Capítulo
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 *antimodelo* por excelência, que é *Codificar e Consertar* (Seção 3.1), que consiste 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 o 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, em que 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 em que 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 já prontas podem ser entregues antes de o projeto ser finalizado e que isso possa ser planejado. O *Modelo Orientado a Cronograma* (Seção 3.11) indica que se podem 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 em que se pode decidir entre seguir o planejamento inicial ao longo das iterações ou incorporar mudanças oriundas do *feedback* do cliente. Os *modelos orientados a ferramentas* (Seção 3.13) são modelos específicos de ferramentas CASE (*Computer Aided Software Engineering*) e geradores de código aplicados com essas ferramentas. Por fim, o capítulo apresenta as *Linhas de Produto de Software* (Seção 3.14), que são uma abordagem moderna para a reusabilidade planejada em nível organizacional.

Para que o desenvolvimento de sistemas deixe de ser artesanal e aconteça de forma mais previsível e com maior qualidade, é necessário 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 de processo: os *prescritivos*, abordados neste capítulo, e os *ágeis*, apresentados no Capítulo 4.

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

Cada um dos ciclos de vida, sejam aplicados ou não no desenvolvimento de software atualmente, trouxe uma característica própria. As 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:

- *Codificar e consertar*: é considerado o modelo *ad-hoc*, ou seja, aquele que acaba sendo usado quando não se utiliza conscientemente nenhum modelo. Sendo assim, ele também não traz nenhuma contribuição, sendo considerado o "marco zero" dos modelos ou o antimodelo.
- *Cascata*: introduz a noção de que o desenvolvimento de software ocorre em fases bem definidas e de que é necessário produzir não apenas um código executável, mas também documentos que ajudem a visualizar o sistema de forma mais abstrata do que o código.
- *Sashimi*: derivado do Modelo Cascata, introduz a noção de que as fases do processo de desenvolvimento não são estanques, mas que em determinado momento pode-se estar trabalhando simultaneamente em mais de uma dessas fases.
- *Cascata com subprojetos*: introduz 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, em diferentes momentos, o que se assemelha ao desenvolvimento iterativo em ciclos. Caso o desenvolvimento dos subprojetos ocorra em paralelo, a integração entre os subsistemas desenvolvidos ocorrerá usualmente só no final.
- *Cascata com redução de risco* e *espiral*: introduzem a noção de que riscos são fatores determinantes para o sucesso de um projeto de software e, portanto, devem ser os primeiros aspectos analisados. Assim, uma espiral de desenvolvimento se inicia onde, a cada ciclo, um ou mais riscos são resolvidos ou minimizados. Apenas ao final desses ciclos de redução de risco é que o desenvolvimento propriamente dito se inicia.
- *Modelo V* e *modelo W*: enfatizam a importância do teste no desenvolvimento de software e indicam que essa deve ser uma preocupação constante, e não apenas uma etapa colocada ao final do processo de desenvolvimento.
- *Modelo orientado a cronograma*: introduz a noção de que se pode trabalhar prioritariamente com as funcionalidades mais importantes, deixando as menos importantes para o final. Assim, se houver atrasos e o prazo for rígido, pelo menos as funcionalidades mais importantes serão entregues no prazo.
- *Entrega em estágios*: introduz a noção de que é possível planejar e entregar partes prontas do sistema antes do final do projeto.
- *Prototipação evolucionária*: propõe o uso de protótipos para ajudar na compreensão da arquitetura, da interface e dos requisitos do sistema, o que é extremamente útil quando não é possível conhecer bem esses aspectos *a priori*.
- *Entrega evolucionária*: combina a prototipação evolucionária com a entrega em estágios, mostrando que é possível fazer um planejamento adaptativo em que, a cada nova iteração, o gerente de projeto decide se vai acomodar as requisições de mudança que surgiram ao longo do projeto ou manter-se fiel ao planejamento inicial.
- *Modelos orientados a ferramentas*: é uma família de modelos cuja única semelhança costuma consistir no fato de que eles são indicados para uso com ferramentas específicas.
- *Linhas de produto de software*: é um tipo de modelo que se aplica somente se a organização 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.
- *Modelos ágeis*: é uma família de modelos cujo foco está nos fatores humanos, e não na descrição de tarefas, o que os diferencia dos modelos prescritivos. Tais modelos tem se tornado dominantes na indústria a partir dos anos 2000.
- *Processo unificado*: 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 mais antiga e conhecida, RUP, as ágeis DAD 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 o que for mais adequado a sua equipe e ao projeto que ele vai desenvolver. Se 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.

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

Porém, não existe a necessidade de se optar por um modelo em detrimento de outros. Muitas vezes é possível combinar características interessantes de modelos diferentes gerando assim um processo adaptado exatamente às necessidades de um projeto específico.

Projetos diferentes têm necessidades diferentes. Assim, não há um modelo que seja sempre melhor do que os outros. Nem mesmo o processo unificado se adapta bem a sistemas cujos requisitos não sejam baseados em casos de uso, tais como o software científico, os jogos e os 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:

- *Quão bem os analistas e o cliente podem conhecer os requisitos do sistema?* O entendimento sobre o sistema poderá mudar à medida que o desenvolvimento avançar? Se os requisitos são estáveis, pode-se trabalhar com modelos mais previsíveis, como o Modelo Cascata ou os Modelos V e W. Requisitos instáveis ou mal compreendidos, porém, exigem ciclos de redução de risco e modelos baseados em prototipação, espiral ou ainda métodos ágeis.
- *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 as variações do Modelo Cascata, especialmente o Cascata com Subprojetos, mas arquiteturas mal compreendidas precisam de protótipos e de um desenvolvimento iterativo para reduzir o risco. O Processo Unificado, em especial, oferece uma fase, a Elaboração, especificamente para resolver problemas e estabilizar a arquitetura do sistema; conforme o caso, a duração da fase poderá ser mais curta ou mais longa.
- *Qual o grau de confiabilidade necessário em relação ao cronograma?* O Modelo Orientado a Cronograma, o Processo Unificado e os 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 ao cronograma.
- *Quanto planejamento é efetivamente necessário?* Os modelos prescritivos costumam privilegiar o planejamento com antecedência. Já os modelos ágeis preferem um planejamento menos detalhado e a adaptação às condições do projeto à medida que ele vai evoluindo. O Processo Unificado faz um planejamento genérico a longo prazo e um planejamento detalhado para o próximo ciclo iterativo que vai iniciar.
- *Qual é o grau de risco que esse 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.
- *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. Métodos ágeis também valorizam muito as entregas no prazo, priorizando elementos do escopo quando for o caso e nunca deixando a desejar na qualidade.
- *Será necessário entregar partes do sistema funcionando antes de terminar o projeto?* Alguns ciclos de vida, como Cascata com Subprojetos, preveem a integração do software apenas no final do desenvolvimento. Já os Métodos Ágeis e o Processo Unificado sugerem que se pratique a integração contínua, o que pode viabilizar a entrega de partes do sistema ao longo do processo de desenvolvimento.

- *Qual é o grau de treinamento e adaptação necessário para a equipe poder utilizar o ciclo de vida que parece mais adequado ao projeto?* Nenhum ciclo de vida, exceto Codificar e Consertar, é trivial. Todos exigem certa dose de preparação da equipe. Porém, os prescritivos, por definirem as tarefas detalhadamente, podem ser mais adequados a equipes inexperientes, enquanto os métodos ágeis, focados em valores humanos, usualmente necessitam de desenvolvedores mais experientes.
- *Será desenvolvido um único sistema ou uma família de sistemas semelhantes?* Caso mais de dois sistemas semelhantes sejam desenvolvidos, pode ser o caso de investir em uma linha de produtos de software, que permite lidar com as partes em comum e as diferenças entre os sistemas, aplicando o reuso de forma planejada e institucionalizada.
- *Qual o tamanho do projeto?* Projetos que possam ser realizados por equipes pequenas (de até oito ou dez desenvolvedores) adequam-se melhor aos métodos ágeis originais, enquanto os projetos de grande porte precisam de processos mais formais, como *RUP-SE*. A família *Crystal*, por outro lado, apresenta modelos que se adaptam ao tamanho do projeto. Há também opções que escalam métodos ágeis para equipes grandes, como *LeSS*, por exemplo.

Em geral, 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, porém executá-lo e gerenciá-lo mal. Porém, é importante que sempre se considere o tamanho e complexidade do projeto para que um modelo na medida certa seja utilizado.

3.1 Codificar e Consertar

O *Modelo Codificar e Consertar* (*Code and Fix*) tem uma filosofia muito simples, mostrada na Figura 3.1.

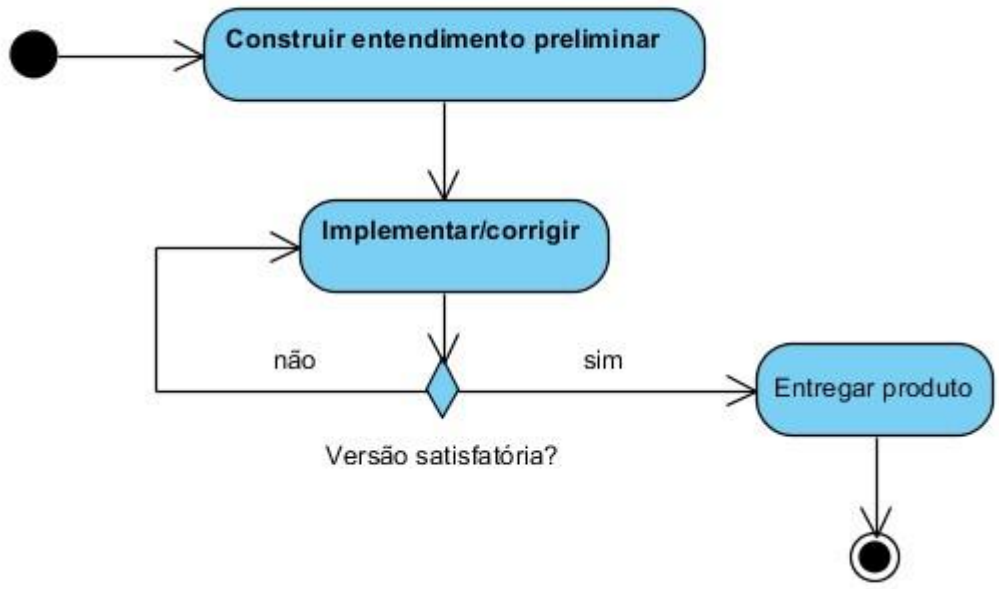

Figura 3.1 Modelo Codificar e Consertar[1]

[1]Todos os diagramas deste livro, salvo afirmação em contrário, seguem a notação UML 2 - *Unified Modeling Language* (Guedes, 2018).

Em resumo, esse modelo consiste em:

- Construir com o cliente um entendimento preliminar sobre o sistema que deve ser desenvolvido.
- Implementar uma primeira versão desse sistema.

- Interagir com o cliente de forma a corrigir a versão preliminar até que esta satisfaça o cliente.
- Fazer testes e corrigir os erros inevitáveis.
- Entregar o produto.

Pode-se dizer que se trata de uma forma bastante ingênua de modelo de processo e também que sequer parece um processo, pois não há previsibilidade em relação às atividades e aos resultados obtidos. Esse 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, por causa da pressão do cronograma. Empresas que não usam modelo de processo algum possivelmente utilizam o Modelo Codificar e Consertar por *default*.

Possivelmente, esse modelo é bastante usado em empresas de pequeno porte, mas deve ser entendido mais como uma maneira de pressionar os programadores (*code rush*) do que como um processo organizado (McConnell, 1996). Se o sistema não satisfaz o cliente, cobra-se do programador uma versão funcional adequada no menor tempo possível.

Apesar de tudo, esse modelo ainda pode ser usado 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:

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

Um dos problemas com essa técnica é que o código, que já sofreu várias correções, fica cada vez mais difícil de ser modificado. Além disso, aquilo de que o cliente menos precisa é um 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 seus objetivos. Outras desvantagens são:

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

Além disso, usar o Modelo Codificar e Consertar sem um plano de teste sistemático tende a produzir sistemas instáveis e com grande probabilidade de conter 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 vida e baseia-se na filosofia *BDUF* (*Big Design Up Front*, ou *design completo antes de tudo*): ela propõe que, antes de produzir linhas de código, deve-se fazer um trabalho detalhado de análise e design, 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 passar à fase seguinte. Se a revisão mostrar que o projeto não está pronto para passar à fase seguinte, deve permanecer na mesma fase (Ellis, 2010).

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 (Tabela 3.1).

Tabela 3.1: Marcos e entregáveis do Modelo Cascata

Marco/Fase/Objetivo	**Entregáveis**
LCR, *Life-cicle Concept*	Arquitetura de sistema aprovada e validada, incluindo questões básicas

Review Início da fase de planejamento e requisitos. Completar a revisão dos conceitos do ciclo de vida.	cas de hardware e software.
	Conceito de operação aprovado e validado, incluindo questões básicas de interação humano-computador.
	Plano de ciclo de vida de alto nível, incluindo marcos, recursos, responsabilidades, cronogramas e principais atividades.
SRR*, Software Requirements Review* Fim da fase de planejamento e requisitos. Completar a revisão dos requisitos do software.	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. Plano de uso detalhado: contraparte para os itens do plano de desenvolvimento como treinamento, conversão, instalação, operações e suporte.
	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.
	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.
	Contrato de desenvolvimento (formal ou informal) aprovado com base nos itens anteriores.
PDR*, Product Design Review* Fim da fase de design de produto. Completar a revisão do design do produto.	Especificação do design do produto de software verificada.
	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 costuma ter de 100 a 300 linhas de código).
	Estruturas de dados lógicas e físicas detalhadas em nível de seus campos.
	Orçamento para recursos de processamento de dados (incluindo especificações de eficiência de tempo, capacidade de armazenamento e precisão).
	Verificação do design com referência a completude, consistência, exequibilidade e rastreabilidade dos requisitos.
	Identificação e resolução de todos os riscos de alta importância.
	Plano de teste e integração preliminar, plano de teste de aceitação e manual do usuário.
CDR, *Critical Design Review* Fim da fase de design detalhado. Completar o design e revisar aspectos críticos das unidades.	Especificação de design detalhado revisada para cada unidade.
	Para cada rotina (menos de 100 instruções) dentro de uma unidade, especificar nome, propósito, hipóteses, tamanho, sequência de chamadas, entradas, saídas, exceções, algoritmos e fluxo de processamento.
	Descrição detalhada da base de dados.
	Especificações e orçamentos de design verificados em relação a completude, consistência e rastreabilidade dos requisitos.
	Plano de teste de aceitação aprovado.
	Manual do usuário e rascunho do plano de teste e integração completados.
UTC, Unit Test Criteria Fim da fase de codificação e teste de unidade.	Verificação de todas as unidades de computação usando-se não apenas valores nominais, mas também valores singulares e extremos.
	Verificação de todas as entradas e saídas unitárias, incluindo mensa-

Satisfação dos critérios de teste de unidade.	gens de erro.
	Exercício de todos os procedimentos executáveis e todas as condições de teste.
	Verificação de conformação a padrões de programação.
	Documentação em nível de unidade completada.
SAR, *Software Acceptance Review* Fim da fase de integração e teste. Completar a revisão da aceitação do software.	Testes de aceitação do software satisfeitos.
	Verificação da satisfação dos requisitos do software.
	Demonstração de performance aceitável acima do nominal, conforme especificado.
	Aceitação de todos os produtos do software: relatórios, manuais, especificações e bases de dados.
System Acceptance Review Fim da fase de implantação. Completar a revisão da aceitação do sistema.	Satisfação do teste de aceitação do sistema.
	Verificação da satisfação dos requisitos do sistema.
	Verificação da prontidão operacional de software, hardware, instalações e pessoal.
	Aceitação de todas as entregas relacionadas ao sistema: hardware, software, documentação, treinamento e instalações.
	Todas as conversões especificadas e atividades de instalação foram completadas.
Fim da fase de operação e manutenção. Corresponde à desativação do sistema.	Foram completadas todas as atividades do plano de desativação: conversão, documentação, arquivamento e transição para um novo sistema

As fases e marcos, conforme apresentados por Boehm, são resumidos na Figura 3.2.

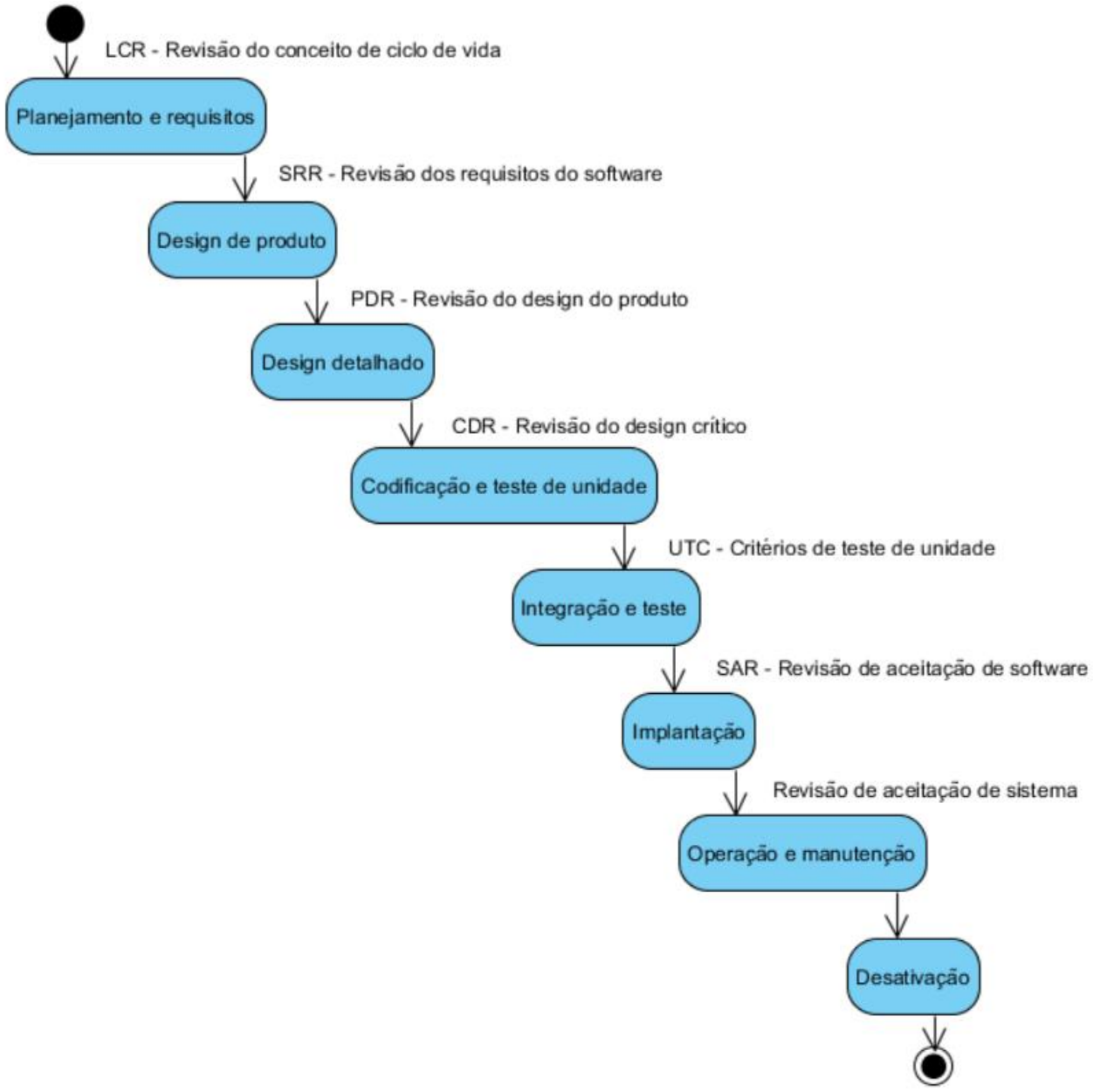

Figura 3.2 Fases e marcos do Modelo Cascata (Fonte: Boehm, 1981)

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

- A existência de fases bem definidas ajuda a detectar erros cedo; dessa forma, é mais barato corrigi-los.
- O modelo procura promover a estabilidade dos requisitos; assim, o projeto só segue em frente quando os requisitos são aceitos.
- Funciona bem com projetos nos quais os requisitos são bem conhecidos e estáveis, já que esse tipo de projeto se beneficia de uma abordagem organizada e sistemática.
- 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 essa abordagem é que, em geral, é fácil verificar se o código funciona direito, mas não é tão fácil verificar se modelos e projetos estão bem escritos. Para ser efetivamente viável, esse tipo de ciclo de vida necessitaria de ferramentas de análise automatizada de diagramas e

documentos para verificar sua exatidão.

O Modelo Cascata também tem sido um dos mais criticados da história, especialmente pelos adeptos dos modelos ágeis, que valorizam princípios diametralmente opostos aos desse modelo. Ellis (2010) aponta uma série de problemas com o Modelo Cascata:

- *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.
- *É difícil estabelecer requisitos completos antes de começar a codificar*: hoje, o desenvolvimento de software é entendido mais como um processo de amadurecimento do que como uma construção que pode ser baseada em um projeto detalhado desde o início. É natural que alguns requisitos só sejam descobertos durante o desenvolvimento de um projeto de software.
- *Desenvolvedores sempre reclamam que os usuários não sabem expressar aquilo de que precisam*: como os usuários não são especialistas em computação, muitas vezes não mencionam os problemas mais óbvios, que só aparecerão quando o produto estiver em operação.
- *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), que o apresentou pela primeira vez, embora não usasse, na época, a expressão *Waterfall* para designá-lo. O mais irônico nessa questão é que Royce apresentou justamente esse modelo como algo que *não* deveria ser seguido. Ele comenta que, embora acreditasse no modelo como filosofia de projeto organizado, achava sua implementação bastante arriscada, já que apenas na fase de teste vários aspectos do sistema seriam experimentados na prática pela primeira vez. Dessa forma, ele acreditava (e isso se confirma na prática) que, após a fase de testes, 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 podem ser resolvidos retornando-se 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 3.3.

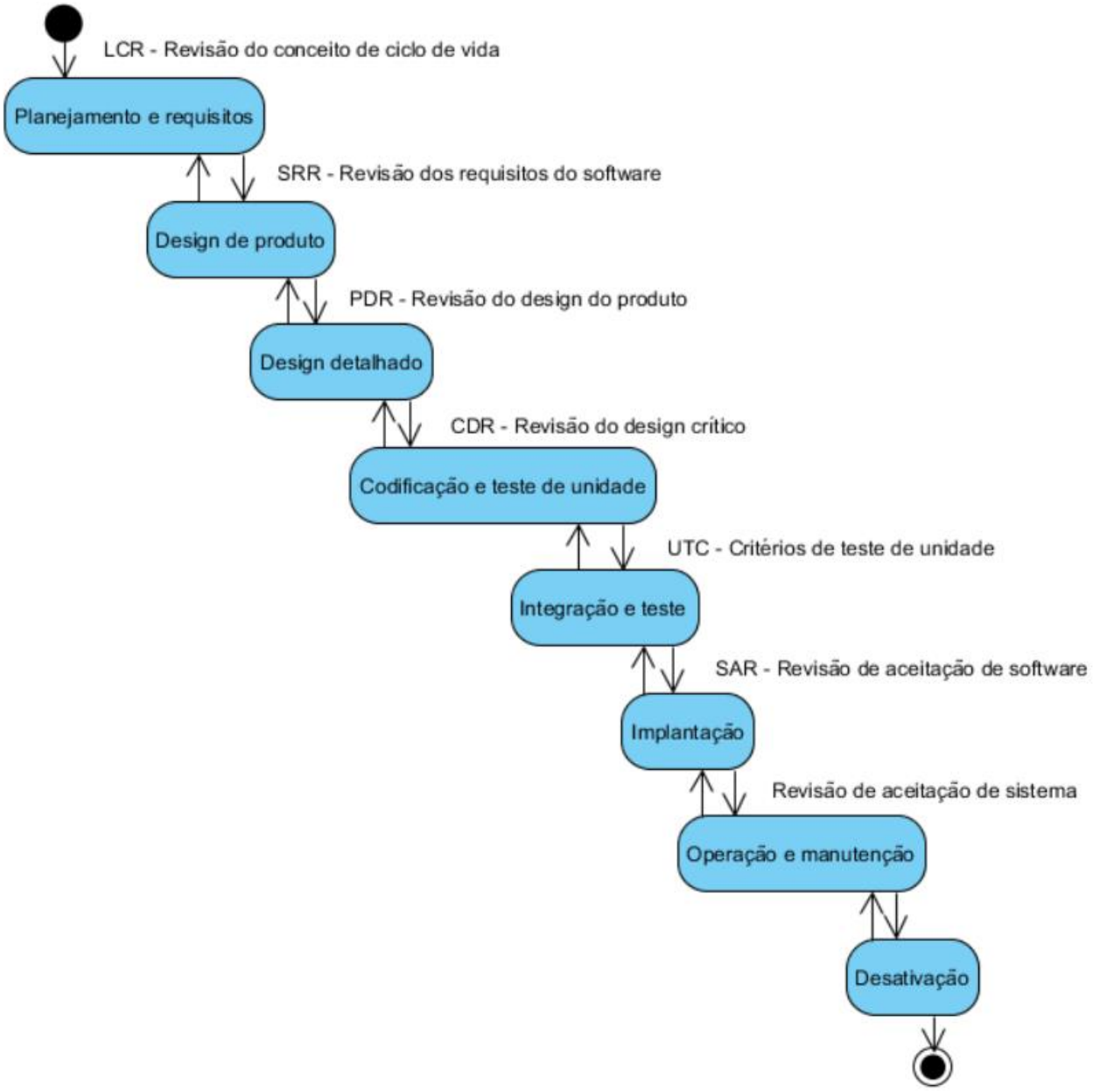

Figura 3.3 Como as interações entre fases poderiam ser (Modelo Cascata Dupla) (Fonte: Royce, 1970)

Novamente, Royce apresenta esse modelo como 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 3.3. Mas, como mostra a Figura 3.4, 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 esse problema.

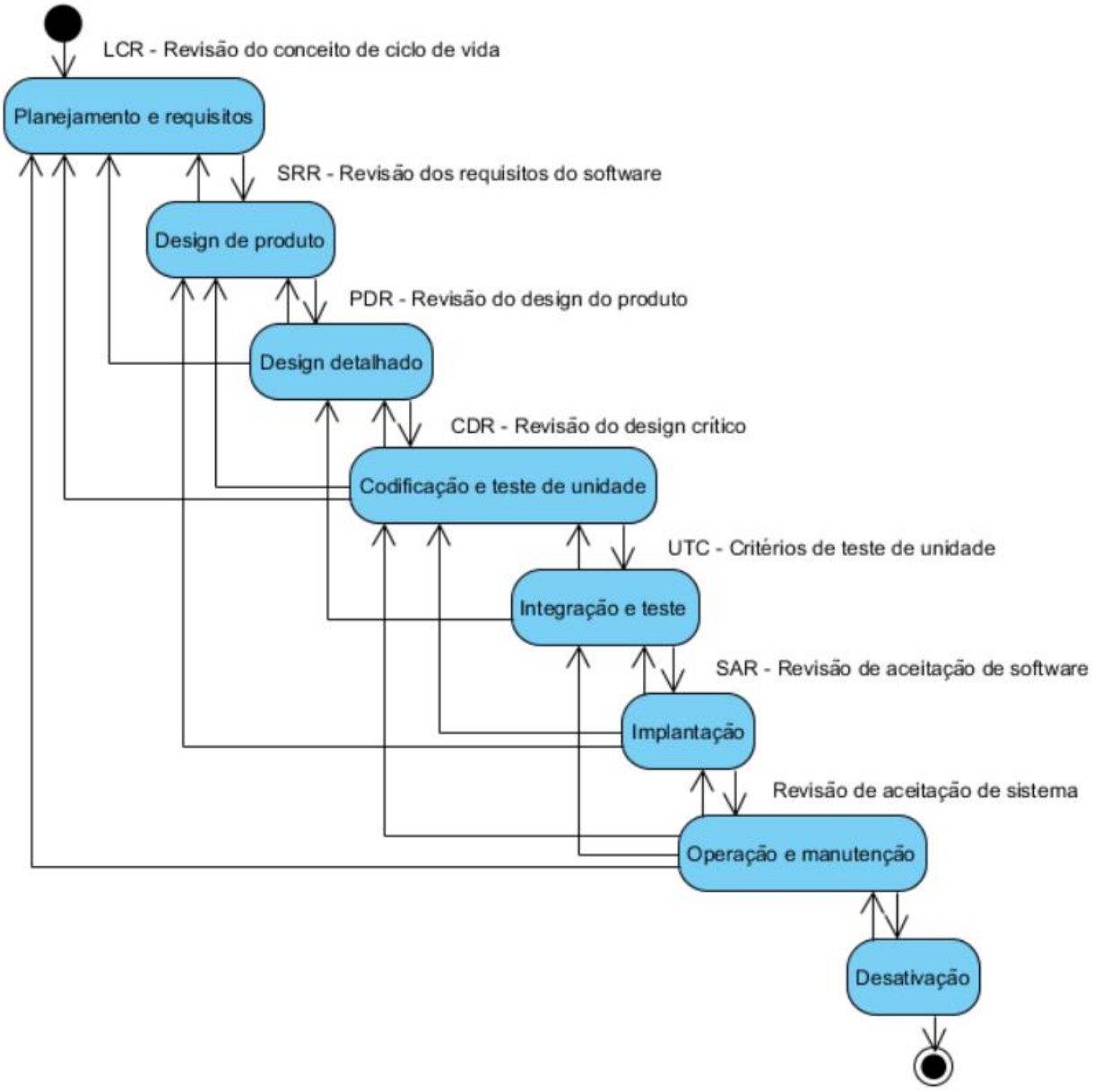

Figura 3.4 Como as interações entre fases do Modelo Cascata acabam acontecendo na prática (Fonte: Royce, 1970)

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

- *Inserir uma fase de* design *entre o levantamento dos requisitos e sua análise*: se *designers* que conhecem as limitações dos sistemas computacionais puderem verificar os requisitos antes de os analistas iniciarem seus trabalhos, poderão adicionar preciosos requisitos suplementares referentes às limitações físicas do sistema. Nos modelos modernos, os ciclos de redução de risco permitem realizar essa atividade.
- *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.
- *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 conhe-

cimento aprendido na primeira rodada para gerar um produto melhor na segunda vez. Essa ideia foi incorporada aos ciclos baseados em prototipação e iterações, como o Espiral.

- *Planejar, controlar e monitorar o teste*: testes devem ser sistemáticos e realizados por especialistas, já que são essenciais para o sucesso do sistema. Hoje, o Modelo V (Seção 3.4), o Modelo W (Seção 3.5), o Processo Unificado (Capítulo 5) e os modelos ágeis (Capítulo 4) apresentam a disciplina de teste (Capítulo 13) como algo fundamental no projeto do software.
- *Envolver o cliente*: é importante envolver o cliente formalmente no processo, e não apenas na aceitação do produto final. Os métodos ágeis, especialmente, consideram o cliente parte da equipe de desenvolvimento.

O Modelo Cascata, na sua forma mais simples, acaba sendo impraticável na maioria dos projetos, especialmente por conta da impossibilidade de se conhecer todos os requisitos *a priori*. Algumas variações foram propostas ao longo do tempo para permitir a aplicação desse tipo de ciclo de vida em processos de desenvolvimento de software reais. Algumas dessas variações são apresentadas nas seções seguintes.

3.3 Modelo *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. Em vez de cada fase produzir documentação completa para a fase seguinte, o Modelo *Sashimi* propõe que cada fase continue tratando as questões da fase anterior e procure iniciar o tratamento de questões da fase seguinte.

O diagrama de atividades da UML não é adequado para representar as atividades que se entrelaçam no Modelo *Sashimi* em razão de seu paralelismo difuso. Classicamente, ele tem sido representado como na Figura 3.5, com aparência de comida japonesa, composta por cortes de peixe sobrepostos, de onde vem seu nome.

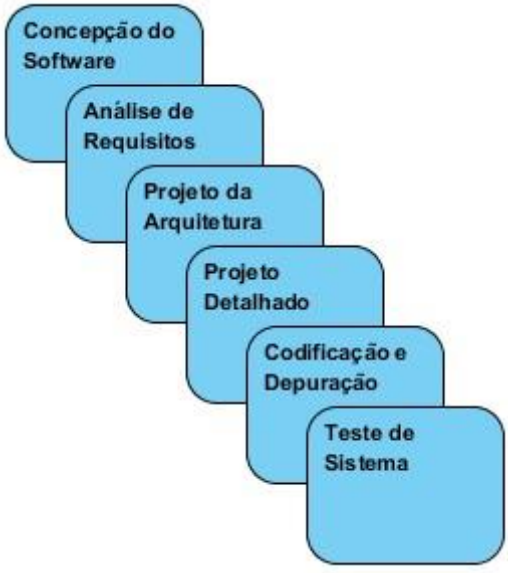

Figura 3.5 Modelo *Sashimi*

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 3.4. Segundo essa observação, não seria suficiente entrelaçar fases contíguas, pois pode-se necessitar retornar a outras fases anteriores a elas.

Em função disso, uma das evoluções mais importantes do Modelo *Sashimi* é o Modelo *Scrum* (Seção 4.1), que consiste em levar a ideia das fases entrelaçadas ao extremo pela redução do processo a uma única fase, realizada em iterações, na qual todas as etapas do Modelo Cascata seriam realizadas por profissionais especializados trabalhando em equipe.

O Modelo *Sashimi*, porém, tem o mérito de indicar que a fase de análise de requisitos só estará completa depois que as questões referentes ao *design* da arquitetura tiverem sido consideradas, e assim por diante para as outras fases subsequentes.

Esse 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 é mais difícil definir marcos (*milestones*), pois não fica muito claro quando uma fase termina e outra começa. Além disso, a realização de atividades paralelas com esse modelo pode levar a falhas de comunicação, à aceitação de hipóteses erradas e à ineficiência no trabalho.

3.4 Modelo V

O *Modelo V* (*V Model*) é uma variação do Modelo Cascata. Ele 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).

O Modelo V é sequencial e, como o Modelo Cascata, é dirigido por documentação. A Figura 3.6 mostra o diagrama de atividades com as fases do Modelo V, com as dependências de verificação indicadas entre as atividades.

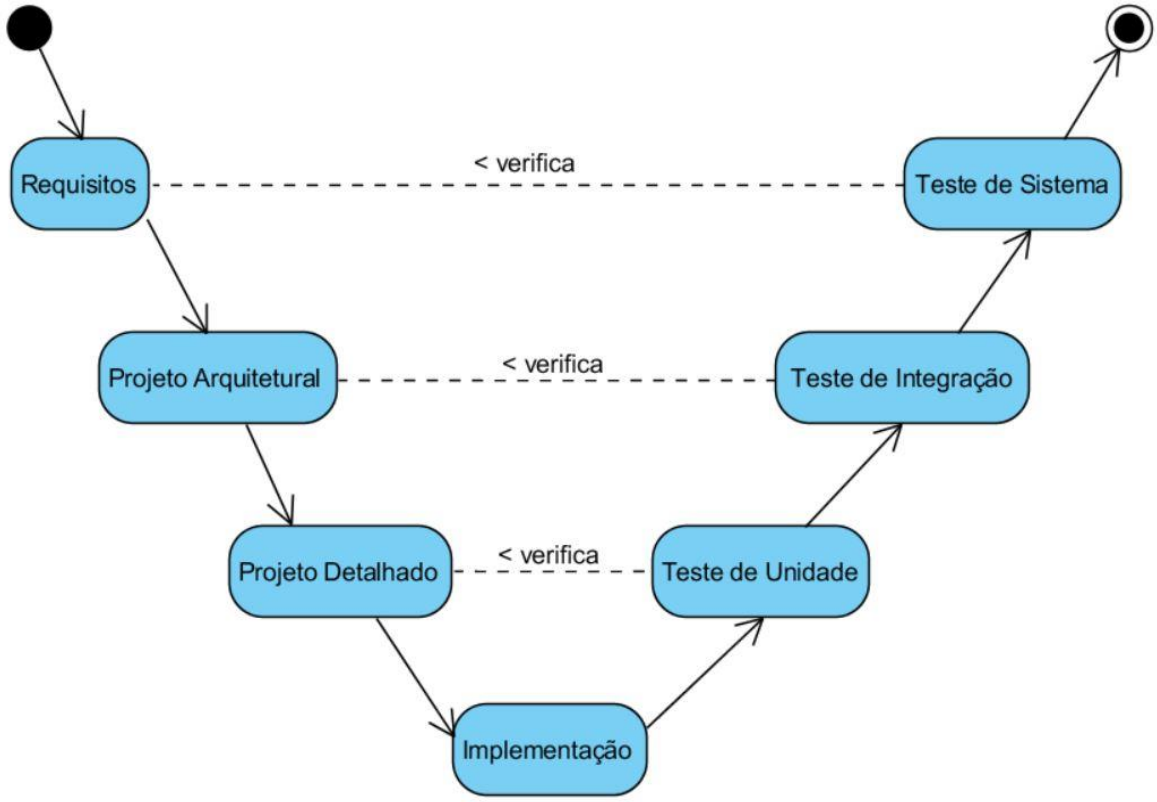

Figura 3.6 Modelo V

Na *fase de requisitos*, a equipe e o cliente eliciam e elaboram o documento de requisitos, que é

aprovado conjuntamente, na forma de um contrato de desenvolvimento. Uma estimativa de esforço (Capítulo 7) também deve ser produzida nessa fase.

Na *fase de* design *arquitetural*, a equipe organiza os requisitos em unidades funcionais coesas, definindo como as diferentes partes arquiteturais do sistema vão se interconectar e colaborar. Nessa 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 elas serão implementadas. Essa 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 nesses 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 desse modelo são os mesmos do Modelo Cascata puro, entre eles o fato de que mudanças nos requisitos geram muito retrabalho. Além 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.

3.5 Modelo W

Spillner (2002) apresenta uma variação ao Modelo V ainda mais voltada à área de testes: o *Modelo W*. A principal motivação para essa 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 se inicie durante a fase construtiva, mesmo sendo executado depois, e que o lado direito do V não seja considerado apenas um conjunto de atividades de testes, mas também de depuração e reconstrução.

A figura resultante desse modelo assemelha-se graficamente a uma letra W (Figura 3.7), em que as fases interiores do braço esquerdo são as fases construtivas do Modelo V e as exteriores do mesmo braço são fases de preparação para o teste. Já no braço direito, as fases interiores são as fases de teste do Modelo V, e as exteriores são fases de depuração que implicam na detecção dos defeitos encontrados no teste e correção dos mesmos.

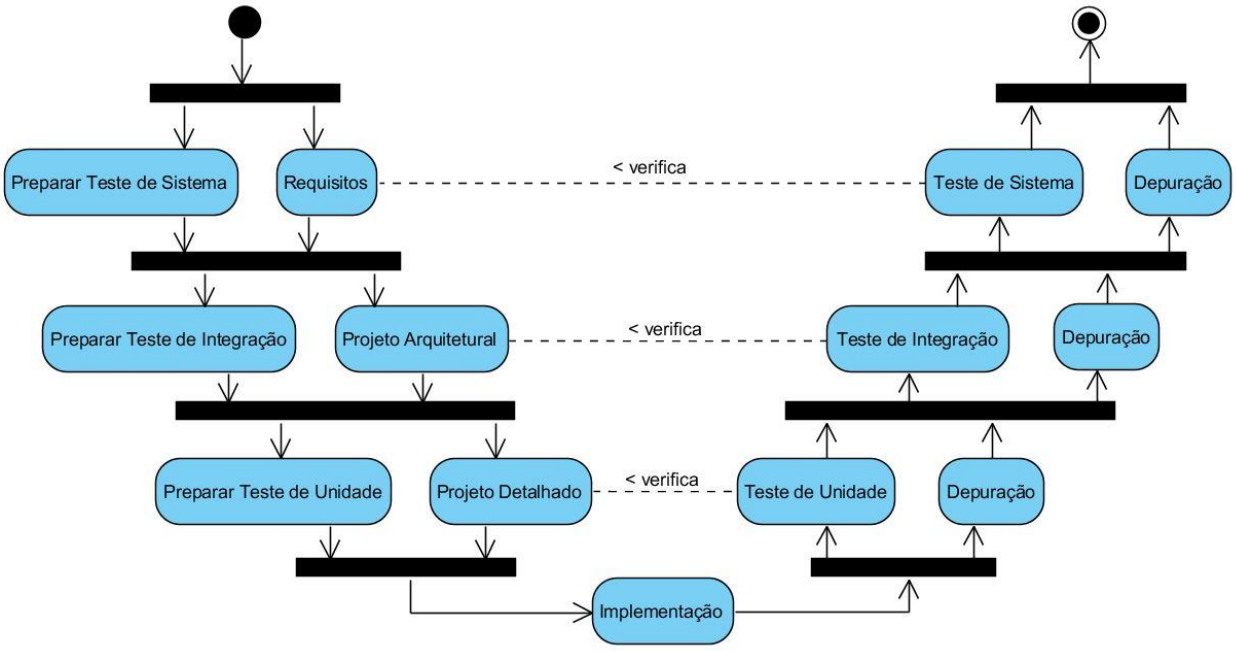

Figura 3.7 Modelo W

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 dessa fase. 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 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 cedo e *designs* excessivamente complexos sejam simplificados.

O Modelo W, assim, incorpora o teste nas atividades de desenvolvimento desde seu início, e não apenas nas fases finais. Tal característica também é fortemente utilizada pelos métodos ágeis, que preconizam o uso do Desenvolvimento Dirigido pelo Teste (TDD, do inglês *Test-Driven Development*) que estabelece 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 3.8 mostra um diagrama de atividades UML que representa o ciclo de vida Cascata com Subprojetos. Esse 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. Essa técnica explora melhor as potencialidades de modularidade do projeto e, com ela, o progresso é visto mais facilmente, porque podem-se produzir várias entregas de partes funcionais do sistema à medida que elas ficam prontas.

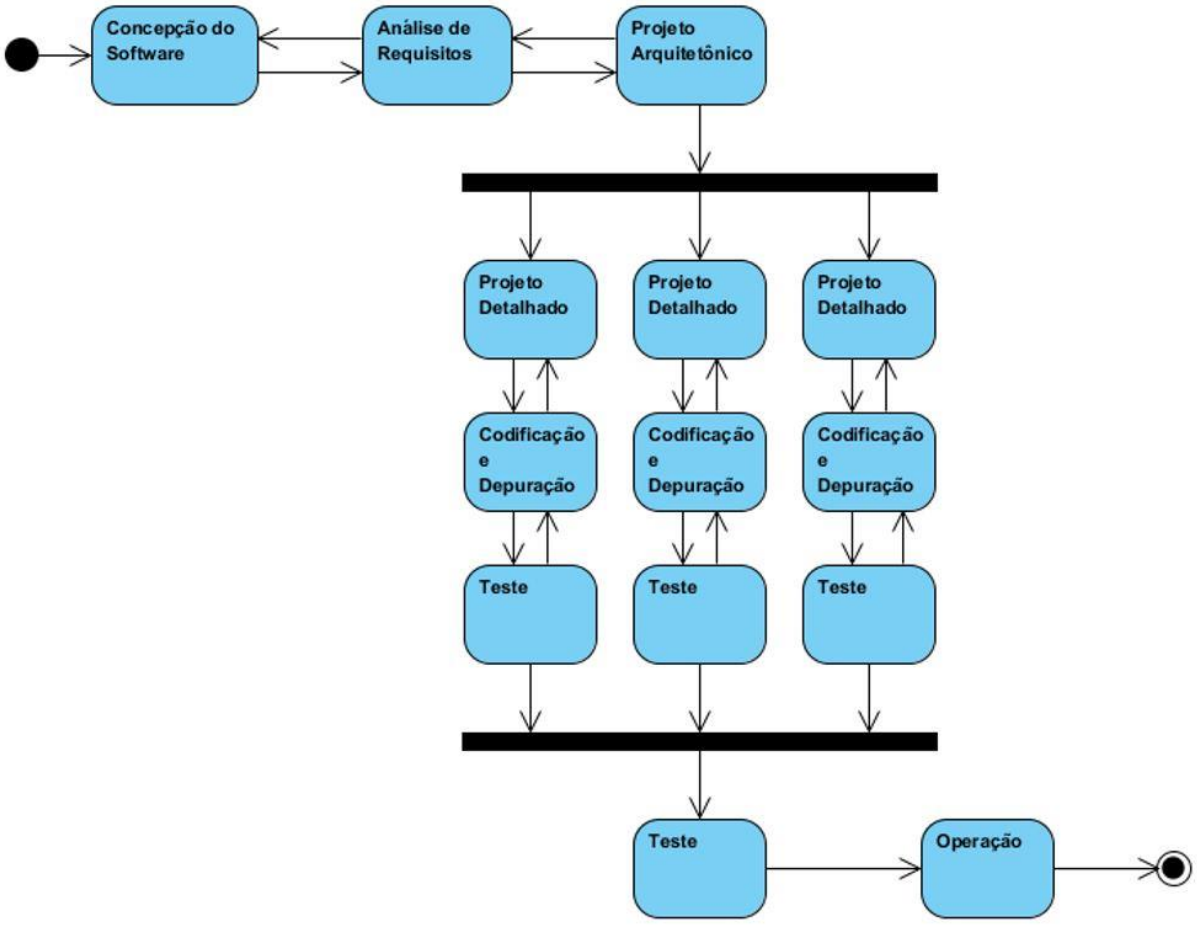

Figura 3.8 Modelo Cascata com Subprojetos

A maior dificuldade relacionada a esse 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, esse 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 a grandes fases de integração no final do desenvolvimento.

Apesar disso, esse 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 o 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 de ter uma boa definição dos requisitos do projeto nas fases iniciais. Esse 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 das técnicas utilizadas

durante a fase de redução de risco são:

- *Desenvolver protótipos de interface com o usuário*: nesse caso, o modelo por vezes é chamado de "*Cascata com Prototipação*". Essa 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.
- *Desenvolver* storyboards *com o usuário*: a técnica de *storyboards* (Gomes et al., 2007) utiliza imagens para descrever situações de uso do sistema. É similar à técnica de cenários (Jacobson, 1995), mas em geral os cenários são apenas resultado de dinâmica de grupo e documentados por texto.
- *Conduzir vários ciclos de entrevistas com o usuário e o cliente*: em vez de realizar apenas uma entrevista e obter os requisitos a partir dela, a técnica especifica que o cliente deve sempre receber um retorno sobre os requisitos levantados a cada entrevista e fazer uma validação contínua destes. Durante o processo de validação, uma nova entrevista poderá esclarecer aspectos confusos ou detalhar aspectos ainda muito gerais.
- *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 os *storyboards* e os cenários está no fato de que, normalmente, a atuação nos filmes não pode ser super simplificada nem falsificada pelos usuários. A desvantagem está no fato de que nem sempre surgem todos os fluxos de exceção ou variantes de um processo do usuário.
- *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 a modelar adequadamente os requisitos são bem-vindas.

A Figura 3.9 apresenta graficamente o 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. Ela não precisa ficar restrita aos requisitos do projeto; pode ser aplicada a quaisquer outros riscos identificados. Esse tipo de modelo é adequado a projetos com muitos riscos significativos. A espiral de redução de risco é uma forma de a equipe garantir alguma estabilidade ao projeto antes de comprometer muitos recursos na sua execução.

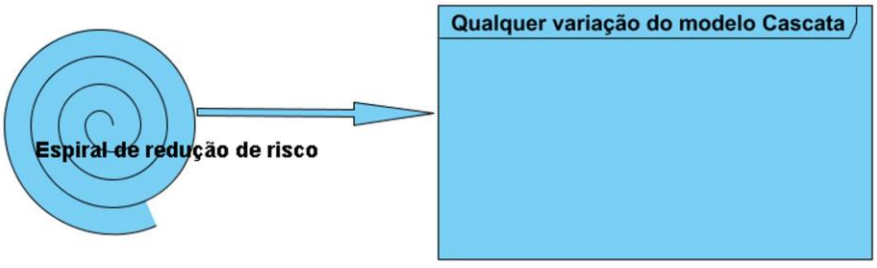

Figura 3.9 Modelo Cascata com Redução de Risco

Como principais desvantagens podem-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 por que as iterações eram necessárias. O projeto é dividido

em subprojetos, cada qual abordando 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 à recomendação de Royce (1970) de 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 essas 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 8).

Depois que os principais riscos foram mitigados, o Modelo Espiral prossegue de forma semelhante ao Modelo Cascata ou uma de suas variantes. A Figura 3.10 apresenta a definição clássica desse ciclo, a partir da qual seu nome foi escolhido.

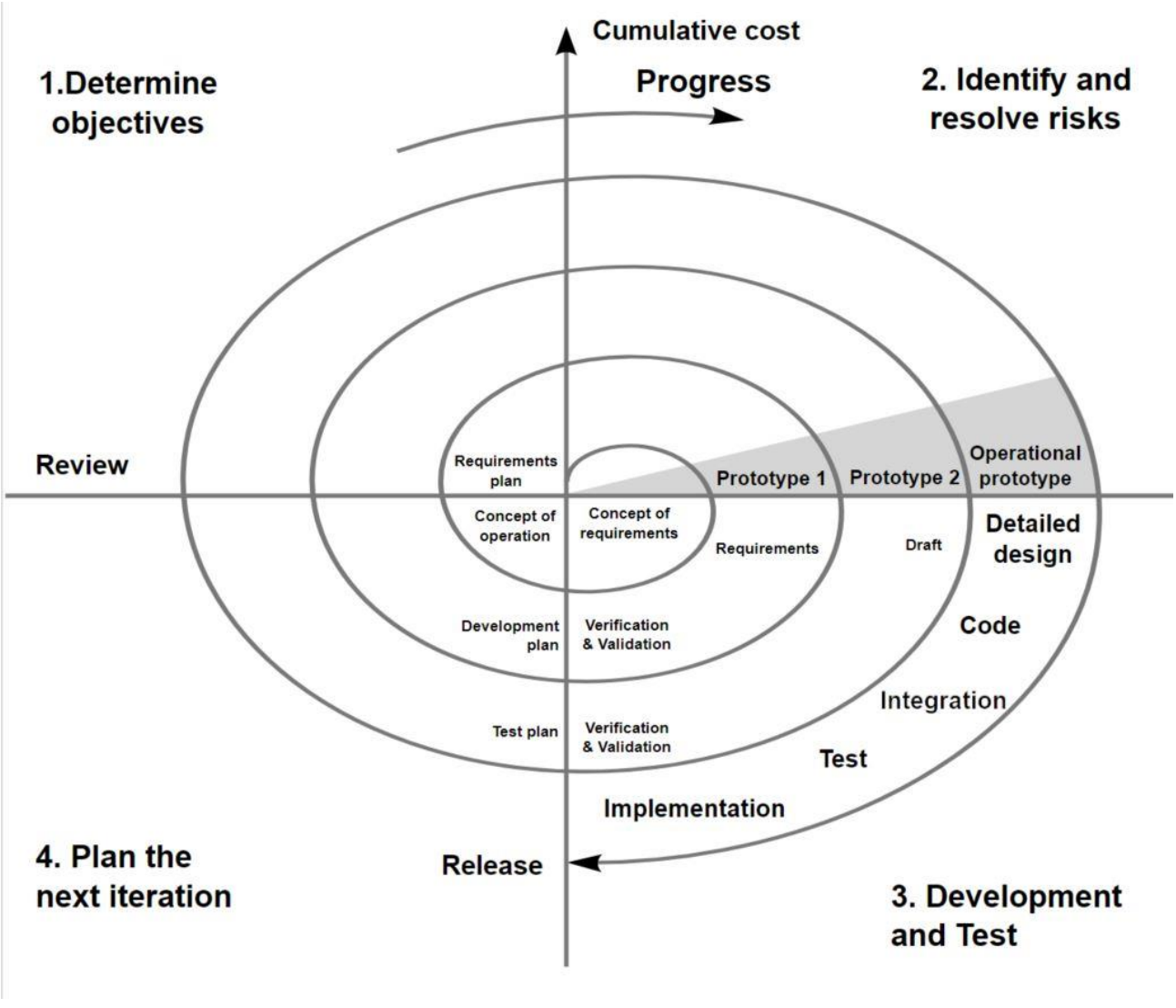

Figura 3.10 Modelo Espiral (Fonte: Boehm, 1986)

INFORMAÇÕES PARA TRADUÇÃO/EDIÇÃO DA FIGURA:
1. Determine objectives//1. Determinar objetivos
Cumulative cost//Custo cumulativo
Progress//Progresso

Wazlawick, R. S. *Engenharia de Software: Conceitos e práticas*, 2ª. ed. Elsevier, 2019.

2. Identify and resolve risk//2. Identificar e resolver riscos
Review//Revisão
Requirements plan//Plano de requisitos
Prototype 1//Protótipo 1
Prototype 2//Protótipo 2
Operational prototype//Protótipo operacional
Concept of operation//Conceito de operação
Concept of requirements//Conceito de requisitos
Requirements//Requisitos
Dratf//Rascunho
Detailed design//Design detalhado
Development plan//Plano de desenvolvimento
Verification & Validation//Verificação & Validação
Code//Código
Test plan/Plano de teste
Verification & Validation//Verificação & Validação
Integration//Integração
Test//Teste
Implementation//Implementação
4. Plan the next iteration//4. Planejar a próxima iteração
Release//Entrega
3. Development and test//3. Desenvolvimento e teste

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

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

Cada iteração do modelo envolve seis passos:

- Determinar inicialmente os objetivos, alternativas e restrições relacionadas à iteração que vai se iniciar.
- Identificar e resolver riscos relacionados à iteração em andamento.
- Avaliar as alternativas disponíveis. Nessa fase, podem ser utilizados protótipos para verificar a viabilidade de diferentes alternativas.
- Desenvolver os artefatos (possivelmente entregas) relacionados a essa iteração e certificar-se de que estão corretos.
- Planejar a próxima iteração.
- 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, também, aquelas 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 nessas fases iniciais podem variar muito de projeto para projeto.

À medida que os custos aumentam, porém, 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 é bastante adequado a projetos complexos, com alto risco e requisitos pouco conhecidos, como projetos de pesquisa e desenvolvimento (P&D). Não se recomenda esse ciclo de vida para projetos de pequeno e médio porte ou com requisitos estáveis e conhecidos.

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

3.9 Prototipação Evolucionária

Em geral, distinguem-se duas abordagens de prototipação:

- *Throw-away* (*descartável*), que consiste na construção de protótipos que são usados unicamente para estudar aspectos do sistema, entender melhor seus requisitos e reduzir riscos. O protótipo, depois de cumprir essas finalidades, é descartado (Crinnion, 1991).
- *Cornerstone* (*pedra fundamental*), que consiste na construção de protótipos que também são usados para estudar aspectos do sistema, entender melhor seus requisitos e reduzir riscos. O protótipo será parte do sistema final, ou seja, ele vai evoluindo até se tornar um sistema que possa ser entregue (Budde et al., 1992).

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

O Modelo de Prototipação Evolucionária sugere que a equipe de desenvolvimento trabalhe com o cliente os aspectos mais visíveis do sistema, na forma de protótipos (em geral de interface), até que o produto seja aceitável. A Figura 3.11 apresenta o diagrama de atividades UML simplificado para esse modelo.

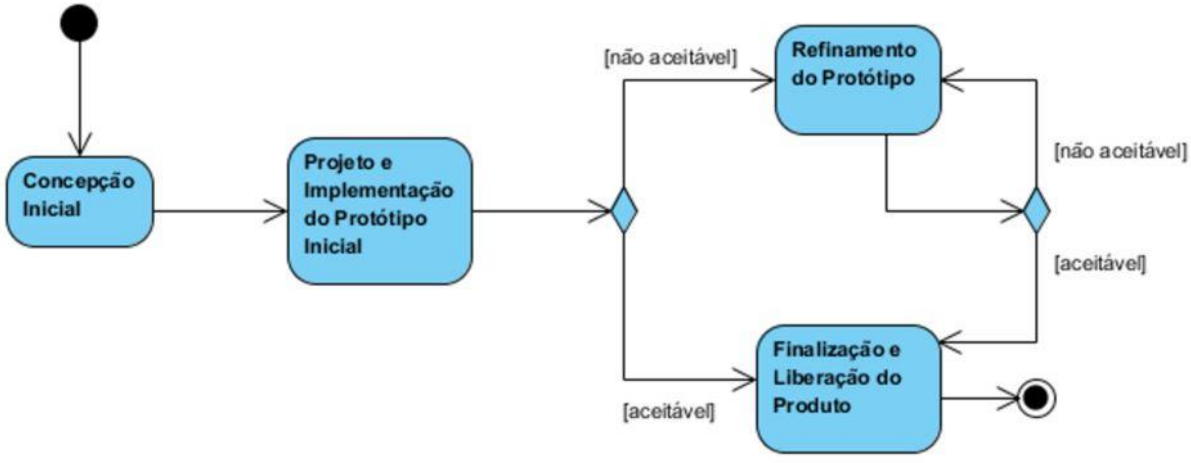

Figura 3.11 Modelo Prototipação Evolucionária

A principal diferença entre este modelo e o Codificar e Consertar reside no fato de que na Prototipação Evolucionária cada protótipo é construído com um ou mais objetivos de verificação e valida-

ção, sejam requisitos, interfaces, funcionalidades ou regras de negócio, enquanto que no caso do Codificar e Consertar, tenta-se aleatoriamente chegar ao produto final por tentativa e erro.

Esse modelo pode ser particularmente interessante se for difícil fazer o cliente comunicar os requisitos. Nesse 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.

Esse modelo também pode ser interessante quando nem a equipe nem o cliente 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 ou em relação à gerência do projeto, já que é difícil avaliar quando cada fase foi efetivamente realizada. O projeto que seguir esse modelo pode até regredir para o Modelo 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 um projeto realista antes de iniciar a codificação propriamente dita.

3.10 Entregas em Estágios

O *Modelo Entregas em Estágios* (*Staged Deliveries*) ou *Implementação Incremental* é uma variação mais bem estruturada do Modelo de Prototipação Evolucionária, embora também seja considerado uma variação do Modelo Cascata com Subprojetos. A principal diferença entre este modelo e o modelo de prototipação está no fato de que a prototipação vai organizar seus ciclos basicamente em torno da ideia de refinamentos sucessivos, iniciando com protótipos mais abstratos e incorporando aspectos mais concretos ao longo das iterações. Já o modelo de entrega em estágios organiza seus ciclos em torno de entregas de partes do sistema. Assim, neste caso, não se trata de entregar protótipos, mas de entregar partes do sistema completas e finalizadas, que não precisarão mais ser refinadas.

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, em que o sistema só é entregue quando totalmente acabado.

A Figura 3.12 mostra o diagrama de atividades UML para o modelo. Assim que o *design* arquitetural estiver completo, será possível iniciar a implementação e a entrega de partes funcionais do produto.

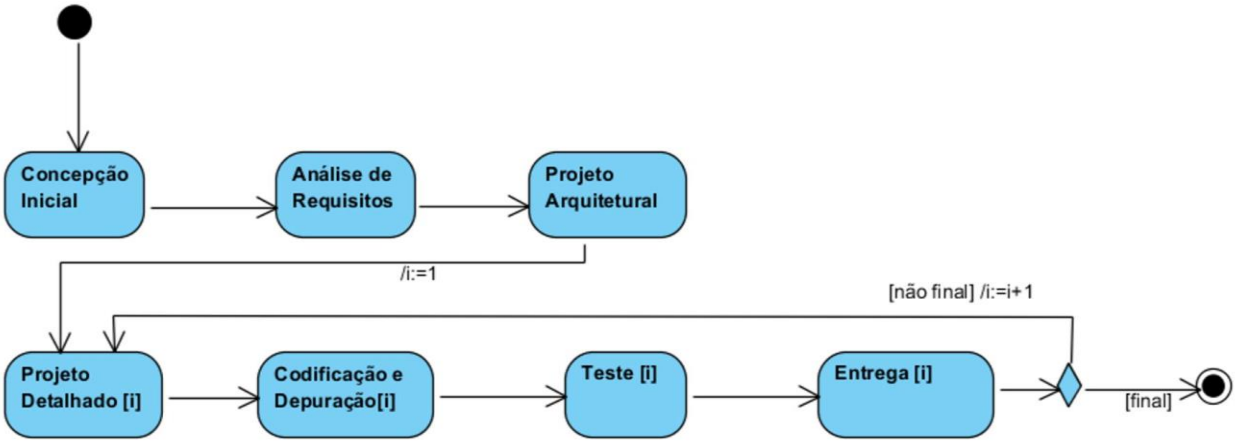

Figura 3.12 Modelo Entregas em Estágios

Uma das principais vantagens desse modelo (Ellis, 2010) é o fato de colocar funcionalidades úteis nas mãos do cliente antes de completar o projeto. Se os estágios forem planejados cuidadosamente, funcionalidades importantes estarão disponíveis muito mais cedo do que com outros ciclos de vida.

Além disso, esse modelo provê entregas mais cedo e de forma contínua, o que pode aliviar um pouco a pressão de cronograma colocada na equipe.

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

- *Técnico*: as dependências técnicas entre os diferentes módulos entregáveis devem ser cuidadosamente verificadas. Se um módulo tem dependências com outro, possivelmente o segundo deverá ser entregue antes deste, ou então, técnicas de prototipação como a construção de *stubs* (Seção 13.1.4) poderão ser necessárias.
- *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. Além disso, o cliente poderá ter suas prioridades em relação a quais funcionalidades são mais importantes e mais urgentes.

Para essa técnica funcionar, é necessário que os requisitos sejam bem compreendidos e o planejamento seja efetivo. Ela também pode ser considerada uma precursora dos modelos iterativos, como UP e métodos ágeis.

Também se recomenda que o gerente de projeto, neste caso, seja bastante experiente, para que possa justamente perceber e planejar as atividades de desenvolvimento nos níveis técnico e gerencial.

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 apenas 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 3.13 apresenta esse modelo.

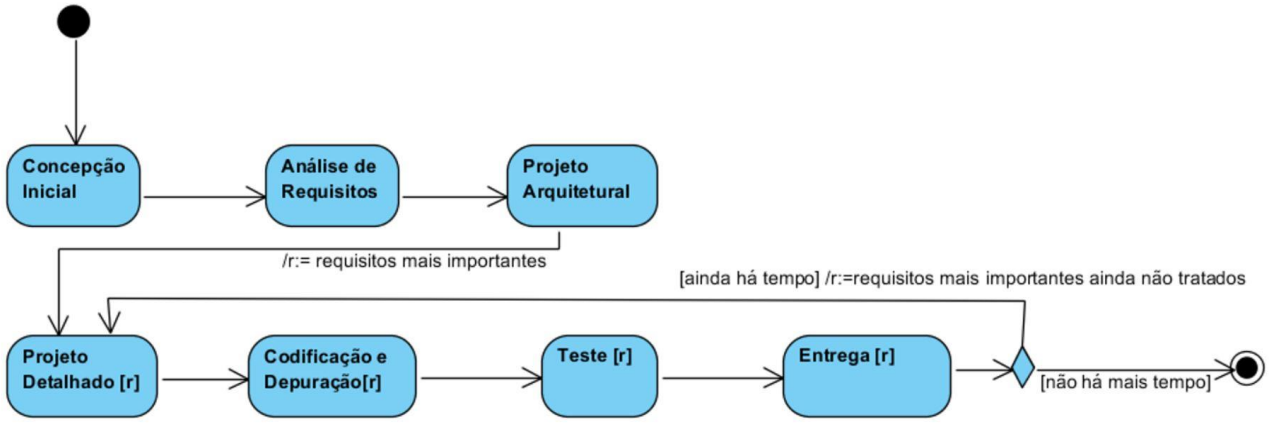

Figura 3.13 Modelo Orientado a Cronograma

Na figura, a cada iteração do ciclo de desenvolvimento, desenvolve-se um conjunto de requisitos, tomando primeiro aqueles que ainda não tiverem sido abordados. Encerra-se o projeto quando o tempo limite for atingido ou quando todos os requisitos tiverem sido atendidos. Espera-se que, mesmo que não tenha sido possível atender a todos os requisitos, pelo menos os que ficaram de fora sejam os menos importantes.

Esse modelo é uma boa estratégia para garantir que haverá algum produto disponível em determinada data, se isso for absolutamente imprescindível, o que faz dele um modelo apropriado para

quando existe uma data limite para a entrega, que é intransferível. Porém, se a equipe é altamente confiante na sua capacidade de previsão de esforço (se planeja e cumpre prazos constantemente), essa abordagem não trará necessariamente maiores vantagens. O modelo é especialmente útil quando a equipe tem dificuldade em prever e cumprir um cronograma no longo prazo.

Uma das desvantagens desse modelo é que, caso nem todas as funcionalidades sejam entregues, a equipe terá perdido tempo analisando-as nas etapas iniciais.

Em relação ao Processo Unificado e aos métodos ágeis, esse modelo difere na forma como concebe as iterações. No Modelo Orientado a Cronograma, a duração das iterações não é estabelecida *a priori*. Já nos modelos mais modernos se estabelece uma duração fixa para as iterações e se tenta implementar um conjunto de funcionalidades dentro dos prazos fixos estabelecidos. Dessa forma, é mais fácil verificar se o projeto está andando bem ou atrasando. Mas, para que isso funcione, é necessário ser capaz de estimar o esforço necessário para desenvolvê-lo (Capítulo 7).

3.12 Entrega Evolucionária

O *Modelo Entrega Evolucionária* (*Evolutionary Delivery*) é uma combinação da Prototipação Evolucionária (Seção 3.9) com a Entrega em Estágios (Seção 3.10). Nesse modelo, a equipe também desenvolve uma versão do produto, mostra ao cliente e cria novas versões baseadas no *feedback* dado por ele.

O quanto esse modelo se aproxima da Prototipação Evolucionária ou da Entrega em Estágios depende do grau em que se pretende acomodar as modificações solicitadas no *feedback*:

- Se a ideia é acomodar todas ou a grande maioria das modificações, então a abordagem tende mais para a Prototipação Evolucionária.
- Se, entretanto, as entregas continuarem sendo executadas de acordo com o planejado e as modificações acomodadas aos poucos nas entregas, então a abordagem se parece mais com a Entrega em Estágios.

Assim, esse modelo permite ajustes que lhe deem um pouco da flexibilidade do Modelo de Prototipação Evolucionária, ao mesmo tempo que se tenha o benefício do planejamento da Entrega em Estágios.

As diferenças entre esse modelo e os anteriores 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.

De certa maneira, o Processo Unificado e os modelos ágeis implementam essa flexibilidade ao permitir que, ao planejar cada iteração, o gerente de projeto escolha se vai abordar um *caso de uso* (equivalente a um conjunto de requisitos) conforme planejado, um *risco* que precisa ser mitigado ou uma *requisição de modificação*, baseando-se em suas prioridades.

3.13 Modelos Orientados 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 *jCompany* (Alvim, 2008) ou *WebRatio* (Ceri et al., 2003). [QRC 3.1, 3.2] É uma abordagem extremamente rápida de desenvolvimento e prototipação, mas é limitada pelas funcionalidades oferecidas pelas ferramentas específicas.

Assim, conforme mostrado na Figura 3.14, 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 grande parte dos requisitos rapidamente.

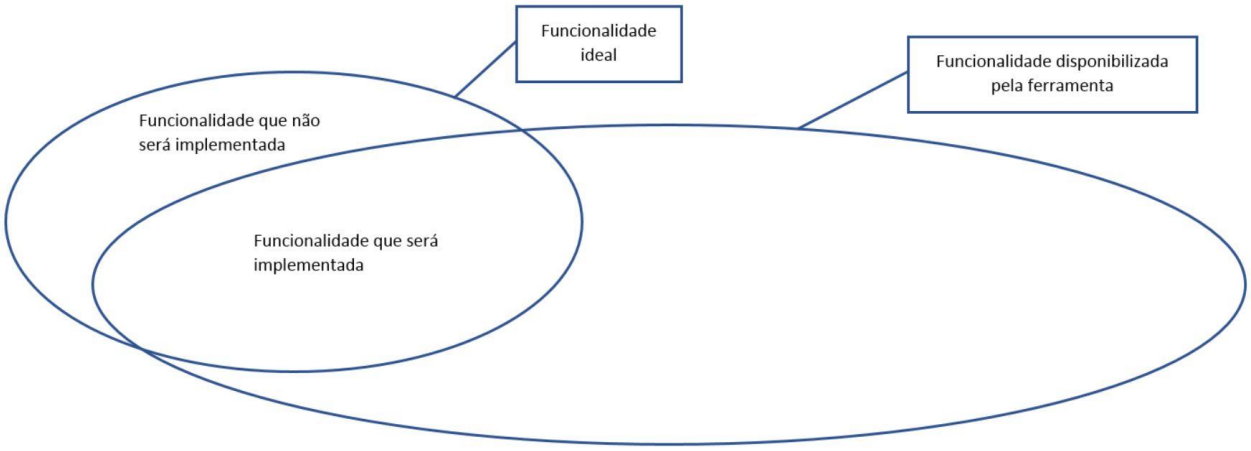

Figura 3.14 Como ficam os requisitos em Modelos Orientados a Ferramentas

Essa abordagem pode ser combinada com outros modelos de processo. A prototipação necessária no Modelo Espiral, por exemplo, pode ser feita com 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 de um conjunto de sistemas de software que compartilham características comuns gerenciadas de maneira 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), as SPLs 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. Dessa forma, o reuso deixa de ser realizado de forma imprevisível e *ad hoc* para ser incorporado sistematicamente aos 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 de produtos (Figura 3.15). Segundo Rombach (2005), o investimento em uma SPL começa a compensar a partir do terceiro produto gerado.

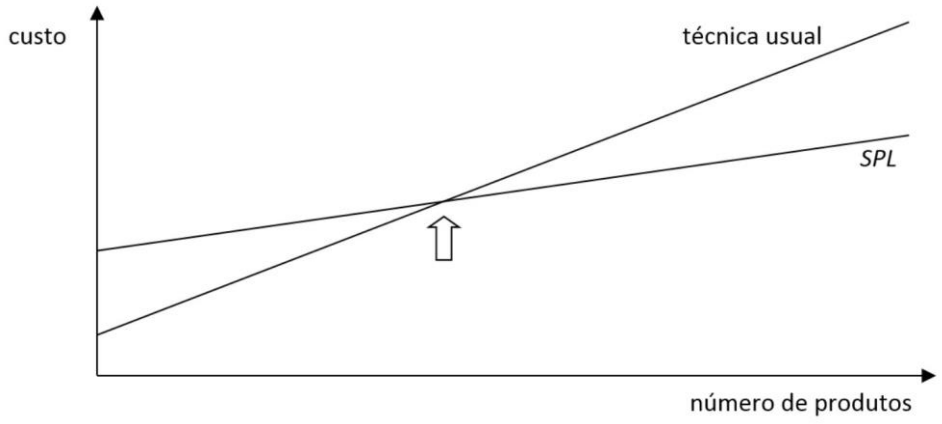

Figura 3.15 Custo/benefício de SPL

A técnica tem a ver com o desenvolvimento de vários produtos diferenciados, mas com característi-

cas comuns a partir de um núcleo comum. Assim, SPL não se aplica a situações nas quais um único produto é desenvolvido ou vários produtos não relacionados são desenvolvidos.

SPL trata de famílias de produtos como sistemas relacionados a uma mesma área (software para eventos esportivos, 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. 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 escolha da arquitetura é fundamental para se 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, entre todas as novas disciplinas relacionadas a SPL, três atividades essenciais:

- Desenvolvimento de um núcleo de ativos de produtos (*core asset*).
- Desenvolvimento de produtos.
- Gerência.

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

A Figura 3.16 apresenta um diagrama de comunicação UML que esquematicamente mostra a organização do desenvolvimento usando SPL.

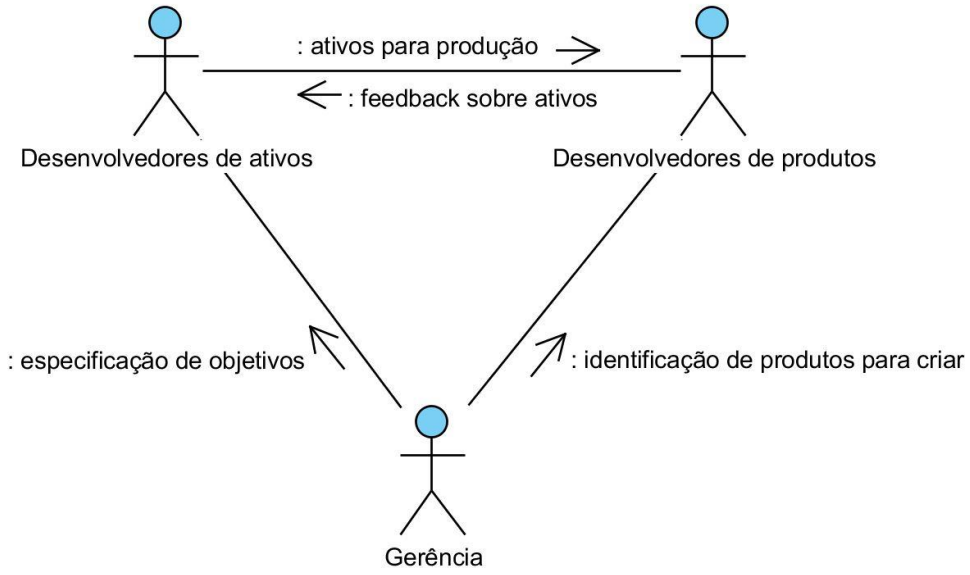

Figura 3.16 Organização do Desenvolvimento com SPL (Adaptado de McGregor, 2004)

Ao contrário de outras figuras neste capítulo, que na maioria dos casos indicavam as dependências entre atividades, esta mostra três grupos de pessoas interagindo de forma não sequencial. Internamente as equipes podem usar os processos que lhes parecerem mais adequados para o desenvolvimento dos ativos e dos produtos; mas esta comunicação entre os diferentes grupos deve ocorrer de forma constante para que a SPL apresente bons resultados.

3.14.1 Desenvolvimento do Núcleo de Ativos

O objetivo da atividade de *desenvolvimento do núcleo de ativos* é estabelecer um potencial para a produção de produtos de software. O núcleo de ativos é o conjunto de elementos que podem ser

reusados na SPL. Esse reuso nem sempre ocorre da mesma forma, por isso talvez precisem ser identificados pontos de variação. *Pontos de variação* são as características dos elementos que podem variar de um reuso para outro. Em geral, consistem em um conjunto de restrições às quais suas eventuais instanciações devem se conformar. Além disso, não se apresentam apenas como módulos de software a serem reusados, mas também incluem um conjunto de instruções sobre como proceder ao seu reuso.

O desenvolvimento do núcleo de ativos, assim como as outras duas atividades, é iterativo. Ele ocorre sob a influência de certos fatores ambientais, que são definidos assim**:**

- *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 (interfaces com sistemas externos, por exemplo) elas devem observar? Quais são os requisitos de qualidade (como segurança e disponibilidade)?
- *Restrições de produção*: qual é o prazo esperado para disponibilização de um novo produto? Quais ferramentas de produtividade serão disponibilizadas? Quais padrões de processo de desenvolvimento serão adotados?
- *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?
- *Ativos preexistentes*: quais ativos da organização podem ser usados na SPL? Existem tanto bibliotecas, frameworks, componentes, *web services* produzidos internamente quanto obtidos fora?

Em relação à estratégia de produção, ela é *proativa* quando os componentes que serão reusados são planejados antes do desenvolvimento dos produtos em si. Por outro lado, ela é *reativa* quando durante o desenvolvimento dos produtos percebe-se oportunidades de desenvolvimento de componente reusáveis.

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

- *Escopo da SPL*: é uma descrição dos produtos que a SPL é capaz de produzir. Essa 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.
- *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 oportunidades de reuso que justifiquem seu gerenciamento na base de ativos. Cada ativo deve ter anexado um processo que especifica como ele deve ser usado na produção de novos produtos.
- *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 com a descrição do produto individual.

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 essa atividade são:

- 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.
- O escopo da SPL, que indica se é viável incluir um novo produto na linha.
- O núcleo de ativos a partir do qual o produto é construído.
- O plano de produção, que detalha como o núcleo de ativos pode ser usado para produzir o produto.

Já as saídas previstas para a atividade de desenvolvimento do produto incluem:

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

3.14.3 GERÊNCIA DE SPL

A atividade de gerência, 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 de oportunidades com a SPL. Pode ser considerada a responsável pelo sucesso ou pelo 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.

As três disciplinas (desenvolvimento do núcleo e ativos, desenvolvimento do produto e gerência) são organizadas de forma dinâmica, e 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) apresenta um conjunto de orientações para empresas que desejam adotar SPLs.

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

Uma excelente leitura para aprofundar aspectos práticos de SPL está disponível no *site* do SEI (vide *QR Code* ao lado). [QRC 3.3] Nas descrições das práticas de engenharia, de gerência e organizacionais são destacadas as diferenças fundamentais entre os processos usuais e os processos envolvendo SPLs.

REMISSIVO DO CAPÍTULO