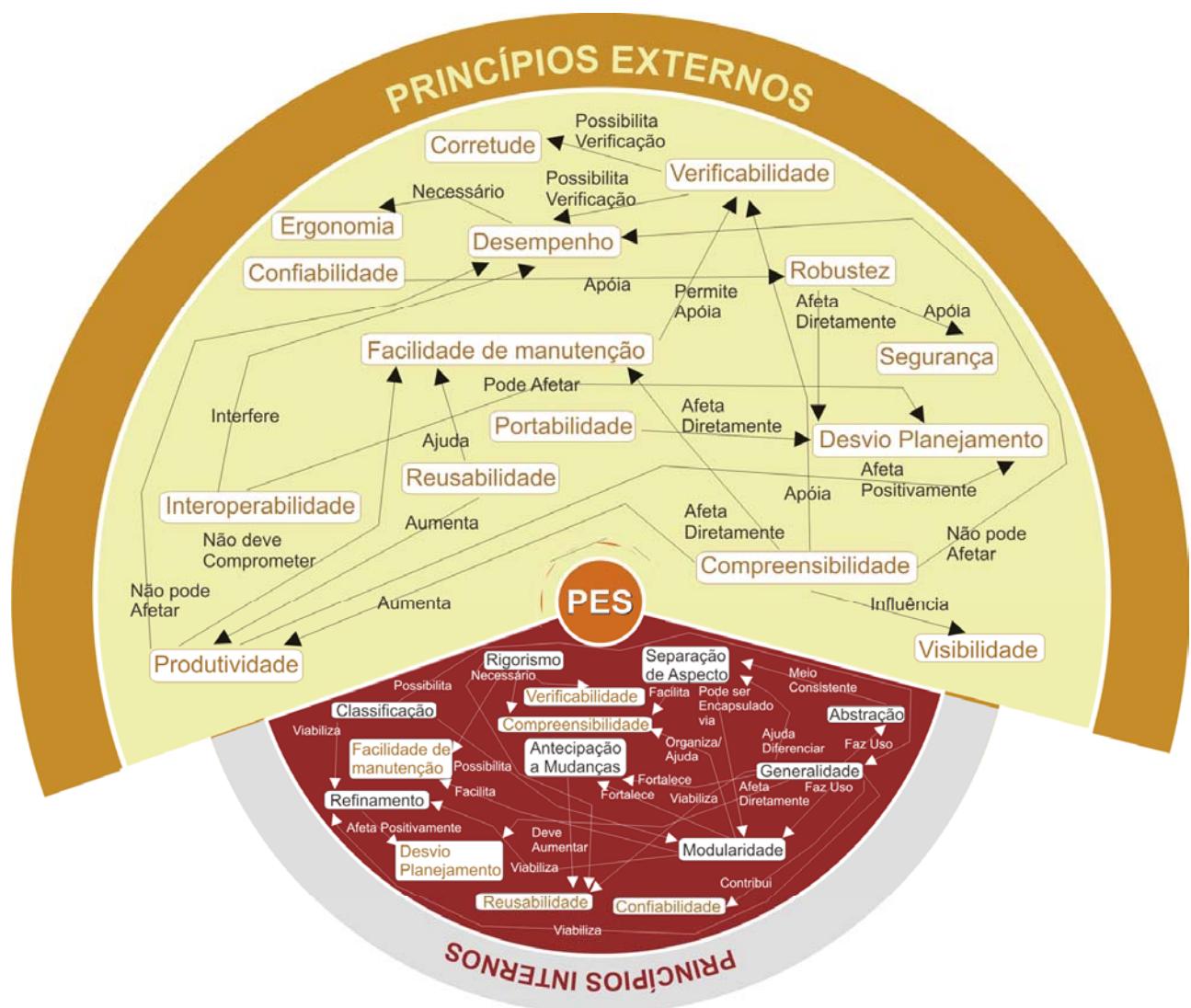
Os fatores método, técnica e processo compõem a dimensão de Processo. O fator ferramenta compõe a dimensão Tecnologia. O fator produto compõe a dimensão Produto. E por último, o fator recursos humanos a dimensão Pessoas. Os Princípios de Engenharia de Software (**PES**) representam o fundamento destas quatro dimensões. Os princípios podem ser visto como axiomas ou verdades fundamentais que deverão estar presentes em qualquer software.



## 2. Princípios Externos

Os princípios externos estão estritamente relacionados com a dimensão Produto de um software e podem, no entanto, ser aplicados também às outras dimensões. Certamente, nos falta alguma compreensão, do por que, destes princípios serem utilizados para avaliar e melhorar a qualidade de um software somente no seu término, e não ao contrário, sendo a fundamentação básica e primitiva na qual todo software deve ser construído.

Os princípios externos são os quantificadores que determinarão a qualidade externa do produto final. Qualidade está intrinsecamente ligada à aceitação do produto, e assim, obviamente com seu sucesso ou fracasso.



## 2.1. Corretude

Na fase de Análise de Requisitos, serão diferenciados claramente os requisitos denominados funcionais dos requisitos não funcionais. Basicamente o primeiro refere-se àquilo que o usuário definiu como característica do software, por exemplo, ter um cadastro de todos os clientes de uma companhia de seguros. O segundo, refere-se a aspectos gerais que os requisitos funcionais devem obedecer, como tempo de resposta inferior a 5 segundos, por exemplo.

Portanto, um software ou componente de software é dito estar correto, se ele comporta-se de acordo com o que foi definido em seus requisitos funcionais.

Junto com a confiabilidade, deverá ser a base de nossos testes. Deverá definir ou exemplificar o que significa determinar se um determinado requisito está correto.

Sendo assim, para dizermos se um software é correto ou não, temos que ter em mãos a especificação na forma de requisitos funcionais. Se esta não existe, como acontece na maioria dos casos, não temos como dizer se um software está correto ou não, pois, sua ausência implica, diretamente, que sua construção foi realizada seguindo o princípio do “ontem”. Ou seja, o gerente solicita e diz: é para ontem. Este princípio contém em si várias ambigüidades, portanto, coexistem no software várias visões conflitantes. É uma questão de tempo, para que elas se tornem visíveis e o conflito se estabeleça.

## 2.2. Confiabilidade

A corretude de um software está relacionada a um fator absoluto. Ou seja, ou o software atende às especificações e é correto ou o contrário.

Já a Confiabilidade está relacionada a um fator relativo. Ou seja, podemos falar em probabilidade, e quem diria que um software que erra apenas 1% de todo o seu funcionamento não é confiável?

Um software pode até mesmo errar em 10% dos casos e ser mais confiável do que outro que erre apenas 5%. Isto se quantificarmos também o grau de severidade do erro.

A Confiabilidade é o verdadeiro componente químico responsável em externar a imaturidade da Engenharia de Software. Pois, enquanto as outras Engenharias fornecem graus variados de confiabilidade em seus produtos, nós da área de software, acreditamos firmemente que a presença de “bugs” faz parte de qualquer software. E este condicionamento é, às vezes, forte demais, o que gera até mesmo desentendimentos freqüentes entre gerência e programadores, ou mais precisamente entre programadores e homologadores.

Assim, consideramos extremamente natural entregar um software com uma lista de erros. Alguém já imaginou se comprássemos uma televisão, e junto viesse uma lista de erros e problemas? Ou se em uma ponte tivesse uma placa: “não passe por este lado, esta ponte pode cair”. Assim, trocamos a garantia de confiabilidade de até 3 anos de outras engenharias por uma lista garantindo os problemas para hoje mesmo, para que esperar?.

Junto com a corretude, deverá ser a base de nossos testes. Implica em descrever o que realmente interessa no requisito e o aspecto que o tornará confiável. Veja que poderão existir erros, contudo, se o erro for em X o software não será confiável, mas se o erro for em Y, não trará maiores danos.

## 2.3. Robustez

Um software é dito Robusto se seu comportamento se mantém aceitável quando condições não previstas na especificação acontecem, como por exemplo, uma entrada com dados incorretos ou pouca memória disponível.

Portanto, um programa que só funcione especificamente nas condições normais de temperatura e pressão - CNTP - não é dito robusto se no primeiro desvio de suas funcionalidades pré-definidas ele pendurar a máquina.

Novamente uma comparação com outras Engenharias seria interessante. Imagine que durante uma forte chuva, a tensão elétrica flutuasse consideravelmente acima dos 115 V especificados para um determinado aparelho de TV. Se este aparelho queima na primeira variação, significa que ele não é robusto, porém se dependendo da flutuação o aparelho permaneça sem maiores danos, significa que ele é robusto.

Robustez está relacionada a prazos, valores de investimento e área de aplicação. Um jogo de vídeo game pode ser pouco robusto e funcionar somente com as configurações mínimas sugeridas. Um software embarcado, de avião, por

exemplo, não pode se dar ao mesmo luxo. E obviamente quanto mais robusto um software maior é o prazo de desenvolvimento e também os valores de investimento.

## 2.4. Desempenho

Desempenho é mais conhecido como “Performance”. Desempenho refere-se ao tempo de resposta do software a uma determinada requisição. Nas outras Engenharias o termo eficiência é mais utilizado. Contudo, eficiência em software refere-se ao quanto um software é econômico em relação aos recursos de máquina utilizados, como memória e disco, por exemplo.

Obviamente eficiência e desempenho estão relacionados. Pois um software pode exigir bastante memória para ter desempenho, em detrimento da eficiência no uso do recurso memória.

Existe também o entrelaçamento entre Desempenho e Escalabilidade. Pois, um algoritmo poderá funcionar bem até um determinado volume ou tamanho do software. Em outras condições, o desempenho pode ser exponencialmente prejudicado.

A fase de Analise de Requisito definirá as restrições de desempenho ou performance para a execução do requisito em questão. Por exemplo, no máximo 10 segundos para cadastrar um cliente pela Internet. Dados como quantidade de usuários, acesso concorrente, numero de transações por hora etc. devem ser fornecidos. A disponibilidade também deverá ser fornecida aqui. Ou seja, se o componente ou requisito poderá sair do ar ou nunca sair do ar.

## 2.5. Ergonomia - Facilidade de Uso

Este requisito, quando implementado, permitirá que o software desenvolvido seja utilizado por um profissional com graduação de doutor ou por um senhor aposentado. Portanto, variáveis externas condicionam o nível de facilidade que deverá ser fornecido por este requisito.

Ergonomia implica na facilidade de uso do software pela comunidade usuária.

Talvez em um grau superior aos outros princípios este seja o mais subjetivo de todos. Um usuário experiente irá preferir atalhos ao uso de menu. Já um usuário novato irá preferir um software com mensagens que guiem em eventuais erros.

A interface do usuário está estritamente relacionada a facilidade de uso, embora possamos definir a facilidade de uso para software que não possua interface de usuário. Neste caso, poderíamos estipular a facilidade de uso com sua aderência e facilidade de integração com outros softwares.

Podemos também relacionar a facilidade de uso com a corretude e desempenho. Pois, de nada adianta um software mandar uma linda mensagem de erro para o usuário constantemente. O mesmo vale para desempenho, nada adianta mandar 10 minutos depois uma linda mensagem dizendo que o cliente foi cadastrado com sucesso.

Um estudo sério sobre a facilidade de uso do software já pode ser encontrado na literatura de Engenharia de Software. Esta tendência segue na verdade o caminho trilhado por outras Engenharias. Como elas, percebemos que os fatores psicológicos humanos determinam se um software é de fácil uso ou não.

A unificação das interfaces do usuário através de sistemas operacionais possibilitou uma melhora substancial nesta direção. Gostaria de aproveitar e fazer uma reclamação, pois os controles remotos estão ficando cada vez mais complicados. Não posso perder a oportunidade de “depreciar” as outras engenharias, pois as oportunidades são poucas.

## 2.6. Verificabilidade

Verificabilidade está relacionada à capacidade de verificar as propriedades ou, mais concretamente, os atributos de um sistema.

A capacidade de verificar a corretude ou o desempenho de um sistema é um exemplo de verificabilidade.

Veja que a inserção de invariantes em nossos tipos de negócio e a realização da programação dos processos de um sistema via pré-condição, pós-condição e semântica, nos fornece um controle maior sobre a verificabilidade de um sistema.

A inserção de códigos extra, chamados monitores, nos possibilita a criação de softwares que monitoram, em tempo real, a saúde de um sistema.

Este requisito fornece os meios ou exemplos que deverão ser utilizados para verificar ou inferir se os outros requisitos estão sendo cumpridos ou obedecidos. Por exemplo, para que este requisito (funcional) seja considerado com facilidade de uso ele deverá possuir uma lista onde o usuário poderá clicar e obter instantaneamente a vida financeira do cliente.

## 2.7. Facilidade de Manutenção

A manutenção de um software está tradicionalmente relacionado a correção de bugs.

No entanto, estudos recentes têm demonstrado que a maior parte do tempo gasto em manutenção, está na verdade relacionada a melhorias no software e também na correção de erros de especificação na versão original.

Se compararmos o termo manutenção com outras Engenharias, veremos que existe uma semântica diferente. Nas outras Engenharias manutenção refere-se a um processo de atualização de um produto devido à deterioração.

Em nosso caso, manutenção possui o sentido de evolução ou aperfeiçoamento do produto, ou software. Contudo, no estágio atual da Engenharia de Software, o uso do termo manutenção passou a ser padrão geral, portanto não há como modificá-lo.

Estatisticamente a manutenção de um software custa no mínimo 90% do valor original e não há limite para o máximo. Todavia, intrinsecamente ligado a este fator está a realização correta da especificação do sistema. Falhas neste processo representa o principal responsável pelo alto custo da manutenção dos sistemas.

A manutenção de um software é dividida em 3 etapas:

1. Manutenção Corretiva
2. Manutenção Adaptativa
3. Manutenção Evolutiva

A Manutenção Corretiva é responsável pela solução de eventuais erros residuais presentes no software, oriunda da liberação de uma nova versão ou de manutenções evolutivas. Em relação ao mínimo de 90%, podemos dizer que a Manutenção Corretiva é responsável por 20% deste valor.

A Manutenção Adaptativa é primariamente atribuída a mudanças no ambiente ao qual o sistema está em funcionamento. Por exemplo, a mudança de um sistema operacional ou banco de dados da aplicação. Contudo, talvez de forma única, nos referimos a manutenção adaptativa ou as mudanças necessárias em um software quando a versão 1.0 da aplicação é implantada. Pois, nestes casos, a especificação poderá estar completa e todos os protótipos de tela aprovados. No entanto, na implantação várias adaptações à realidade do usuário são necessárias. Adaptações estas que só foram percebidas ou necessárias durante a primeira semana de uso do software. Concordamos que aparentemente esta tarefa poderia ser antecipada e, portanto, esta adaptação não seria necessária. Contudo, imagine a figura do alfaiate de algum tempo atrás. Podemos supor que o cliente experimentou a roupa várias vezes, porém, em seu primeiro uso real, após passar horas realmente com a roupa, pequenos ajustes finais sejam necessários. Estes ajustes ou adaptações são mandatórios. Esta é nossa experiência na implantação de vários sistemas. Em relação ao mínimo de 90%, podemos dizer que a Manutenção Adaptativa é responsável por 20% deste valor.

E por último, a Manutenção Evolutiva refere-se a inserção de novos requisitos ao software. Requisitos estes que não estavam presentes na especificação original e representam novas funcionalidades no software. Em relação ao mínimo de 90%, podemos dizer que a Manutenção Evolutiva é responsável por 50% deste valor.

Existem ainda duas divisões úteis presentes na manutenção de um software:

1. Software Reparável
2. Software Evolutivo

## Software Reparável

Um software é dito reparável se seus defeitos podem ser corrigidos utilizando-se um conjunto razoável de esforço em sua manutenção.

A indústria automobilista projeta seus componentes mais danificáveis de forma a serem os mais acessíveis à manutenção.

Quando um produto torna-se uma commodity e consequentemente seus preços tornam-se baratos, não há mais necessidade de repará-lo, estes componentes passam a ser objeto de troca.

Existe um elemento fundamental que permite a reparabilidade de componentes. Este elemento é a padronização. Para que possamos trocar ou reparar peças ou componentes, necessitamos padronizar suas interfaces e seu funcionamento interno.

Praticamente todas as Engenharias, excetuando-se obviamente a nossa, já alcançaram um alto nível de padronização de suas peças ou componentes.

Nossos microcomputadores atuais, por exemplo, possuem um número elevado de fabricantes. Pois, desde que exista padrões para os componentes de um computador, podem surgir novos fabricantes para estes componentes. Este processo torna a peça ou componente mais barato e também mais confiável, e ainda a concorrência faz com que os produtos tenham mais qualidade.

A única teoria disponível atualmente visando fornecer a Engenharia de Software padrões industriais é a Component Based Development - CBD.

Em resumo, um software tem que ser projetado para ser reparável. Este é um problema de projeto (Design) dos componentes. Se esta concepção na produção de componentes de software não for utilizada, a reparabilidade de um software estará condicionada à mágicas de programação, e esta nem sempre é possível.

Obviamente que um software tradicional de cunho monolítico dificulta a reparabilidade, enquanto que um software subdividido em componentes as facilita. Assim, tendo em vista que um software é feito para durar, sua durabilidade depende de seu poder de reparabilidade, a teoria de componentização e sua prima mais velha, orientação a objetos, passam a ser elementos fundamentais para industrialização e profissionalização de nossa engenharia.

## Software Evolutivo

Um software é dito evolutivo se ele ainda comporta acréscimos de requisitos em suas funcionalidades, sem distorcer ou danificar outras funcionalidades do software.

A maleabilidade do software decididamente representa um benefício imenso, contudo, por outro lado, um prejuízo do mesmo tamanho.

Na prática, os gerentes e diretores sabem que é fácil colocar um novo campo em uma tela. Portanto, caso o programador se recuse, dizendo que precisa fazer uma análise do problema, com muita probabilidade estará no departamento pessoal no fim do mês. Desta forma, programador bom é aquele que vai lá e coloca o novo campo, não importando se daqui a três dias, cinco relatórios deixam de funcionar, e outras partes do sistema entrem em colapso.

Portanto, esta falta de conscientização geral na área de software é causada pela maleabilidade de nosso produto.

Todos sabem que nas outras Engenharias, o acréscimo de novas funcionalidades inicia-se no projetista e passa por vários estudos de viabilidade antes do acréscimo. Em oposição, a área de software simplesmente faz a alteração e cruza os dedos esperando que nada deixe de funcionar.

A falta de encapsulamento e outras características dos softwares atuais, atacados diretamente na teoria CBD, tornam nossos softwares fadados a chamada quebra de manutenção.

Um software atinge o ponto de quebra de manutenção quando qualquer alteração a que seja submetido seja demasiadamente custosa e possua como efeito colateral alterações em funcionalidades já existentes.

## 2.8. Reutilização

A reutilização garante ou permite uma melhor reparabilidade do software bem como garante um maior tempo de vida de um software (evolução). A Reutilização não se restringe ao aproveitamento de código fonte. Esta visão limita demasiadamente os benefícios da reutilização, na verdade até mesmo anula qualquer ganho advindo de tal prática.

A Componentização (CBD) atacou de frente este problema, e vem alcançando êxito em seus esforços. Pois, na Componentização todo processo do software gira em torno de Componentes, assim, o conceito de Reutilização estende-se não só ao código fonte, como também a todos os artefatos presentes desde a especificação até a distribuição do Componente.

Portanto, a reutilização de modelos, requisitos e outros artefatos fazem parte do escopo geral de reutilização de um software, e não só os códigos fonte.

É importante não esquecer, que como qualquer indústria, o aperfeiçoamento de componentes leva a diminuição de custos e a um aumento de qualidade.

Se este requisito não for definido antes ou durante o início da especificação ou programação, sempre teremos uma reinvenção da roda. Assim, este requisito sugere a reutilização de outros artefatos.

## 2.9. Portabilidade

Portabilidade representa a capacidade de um software de ser executado em ambientes diferentes. Entendemos por ambiente um tipo específico de hardware, sistema operacional, rede, banco de dados etc.

Embora a Portabilidade seja um princípio importante em um software, nem sempre é possível atingir este objetivo e, as vezes, os valores de investimentos envolvidos inibem a construção de software para multiplataforma.

A linguagem Java é uma alternativa interessante para a Portabilidade juntamente com a arquitetura .NET. Contudo, depositamos nossas fichas na Model Driven Architecture - MDA, e consequentemente nos modelos executáveis. Discutiremos esta questão no momento apropriado.

Obviamente a Internet permitiu a separação completa entre servidor e cliente. Assim, podemos ter um servidor dependente de plataforma e um cliente não. Esta abordagem também minimiza e permite a portabilidade do ambiente cliente. O que já representa uma vantagem considerável.

## 2.10. Compreensibilidade

A Compreensibilidade expressa o fato de um artefato ser mais compreensível que outro, durante o estudo ou manutenção de um software.

Obviamente o fator determinante de Compreensibilidade aplica-se ao código fonte e ao modelo, se existir. Pois, 70% do tempo gasto na manutenção de um software referem-se ao estudo e entendimento da lógica do programa e modelo atual, para que posteriormente ele possa ser modificado.

Este realmente é um tópico desprezado pelos programadores. Os antigos programadores ou experts acreditam firmemente que basta resolver um problema, e alguns acreditam ainda que quanto mais complexo e menos linhas de código forem feitas, melhor a solução. Minha experiência pessoal me levou a contínuas decepções e desentendimentos com estes programadores, que realmente programam de forma artesanal, e dificilmente enquadram-se em uma visão fundamentada nos princípios de Engenharia de Software.

A utilização de um processo de desenvolvimento formal impõe o uso de padrões, onde inúmeros artifícios são usados para manter um padrão e garantir o entendimento de qualquer artefato. Portanto pequenos padrões são na verdade os verdadeiros responsáveis pela inteligibilidade de nossos artefatos, e sendo assim, eles não podem ser baseados em critério pessoal.

Geralmente compreensibilidade está relacionada ao artefato documentação, entretanto também está relacionada com o código-fonte e modelo.

## 2.11. Interoperabilidade

Interoperabilidade refere-se à capacidade de um sistema em cooperar e coexistir com outros sistemas.

Exemplo típico representa a capacidade de incorporarmos uma planilha eletrônica a um texto em um processador de texto.

Obviamente que as outras Engenharias permitem um alto grau de interoperabilidade. Uma televisão, por exemplo, permite que seja interligada a vídeos cassete, aparelhos de CD, videoquê, e até mesmo aparelhos de som.

Esta capacidade está relacionada à produção de interfaces externas ao produto. Outros produtos podem ser conectados a esta interface, desde que se obedeça a sua especificação.

A capacidade de um determinado software externar interfaces ou assinaturas de funções a serem chamadas por outro software é denominada sistema aberto. Um sistema é aberto (open system) se ele garante sua extensibilidade via publicação de interfaces externas.

O caminho proposto pela abordagem CBD, sem dúvida nenhuma, contém em si os atributos necessários para a criação de softwares com máxima interoperabilidade. Em termos tecnológicos, os framework J2EE e .NET também estão

caminhando nesta direção. E não podemos deixar de citar a proposta CORBA e DCOM que também tentaram atingir este objetivo, mais recentemente temos o SOA que também compartilha desta proposta.

Se existir a necessidade de comunicação com outros sistemas, tanto sob o ponto de vista de receber informações, insumo, como fornecer informações, gerar produtos, deverá constar uma referência aqui e também a forma que esta comunicação será realizada.

## 2.12. Produtividade

Produtividade representa a qualidade relacionada ao processo de produção de um software, visando garantir sua eficiência e seu desempenho sem comprometer as outras qualidades do software.

Assim, a técnica utilizada é fundamental quando falamos em produtividade. Obviamente, o reuso de componentes e outros fatores como Compreensibilidade e Ergonomia também afetam a produtividade.

Se quantificarmos uma determinada tarefa de um software na unidade esf, esforços necessário a sua construção, poderemos obviamente afirmar que cada desenvolvedor possui sua taxa de produtividade medida em esf/h (esforço por hora). Obviamente que a união de vários desenvolvedores em uma mesma equipe interfere na taxa de produtividade da equipe como um todo. Podemos ter um ligeiro aumento na produtividade dos iniciantes, e termos uma considerável redução na produtividade dos programadores mais eficientes. A otimização máxima deverá ser objeto de estudo do gerente da equipe.

O elemento fundamental para medir a Produtividade obviamente é a métrica. Existem várias métricas de software disponíveis atualmente, e até mesmo alguns softwares que automatizam este processo.

Outro fator determinante é a utilização de ferramentas adequadas. Uma determinada ferramenta de programação pode ser mais amigável que outra. Isto vale para todas as ferramentas utilizadas na produção de um software.

Indiscutivelmente todos queremos alta produtividade, contudo existem componentes que necessitam de garantias de construção.

Por exemplo, se alguns estagiários podem ser inseridos ao contexto de desenvolvimento, neste caso podemos definir que este requisito neste componente poderá ser realizado com abaixa produtividade. Por outro lado, um recurso avançado do software ou que exija um alto índice de corretude ou confiabilidade deverá ser realizado por uma equipe sênior onde se espera uma alta produtividade.

## 2.13. Desvio de Planejamento

O Desvio de Planejamento representa a qualidade relacionada ao processo de produção de um software, que visa garantir a entrega dos artefatos no prazo e no custo planejado.

Na década de 60 assistimos a um grande desvio de planejamento nos softwares produzidos. Estes desvios obviamente representavam sempre uma postergação dos prazos, consequentemente um aumento nos custos. Este fator aliado à complexidade dos novos softwares, foi o responsável pela chamada crise do software, que motivou a criação de nossa disciplina, a Engenharia de Software.

Obviamente que podemos aqui comparar nossos desvios com outras engenharias e nos admirarmos com sua precisão. Acredito ser necessário, neste ponto em particular, fazer a defesa e apresentar nossas dificuldades.

Existe uma dificuldade inerente ao software relacionado a sua natureza abstrata. Estamos o tempo todo trabalhando com modelos abstratos da realidade. Nossa dificuldade está em garantir a consistência dos requisitos levantados e estimar o esforço necessário para sua implementação.

Outro fator determinante nos desvios de planejamento refere-se à mudança contínua dos requisitos do sistema, mesmo durante o seu desenvolvimento. Neste caso, estamos atirando em um alvo móvel. Portanto, prever mudanças passa a ser um atributo necessário a nossa disciplina.

O modelo iterativo e incremental minimiza o desvio de planejamento, tendo em vista a eliminação de erros e risco, que é feita na liberação e uso de cada módulo ou componente em particular.

Contudo, o modelo iterativo e incremental só é eficiente, se os requisitos forem agrupados corretamente em subsistemas visando atender às normas de uma boa arquitetura de software.

O quanto é tolerável um atraso no prazo e consequentemente aumento nos valores de investimento é uma questão a ser avaliada na fase de Análise de Requisito

## 2.14. Visibilidade

Um processo de desenvolvimento de software é dito possuir visibilidade, se seu estado atual de desenvolvimento estiver devidamente documentado.

Lembramos nosso processo APD está dividido em 4 ciclos e 11 atividades. E ainda, utilizaremos o desenvolvimento iterativo e incremental, significando que podemos ter equipes em análise de negócio; outra em projeto; outra em implementação e outra em teste. Portanto, em um certo momento, como podemos afirmar em que ciclo e em que atividade encontra-se o desenvolvimento de um software?

O desenvolvimento iterativo e incremental responde apenas onde se encontra o FOCO de desenvolvimento, nada mais. Contudo, até mesmo fornecer esta informação pode ser difícil.

Portanto, visibilidade está relacionada a gerência de projeto, na verdade não só esta. O objetivo real é permitir um acompanhamento do projeto por todos os envolvidos. É por isso que esta qualidade também é denominada transparência.

Obviamente várias são as dificuldades em manter a documentação em dia. Pois, visando evitar retrabalho, portanto poupar recursos e dinheiro, a documentação deverá estar a certo “delay” de um instantâneo do software. E sendo assim, dificilmente estaremos em dia com a documentação.

Outro fator que determina atrasos e mudanças constantes de cronograma está relacionado a entrada e saída de profissionais de desenvolvimento. A mente humana possui uma rede de associações inconsciente que não são captadas pela formalização de nossas técnicas. Isto significa que por mais que um componente esteja documentado, a saída de um profissional requer uma nova catequização do ambiente de negócios circunscrito ao software.

Uma forma eficiente de minimizar consideravelmente este estado de coisas é a utilização de modelos. Um software centrado em modelos resolve não só este, como vários outros problemas.

Define quais serão os pontos de controle que poderão ser utilizados para a equipe de gerência medir o quanto do fim a construção do requisito está. Se um requisito foi estimado para especificação e construção para três dias, como podemos medir no segundo dia se estamos atrasados ou não ?

## 2.15. Segurança

O princípio de Segurança não é universalmente reconhecido como um princípio que mede a qualidade do software. Contudo, com o avanço das tecnologias atuais e nossa vida dependendo cada vez mais dos recursos do software, este princípio passou a ser de extrema importância para os softwares atuais.

Um sistema seguro é aquele que permite o acesso somente a pessoas ou sistemas autorizados. Entretanto, no mundo virtual, facilmente um intruso pode se passar por uma pessoa ou sistema autorizado, invadir e obter informações sigilosas da corporação.

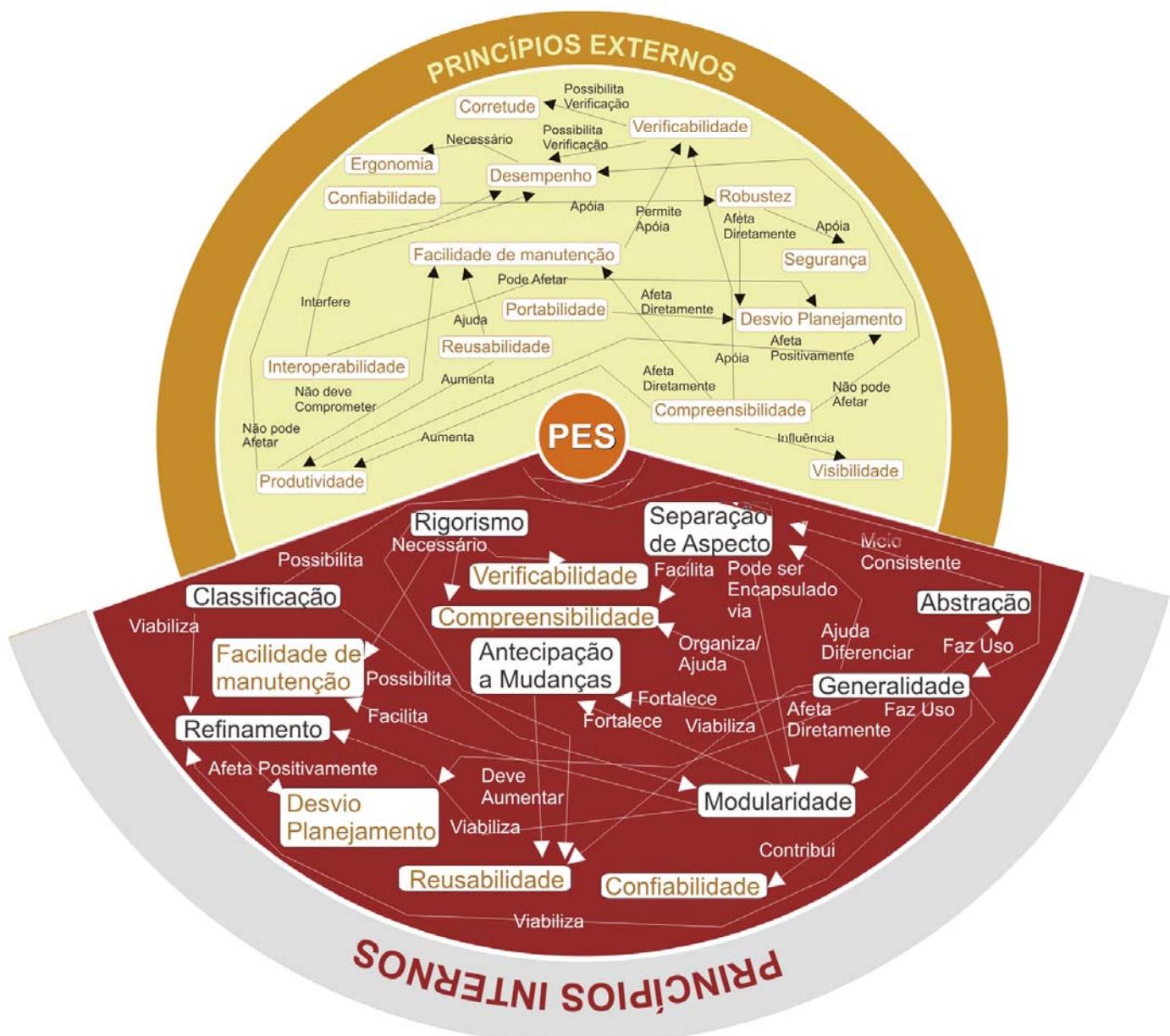
Segurança abrange também um mecanismo embutido de auditoria dos acessos e atualizações do sistema. Assim, cada relatório tirado, cada consulta realizada, cada objeto atualizado deveria teoricamente fazer parte de um mecanismo de auditoria.

Este princípio tende a se tornar forte nos próximos anos. Um software sem segurança, nem que seja mínima, não será mais aceitável, mesmo que seja para finalidades sem risco aparente. Isto porque a informação, seja pessoal ou de negócio, é altamente valiosa e não pode ser objeto de acesso não autorizado.

Define qual o nível de privacidade do requisito deverá ser utilizado como parâmetro visando limitar o acesso. Geralmente é promovido a requisito de ambiente, embora alguns requisitos possam ter graus diferentes de segurança. O framework AWFS deverá controlar todo acesso à informação e registrar cada passo do usuário no sistema. Preferencialmente deverão existir níveis de auditoria distintos. Que vão desde saber quem entrou no sistema, passam pelo que o usuário fez ou acessou, e por fim armazena todas as entidades que foram atualizadas tanto antes quanto depois da atualização. Este mecanismo deve ser transparente, sem a participação direta do programador. O mecanismo de auditoria também pode ser usado para medir desempenho, performance, pois poderá armazenar quanto tempo durou cada operação macro do software.

### **3. Princípios Internos**

Os princípios internos de um software estão ligados diretamente à qualidade técnica do produto final. Forcem um guia que deverá estar presente em praticamente todas as técnicas de um processo de desenvolvimento de software. Juntamente com os princípios externos, e pela mesma razão, estes princípios serão considerados nossos axiomas. Tentaremos, sempre que possível, construir técnicas que levem estes princípios em consideração. Sendo desnecessário, portanto, a boa vontade dos desenvolvedores e programadores em seu cumprimento.



### 3.1. Rigorismo

Rigorismo representa a atividade oposta a criatividade. Assim, refere-se à precisão à exatidão que devem ser aplicadas nas atividades presentes em um processo de desenvolvimento.

Neste princípio temos a eterna luta entre a criatividade, necessária ao desenvolvimento de um software, e o rigor inerente a qualquer disciplina de Engenharia.

Sem rigor na aplicação das técnicas colocaremos em risco no mínimo os princípios de facilidade de manutenção, reusabilidade, comprehensibilidade, verificabilidade e confiabilidade. Não podemos deixar de mencionar o desvio de planejamento, princípio que se negligenciado poderá levar a falência do software.

Por outro lado, a criatividade preenche os espaços onde o rigor perde o apoio. A criatividade permite que o rigor possa ser utilizado em sua plenitude. O aparente paradoxo é desfeito quando enxergarmos o rigor sendo aplicado a problemas específicos e fechados. Contudo, a criatividade permite que vários destes problemas sejam solucionados de forma encadeada, e assim o rigor e a criatividade se reforçam mutuamente.

Todavia, na prática, dificilmente encontramos um desenvolvedor que conhece estes limites. Geralmente o desenvolvedor apóia irrestritamente a criatividade ou apóia irrestritamente o rigor.

No primeiro caso, temos que tentar empreender uma catequização do desenvolvedor, contudo, na maioria dos casos, os resultados são negativos, ou seja, o desenvolvedor não se adequa a um processo de desenvolvimento com base em Engenharia de Software. Neste caso, é melhor trocar de profissional antes que seja tarde demais.

No segundo caso, estamos na verdade saindo do rigor e entrando na formalidade, ou seja, um software formal regido por leis matemáticas. É desnecessário dizer que esta visão está fadada aos meios acadêmicos. Obviamente que se estamos resolvendo um problema matemático, físico ou algo relacionado, teremos que usar o devido formalismo. Contudo, geralmente não é este o caso, o formalismo excessivo irá aumentar consideravelmente o tempo de desenvolvimento do software, bem como seu custo. Portanto, na maioria das vezes, não vale a pena seguir este caminho.

Um software deverá ser desenvolvido utilizando-se um processo de desenvolvimento. Portanto, cabe ao processo de desenvolvimento estipular técnicas e métodos rigorosos o suficiente, para que o rigorismo seja alcançado. Pois, não podemos esperar que este princípio, aliás como todos os outros, fique a cargo de cada desenvolvedor. Assim, o rigorismo seria aplicado a todas as atividades de um processo de desenvolvimento, e não somente a programação.

A UML é um exemplo de aplicação do rigorismo. Pois, somente em alguns casos raros, o formalismo matemático foi utilizado. Assim, para aqueles que tiveram acesso a sua definição, ficará claro o quanto o rigorismo por si só é eficiente.

### 3.2. Separação de Aspectos

Como em toda Engenharia, durante o desenvolvimento de um software nos deparamos com vários problemas. Estes problemas, embora estejam de certa forma entrelaçados, podem ser filtrados e resolvidos separadamente. O princípio que estabelece a separação dos aspectos de um determinado problema, de forma a permitir a aplicação de técnicas específicas para sua solução é denominada Separação de Aspectos.

A Separação de Aspectos também é conhecida como Dimensões de um software, ou Hiperespaços de um software, contudo iremos fazer a devida distinção. Iremos propor uma definição e uma relação maior entre estes termos, de forma a retirar suas ambiguidades e fornecer uma semântica determinada para cada um deles.

Importante ressaltar que um problema é expresso através de objetos e dos relacionamentos entre objetos. Portanto, separar os aspectos de um problema significa separar os objetos presentes neste problema e seus relacionamentos.

**Objeto:** Possui uma representação em um **hiperespaço** ou **universo**. Um **hiperespaço** está dividido em **dimensões**. Uma **dimensão** está dividida em **aspectos**. Um **aspecto** está dividido em **camadas**.

Com o único intuito de exemplificar uma possível aplicabilidade destes conceitos, estaremos assumindo um objeto sendo o equivalente a um componente de um software. O importante aqui é ter em mente que em CBD, todo o processo gira em torno de um componente.

Um componente poderia ser dividido em 3 hiperespaço:

1. Hiperespaço Técnico
2. Hiperespaço Gerencial
3. Hiperespaço Artefato

Poderíamos subdividir o Hiperespaço Técnico nas seguintes dimensões:

1. Dimensão Atômica
2. Dimensão Primitiva
3. Dimensão Negócio
4. Dimensão Condicionante
5. Dimensão Existencial

Pode-se imaginar, por alguns instantes, que estamos modelando algum software relacionado à conta corrente de um banco. Portanto, nosso componente alvo chama-se conta corrente. Contudo, o exposto aqui poderá ser aplicado a qualquer Componente. Obviamente, estamos de certa forma, admitindo um desenvolvimento baseado em componentes (CBD) e orientação a objetos.

Vamos então analisar superficialmente cada uma das dimensões acima. Não poderá retirar da mente, em nenhum momento qualquer, o caráter imaginativo deste nosso exemplo.

## Dimensão: Atômica

O que queremos dizer com Atômica? Estamos realmente querendo denotar uma idéia de indivisibilidade, como um átomo da física ou uma célula da biologia. Incluindo a constatação de que mesmo dando a idéia de indivisibilidade, pode ser dividido em porções menores, como o átomo e a célula. Contudo, estas porções menores perdem o vínculo com o todo, portanto não serão objetos de nosso estudo.

Poderíamos dizer que um componente em sua dimensão atômica pode ser representado ou é equivalente a um objeto da teoria de orientação a objetos.

Durante as últimas décadas o desenvolvimento de software foi confinado a esta única dimensão. Por isso, a dimensão atômica é a única dimensão valorizada pela comunidade de software. As linguagens de programação, as linguagens de modelagem e todas as ferramentas de desenvolvimento visam suportar esta dimensão.

Dividiremos ainda esta dimensão em 3 aspectos:

1. Estrutura
2. Comportamento
3. Comunicação

### Aspecto: Estrutura

A estrutura de um componente representa a forma de vermos um componente através de seus atributos, valores e seus relacionamentos.

### Aspecto: Comportamento

O comportamento representa a forma de vermos um componente através de sua reação a estímulos internos ou externos.

### Aspecto: Comunicação

A comunicação representa a forma com que um componente se comunica com o outro. A forma padrão é o método em uma linguagem de programação. Porém, numa visão contratual podemos subdividi-la em aberta – XML- e fechada – assinatura - , que representa uma visão mais produtiva.

## Dimensão: Primitiva

A dimensão Primitiva representa os aspectos que estão presentes na maioria dos softwares, contudo não são diferenciados, e na maioria das vezes se misturam entre si e com outras dimensões. O resultado é um empobrecimento significativo do modelo final, e um impacto destruidor na manutenção e extensão do software construído, isto quando o software em si consegue sobreviver ao desenvolvimento.

A abordagem correta seria o tratamento diferenciado de cada aspecto primitivo. Seu entrelaçamento com a dimensão condicional seria interessante para evitar desvios dos profissionais que produzem o software.

Uma possível divisão em aspectos seria:

- Aspecto: Tratamento Erro
- Aspecto: Interface com Usuário
- Aspecto: Auditoria
- Aspecto: Evento
- Aspecto: Persistência
- Etc.

## Dimensão: Negócio

A dimensão do negócio representa a modelagem propriamente dita do problema. Contém os componentes do negócio que estão sendo modelados.

Esta visão é geralmente captada pela dimensão existencial, contudo desprezar os vários aspectos presentes em cada setor é um erro grave. Um mesmo componente é visto de forma diferente dependendo do ponto de vista adotado, ou mais concretamente, por exemplo, dependendo de cada setor da empresa. Além da variação setorial, podemos ter uma variação na própria granularidade das regras envolvendo cada componente. Uma regra aplicada a um componente em particular representa uma regra atômica. Uma regra aplicada a um componente e seus relacionados é dita ser uma regra gerencial. E por fim, uma regra envolvendo vários componentes em um processo é denominada regra colaborativa.

Uma possível divisão em aspectos seria:

- Aspecto: Marketing
- Aspecto: Administração
- Aspecto: Engenharia
- Etc.

## Dimensão: Condicionante

A dimensão condicionante representa uma solução pré-moldada envolvendo um conjunto de componentes. Desta forma, ao nascer um componente será condicionado a funcionar exatamente igual a um padrão de componente pré-estabelecido.

Um conjunto de padrões operando conjuntamente pode dar origem aos Frameworks. As aplicações estarão completamente condicionadas à eles em um futuro próximo, e esta dimensão tende a crescer rapidamente.

Veja que nada impede que a dimensão primitiva seja condicionada em seu funcionamento, e na verdade esta é a principal aplicação da dimensão condicional.

Uma possível divisão em aspectos seria:

- Aspecto: Padrões
  - Camadas: Negócio
  - Camadas: Analise
  - Camadas: Projeto
- Aspecto: Framework
- Aspecto: Estilo Arquitetural

## Dimensão: Existencial

Esta dimensão está relacionada a existência de um componente. Para que um componente exista devemos percebê-lo. Por exemplo, existe a percepção real de um componente garantido pelos nossos sentidos, ou seja, quando vejo um Navio no mar.

Existe a especificação do componente, por exemplo, o modelo e o código de um Navio em formato computacional.

Existe a execução deste componente em um Ambiente de Execução de Componentes – AEC

Uma possível divisão em aspectos seria:

Aspecto: Abstração

Aspecto: Realidade

Aspecto: Especificação

Aspecto: Execução

    Camada: Usuário

    Camada: Transporte

    Camada: Servidor

    Camada: Banco de Dados

O Hiperespaço Gerencial poderá ser dividido em várias dimensões, como controle de mudanças e controle de versão, entre outras. O mesmo se pode dizer dos artefatos.

Veja que vários autores irão propor aspectos diferentes para abordar um software. O importante aqui é conscientizar de que geralmente em um curto programa de dez linhas de código talvez, vários aspectos estejam sendo abordados simultaneamente.

Esta falta de consciência eleva a complexidade de um software à potência máxima.

### 3.3. Modularidade

A regra geral para resolver problemas complexos é dividi-los em partes menores e resolvê-los independentemente. Esta regra é conhecida como “dividir para conquistar”.

Em software chamamos esta subdivisão de módulos, um software que atende a este princípio é dito ser modular.

Neste momento iremos substituir o termo módulo por componente, o relacionamento entre os módulos será denominado arquitetura de software. Contudo, para que não sejamos alvos de críticas, lembrando que os princípios de Engenharia de Software são independentes das técnicas, tentaremos manter o termo módulo apenas neste tópico.

Algumas autoridades mundiais relacionam modularidade à separação de aspectos. Este é um dos conceitos onde existe um abismo entre a teoria e a prática. Em software, dificilmente aspectos e modularidade se combinam. Isto porque, geralmente, vários aspectos estão presentes e mesclados no mesmo produto final. A separação de aspectos possui validade como consciência ou na dimensão de condicionante apresentada acima. Portanto, separar o tratamento do erro, dos erros que ele trata, simplesmente não faz sentido.

Como era de se esperar, as outras Engenharias, atingiram altos níveis de modularização em seus produtos. Um carro, por exemplo, possui vários módulos. Cada um pode ser fabricado independentemente, sendo a junção final o fator crítico de sucesso da modularização.

A divisão de um grande módulo em vários módulos menores, ou melhor, de um grande componente (software) em vários componentes menores, é denominado arquitetura de software e representa a pedra angular do fracasso ou sucesso de um software.

Exemplificando, em um carro temos vários componentes. Vamos imaginar que foi definida a existência de um componente denominado Assento. Com certeza, este componente será especificado internamente em relação as suas funcionalidades. Porém, seu relacionamento externo ou arquitetural com o todo (carro) também deve ser especificado. Pois, corremos o risco do Assento ser alto demais ou baixo demais e não encaixar na arquitetura geral do carro.

Neste sentido, um componente é dependente da arquitetura. Todavia, um componente está mais relacionado com um grande módulo de um carro, do que com um transistor em engenharia eletrônica. Obviamente, poderíamos indagar se os componentes de alta granularidade não seriam mais reutilizáveis; os de baixa granularidade menos reutilizáveis. Concordamos com esta abordagem, contudo ainda assim os componentes são construídos para operarem em um AEC específico, o que limita também o reuso do binário final.

Existem basicamente quatro conceitos fundamentais relacionados aos módulos, são eles, coesão, acoplamento, granularidade e recursividade.

O ideal é termos módulos coesos, com baixo acoplamento, sem recursividade e com granularidade média. Contudo, existem aqui alguns “trade-offs” (ganha aqui perde ali) que temos que gerenciar. Para aumentar a coesão, as vezes, temos que diminuir a granularidade, ou seja, aumentar consideravelmente o módulo. O mesmo vale nos casos onde queremos baixar o acoplamento entre módulos.

Podemos manter o nível de coesão e acoplamento no ótimo, entretanto, desde que tenhamos recursividade entre os módulos. O custo de recursividade é muito alto. Portanto, estas questões são fundamentais para garantir a correta subdivisão de módulos e garantir o sucesso do software através da definição de uma boa arquitetura de software.

### 3.4. Abstração

A Abstração permite que problemas complexos sejam vistos e analisados em níveis distintos de profundidade. Uma abstração fortalece e torna relevante apenas alguns aspectos, ocultando detalhes desnecessários.

Obviamente os aspectos que serão analisados e os detalhes que serão omitidos dependem da visão imposta pela abstração. Assim, podemos ter várias visões abstratas para o mesmo problema, cada uma delas ressaltando ou omitindo alguns aspectos ou detalhes respectivamente.

Podemos então diferenciar dois tipos de abstração. A Abstração horizontal representa várias abstrações do mesmo problema. A abstração vertical representa uma descida ao nível de detalhes de um único problema, partindo do problema em direção a sua completa implementação.

Um problema inicialmente é formalizado no contexto de seus processos e paralelamente através dos chamados requisitos funcionais. Estes requisitos são transformados ou ligados diretamente aos modelos. Estes modelos serão construídos usando a UML, unificando os conceitos de processo, componentes, regras e objetos.

Via abstração vertical poderemos dar o chamado “zoom out/in” e caminhar na máxima abstração expressa nos processos e requisitos, passar pelos modelos, descer nos componentes e chegar as classes e sua implementação. Esta visão fechada é facilitada no desenvolvimento CBD, onde tratamos um único componente como um mundo próprio. Já em orientação a objetos as soluções são esparsas: o tudo ligado com o tudo. Portanto, rapidamente um requisito se perde nos meandros da complexidade de um grande software.

A Abstração deve e pode ser aplicada em qualquer atividade de um processo de desenvolvimento. Na atividade de implementação, alguns programadores sentem dificuldade extra em trabalhar com abstração. Esta dificuldade resulta em códigos enormes, complexos e de difícil manutenção até mesmo para quem os desenvolveu. A modularização do código juntamente com a abstração, fornece um meio eficaz de separar os aspectos relevantes de cada nível do problema, e via criação de métodos privados e encapsulados, garantir um eficiente mecanismo de abstração.

Obviamente o mecanismo de herança permite também que vários níveis de complexidade sejam decompostos hierarquicamente. Esta decomposição da complexidade e alguns outros fatores, como o polimorfismo, fazem da orientação a objetos uma poderosa aliada na garantia de levar o princípio de abstração até o código fonte.

### 3.5. Generalidade

Generalidade refere-se ao princípio de solução do problema ontológico (aquilo que é) ao invés do epistemológico (aquilo que parece ser). Obviamente esta definição está um pouco filosófica para nossos fins. Assim, iremos alimentar nossa definição com uma pequena discussão relacionada a um problema real. Todo problema encerra em si um conjunto de outros problemas mais genéricos. Imagine que você seja contratado para implementar um arquivo de troca de informação de cobrança bancária (CNAB).

Ao estudar o problema você verifica que todos os bancos de uma forma ou de outra trabalham com o formato CNAB. Portanto, você terá duas opções. Ou implementa diretamente o formato CNAB do banco em questão, ou prevê futuramente a cobrança com novos bancos. Neste caso, você deverá resolver dois problemas. Um seria generalizar sua solução para qualquer formato CNAB. O outro, seria fornecer meios de personalizar a solução genérica para cada banco em particular. Este último mecanismo é conhecido em modelagem como ponto de variação, e será discutido no devido tempo.

Obviamente, poderíamos generalizar ainda mais a solução, prevendo uma solução qualquer de comunicação via arquivos. Onde, a troca de arquivos via CNAB, seria somente mais um tópico ou especialização de uma solução maior de troca de arquivos.

A grande vantagem deste processo é o reuso e a confiabilidade da solução. Pois, uma vez implementado para um Banco, poderemos reusá-lo para qualquer novo banco, com garantia de funcionamento do código genérico.

A desvantagem seria um aumento do prazo e valores de investimento. Pois, uma solução deste tipo, exige um projeto e uma análise mais detalhada do problema.

Às vezes, a divisão de um problema em partes genéricas e específicas nos leva a componentes prontos no mercado, sendo assim, podemos comprá-los ao invés de implementá-los.

Indiretamente a Generalidade também produz o efeito de latência zero. Ou seja, software com alto grau de Generalidade, são mais fáceis de serem adaptados às novas regras do negócio e assim o princípio de Antecipação de Mudanças é fortalecido.

### 3.6. Antecipação a Mudanças

Em um passado não tão distante assim, analistas de sistemas sentiam-se justificados após uma reunião com o usuário. O motivo era a afirmativa em tom de plena superioridade: Vai atrasar sim, pois o usuário mudou tudo. Não era nada daquilo. Vou ter que refazer tudo...

Contudo, na maioria dos casos, não em todos, esta frase condena mais o analista que o usuário. Isto porque cabe ao analista, levantar não só as necessidades expressas pelo usuário, mas também garantir a consistência destas mesmas demandas. Muitos analistas consideram como verdade a palavra do usuário. Este erro custa caro, pois certamente a forma com que o problema se apresenta ao usuário é diferente da forma como o problema se apresenta ao analista. Para o primeiro, ambigüidades e inconsistências não são percebidas, para o segundo, a imposição da lógica de programação, torna impossível tais condições.

Somando-se a este cenário temos a constatação de que um software geralmente está em contínua evolução. Portanto, diferente das outras indústrias, a indústria de software produz um produto para ser modificado. O produto é mutante e com certeza será otimizado e aperfeiçoado ao longo dos anos.

As outras indústrias também permitem tal evolução. Contudo, não é trocando peças que você fará um “upgrade” de sua televisão. Para adquirir uma televisão com novas tecnologias forçosamente você terá que comprar outra inteiramente nova. Em software, podemos, ao longo do tempo inserir novas funcionalidades ao sistema, alterar ou retirar funcionalidades antigas. Isto sem desativar nem parar o software original. Embora pareça natural, esta capacidade é um diferencial somente presente em nossa indústria. No entanto, não é por milagre nem mágica, nem por um acidente de percurso, que um software permitirá uma contínua evolução de suas funcionalidades.

Visando garantir a contínua evolução do software e aumentar sua capacidade de absorver mudanças, devemos nos antecipar a elas, ou projetar o sistema já esperando por elas.

Um mecanismo que ajuda a suportar este princípio é denominado ponto de variação e está presente na UML 2.0. Podemos prever explicitamente um ponto de variação em uma determinada solução em um software. Esta variação representa uma referência explícita de mudança nas funcionalidades de um sistema.

Obviamente os mecanismos de Generalização e Modularidade deverão ser utilizados conjuntamente para fortalecer o princípio de Antecipação de Mudanças. Um software projetado para mudar é um software projetado para evoluir. E ainda, um componente projetado para mudar é muito mais útil que um componente congelado. Portanto, Antecipação de Mudanças aumenta a reusabilidade.

### 3.7. Refinamento

Refinamento refere-se à capacidade de resolvemos um problema em passos incrementais de funcionalidades.

Imagine, por exemplo, que iremos contratar um pintor para pintar uma paisagem. Iremos contrastar duas formas de realizar esta pintura. Na primeira, o pintor decide pintar dos cantos superiores e inferiores para o centro. Neste caso, somente no final da pintura teremos uma idéia do todo. Toda pintura fará sentido. O pintor tem o quadro na cabeça, contudo, somente em incrementos sucessivos ele passa para o papel. Somente ele sabe para onde está indo e qual a forma final. Olhando de fora, dificilmente poderemos dizer o que será preenchido em um espaço vazio que ainda não foi pintado. No segundo, o pintor decidiu marcar os pontos estratégicos da pintura, realizando o desenho com lápis, sem efetivamente iniciar a pintura. No entanto, mesmo sem iniciar já temos a idéia do todo. A cada incremento na pintura, novas partes do quadro irão ganhando vida.

No primeiro caso, teremos praticamente que esperar a pintura terminar, para decidirmos se gostamos ou não do quadro. No segundo, logo no início podemos julgar a pintura, e caso sejamos o comprador, podemos solicitar mudanças ao pintor, de acordo com nosso gosto pessoal.

Este princípio diminui enormemente os riscos de falha em um projeto. Alguns desenvolvedores limitam o uso deste princípio somente a protótipos de tela. Iremos utilizar este princípio em todas as atividades de nosso processo de desenvolvimento.

Na prática, este princípio torna os programadores desconfiados. Pois, muitos problemas transformam-se em regras de objetos. Estas regras devem ser inseridas no sistema vagarosamente, mesmo porque, se todas as regras de um componente forem inseridas, o sistema fica sem possibilidade de teste, pois outros componentes ainda estão em desenvolvimento, e várias pré-condições irão falhar. No entanto, estes mesmos programadores não conseguem dominar sua ansiedade, insistem em programar até o último bit. Muita catequização é necessária.

Uma sugestão para documentar partes do sistema que precisam de refinamento é colocar o mnemônico TBD (To Be Determined) seguido de C (crítico), I (importante) ou U (útil). Assim TBDC significa necessidade de um refinamento crítico para o funcionamento do componente. Este princípio está relacionado diretamente com o custo do projeto. A não aplicação deste princípio, faz com que um trabalho de uma semana, ao ser apresentado, seja condenado. Estimo um prejuízo de no mínimo 50% nos projetos que não se utilizam de refinamentos sucessivos em suas técnicas.

Agrava-se a isso, ao gosto pessoal do programador, que as vezes em um simples protótipo de tela, constrói uma combo utilizando uma estrutura de árvore binária, e tenta durante a apresentação dar ênfase total a combo, afinal importante para ele é a combo, pois esta deu trabalho. Contudo, na maioria das vezes, a combo nem mesmo era necessária, e o usuário poderá pedir sua retirada. Neste caso, vários dias de programação foram perdidos. Isto é equivalente a jogar matéria prima no lixo, se compararmos com outras engenharias. Alerto para que os coordenadores de projeto fiquem atentos e façam deste princípio sua regra de ouro.

### 3.8. Classificação

Classificação está relacionado à formalização de conhecimento em um universo de discurso em particular, através do agrupamento de entidades que possuem propriedades semelhantes.

A função da classificação em um software é agrupar entidades que possuem propriedades parecidas. A importância da classificação advém do método prático e intuitivo de agruparmos entidades. Por exemplo, digamos que tenhamos um Leão, um Cachorro, um Gato, uma Bota, uma Sandália e um Tênis. Inicialmente classificar estas entidades em 2 grupos. Intuitivamente um grupo Animal e outro grupo de Calçados podem ser inferidos.

Esta aparente simplicidade é responsável em garantir nada menos do que a arquitetura de software. Ou seja, após a Classificação podemos aplicar os princípios de Abstração, Generalidade e Modularidade. Estes princípios são os princípios chaves de modelagem e componentização.

Veja que utilizamos a palavra intuitivo no sentido de verdade desprovida de propósito. Pois, veremos na área de recursos humanos que nem todo profissional é um bom modelador. Aquele profissional que procura a todo instante se impor aos demais e fazer valer sua verdade, distorce constantemente o modelo segundo suas concepções pessoais. E poderia até mesmo dividir o grupo acima em Feio e Bonito, colocando o Cachorro, Gato e Tênis como Bonito e os outros como Feio. E ninguém seria capaz de convencê-lo de que Animal e Calçado são os grupos mais intuitivos.