3 Modelos de Processo Prescritivos

Este capítulo apresenta os modelos de processo prescritivos, ou seja, aqueles que se baseiam em uma descrição de como as atividades são feitas. Inicialmente é apresentado o *anti-modelo* por excelência, que é *Codificar e Consertar* (Seção 3.1), consistindo na absoluta falta de processo. O *Modelo Cascata* (Seção 3.2) introduz a noção de fases bem definidas e a necessidade de documentação ao longo do processo, e não apenas para o código final. O *Modelo Sashimi* (Seção 3.3) relaxa a restrição sobre o início e fim estanques das fases do Modelo Cascata. O *Modelo V* (Seção 3.4) é uma variação do Modelo Cascata que enfatiza a importância dos testes em seus vários níveis. O *Modelo W* (Seção 3.5) enriquece o Modelo V com um conjunto de fases de planejamento de testes em paralelo com as atividades de análise e não apenas no final do processo. O *Modelo Cascata com Subprojetos* (Seção 3.6) introduz a possibilidade de dividir para conquistar, onde subprojetos podem ser desenvolvidos em paralelo ou em momentos diferentes, e o *Modelo Cascata com Redução de Risco* (Seção 3.7) enfatiza a necessidade de tratar inicialmente os maiores riscos e incertezas do projeto antes de se iniciar um processo de desenvolvimento com fases bem definidas. O *Modelo Espiral* (Seção 3.8) é uma organização de ciclo de vida voltada ao tratamento de risco, iteratividade e prototipação. A *Prototipação Evolucionária* (Seção 3.9) é uma técnica que pode ser entendida como um modelo independente ou parte de outro modelo, onde protótipos cada vez mais refinados são apresentados ao cliente para que o entendimento sobre os requisitos evolua de forma suave e consistente. O *Modelo Entrega em Estágios* (Seção 3.10) estabelece que partes do sistema, que já estejam prontas, podem ser entregues antes de o projeto ser finalizado, e que isso pode ser planejado. O *Modelo Orientado a Cronograma* (Seção 3.11) indica que se pode priorizar requisitos de forma que se o tempo disponível acabar antes do projeto, pelo menos os requisitos mais importantes terão sido incorporados. A *Entrega Evolucionária* (Seção 3.12) é um misto de Prototipação Evolucionária e Entrega em Estágios onde se pode adaptar o cronograma de desenvolvimento às necessidades de última hora, em função de prioridades definidas. Os *modelos orientados a ferramentas* (Seção 3.13) são modelos específicos de ferramentas CASE e geradores de código que são aplicados juntamente com essas ferramentas. Finalmente, o capítulo apresenta as *Linhas de Produto de Software* (Seção 3.14) que são uma abordagem moderna para reusabilidade planejada em nível organizacional.

Para que o desenvolvimento de sistemas deixe de ser artesanato e aconteça de forma mais previsível e com maior qualidade é necessário inicialmente que se compreenda e se estabeleça um processo de produção.

Processos normalmente são construídos de acordo com modelos ou estilos. *Modelos de processo* aplicados ao desenvolvimento de software também são chamados de "*ciclos de vida*". Pode-se afirmar que existem duas grandes famílias de modelos: os *prescritivos*, abordados neste capítulo, e os *ágeis*, que são apresentados no Capítulo 4.

O mais emblemático modelo prescritivo é o modelo *Waterfall* ou *Cascata*, com suas variações, cuja característica é a existência de fases bem definidas e sequenciais. Também se pode citar o Modelo Espiral, cuja característica é a realização de ciclos de prototipação para redução de riscos de projeto.

Cada um dos ciclos de vida existentes, sejam eles correntemente aplicados ou não no desenvolvimento de software, trouxeram uma característica própria. Essas características inovadoras dos diferentes modelos foram em grande parte capitalizadas pelo Processo Unificado, descrito no Capítulo 5. Pode-se resumir da seguinte forma a contribuição de cada um dos modelos listados neste livro:

a) *Codificar e Consertar*. É considerado o modelo *ad-hoc*, ou seja, aquele que acaba sendo utilizado quando não se utiliza conscientemente nenhum modelo. Sendo assim, ele não traz também nenhuma contribuição, sendo considerado o marco zero dos modelos.
b) *Cascata*. O Modelo Cascata introduz a noção de que o desenvolvimento de software ocorre em fases bem definidas, e que é necessário produzir não apenas o código executável, mas também documentos que ajudem a visualizar o sistema de forma mais abstrata do que o código.
c) *Sashimi*. O modelo Sashimi, derivado do Modelo Cascata, introduz a noção de que as fases de desenvolvimento não são estanques, mas que em um determinado momento o processo de desenvolvimento pode estar simultaneamente em mais de uma das suas fases definidas.
d) *Cascata com Subprojetos*. Este modelo introduziu a divisão do projeto em subprojetos de menor porte. Seu lema é "dividir para conquistar". Os projetos menores podem ser desenvolvidos em paralelo por várias equipes, ou por uma única equipe, e, diferentes momentos do tempo, o que se assemelha com o desenvolvimento iterativo em ciclos. Porém, no Modelo Cascata com Subprojetos a integração usualmente ocorre apenas no final do desenvolvimento de todos os subprojetos.
e) *Cascata com Redução de Risco* e *Espiral*. Estes dois modelos introduzem a noção de que riscos são fatores determinantes para o sucesso de um projeto de software e, portanto, devem ser os primeiros aspectos a serem analisados. Assim, uma espiral de desenvolvimento é iniciada onde, a cada ciclo, um ou mais riscos são resolvidos ou minimizados. Apenas ao final destes ciclos de redução de risco é que o desenvolvimento propriamente dito inicia.
f) *Modelo V* e *Modelo W*. Estes modelos enfatizam a importância do teste no desenvolvimento de software e indicam que esta deve ser uma preocupação constante, e não apenas uma etapa colocada ao final do processo de desenvolvimento.
g) *Modelo Orientado a Cronograma*. Este modelo introduziu a noção de que se pode trabalhar prioritariamente com as funcionalidades mais importantes, deixando as menos importantes para o final, de forma que se houverem atrasos e o prazo for rígido, pelo menos as funcionalidade mais importantes serão entregues no prazo.
h) *Entrega em Estágios*. – Este modelo introduz a noção de que é possível planejar e entregar partes prontas do sistema antes do final do projeto.
i) *Prototipação Evolucionária.* Este modelo propõe o uso de protótipos para ajudar a compreensão da arquitetura, interface e mesmo dos requisitos do sistema, o que é extremamente útil quando não é possível conhecer bem estes aspectos a priori.
j) *Entrega Evolucionária*. Este modelo combina Prototipação Evolucionária com Entrega em Estágios, mostrando que é possível se fazer um planejamento adaptativo, onde o

gerente de projeto decide a cada nova iteração se vai acomodar as requisições de mudança que surgiram ao longo do projeto ou se vai se manter fiel ao planejamento inicial.

k) *Modelos Orientados a Ferramentas*. É uma família de modelos cuja única semelhança usualmente consiste no fato de que são indicados para uso com ferramentas específicas.
l) *Linhas de Produto de Software*. É um tipo de processo que se aplica somente se a empresa vai desenvolver uma família de produtos semelhantes. A linha de produto de software permite, então, que a reusabilidade de componentes seja efetivamente planejada, e não apenas fortuita.
m) *Modelos Ágeis*. É uma família de modelos onde o foco está nos fatores humanos e não na descrição de tarefas, o que os diferencia dos modelos prescritivos.
n) *Processo Unificado*. Este modelo busca capitalizar e aprimorar todas as características dos modelos anteriores, consistindo de um modelo iterativo, focado em riscos, que valoriza o teste e a prototipação entre outras características. Ele possui diferentes implementações, como a clássica RUP, as ágeis AUP e OpenUP, a orientada a ferramentas OUM e a específica para sistemas de grande porte, RUP-SE.

Portanto, existem vários tipos de ciclos de vida, alguns vêm caindo em desuso, enquanto outros evoluem a partir deles. O engenheiro de software deve escolher aquele mais adequado à sua equipe e ao projeto que vai desenvolver. Se ele escolher bem, terá um processo eficiente, baseado em padrões e lições aprendidas e com possibilidade de capitalizar experiências. O controle será eficiente e os riscos, erros e retrabalho serão minimizados.

Mas, se o engenheiro de software escolher um modelo inadequado para sua realidade poderá estar gerando trabalho repetitivo e frustrante para a equipe, o que, aliás, pode acontecer também quando não se escolhe modelo algum.

Projetos diferentes têm diferentes necessidades. Assim, não há um modelo que seja sempre melhor do que outros. Mesmo o Processo Unificado não se adapta bem a sistemas cujos requisitos não sejam baseados em casos de uso, como é o caso do software científico, jogos e compiladores. Para escolher um ciclo de vida, ou pelo menos as características de processo necessárias para seu projeto e empresa, o engenheiro de software pode tentar responder às seguintes perguntas:

a) *Quão bem os analistas e o cliente podem conhecer os requisitos do sistema*? O entendimento sobre o sistema poderá a mudar à medida que o desenvolvimento avançar? Se os requisitos são estáveis, pode-se trabalhar com modelos mais previsíveis como Modelo Cascata ou Modelo V. Porém, requisitos instáveis ou mal compreendidos exigem ciclos de redução de risco e modelos baseados em prototipação, Espiral ou ainda métodos ágeis.
b) *Quão bem é compreendida a arquitetura do sistema*? É provável que sejam feitas grandes mudanças de rumo ao longo do projeto? Uma arquitetura bem compreendida e estável permite o uso de modelos como Cascata com Subprojetos, mas arquiteturas mal compreendidas precisam de protótipos e desenvolvimento iterativo para reduzir o risco.

c) *Qual o grau de confiabilidade necessário em relação ao cronograma*? O Modelo Orientado a Cronograma, Processo Unificado e Métodos Ágeis prezam o cronograma, ou seja, na data definida esses modelos deverão permitir a entrega de alguma funcionalidade, possivelmente não toda, mas alguma coisa estará pronta. Já os modelos Cascata, Espiral e Prototipação são bem menos previsíveis em relação a cronograma.

d) *Quanto planejamento é efetivamente necessário*? Os modelos prescritivos usualmente privilegiam o planejamento com antecedência. Já os modelos ágeis preferem um planejamento menos detalhado e adaptação às condições do projeto à medida que ele vai evoluindo. O Processo Unificado faz um planejamento genérico para o longo prazo e planejamento detalhado apenas para o próximo ciclo iterativo que vai iniciar.

e) *Qual o grau de risco que este projeto apresenta*? Existem ciclos de vida especialmente voltados à minimização dos riscos de projeto nos primeiros instantes. Entre eles: Modelo Espiral, Cascata com Redução de Risco, métodos ágeis e o Processo Unificado.

f) *Existe alguma restrição de cronograma*? Se a data de entrega do sistema é definitiva e inadiável o Modelo Orientado a Cronograma ou uma variante deste deveria ser escolhida.

g) *Será necessário entregar partes do sistema funcionando antes de terminar o projeto todo*? Alguns ciclos de vida, como Cascata com Subprojetos, prevêem a integração do software apenas no final do desenvolvimento. Já os métodos ágeis e Processo Unificado sugerem que se pratique a integração contínua, o que pode viabilizar entregas de partes do sistema ao longo do processo de desenvolvimento.

h) *Qual o grau de treinamento e adaptação necessários para a equipe poder utilizar o ciclo de vida que parece mais adequado ao projeto*? Nenhum ciclo de vida, exceto Codificar e Testar, é trivial. Todos exigem certa dose de preparação por parte da equipe. Porém, os prescritivos, por definirem as tarefas detalhadamente, podem ser mais adequados a equipes inexperientes, enquanto que os métodos ágeis, focados em valores humanos, usualmente necessitam de desenvolvedores mais experientes.

i) *Será desenvolvido um único sistema ou uma família de sistemas semelhantes*? Caso mais do que dois sistemas semelhantes sejam desenvolvidos, pode ser o caso de se investir em uma linha de produto de software, que permite lidar com as partes em comum e as diferenças entre os sistemas aplicando reuso de forma planejada e institucionalizada.

j) *Qual o tamanho do projeto*? Projetos que possam ser realizados por equipes pequenas (tipicamente até 8 ou 10 desenvolvedores) podem se adequar melhor aos modelos ágeis, enquanto que projetos de grande porte precisarão de processos mais formais como *RUP-SE*. *Crystal clear*, por outro lado, é um modelo de processos que se adapta ao tamanho do projeto.

Via de regra, o que se observa é que é mais útil escolher um modelo de processo simples, mas executá-lo coerentemente e de forma bem gerenciada, do que escolher um modelo sofisticado, mas mal executado e mal gerenciado.

3.1 Codificar e Consertar

O *Modelo Codificar e Consertar* (*Code and Fix*) tem uma filosofia muito simples, mostrada na Figura 3-1.

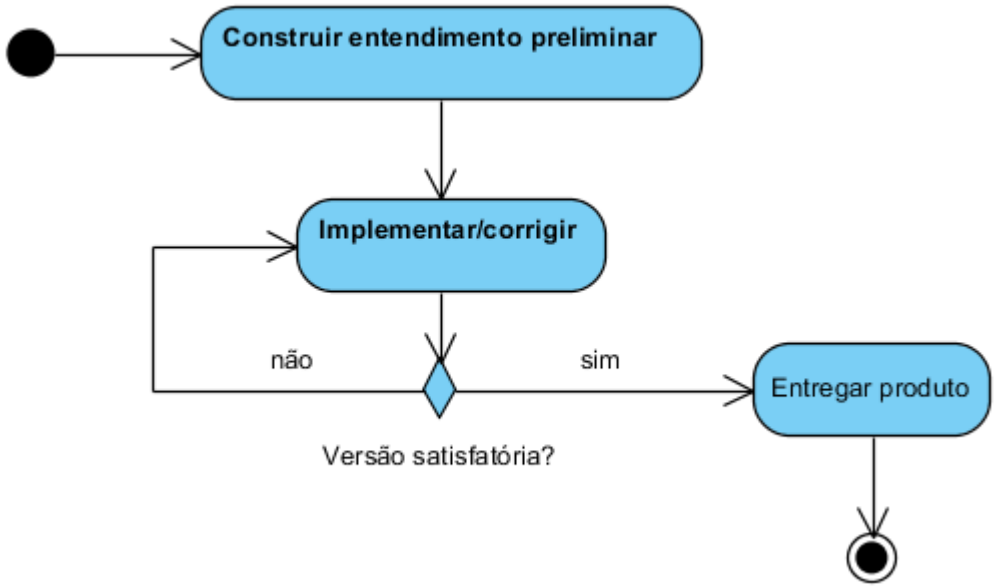

Figura 3-1: Modelo Codificar e Consertar[17].

Em resumo, o modelo consiste em:

a) Construir junto com o cliente um entendimento preliminar sobre o sistema que deve ser desenvolvido.
b) Implementar uma primeira versão deste sistema.
c) Interagir com o cliente de forma a corrigir a versão preliminar até que esta satisfaça ao cliente.
d) Fazer testes e corrigir os inevitáveis erros.
e) Entregar o produto.

Pode-se dizer que é uma forma bastante ingênua de modelo de processo. Pode-se dizer também que nem sequer parece ser um modelo de processo, pois não há previsibilidade em relação às atividades e resultados obtidos.

O modelo é usado porque é simples, não porque funciona bem. Muitos projetos reais, mesmo dirigidos por outros modelos de ciclo de vida, por vezes caem na prática de Codificar e Consertar em função de pressão de cronograma.

Empresas que não usam modelo de processo algum possivelmente usam Codificar e Consertar por *default*.

O Modelo Codificar e Consertar é, possivelmente, bastante usado em empresas de pequeno porte, mas deve ser entendido mais como uma maneira de pressionar programadores (*code rush*) do que como um processo organizado (McConnell, 1996)[18]. Se o sistema não satisfaz o cliente, cobra-se do programador uma versão funcional adequada no menor tempo possível.

[17] Todos os diagramas do livro seguem a notação UML, exceto quando for explicitamente informado.
[18] my.safaribooks*online*.com/9780735634725

Apesar de tudo, esta técnica ainda pode ser usada quando se trata de desenvolver sistemas muito pequenos, em intervalos de poucos dias. Também pode ser usado para implementar sistemas que serão descartados, como provas de conceito ou protótipos (na abordagem de prototipação *throw-away*). Suas vantagens são:

a) Não se gasta tempo com documentação, planejamento ou projeto: vai-se direto à codificação.
b) O progresso é facilmente visível à medida que o programa vai ficando pronto.
c) Não há necessidade de conhecimentos ou treinamento especiais. Qualquer pessoa que programe pode desenvolver software com este modelo de ciclo de vida.

Um dos problemas com esta técnica é que o código que sofreu várias correções fica cada vez mais difícil de modificar (ver Capítulo 14). Além disso, com bastante frequência o que o cliente menos precisa é de código ruim produzido rapidamente; ele precisa ter certas necessidades atendidas. Essas necessidades são levantadas na fase de requisitos, que se for feita às pressas, deixará de atingir estes objetivos. Outras desvantagens citadas são:

a) É muito difícil avaliar a qualidade e os riscos do projeto.
b) Se no meio do projeto a equipe descobrir que decisões arquiteturais estavam erradas, não há solução a não ser começar tudo de novo.

Além disso, Codificar e Consertar sem um plano de teste sistemático tende a produzir sistemas instáveis e com grande probabilidade de conterem erros.

3.2 Modelo Cascata

O *Modelo Cascata* (*Waterfall* ou *WFM*) começou a ser definido nos anos 1970 e apresenta um ciclo de desenvolvimento bem mais detalhado e previsível do que o Modelo Codificar e Consertar. O Modelo Cascata é considerado o avô de todos os ciclos de vidae baseia-se na filosofia *BDUF* (*Big Design Up Front*), que propõe que antes de produzir linhas de código deve-se fazer um trabalho detalhado de análise e projeto, de forma que, quando o código for efetivamente produzido, esteja o mais próximo possível dos requisitos do cliente.

O modelo prevê uma atividade de revisão ao final de cada fase para que se avalie se o projeto pode seguir para a fase seguinte. Se a revisão mostrar que o projeto não está pronto para seguir para a fase seguinte, então ele deve permanecer na mesma fase (Ellis, 2010)[19].

O Modelo Cascata é dirigido por documentação, já que é ela que determina se as fases foram concluídas ou não.

Boehm (1981) apresenta uma série de marcos (*milestones*) que podem ser usados para delimitar as diferentes fases do Modelo Cascata:

a) *Início da fase de planejamento e requisitos*. Marco *LCR*, *Life-cicle Concept Review*, completar a revisão dos conceitos do ciclo de vida:
    a. Arquitetura de sistema aprovada e validada, incluindo questões básicas de hardware e software.

[19] www.eng.uwi.tt/depts/civil/pgrad/proj_mgmt/courses/prmg6009/Notes%20PDF/04_L3.pdf

b. Conceito de operação aprovado e validado, incluindo questões básicas de interação humano-computador.
c. Plano de ciclo de vida de alto nível, incluindo marcos, recursos, responsabilidades, cronogramas e principais atividades.

b) *Fim da fase de planejamento e requisitos – início da fase de design de produto*. Marco *SRR*, *Software Requirements Review*, completar a revisão dos requisitos do software:
   a. Plano de desenvolvimento detalhado: detalhamento de critérios de desenvolvimento de marcos, orçamento e alocação de recursos, organização da equipe, responsabilidades, cronograma, atividades, técnicas e produtos a serem usados.
   b. Plano de uso detalhado: contraparte para os itens do plano de desenvolvimento como treinamento, conversão, instalação, operações e suporte.
   c. Plano de controle de produto detalhado: plano de gerenciamento de configuração, plano de garantia de qualidade, plano geral de V&V (verificação e validação), excluindo detalhes dos planos de testes.
   d. Especificações de requisitos de software aprovadas e validadas: requisitos funcionais, de performance e especificações de interfaces validadas em relação a completude, consistência, testabilidade e exequibilidade.
   e. Contrato de desenvolvimento (formal ou informal) aprovado baseado nos itens acima.

c) *Fim da fase de design de produto – início da fase de design detalhado*. Marco *PDR*, *Product Design Review*, completar a revisão do *design* do produto:
   a. Especificação do design do produto de software verificada.
   b. Hierarquia de componentes do programa, interfaces de controle e dados entre as unidades (uma *unidade* de software realiza uma função bem definida, pode ser desenvolvida por uma pessoa e tipicamente tem de 100 a 300 linhas de código).
   c. Estruturas de dados lógicas e físicas detalhadas em nível de seus campos.
   d. Orçamento para recursos de processamento de dados (incluindo especificações de eficiência de tempo, capacidade de armazenamento e precisão).
   e. Verificação do *design* com referência a completude, consistência, exequibilidade e rastreabilidade aos requisitos.
   f. Identificação e resolução de todos os riscos de alta importância.
   g. Plano de teste e integração preliminar, plano de teste de aceitação e manual do usuário.

d) *Fim da fase de design detalhado – início da fase de codificação e teste de unidade*. Marco *CDR*, *Critical Design Review*, completar o design e revisar aspectos críticos do *design* das unidades:
   a. Especificação de *design* detalhado revisada para cada unidade.
   b. Para cada rotina (menos de 100 instruções) dentro de uma unidade, espeficicar nome, propósito, hipóteses, tamanho, sequencia de chamadas, entradas, saídas, exceções, algoritmos e fluxo de processamento.
   c. Descrição detalhada da base de dados.

    d. Especificações e orçamentos de *design* verificados em relação a completude, consistência e rastreabilidade em relação aos requisitos.
    e. Plano de teste de aceitação aprovado.
    f. Manual do usuário, e rascunho do plano de teste e integração completados.

e) *Fim da fase de codificação e teste de unidade – início da fase de integração e teste*. Marco *UTC*, *Unit Test Criteria*, satisfação dos critérios de teste de unidade:
    a. Verificação de todas as unidades de computação usando não apenas valores nominais, mas também valores singulares e extremos (ver Seção 13.5.2).
    b. Verificação de todas as entradas e saídas unitárias, incluindo mensagens de erro.
    c. Exercitar todos os procedimentos executáveis e todas as condições de teste (ver Seção 13.4).
    d. Verificação de conformação a padrões de programação.
    e. Documentação em nível de unidade completada.

f) *Fim da fase de integração e teste – início da fase de implantação*. Marco *SAR*, *Software Acceptance Review*, completar a revisão da aceitação do software:
    a. Testes de aceitação do software satisfeitos.
    b. Verificação da satisfação dos requisitos do software.
    c. Demonstração de performance aceitável acima do nominal, conforme especificado.
    d. Aceitação de todos os produtos do software: relatórios, manuais, especificações e bases de dados.

g) *Fim da fase de implantação – início da fase de operação e manutenção*. Marco *SyAR*, *System Acceptance Review*, completar a revisão da aceitação do sistema:
    a. Satisfação do teste de aceitação do sistema.
    b. Verificação da satisfação dos requisitos do sistema.
    c. Verificação da prontidão operacional do software, hardware, instalações e pessoal.
    d. Aceitação de todas as entregas relacionadas ao sistema: hardware, software, documentação, treinamento e instalações.
    e. Todas as conversões especificadas e atividades de instalação foram completadas.

h) *Fim da fase de operação e manutenção*. Corresponde à aposentadoria do sistema:
    a. Foram completadas todas as atividades do plano de aposentadoria: conversão, documentação, arquivamento e transição para um novo sistema.

As fases e marcos, conforme apresentados por Boehm são resumidos na Figura 3-2.

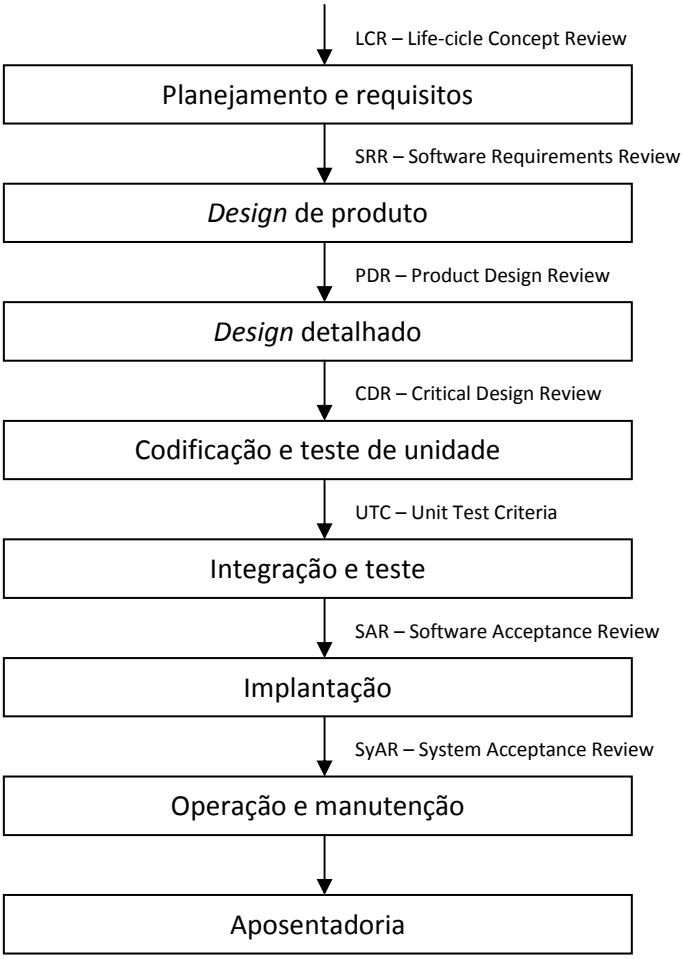

Figura 3-2: Fases e marcos do Modelo Cascata de acordo com Boehm.

As ideias fundamentais do Modelo Cascata são coerentes, em uma primeira abordagem, e podem gerar benefícios relevantes:

a) *A existência de fases bem definidas ajuda a detectar erros cedo*. Desta forma é mais barato corrigi-los.
b) *O modelo procura promover a estabilidade dos requisitos*. O projeto só se move em frente quando os requisitos são aceitos.
c) *Funciona bem com projetos onde os requisitos são bem conhecidos e estáveis*, já que este tipo de projeto se beneficia de uma abordagem organizada e sistemática.
d) *Funciona bem quando a preocupação com qualidade está acima das preocupações com custo ou tempo de desenvolvimento*, visto que a eliminação de mudanças ao longo do projeto minimiza uma conhecida fonte de erros.
e) *É adequado para equipes tecnicamente fracas ou inexperientes*, pois dá estrutura ao projeto, servindo de guia e evitando o esforço inútil.

Um dos problemas com esta abordagem é que é mais fácil, usualmente, verificar se código funciona direito, mas não é tão fácil verificar se modelos e projetos estão bem escritos. Para ser efetivamente viável, este tipo de ciclo de vida necessitaria de ferramentas de análise automatizada de diagramas e documentos para verificar sua exatidão. Mas tais ferramentas são ainda bastante limitadas.

O Modelo Cascata também tem sido um dos mais criticados da história, especialmente pelos adeptos dos modelos ágeis que valorizam princípios diametralmente opostos aos do Modelo Cascata.

Ellis (2010), aponta uma série de problemas com o Modelo Cascata:

a) *Não produz resultados tangíveis até a fase de codificação*, exceto para as pessoas familiarizadas com as técnicas de documentação, que poderão ver significado nos documentos.
b) *É difícil estabelecer requisitos completos antes de começar a codificar*. O desenvolvimento de software é entendido hoje mais como um processo de amadurecimento do que uma construção que possa ser baseada em um projeto detalhado desde o início. É natural, em projetos de software, que alguns requisitos só sejam descobertos durante o desenvolvimento do projeto.
c) *Desenvolvedores sempre reclamam que os usuários não sabem expressar o que precisam*. Os usuários não são especialistas em computação então não mencionam muitas vezes os problemas mais óbvios, que só aparecerão quando o produto estiver em operação.
d) *Não há flexibilidade com requisitos*. Voltar atrás para corrigir requisitos mal estabelecidos ou que mudaram é bastante trabalhoso.

O Modelo Cascata é estritamente sequencial. Sua criação é atribuída a Royce (1970)[20], que o apresentou pela primeira vez, embora não usasse, na época, a expressão "*Waterfall*" para designá-lo (Figura 3-3).

[20] www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf

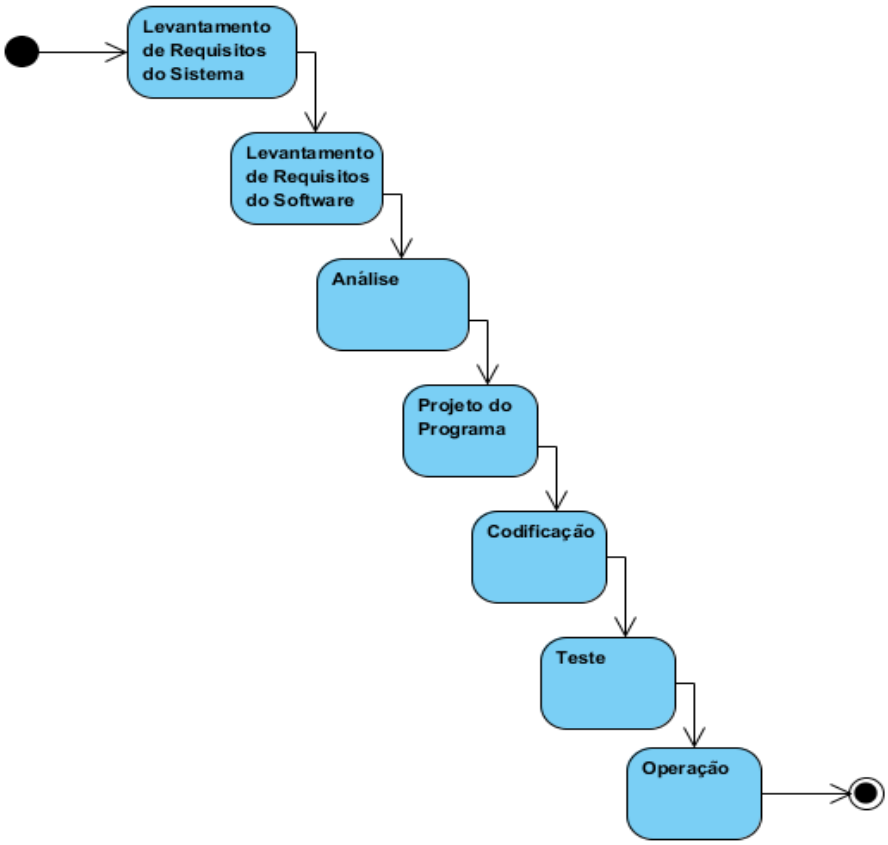

Figura 3-3: Modelo Cascata original (1970).

O irônico da questão é que Royce justamente apresentou este modelo como algo que *não* poderia ser seguido. Royce comenta que embora acreditasse no modelo como filosofia de projeto organizado, ele achava a implementação (conforme mostrado na figura) bastante arriscada, já que apenas na fase de teste vários aspectos do sistema seriam, pela primeira vez, experimentados na prática. Desta forma, ele esperava (e isso se confirma na prática), que após a fase de teste, muito retrabalho seria necessário para alterar os requisitos e a partir deles todo o projeto.

Na sequência de seu artigo, Royce busca apresentar sugestões que diminuam a fragilidade desse modelo. Inicialmente, ele propõe que problemas encontrados em uma fase possam ser resolvidos retornando à fase anterior para efetuar as correções. Por exemplo, problemas na codificação poderiam ser resolvidos se o projeto fosse refeito. O modelo resultante, muitas vezes apresentado como ciclo de vida *Cascata Dupla* é mostrado na Figura 3-4.

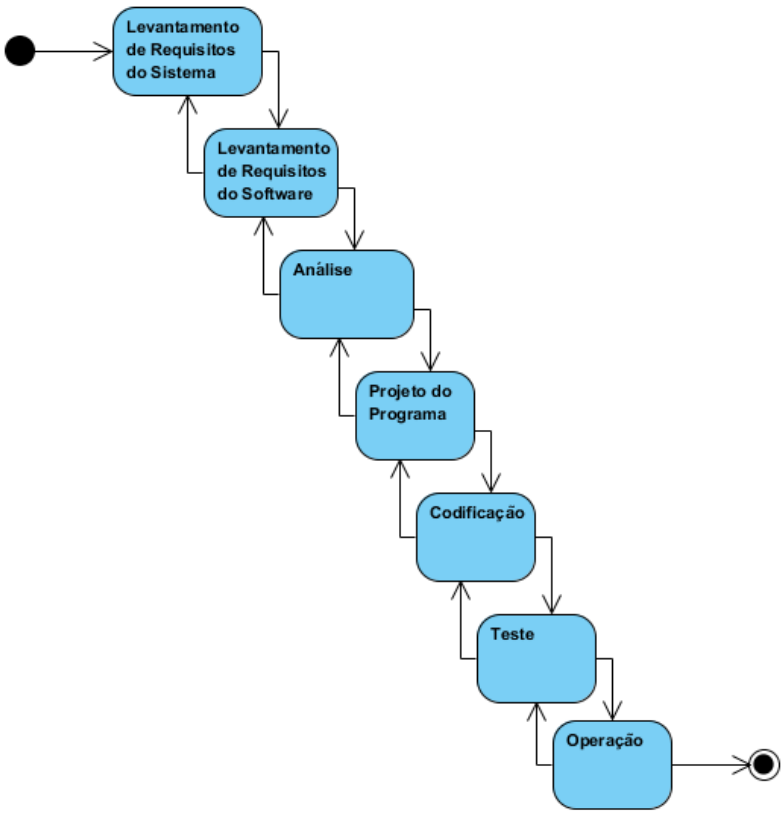

Figura 3-4: Como as interações entre fases *poderiam* ser (Modelo Cascata Dupla).

Novamente, Royce apresenta este modelo como um algo que não poderia funcionar bem na prática. Ele diz que com alguma sorte as interações entre as fases poderiam ser feitas como na Figura 3-4. Mas, como mostra a Figura 3-5, não é isso o que acaba acontecendo. De acordo com essa figura, problemas encontrados em uma fase algumas vezes não foram originados na fase imediatamente anterior, mas várias fases antes. Assim, nem o Modelo Cascata Dupla, nem o Modelo Sashimi (Seção 3.3) apresentam uma solução satisfatória para este problema.

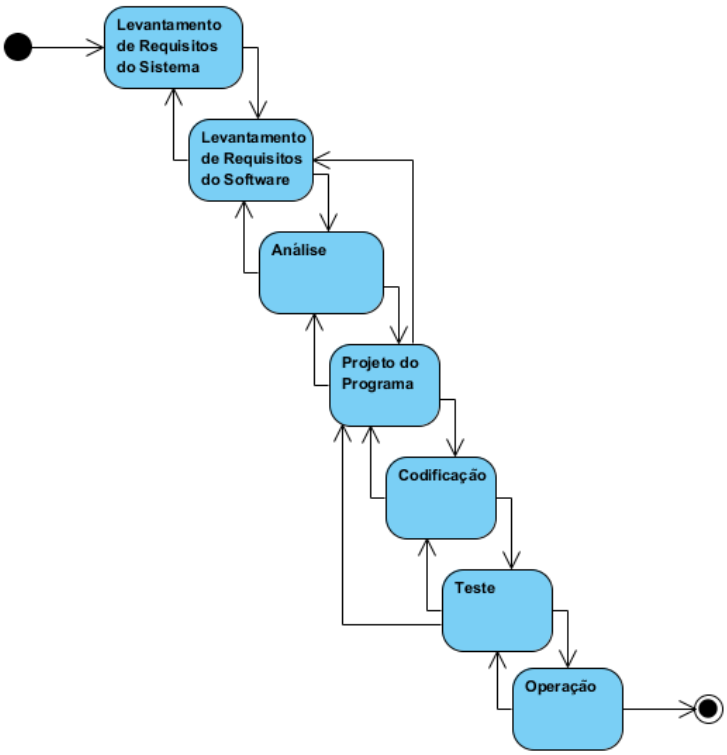

Figura 3-5: Como as interações entre fases do Modelo Cascata acabam acontecendo na prática.

Então, para contornar o problema de retornar às fases anteriores, Royce apresenta cinco propostas, que visam produzir uma maior estabilidade dentro das fases do modelo, minimizando a necessidade de retornos:

a) *Inserir uma fase de design entre o levantamento dos requisitos e sua análise*. Se *designers* que conheçam as limitações de sistemas computacionais puderem verificar os requisitos antes dos analistas iniciarem seus trabalhos, eles poderão adicionar preciosos requisitos suplementares referentes às limitações físicas do sistema. Nos modelos modernos, os ciclos de redução de risco permitem realizar esta atividade.
b) *Produzir documentação*. Nas fases iniciais a documentação é o produto esperado, e deve ser feita com a mesma qualidade com que se procura fazer o produto. Caso se trabalhe com a filosofia de gerar código automaticamente, os modelos deverão ser tão precisos quanto o código seria.
c) *Fazer duas vezes*. Sugere-se que o produto efetivamente entregue ao cliente seja a segunda versão produzida. Ou seja, executam-se as fases do Modelo Cascata duas vezes, usando o conhecimento aprendido na primeira rodada para produzir um produto melhor na segunda vez. Esta ideia foi incorporada nos ciclos baseados em prototipação e iterações.
d) *Planejar, controlar e monitorar o teste*. Testes devem ser sistemáticos e realizados por especialistas, já que são críticos para o sucesso do sistema. Modernamente o Modelo V (Seção 3.4), Modelo W (Seção 3.5), Processo Unificado (Capítulo 5) e modelos ágeis (Capítulo 4) apresentam a disciplina de teste (Capítulo 13) como algo fundamental no projeto do software.

e) *Envolver o cliente*. É importante envolver o cliente formalmente no processo e não apenas na aceitação do produto final. Os modelos ágeis, especialmente, consideram o cliente como parte da equipe de desenvolvimento.

O Modelo Cascata, na sua forma mais simples é impraticável. Algumas variações foram então propostas ao longo do tempo para permitir a aplicação deste tipo de ciclo de vida em processos de desenvolvimento de software reais. Algumas destas variações são apresentadas nas seções seguintes.

3.3 Sashimi (Cascata Entrelaçado)

O modelo conhecido como *Sashimi* (DeGrace & Stahl, 1990), ou *Cascata Entrelaçado* (*Overlapped Waterfall*), é uma tentativa de atenuar a característica *BDUF* do Modelo Cascata. Ao invés de cada fase produzir documentação completa para a fase seguinte, o modelo *Sashimi* propõe que cada fase procure iniciar o tratamento de questões da fase seguinte e continue tratando as questões da fase anterior.

O diagrama de atividades da UML não é adequado para representar as atividades que se entrelaçam no modelo Sashimi devido ao seu paralelismo difuso. Classicamente ele tem sido representado como na Figura 3-6, de onde vem seu nome devido à aparência com a comida japonesa composta por cortes de peixe superpostos.

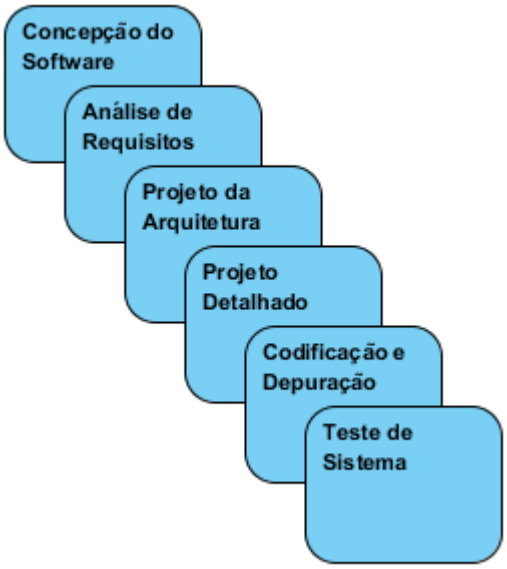

Figura 3-6: Ciclo de Vida Sashimi.

A ideia do modelo Sashimi, de que cada fase se entrelaça apenas com a anterior e a posterior, entretanto, vai contra a observação de Royce, representada na Figura 3-5. Segundo essa observação, não seria suficiente entrelaçar fases contíguas, pois pode-se necessitar retornar a outras fases anteriores a estas.

Em função disso, uma das evoluções mais importantes de Sashimi é o Modelo *Scrum* (Seção 4.1). *Scrum* consiste em levar a ideia de fases entrelaçadas ao extremo pela redução do

processo a uma única fase na qual todas as fases do Modelo Cascata são realizadas paralelamente por profissionais especializados trabalhando em equipe.

Porém, o modelo Sashimi tem o mérito de indicar que, por exemplo, a fase de análise de requisitos só estará completa depois que questões referentes ao *design* da arquitetura tenham sido consideradas, e assim por diante.

Este estilo de processo é adequado se o engenheiro de software avaliar que poderá obter ganhos de conhecimento sobre o sistema ao passar de uma fase para outra. O modelo também provê uma substancial redução na quantidade de documentação, pois as equipes das diferentes fases trabalharão juntas boa parte do tempo.

Entre os problemas do modelo está o fato de que fica mais difícil definir marcos (*millestones*), pois não fica muito claro quando exatamente uma fase termina. Além disso, a realização de atividades paralelas com este modelo pode levar a falhas de comunicação, aceitação de hipóteses erradas e ineficiência no trabalho.

3.4 Modelo V

O *Modelo V* (*V Model*) é uma variação do Modelo Cascata que prevê uma fase de validação e verificação para cada fase de construção. O Modelo V pode ser usado com projetos que tenham requisitos estáveis e dentro de um domínio conhecido (Lenz & Moeller, 2004)[21].

O Modelo V é sequencial e, como o Modelo Cascata, é dirigido por documentação. A Figura 3-7 mostra o diagrama de atividades do Modelo V, com as dependências de verificação indicadas entre as atividades.

[21] books.google.com.br/books?id=64XC28ZMhdEC&printsec=frontcover#v=onepage&q=&f=false

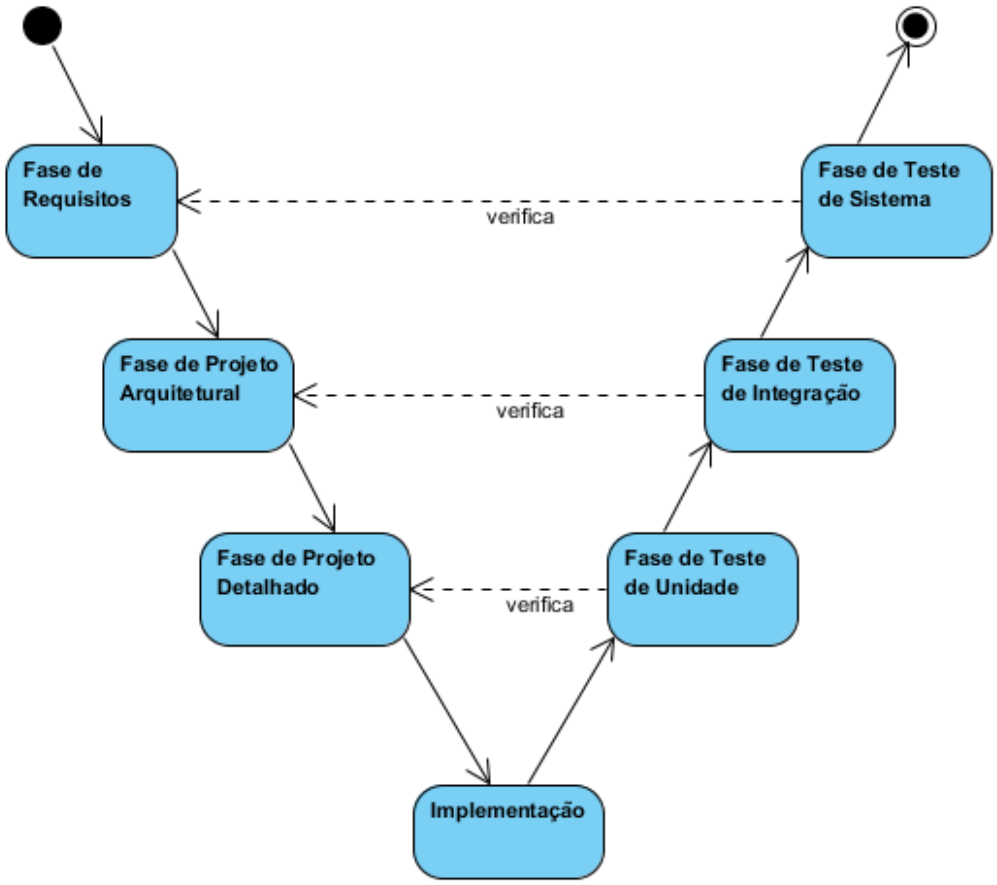

Figura 3-7: O Modelo V.

Na *fase de requisitos*, a equipe, juntamente com o cliente elicia e elabora o documento de requisitos, que é aprovado conjuntamente na forma de um contrato de desenvolvimento. Uma estimativa de esforço (Capítulo 7) deve ser produzida nesta fase também.

Na *fase de design arquitetural*, a equipe organiza os requisitos em unidades funcionais coesas, define como as diferentes partes arquiteturais do sistema vão se interconectar e colaborar. Nesta fase deve ser produzido um documento de especificação funcional do sistema, e as estimativas de esforço podem ser revistas.

Na *fase de design detalhado* a equipe vai aprofundar a descrição das partes do sistema e tomar decisões sobre como serão implementadas. Esta fase produz o documento de especificação detalhado dos componentes do software.

Na *fase de implementação* o software é implementado de acordo com a especificação detalhada.

A *fase de teste de unidade* tem como objetivo verificar se todas as unidades se comportam como especificado na fase de *design* detalhado. O projeto só segue em frente se todas as unidades passam em todos os testes.

A *fase de teste de integração* tem como objetivo verificar se o sistema se comporta conforme a especificação do *design* arquitetural.

Finalmente, a *fase de teste de sistema* verifica se o sistema satisfaz os requisitos especificados. Se o sistema passar nestes testes estará pronto para ser entregue ao cliente.

A principal característica do Modelo V está na sua ênfase nos testes e validações simétricos ao *design*. Essas etapas, porém, podem ser incorporadas a outros modelos de processo.

Os pontos negativos deste modelo são os mesmos do Modelo Cascata puro, entre eles o fato de que mudanças nos requisitos provocam muito retrabalho. Alem disso, em muitos casos, os documentos produzidos no lado esquerdo do V são ambíguos e imprecisos, impedindo ou dificultando os testes necessários representados no lado direito do V.

3.5 Modelo W

Spillner (2002)[22] apresenta uma variação ao Modelo V ainda mais voltada à área de teste: o *Modelo W*. A principal motivação para esta variação está na observação de que há uma divisão muito estrita entre as atividades construtivas do lado esquerdo do V e as atividades de teste no lado direito. Spillner propõe que o planejamento dos testes inicie durante a fase construtiva, mesmo sendo executado depois, e que o lado direito do V não seja considerado apenas como um conjunto de atividades de testes, mas também de reconstrução.

A figura resultante deste modelo assemelha-se graficamente a uma letra W (Figura 3-8), onde as atividades exteriores são construtivas e as interiores são atividades de teste.

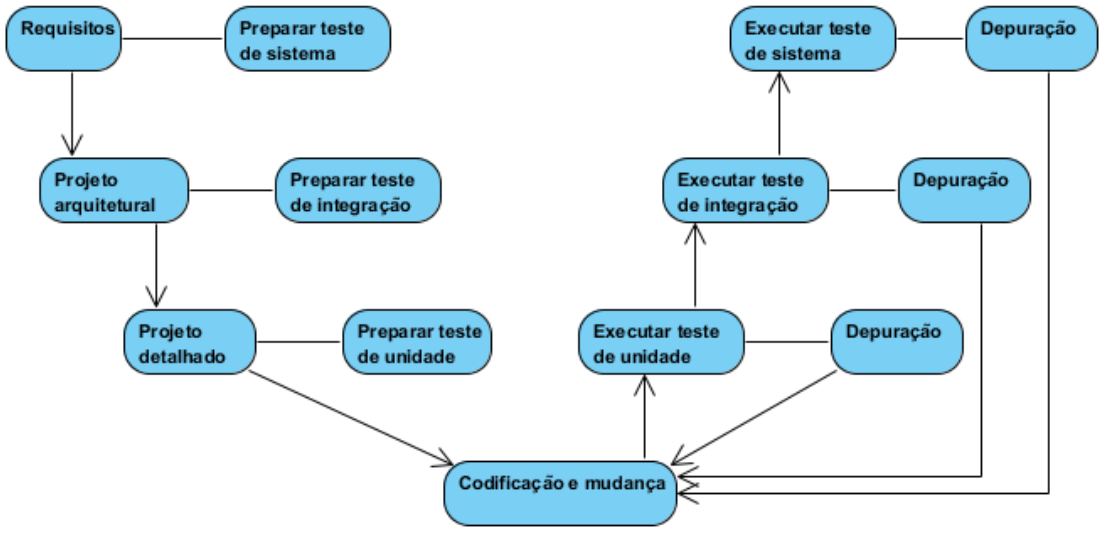

Figura 3-8: Modelo W[23,24].

Uma das questões colocadas já na fase de requisitos diz respeito ao fato de eles serem ou não testáveis. Apenas requisitos que possam ser testados são aceitáveis ao final da fase de requisitos. A mesma questão é colocada em relação à arquitetura na fase de *design* arquitetural. Arquiteturas simples devem ser fáceis de testar, caso contrário, talvez a arquitetura seja demasiadamente complexa e necessite ser refatorada. Na fase de projeto

[22] www.stickyminds.com/getfile.asp?ot=XML&id=3572&fn=XDD3572filelistfilename1%2Epdf

[23] Fonte: www.informatik.hs-bremen.de/spillner/WWW-Talks/Valencia.html. Consultado em: 18/08/2011

[24] As linhas sem setas ligam atividades que são feitas em paralelo.

detalhado, a mesma questão se coloca em relação às unidades. Unidades coesas são mais fáceis de testar.

Spillner argumenta que envolver os responsáveis pelos testes já nas fases iniciais de desenvolvimento faz com que mais erros sejam detectados mais cedo, e que *designs* excessivamente complexos sejam simplificados.

O Modelo W, assim, incorpora o teste nas atividades de desenvolvimento desde o seu início, e não apenas nas fases finais. Tal característica é também utilizada fortemente pelos métodos ágeis, que preconizam que o caso de teste deve ser produzido antes do código que será testado.

3.6 Cascata com Subprojetos

O Modelo Cascata *com Subprojetos* (*Waterfall with Subprojects*) permite que algumas fases do Modelo Cascata sejam executadas em paralelo. Após a fase de *design* da arquitetura, o projeto pode ser subdividido de forma que vários subsistemas sejam desenvolvidos em paralelo por equipes diferentes, ou pela mesma equipe em momentos diferentes.

A Figura 3-9 mostra um diagrama de atividades UML que representa o ciclo de vida Cascata com Subprojetos.

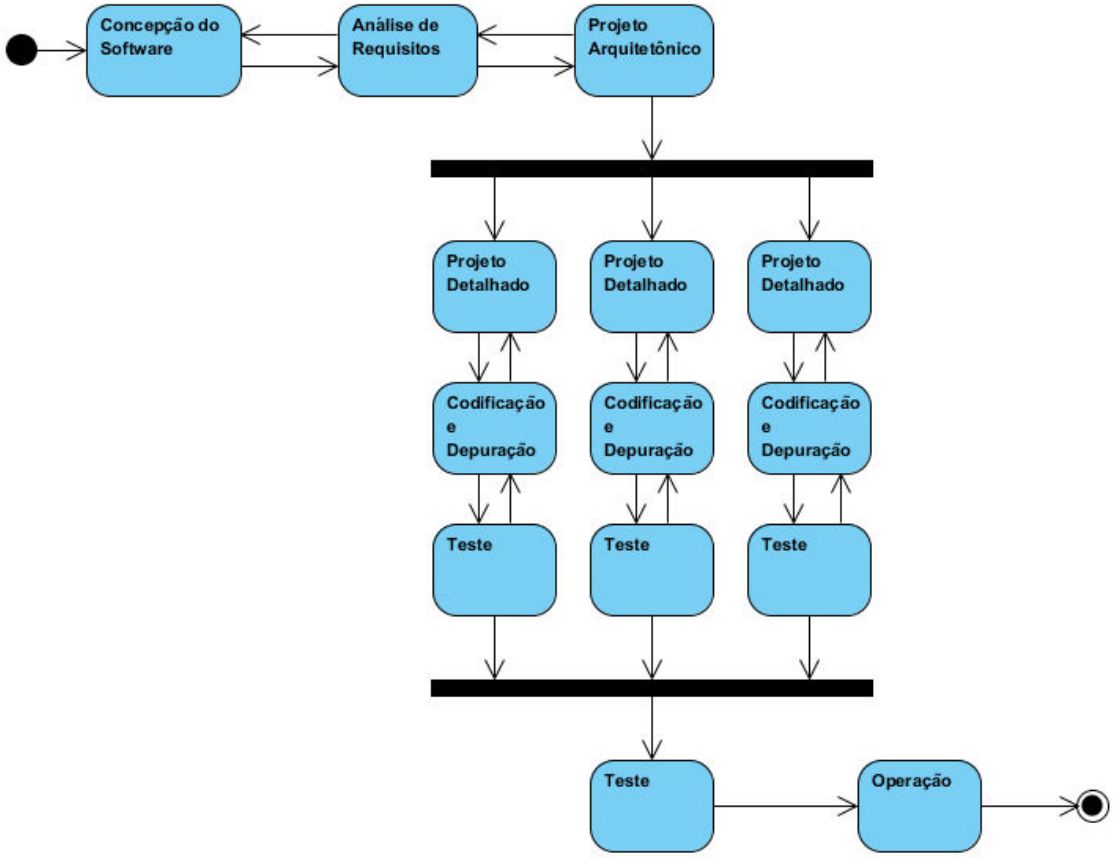

Figura 3-9: Modelo Cascata com Subprojetos.

Este modelo é bem mais razoável de se utilizar do que o Modelo Cascata puro, visto que o fato de quebrar o sistema em subsistemas menores permite que subprojetos mais rápidos e fáceis

de gerenciar sejam realizados. Esta técnica explora melhor as potencialidades de modularidade do projeto. E com ela o progresso é mais facilmente visível, porque se pode produzir várias entregas de partes funcionais do sistema à medida que ficam prontas.

A maior dificuldade com este modelo está na possibilidade de surgirem interdependências imprevistas entre os subsistemas. O *design* arquitetônico deve ser bem feito de forma a minimizar tais problemas. Além disso, este modelo exige maior capacidade de gerência para impedir que sejam criadas inconsistências entre os subsistemas.

Além disso, a integração final de todos os subsistemas pode ser um problema caso as interdependências não tenham sido adequadamente gerenciadas. Modelos de desenvolvimento ágeis preferem a integração contínua de pequenos pacotes de funcionalidade ao invés de grandes fases de integração no final do desenvolvimento.

Apesar disso, este modelo oficializou uma prática bastante razoável em desenvolvimento de sistemas que é "dividir para conquistar", pois é relativamente mais fácil conduzir um processo de desenvolvimento de vários subsistemas parcialmente dependentes do que de um grande sistema completo. Inclusive, o desenvolvimento de subprojetos sequencialmente pode ser entendido como um ancestral do desenvolvimento iterativo, uma das grandes características dos métodos ágeis e do Processo Unificado.

3.7 Cascata com Redução de Risco

O Modelo Cascata *com Redução de Risco* (*Waterfall with Risk Reduction*) procura resolver um dos principais problemas do *BDUF* que é a dificuldade em se ter uma boa definição dos requisitos do projeto nas fases iniciais. Este modelo basicamente acrescenta uma fase de redução de riscos antes do início do processo em cascata.

O objetivo do modelo é a redução do risco com os requisitos. A tônica é a utilização de técnicas que garantam que os requisitos serão os mais estáveis possíveis. Algumas técnicas utilizadas durante a fase espiral são:

a) *Desenvolver protótipos de interface com o usuário*. Neste caso, o modelo por vezes também é chamado de "*Cascata com Prototipação*". Esta técnica realiza uma das sugestões de Royce (fazer duas vezes). A elaboração de um protótipo, antes de comprometer recursos com o desenvolvimento de um sistema real, permite que questões relacionadas a requisitos e organização da arquitetura do sistema sejam analisadas e resolvidas.
b) *Desenvolver storyboards com o usuário*. A técnica de *storyboard* (Gomes, Medeiros, Alves, Caparica, Nibon, & Vasconcelos, 2007)[25] utiliza imagens para descrever situações de uso do sistema. É similar à técnica de cenários (Jacobson, 1995), mas os cenários usualmente são apenas resultado de dinâmica de grupo e documentados por texto.
c) *Conduzir vários ciclos de entrevistas com o usuário e cliente*. Ao invés de realizar apenas uma entrevista e obter os requisitos a partir dela, a técnica especifica que o cliente deva sempre receber um retorno sobre os requisitos levantados a cada

[25] www.cin.ufpe.br/~asg/publications/files/WDDS_Final.pdf

entrevista e que faça uma validação destes. Durante o processo de validação, uma nova entrevista poderá esclarecer aspectos ainda confusos ou detalhar aspectos ainda muito gerais.

d) *Filmar os usuários utilizando os sistemas antigos, sejam eles informatizados ou não*. A análise dos filmes poderá ajudar a compreender os fluxos de trabalho dos usuários. A vantagem dos filmes reais sobre *storyboards* e cenários está no fato de que normalmente a atuação nos filmes não pode ser supersimplificada ou falsificada pelos usuário. A desvantagem está no fato de que nem sempre surgem com frequência os fluxos de exceção ou variantes de um processo do usuário.

e) *Utilizar extensivamente quaisquer outras práticas de eliciação de requisitos*. A área de levantamento ou eliciação de requisitos é, por si só, uma área de pesquisa altamente frutífera dentro da engenharia de software. Quaisquer técnicas novas ou antigas que possam ajudar o analista a compreender e modelar adequadamente os requisitos é bem vinda.

A Figura 3-10 apresenta graficamente o Modelo Cascata com Redução de Risco

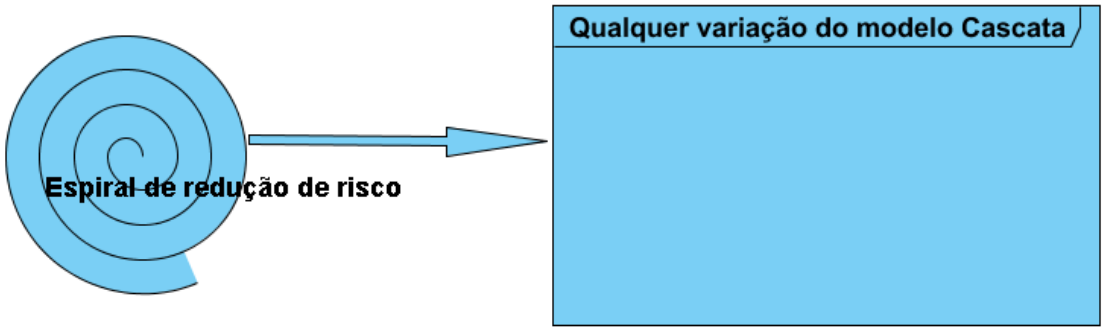

Figura 3-10: Modelo Cascata com Redução de Risco.

A espiral de redução de risco é uma fase extra que pode ser colocada à frente de qualquer variação do Modelo Cascata: Sashimi, Cascata com Subprojetos, Modelo V etc.

A espiral de redução de riscos não precisa ficar restrita aos requisitos do projeto, ela pode ser aplicada a quaisquer outros riscos identificados (Capítulo 7). Este tipo de modelo é adequado a projetos com um grande número de riscos importantes. A espiral de redução de riscos é uma forma de a equipe garantir alguma estabilidade ao projeto antes de comprometer um grande número de recursos na sua execução.

Como principal desvantagem pode-se citar a dificuldade de definir um cronograma preciso para a fase espiral, bem como as demais desvantagens do Modelo Cascata e suas variantes.

3.8 Modelo Espiral

O *Modelo Espiral* (*Spiral*) foi originalmente proposto por Boehm (1986) e é fortemente orientado à redução de riscos. A proposta de Boehm não foi a primeira a apresentar a ideia de ciclos iterativos, mas foi a primeira a realmente explicar porque as iterações eram necessárias. O projeto é dividido em subprojetos, cada um dos quais aborda um ou mais elementos de alto risco, até que todos os riscos identificados tenham sido tratados.

Pode-se dizer que, de certa forma, o Modelo Espiral também atende a uma das recomendações de Royce (1970), que propunha que o projeto fosse desenvolvido pelo menos duas vezes para que as lições aprendidas na primeira vez pudessem ser aproveitadas na segunda. O Modelo Espiral é uma forma de realizar estas iterações de forma mais organizada, iniciando com pequenos protótipos e avançando para projetos cada vez maiores.

O conceito de risco é definido de maneira abrangente e pode envolver desde requisitos mal compreendidos até problemas tecnológicos incluindo arquitetura, desempenho, dispositivos eletromecânicos, etc. (Capítulo 7).

Depois que os principais riscos foram mitigados, o Modelo Espiral prossegue de forma semelhante ao Modelo Cascata ou uma de suas variantes. A Figura 3-11 apresenta a definição clássica deste ciclo, a partir da qual seu nome foi escolhido.

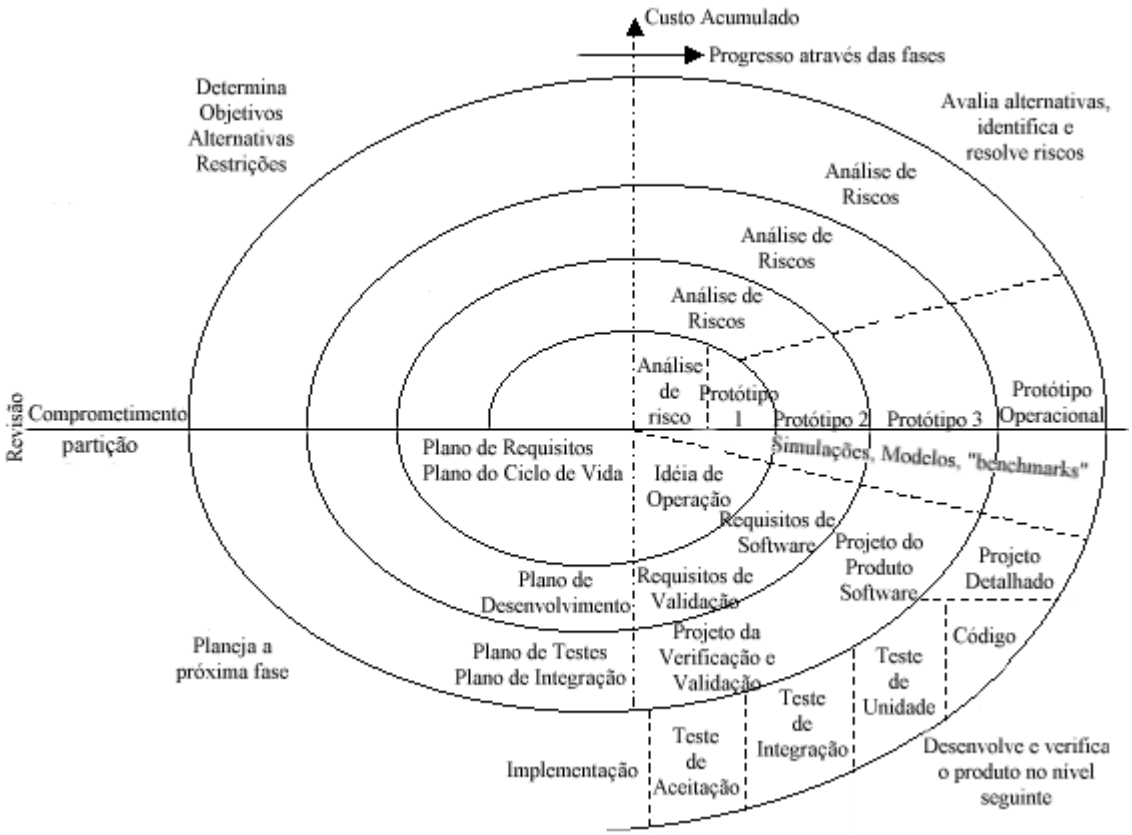

Figura 3-11: Ciclo de Vida Espiral[26].

A ideia deste ciclo é iniciar com mini projetos, abordando os principais riscos e expandir este projeto através da construção de protótipos, testes e replanejamento de forma a abarcar os principais riscos identificados. Após a equipe ter adquirido um conhecimento mais completo dos potenciais problemas com o sistema, ela passa a desenvolver um ciclo final semelhante ao Modelo Cascata.

No início do processo espera-se que a equipe explore os riscos, construa um plano para gerenciar os riscos e planeje e concorde com uma abordagem para o próximo ciclo. Cada volta no ciclo (ou iteração) faz o projeto avançar um nível em entendimento e mitigação de riscos.

[26] Fonte: www2.dem.inpe.br/ijar/CicoloVidaSoftPrado.html. Consultado em 06/03/2010.

Cada iteração do modelo envolve seis passos:

a) Determinar inicialmente os objetivos, alternativas e restrições relacionadas à iteração que vai se iniciar.
b) Identificar e resolver riscos relacionados à iteração em andamento.
c) Avaliar as alternativas disponíveis. Tipicamente podem ser utilizados protótipos nesta fase para verificar a viabilidade de diferentes alternativas.
d) Desenvolver os artefatos (possivelmente entregas) relacionados a esta iteração e certificar que estão corretos.
e) Planejar a próxima iteração.
f) Obter concordância em relação à abordagem para a próxima iteração, caso se resolva realizar uma.

Uma das vantagens do Modelo Espiral é que as primeiras iterações são as mais baratas do ponto de vista de investimento de tempo e recursos e ao mesmo tempo são as que resolvem os maiores problemas do projeto. A escolha dos riscos a serem mitigados é feita em função das necessidades de projeto. O método não preconiza este ou aquele risco. Então as atividades concretas nestas fases iniciais podem variar muito de projeto para projeto.

Porém, à medida que os custos aumentam, o risco diminui, o que é altamente desejável em projetos de envergadura. Se o projeto não puder ser concluído por razões técnicas, isso será descoberto cedo. Além disso, o modelo possui fases bem definidas, o que permite o acompanhamento objetivo do desenvolvimento.

O modelo não provê a equipe com indicações claras sobre a quantidade de trabalho esperada a cada ciclo, o que pode tornar o tempo de desenvolvimento nas primeiras fases bastante imprevisível. Além disso, o movimento complexo entre as diferentes fases ao longo das várias iterações da espiral exige uma gerência complexa e eficiente.

Esse ciclo de vida se adéqua bem a projetos complexos com alto risco e requisitos pouco conhecidos como, por exemplo, projetos de pesquisa e desenvolvimento (P&D). Não se recomenda este ciclo de vida para projetos de pequeno e médio porte ou com requisitos bem conhecidos.

Segundo Schell (2008), o Modelo Espiral é bastante adequado para a área de jogos eletrônicos, onde os requisitos usualmente não são conhecidos sem que protótipos tenham sido testados, e onde os riscos se apresentam altos tanto do ponto de vista tecnológico quanto do ponto de vista da usabilidade do sistema.

3.9 Prototipação Evolucionária

Usualmente distinguem-se duas abordagens de prototipação:

a) Prototipação *throw-away*[27], que consiste na construção de protótipos que são usados unicamente para estudar aspectos do sistema, entender melhor seus requisitos e reduzir riscos, mas o protótipo, depois de cumprir essas finalidades é descartado. (Crinnion, 1991).

[27] "Descartável".

b) Prototipação *cornerstone*[28], que consiste na construção de protótipos que também são usados para estudar aspectos do sistema, entender melhor seus requisitos e reduzir riscos, mas o protótipo será parte do sistema final, ou seja, ele vai evoluindo até se tornar um sistema entregável (Budde, Krautz, Kuhlenkamp, & Zullighoven, 1992).

O modelo de *Prototipação Evolucionária* (Brooks, 1975) ou *Evolutionary Prototyping* baseia-se na técnica de prototipação cornerstone. Essa técnica exige um planejamento de protótipos muito mais cuidadoso do que a técnica *throw-away*, porque defeitos nos protótipos iniciais serão propagados ao sistema final se não forem consertados.

O Modelo Prototipação Evolucionária sugere que a equipe de desenvolvimento trabalhe junto ao cliente os aspectos mais visíveis do sistema, na forma de protótipos (usualmente de interface), até que o produto seja aceitável. A Figura 3-12 apresenta o diagrama de atividades UML simplificado para este modelo.

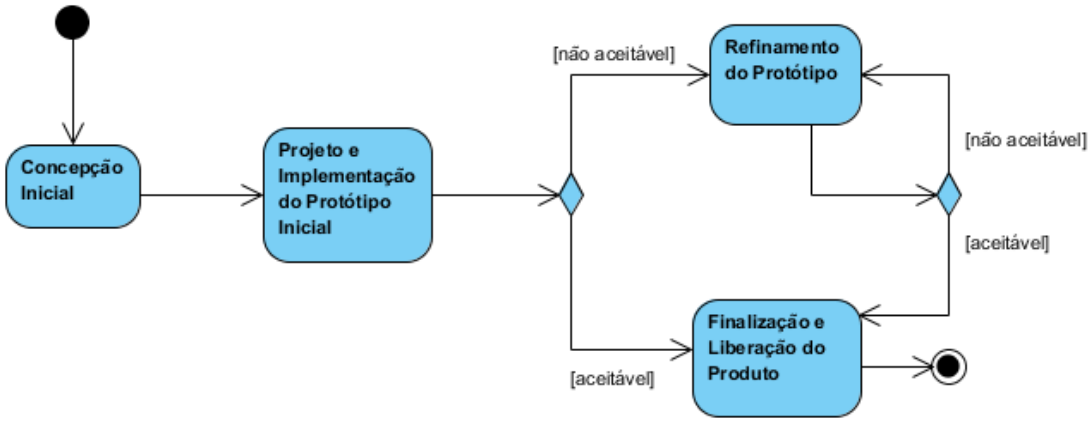

Figura 3-12: Modelo Prototipação Evolucionária.

Este modelo pode ser particularmente interessante se for difícil fazer o cliente comunicar os requisitos. Neste caso, um protótipo do sistema será uma ferramenta mais fácil para o analista se comunicar com o cliente e chegar a um acordo sobre o que deve ser desenvolvido.

Este modelo também pode ser interessante quando tanto a equipe quanto o cliente não conhecem bem os requisitos do sistema. Pode ser difícil elaborar requisitos quando não se sabe exatamente o que é necessário, sem ver o software funcionando e sendo testado.

Entretanto, o modelo não é muito bom em relação à previsão de tempo para desenvolvimento e também em relação à gerência do projeto, já que é difícil avaliar quando cada fase foi efetivamente realizada. Pode até acabar acontecendo que o projeto seguindo este modelo regrida para Codificar e Consertar. Para evitar isso, deve-se garantir que o processo efetivamente obtenha uma concepção real do sistema, com requisitos definidos da melhor forma possível, e com um projeto realista antes de iniciar a codificação propriamente dita.

[28] "Pedra fundamental".

3.10 Entregas em Estágios

O Modelo Entregas *em Estágios* (*Staged Deliveries* ou *Implementação Incremental*) é uma variação mais bem estruturada do modelo Prototipação Evolucionária embora também seja considerado uma variação do Modelo Cascata. Ao contrário da Prototipação Evolucionária, o Modelo Entregas em Estágios prevê que a cada ciclo a equipe planeje e saiba exatamente o que vai entregar ao cliente.

A abordagem é interessante porque haverá vários pontos de entrega e o cliente poderá acompanhar mais diretamente a evolução do sistema. Não existe, portanto, o problema do Modelo Cascata, onde o sistema só é entregue quando está totalmente acabado.

A Figura 3-13 mostra o diagrama de atividades UML para o modelo. Assim que o *design* arquitetural estiver completo é possível iniciar a implementação e entrega de partes do produto final.

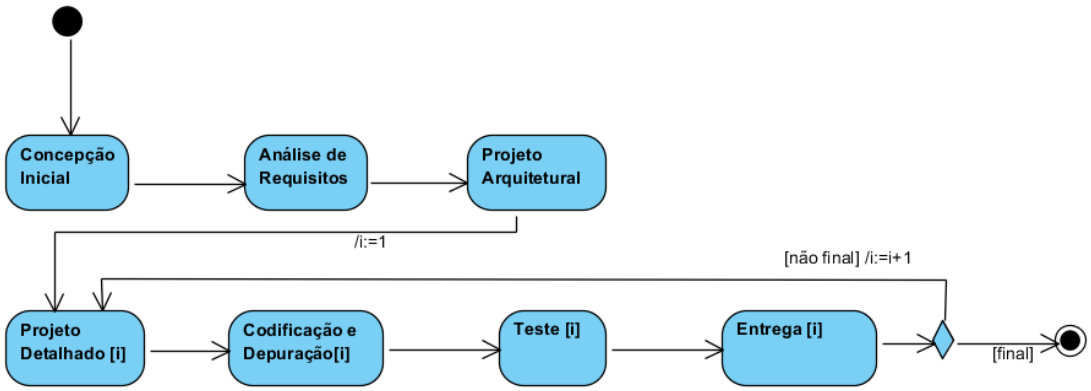

Figura 3-13: Modelo Entregas em Estágios.

Uma das principais vantagens deste modelo (Ellis, 2010) é o fato de colocar funcionalidades úteis nas mãos do cliente antes de completar todo o projeto. Se os estágios forem planejados cuidadosamente, funcionalidades importantes estarão disponíveis muito mais cedo do que com outros ciclos de vida. Além disso, este modelo provê entregas mais cedo e de forma contínua, o que pode aliviar um pouco a pressão de cronograma colocada na equipe.

Entretanto, este modelo não funcionará se as etapas não forem cuidadosamente planejadas nos seguintes níveis:

a) *Técnico*: as dependências técnicas entre os diferentes módulos entregáveis devem ser cuidadosamente verificadas. Se um módulo tem dependências com outro, o segundo deve ser entregue antes deste.
b) *Gerencial*: deve-se procurar garantir que os módulos sejam efetivamente significativos para o cliente. Será menos útil entregar funcionalidades parciais que não possam produzir nenhum trabalho consistente.

Para esta técnica funcionar é necessário que os requisitos sejam bem compreendidos para que o planejamento possa ser efetivo.

Essa técnica também pode ser considerada uma precursora dos modelos iterativos como UP e métodos ágeis.

3.11 Modelo Orientado a Cronograma

O *Modelo Orientado a Cronograma* (*Design to Schedule*) é similar ao Modelo Entregas em Estágios, exceto pelo fato de que ao contrário deste último, não se sabe *a priori* quais funcionalidades serão entregues a cada ciclo.

O Modelo Orientado a Cronograma prevê que os ciclos terminarão em determinada data e as funcionalidades implementadas até ali serão entregues. É importante priorizar, portanto, as funcionalidades, de forma que as mais importantes sejam abordadas e entregues primeiro, enquanto as menos importantes ficam para depois. A Figura 3-14 apresenta o modelo.

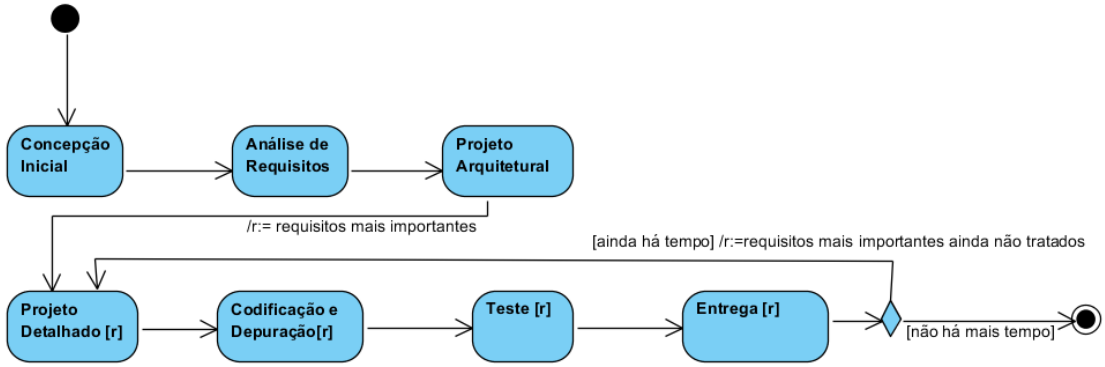

Figura 3-14: Modelo Orientado a Cronograma.

Na figura, a cada iteração do ciclo de desenvolvimento desenvolve-se um conjunto de requisitos tomando sempre primeiro os mais importantes que ainda não tenham sido abordados. Encerra-se o projeto quando o tempo limite for atingido, ou quando todos os requisitos tenham sido atendidos.

Espera-se que mesmo que não tenha sido possível atender a todos os requisitos pelo menos os que ficaram de fora tenham sido os menos importantes.

Este modelo é uma boa estratégia para garantir que haverá algum produto disponível em uma determinada data, se isso for absolutamente imprescindível. Então é um modelo apropriado quando existe uma data limite para entrega que é intransferível.

Porém, se a equipe é altamente confiante na sua capacidade de previsão de esforço (cumpre prazos constantemente), essa abordagem não é recomendada.

Uma das desvantagens deste modelo é que, caso nem todas as funcionalidades sejam entregues a equipe terá perdido tempo fazendo a análise delas nas etapas iniciais.

Em relação ao Processo Unificado e métodos ágeis, esse modelo difere na forma como concebe os ciclos. No modelo orientado a cronograma a duração dos ciclos não é estabelecida a priori. Já nos modelos mais modernos estabelece-se uma duração fixa para os ciclos e tenta-se implementar um conjunto de funcionalidades dentro destes prazos fixos estabelecidos. Desta forma é mais fácil verificar se o projeto está andando bem ou se está atrasando. Mas

para que isso funcione é necessário que se seja capaz de estimar o esforço necessário para desenvolver o projeto (Capítulo 7).

3.12 Entrega Evolucionária

O Modelo Entrega *Evolucionária* (*Evolutionary Delivery*) é um meio termo entre a Prototipação Evolucionária (Seção 3.9) e a Entrega em Estágios (Seção 3.10). Neste modelo a equipe também desenvolve uma versão do produto, mostra ao cliente, e desenvolve novas versões baseadas no *feedback* do cliente.

O quanto ela se aproxima da Prototipação Evolucionária ou da Entrega em Estágios depende do grau em que se pretende acomodar as modificações solicitadas:

a) Se a ideia é acomodar todos ou a grande maioria das modificações, então a abordagem tende mais para Prototipação Evolucionária.
b) Se, entretanto, as entregas continuarão sendo planejadas de acordo com o previsto e as modificações acomodadas aos poucos nas entregas, então a abordagem se parece mais com Entrega em Estágios.

Este modelo permite, então, ajustes que lhe dão um pouco da flexibilidade do modelo de Prototipação Evolucionária ao mesmo tempo em que se tem o benefício do planejamento da Entrega em Estágios.

As diferenças entre este modelo e os anteriores então está mais na ênfase do que nas atividades relacionadas. No modelo de Prototipação Evolucionária a ênfase está nos aspectos visíveis do sistema. Na Entrega Evolucionária, porém, a ênfase está nas funcionalidades mais críticas do sistema.

O Processo Unificado implementa, de certa maneira, essa flexibilidade ao permitir que o gerente de projeto escolha, ao planejar cada ciclo iterativo, se vai abordar um *caso de uso* (equivale a um conjunto de requisitos), um *risco* ou uma *requisição de modificação*, baseando-se nas suas prioridades.

3.13 Modelos Orientado a Ferramentas

Chama-se de *Modelo Orientado a Ferramentas* (*Design to Tools*) qualquer modelo baseado no uso intensivo de ferramentas de prototipação e geração de código, que permitem a rápida produção de sistemas executáveis a partir de especificações em alto nível, como por exemplo, *jCompany* (Alvim, 2008) ou WebRatio (Ceri, Fraternali, Bongio, Brambilla, Comai, & Matera, 2003). É uma abordagem extremamente rápida de desenvolvimento e prototipação, mas é limitada pelas funcionalidades oferecidas pelas ferramentas específicas.

Assim, requisitos só são atendidos se a ferramenta de produção permite atender à funcionalidade requerida. Se as ferramentas forem cuidadosamente escolhidas, entretanto, pode-se conseguir implementar uma grande parte dos requisitos rapidamente.

Esta abordagem pode ser combinada com outros modelos de processo. A prototipação necessária no Modelo Espiral, por exemplo, pode ser feita utilizando-se ferramentas de geração de código para produzir protótipos rapidamente.

3.14 Linhas de Produto de Software

Uma linha de produto de software (*Software Product Line* – *SPL*) consiste, segundo o SEI[29] de um conjunto de sistemas de software que compartilham um conjunto de características comuns gerenciadas de forma a satisfazer as necessidades específicas de um segmento de mercado ou missão particular, e que foram desenvolvidos a partir de um núcleo comum de forma sistemática.

Segundo Northrop (2008)[30], *SPL*s podem ser vistas como uma evolução das estratégias de reutilização na indústria de software. Essas estratégias seriam caracterizadas pela reutilização de *sub-rotinas* nos anos 1960, *módulos* nos anos 1970, *objetos* nos anos 1980, *componentes* nos anos 1990 e *serviços* nos anos 2000. Porém, enquanto as abordagens citadas são meramente técnicas, o reuso obtido com *SPL* consiste em uma abordagem estratégica para a indústria de software. Desta forma, o reuso deixa de ser realizado de forma imprevisível e *ad-hoc* para ser incorporado sistematicamente nos processos produtivos.

Segundo Weiss e Lay (1999), o uso de uma tecnologia como *SPL* só se torna financeiramente praticável a partir de certo ponto na escala do número e produtos de uma família. A técnica tem a ver com o desenvolvimento de vários produtos diferenciados, mas com características comuns a partir de um núcleo comum. Assim, *SPL* não se aplica em situações onde um único produto é desenvolvido ou onde vários produtos não relacionados são desenvolvidos.

SPL trata de famílias de produtos como sistemas relacionados a uma mesma área (software para telefones celulares, por exemplo), ou personalizações planejadas. Então, se o número de produtos for pequeno, o investimento para criar e gerenciar uma *SPL* não se justifica. Mas para famílias de produtos a partir de certo tamanho a técnica pode ser a melhor escolha, pois permite o gerenciamento de versões e a otimização do processo produtivo (Figura 3-15).

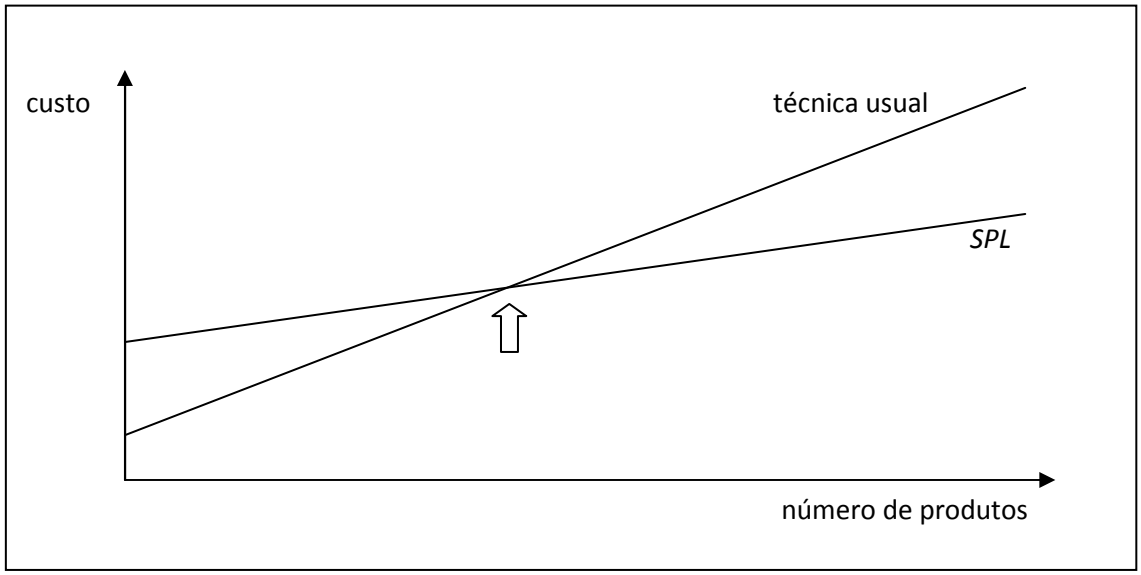

Figura 3-15: Custo/benefício de *SPL*[31].

[29] www.sei.cmu.edu/productlines/ (Consultado em: 28/03/2010)
[30] www.sei.cmu.edu/library/assets/*SPL*-essentials.pdf
[31] Fonte: Weiss e Lai (1999).

Segundo Rombach (2005), o investimento em uma *SPL* usualmente começa a compensar a partir do terceiro produto gerado.

A implantação de uma *SPL* exige mudanças de gerência, forma de desenvolvimento de software, organização estrutural e de pessoal, abordagem de negócio da empresa e principalmente na concepção arquitetônica dos produtos. A arquitetura é fundamental porque sua escolha é crítica para obter as funcionalidades corretas com as propriedades desejadas. Uma arquitetura ruim poderá ser causa de fracasso não só em iniciativas de *SPL*, mas em qualquer projeto de software.

Northrop (2008) destaca dentre todas as novas disciplinas relacionadas a *SPL* três atividades essenciais (Figura 3-16):

a) Desenvolvimento de um núcleo de ativos de produtos (*core asset*).
b) Desenvolvimento de produtos.
c) Gerência.

Não há uma ordem predefinida para a execução destas atividades. Muitas vezes o produto é produzido a partir do núcleo de ativos, outras vezes o núcleo de ativos é gerado a partir de produtos já existentes, ou ainda estes podem ser desenvolvidos em paralelo.

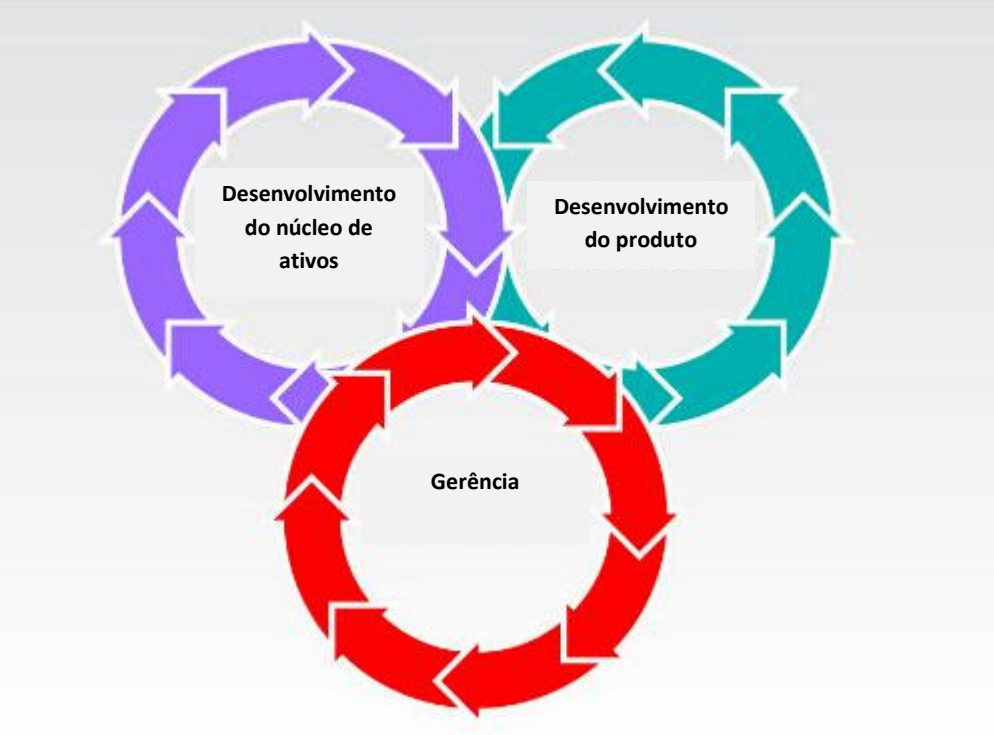

Figura 3-16: As três atividades essenciais para *SPL*[32].

Como se vê na figura não há uma dependência ou precedência entre as atividades. Elas estão sempre em andamento e estão mutuamente interligadas.

[32] Fonte: www.sei.cmu.edu/productlines/frame_report/PL.essential.act.htm

3.14.1 Desenvolvimento do Núcleo de Ativos

O objetivo da atividade de *desenvolvimento do núcleo de ativos* é estabelecer um potencial para produção de produtos de software. Esse núcleo de ativos consiste no conjunto de elementos que poderão ser reusados na SPL. Esses elementos nem sempre são reusados da mesma forma, assim, *pontos de variação* poderão precisar ser identificados. Esses pontos de variação são características dos elementos que podem variar de um reuso para outro. Usualmente, os pontos de variação são caracterizados por um conjunto de restrições às quais suas eventuais instanciações devem se conformar. Além disso, esses elementos não se apresentam apenas como módulos de software a serem reusados, eles também incluirão um conjunto de instruções sobre como proceder ao seu reuso.

A Figura 3-17 apresenta esquematicamente a atividade de desenvolvimento do núcleo de ativos.

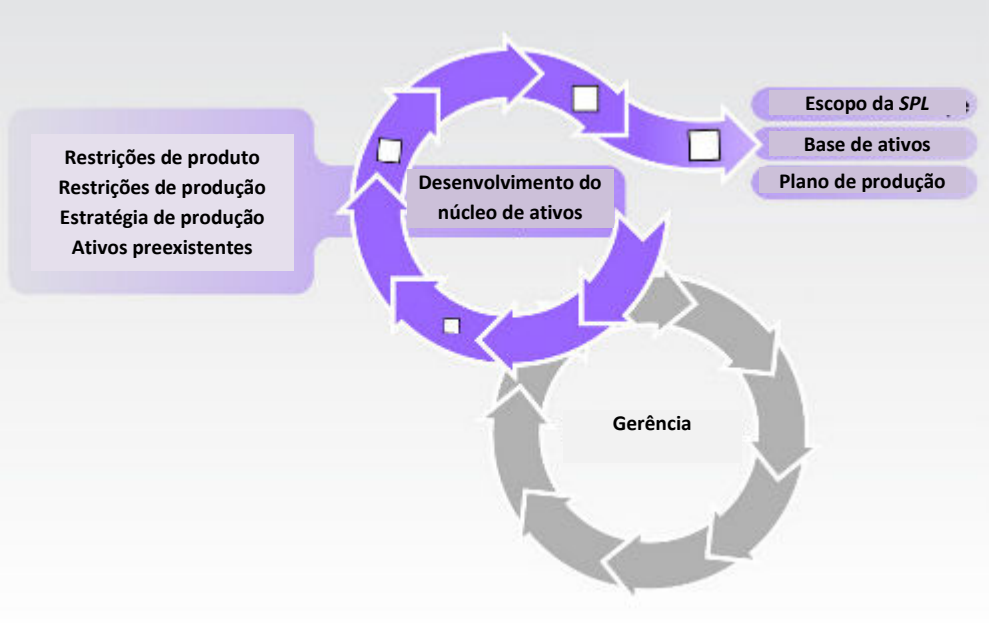

Figura 3-17: A atividade de desenvolvimento do núcleo de ativos[33].

Esta atividade, como as outras duas, é iterativa. Ela ocorre sob a influência de certos fatores ambientais que são definidos assim[34]:

a) *Restrições de produto*. Quais são as coisas em comum e as especificidades dos produtos que constituem a *SPL*? Quais características de comportamento elas provêm? Quais são as expectativas de mercado e de tecnologia? Quais padrões se aplicam? Quais são suas limitações de performance? Quais limitações físicas, como por exemplo, interfaces com sistemas externos, elas devem observar? Quais são os requisitos de qualidade (como segurança e disponibilidade)?
b) *Restrições de produção*. Qual o prazo esperado para disponibilização de um novo produto? Quais ferramentas de produtividade serão disponibilizadas? Quais padrões de processo de desenvolvimento serão adotados?

[33] Fonte: www.sei.cmu.edu/productlines/frame_report/coreADA.htm
[34] www.sei.cmu.edu/productlines/ (Consultado em: 28/03/2010)

c) *Estratégia de produção*. A *SPL* será construída de forma proativa, reativa ou como uma combinação das duas? Como será a estratégia de preço? Os componentes chave serão produzidos ou comprados?
d) *Ativos preexistentes*. Quais ativos da organização podem ser usados na *SPL*? Existem bibliotecas, *frameworks*, componentes, web-services disponíveis tanto produzidos internamente quanto obtidos fora?

As saídas da atividade de desenvolvimento de núcleo de ativos podem ser definidas assim:

a) *Escopo da SPL*. É uma descrição dos produtos que a *SPL* é capaz de produzir. Esta descrição pode ser uma simples lista, ou uma estrutura de similaridades e diferenças. Se o escopo for muito grande os produtos vão variar demais, haverá poucos pontos em comum e pouca economia no processo produtivo.
b) *Base de ativos*. É a base para a produção da *SPL*. Nem todo ativo é necessariamente usado em todos os produtos. Entretanto, todos eles devem ter uma quantidade de usos e reusos que justifiquem seu gerenciamento na base de ativos. Cada ativo deve ter um processo anexado que especifica como ele deve ser usado na produção de novos produtos.
c) *Plano de produção*. Um plano de produção prescreve como os produtos serão produzidos a partir dos ativos. Ele inclui o processo de produção

3.14.2 Desenvolvimento do Produto

O desenvolvimento do produto depende de três insumos: o escopo da *SPL*, o núcleo de ativos e o plano de produção juntamente com a descrição do produto individual (Figura 3-18).

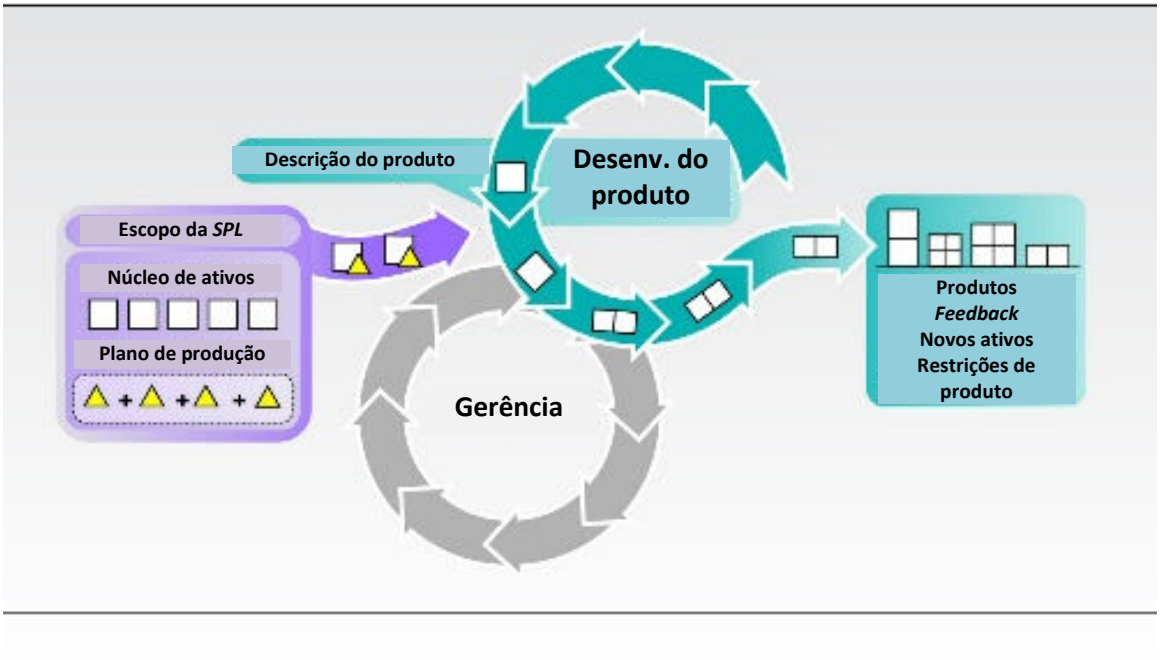

Figura 3-18: Atividade de desenvolvimento do produto em uma SPL[35].

[35] Fonte: www.sei.cmu.edu/productlines/frame_report/productDA.htm

A atividade de desenvolvimento de produto é iterativa e integrada com as outras duas atividades (desenvolvimento do núcleo de ativos e gerência). As entradas para esta atividade são:

a) A descrição de um produto em particular, frequentemente expressa como um delta ou variação a partir de alguma descrição genérica de produto contida no escopo da *SPL*.
b) O escopo da *SPL* que indica se é viável incluir um novo produto na linha.
c) O núcleo de ativos a partir do qual o produto é construído.
d) O plano de produção, que detalha como o núcleo de ativos pode ser usado para produzir o produto.

Já as saídas previstas para a ativade incluem:

a) O produto em si, podendo consistir de um ou mais sistemas personalizados.
b) *Feedback* para o processo produtivo, o que permite capitalizar lições aprendidas e melhorar o processo.
c) Novos ativos, que podem ser gerados ou identificados durante a produção de um produto específico.
d) Novas restrições de produto, semelhantes às já definidas na Seção 3.14.1.

3.14.3 Gerência

A atividade de gerência (Capítulo 9), como em qualquer processo de produção de software, desempenha um papel crítico nas *SPL*. Atividades e recursos devem ser atribuídos e então coordenados e supervisionados. A gerência tanto em nível de projeto quanto em nível organizacional deve estar comprometida com a *SPL*.

A gerência organizacional identifica a estratégia de negócio e oportunidades com a *SPL*. A gerência organizacional pode ser considerada como a responsável pelo sucesso ou fracasso de um projeto.

Já a gerência técnica deve acompanhar as atividades de desenvolvimento, verificando se os padrões são seguidos, se as atividades são executadas e se o processo pode ser melhorado.

3.14.4 Juntando Tudo

As três disciplinas (desenvolvimento do núcleo e ativos, desenvolvimento do produto e gerência) são organizadas de forma dinâmica, e mesmo sua adoção pode se dar de diferentes formas em diferentes empresas. Algumas empresas iniciam pela produção do núcleo de ativos (abordagem *proativa*), outras tomam produtos existentes e identificam as partes em comum, para produzir o núcleo (abordagem *reativa*). Northrop (2004)[36] apresenta um conjunto de orientações para empresas que desejam adotar *SPL*s.

As duas abordagens mencionadas acima podem ser atacadas incrementalmente, isto é, pode-se iniciar com um núcleo de ativos pequeno e gradativamente ir aumentando tanto o núcleo de ativos quanto o número de produtos.

[36] www.sei.cmu.edu/library/abstracts/reports/04tr022.cfm

Uma excelente leitura para aprofundar aspectos práticos de SPL está disponível no site do SEI[37]. Nas descrições das práticas de engenharia, de gerência e organizacionais são destacadas as diferenças fundamentais entre os processos usuais e os processos envolvendo SPLs.

[37] www.sei.cmu.edu/productlines/frame_report/productLPAs.htm (consultado em 18/02/2012).

