Modelos de Processo de Software

12 de novembro

2011

Documento elaborado com parte do material utilizado para ministrar as aulas de Engenharia de Software na UNIP. Agrupando o conteúdo em um único documento eu quis facilitar o estudo e o entendimento dos alunos sobre o assunto.

Professora Rosineide Aparecida de Lira Volgarin

Sumário

O que são Modelos de Processo de Software

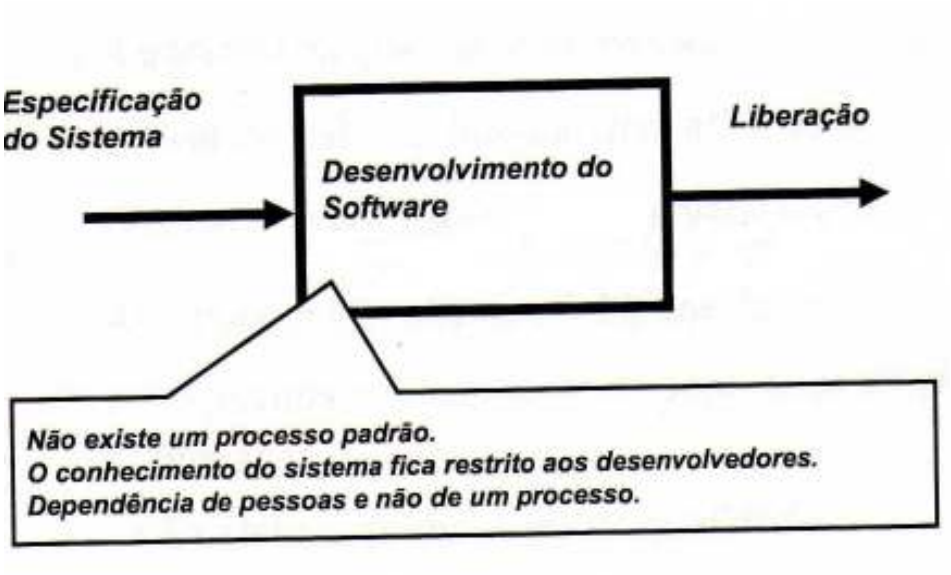
Um modelo de processo de software é uma representação abstrata de um processo de software. Um processo de software é um conjunto de atividades que leva à produção de um produto de software.

Cada modelo de processo representa um processo sobre determinada perspectiva e, dessa forma, fornece somente informações parciais sobre esse processo.

Esses modelos não são descrições definitivas de processo de software, como já dito anteriormente, são abstrações do processo que podem ser usadas pra explicar diferentes abordagens para o desenvolvimento de software. Para muitos sistemas de grande porte, naturalmente, não existe apenas um processo de software que possa ser utilizado. Processos diferentes são utilizados para desenvolver diferentes partes do sistema.

Modelo Caixa-Preta ou Codifica – Remenda

A figura que caracteriza esse processo é:



- Neste modelo com muita sorte existe uma especificação por escrito.
- A codificação é iniciada imediatamente e os erros são corrigidos na medida em que são encontrados.
- Este modelo não oferece nenhum mecanismo de gerenciamento, a não ser liberar um produto depois que os volumes de defeitos críticos tenham sido reduzido a níveis aceitáveis.
- A garantia da qualidade do software é restrita a teste de níveis de sistema (testes do tipo caixa –preta, aquele que você entra com um dado e espera um resultado).

- Essa abordagem é inaceitável para alguns cenários:
  - Requisitos que implicam na segurança de vidas humanas;
  - Existência de requisitos de qualidade;
  - Previsão de custo e cronograma preciso.

Modelo Waterfall – Cascata

O primeiro modelo de processo de desenvolvimento de software publicado originou-se dos processos mais gerais de engenharia de sistemas. Devido ao encadeamento de uma fase com a outra, esse modelo é conhecido como modelo cascata ou modelo de ciclo de vida de software.

Para ocasiões em que os requisitos de um problema são razoavelmente bem compreendidos – quando o trabalho flui de modo razoavelmente linear, isso algumas vezes acontece em uma adaptação bem definidas de sistemas já existentes ou aperfeiçoamento dos mesmos. O modelo cascata pode ser melhor utilizado.

Os principais estágios desse modelo demonstram as atividades fundamentais de desenvolvimento:

- *ANÁLISE E DEFINIÇÃO DE REQUISITOS*: Os serviços, restrições e objetivos são definidos em conjunto com os usuários.
- *PROJETO DE SISTEMA DE SOFTWARE*: Divide os requisitos em hardware e software, estabelece uma arquitetura geral do sistema.
- *IMPLEMENTAÇÃO E TESTE DE UNIDADE*: O projeto de software é realizado, e testes unitários são realizados.
- *INTEGRAÇÃO E TESTE DE SISTEMA*: As unidades individuais do sistema são integrados e testados como um sistema completo para garantir que os requisitos foram atendidos.
- *OPERAÇÃO E MANUTENÇÃO*: Geralmente é a fase mais longa. O sistema é instalado e colocado em operação. A manutenção envolve corrigir erros que não foram descobertos em estágios anteriores do ciclo de vida, melhorando a implementação das unidades de sistema e aumentando as funções desse sistema à medida que novos requisitos são descobertos.

Em princípio o resultado de cada fase resulta em um ou mais documentos aprovados. A fase seguinte não pode começar sem que a anterior tenha terminado. Porém na prática esses estágios se sobrepõem e trocam informações entre si.

Devido aos custos de produção e aprovação de documentos, as iterações são onerosas e envolvem um retrabalho significativo. É norma que depois de algumas iterações suspender parte dos desenvolvimentos com as especificações e prosseguir com os estágios posteriores do desenvolvimento.

Durante a fase final do ciclo de vida o software é colocado em uso. Erros e omissões nos requisitos originais de software são descobertos. Os erros de programação e projeto emergem e a necessidade de novas funcionalidades é identificada.

A vantagem do modelo cascata consiste na documentação produzida em cada fase e a sua aderência a outros modelos. Seu maior problema é a divisão inflexível do projeto em estágios distintos. Os acordos devem ser feitos em um estágio inicial do processo, e isso significa que é difícil responder aos requisitos

do cliente, que sempre se modificam. *Portanto, o modelo em cascata deve ser utilizado somente quando os requisitos forem bem compreendidos. Contudo, ele reflete a prática da engenharia.*

Hoje em dia com o ritmo rápido dos desenvolvimentos de software e as grandes torrentes de modificações de requisitos que os softwares sofrem o modelo cascata é inviável. *Os processos de software com base nessa abordagem ainda são utilizados no desenvolvimento de software, em particular quando fazem parte de um projeto maior de engenharia de sistemas.*

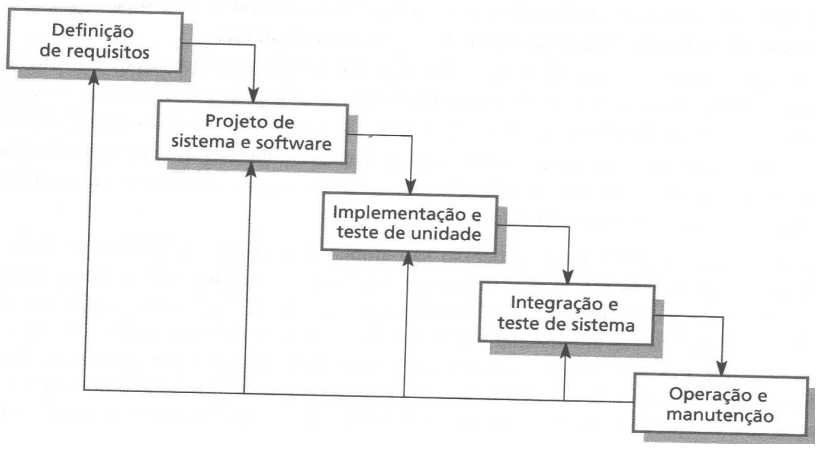


Figura 1 – Ciclo de vida do software

Iteração de Processo

Todos os modelos de processo apresentam vantagens e desvantagens. Para a maioria dos grandes sistemas, existe a necessidade de utilizar diferentes abordagens para diferentes partes do sistema, de maneira que um modelo híbrido tem de ser utilizado. Além disso, há também a necessidade de apoiar o processo de iteração, em que partes do processo são repetidas, à medida que os requisitos do sistema evoluem. O projeto de sistema e o trabalho de implementação devem ser refeitos, a fim de implementar os requisitos modificados.

Veja um exemplo de dois modelos híbridos, que apóiam diferentes abordagens do desenvolvimento e que foram explicitamente projetados para apoiarem a iteração de processo:

1. Desenvolvimento incremental, quando as fases de especificação, projeto e implementação de software são desdobradas em uma série de estágios, que são, por sua vez, desenvolvidos;
2. Desenvolvimento em espiral, quando o desenvolvimento do sistema evolui a partir de um esboço inicial, em direção ao sistema final desenvolvido.

A essência dos processos iterativos é que a especificação é desenvolvida em conjunto com o software. Contudo, isso entra em conflito com o modelo de suprimento de muitas organizações, em que a especificação completa do sistema faz parte do contrato para o desenvolvimento do sistema. Na abordagem gradativa, não há uma especificação completa de sistema, até que o estágio final seja especificado. Isso requer um novo tipo de contrato, que os grandes clientes, como os órgãos de governo, podem ter dificuldade em aceitar.

Modelos Incrementais de Processo de Software

Há muitas situações em que os requisitos iniciais do software são razoavelmente bem definidos, mas o escopo global do esforço de desenvolvimento elimina um processo puramente linear. Além disso, pode haver uma necessidade compulsiva de fornecer rapidamente um conjunto limitado de funcionalidades em versões subseqüentes do software. Em tais casos, um modelo de processo que é destinado a produzir o software em incrementos é escolhido.

Modelo Incremental

O modelo de desenvolvimento em cascata requer que os clientes de um sistema se comprometam com um conjunto de requisitos específicos, antes do início do projeto, e que os projetistas se comprometam com estratégias de projeto em particular, antes da implementação. As mudanças nos requisitos durante o desenvolvimento requerem o 'retrabalho' referente aos requisitos, ao projeto e à implementação. Contudo, as vantagens do modelo em cascata são as de um modelo simples de gerenciamento, e sua separação de projeto e implementação devem levar a sistemas robustos, que são suscetíveis a mudanças.

Por outro lado, a abordagem do desenvolvimento evolucionário permite que os requisitos e as decisões de projeto sejam postergados, mas também leva a um software que pode ser mal-estruturado e de difícil compreensão e manutenção. O desenvolvimento incremental é uma abordagem intermediária, que combina vantagens de ambos esses modelos.

O modelo de desenvolvimento incremental combina elementos do modelo cascata aplicado de maneira iterativa. Aplica seqüências lineares de uma forma racional à medida que o tempo passa.

Com o modelo incremental é possível entregar parte do projeto, ou chamada parte de funcionalidades básicas de um projeto, antes que o mesmo seja concluído na sua totalidade.

Quando um módulo incremental é usado, o primeiro incremento é freqüentemente chamado de núcleo do produto. Isto é os requisitos básicos são satisfeitos, mas muitas características suplementares deixam de ser elaboradas.

O processo de desenvolvimento incremental tem uma série de vantagens:

1. Os clientes não precisam esperar até a entrega do sistema inteiro para se beneficiarem dele.
2. Os clientes podem usar incrementos iniciais como protótipos e ganhar experiência, obtendo informações sobre os incrementos posteriores.
3. Existe um risco menor de falha geral do projeto

No entanto, existem problemas com a entrega incremental. Os incrementos devem ser relativamente pequenos (composto de não mais de 20 mil linhas de código), e cada incremento deve entregar alguma funcionalidade do sistema. Pode, portanto, ser difícil mapear os requisitos do cliente em incrementos do tamanho adequado.

Como os requisitos não são definidos detalhadamente até que um incremento seja implementado, pode ser difícil identificar os recursos comuns exigidos por todos os incrementos.

Uma recente evolução dessa abordagem incremental, chamada de 'programação extrema' (*extreme programming*), foi desenvolvida (Beck, 1999). Ela tem como base o desenvolvimento e a entrega de incrementos de funcionalidade muito pequenos, o envolvimento do cliente no processo, a constante melhoria do código e a programação impessoal.

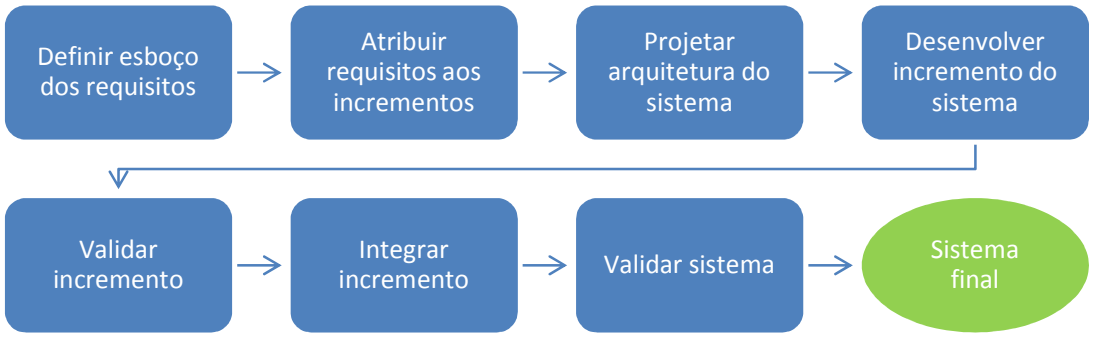


Figura: Desenvolvimento incremental

O Modelo RAD

O RAD (Rapid Application Development, Desenvolvimento Rápido de Aplicação) é um modelo de processo de software incremental que enfatiza um ciclo de desenvolvimento curto. O Modelo RAD é uma adaptação "de alta velocidade" do modelo em cascata, no qual o desenvolvimento rápido é conseguido com o uso de uma abordagem de construção baseada em componentes. Se os requisitos forem bem compreendidos e o objetivo do projeto for restrito, o processo RAD permite a uma equipe de desenvolvimento criar um "sistema plenamente funcional", dentro de um período de tempo muito curto (de 60 a 90 dias).

Como outros modelos de processo, a abordagem RAD se enquadra nas atividades genéricas de arcabouço já estudadas anteriormente:

No modelo RAD a comunicação trabalha para entender os problemas do negócio e as características informais que o software precisa acomodar.

O planejamento é essencial, porque várias equipes de software trabalham em paralelo em diferentes funções do sistema.

A modelagem abrange três das fases principais – modelagem de negócios, modelagem dos dados e modelagem dos processos.

A construção enfatiza o uso de componentes de software preexistentes e a aplicação de geração de códigos automática.

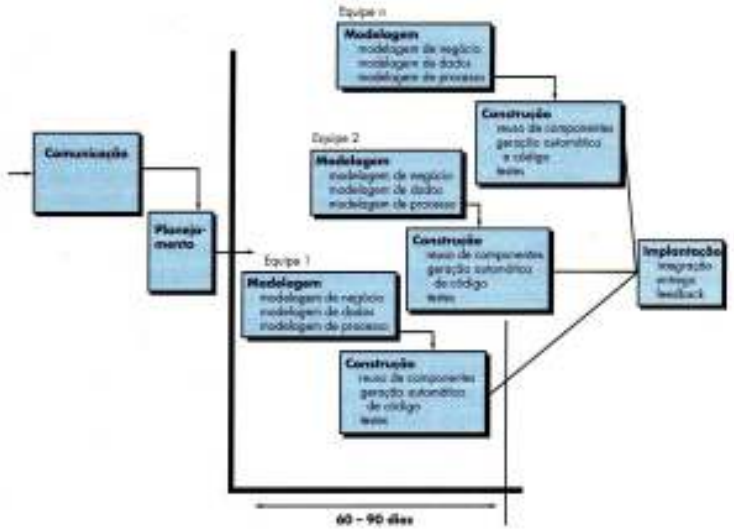
A implantação estabelece a base das iterações subseqüentes se necessárias.

Se uma aplicação comercial pode ser modularizada de modo a permitir que cada função principal possa ser completada em menos de três meses é uma candidata ao RAD. Cada função principal pode ser tratada por uma equipe RAD distinta e, depois, integrada para formar um todo.

Como todos os modelos de processos, a abordagem RAD tem desvantagens:

1. Para projetos grandes, mas passíveis de sofrer aumento, o RAD exige recursos humanos suficientes para criar um número adequado de equipes RAD.
2. Se desenvolvedores e clientes não estiverem comprometidos com as atividades continuamente rápidas, para completar o sistema em curtíssimo espaço de tempo, o projeto RAD falhará.
3. Se o sistema não puder ser adequadamente modularizado, as construções dos componentes necessários ao RAD serão problemáticas.
4. Se for necessário um alto desempenho e esse desempenho tiver sido conseguido ajustando as interfaces dos componentes do sistema, a abordagem RAD pode não funcionar.
5. O RAD pode não ser adequado quando os riscos técnicos são altos (por exemplo, quando uma nova aplicação faz bastante uso de tecnologia nova).

Figura: modelo de processo RAD



Modelos Evolucionários de Processo de Software

O software, como todo sistema complexo, evolui com o passar do tempo. Os requisitos do negócio e do produto também mudam freqüentemente à medida que o desenvolvimento prossegue, dificultando um caminho direto para o produto final; prazos reduzidos de mercado tornam impossível completar um produto de software abrangente, mas uma versão reduzida pode ser elaborada para fazer face à competitividade ou às pressões do negócio; um conjunto de requisitos básicos de um produto ou sistema é bem entendido, mas os detalhes das extensões do produto ou sistema ainda precisam ser definidos. Nesse caso, e em situações semelhantes, os engenheiros de software precisam de um modelo de processo que tenha sido explicitamente projetado para acomodar um produto que evolui com o tempo.

Os modelos evolucionários são iterativos. Eles são caracterizados de forma a permitir aos engenheiros de softwares desenvolverem versões cada vez mais completas do software.

Modelo Prototipagem

O cliente, freqüentemente, define um conjunto de objetivos gerais para o software, mas não identifica detalhadamente os requisitos de entrada, processamento ou saída.

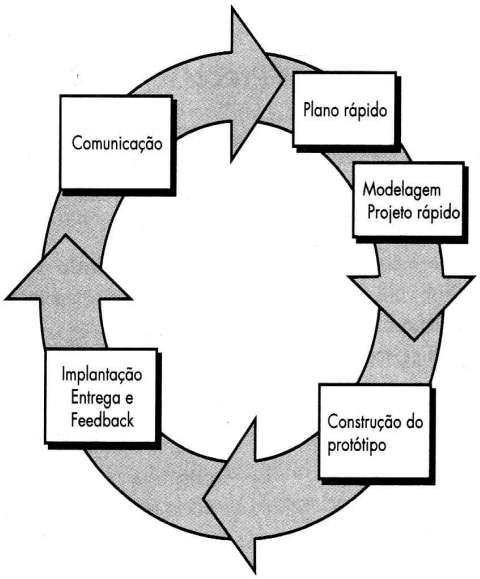
Em outros casos, o desenvolvedor pode estar inseguro da eficiência de um algoritmo, da adaptabilidade de um sistema operacional ou da forma que a interação homem/máquina de assumir. Nessas e em muitas outras situações, uma paradigma de prototipagem pode oferecer a melhor abordagem.

Apesar de a prototipagem poder ser usada como um modelo de processo independente ela é mais comumente usada como uma técnica que pode ser implementada dentro do contexto de qualquer modelo. Independentemente da maneira como é aplicado, o paradigma da prototipagem auxilia o engenheiro de software e o cliente e entenderem melhor o que deve ser construído quando os requisitos estão confusos

A prototipagem começa com a comunicação. O engenheiro de software e o cliente encontram-se e definem os objetivos gerais do software, identificam as necessidades conhecidas e delineiam áreas que definem de mais definições.

O projeto rápido concentra-se na representação daqueles aspectos dos softwares que estarão visíveis para o cliente e leva à construção de um protótipo que é implantado e depois avaliado pelo cliente.

> *"Na maioria dos projetos, o primeiro sistema construído consegue ser apenas utilizável. Pode ser muito lento, muito grande, muito complicado de usar, ou tudo isso ao mesmo tempo. Não há outra alternativa senão começar de novo e construir uma versão reprojetada, na qual esses problemas são resolvidos. Quando um novo conceito de sistema ou nova tecnologia é usada deve-se construir um sistema para ser descartado, porque nem mesmo o melhor planejamento é tão onisciente para fazer certo da primeira vez. A questão da gerência, entretanto, não é se o sistema piloto elaborado deve ser descartado. Ele será."*



O modelo de prototipagem

O protótipo pode servir como “o primeiro sistema”, aquele que se recomenda ser descartado. Mas essa pode ser uma visão idealizada. É verdade que tanto clientes quanto desenvolvedores gostam da prototipagem. Os usuários têm o sabor de um sistema real e os desenvolvedores conseguem construir algo imediatamente. Ainda assim, a prototipagem pode ser problemática pelas seguintes razões:

- O cliente vê o que parece ser uma versão executável do software, ignorando que o protótipo apenas consiga funcionar precariamente, sem saber que, na pressa de fazê-lo rodar, ninguém considerou sua qualidade global ou manutenção no longo prazo.
- Quando informado de que o produto deve ser refeito para que altos níveis de qualidade possam ser atingidos, o cliente reclama e exige “alguns acertos” para transformar o protótipo num produto executável.
- O desenvolvedor freqüentemente faz concessões na implementação a fim de conseguir rapidamente um protótipo executável. Um sistema operacional, ou uma linguagem de programação inapropriada, pode ser usado simplesmente por estar disponível e ser conhecido; um algoritmo ineficiente pode ser implementado simplesmente para demonstrar capacidade. Passado certo tempo, o desenvolvedor pode ficar familiarizado com essas escolhas e esquecer-se de todas as razões por que elas eram inadequadas. A escolha muito aquém da ideal tornou-se agora parte integral do sistema.

Apesar disso, a prototipagem pode ser um paradigma efetivo para a engenharia de software. O importante é definir as regras do jogo no início; isto é, cliente e desenvolvedor devem estar de acordo que o protótipo é construído para servir como um mecanismo de definição dos requisitos. Depois ele será descartado (pelo menos em parte), e o software real será submetido à engenharia com vistas na qualidade.

Modelo Espiral

O modelo espiral é um modelo evolucionário de processo de software que combina a natureza iterativa de prototipagem com os aspectos controlados e sistemáticos do modelo cascata. Ele fornece desenvolvimento rápido de versões cada vez mais completas.

Seu criador o descreve da seguinte maneira:

> *"O modelo espiral de desenvolvimento é um gerador de modelo de processo guiado por risco usado para guiar a engenharia de sistemas intensivos em software com vários interessados concorrentes. Ele tem duas principais características distintas. A primeira é uma abordagem cíclica, para aumentar incrementalmente o grau de definição e implementação de um sistema enquanto diminui seu grau de risco. A outra é um conjunto de marcos de ancoragem, para garantir o comprometimento dos interessados com soluções exeqüíveis e mutuamente satisfatórias para o sistema."*

Em vez de representar o processo de software como seqüências de atividades com algum retorno entre uma atividade e outra, o processo é representado por uma espiral, onde cada loop da espiral representa uma fase do processo de software.

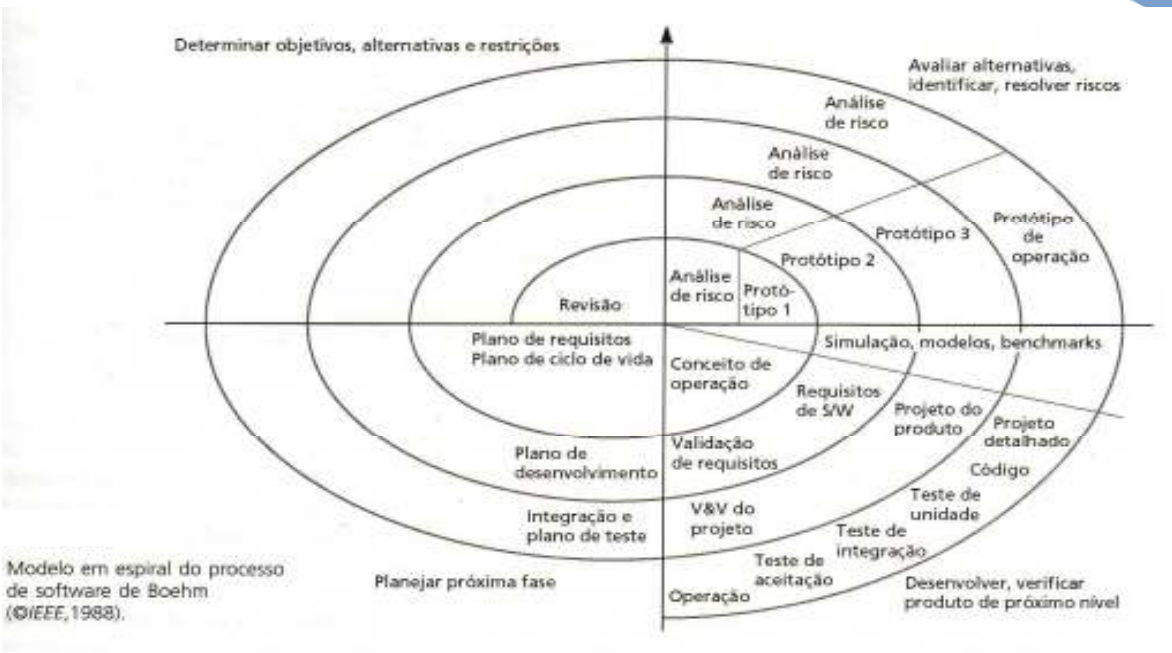
Cada loop da espiral esta dividido em quatro setores:

- *Definição dos objetivos: Os objetivos específicos dessa fase do projeto são definidos, as restrições são identificadas e um plano de gerenciamento detalhado e elaborado. Os riscos do projeto são identificados.*
- *Avaliação e redução de riscos: Para cada risco do projeto identificado uma análise detalhada é realizada e providências são tomadas para reduzir os riscos.*
- *Desenvolvimento e Validação: Após a avaliação do risco um modelo de desenvolvimento para o sistema é selecionado.*
- *Planejamento: O projeto é revisado e uma decisão é tomada para prosseguimento ao próximo loop da espiral. Se o próximo loop for começar então são elaborados planos para a próxima fase do projeto.*

Diferente dos outros modelos, o modelo espiral reconhece o risco.

O modelo espiral não termina quando o software é entregue, ele pode ser adaptado para aplicação ao longo da vida do software que foi construído.

O modelo espiral é uma abordagem realista do desenvolvimento de sistema e softwares de grande porte. Como o software evolui à medida que o processo avança, o desenvolvedor e o cliente entendem melhor e reagem aos riscos de cada nível evolucionário.



Modelo em espiral do processo de software de Boehm (©IEEE, 1988).

Esse modelo usa a prototipagem como um mecanismo de redução de riscos, porém, mais importante, permite ao desenvolvedor aplicar prototipagem em qualquer estágio da evolução do produto. Ele também mantém a abordagem sistemática passo-a-passo, sugerida pelo ciclo de vida clássico, mas o incorpora a um arcabouço iterativo que reflete mais realisticamente o mundo real.

No entanto, o modelo espiral não é uma panaceia. Pode ser difícil convencer os clientes (particularmente em situações de contrato) que a abordagem evolucionária é confortável. Ela exige competência considerável na avaliação de riscos e depende dessa competência para ter sucesso. Se um risco importante não for descoberto e gerenciado, fatalmente ocorrerão problemas.

Projeto com reuso

O processo de projeto, na maioria das disciplinas de engenharia, tem como base o reuso de componentes. Engenheiros mecânicos e elétricos não especificam um projeto em que cada componente tenha de ser fabricado especialmente para aquele projeto. Eles baseiam o projeto em componentes que já foram experimentados e testados em outros sistemas. Esses componentes não são apenas pequenos componentes, como flanges e válvulas, mas incluem subsistemas principais como motores, condensadores ou turbinas.

Atualmente, em geral, é aceito o princípio de que precisamos de uma abordagem comparável para o desenvolvimento de software. O software deve ser considerado um ativo e o reuso desses ativos é essencial para aumentar o retorno de seus custos de desenvolvimento. Os requisitos relativos a menores custos de produção e manutenção de software, à maior rapidez na entrega de sistemas e o aumento da qualidade só podem ser atendidos pelo reuso generalizado e sistemático de softwares.

O reuso de software deve ser considerado durante o projeto de engenharia ou de requisitos. O reuso oportuno é possível durante a programação, quando se descobrem componentes que por acaso atendam a um requisito. Contudo, o reuso sistemático requer um processo de projeto que considere como os projetos podem ser reutilizados e que organize explicitamente o projeto em torno de componentes de software disponíveis.

A engenharia de software baseada no reuso é uma abordagem para o desenvolvimento que tenta maximizar o reuso do software já existente. As unidades de software que são reutilizadas podem ser de tamanhos radicalmente diferentes. Por exemplo:

1. O reuso de sistemas de aplicações	Todo o sistema de aplicações pode ser reutilizado pela sua incorporação, sem mudança, em outros sistemas ou pelo desenvolvimento de famílias de aplicações, que podem ser executadas em plataformas diferentes ou ser especializadas para as necessidades de determinados clientes.
2. Reuso de componentes	Os componentes de uma aplicação, que variam em tamanho incluindo desde subsistemas até objetos isolados, podem ser reutilizados. Por exemplo, um sistema de combinação de padrões, desenvolvido como parte de um sistema de processamento de texto, pode ser reutilizado em um sistema de gerenciamento de banco de dados.
3. Reuso de funções	Os componentes de software que implementam uma única função, como uma função matemática, podem ser reutilizados. Esse tipo de reuso com base em bibliotecas de padrões tem sido comum nos últimos 40 anos.

O reuso de sistemas de aplicações tem sido amplamente praticado há muitos anos, à medique que as empresas de software implementam seus sistemas em uma série de máquinas que fazem ajustes para diferentes ambientes. Também o reuso de funções está bem estabelecido por meio de bibliotecas de funções reutilizáveis, como gráficos e bibliotecas de matemática. Contudo, embora tenha havido interesse no reuso de componentes desde o início da década de 1980, foi somente nos últimos anos que ele se tornou aceito como uma abordagem prática para o desenvolvimento de sistemas de software.

Uma vantagem óbvia do reuso de software é que os custos gerais de desenvolvimento ficam reduzidos. Menos componentes de software têm de ser especificados, projetados, implementados e validados. Contudo, a redução de custos é apenas uma vantagem em potencial do reuso, pois existe uma série de outras vantagens no reuso de ativos de software, conforme se demonstra a seguir:

Benefícios	Explicação
Maior confiabilidade	Os componentes reutilizados que são empregados nos sistemas em operação devem ser mais confiáveis do que os componentes novos. Eles já foram experimentados e testados em diferentes ambientes. Os defeitos de projeto e de implementação são descobertos e eliminados no uso inicial dos componentes, reduzindo, assim, o número de falhas quando o componente é reutilizado.
Redução dos riscos de processo	Se recorrermos a um componente já existente, serão menores as incertezas sobre os custos relacionados ao reuso desse componente do que sobre os custos de desenvolvimento. Esse é um fator importante para o gerenciamento de projetos, pois reduz as incertezas nas estimativas de custos de projeto. É particularmente verdadeiro quando reutilizamos componentes grandes, como subsistemas.
Uso efetivo de especialistas	Em vez de os especialistas em aplicações fazerem o mesmo trabalho em diferentes projetos, eles podem desenvolver componentes reutilizáveis, que englobam seu conhecimento.
Conformidade com os padrões	Alguns padrões, como os padrões de interface com o usuário, podem ser implementados como um conjunto de componentes-padrão. Por exemplo, os componentes reutilizáveis podem ser desenvolvidos para implementar menus em uma interface com o usuário. Todas as aplicações apresentam o mesmo formato de menu para os usuários. O uso de interfaces-padrão com o usuário melhora a confiabilidade, uma vez que os usuários possivelmente cometem menos enganos quando utilizam uma interface familiar.
Desenvolvimento acelerado	De modo geral, é mais importante fornecer um sistema para o mercado o mais rápido possível do que se prender aos custos gerais de desenvolvimento. O reuso de componentes acelera a produção, porque o tempo de desenvolvimento e o de validação devem ser reduzidos.

Tabela: Benefícios do reuso de software

Existem três requisitos fundamentais para o projeto e o desenvolvimento de software baseado em reuso:

1. Deve ser possível encontrar componentes reutilizáveis apropriados. As organizações necessitam de uma base de componentes reutilizáveis adequadamente catalogados e documentados. Deve ser fácil encontrar componentes nesse catálogo, se ele existir.
2. O responsável pelo reuso dos componentes precisa ter certeza de que os componentes se comportarão como especificado e de que serão confiáveis. Idealmente, todos os componentes no catálogo de uma organização devem estar certificados, a fim de confirmar que atingiram determinados padrões de qualidade. Na prática, essa situação não é realista e as pessoas em uma empresa aprendem de maneira informal sobre componentes confiáveis.
3. Os componentes devem ter a documentação associada para ajudar o usuário a compreendê-los e adaptá-los a uma nova aplicação. A documentação deve incluir informações sobre onde os componentes foram reutilizados e sobre quaisquer problemas de reuso que tenham sido encontrados.

Contudo, existem alguns custos e problemas associados com o reuso, veja tabela abaixo, que podem inibir a introdução desse método e significar que as reduções no custo total de desenvolvimento, com o reuso, serão menores do que as previstas.

Benefícios	Explicação
Aumento nos custos de manutenção	Se o código-fonte do componente não estiver disponível, então os custos de manutenção poderão aumentar, uma vez que os elementos reutilizados no sistema podem se tornar crescentemente incompatíveis com as mudanças do sistema.
Falta de ferramentas de apoio	Os conjuntos de ferramentas CASE não apóiam o desenvolvimento com reuso. Pode ser difícil ou impossível integrar essas ferramentas com um sistema de biblioteca de componentes. O processo de software assumido por essas ferramentas pode não levar em conta o reuso.
Síndrome do 'não-foi-inventado-aqui'	Alguns engenheiros de software às vezes preferem reescrever componentes, porque acreditam que podem fazer melhor que o componente reutilizável. Isso tem a ver em parte com confiança e em parte com o fato de que escrever um software original é visto como mais desafiador do que reutilizar o software de outras pessoas.
Manutenção de uma biblioteca de componentes	Implementar uma biblioteca de componentes e assegurar que os desenvolvedores de software utilizem essa biblioteca pode ser dispendioso. Nossas técnicas atuais de classificação, catalogação e recuperação de componentes de software são imaturas.
Encontrar e adaptar componentes reutilizáveis	Os componentes de software devem ser encontrados em uma biblioteca, compreendidos e, algumas vezes, adaptados, a fim de trabalharem em um novo ambiente. Os engenheiros precisam ter uma razoável certeza de poder encontrar um componente em uma biblioteca, antes de incluírem, a rotina de busca de componente como parte de seu processo normal de desenvolvimento.

Tabela: Problemas com o reuso de software

Modelos Especializados de Processos de Software

Os modelos especializados têm muitas das características de um ou mais modelos convencionais já mencionados. Entretanto, os modelos especializados tendem a ser aplicados quando uma abordagem de engenharia de software estreitamente definida é escolhida.

Desenvolvimento Baseado em Componentes

O desenvolvimento baseado em componentes, ou a engenharia de software baseada em componentes, emergiu no final de década de 1990 como uma abordagem baseada no reuso para o desenvolvimento de sistemas de software. Sua motivação foi a frustração de que o desenvolvimento orientado a objetos não tinha conduzido a um extensivo reuso, como originalmente foi sugerido. As classes de objetos individuais eram muito detalhadas e específicas e tinham de ser associadas a uma aplicação em tempo de compilação ou quando o sistema estivesse conectado. O conhecimento detalhado das classes era necessário para sua utilização, e isso, geralmente, significava que o código-fonte precisava estar disponível, apresentando problemas difíceis para a comercialização de componentes. Apesar das primeiras previsões otimistas, nenhum mercado significativo para os objetos individuais foi desenvolvido.

Os componentes de software comercial de prateleira, desenvolvidos por vendedores que os oferecem como produtos, podem ser usados quando o software precisa ser construído. Esses componentes fornecem funcionalidades-alvo com interfaces bem definidas que permitem ao componente ser integrado no software.

O modelo de desenvolvimento baseado em componentes incorpora muitas das características do modelo espiral. Evolucionário por natureza, demanda uma abordagem iterativa para criação de software.

O modelo compõe aplicações a partir de componentes de software previamente preparados. As atividades de modelagem e construção começam com a identificação dos componentes candidatos. Esses componentes podem ser projetados como módulos de software convencional ou como classes ou pacotes de classes orientados a objetos.

Independente da tecnologia usada para criar os componentes, o modelo de desenvolvimento baseado em componentes incorpora os seguintes passos:

1. Produtos baseados em componente disponíveis são pesquisados e avaliados para o domínio da aplicação em questão.
2. Tópicos de integração de componentes são considerados.
3. Uma arquitetura de software é projetada para acomodar os componentes.
4. Componentes são integrados a arquitetura.
5. Testes abrangentes são realizados para garantir a funcionalidade adequada.

O modelo de desenvolvimento baseada em componentes leva ao reuso do software, e a reusabilidade fornece aos engenheiros vários benefícios mensuráveis.

- Redução de 70% do prazo do ciclo de desenvolvimento.
- Redução de 84% do custo do projeto
- Um índice de produtividade de 26,2, comparado com o padrão da indústria que é de 16,9%

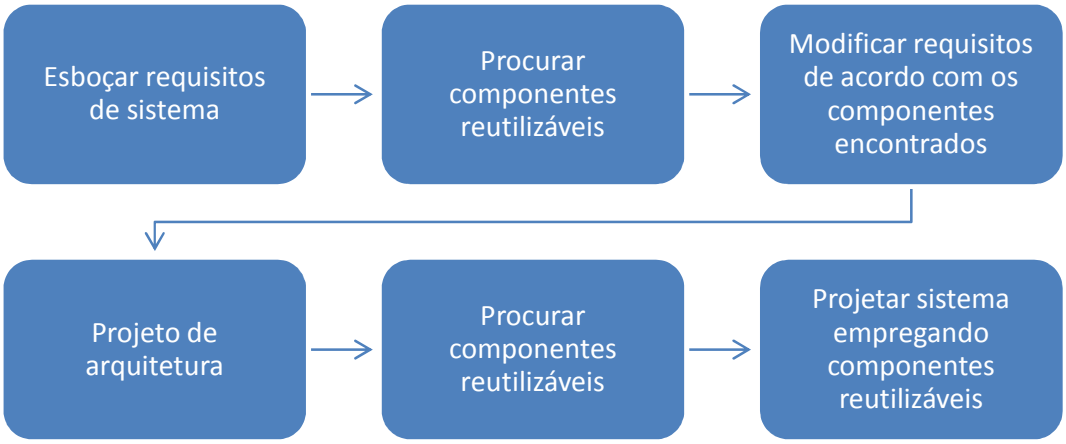


Figura. Desenvolvimento com reuso

Apesar desses resultados serem em função da robustez da biblioteca de componentes, existe pouca dúvida de que o modelo de desenvolvimento baseado em componentes fornece vantagens significativas para os engenheiros de software.

No desenvolvimento orientado a reuso, os requisitos de sistema são modificados de acordo com os componentes reutilizáveis disponíveis. O projeto também se baseia nos componentes existentes. Naturalmente, isso significa que é possível que haja uma conciliação dos requisitos. O projeto pode ser menos eficiente do que um projeto de propósito especial. Contudo, os menores custos de desenvolvimento, a entrega mais rápida do sistema e o aumento da confiabilidade do sistema devem compensar esse aspecto.

O Modelo de Métodos Formais

Em disciplinas tradicionais de engenharia, com a engenharia civil e a elétrica, o progresso geralmente envolveu o desenvolvimento de melhores técnicas matemáticas. A indústria de engenharia não teve nenhuma dificuldade em aceitar a necessidade da análise matemática e em incorporá-la a seus processos. A análise matemática é uma parte rotineira do processo de desenvolvimento e validação de um projeto de produto.

Contudo, a engenharia de software não seguiu o mesmo caminho. Embora já tenham passado mais de 25 anos de pesquisa sobre o uso de técnicas matemáticas no processo de software, essas técnicas tiveram um impacto limitado. Os denominados 'métodos formais' de desenvolvimento de software não são amplamente utilizados no desenvolvimento de software industrial. A maioria das empresas de desenvolvimento de software não considera que seja eficaz, em termos de custo, aplicá-los em seus processos de software.

O modelo de métodos formais abrange um conjunto de atividades que levam à especificação matemática formal do software de computador. Os métodos formais permitem ao engenheiro de software especificar, desenvolver e verificar um sistema baseado em computador pela aplicação rigorosa de uma notação matemática.

Quando métodos formais são usados durante o desenvolvimento, eles fornecem um mecanismo para eliminação de muitos problemas que são difíceis de resolver usando outros paradigmas de engenharia de software.

Quando usados durante o projeto, métodos formais servem de base para a verificação do programa e, assim, permite que o engenheiro descubra e corrija erros que poderiam passar despercebidos.

Apesar de não vir a ser uma abordagem de uso geral, o modelo de métodos formais oferece a promessa de softwares livres de defeitos. Todavia, foram formuladas as seguintes preocupações sobre a sua aplicabilidade em um ambiente comercial:

- O desenvolvimento de modelos formais é atualmente muito lento e dispendioso.
- Como poucos desenvolvedores de software têm o preparo necessário para aplicar métodos formais, torna-se necessário um treinamento extensivo.
- É difícil usar os modelos como um mecanismo de comunicação, com clientes despreparados tecnicamente.

Apesar dessas preocupações, a abordagem de métodos formais tem ganhado adeptos entre os desenvolvedores de softwares que precisam construir softwares críticos em termos de segurança (por exemplo, desenvolvedores de aviônica de aeronaves e de dispositivos médicos) e entre os desenvolvedores de software que sofreriam pesadas sanções econômicas se ocorressem erros nos softwares.

Dentre os sistemas críticos, nos quais métodos formais foram aplicados com sucesso, destacam-se o sistema de informação de controle de tráfego aéreo (Hall, 1996), os sistemas de sinalização de estrada de ferro (Dehbonei e Mejia, 1995), os sistemas de naves espaciais (Easterbrook, 1998) e os sistemas de controle médico (Jacky, 1995, 1997).

No Reino Unido, o uso de métodos formais é obrigatório, conforme definido pelo Ministério de Defesa, para sistemas em que a segurança é fundamental (MOD, 1995).

Desenvolvimento de Software Orientado a Aspectos (DSOA)

O desenvolvimento de software orientado por aspectos é uma técnica nova cujo objetivo é permitir a definição separada de requisitos transversais às classes de um sistema orientado por objetos. Por atravessarem todo o código, tais requisitos são, em geral, de difícil modularização em linguagens orientadas por objetos puras. Com a orientação por aspectos, requisitos transversais, tais como geração de registros de operações, controle sincronização e comunicação, podem ser implementados de maneira elegante, eficiente e modular, aumentando o nível de reutilização de código em sistemas.

A programação orientada por projetos (AOP), proposta por Gregor Kiczales em 1997, tem por objetivo modularizar decisões de projetos que não podem ser adequadamente definidas por meio da programação orientada por objetos (POO). Isto se deve ao fato de que alguns requisitos violam a modularização natural do restante da implementação.

A AOP foi desenvolvida a partir da constatação de que certos requisitos não se encaixam em um único módulo de programa, ou pelo menos em um conjunto de módulos altamente relacionados. Em sistemas orientados por objetos, a unidade de modularização é a classe, e os requisitos transversais se espalham por múltiplas classes. Esses requisitos são também denominados *aspectos*.

Independentemente do processo de software escolhido, os construtores de softwares complexos invariavelmente implementam um conjunto de características, funções e conteúdo de informações localizadas. Essas características localizadas de software são modeladas como componentes e depois construídas dentro do contexto de uma arquitetura de sistemas. Na medida em que os sistemas modernos baseados em computador tornam-se mais sofisticados e complexos, certas preocupações – propriedades solicitadas pelo cliente ou áreas de preocupação técnica – cobrem toda a arquitetura. Algumas preocupações são propriedades de alto nível do sistema (por exemplo, segurança e tolerância a falhas). Outras preocupações afetam funções (por exemplo, aplicação de regras de negócio), enquanto outras são sistêmicas (por exemplo, sincronização de tarefas ou gestão de memória).

Quando as preocupações diversas de um sistema entrecortam várias funções, características e informações do sistema, elas são freqüentemente referidas como preocupações transversais. Requisitos referentes a aspectos definem essas preocupações transversais, que têm impacto em todas as características do software. Como já dito anteriormente, a programação orientada a aspectos é um paradigma relativamente novo da engenharia de software que fornece um processo e abordagem metodológica para definir, especificar, projetar e construir aspectos – "*mecanismos que transcendem sub-rotinas e herança para localizar a expressão de uma preocupação transversal*".

O desenvolvimento de software orientado por aspectos é realizado em três fases: a decomposição, a implementação e a recomposição de requisitos.

Processos Unificados

De certo modo, o Processo Unificado é uma tentativa de apoiar-se nos melhores recursos e características dos modelos convencionais de processo de software, mas caracterizá-los de um modo que implemente muitos dos melhores princípios de desenvolvimento ágil de softwares. O processo unificado reconhece a importância da comunicação com o cliente e dos métodos diretos para descrever a visão do cliente de um sistema (isto é, caso de uso). Ele enfatiza importante papel da arquitetura de software e "ajuda o arquiteto a se concentrar nas metas corretas, tais como compreensibilidade, abertura a modificações futuras e reuso". Ele sugere um fluxo de processo que é iterativo e incremental, dando a sensação evolucionária que é essencial no desenvolvimento moderno de software.

Um breve histórico

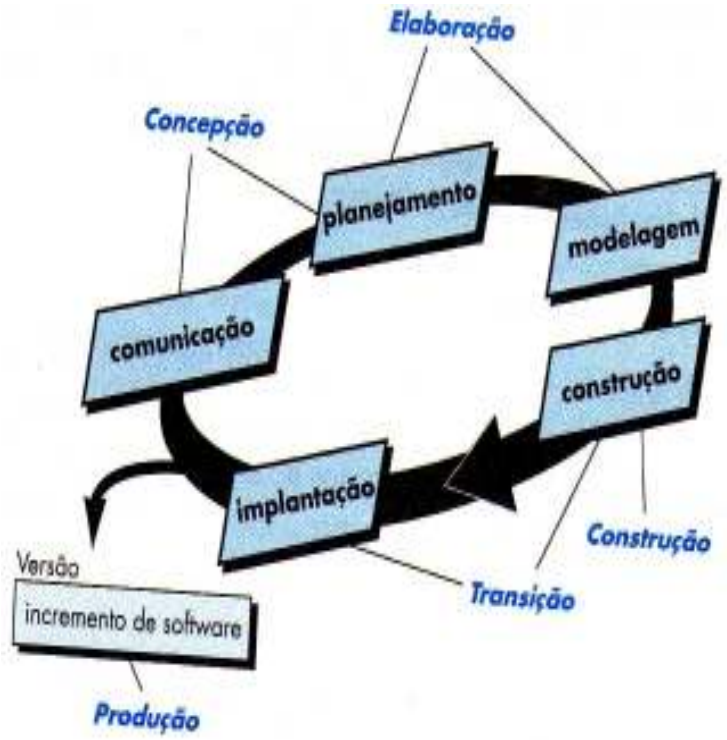
Durante a década de 1980 e início da década de 1990, métodos e linguagens de programação orientada a objetos (OO) ganharam uma audiência espalhada por toda a comunidade de engenharia de software. Uma grande variedade de métodos de análise orientada a objetos (AOO) e projeto orientado a objetos (POO) foram propostos durante o mesmo período de tempo, e um modelo de processo orientado a objetos de propósito geral foi introduzido. Como a maioria dos "novos" paradigmas de engenharia de software, os adeptos de cada um dos métodos de AOO e POO argumentaram sobre qual era o melhor, mas nenhum método ou linguagem individual dominou a paisagem da engenharia de software.

No início da década de 1990, James Rumbaugh, Grady Booch e Ivar Jacobson começaram a trabalhar em um "método unificado" que combinaria as melhores características de cada um dos seus métodos individuais e adotaria características adicionais propostas por outros especialistas no ramo da OO. O resultado foi a UML – uma *linguagem unificada de modelagem* que contém uma notação robusta para a modelagem e desenvolvimento de softwares orientados a objeto. Ao mesmo tempo, a Rational Corporation e outros vendedores desenvolveram ferramentas automatizadas para apoiar métodos da UML.

A UML fornece a tecnologia necessária para apoiar a prática de engenharia de software orientada a objetos, mas não fornece o arcabouço de processo para guiar as equipes de projeto na aplicação da tecnologia. Ao longo dos cinco anos seguintes, Jacobson, Rumbaugh e Booch desenvolveram o *Processo Unificado*, um arcabouço para a engenharia de software orientada a objetos usando a UML. Hoje em dia, o Processo Unificado e a UML são amplamente usados em projetos OO de todas as naturezas. O modelo iterativo e incremental proposto pelo PU pode, e deve, ser adaptado para satisfazer às necessidades específicas de projeto.

Uma variedade de produtos de trabalho (por exemplo, modelos e documentos) pode ser produzida como conseqüência da aplicação da UML. No entanto, eles são freqüentemente mutilados por engenheiros de software para tornar o desenvolvimento mais ágil e mais suscetível a modificações.

Fases do Processo Unificado



Relembrando:

- O processo unificado reconhece a importância da comunicação com o cliente e dos métodos diretos para escrever a visão do cliente de um sistema (o caso de uso).
- Ele enfatiza importantes papéis da arquitetura de software e ajuda o arquiteto a se concentrar nas metas corretas.
- Ele sugere um fluxo de processo de software que é iterativo e incremental, dando a sensação evolucionária, que é essencial no desenvolvimento moderno de software.

O processo unificado de software possui 5 fases:

- Fase da Concepção: Abrange atividades de comunicação com os clientes e de planejamento. Em colaboração com os clientes e usuários finais, os requisitos de negócio para o software são identificados, um rascunho de arquitetura é proposto. Os requisitos de negócios são escritos por meio de casos de uso preliminares. Um caso de uso descreve uma seqüência de ações que são realizadas por um ator.
- Fase de elaboração: Refina e expande os casos de uso preliminares que foram desenvolvidos como parte da fase de concepção e expande a representação arquitetural para incluir cinco visões diferentes do software – o modelo de caso de uso, o modelo de analise, o modelo de projeto, o modelo de implementação e o modelo de implantação;
- A fase da construção: É idêntica as atividades de construção definidas pelos modelos genéricos, essa fase desenvolve ou adquire componentes de software que vão tornar cada caso de uso operacional para os usuários finais. Todas as características e funções necessárias e requeridas do

incremento de software serão implementadas no código fonte. À medida que os componentes são implementados, testes unitários são realizados.

- Fase de Transição: Abrange os últimos estágios da vida genérica de construção e a primeira parte genérica da atividade de implantação. O software é dado ao usuário final para testes beta e relatórios de feedback do usuário sobre defeitos e modificações necessárias.
- Fase de Produção: Coincidem com as atividades do processo genérico de implantação, durante essa fase o uso do software é monitorado, é fornecido suporte para o ambiente de operação e os relatórios de defeitos e solicitações de modificações são submetidos e avaliados.

Rational Unified Process (RUP)

O RUP é um exemplo de modelo de processo moderno que foi derivado do trabalho sobre a UML e do Processo Unificado de desenvolvimento de software (UP).

O RUP reconhece que os modelos convencionais de processo apresentam uma visão única de processo. Por outro lado, o RUP e descrito a partir de três perspectivas:

1. Uma perspectiva dinâmica, que mostra fases do modelo ao longo do tempo.
2. Uma perspectiva estática, que mostra as atividades realizadas no processo.
3. Uma perspectiva prática que sugere as boas práticas a serem usadas durante o processo.

O RUP é um modelo constituído por fases que identifica quatro fases discretas no processo de software. As fases do RUP estão relacionadas mais estritamente aos negócios do que aos assuntos técnicos. São elas:

- Concepção: O objetivo dessa fase de concepção é estabelecer um business case para o sistema. Você deve identificar todas as entidades externas (pessoas e sistemas) que irão interagir com o sistema, e definir essas interações. Depois essas informações são usadas para avaliar a contribuição do sistema com o negócio.
- Elaboração: Os objetivos dessa fase é desenvolver um entendimento do domínio do problema, estabelecer um framework de arquitetura para o sistema, desenvolver o plano de projeto e identificar os riscos principais do projeto. (UML)
- Construção: A fase de construção esta essencialmente relacionada ao projeto, programação e teste de sistema. As partes do sistema são desenvolvidas paralelamente e integradas durante essa fase.
- Transição: A fase final do RUP esta relacionada à transferência do sistema da comunidade de desenvolvimento para a comunidade dos usuários e com a entrada do sistema em funcionamento em ambiente real.

Cada fase do RUP pode ser realizada de forma iterativa, com os resultados desenvolvidos incrementalmente, além disso, o conjunto total de fases pode ser realizada de forma incremental.

A visão estática do RUP enfoca as atividades que ocorrem durante o processo de desenvolvimento. Elas são chamadas de workflow, dos quais existem seis.

- Modelagem de Negócio;

- Requisitos;
- Análise de Projeto;
- Implementação;
- Teste;
- Implantação;
- Gerenciamento de Configuração;
- Gerenciamento de Projetos;
- Ambiente.

Iconix

Iconix é um processo de desenvolvimento de software-Modelo de Engenharia de Software desenvolvido pela Iconix Software Engineering. Trata-se de uma metodologia prática e simples, mas também poderosa e com um componente de análise e representação de problemas sólidos e eficazes.

É um processo não tão burocrático quanto o RUP, ou seja, não gera tanta documentação, mas também não pode ser considerado tão simples como o XP.

O Iconix é um processo que está adaptado ao padrão UML, possuindo uma característica exclusiva chamada Rastreabilidade dos Requisitos, que através dos seus mecanismos permite checar em todas as fases se os requisitos estão sendo atendidos.

Os principais diagramas usados no processo do Iconix, são:

- Modelo de Domínio: é uma parte essencial do Iconix, ele constrói uma porção estática inicial de um modelo que é essencial para dirigir a fase de design a partir dos casos de uso. Basicamente o modelo de domínio consiste em descobrir objetos de um problema do mundo real. Para realizar o modelo de domínio é preciso tentar descobrir o maior número possível de classes existentes no problema para o qual se pretende desenvolver o software.

- Modelo de Caso de Uso: Esse modelo é usado para representar as exigências dos usuários, seja para um sistema novo, ou partindo de um já existente. Ele deve detalhar de forma clara e legível todos os cenários que os usuários executarão para realizar alguma tarefa.

- Diagrama de Robustez: Essa fase tem como objetivo conectar a fase de analise com a parte de projeto assegurando que a descrição dos casos de uso está correta, além de descobrir novos objetos através do fluxo de ação.
- Diagrama de Seqüência: Tem como objetivo construir um modelo dinâmico entre o usuário e o sistema. Para isso é necessário usar os objetos e suas iterações identificados na análise robusta, porém detalhando todo o fluxo de ação.

- Diagrama de Classe: Nada mais é do que o modelo de domínio que foi atualizado ao longo de todas as fazes do processo e representa todas as funcionalidades do sistema de modo estático sem as iterações com o usuário.

O Iconix é dividido em dois grandes setores, modelo estático e modelo dinâmico, esses podem ser desenvolvidos paralelamente e de forma recursiva, o modelo estático é formado pelo diagrama de domínio e de classe e o dinâmico pelos demais.

O Iconix trabalha a partir de um protótipo de Interfaces onde se desenvolvem os diagramas de caso de uso baseados nos requisitos levantados. A partir do diagrama do caso de uso, se faz analise robusta para cada caso de uso. Com os resultados obtidos é possível desenvolver o diagrama de seqüência e posteriormente complementar o domínio com novos métodos e atributos descobertos.

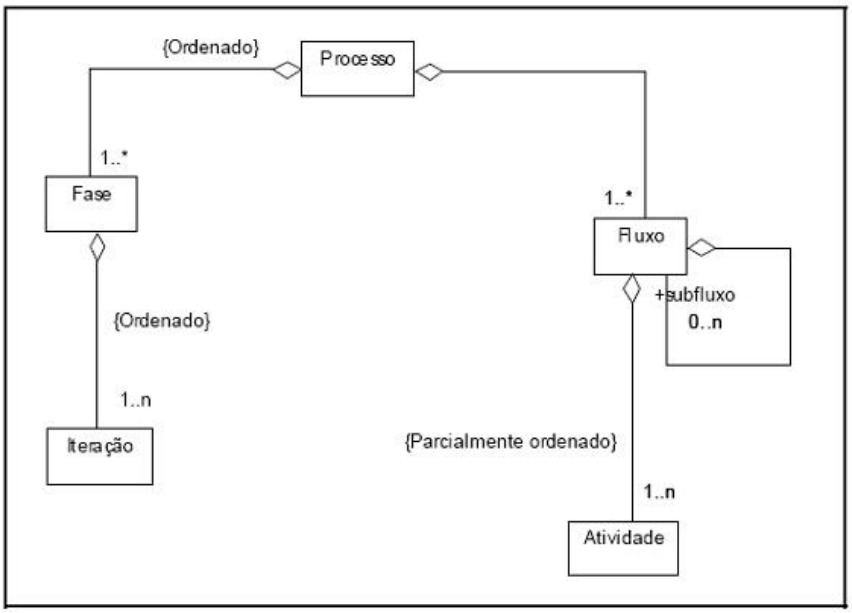
Praxis

O Praxis é um processo de software, baseado no Processo Unificado (UP), desenhado para dar suporte a projetos didáticos. A sigla Praxis significa Processo para Aplicativos e Extensíveis Interativos, refletindo em uma ênfase no desenvolvimento de aplicativos gráficos interativos, baseados na tecnologia orientada a objeto.

O Praxis propõe um ciclo de vida composto por fases que produzem um conjunto precisamente definido de artefatos (documentos e modelos). Para construir cada um dos artefatos, o usuário de processo (estudante ou Engenheiro) precisa exercitar um conjunto de praticas recomendáveis. Na construção desses artefatos, o usuário do processo é guiado por padrões e auxiliado pelos modelos de documentos e exemplos constantes de material de apoio.

O Praxis adota as mesmas fases e os mesmos fluxos do Processo Unificado. Os principais elementos que constituem o praxis são:

- Passo: Divisão formal de um processo, com pré-requisitos, entradas, critérios de aprovação e resultados definidos.
- Fase: Divisão maior de um processo, para fins gerenciais, que corresponde aos pontos principais de aceitação por parte do cliente.
- Interação: Passo constituinte de uma fase, no qual se atinge um conjunto bem definido de metas parciais de um projeto.
- Script: Conjunto de instruções que define como uma iteração deve ser executada.
- Fluxo: Sub-processo caracterizado por um tema Técnico ou Gerencial.
- Sub-fluxo: Conjunto de atividades mais estreitamente correlatas faz parte de um fluxo maior.
- Atividade: Passo constituinte de um fluxo
- Técnica: Método ou prática aplicável à execução de um conjunto de atividades.



Cleanroom

O desenvolvimento de software Cleanroom é uma filosofia de desenvolvimento de software que usa métodos formais para apoiar inspeção rigorosa de software.

O objetivo dessa abordagem para o desenvolvimento de software é o software com defeito zero. O nome Cleanroom foi derivado por analogia com unidades de fabricação de semicondutores, em que os defeitos são evitados na manufatura em uma atmosfera ultralimpa.

A abordagem Cleanroom para desenvolvimento de software baseia-se em cinco estratégias principais:

1. Especificação formal – O software a ser desenvolvido é especificado formalmente. Um modelo de estado e transição que mostra resposta dos sistemas por estímulos é usado para especificar.
2. Desenvolvimento Incremental – O software é particionado em incrementos desenvolvidos e validados separadamente, por meio de processo Cleanroom.
3. Programação Estruturada – Somente um número limitado de construções abstratas de controle de dados são usados. Um número limitado de construções é usado e o objetivo é transformar sistematicamente a especificação para criar o código de programa.
4. Verificação estática – O software desenvolvido é verificado estaticamente por meio de inspeções rigorosas de software.
5. Testes estáticos do sistema – O incremento de software integrado é testado estaticamente.

Há três equipes de software envolvidas quando o processo Cleanroom é usado para desenvolvimento de sistema de grande porte.

- A equipe de Especificação – Esse grupo é responsável pelo desenvolvimento e manutenção das especificações de sistema.
- A equipe de desenvolvimento – É a equipe responsável pelo desenvolvimento e pela verificação do software.
- A equipe de certificação – É responsável pelo desenvolvimento de um conjunto de testes estatísticos para exercitar o software depois de desenvolvido. Os testes baseiam-se em especificações formal.

O uso da abordagem Cleanroom tem conduzido geralmente a um software com poucos ou nenhum erro.

A abordagem para desenvolvimento incremental no processo Cleanroom é entregar a funcionalidade crítica do cliente em incrementos mais cedo. As funções de sistema menos importantes são incluídas em incrementos mais tarde.

A inspeção rigorosa de programa é parte fundamental do processo Cleanroom. A abordagem baseia-se em transformações bem definidas que tentam preservar a correção, a cada transformação, para uma representação mais detalhada. A cada estágio a nova representação é inspecionada e argumentos matematicamente rigorosos são desenvolvidos e demonstrando que a saída da transformação é consistente com a entrada.

O desenvolvimento do Cleanroom funciona quando é praticado por engenheiros habilidosos e comprometidos.

Desenvolvimento Ágil

Em 2001, Kent Beck e 16 outros notáveis desenvolvedores, produtores e consultores de software – conhecidos como a "Aliança Ágil" - assinaram o "Manifesto para o Desenvolvimento Ágil de Software".

Estamos descobrindo melhores modos de desenvolvimento de software fazendo-o e ajudando outros a fazê-lo. Por meio desse trabalho passamos a valorizar:

Indivíduos e iteracões	Em vez de	Processos e ferramentas
Softwares funcionando	Em vez de	Documentação abrangente
Colaboração do cliente	Em vez de	Negociação de contratos
Resposta a modificacões	Em vez de	Seguir um plano

Isto é, ainda que haja valor nos itens à direita, valorizamos mais os itens à esquerda.

Um manifesto é normalmente associado com um movimento político emergente – um movimento que ataque a velha guarda e sugira mudanças revolucionárias (espera-se que seja para melhor). De algum modo, isso é exatamente o que o desenvolvimento ágil é.

Embora as idéias subjacentes que guiam o desenvolvimento ágil tenham estado conosco há muitos anos, somente durante a década de 1990 é que essas idéias foram cristalizadas em um "movimento". Em essência, os métodos ágeis foram desenvolvidos em um esforço para vencer as fraquezas percebidas e reais da engenharia de software convencional. O desenvolvimento ágil pode fornecer importantes benefícios, mas ele não é aplicável a todos os projetos, pessoas e situações. Ele também não é contrário à sólida prática de engenharia de software e pode ser aplicado como uma filosofia prevalecente a todo trabalho de software.

Na economia moderna, é freqüentemente difícil ou impossível prever como um sistema baseado em computador (por exemplo, uma aplicação com base Web) evoluirá com o passar do tempo. Condições de mercado mudam rapidamente, necessidades dos usuários finais evoluem e novas ameaças de competição emergem sem alerta. Em muitas situações, não podemos mais definir completamente os requisitos antes do início do projeto. Os engenheiros de software devem ser ágeis o suficiente para responder a um ambiente de negócios mutante.

Isso significa que um reconhecimento dessas causas realísticas modernas nos obrigam a descartar princípios, conceitos, métodos e ferramentas valiosos de engenharia de software? Certamente não! Como todas as disciplinas de engenharia, a engenharia de software continua a evoluir. Ele pode ser adaptado facilmente para encarar os desafios colocados pela demanda por agilidade.

Em um livro intrigante sobre o desenvolvimento ágil de softwares, Alistair Cockburn argumenta que os modelos prescritivos de processos têm uma deficiência importante: *eles esquecem as fragilidades das pessoas que constroem softwares de computador.* Engenheiros de software não são robôs. Eles exibem grande variedade de estilos de trabalho e diferenças significativas em nível de habilidade, criatividade, regularidade, consistência e espontaneidade. Alguns se comunicam bem na forma escrita, outros não. Cockburn argumenta que os modelos de processo podem "tratar as fraquezas comuns das pessoas com [ou] disciplina ou tolerância", e que a maioria dos modelos prescritivos de processo escolhe a disciplina. Ele afirma: "Como a consistência em ação é uma fraqueza humana, metodologias de alta disciplina são frágeis".

Se os modelos de processos têm de funcionar, eles precisam fornecer um mecanismo realístico de encorajar a disciplina necessária, ou precisam ser caracterizados de um modo que mostre "tolerância" com as pessoas que fazem o trabalho de engenharia de software. Invariavelmente, práticas tolerantes são mais fáceis de serem adotadas e sustentadas pelo pessoal de software, mas (como Cockburn admite) elas podem ser menos produtivas. Como a maioria das coisas na vida, negociações devem ser consideradas.

12 Princípios para aqueles que querem alcançar agilidade

1. Nossa maior prioridade é satisfazer ao cliente desde o início por meio de entrega contínua de software valioso.
2. Modificações de requisitos são bem-vindas, mesmo que tardias no desenvolvimento. Os processos ágeis aproveitam as modificações como vantagens para a competitividade do cliente.
3. Entrega de softwares funcionando freqüentemente, a cada duas semanas até dois meses, de preferência no menor espaço de tempo.
4. O pessoal de negócio e desenvolvedores devem trabalhar juntos diariamente durante todo o projeto.
5. Construção de projetos em torno de indivíduos motivados. Forneça-lhes o ambiente e apoio que precisam e confie que eles farão o trabalho.
6. O método mais eficiente e efetivo de levar informação para e dentro de uma equipe de desenvolvimento é a conversa face a face.
7. Software funcionando é a principal medida de progresso.
8. Processos ágeis promovem desenvolvimento sustentável. Os patrocinadores, desenvolvedores e usuários devem ser capazes de manter o ritmo constante, indefinidamente.
9. Atenção contínua à excelência técnica e ao bom projeto facilitam a agilidade.
10. Simplicidade – a arte de maximizar a quantidade de trabalho não efetuado – é essencial.
11. As melhores arquiteturas, requisitos e projetos surgem de equipes auto-organizadas.
12. Em intervalos regulares, a equipe reflete sobre como se tornar mais efetiva, então sintoniza e ajusta adequadamente o seu comportamento.

Processo Ágil

A metodologia de desenvolvimento que vimos até os dias de hoje são métodos bem complexos usados geralmente para desenvolvimento de softwares grandes e aplicados para empresas geralmente grandes com equipes de desenvolvimento bastante volumosas e que nem sempre ficavam localizados no mesmo lugar, por essa causa os métodos eram complexos e envolviam grandes documentações.

Porém quando essas mesmas metodologias foram aplicadas em sistemas pequenos e empresas que não possuíam um grande volume de desenvolvedores, percebeu-se que o tempo perdido com as especificações e com a documentação era muito dispendioso e o tempo gasto com o desenvolvimento era muito menor, pensando nisso foi que surgiu o que chamamos de métodos ágeis.

Os métodos ágeis contam com uma abordagem interativa para especificação, desenvolvimento e entrega de software, e foram criados principalmente para apoiar o desenvolvimento de aplicações de negócios nas quais os requisitos de sistema mudam rapidamente durante o processo de desenvolvimento. Eles destinam-se a entregar um software de trabalho rapidamente aos clientes, que podem então propor novos requisitos e alterações a serem incluídos nas iterações posteriores do sistema.

Os princípios dos métodos ágeis são:

1. Envolvimento do cliente: Clientes devem ser profundamente envolvidos no processo de desenvolvimento. Seu papel é fornecer e priorizar novos requisitos do sistema e avaliar as iterações do sistema.
2. Entrega Incremental: O software é desenvolvido em incrementos e o cliente especifica os requisitos a serem incluídos em cada incremento.
3. Pessoas, não processos: As habilidades da equipe de desenvolvimento devem ser reconhecidas e exploradas. Os membros da equipe devem desenvolver suas próprias maneiras de trabalhar sem processos prescritos.
4. Aceite as mudanças: Tenha em mente que os requisitos do sistema vão mudar, por isso projete o sistema para acomodar essas mudanças.
5. Mantenha a Simplicidade: Concentre-se na simplicidade do software que esta sendo desenvolvido e do processo de desenvolvimento. Sempre que possível, trabalhe ativamente para eliminar a complexidade do sistema.

Extreme Programming (XP)

O extreme programming é talvez o mais conhecido e mais amplamente usado método ágil.

Na extreme programming, todos os requisitos são expressos como cenários, que são implementados diretamente como uma série de tarefas. Os programadores trabalham em pares e desenvolvem testes para cada tarefa antes da escrita do código. Todos os testes devem ser executados com sucesso quando o código é integrado ao sistema.

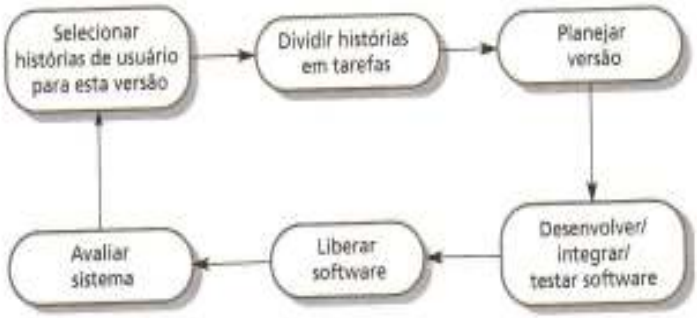
O extreme programming envolve práticas que se enquadram nos princípios dos métodos ágeis.

Em um processo XP, os clientes estão intimamente envolvidos na especificação e priorização dos requisitos do sistema, o cliente é parte da equipe de desenvolvimento e discute cenários com os outros membros da equipe. Junto eles desenvolvem um “cartão de histórias” que engloba as necessidades do cliente.

Após o desenvolvimento dos cartões de histórias, a equipe de desenvolvimento os dividirá em tarefas e estimará o esforço e os recursos necessários para a implementação. O cliente então prioriza as histórias para implementação, escolhendo as que podem ser usadas imediatamente para proporcionar apoio útil ao negócio.

A extreme programming exige uma abordagem “extrema” para o desenvolvimento iterativo. Novas versões de software podem ser compiladas varias vezes por dia e os incrementos são entregues para os clientes aproximadamente a cada duas semanas.

A extreme programming defende que os softwares devem passar por *refactoring* constantemente. Isso significa que a equipe de programação procura por possíveis melhorias no software, implementando-as imediatamente, portanto o software deve ser sempre fácil de compreender e alterar quando novas histórias são implementadas.



SCRUM

O *Scrum* (o nome é derivado de uma atividade que ocorre durante um jogo de *rugby)* é um modelo de processo que foi desenvolvido por Jeff Sutherland e por sua equipe no início da década de 1990. Nos últimos anos foi realizado desenvolvimento adicional de métodos Scrum para SCwaber e Beedle. Os princípios do Scrum são consistentes com o manifesto ágil:

- Pequenas equipes de trabalho são organizadas de modo a "maximizar" a construção, minimizar a supervisão e maximizar o compartilhamento de conhecimento informal.
- O processo precisa ser adaptável tanto a modificações técnicas quanto de negócios para garantir que o melhor produto possível seja produzido.
- O processo produz freqüentes incrementos de software que podem ser inspecionados, ajustados, testados, documentados e expandidos.
- O trabalho de desenvolvimento e o pessoal que o realiza é dividido em partições claras de baixo acoplamento ou em pacotes.
- Testes e documentações constantes são realizados à medida que o produto é produzido.
- O processo SCRUM fornece a habilidade de declarar o produto como pronto sempre que necessário.

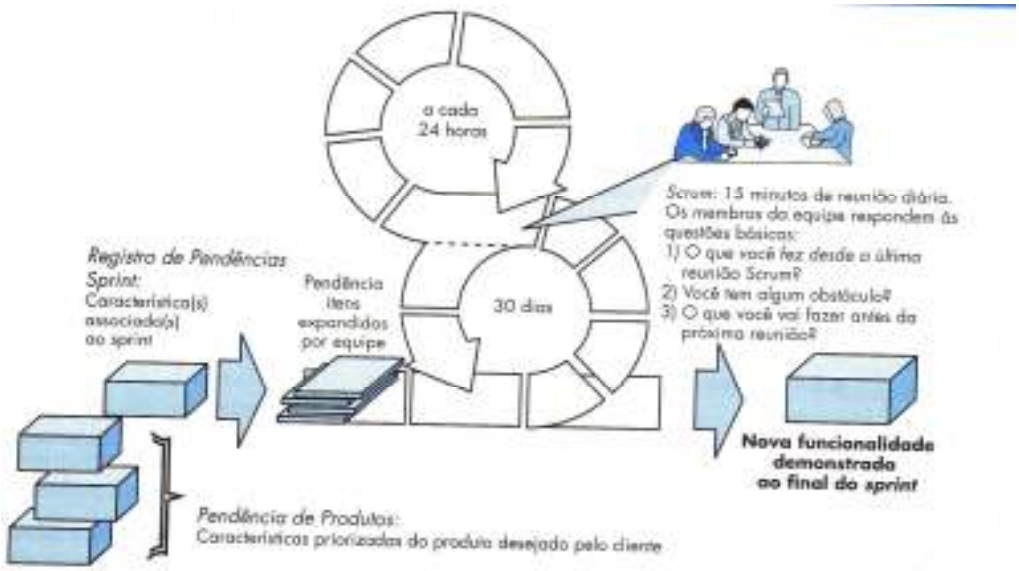
Os princípios do Scrum são usados para guiar as atividades de desenvolvimento dentro de um processo que incorpora as seguintes atividades: requisitos, análise, projeto, evolução e entrega. Em cada atividade as tarefas de trabalho ocorrem dentro de um padrão de processos chamados de sprint. O trabalho conduzido dentro de um sprint é adaptado ao problema em mãos e é definido e freqüentemente, modificado em tempo real pela equipe do SCRUM.

No Scrum temos:

- Pendências: Uma lista priorizada de requisitos ou características do projeto que fornecem valor de negócio para o cliente.
- Sprint: Consiste de unidades de trabalho que são necessárias para satisfazer a um requisito definido na pendência que precisa ser cumprido num intervalo de tempo predefinido. Durante o sprint, os itens em pendências a que as unidades de trabalho do sprint se destinam são congelados
- Reuniões Scrum: São reuniões curtas feitas diariamente pela equipe Scrum. Três questões chaves são formuladas e respondidas por todos os membros.
  - O que você fez desde a última reunião?
  - Que obstáculo você esta encontrando?
  - O que você planeja realizar até a próxima reunião da equipe?

Um líder de equipe chamado Scrum máster que lidera essas reuniões e avalia as respostas de cada membro.

Demos: Entregas incremental de software ao cliente de modo que a funcionalidade implementada possa ser demonstrada e avaliada pelo cliente.

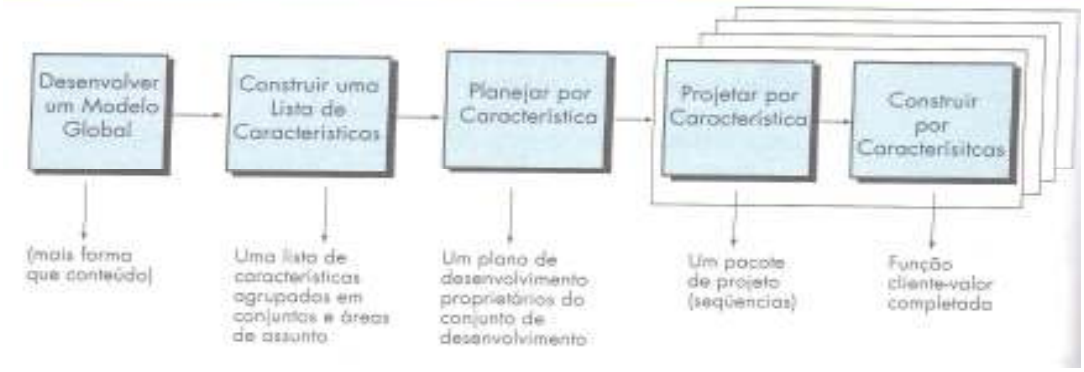


FDD (Feature Driven Development – Desenvolvimento Guiado por Características)

O FDD surgiu a principio como um modelo prático de processos para engenharia de softwares orientado a objetos, depois esse modelo foi melhorado e adaptado para ser aplicado a projetos de softwares de tamanho moderado e grande.

No contexto do FDD, uma característica é uma função valorizada pelo cliente que pode ser implementada em duas semanas ou menos. A ênfase na definição de característica fornece os seguintes benefícios.

- Como as características são pequenos blocos de funcionalidades passíveis de entrega, os usuários podem descrevê-las mais facilmente, entender como elas se relacionam uma com a outra mais prontamente e revisá-las melhor.
- As características podem ser organizadas em agrupamento hierárquico relacionados ao negocio.
- Como uma característica é um incremento de software passível de entrega do FDD, a equipe desenvolve características operacionais a cada duas semanas.
- Como características são pequenas, suas representações de projeto e de código são mais fáceis de serem inspecionadas.
- Planejamento de projeto, conogramação e monitoração são guiados pela hierarquia de característica em vez de por um conjunto de tarefas de engenharia de software.



ASD (Adaptative Software Development - Desenvolvimento Adaptativo de Software)

O DAS ou ASD foi proposto como uma técnica para a construção de softwares complexos. O DAS se concentra na colaboração humana e na auto-organização da equipe.

Ele define um ciclo de vida que incorpora três fazes:

- Especulação: Durante a especulação o projeto é iniciado e o planejamento de um ciclo adaptativo é conduzido. Planejamento do ciclo adaptativo usa informações de iniciação do projeto, a declaração de missão é feita pelo cliente, as restrições do projeto e requisitos básicos são definidos e também é definido a quantidade de ciclos que serão realizados.
- Colaboração: Pessoal motivado trabalha junto de um modo que multiplica seus talentos e resultados criativos além de seus números absolutos. A colaboração não é fácil. Pessoas que trabalham juntas precisam confiar uma nas outras para: criticar sem animosidade, ajudar sem ressentimentos, trabalhar tão duro ou mais duro do que costumam, ter um conjunto de habilidades para contribuir e comunicar problemas e preocupações de modo efetivo.
- Aprendizado: A medida que os membros da equipe DAS começam a desenvolver os componentes que fazem parte de um ciclo adaptativo,a ênfase esta tanto no aprendizado quando no desenvolvimento do ciclo.

DSDM (Dynamic Systems Development Method – Método de Desenvolvimento Dinâmico de Sistemas)

O DSDM é uma abordagem ágil de desenvolvimento de software que fornece um arcabouço para construir e manter sistemas que satisfazem às restrições de prazos apertadas por meio do uso de prototipagem incremental em um ambiente controlado de projeto.

A abordagem DSDM a cada iteração segue a abordagem dos 80%, isto é, apenas um certo trabalho é necessário para que cada incremento facilite o avanço para o incremento seguinte. Os detalhes restantes podem ser completados depois.

O ciclo de vida do DSDM define três ciclos iterativos diferentes, precedidos por duas atividades adicionais de ciclo de vida:

- Estudo de viabilidade: Estabelece os requisitos básicos e restrições do negócio associados à aplicação em construção.
- Estudo de Negócio: Estabelece os requisitos funcionais e de informação que permitirão à aplicação fornecer valor ao negócio.
- Iteração do modelo funcional: Produz um conjunto de protótipos incrementais que demonstram a funcionalidade para o cliente. O intuito durante esse ciclo iterativo é obter requisitos adicionais.
- Iteração de projeto e construção: Revisita os protótipos construídos durante a iteração do modelo funcional para garantir que cada um tenha passado por engenharia de modo que seja capaz de fornecer valor de negócio operacional para os usuários finais.
- Implementação: Coloca o ultimo incremento de software, deve-se notar que o incremento não precisa estar 100% completo, modificações podem ser feitas enquanto o incremento é colocada em ação.

Crystal

Crystal na verdade é uma família de métodos ágeis que possui uma abordagem de desenvolvimento de software que premia a manobrabilidade, caracteriza-se como “um jogo cooperativo de invenção e comunicação de recursos limitados, com o principal objetivo de entregar softwares úteis funcionando e como objetivo secundário de preparar-se para o jogo seguinte”.

Para conseguir manobralidade, seus criadores definiram um conjunto de metodologias, cada qual com elementos centrais que são comuns a todas, e papéis, padrões de processos, produtos de trabalho e práticas específicas de cada uma. A família Crystal é, na verdade, um conjunto de processos ágeis que se mostraram efetivos para diferentes tipos de projeto. A intenção é permitir que equipes ágeis selecionem o membro da família Crystal mais apropriado para o seu projeto e ambiente.

Modelagem Ágil (AM)

A modelagem Ágil foi inspirada para a utilização em softwares muito grandes.

Apesar da AM sugerir uma ampla gama de princípios de modelagem centrais e suplementares os eu tornam AM peculiar são:

- Modelar com uma finalidade: Um desenvolvedor que usa AM deve ter uma meta especifica em mente antes de criar o modelo. Uma vez identificada a meta do modelo, o tipo de notação a ser usada e o nível de detalhes a ser exigido serão óbvios.
- Usar modelos múltiplos: Há muitos modelos e notações diferentes que podem ser usados para descrever software. Apenas um pequeno subconjunto é essencial para a maioria dos projetos. A AM sugere que, para fornecer a visão necessária, cada modelo apresente um aspecto diferente do sistema e que apenas aqueles modelos que ofereçam valor seja usados.
- Viajar leve: À medida que o trabalho de engenharia de software prossegue, conserve apenas aqueles modelos que fornecerão valor ao longo do prazo e livre-se do resto.

- O conteúdo é mais importante que a representação: Um modelo sistematicamente perfeito que leva pouco conteúdo útil não são tão valiosos com um modelo com notação defeituosa, mas que fornece conteúdo valioso.
- Conhecer os modelos e ferramentas que você usa para criá-los: Entenda os pontos fortes e fracos de cada modelo e das ferramentas que são usadas para criá-los
- Adaptar localmente: A abordagem de modelagem deve ser adaptada às necessidades da equipe ágil.

Resumo

Uma filosofia ágil para engenharia de software ressalta quatro tópicos-chave: a importância de equipes auto-organizadas que têm controle sobre o trabalho que executam; comunicação e colaboração entre os membros da equipe e entre os profissionais e seus clientes; um reconhecimento de que modificações representam uma oportunidade; e uma ênfase na entrega rápida de softwares que satisfaçam ao cliente. Os modelos ágeis de processos foram projetados para atender a cada um desses tópicos.

XP (Extreme Programming) é o processo ágil mais amplamente usado. Organizado como quatro atividades de arcabouço – planejamento, projeto, codificação e teste – o XP sugere um número de técnicas inovadoras e potentes que permitem a equipes ágeis criar freqüentemente versões de software que possuem características e funcionalidades descritas e priorizadas pelo cliente.

O DAS (Desenvolvimento Adaptativo de Software) ressalta a colaboração humana e a auto-organização da equipe. Organizado como três atividades de arcabouço – especulação, colaboração e aprendizado – o DAS usa um processo iterativo que incorpora planejamento do ciclo adaptativo, métodos relativamente rigorosos para o levantamento de requisitos e um ciclo de desenvolvimento iterativo que incorpora grupos enfocados nos clientes e revisões técnicas formais como mecanismos de *feedback* em tempo real. O DSDM (Método de Desenvolvimento Dinâmico de Sistemas) define três diferentes ciclos iterativos – iteração do modelo funcional, iteração de projeto e construção e implementação – precedidos por duas atividades de ciclo de vida adicionais – estudo de viabilidade e estudo de negócio. O DSDM recomenda o uso de cronogramação a cada intervalo de tempo e sugere que, em cada incremente de software, é necessário apenas o trabalho suficiente a fim de facilitar o avanço para o incremento seguinte.

O SCRUM enfatiza o uso de um conjunto de padrões de processo de software que tem comprovada efetividade para projetos com prazos apertados, requisitos mutáveis e criticalidade do negócio. Cada padrão de processo define um conjunto de tarefas de desenvolvimento e permite à equipe SCRUM construir um processo que seja adaptado às necessidades do projeto.

O Crystal é uma família de modelos ágeis de processo que podem ser adotados para as características específicas de um projeto. Como outras abordagens ágeis, o Crystal adota uma estratégia iterativa, mas ajusta o rigor do processo de modo a acomodar projetos de diferentes tamanhos e complexidades.

O FDD (Desenvolvimento Guiado por Características) é algo mais “formal” do que os outros métodos ágeis, mas ainda mantém agilidade para concentrar a equipe de projeto no desenvolvimento das características – funções valiosas para o cliente que podem ser implementadas em duas semanas ou menos. O FDD fornece maior ênfase em gestão de projeto e qualidade do que outras abordagens ágeis. A Modelagem Ágil (AM) sugere que a modelagem é essencial para todos os sistemas, mas que a complexidade, tipo e tamanho do modelo devem estar sincronizados como o software a ser construído. Por meio da proposição de um conjunto de princípios de modelagem centrais e suplementares, a AM fornece um guia útil para os profissionais durante as tarefas de análise de projeto.

Princípios Centrais, Comunicação e Planejamento

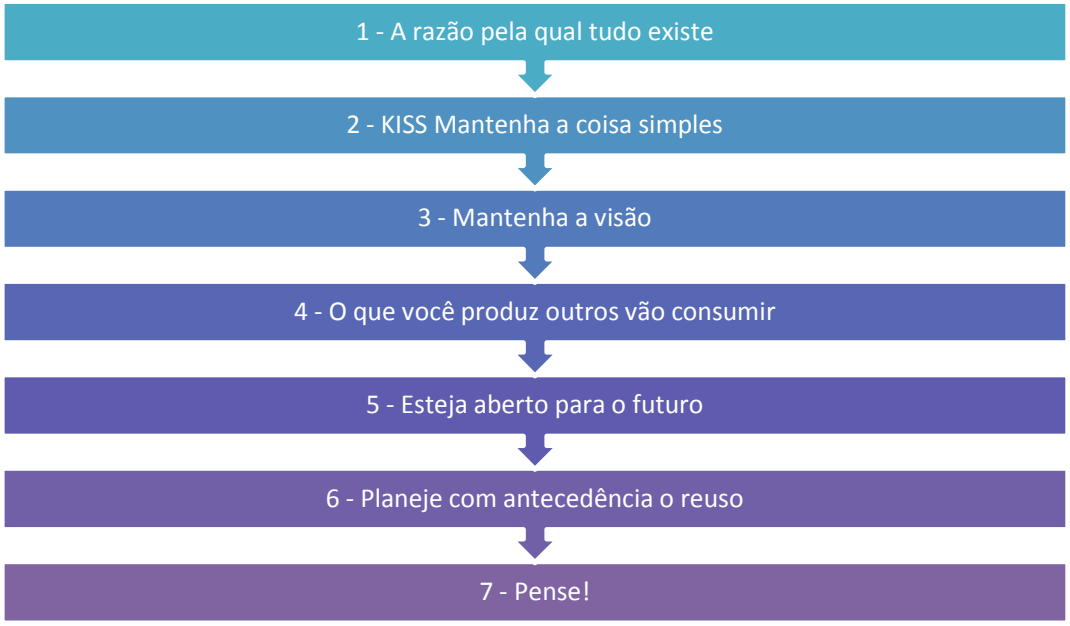
De modo genérico, a prática é uma coleção de conceitos, princípios, métodos e ferramentas da qual um engenheiro de software faz uso diariamente. A prática preenche um modelo de processo de software com as receitas técnicas e gerenciais necessárias para fazer o serviço.

A essência da prática da engenharia de software baseou-se na essência da resolução de problemas que é:

1. Entenda o problema (comunicação e análise)
2. Planeje uma solução (modelagem e projeto de software)
3. Execute o plano (geração do código)
4. Examine o resultado quanto à precisão (teste e garantia da qualidade).

Princípios Centrais

Existem sete princípios centrais para a prática da engenharia de software:



O Primeiro Princípio: A razão pela qual tudo existe

Um sistema de software existe por uma razão: para fornecer valor aos seus usuários. Todas as decisões devem ser tomadas com isso em mente. A pergunta deve ser feita: Isso adiciona valor real ao sistema? Se a resposta for não, não faça. Todos os outros princípios se apóiam nesse.

O Segundo Princípio: KISS(Keep it Simple,Stupid – Mantenha a coisa simples)

Todo projeto deve ser tão simples quanto possível. Isso facilita ter um sistema mais fácil de entender e de manter, mas não quer dizer que características internas, devam ser descartadas em nome da simplicidade. Simples também não significa rápido e sujo.

O Terceiro Princípio: Mantenha a Visão

Uma visão clara é essencial para o sucesso de um projeto de software. Ter um arquiteto fortalecido que possa manter a visão e exige que ela seja respeitada ajuda a garantir um projeto de software muito bem sucedido.

O Quarto Princípio: O que você produz outros vão consumir

De um modo ou de outro alguém mais vai usar, manter, documentar ou precisará entender o seu sistema. Assim, sempre especifique, projete e implemente sabendo que mais alguém terá de entender o eu você está fazendo.

O Quinto Princípio: Esteja Aberto para o Futuro

Os sistemas de software com verdadeira "força industrial" precisam durar muito mais. Para fazer isso com sucesso, eles precisam estar prontos para se adaptar a essas e outras modificações. Sistemas que fazem isso com sucesso são aqueles que foram projetados dessa forma desde o início.

O Sexto Princípio: Planeje com Antecedência o Reuso

Reuso poupa tempo e esforço. O reuso de código e de projetos tem sido proclamado como um importante benefício do uso de tecnologia orientada a objetos.

Planejar o reuso com antecedência reduz custo e aumenta o valor tanto dos componentes reusáveis quanto do sistema ao qual será incorporado.

O Sétimo Princípio: Pense!

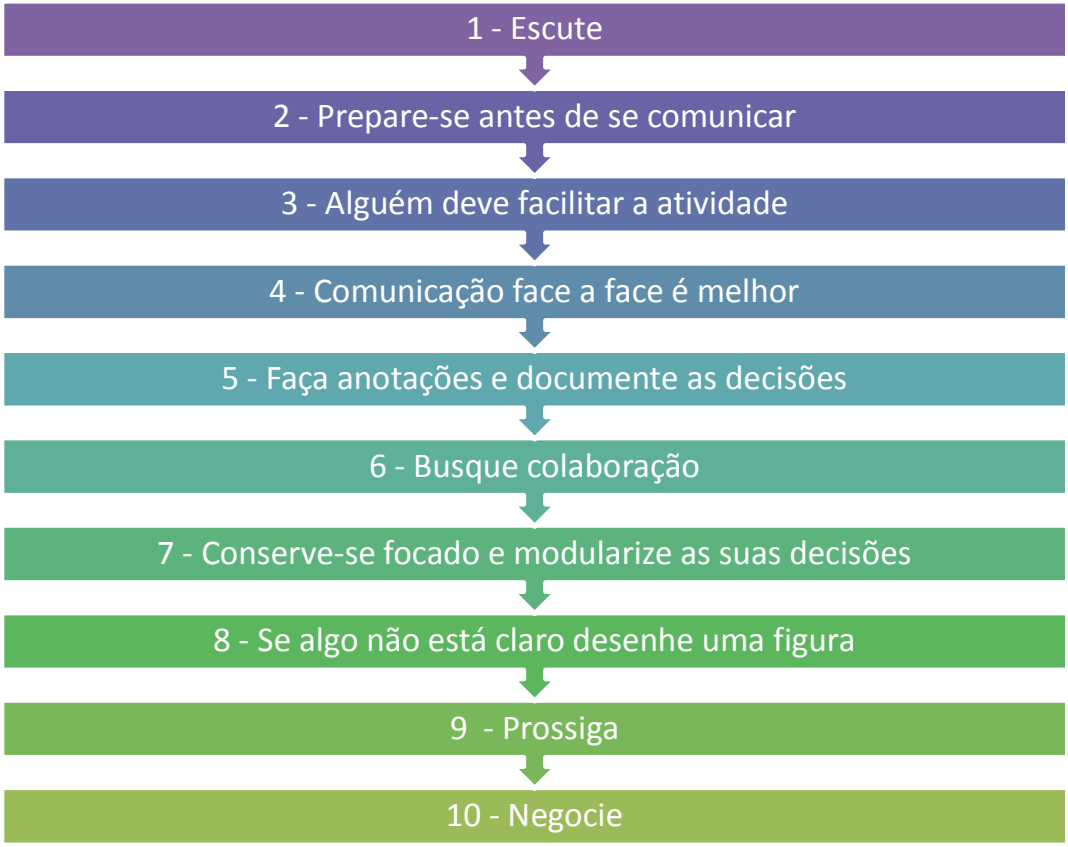
Raciocinar clara e completamente antes da ação quase sempre produz melhores resultados. Quando você pensa sobre algo é mais provável que o faça corretamente. Você também adquire conhecimento sobre como fazê-lo novamente.

Prática da Comunicação

Antes que os requisitos do cliente possam ser analisados, modelados ou especificados, eles precisam ser coletados por meio de uma atividade de comunicação. O caminho da comunicação para o entendimento é freqüentemente cheio de buracos.

A comunicação efetiva está entre as atividades mais desafiadoras com as quais se confronta um engenheiro de software.

Os princípios da comunicação são:



Princípio 1: Escute

Tente se concentrar nas palavras do interlocutor ao invés de pensar na formulação de sua resposta a essas palavras. Peça esclarecimento se algo estiver obscuro. Evite interrupções constantes.

Princípio 2: Prepare-se antes de se comunicar

Gaste tempo em entender o problema antes de se encontrar com os outros, pesquise a respeito para entender a "linguagem" do usuário.

Princípio 3: Alguém deve facilitar a atividade

Toda reunião de comunicação deve ter um líder(facilitador) para manter a conversa se movendo em uma direção produtiva, para mediar qualquer conflito e para garantir que os outros princípios sejam seguidos.

Princípio 4: Comunicação face a face é melhor

Costuma funcionar melhor quando alguma outra representação da comunicação relevante é apresentada.

Princípio 5: Faça anotações e documente as decisões

As coisas têm um modo de escorrer por entre os dedos. Alguém que participe da comunicação deve servir como um registrador e anotar todos os pontos importantes.

Princípio 6: Busque Colaboração

Colaboração e consenso ocorrem quando o conhecimento coletivo dos membros da equipe é combinado para descrever funções e características do sistema.

Principio 7: Conserve-se focado, modularize sua discussão

Quanto mais pessoas estiverem envolvidas em uma comunicação, mais provavelmente aquela discussão vai ficar saltando de um tópico para o outro. O facilitador deve manter a conversa modular, abandonando um tópico apenas depois que estiver resolvido.

Principio 8: Se algo não esta claro desenhe uma figura

A comunicação verbal vai só até um certo ponto, um esboço ou um desenho pode freqüentemente fornecer esclarecimento.

Principio 9: Prossiga

A comunicação, como qualquer atividade de engenharia leva tempo. Em vez de iterar sem fim, as pessoas que participam devem reconhecer que muito tópicos requerem discussão e devem prosseguir com a reunião sem se prender a eles nesse momento.

Principio 10: Negociação

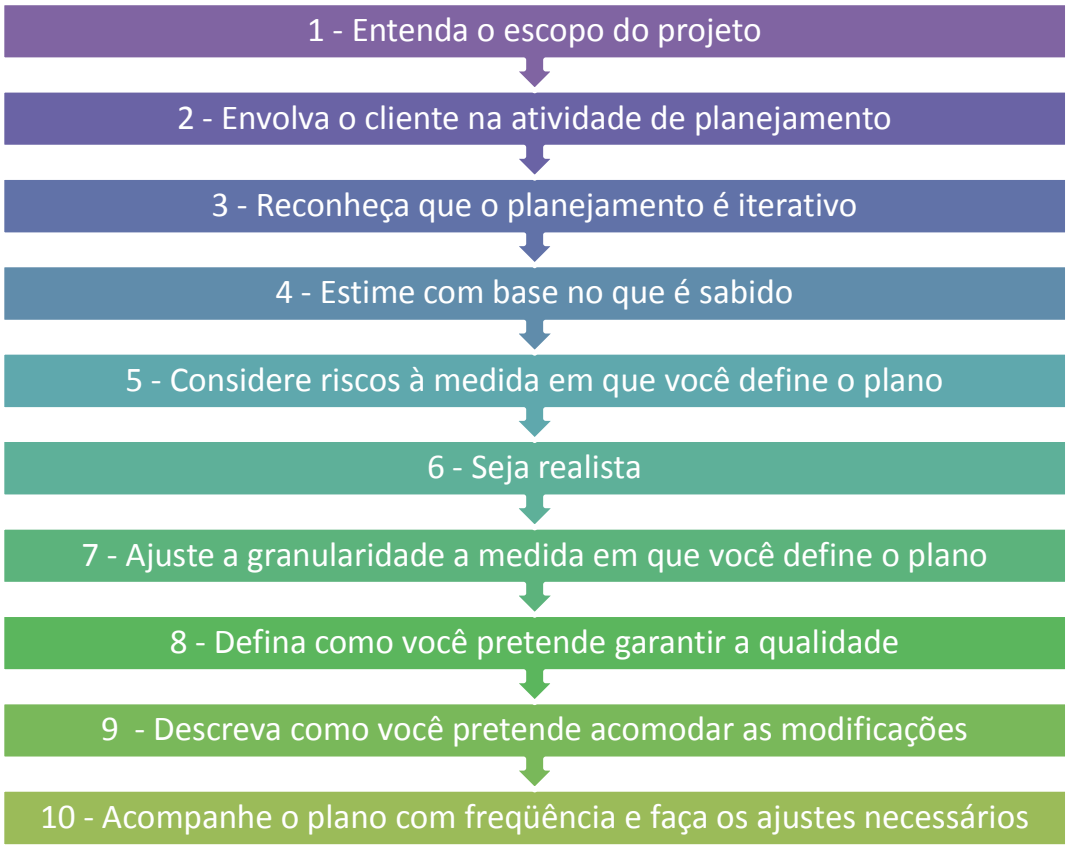
Existem muitas instâncias em que os engenheiros de software e os clientes devem negociar funções e características, isso deve ser feito de modo que ambas as partes saiam ganhando, assim sendo negociação exige comprometimento de ambas as partes.

Práticas do Planejamento

A atividade de planejamento inclui um conjunto de práticas gerenciais e técnicas que permitem a equipe de software definir um roteiro enquanto ela se move em direção a sua meta estratégica e seus objetivos táticos.

Em muitos projetos o planejamento excessivo consome tempo e não produz frutos, mas a falta de planejamento é uma receita para o caos. O planejamento deve ser conduzido com moderação.

Independente do rigor com o qual o planejamento é conduzido os princípios a seguir se aplicam:



Principio 1: Entenda o escopo do Projeto

É impossível usar um roteiro se você não souber para onde está indo. O escopo fornece a equipe de software um destino

Principio 2: Envolva o cliente na atividade de planejamento

O cliente que define as prioridades e oferece as restrições do projeto, por isso ele deve ser envolvido no planejamento.

Principio 3: Reconheça que o planejamento é iterativo

Um plano de projeto nunca é gravado em pedra. Quando o trabalho tem início, é muito provável que as coisas se modifiquem, como conseqüência o plano deve ser ajustado para acomodar as modificações.

Principio 4: Estime com base no que é sabido

A intenção de uma estimativa é fornecer uma indicação de esforço, o custo, a duração de tarefas com base no entendimento atual da equipe quanto ao trabalho a ser feito.

Principio 5: Considere riscos à medida que você define o plano

Se a equipe tiver definido riscos que têm grande impacto e alta probabilidade, é necessário planejamento e contingência.

Principio 6: Seja realista

As pessoas não trabalham 100% do tempo do dia, sempre há ruídos em qualquer comunicação humana, omissões, ambigüidade.

Principio 7: Ajuste a granularidade a medida que você define o plano

Granularidade refere-se ao nível de detalhes introduzido à medida que um plano de projeto é desenvolvido.

Principio 8: Defina como você pretende garantir a qualidade

O plano deve identificar como a equipe de software pode garantir a qualidade.

Principio 9: Descreva como você pretende acomodar as modificações

Mesmo o melhor planejamento pode ser comprometido por modificações descontroladas. A equipe de software deve identificar como as modificações devem ser acomodadas a medida que o trabalho prossegue.

Principio 10: Acompanhe o plano com freqüência e faça os ajustes necessários

Projetos de software se atrasam um dia de cada vez. Assim faz sentido acompanhar o progresso diariamente, procurando áreas problemáticas e situações em que o trabalho não esta de acordo com o programado.

Modelagem, Construção e Implantação

Prática da Modelagem

No trabalho de engenharia de software, duas classes de modelos são criadas: modelos de análise e modelos de projeto. Os modelos de analise representam os requisitos do cliente mostrando o software em três domínios diferentes: o domínio de informação, o domínio funcional e o domínio comportamental. Os modelos de projeto representam características de software que ajudam os profissionais a construí-lo efetivamente: a arquitetura, a interface com o usuário e detalhes em nível de componentes.

Princípios da Modelagem de Análise

Princípio 1: O domínio da informação de um problema precisa ser representado e entendido

O domínio de informação abrange os dados que fluem para dentro do sistema, os dados que fluem para fora do sistema e os depósitos de dados que coletam e organizam os objetos de dados persistentes.

Principio 2: As funções a serem desenvolvidas pelo software devem ser definidas.

As funções do software fornecem benefícios diretos aos usuários finais. Algumas funções transformam os dados que fluem para dentro do sistema. Em outros casos, as funções efetuam algum nível de controle sobre o processamento interno do software. As funções podem ser descritas em vários níveis de abstração diferentes que vão desde uma declaração geral até uma descrição detalhada.

Princípio 3: O comportamento do software precisa ser representado

O comportamento do software do computador é guiado por suas iterações com o ambiente externo. Entradas fornecidas pelos usuários finais, dados de controle fornecidos por um sistema externo ou monitoramento de dados coletados em uma rede, todos fazem com que o software se comporte de um modo específico.

Princípio 4: Os modelos que mostram informação, função e comportamento devem ser particionados de um modo que revele detalhes em forma de camadas.

A modelagem de analise é o primeiro passo na solução de um problema de engenharia de software. Ela permite que os profissionais entendam melhor o problema e estabelece uma base para a solução. Um problema complexo, e grande é dividido em subproblemas. Esse é o conceito chamado de particionamento e é uma estratégia chave na modelagem de análise.

Princípio 5: A tarefa de analise deve ir da informação essencial até os detalhes de implementação.

A modelagem de análise começa descrevendo o problema na perspectiva do usuário final. A "essência" do problema é descrita sem qualquer consideração de como uma solução será implementada. Os detalhes de implementação indicam como a essência será implementada.

Princípios de modelagem de Projeto

O modelo de projeto é equivalente às plantas de engenharia de uma casa. Ele começa com a representação da totalidade do objeto a ser construído e lentamente refina o objeto.

Princípio 1: O projeto deve estar relacionado ao modelo de analise.

O modelo de projeto traduz essa informação em uma arquitetura: um conjunto de subsistemas que implementam as funções principais e um conjunto de projetos em nível de componentes que são a realização das classes de analise.

Principio 2: Sempre considere a arquitetura do sistema a ser construído.

A arquitetura de software é o esqueleto do sistema a ser construído. Ela afeta as interface, estruturas de dados, fluxo de controle e comportamento do programa. Por todas essas razões o projeto deve começar com considerações arquiteturais.

Principio 3: O projeto dos dados é tão importante quanto o projeto de funções de processamento.

Os projetos de dados é um elemento essencial no projeto arquitetural. Um projeto de dados bem estruturado ajuda a simplificar o fluxo do programa e torna a implementação mais fácil.

Principio 4: As interfaces precisam ser projetadas com cuidado.

A maneira como os dados fluem entre os componentes de um sistema tem muito a ver com a eficiência do processamento, a programação de erros e a simplicidade do projeto. Uma interface bem projetada torna a implementação mais fácil.

Princípio 5: O projeto de interface do usuário deve estar sintonizado com as necessidades dos usuários finais.

A interface do usuário é a manifestação visível do software. Não importa quão sofisticada sejam suas funções interna quão abrangente suas estruturas de dados, quão bem projetada sua arquitetura, um projeto de interface pobre conduz, muitas vezes, à percepção de que o software é ruim.

Principio 6: O projeto em nível de componentes deve ser funcionalmente independente.

A independência funcional é uma medida da objetividade de um componente de software.

Principio 7: Os componentes devem ser fracamente acoplados uns aos outros e ao ambiente externo.

O acoplamento é conseguido de muitos modos – Via interface de componentes, por mensagem, por meio de dados globais. À medida que o nível de acoplamento aumenta a probabilidade de programação com erros também aumenta.

Principio 8: Representação de projeto deve ser facilmente compreendida.

O objetivo do projeto é comunicar a informação para os profissionais que vão gerar o código, para aqueles que vão testar o código, e para outros que podem vir a manter o software no futuro, portanto a mesma deve ser de fácil entendimento para todos.

Princípio 9: O projeto deve ser desenvolvido iterativamente. A cada iteração o projetista deve lutar por maior simplicidade.

Como quase todas as atividades criativas, o projeto ocorre iterativamente. As primeiras iterações trabalham para refinar o projeto e corrigir erros, mas as últimas iterações devem procurar tornar o projeto o mais simples possível.

Práticas da Construção

A prática da construção compreende um conjunto de tarefas de codificação e teste que levam ao software operacional que está pronto para ser entregue ao cliente ou ao usuário final.

Os princípios e conceitos que dirigem a tarefa de codificação são estilo de programação, linguagem de programação e métodos de programação rigorosamente definidos.

Princípios de Preparação. *Antes de escrever uma linha de código certifique-se de:*

1. Entender o problema que está tentando resolver.
2. Entender os princípios e conceitos básicos do projeto.
3. Escolher uma linguagem de programação que satisfaça as necessidades do software a ser construído e do ambiente que vai operar.
4. Selecionar um ambiente de programação que fornece ferramentas para facilitar o seu trabalho.
5. Criar um conjunto de testes unitários que será aplicado tão logo componente que você está codificando for completado.

Princípios de codificação: *Quando começar a escrever o código certifique-se de:*

1. Restringindo seus algoritmos seguindo a pratica de programação estruturada.
2. Selecionar estruturas de dados que atendam as necessidades do projeto.
3. Entender a arquitetura do software e criar interfaces que sejam consistentes com ela.
4. Conservar a lógica condicional tão simples quanto possível.
5. Criar ciclos aninhados de modo que sejam facilmente testáveis.
6. Selecionar nomes significativos de variáveis e seguir outras normas locais de codificação.
7. Escrever códigos que é auto-documentado.
8. Criar uma disposição visual que auxilie o entendimento.

Princípios de validação: *Depois de completar seu primeiro passo de codificação, certifique-se de:*

1. Conduzir uma inspeção de código quando adequado
2. Realizar testes unitários e descobrir os erros encontrados
3. Refabricar o código quando necessário.

Praticas da Implantação

A implantação não ocorre uma única vez, mas várias vezes, à medida que o software caminha para ficar completo.

A entrega de um incremento de software apresenta um marco importante para qualquer projeto de software. Alguns princípios chave devem ser seguidos à medida que a equipe se prepara para entregar um incremento.

Princípio 1: As expectativas do cliente quanto ao software devem ser geridas

Muito freqüentemente, o cliente espera mais do que a equipe prometeu entregar e o desapontamento é imediato, isso resulta em *feedback* não produtivo e arruína o moral da equipe. Um engenheiro de software deve ser cuidadoso com o envio de mensagens conflitantes ao cliente.

Principio 2: Um pacote completo de entrega deve ser montado e testado.

Um CD ou outra mídia contendo todo o software executável, arquivos de dados de suporte, documentos de suporte e ouras informações relevantes deve ser montado e rigorosamente testado por testes beta com usuários reais.

Princípio 3: Um regime de suporte deve ser estabelecido antes do software ser entregue.

Um usuário final espera receptividade e informação segura quando uma questão ou um problema surge. Se o suporte é ad hoc ou, pior, inexistente, o cliente ficará insatisfeito imediatamente.

Principio 4: Materiais institucionais adequados devem ser fornecidos aos usuários finais.

A equipe de software entrega mais do que um software em si. Ajuda de treinamento adequado deve ser desenvolvida, diretrizes de depuração deve ser fornecida e uma descrição de “o que é diferente nesse incremento de software” deve ser publicada.

Principio 5: Software defeituoso deve ser corrigido primeiro e depois entregue

Pressionados pelo tempo, algumas organizações de software entregam incrementos de baixa qualidade com um aviso ao cliente que defeitos serão consertados na versão seguinte. Isso é um erro.

Referências

PRESSMAN, R. S. *Engenharia de software*. 6. ed. São Paulo: McGraw-Hill, 2006.

SOMMERVILLE, I. *Engenharia de software*. 6. ed. São Paulo: Pearson, 2004.