Ciclos de Vida do Software

Neste contexto, neste artigo apresentaremos alguns modelos de ciclo de vida, quais sejam: Cascata, Modelo em V, Incremental, Evolutivo, RAD, Prototipagem, Espiral, Modelo de Ciclo de Vida Associado ao RUP.

Por que eu devo ler este artigo:

Neste contexto, neste artigo apresentaremos alguns modelos de ciclo de vida, quais sejam: Cascata, Modelo em V, Incremental, Evolutivo, RAD, Prototipagem, Espiral, Modelo de Ciclo de Vida Associado ao RUP.

O ciclo de vida é a estrutura contendo processos, atividades e tarefas envolvidas no desenvolvimento, operação e manutenção de um produto de software, abrangendo a vida do sistema, desde a definição de seus requisitos até o término de seu uso.

O modelo de ciclo de vida é a primeira escolha a ser feita no processo de software. A partir desta escolha definir-se-á desde a maneira mais adequada de obter as necessidades do cliente, até quando e como o cliente receberá sua primeira versão operacional do sistema.

Processo de software é o conjunto de atividades que constituem o desenvolvimento de um sistema computacional. Estas atividades são agrupadas em fases, como: definição de requisitos, análise, projeto, desenvolvimento, teste e implantação.

Em cada fase são definidas, além das suas atividades, as funções e responsabilidades de cada membro da equipe, e como produto resultante, os artefatos.

O que diferencia um processo de software do outro é a ordem em que as fases vão ocorrer, o tempo e a ênfase dados a cada fase, as atividades presentes, e os produtos entregues.

Com o crescimento do mercado de software, houve uma tendência a repetirem-se os passos e as práticas que deram certo. A etapa seguinte foi a formalização em modelos de ciclo de vida.

Em outras palavras, os **modelos de ciclo de vida são o esqueleto**, ou as estruturas pré-definidas nas quais encaixamos as fases do processo. De acordo com a NBR ISO/IEC 12207:1998, o ciclo de vida é a “Estrutura contendo processos, atividades e tarefas envolvidas no desenvolvimento, operação e manutenção de um produto de software, abrangendo a vida do sistema, desde a definição de seus requisitos até o término de seu uso.”

O **modelo de ciclo de vida** é a primeira escolha a ser feita no processo de software. A partir desta escolha definir-se-á desde a maneira mais adequada de obter as necessidades do cliente, até quando e como o cliente receberá sua primeira versão operacional do sistema.

Não existe um modelo ideal. O perfil e complexidade do negócio do cliente, o tempo disponível, o custo, a equipe, o ambiente operacional são fatores que influenciarão diretamente na **escolha do ciclo de vida de software a ser adotado**.

Da mesma forma, também é difícil uma empresa adotar um único ciclo de vida. Na maior parte dos casos, vê-se a presença de mais de um ciclo de vida no processo.

Os ciclos de vida se comportam de maneira sequencial (fases seguem determinada ordem) e/ou incremental (divisão de escopo) e/ou iterativa (retroalimentação de fases) e/ou evolutiva (software é aprimorado).

Neste contexto, neste artigo apresentaremos alguns modelos de ciclo de vida, quais sejam:

- Cascata
- Modelo em V
- Incremental
- Evolutivo
- RAD
- Prototipagem
- Espiral
- Modelo de Ciclo de Vida Associado ao RUP

Modelo em Cascata

Formalizado por Royce em 1970, é o modelo mais antigo. Suas atividades fundamentais são:

- análise e definição de requisitos;
- projeto;
- implementação;
- teste;
- integração.

O modelo em cascata tem o grande mérito de ser o primeiro a impor o planejamento e o gerenciamento ao processo de software, que antes era casual. O nome "cascata" foi atribuído em razão da sequência das fases, onde cada fase só começa quando a anterior termina; e da transmissão do resultado da fase anterior como entrada para a fase atual (o fim de cada fase resulta em um documento aprovado). Nesse modelo, portanto, é dada muita ênfase às fases de análise e projeto antes de partir para a programação, a fim de que o objetivo do software esteja bem definido e que sejam evitados retrabalhos, conforme podemos observar na **Figura 1**.

Figura 1. O modelo em cascata.

Devido à sua simplicidade, o modelo em cascata é fácil de ser entendido pelo cliente. É um modelo que supõe um início e fim claro e determinado, assim como uma estimativa precisa de custo logo no início, fatores importantes na conquista do cliente.

O problema se dá depois, quando o cliente, após esperar até o fim do processo para receber a primeira versão do sistema, pode não concordar com ela. Apesar de cada fase terminar com uma documentação aprovada, certamente haverá lacunas devido a requisitos mal descritos pelo cliente, mal entendido pelo analista ou por mudança de cenário na organização que exija adaptação de requisitos. O modelo em cascata não prevê revisão de fases.

Assim, o risco é muito alto, principalmente para sistemas complexos, de grande porte, afinal o modelo em cascata pressupõe uma realidade estática e bem conhecida, comparado a uma linha de produção fabril. Mas a rotina do negócio do cliente não reflete isso. Manipulação de usuários com diferentes habilidades, ambientes operacionais distintos, tecnologia em crescente evolução, necessidade de integração com outros sistemas (em plataformas antigas ou mais novas), mudanças organizacionais, até mudanças na legislação do município/estado/país, pedem um modelo mais flexível.

Por outro lado, o modelo em cascata adéqua-se bem como um "sub-modelo" para outros modelos. Por exemplo, no modelo "cascata com realimentação" permite-se que, a cada descoberta da fase posterior, haja uma correção da fase anterior.

Modelo em V

Neste modelo, do Ministério de Defesa da Alemanha, 1992, o modelo em cascata é colocado em forma de "V". Do lado esquerdo do V ficam da análise de requisitos até o projeto, a codificação fica no vértice e os testes, desenvolvimento, implantação e manutenção, à direita, conforme **Figura 2**.

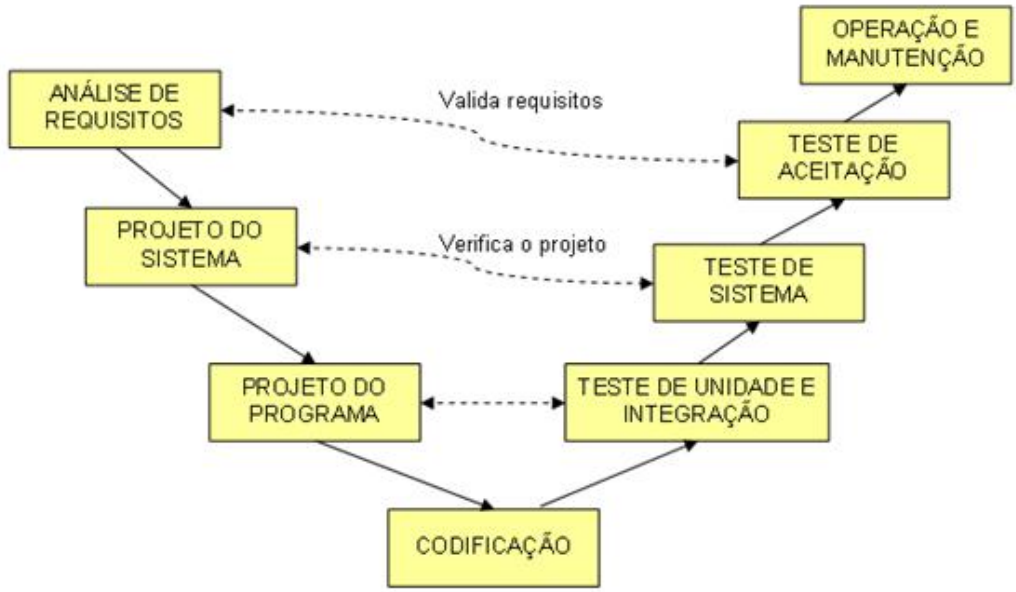

Figura 2. O modelo em V.

A característica principal desse modelo, que o diferencia do modelo em cascata, é a ênfase dada à verificação e validação: cada fase do lado esquerdo gera um plano de teste a ser executado no lado direito.

Mais tarde, o código fonte será testado, do mais baixo nível ao nível sistêmico para confirmar os resultados, seguindo os respectivos planos de teste: o teste de unidade valida o projeto do programa, o teste de sistema valida o projeto de sistema e o teste de aceitação do cliente valida a análise de requisitos.

Da mesma forma que o modelo em cascata, o cliente só recebe a primeira versão do software no final do ciclo, mas apresenta menos risco, devido ao planejamento prévio dos testes nas fases de análise e projeto.

Ciclos de Vida Incremental

Neste modelo, de Mills em 1980, os requisitos do cliente são obtidos, e, de acordo com a funcionalidade, são agrupados em módulos. Após este agrupamento, a equipe, junto ao cliente, define a prioridade em que cada módulo será desenvolvido, escolha baseada na importância daquela funcionalidade ao negócio do cliente.

Cada módulo passará por todas as fases "cascata" de projeto, conforme se observa na **Figura 3**, e será entregue ao cliente um software operacional. Assim, o cliente receberá parte do produto final em menos tempo.

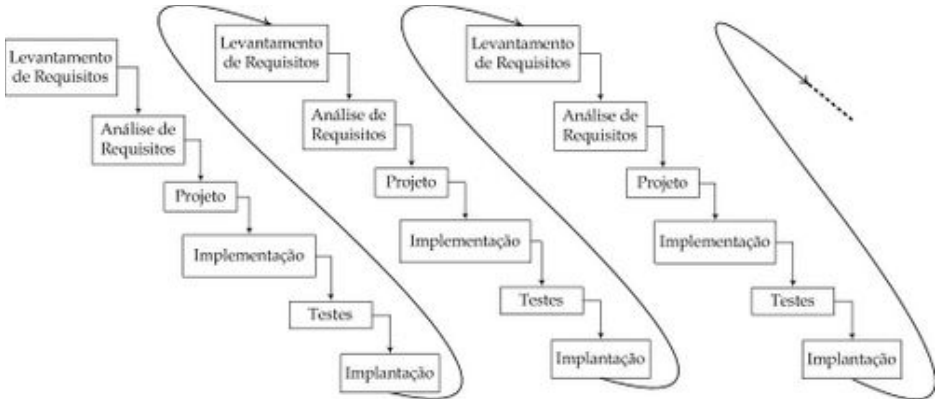

Figura 3. Ciclo de vida Incremental.

Como o cliente já trabalhará no primeiro incremento ou módulo, é muito importante que haja uma especial atenção na integração dos incrementos, o que exige muito planejamento, afinal não é aceitável que o cliente se depare com muitos erros de software a cada incremento, tampouco, que a cada incremento ele precise se readaptar a grandes mudanças. Uma atenção especial deve ser dada ao agrupamento dos requisitos e à qualidade no desenvolvimento das funções comuns a todo o sistema, que inevitavelmente deverão ser entregues no primeiro incremento.

Desta forma, além de atender as necessidades mais críticas do cliente mais cedo, as partes mais importantes serão, também, as partes mais testadas no ambiente real. Será mais difícil gastar recursos em conceitos errados, ou que um mau entendimento dos requisitos alcance uma escala difícil de ser ajustada, visto que durante todo o projeto haverá o feedback do cliente (a opinião do cliente realimenta o sistema).

Esse ciclo de vida não exige uma equipe muito grande, pois a modularização diminui o escopo de cada incremento, e não há um paralelismo nas atividades. Haverá, por outro lado, uma dificuldade em manter a documentação de cada fase atualizada devido às melhorias no sistema e aos ajustes de requisitos solicitados pelos clientes.

Modelo Evolutivo

Neste modelo, os requisitos são adquiridos em paralelo à evolução do sistema. O modelo evolutivo parte do princípio que o cliente não expõe todos os requisitos, ou os requisitos não são tão bem conhecidos, ou os requisitos ainda estão sofrendo mudanças. Desta forma, a análise é feita em cima dos requisitos conseguidos até então, e a primeira versão é entregue ao cliente. O cliente usa o software no seu ambiente operacional, e como feedback, esclarece o que não foi bem entendido e dá mais informações sobre o que precisa e sobre o que deseja (ou seja, mais requisitos).

A partir deste feedback, nova análise, projeto e desenvolvimento são realizados, e uma segunda versão do software é entregue ao cliente que, novamente, retorna com mais feedbacks. Assim, o software vai evoluindo, se tornando mais completo, até atender todas as necessidades do cliente dentro do escopo estabelecido. Tem-se assim a versão final, pelo menos até novos requisitos aparecerem (ver **Figura 4**).

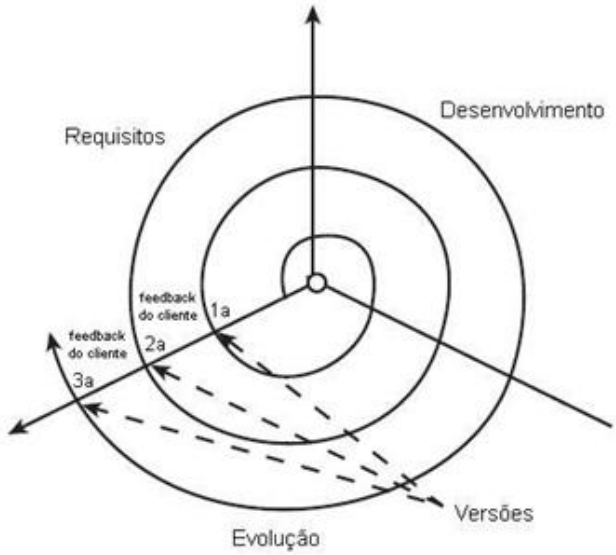

Figura 4. Ciclo de vida Evolutivo.

A participação constante do cliente é uma grande vantagem desse modelo, o que diminui o risco de má interpretação de requisitos dos modelos que só oferecem a primeira versão do software no final do processo. Da mesma forma, o software já atende algumas necessidades do cliente muito mais cedo no processo.

Não é dada muita ênfase à documentação, pois a geração de versões torna este trabalho muito árduo. Além disso, como a análise de requisitos e desenvolvimento estão sempre acontecendo, a preocupação em documentar todo o processo pode fazer com que haja atrasos na entrega.

Há uma alta necessidade de gerenciamento nesse tipo de modelo, pois a falta de documentação adequada, o escopo de requisitos não determinado, o software crescendo e estando ao mesmo tempo em produção podem ter conseqüências negativas. Seguem alguns exemplos: o sistema nunca terminar, pois o cliente sempre pede uma alteração; o sistema não ter uma estrutura robusta a falhas nem propícia a uma fácil manutenção, pelas constantes alterações; o cliente mudar de idéia radicalmente entre uma versão e outra ou revelar um requisito que exija uma versão bem diferente da anterior, fazendo com que toda a base (de dados ou de programação) precise ser revista. Os citados problemas podem implicar em um grande ônus financeiro e de tempo.

É muito importante que o cliente esteja ciente do que se **trata este ciclo de vida** e que sejam esclarecidos os limites de escopo e de tempo, para que não haja frustrações de expectativas.

RAD – “Rapid Application Development”

Este modelo formalizado por James Martin em 1991, como uma evolução da “prototipagem rápida”, destaca-se pelo desenvolvimento rápido da aplicação. O ciclo de vida é extremamente comprimido, de forma a encontrarem-se exemplos, na literatura, de duração de 60 e 90 dias. É ideal para clientes buscando lançar soluções pioneiras no mercado.

É um ciclo de vida incremental, iterativo, onde é preferível que os requisitos tenham escopo restrito. A diferença principal do ciclo anterior é o forte paralelismo das atividades, requerendo, assim, módulos bastante independentes. Aqui os incrementos são desenvolvidos ao mesmo tempo, por equipes diferentes.

Além do paralelismo, a conquista do baixo tempo se dá graças à compressão da fase de requisitos e da fase de implantação. Isso significa que, na obtenção dos requisitos, costumam-se optar por metodologias mais dinâmicas e rápidas, como workshops ao invés de entrevistas. Permite-se também um desenvolvimento inicial no nível mais alto de abstração dos requisitos visto o envolvimento maior do usuário e visibilidade mais cedo dos protótipos (ver **Figura 5**).

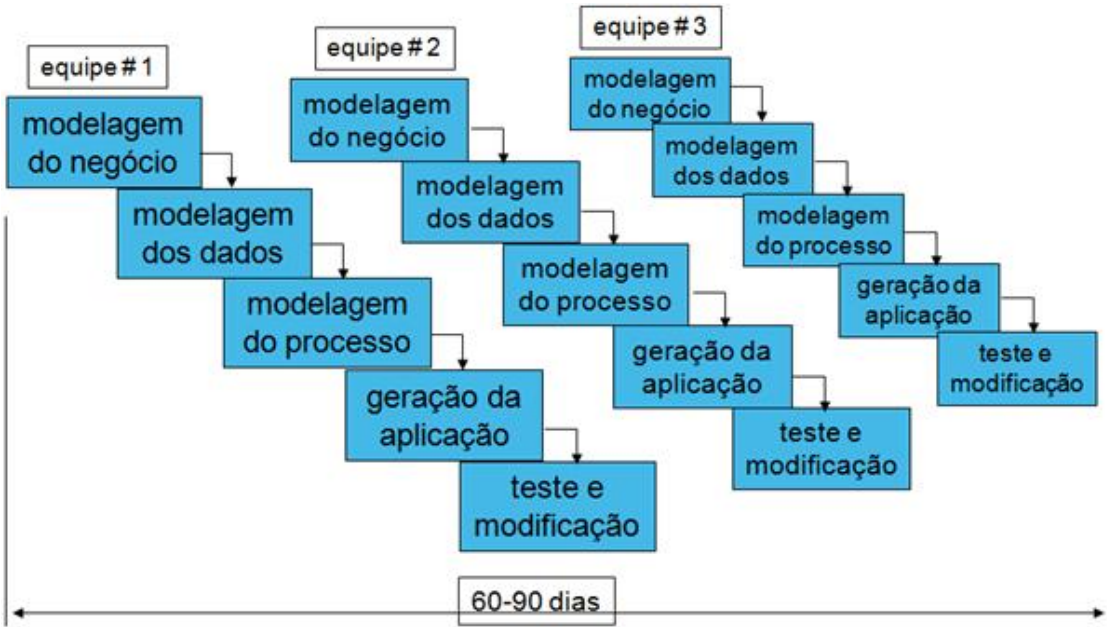

Figura 5. Ciclo de vida RAD.

As fábricas de software que resolvem por adotar este modelo devem ter uma estrutura prévia diferencial de pessoas e ferramentas, tais como:

- Pessoas:
  - Profissionais experientes (funcional e gerência);
  - Profissionais de rápida adaptação;
  - Equipes de colaboração mútua;
  - Maior quantidade de pessoas;
- Gerenciamento:
  - Empresas pouco burocráticas que encorajem a eliminação de obstáculos;
  - Alto controle do tempo;
- Uso de Ferramentas:
  - CASE;
  - Muita diagramação;
  - Prévia biblioteca de componentes reutilizáveis (APIs, wizards, templates,...);
  - Fácil manutenção (ex.: linguagens de programação que suportem Orientação a Objetos, tratamento de exceção, ponteiros);

- Adoção de ferramentas maduras, pois não há tempo de atualizar versões e tratar erros inesperados;

Os sistemas desenvolvidos no ciclo RAD tendem a ter uma padronização de telas muito forte, devido a bibliotecas reutilizáveis e templates, porém tendem a perder em desempenho do sistema e na análise de risco (atividades estas que demandam tempo em qualquer projeto). Assim, é preferível seu uso para softwares de distribuição pequena.

Prototipagem

Prototipagem é a construção de um exemplar do que foi entendido dos requisitos capturados do cliente. Pode ser considerado um ciclo de vida ou pode ser usado como **ferramenta em outros ciclos de vida**.

Um protótipo em engenharia de software pode ser o desenho de uma tela, um software contendo algumas funcionalidades do sistema. São considerados operacionais (quando já podem ser utilizados pelo cliente no ambiente real, ou seja, em produção), ou não operacionais (não estão aptos para serem utilizados em produção). Os protótipos podem ser descartados, ou reaproveitados para evoluírem até a versão final.

No ciclo de vida de prototipagem, não é exigido um conhecimento aprofundado dos requisitos num primeiro momento. Isso é bastante útil quando os requisitos não são totalmente conhecidos, são muitos complexos ou confusos. Desta forma, se o cliente não sabe expressar o que deseja (o que ocorre bastante quando não é um sistema legado), a melhor maneira de evitar que se perca tempo e recursos com uma má interpretação é a construção de modelos, ou seja, de protótipos do que o software faria.

Assim, o cliente experimentará, na prática, como o sistema ou parte dele funcionará. A partir desse primeiro contato, o cliente esclarece o que não foi bem interpretado, aprofunda alguns conceitos e até descobre um pouco mais sobre o que realmente precisa. A partir deste feedback, novos requisitos são colhidos e o projeto ganha maior profundidade. Outro protótipo é gerado e apresentado ao cliente, que retorna com mais feedbacks. Ou seja, o cliente participa ativamente do início ao fim do processo (ver **Figura 6**).

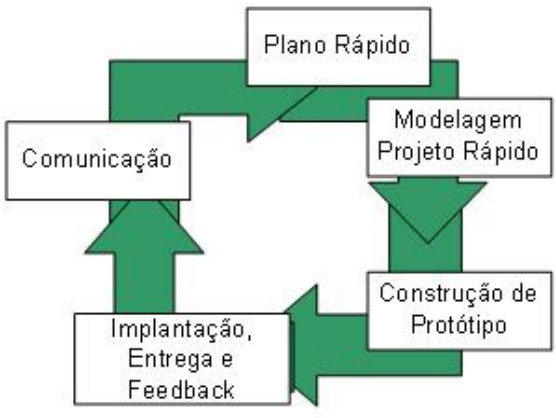

Figura 6. O modelo e prototipagem (Pressman, adaptado).

A geração de protótipos pode ser facilitada por ferramentas geradoras de telas, de relatórios, poupando esforço de programação e diminuindo o tempo de entrega.

Cada protótipo tem uma finalidade diferente. Um protótipo pode servir para esclarecer dúvidas sobre uma rotina, demonstrar a aparência das telas, conteúdo de tabelas, formato de relatórios. Os protótipos podem também ser utilizados para apresentar opções ao cliente para que ele escolha a que mais lhe agrade, como opções de navegação, de fluxo de telas, entre outras.

Por isso, é muito importante explicar previamente ao cliente que protótipos são apenas modelos para melhorar a comunicação. Caso contrário, pode causar uma frustração por não funcionar corretamente, ter funções limitadas, ter resposta lenta, ou a aparência ruim. Certamente um protótipo construído para esclarecer uma rotina provavelmente terá uma “cara feia”; para demonstrar a aparência das telas, não terá funcionalidade; para apresentar o formato dos relatórios, os dados não serão coerentes.

O cliente fará comparações entre o sistema final e o que foi “prometido” através do protótipo e pode ficar insatisfeito. Por exemplo, geralmente o protótipo não acessa rede ou banco de dados, pois as informações são “desenhadas” com a tela, fazendo com que tudo fique muito rápido. Já no ambiente operacional haverá uma degradação de desempenho e o cliente pode se decepcionar.

Faz parte de um bom gerenciamento no modelo de prototipagem planejar se, quais e que funções dos protótipos não operacionais serão reaproveitadas na versão operacional, para que sua confecção siga as boas práticas de engenharia de software. Os protótipos não operacionais são construídos com pouca qualidade em prol da velocidade. Ou seja, não há preocupação na programação, em refinar o código, em usar comentários, em aproveitar eficientemente os recursos de hardware e software, na manutenção, no reuso de componentes e na integração com outras funções ou sistemas. Com certeza será um problema se a equipe sucumbir à pressão do cliente, cada vez mais ansioso para ver a versão final daquele trabalho, e transformar à revelia, protótipos não operacionais em operacionais.

O gerente também deve se preocupar com o escopo do projeto versus a quantidade de protótipos, para que não se perca muito tempo nesse processo, tampouco se transforme num processo de “tentativa e erro”.

Não é uma tarefa fácil documentar o modelo de ciclo de vida baseado na prototipagem devido aos requisitos não serem totalmente conhecidos no primeiro momento e a consequente quantidade de mudanças ocorridas.

Modelo Espiral

O modelo proposto por Boehm em 1988 trata de uma abordagem cíclica das fases do processo, onde a cada “volta” ou iteração temos versões evolucionárias do sistema.

Este é um modelo guiado por risco, suporta sistemas complexos e/ou de grande porte, onde falhas não são toleráveis. Para isso, a cada iteração há uma atividade dedicada à análise de

riscos e apoiada através de geração de protótipos, não necessariamente operacionais (desenhos de tela, por exemplo) para que haja um envolvimento constante do cliente nas decisões.

Cada iteração ou volta é dedicada a uma fase do processo de vida de um software (viabilidade do projeto, definição de requisitos, desenvolvimento e teste,...). Ao mesmo tempo, cada volta é seccionada em 4 setores, da seguinte forma:

1ª Iteração: Viabilidade do projeto:

- 1.1. Definição de objetivos;
- 1.2. Avaliação e redução de riscos;
- 1.3. Desenvolvimento e validação;
- 1.4. Planejamento da próxima fase;

2ª Iteração: Definição de requisitos do sistema:

- 2.1. Definição dos objetivos;
- 2.2. Avaliação e redução de riscos;
- 2.3. Desenvolvimento e validação;
- 2.4. Planejamento da próxima fase;

3ª Iteração: Projeto do sistema:

- 3.1. ...
- 3.2. ...
- 3.3. ...
- 3.4. ...

4ª Iteração: Desenvolvimento e teste de unidade

- 4.1. ...
- 4.2. ...
- ...

5ª Iteração: Implantação

- ...

Ou, na representação gráfica deste modelo conforme **Figura 7**.

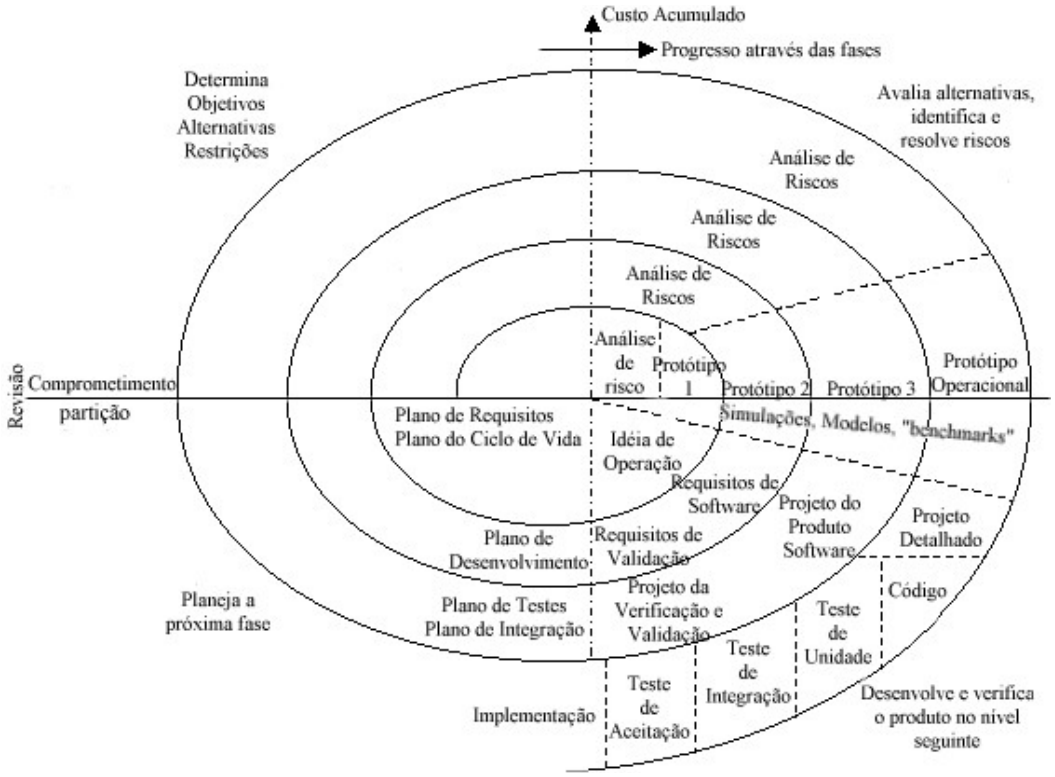

Figura 7. Modelo Espiral.

Os quatro setores são explicados da seguinte forma:

1. Na Definição de Objetivos, desempenhos, funcionalidade, entre outros objetivos, são levantados. Visando alcançar esses objetivos são listadas alternativas e restrições, e cria-se um plano gerencial detalhado.
2. Na Análise de Riscos, as alternativas, restrições e riscos anteriormente levantados são avaliados. Neste setor (porém não apenas neste) protótipos são utilizados para ajudar na análise de riscos.
3. No Desenvolvimento e Validação um modelo apropriado para o desenvolvimento do sistema é escolhido, de acordo com o risco analisado no setor anterior (cascata, interativo,...).
4. No Planejamento da Próxima fase ocorre a revisão do projeto e a decisão de partir para a próxima fase.

Ou seja, cada volta ou iteração do processo é vista por quatro ângulos.

No final da Viabilidade do Projeto teremos como resultado a Concepção das Operações; da Definição de Requisitos o produto serão os requisitos; no final do Desenvolvimento e Testes o projeto é criado e os testes habilitados. Pode-se parar por aí, pode-se incluir mais fases, pode a espiral ficar adormecida até uma nova alteração do sistema se requisitada, e desta forma estender até o fim de vida do sistema.

Neste modelo, apenas o início é definido. A evolução e amadurecimento dos requisitos demandam tempo ajustável (assim como custo). Isto torna o sistema difícil de ser vender ao cliente e exige um alto nível de gerenciamento em todo o processo.

Modelo de Ciclo de Vida Associado ao RUP

Derivado da UML e do Processo Unificado de Desenvolvimento de Software, o RUP, Rational Unified Process, é um modelo de processo iterativo e incremental, dividido em fases, orientado a casos de uso. Possui framework (esqueleto) de processo e manuais que guiam na utilização das melhores práticas de especificação de projeto (Vídeo Aula sobre Ciclo de Vida de Software, parte 3, revista Engenharia de Software Magazine).

O objetivo do RUP é produzir software com qualidade (melhores práticas de engenharia de software) que satisfaça as necessidades dos clientes dentro de um prazo e orçamento estabelecidos.

Este modelo foi desenvolvido pela Rational Software Corporation e adquirido pela IBM, que o define da seguinte maneira: “IBM Rational Unified Process®, ou RUP, é uma plataforma de processo de desenvolvimento de software configurável que oferece melhores práticas comprovadas e uma arquitetura configurável. (ver **Figura 8**)”

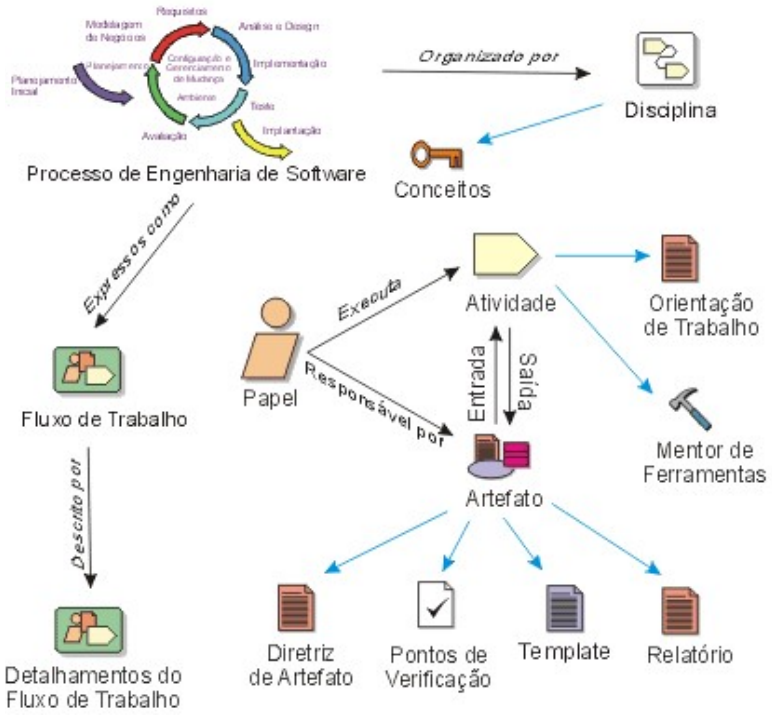

Figura 8. Conceitos chaves do RUP.

O RUP possui quatro fases de negócio. O nome de cada fase revela o que será entregue por ela (ver **Figura 9**):

- Concepção: define o escopo do projeto, ou “business case”; onde é julgado se o projeto deve ir adiante ou ser cancelado.
- Elaboração: elabora modelo de requisitos, arquitetura do sistema, plano de desenvolvimento para o software e identificar os riscos.
- Construção: constrói o software e a documentação associada.
- Transição: finaliza produto, define-se plano de entrega e entrega a versão operacional documentada para o cliente.

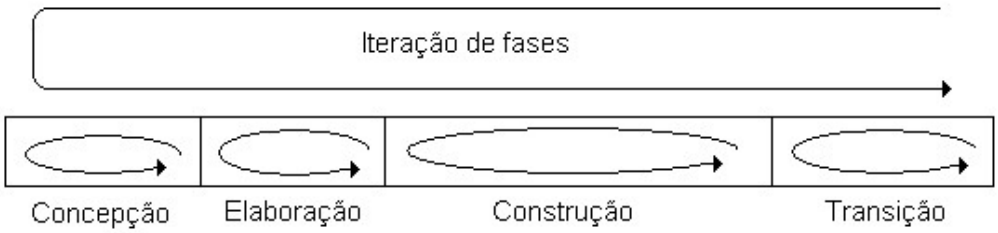

Figura 9. RUP.

A iteração no RUP tem por objetivo minimizar os riscos. Como pode ser visto na **Figura 9**, a iteração pode acontecer dentro de cada fase, gerando incrementos, ou em todo o processo. Por exemplo, dentro da concepção, a iteração pode ocorrer até que todos os requisitos sejam perfeitamente entendidos. O plano de iterações identificará quais e quantas iterações são necessárias durante o processo.

Em geral, essas fases demandam esforço e programação diferentes. Para um projeto de médio porte, de acordo com o fabricante será seguida a distribuição apresentada na **Tabela 1**.

	Concepção	Elaboração	Construção	Transição
Esforço	~5 %	20 %	65 %	10%
Programação	10 %	30 %	50 %	10 %

Tabela 1. Distribuição prevista de esforço e programação para um **ciclo de desenvolvimento inicial** típico de um projeto de médio tamanho.

O RUP usa templates que descrevem o que é esperado no resultado de cada fase ou cada iteração (IBM, 2004), identificando as competências e responsabilidades (arquiteto, analista, testador,...), as atividades e os artefatos.

Para descrever as atividades (codificação de uma classe, integração de sistemas,...) o RUP faz o uso de manuais (guidelines), que descrevem técnicas e heurísticas; e de “Mentores de Ferramentas”, que explicam o uso da ferramenta para executar a atividade. Os artefatos de cada fase (documentos, modelos, códigos, etc.) são criados, juntamente com templates e exemplos, para melhor entendimento da equipe e do cliente (ver **Figura 10**).

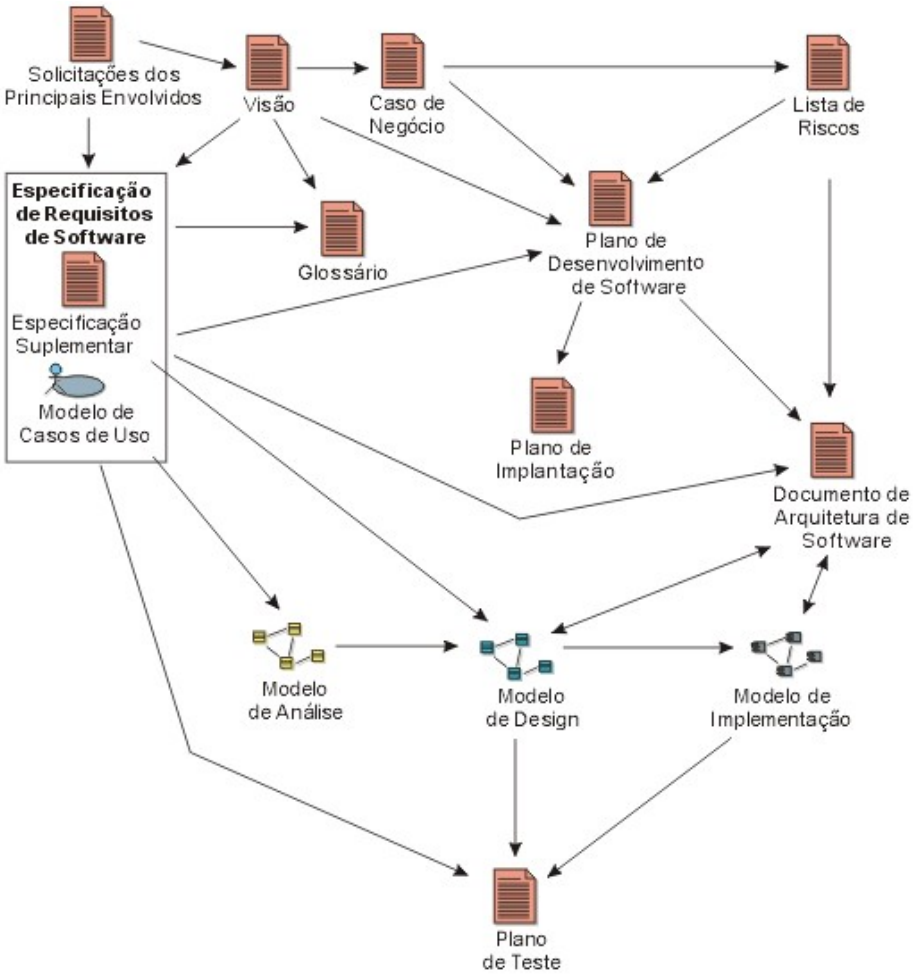

Figura 10. Os principais artefatos do Rational Unified Process e o fluxo de informações existente entre eles.

Os templates também ajudam no gerenciamento, pois definem o que precisa ser executado. Servem também como guia para que as boas práticas de especificação de projeto não sejam esquecidas no processo de desenvolvimento daquele software.

Assim, toda a preocupação dada pelo RUP em disciplinar o processo através de frameworks, guias, templates, faz com que haja uma melhor alocação de pessoas na equipe, padronização do sistema, visão concreta do andamento do projeto.

A escolha do RUP deve ser feita por empresas de software com prévia experiência, pois a definição de framework, templates, guias, métodos, entre outros, demandam tempo e exigem aderência às boas práticas de processo de software.

Considerações Finais

Finalizando este artigo sobre os **modelos de ciclo de vida de software**, segue uma tabela comparativa das principais características que devem ser observadas antes de escolher o ciclo ou os ciclos de vida a serem adotados (ver **Tabela 2**).

Vale ressaltar que, conforme já mencionado anteriormente, não existe um modelo ideal e na maioria dos softwares desenvolvidos são utilizados mais de um modelo de ciclo de vida.

Modelo	Foco	Requisitos	1ª versão p/ cliente	Gerenciamento (1=mais simples)	Sistemas (tamanho complexidade máximos)
Cascata	Documento e artefato	Bem conhecido/congelado	Fim do ciclo	1	Simples
V	Planejamento de testes	Bem conhecido/congelado	Fim do ciclo	2	Simples
Incremental	Incrementos operacionais	Maior abstração / Tratado em módulos	Protótipos operacionais	3	Médio
Evolutivo	Evolução dos requisitos	Pouco conhecidos	Protótipos operacionais	4	Médio
RAD	Rapidez	Escopo restrito / Maior abstração / Tratado em módulos	Protótipos operacionais	4	Médio
Prototipagem	Dúvidas nos Requisitos	Abstratos	Protótipos não operacionais	5	Médio
Espiral	Análise de risco	Maior abstração / evoluídos com o tempo	Protótipos operacionais ou não operacionais	5	Complexos
RUP	Frameworks e boas práticas	Maior abstração / evoluídos com o tempo	Protótipos operacionais ou não operacionais	5	Complexos

Tabela 2. Comparação entre os modelos de ciclo de vida

Saiu na DevMedia!

- Entre de cabeça no REST:
  Você já deve ter notado que o prazo para o lançamento de uma aplicação nem sempre corresponde a complexidade dos seus requisitos. Por esse motivo, é cada vez mais importante que o desenvolvedor saiba como criar e consumir APIs. Veja como nesta série.

Bibliografia

- SOMMERVILLE, Ian, Engenharia Software. Addison Wesley. 8ª ed
- PRESSMAN, Roger, Engenharia Software, McGraw-Hill. 6ª ed

- Case Maker Inc., What is Rappid Application Development?
- PISKE, Otavio. SEIDEL, Fabio, Rapid Application Development
- Norma NBR ISO/IEC 12207:1998
- SPINOLA, Rodrigo, Boas Práticas de Engenharia de Software, 2011
- PFLEGER, Shari, Engenharia de Software – Teoria e Prática, Prentice Hall, 2ª Ed
- PAULA Filho, Wilson, Engenharia de Software: fundamentos, métodos e padrões, LTC, 3ª Ed
- Vídeo Aula sobre Ciclo de Vida de Software, Revista digital Engenharia de Software Magazine

---

Saiba mais sobre Engenharia de Software ;)

- Guias de Engenharia de Software:
  Encontre aqui os Guias de estudo sobre os principais temas da Engenharia de Software. De metodologias ágeis a testes, de requisitos a gestão de projetos!
- Engenharia de Software para programadores:
  Ter boas noções sobre engenharia de Software pode alavancar muito sua carreira e a sua forma de programar. Descubra nesse guia tudo o que um programador precisa saber sobre a ciência que existe por trás dos códigos.
- Requisitos, Modelagem e UML:
  Neste guia você encontrará o conteúdo que precisa para saber como elicitar requisitos, gerenciá-los e modelar o software com as principais técnicas do mercado. Abaixo, confira os posts que te auxiliarão ao longo desse aprendizado.
- Guia de Scrum:
  Com o aumento das exigências dos clientes e prazos cada vez mais curtos, encontrar opções para possibilitar o projeto, implementação e implantação de um sistema com mais qualidade e em menos tempo é fundamental. Veja a Guia Completa sobre Scrum!