2
Métodos de desenvolvimento de produtos de interfaces gráficas digitais: da abordagem em cascata à ágil

Este capítulo apresenta os métodos para desenvolver interfaces gráficas digitais, enfatizando o método ágil, que faz parte do recorte desta pesquisa. Mas, antes de apresentar cada um dos métodos, é preciso definir o significado de interface gráfica digital. No entanto, como uma interface gráfica digital só pode existir a partir de um *software*, o segundo tópico deste capítulo aborda este tema. Apresenta-se a história do *software* ao longo das décadas, o processo de avanço da tecnologia do *hardware*[1], que gerou um aumento dos custos e dos prazos para o *software* ser desenvolvido, a crise de 1968 por causa do processo informal de desenvolvimento utilizado na época e o surgimento do conceito de engenharia de *software*, que se tornou uma forma de propor métodos mais organizados de produção. Também apresenta-se os métodos mais famosos de desenvolvimento, com o objetivo de apresentar o que existe de mais conhecido sobre o assunto.

Em seguida, os métodos em cascata e ágil são aprofundados, por serem importantes para o recorte desta pesquisa. São traçadas as vantagens e desvantagens de cada um, a partir do ponto de vista de autores especializados no assunto. Apresenta-se a transição do método em cascata para o ágil, assim como os motivos que levaram a indústria a adotar métodos mais ágeis de desenvolvimento de *software*, principalmente as empresas que trabalham com tecnologia de ponta e inovação, mídias digitais e internet.

Para finalizar, há uma discussão sobre os impactos desta mudança de metodologia de desenvolvimento, principalmente em relação ao trabalho do designer, enquanto projetista e pesquisador inserido nesse cenário.

[1] ***Hardware*****:** é um termo utilizado na informática para designar um conjunto de componentes físicos (placas, monitor, equipamentos periféricos, etc.) de um computador.

2.1. Produtos de interfaces gráficas digitais

Ao analisarmos a definição de "interface" pelo lado da informática, encontramos duas explicações: (1) "fronteira compartilhada por dois dispositivos, sistemas ou programas que trocam dados e sinais" e (2) "meio pelo qual o usuário interage com um programa ou sistema operacional (p.ex., *DOS*, *Windows*)" (Dicionário HOUAISS Eletrônico, 2009).

De acordo com o complemento da definição:

> "Meio de interação do usuário com um programa ou sistema operacional que emprega recursos gráficos (ícones e janelas) na edição de documentos, na utilização de programas, dispositivos e outros elementos, tendo como principal dispositivo de entrada o mouse" (Dicionário HOUAISS Eletrônico, 2009).

Como exposto acima, pode-se observar que um produto de interface gráfica digital requer um sistema por trás para gerar esta interface. Pode-se concluir que para haver um produto de interface gráfica digital, deve haver um *hardware* (máquina) e, minimamente, um *software* (que podemos considerar como a reunião de um ou mais sistemas) por trás, ou seja, o *software* é a fonte geradora desta interface.

O conceito de que uma interface gráfica digital é formada por uma linguagem de computador, fica ainda mais claro ao analisarmos a citação abaixo:

> "A interface atua como uma espécie de tradutor da linguagem do computador numa relação semântica entre significado e expressão, pois o computador trabalha com sinais e símbolos por intermédio de uma linguagem binária, e os seres humanos pensam a partir de conceitos, imagens, sons e associações. O computador só pode 'representar a si mesmo' numa linguagem compreensível ao ser humano, graças ao invento da interface gráfica" (JOHNSON, 2001 apud VIANNA, 2005).

Apesar de haver uma clara associação entre a formação de uma interface gráfica digital e a linguagem de computador que a compõe, para analisarmos os métodos para desenvolvimento de produtos de interfaces gráficas digitais, antes, é preciso aprofundar a definição de *software*.

No capitulo 3 desta dissertação de mestrado, é descrito com maior profundidade como a interface de um produto digital deve ser projetada e avaliada pelas pessoas que irão utilizá-la. Mas, antes de abordar o projeto da interface em si, é preciso entender as diferenças da construção de um *software* para outros tipos de projetos. Por isso, o foco deste capítulo será apenas no

sistema que gera a interface (*software*) e em como se dá o seu processo de desenvolvimento.

2.2. Software

Segundo PRESSMAN (1995), uma descrição de *software* num livro didático poderia assumir a seguinte forma:

> "(1) instruções (programas de computador) que, quando executadas, produzem a função e o desempenho desejados; (2) estruturas de dados que possibilitam que os programas manipulem adequadamente a informação; e (3) documentos que descrevem a operação e o uso dos programas".

Para SOMMERVILLE (2007), uma definição resumida de *software* seria:

> "Programas de computador e documentação associada. Os produtos de *software* podem ser desenvolvidos para um cliente específico ou para um mercado geral".

É possível verificarmos que, nas duas definições acima descritas, há o conceito de que o *software* é desenvolvido para uso, seja específico por um outro *software* ou pelas pessoas, o que, neste caso, requer uma interface para interação, que precisa ser projetada especificamente. PRESSMAN (1995) afirma que para entender o conceito de *software*, e até o que é engenharia de *software*, é preciso examinar as características que determinam que seja algo diferente das outras coisas que os seres humanos constroem. Como trata-se de uma construção lógica e não física, não há um objeto palpável sendo desenvolvido e observado. Por isso, possui características bem diferentes das encontradas em um *hardware*, por exemplo.

Na citação a seguir, PRESSMAN (1995) ressalta que, apesar de assemelhar-se com o processo de projeto de um produto, na fabricação de um *hardware*, por exemplo, a abordagem é - e deve ser tratada como - bem diferente do desenvolvimento de um *software*.

> "O *software* é desenvolvido ou projetado por engenharia, não manufaturado no sentido clássico. Não obstante existam algumas semelhanças entre o desenvolvimento de *software* e a manufatura de *hardware*, as duas atividades são fundamentalmente diferentes. Em ambas atividades, a alta qualidade é obtida mediante um bom projeto, mas a fase de manufatura do *hardware* pode introduzir problemas de qualidade que inexistem (ou são facilmente corrigidos) para o *software*. Ambas atividades dependem de pessoas, mas a relação entre as pessoas envolvidas e o

> trabalho executado é inteiramente diferente. Ambas atividades exigem a construção de um 'produto', mas as abordagens são diferentes. Os Custos do *software* estão concentrados no trabalho de engenharia. Isso significa que os projetos de *software* não podem ser geridos como se fossem projetos de manufatura". (PRESSMAN, 1995).

SOMMERVILLE (2007) reforça o conceito de que, apesar de não ter que se ater às leis da física, o projeto de *software* pode se tornar muito complexo:

> "A engenharia de *software* é um ramo da engenharia cujo foco é o desenvolvimento dentro de custos adequados de sistemas de *software* de alta qualidade. *Software* é abstrato e intangível. Não é limitado por materiais ou controlado por leis da física ou por processos de manufatura. De alguma maneira, isso simplifica a engenharia de *software*, pois não existem limitações físicas no potencial de *software*. Contudo, a falta de restrições naturais significa que o *software* pode facilmente se tornar extremamente complexo e, portanto, muito difícil de ser compreendido" (SOMMERVILLE, 2007).

O *software* tornou-se mais complexo à medida que a tecnologia permitia a inovação da capacidade de processamento e armazenamento de informações do *hardware*. Da mesma forma, as interfaces com o usuários tornaram-se mais "ricas" e aumentavam o grau de interação. Na figura 2.1. abaixo, é possível observar essa evolução:

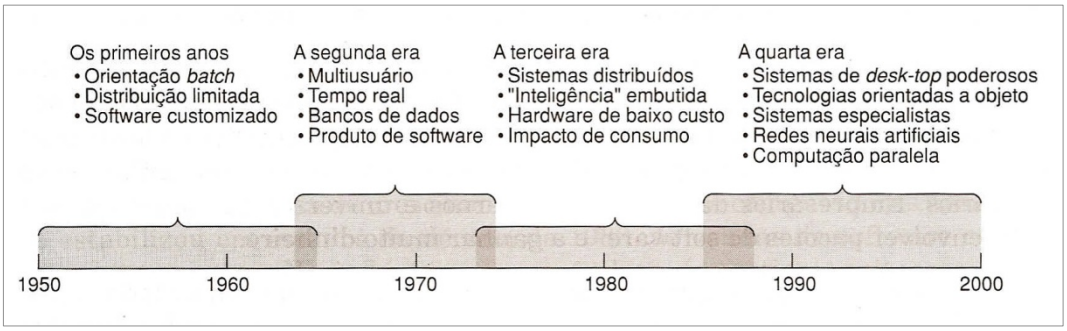

Figura 2.1. - Evolução do software. Fonte: PRESSMAN (1995).

Segundo PRESSMAN (1995), o *hardware* sofreu contínuas mudanças, enquanto o *software* era visto por muitos como uma reflexão posterior. Em meados da década de 1960 até o final da década de 1970, a multiprogramação e os sistemas multiusuários introduziram novos conceitos de interação homem-máquina. Os avanços da armazenagem *on-line* levaram à primeira geração de sistemas de gerenciamento de bancos de dados e criação das "*software houses*" (empresas de desenvolvimento de *software*). Então, surgiu o conceito de manutenção de *software*.

Segundo SOMMERVILLE (2007), em 1968 houve uma conferência para discutir a "crise do *software*". Devido à evolução da tecnologia utilizada no *hardware* para a adoção de circuitos integrados, os custos começaram a cair, enquanto que os projetos importantes de *software* apresentavam, algumas vezes, um atraso de anos pela sua construção ser baseada em um processo informal. Esse processo informal fazia com que houvesse grandes atrasos na entrega e aumento nos custos. O *software* estava em crise, pois não era confiável, tinha desempenho insatisfatório, além de ter uma manutenção bastante complicada.

A partir dessa crise, o conceito de engenharia de *software* ganhou destaque, devido à sugestão de uma organização no seu desenvolvimento ao invés de um processo informal:

> "Novas técnicas e métodos eram necessários para controlar a complexidade inerente aos grandes sistemas de *software*. Essas técnicas tomaram-se parte da engenharia de *software* e são amplamente usadas hoje em dia. No entanto, assim como aumentou a habilidade de produzir *software*, cresceu também a necessidade por sistemas de *software* mais complexos. Novas tecnologias resultantes da convergência de computadores e sistemas de comunicação, e as complexas interfaces com o usuário, impuseram novos desafios aos engenheiros de *software*" (SOMMERVILLE, 2007).

Para PRESSMAN (1995), "crise do *software*" não é o termo adequado para o momento vivido na época e sim "aflição crônica", terminologia que foi sugerida pelo Professor Daniel TEICHROEW, da Universidade de Michigan, numa palestra apresentada em Genebra, Suíça, em abril de 1989:

> "Qualquer um que procure a palavra 'crise' no dicionário, encontrará outra definição: 'o ponto decisivo no curso de uma doença, quando se torna claro se o paciente viverá ou morrerá'. Essa definição nos dá uma pista sobre a verdadeira natureza dos problemas que têm importunado o desenvolvimento de *software*. Ainda temos de chegar ao estágio de crise no *software* de computador. O que realmente temos é uma aflição crônica. A palavra 'aflição' é definida no *Webster's Dictionary* como 'algo que causa dor ou sofrimento'. Mas a definição do adjetivo 'crônica' é a chave de nossa argumentação: 'que dura um longo tempo ou retorna freqüentemente; que continua indefinidamente'. Ela é bem mais precisa para descrever o que enfrentamos nas últimas três décadas como uma aflição crônica e não como uma crise. Não existem curas milagrosas, mas há muitas maneiras pelas quais podemos reduzir a dor, enquanto lutamos para descobrir a cura" (PRESSMAN, 1995).

Independente do termo utilizado por cada autor, tanto PRESSMAN (1995) quanto SOMMERVILLE (2007) alegam que o problema não está no avanço da tecnologia do *hardware* e sim na forma como o *software* é desenvolvido:

> "Quer o chamemos de crise de *software* ou aflição de *software*, o termo alude a um conjunto de problemas que são encontrados no desenvolvimento de *software* de computador. Os problemas não se limitam a *software* que 'não funciona adequadamente'. Ao contrário, a aflição abrange problemas associados a como desenvolvemos o *software*, como mantemos um volume crescente de *software* existente e como podemos esperar acompanhar a crescente demanda por mais *software*" (PRESSMAN, 1995).

De meados da década de 1970 até hoje, os sistemas passaram a permitir a utilização por múltiplos computadores e, por isso, aumentaram a complexidade. Além disso, houve o surgimento das redes globais e locais, as comunicações digitais e o aumento da demanda de acesso "instantâneo" a dados. O uso de microprocessadores, computadores pessoais, gerou um amplo conjunto de produtos inteligentes - de automóveis a fornos de microondas, etc.

Em meados da década de 1980, a taxa de crescimento das vendas de computadores pessoais se estabilizou e as vendas de *software* continuaram a crescer. A quarta era do *software* de computador está apenas começando. As tecnologias orientadas a objetos estão rapidamente ocupando o lugar das abordagens mais convencionais para o desenvolvimento de *software* em muitas áreas de aplicação (PRESSMAN, 1995).

Desde a crise do *software*, em 1968, que o conceito de engenharia de *software* ganhou destaque quando propôs uma organização para o processo informal de desenvolvimento. A partir disso, surgiram os métodos de desenvolvimento mais estruturados e as diversas abordagens que são utilizadas até hoje.

2.3. Métodos, modelos e paradigmas de desenvolvimento de *software*

Para PRESSMAN (1995), a engenharia de *software* compreende um conjunto de etapas que envolve métodos, ferramentas e procedimentos. Em uma equipe de desenvolvimento de *software*, estes são os três elementos fundamentais que possibilitam, ao gerente, um controle maior do processo de desenvolvimento do *software* e, ao programador, o oferecimento de uma base

para a construção de alta qualidade e aumento da produtividade. Podemos compreender métodos, ferramentas e procedimentos como:

- **Métodos:** fornecem os detalhes da forma de construção, envolvendo um grande número de tarefas como: planejamento e estimativa de projeto, análise de requisitos, definição da estrutura de dados, arquitetura do programa, algoritmo de processamento, codificação, teste e manutenção. Além disso, os métodos introduzem um conjunto de critérios para a qualidade do *software*.
- **Ferramentas:** proporcionam apoio automatizado aos métodos de forma a melhorar a qualidade e a ajudar na otimização do tempo de desenvolvimento e teste.
- **Procedimentos:** são as ligações entre os métodos e as ferramentas. Eles são os responsáveis por definir a sequência em que os métodos serão aplicados, os produtos que devem ser entregues em cada etapa, os controles que ajudam a garantir a qualidade, a gerenciar as mudanças e que possibilitam aos gerentes a avaliarem o andamento da construção do *software*.

SOMMERVILLE (2007) cita o processo de *software* como um conjunto de atividades e resultados associados que produz um produto de *software*. Segundo ele, existem quatro atividades fundamentais comuns a todos os processos de *software*:

- **Especificação de *software*:** clientes e engenheiros definem o *software* a ser produzido e as restrições para a sua operação.
- **Desenvolvimento de *software*:** o *software* é projetado e programado.
- **Validação de *software*:** o *software* é verificado para garantir que está de acordo com o que o cliente deseja.
- **Evolução de *software*:** o *software* é modificado para se adaptar às mudanças dos requisitos do cliente e do mercado.

Para SOMMERVILLE (2007), além do processo de desenvolvimento do *software* em si, há também uma visão macro destes processos que inclui as atividades, os produtos e os papéis das pessoas envolvidas na engenharia de *software*. Esta visão deve ser muito clara, independente do método utilizado para desenvolver o produto. Desta forma, é possível obter os seguintes tipos de visão dos modelos de processo:

- ***Workflow***: mostra a sequência de atividades ao longo do processo, com suas entradas, saídas e dependências entre elas. As atividades representam ações humanas.
- **Fluxo de dados ou atividade**: representa o processo como um conjunto de atividades, no qual cada atividade realiza alguma transformação de dados. Mostra como a entrada do processo, como uma especificação, por exemplo, é transformada em uma saída, como um projeto. As atividades, nesse caso, podem representar transformações realizadas por pessoas ou por computadores.
- **Papel/ação**: representa os papéis das pessoas envolvidas no processo de *software* e as atividades pelas quais são responsáveis.

A partir do momento que se tem a visão macro do processo quanto às atividades, papeis e produtos que serão realizados, assim como o domínio de quais ferramentas serão utilizadas e os procedimentos para execução, é preciso escolher o método de desenvolvimento do *software*.

Para cada tipo de produto de *software* que será desenvolvido, há uma abordagem diferente quanto à escolha do método mais adequado. Tais métodos são denominados como modelos ou paradigmas de desenvolvimento de *softwares*:

- **O Ciclo de vida clássico ou modelo em cascata:** durante muitos anos, foi o modelo mais utilizado pela indústria de *software*. Segundo PRESSMAN (1995), requer uma abordagem sistemática e seqüencial. Modelado em função do ciclo da engenharia convencional, o paradigma do ciclo de vida abrange as atividades de análise e engenharia de sistemas, análise de requisitos de *software*, projeto, codificação, testes e manutenção. Para SOMMERVILLE (2007), este modelo considera as atividades apresentadas anteriormente e as representa como fases separadas do processo, como especificação de requisitos, projeto de *software*, implementação, teste e assim por diante. Depois que cada estágio é concluído, ele é aprovado e o desenvolvimento prossegue para o estágio seguinte. Este modelo será detalhado mais adiante, neste mesmo capítulo.
- **Prototipação:** PRESSMAN (1995) relata que, muitas vezes, o cliente define um conjunto de objetivos gerais para o *software*, mas não identifica os requisitos de entrada, processamento e saída de forma detalhada. Em outros casos, o desenvolvedor pode não ter certeza da eficiência de um

algoritmo, da adaptabilidade de um sistema operacional ou da forma que a interação homem-máquina deve assumir. Nessas, e em muitas outras situações, uma abordagem de prototipação à engenharia de *software* pode representar a melhor forma de iniciar o desenvolvimento.

- **Técnicas de quarta geração:** PRESSMAN (1995) descreve o termo "técnicas de quarta geração" (4GT) como algo que abrange um amplo conjunto de ferramentas de *software* que possibilita que o desenvolvedor especifique alguma característica do *software* em um nível elevado. A ferramenta gera, automaticamente, o código-fonte, tendo como base a especificação do desenvolvedor. Quanto mais alto o nível em que o *software* pode ser especificado a uma máquina, mais rapidamente o programa pode ser construído. O paradigma 4GT da engenharia de *software* concentra-se na capacidade de se especificar *software* a uma máquina em um nível que esteja próximo à linguagem natural ou de se usar uma notação que comunique uma função significativa.
- **Desenvolvimento evolucionário:** segundo SOMMERVILLE (2007), esta abordagem intercala as atividades de especificação, desenvolvimento e validação. Um sistema inicial é desenvolvido rapidamente com base em especificações muito abstratas. Então é refinado com as informações do cliente, para produzir um sistema que satisfaça as necessidades dele. O sistema pode, então, ser entregue. Como alternativa, ele pode ser reimplementado, utilizando uma abordagem mais estruturada, para produzir um sistema mais robusto e mais fácil de ser mantido.
- **Engenharia de *software* baseada em componentes:** para SOMMERVILLE (2007), esta técnica supõe que partes do sistema já existam. O processo de desenvolvimento concentra-se mais na integração dessas partes do que no seu desenvolvimento a partir do início.
- **RUP (*Rational Unified Process*)**: SOMMERVILLE (2007) descreve RUP como um modelo de processo moderno que foi derivado do trabalho sobre a UML e do Processo Unificado de Desenvolvimento de *Software* Associado (RUMBAUGH et al., 1999b). Ele traz elementos de todos os modelos genéricos de processo, apóia a iteração e ilustra boas práticas de especificação e projeto. Reconhece que os modelos convencionais apresentam uma visão única de processo. Por outro lado, geralmente, é descrito a partir de três perspectivas: (1) uma perspectiva dinâmica, que mostra as fases do modelo ao longo do tempo. (2) Uma perspectiva

> estática, que mostra as atividades realizadas no processo. (3) Uma perspectiva prática, que sugere as boas práticas a serem usadas durante o processo. Além das perspectivas, é constituído por quatro fases: Concepção, elaboração, construção e transição. No entanto, ao contrário do modelo em cascata, no qual as fases coincidem com as atividades do processo, as fases do RUP estão relacionadas mais estritamente aos negócios do que aos assuntos técnicos.

Para SOMMERVILLE (2007), além dos modelos de desenvolvimento, as iterações do processo são muito importantes à medida que a mudança é inevitável em todos os projetos de grande porte. Os requisitos de sistema não são fixos, eles mudam de acordo com a empresa e as pressões do mercado. As prioridades de negócio mudam, assim como as tecnologias evoluem. Isto significa que o processo de *software* não é executado uma única vez, pelo contrário, as atividades são repetidas regularmente à medida que o sistema é retrabalhado, em resposta às solicitações de mudança.

SOMMERVILLE (2007) também afirma que o desenvolvimento iterativo é tão fundamental ao *software* devido à sustentação destas mudanças. É justamente a adoção deste formato de desenvolvimento que deu início à metodologia ágil, que veremos mais adiante neste capítulo. Apesar das vantagens deste tipo de desenvolvimento, há questões muito profundas em termos de necessidade de mudança na forma dos contratos que devem ser considerados na adoção deste tipo de desenvolvimento:

> "A essência dos processos iterativos é que a especificação é desenvolvida conjuntamente com o *software*. No entanto, isso conflita com o modelo de aquisição de várias organizações, em que a especificação completa é parte do contrato de desenvolvimento do sistema. Na abordagem incremental, não existe uma especificação completa de *software*, até que o incremento final seja especificado. Isso requer um novo formato de contrato, aos quais grandes clientes, tais como os órgãos governamentais, podem encontrar dificuldades para se adequar" (SOMMERVILLE, 2007).

Ainda de acordo com SOMMERVILLE (2007), há dois modelos de processo projetados para apoiar a iteração de processo:

- **Entrega incremental:** A especificação, o projeto e a implementação do *software* são divididos em incrementos desenvolvidos um de cada vez. O modelo em cascata prevê que, tanto os clientes quanto os projetistas se comprometam com um conjunto de requisitos antes do início do projeto. Esses requisitos não poderão sofrer alterações até a entrega do projeto.

Já, na abordagem evolucionária, é permitido que o fechamento dos requisitos seja postergado, o que pode levar a um *software* mal estruturado, pela falta de uma linha de corte. A entrega incremental é uma abordagem intermediária, que combina as vantagens do modelo em cascata e do evolucionário. Desta forma, o cliente identifica, em linhas gerais, os serviços a serem fornecidos pelo sistema e sinaliza quais são mais e menos importantes. Assim, um número de incrementos de entrega é definido, com cada incremento fornecendo um conjunto das funcionalidades do sistema. A partir daí, os requisitos destes primeiros incrementos são definidos para que seja iniciada a construção. Os requisitos dos incrementos posteriores podem ser analisados, mas somente serão fechados quando estes forem desenvolvidos. Já para o desenvolvimento do incremento atual, não há a possibilidade de mudança nos requisitos. Assim que o cliente achar que há um número adequado de incrementos para disponibilizar uma primeira versão do produto, isso pode ser feito sem haver a necessidade de espera dos próximos incrementos. Com isso, o cliente pode experimentar o produto e verificar como deve conduzir a priorização de funcionalidades nos próximos ciclos de desenvolvimento. A figura 2.2. a seguir ilustra o modelo incremental:

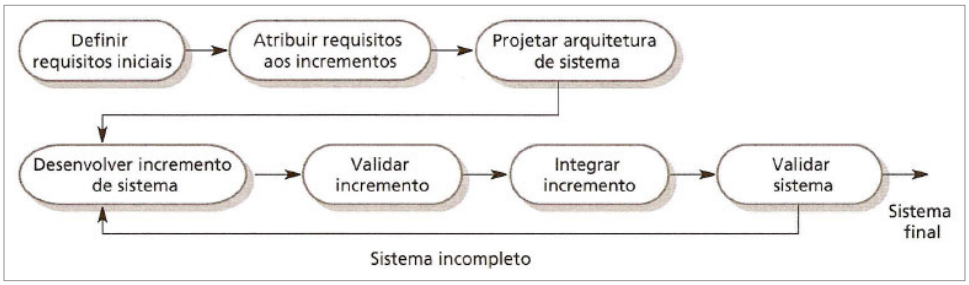

Figura 2.2. - Entrega incremental. Fonte: SOMMERVILLE (2007).

- **Desenvolvimento espiral**: SOMMERVILLE (2007) explica que o desenvolvimento do sistema evolui em espiral para fora, a partir de um esboço inicial até o sistema final. O modelo em espiral do processo de *software* foi originalmente proposto por BOEHM (1988), conforme figura 2.3. Ao invés de representar o processo como uma sequência, apresenta em forma de espiral, onde cada etapa é representada por uma volta e cada volta é dividida em quatro setores: definição de objetivos, avaliação e redução de riscos, desenvolvimento e validação e planejamento. Para PRESSMAN (1995), o modelo espiral para a engenharia de *software* foi desenvolvido para abranger as melhores características, tanto do ciclo de

vida clássico quanto da prototipação, acrescentando, ao mesmo tempo, um novo elemento que faltava a esses paradigmas: a análise dos riscos. SOMMERVILLE (2007) relata que a principal diferença para os outros modelos é o reconhecimento explícito do risco no modelo em espiral. Os riscos podem causar problemas no projeto, assim como ultrapassar o cronograma e os custos. Por isso, a minimização dos riscos é uma atividade de gerenciamento de projeto muito importante.

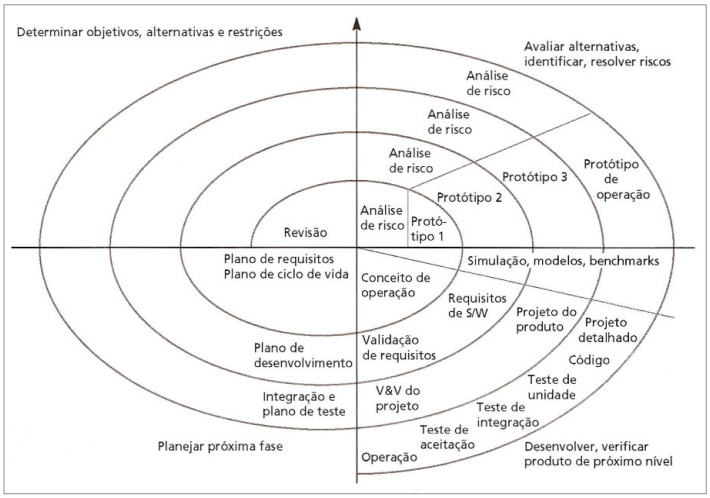

Figura 2.3. - Modelo em espiral do processo de software de Boehm. Fonte: IEEE (1988) apud SOMERVILLE (2007).

Um paradigma ou modelo de engenharia de *software* é escolhido tomando-se como base a natureza do projeto, os métodos, as ferramentas a serem utilizadas, os produtos que precisam ser entregues e o cenário em que a empresa dona do produto está inserida. Diferentes tipos de sistemas necessitam de diferentes processos de desenvolvimento:

> "Não existe um processo ideal, e várias organizações desenvolveram abordagens inteiramente diferentes para o desenvolvimento de *software*. Os processos evoluíram para explorar as capacidades das pessoas em uma organização e as características específicas dos sistemas que estão sendo desenvolvidos. No caso de alguns sistemas, como os sistemas críticos, é necessário um processo de desenvolvimento muito estruturado. Nos sistemas de negócios, com requisitos que mudam rapidamente, um processo flexível e ágil é provavelmente mais eficaz" (SOMMERVILLE, 2007).

O ciclo de vida clássico é o paradigma mais antigo e o mais amplamente usado da engenharia de *software*. Porém, no decorrer da última década, as críticas a este paradigma fizeram com que até mesmo seus ativos defensores questionassem sua aplicabilidade em todas as situações (PRESSMAN, 1995).

De acordo com SOMMERVILLE (2007), o desenvolvimento incremental e iterativo é o mais indicado para negócios que precisam responder rapidamente às mudanças do mercado e cujos requisitos são alterados constantemente:

> "Os negócios atualmente operam em um ambiente global sujeito a rápidas mudanças. Eles têm de responder a novas oportunidades e mercados, mudanças de condições econômicas e ao surgimento de produtos e serviços concorrentes. O *software* é parte de quase todas as operações de negócio e, assim, é essencial que um novo *software* seja desenvolvido rapidamente, para aproveitar as novas oportunidades e responder às pressões competitivas. Desenvolvimento e entrega rápidas são, portanto, muitas vezes o requisito mais crítico para sistemas de *software*" (SOMMERVILLE, 2007).

Apesar dos problemas, PRESSMAN (1995) também relata:

> "O paradigma do ciclo de vida clássico ainda tem um lugar definido e importante no trabalho da engenharia de *software*. Ele produz um padrão no qual os métodos para análise, projeto, codificação, testes e manutenção podem ser colocados. Além disso, as etapas do paradigma do ciclo de vida clássico são muito semelhantes às etapas genéricas, que são aplicáveis a todos os paradigmas de engenharia de *software*. O ciclo de vida clássico continua sendo o modelo procedimental mais amplamente usado pela engenharia de *software*. Embora tenha fragilidades, ele é significativamente melhor do que uma abordagem casual ao desenvolvimento de *software*"

Como esta pesquisa aborda a realização de avaliações com usuários em um ambiente de desenvolvimento ágil de produtos de interfaces gráficas digitais, entende-se que é necessário um aprofundamento deste modelo iterativo e incremental de desenvolvimento. No entanto, antes também é necessário detalhar o modelo em cascata, por este ainda ser um modelo de desenvolvimento de *software* muito utilizado atualmente. Além disso, as vantagens e desvantagens da adoção de cada modelo são elencadas a seguir.

2.3.1. Método clássico - em cascata

"Ciclo de vida clássico de um sistema" foi o termo designado para o primeiro modelo de processo de desenvolvimento de *software* publicado, que originou-se de processos mais gerais de engenharia de sistema (ROYCE, 1970, apud SOMMERVEILLE, 2007).

O termo *waterfall* (cascata) também foi atribuído a este método clássico, devido a sua proposta de desenvolvimento de projeto ser encadeada, na qual é

preciso terminar uma etapa para começar outra. Trata-se de um modelo de desenvolvimento de *software* seqüencial e sistemático, no qual o desenvolvimento é visto como um fluir constante para frente, como uma cascata.

Segundo NIELSEN e NODDER (2008), esta abordagem sequencial (em oposição a iterativo ou incremental) foi projetada para assegurar que nenhuma ponta solta seja deixada antes que o projeto passe para a próxima fase. Na teoria, fazer todo o design antes do desenvolvimento minimizaria a necessidade de retrabalho mais tarde.

Mesmo com algumas diferenças de nomenclatura, praticamente todos os autores consideram cinco fases para o método em cascata. A maior diferença encontrada pela proponente da pesquisa foi o fato de PRESSMAN (1995) considerar seis fases, ao invés de cinco, pois o autor identifica duas fases iniciais: “análise e engenharia de sistemas” e “análise de requisitos de software”. No entanto, para SOMMERVILLE (2007) e outros autores, existe apenas uma fase inicial de levantamento de requisitos chamada de “análise e definição de requisitos”.

Apesar da separação em cinco ou seis etapas, todos citam o mesmo encadeamento. Para as demais fases, os dois autores citam nomenclaturas muito parecidas. Considera-se as cinco fases (ou etapas) mais comuns, conforme a figura 2.4.:

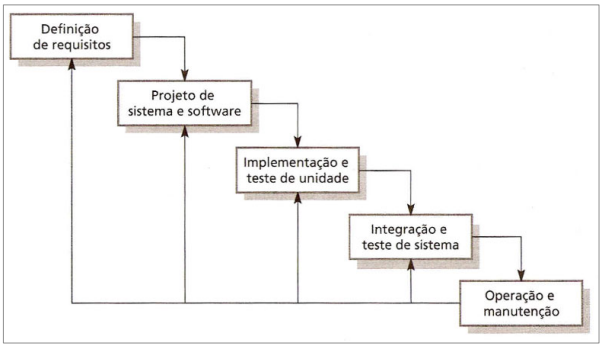

Figura 2.4. - O ciclo de vida do *software* - método em cascata. Fonte: SOMMERVILLE (2007).

- **Análise e definição de requisitos**: envolve a coleta dos requisitos do sistema e dos usuários que utilizarão o sistema, a listagem e entendimento dos serviços, restrições e objetivos do sistema, que devem ser definidos detalhadamente e servirem como uma especificação para o projeto.
- **Projeto de sistema e *software***: estabelece-se uma arquitetura geral do sistema que envolve a identificação e a descrição da estrutura de dados,

detalhes procedimentais, relacionamentos e caracterização da interface. Como os requisitos, o projeto é documentado e torna-se parte da configuração do *software*.

- **Implementação e teste de unidade**: durante essa etapa, o projeto de *software* é realizado como um conjunto de programas, que são codificados em linguagem que a máquina entenda. O teste unitário faz parte do processo de codificar onde é preciso testar o que foi feito para saber se está funcionando adequadamente ao esperado por aquele sistema e, então, seguir com a codificação.
- **Integração e teste de sistema**: assim que todo o código foi gerado, inicia-se a realização de testes para verificar se os aspectos funcionais estão de acordo com os requisitos e sem erros. Após os testes, o *software* é liberado para o cliente.
- **Operação e manutenção**: após o *software* ser instalado e colocado em operação, o processo de manutenção começa. Envolve a correção de erros não detectados nos estágios anteriores e a adaptação à medida que novos requisitos são identificados. A manutenção reaplica cada uma das etapas do método em cascata a um *software* existente, ao invés de um novo.

Apesar do resultado de cada fase ser uma aprovação para o início da fase seguinte, na prática das grandes empresas, esse modelo não funcionava de forma tão linear como se esperava, devido aos problemas que aconteciam em uma fase estarem relacionados à fase anterior. Este fato causava retrabalho e atraso na fase atual:

> "Devido aos custos de produção e aprovação de documentos, as iterações são onerosas e envolvem um 'retrabalho' significativo. Portanto, após um pequeno número de iterações, é normal suspender partes do desenvolvimento, como a especificação, e prosseguir com os estágios posteriores do desenvolvimento. Os problemas são resolvidos em um outro momento, ignorados ou reprogramados. O congelamento prematuro de requisitos pode significar que o sistema não fará o que o usuário deseja. Isso pode também levar a sistemas mal estruturados, pois os problemas de projeto foram contornados por meio de artifícios de implementação" (SOMMERVILLE, 2007).

Apesar do modelo clássico de desenvolvimento ser o mais antigo e amplamente utilizado pelo mercado, em meados da década de 90, grandes discussões foram iniciadas sobre em quais situações este método era realmente

eficaz e em quais ele poderia não funcionar tão bem. Pode-se, então, elencar algumas vantagens e desvantagens da adoção deste modelo:

VANTAGENS:

- Abordagem melhor do que um processo de desenvolvimento informal e de estrutura relativamente fácil de aprendizado na transição de um processo para o outro;
- Por ter etapas muito parecidas com as genéricas de qualquer modelo, facilita a adaptação de qualquer tipo de equipe e sistema desenvolvido em outro modelo;
- Documentação abrangente e mais detalhada do que em outras abordagens. Ideal para sistemas mais complexos;
- Indicada para projetos cujos requisitos de sistema não precisam ser alterados durante o desenvolvimento.

DESVANTAGENS:

- Falta de agilidade no processo, devido ao encadeamento das fases;
- Documentação excessiva que pode ficar desatualizada e ser utilizada por poucos. Muito tempo envolvido em documentação;
- Pouca garantia se todo o escopo do projeto realmente será entregue no final, uma vez que o cenário pode mudar e erros cometidos numa fase podem atrasar o andamento das demais;
- Muitos atrasos em prazos e aumento considerável dos custos devido a erros não imaginados e necessidade de mudanças não detectadas previamente;
- Os requisitos devem ser fechados com uma antecedência grande, na qual muitas vezes o cliente ainda não tem a certeza necessária, o que pode gerar incertezas futuras;
- O cliente só poderá utilizar o produto muito tempo depois de ter contratado o serviço e, por isso, se quiser mudar algo, será tarde para reagir às mudanças do mercado.

Estes problemas ocorrem porque, na metodologia em cascata, não existe espaço para mudanças, principalmente no caso dos requisitos funcionais ou de

produto. Todas as informações do projeto são levantadas no início do processo e estes requisitos se tornam estáticos, intocáveis, até a entrega final do produto, o que, inclusive, pode levar muito tempo.

O mercado é dinâmico, principalmente no mundo da tecnologia. Com o modelo em cascata, as empresas passaram a enfrentar grandes problemas, pois a demanda por mudanças nos requisitos ao longo do processo é algo inevitável. Pelo fato do modelo ser linear, não era possível refazer tudo toda vez que se desejasse alterar algo no projeto.

PRESSMAN (1995) destaca que existe um mito a respeito das mudanças nos requisitos. Muitos clientes imaginam que essas mudanças podem ser acomodadas com facilidade, por acharem que o desenvolvimento de *software* é flexível. A realidade é que qualquer mudança nos requisitos em uma fase mais avançada pode causar atrasos e problemas no desenvolvimento do *software*, principalmente se forem fases encadeadas, como no modelo em cascata.

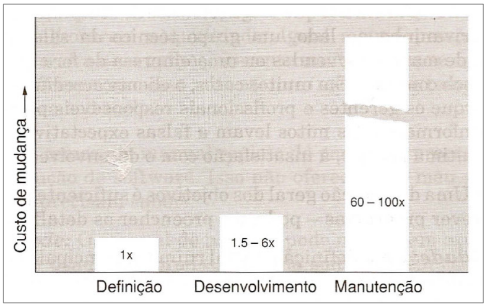

Figura 2.5. - O impacto da mudança de requisitos e o custo por fase. Fonte: PRESSMAN (1995).

O impacto da mudança varia de acordo com o momento em que ela é solicitada. Se for na fase de codificação, pode chegar a um aumento de custo de um e meio a seis vezes. Já na fase de manutenção, na qual o *software* já está pronto, fica inviável, pois os custos podem ser na ordem de sessenta a cem vezes o custo desta fase.

> "A mudança pode causar subelevações que exijam recursos adicionais e grandes modificações de projeto, isto é, custo adicional. Mudanças na função, desempenho, interfaces e outras características durante a implementação (código e teste), exercem um forte impacto sobre o custo. Uma mudança, quando solicitada tardiamente num projeto, pode ser maior do que a ordem de magnitude mais dispendiosa da mesma mudança solicitada nas fases iniciais" (PRESSMAN, 1995).

O STANDISH GROUP realizou uma pesquisa em 1994, que causou uma revolução no mercado de TI com a apresentação, em 1995, do "relatório do

caos" (conhecido em inglês como *CHAOS Report*). Através de entrevistas e grupos de foco com trezentos e sessenta e cinco gerentes e executivos de TI, que representavam oito mil trezentos e oitenta projetos desenvolvidos através da metodologia em cascata, chegou-se aos seguintes dados (alarmantes):

- Somente 16,2% foram considerados projetos bem-sucedidos;
- 31% dos projetos foram cancelados antes de estarem completos;
- 52,7% foram entregues, porém com prazos e custos maiores ou com menos funcionalidades do que especificado nas fases iniciais.

Dentre os projetos que não foram finalizados de acordo com os prazos e custos especificados, a média de custo foi de 189% a mais do que o previsto. A média de atrasos foi de 222%, sendo de 230% para grandes empresas, 202% para empresas de médio e pequeno porte. As principais razões destas falhas estavam relacionadas com o modelo clássico. Assim, a recomendação é que o desenvolvimento de *software* seja baseado em modelos incrementais, o que pode evitar muitas das falhas reportadas.

Por conta dos diversos resultados similares aos descritos acima, surgiu a necessidade de repensar o modelo clássico e produzir um novo método de trabalho, mais adequado a uma época de grandes avanços tecnológicos, um tempo de disseminação do conhecimento e de acesso a Internet.

A partir dessa demanda do mercado, por um modelo mais adequado às mudanças que acontecem ao longo do desenvolvimento de um projeto, ao mesmo tempo, capaz de fornecer uma maior autonomia e suporte às equipes de desenvolvimento, na década de 1990 começaram a aparecer alguns métodos que sugeriam uma abordagem de desenvolvimento ágil.

2.3.2. Métodos ágeis

> "A insatisfação com essas abordagens pesadas levou um número de desenvolvedores de *software*, na década de 1990, a propor novos métodos ágeis. Estes permitiam que a equipe de desenvolvimento se concentrasse somente no *software*, em vez de em seu projeto e documentação. Geralmente, os métodos ágeis contam com uma abordagem iterativa para especificação, desenvolvimento e entrega de *software*, e foram criados, principalmente, para apoiar o desenvolvimento de aplicações de negócios nas quais os requisitos de sistema mudam rapidamente durante o processo de desenvolvimento. Eles destinam-se a entregar um *software* de trabalho rapidamente aos clientes, que podem propor novos requisitos e alterações a serem incluídos nas iterações posteriores do sistema" (SOMMERVILLE, 2007).

A grande diferença entre a metodologia clássica e a metodologia ágil é o fato de que, na metodologia ágil, ao invés de se pensar no projeto completo, pensa-se somente em pequenas partes que serão desenvolvidas naquele *sprint* (período de, aproximadamente, duas a quatro semanas). O que não será desenvolvido no *sprint* não é detalhado. Ao construir cada uma destas partes, passa-se por todas as etapas de desenvolvimento em um curto período de tempo.

Ao invés de ser linear, não existe uma ordem tão rígida de etapas. Apenas as regras da metodologia ágil são seguidas. Desta forma, em pouco tempo é possível ver um produto pronto, mesmo que seja uma parte dele. Porém o conceito de "pronto" é bem diferente. Trata-se de um modelo iterativo e incremental.

Para SY (2007), a figura 2.6. serve como uma boa referência para ilustrar as diferenças entre a metodologia *waterfall* (cascata) e a metodologia *agile* (ágil). No primeiro caso (cascata), desenvolve-se o projeto todo de uma vez, seguindo uma seqüência de etapas (análise, design, codificação e teste), através de um processo que pode demorar meses, ou anos, até a entrega do produto final. No segundo (método ágil), apenas uma parte do produto é desenvolvida em um curto espaço de tempo. Para cada parte desenvolvida, passa-se por todas as etapas de projeto (análise, design, codificação e teste), entregando uma parte pronta do produto. A cada ciclo, novos *releases* (versões) desse produto são entregues.

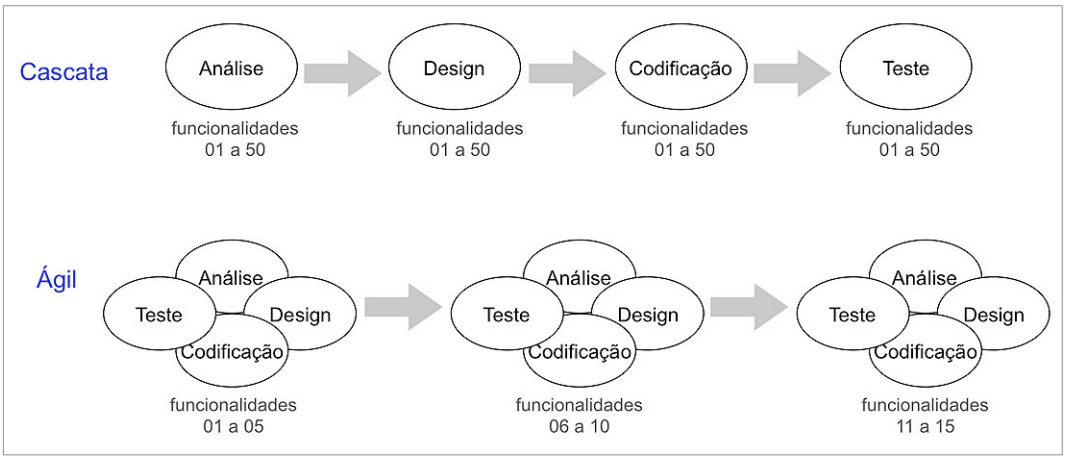

Figura 2.6. - Comparação entre Cascata e Ágil por SY, (2007). Tradução por Christiane Melcher.

Apesar de terem sido apresentados ao mercado na década de 1990, o termo "metodologias ágeis" tornou-se mais popular em 2001, quando dezessete

especialistas em processos de desenvolvimento de *software* estabeleceram princípios comuns compartilhados por todos os métodos. A partir disso, foi criada a Aliança Ágil, que estabeleceu o "manifesto ágil" através de conceitos que serviriam como base para qualquer método ágil:

- Indivíduos e interações ao invés de processos e ferramentas;
- *Software* executável ao invés de documentação;
- Colaboração do cliente ao invés de negociação de contratos;
- Respostas rápidas a mudanças ao invés de seguir planos.

Segundo COCKBURN (2001), para facilitar o entendimento dos conceitos que devem ser seguidos ao adotar metodologias ágeis, a Aliança Ágil criou mais doze princípios que serviram como um detalhamento do manifesto:

1. A prioridade é satisfazer o cliente através de entregas contínuas e frequentes, de *software* de valor;
2. Entregar *softwares* em funcionamento com frequência de algumas semanas ou meses, sempre na menor escala de tempo possível;
3. Ter o *software* funcionando é a melhor medida de progresso;
4. Receber bem as mudanças de requisitos, mesmo em uma fase avançada, dando aos clientes vantagens competitivas;
5. As equipes de negócio e de desenvolvimento devem trabalhar juntas, diariamente, durante todo o projeto;
6. Manter uma equipe motivada fornecendo ambiente, apoio e confiança necessário para a realização do trabalho;
7. A maneira mais eficiente da informação circular dentro da equipe é através de uma conversa face a face;
8. As melhores arquiteturas, requisitos e projetos provêm de equipes organizadas;
9. Atenção contínua à excelência técnica e um bom projeto aumentam a agilidade;
10. Processos ágeis promovem o desenvolvimento sustentável. Todos envolvidos devem ser capazes de manter um ritmo de desenvolvimento constante;
11. Simplicidade é essencial;
12. Em intervalos regulares, a equipe deve refletir sobre como se tornar mais eficaz e, então, se ajustar e adaptar seu comportamento.

Como todo modelo de desenvolvimento de *software*, existem vantagens e desvantagens na utilização dos métodos ágeis. Da mesma forma podem ser mais indicados para alguns tipos de projeto e menos para outros:

VANTAGENS:

- Não há espera grande até a entrega de parte do sistema;
- É possível utilizar os incrementos iniciais como protótipos e ganhar experiência para, só então, estabelecer o que deve ser desenvolvido nos próximos ciclos;
- Risco menor de falha geral do projeto, uma vez que a cada etapa há testes e incrementos que forçam a revisão;
- Serviços mais importantes de sistema recebem mais testes, pois são entregues primeiro;
- Os clientes recebem uma parte do produto pronto em menos tempo, o que ajuda na definição de novos requisitos;
- Pode-se obter uma impressão inicial do produto por quem o utilizará em menos tempo, já que é possível lançar versões menores no mercado;
- As empresas que precisam lançar produtos rapidamente no mercado, para responder à concorrência ou sair na frente em termos de inovação, conseguem um maior controle do desenvolvimento, de forma a priorizar o que faz sentido em termos estratégicos ou até mesmo acertar o rumo, no caso de alguma mudança repentina.

DESVANTAGENS:

- Produtos que são desenvolvidos por diversas equipes podem ter problemas, uma vez que a metodologia ágil prega o conceito que deve haver somente uma equipe multidisciplinar responsável pelo projeto;
- Equipes que precisam dar manutenção em sistemas que foram desenvolvidos por terceiros podem ter problemas, devido à limitação e diminuição da documentação;
- Para os profissionais de criação, torna-se uma tarefa complicada o projeto de somente parte do produto que será desenvolvida naquele ciclo, pela falta de visão e detalhamento do produto todo.

- Devido à impossibilidade de aprofundamento do que será desenvolvido à frente, podem haver problemas em contratar terceiros e até em estabelecer contratos com clientes.

O desenvolvimento ágil engloba um número significativo de metodologias. Entre elas, as mais utilizadas são:

- **XP ou Programação eXtrema (*eXtreme Programming*)**: baseado em técnicas de engenharia para melhorar a qualidade da programação.
- ***Scrum*:** mais focado no processo de desenvolvimento do produto do que em técnicas de programação. Propõe recomeçar um ciclo em vez de tentar corrigir o problema.

Ambas as metodologias possuem técnicas similares para dividir o *software* e é possível utilizar as técnicas de programação do XP em produtos desenvolvidos no *Scrum*.

O Scrum surgiu como um estilo de gerenciamento de projetos em empresas de fabricação de automóveis e produtos de consumo que pretendiam lançar produtos em menos tempo, por isso começaram a produzir de uma forma mais ágil.

Em 1986, Takeuchi e Nonaka publicaram um livro explicando que eles notaram que projetos usando equipes de desenvolvimento, pequenas e multidisciplinares, produziram os melhores resultados. Os autores associaram estas equipes altamente eficazes à formação do *Scrum* no jogo de *Rugby*.

Jeff SUTHERLAND, John SCUMNIOTALES e Jeff MCKENNA documentaram, conceberam e implementaram o *Scrum* na empresa *Easel Corporation* em 1993, incorporando estilos de gerenciamento observados por TAKEUCHI e NONAKA. Em 1995, Ken SCHWABER formalizou a definição de *Scrum* e ajudou a implantá-lo na prática de desenvolvimento de *software* em todo o mundo.

Segundo SCHWABER e SUTHERLAND (2011), *Scrum* é um *framework*[2] para desenvolver e manter produtos complexos. Através dele, pessoas podem tratar e resolver problemas complexos, de forma produtiva e criativa, e entregar

[2] ***Framework***: pode ser conceitual ou de *software*. O empregado pelos autores para explicar o que é *Scrum* se refere ao conceitual, ou seja, um conjunto de conceitos ou diretrizes para resolver o problema de um domínio. Pode-se entender também o uso da palavra como uma espécie de estrutura processual.

produtos com o mais alto valor possível. Os autores ressaltam que *Scrum* é leve, simples de entender e extremamente difícil de dominar. É um modelo de trabalho estrutural, que está sendo usado para gerenciar o desenvolvimento de produtos complexos desde o início de 1990. Não é um processo ou uma técnica para construir produtos, ao invés disso, é uma forma de empregar vários processos ou técnicas. O *Scrum* deixa claro a eficácia relativa das práticas de gerenciamento e desenvolvimento de produtos, dando a possibilidade de melhorá-las.

SCHWABER e SUTHERLAND (2011) ressaltam que o *Scrum* é fundamentado nas teorias empíricas de controle de processo, ou empirismo, que afirma que o conhecimento vem da experiência e da tomada de decisões baseadas no que é conhecido. O *Scrum* emprega uma abordagem iterativa e incremental para aperfeiçoar a previsibilidade e o controle de riscos. Assim, três pilares servem como base para a sua implementação:

- **Transparência:** aspectos significativos do processo devem estar visíveis para os responsáveis pelos resultados. Transparência requer que aqueles aspectos sejam definidos por um padrão comum, para que os observadores compartilhem um mesmo entendimento do que está sendo visto.
- **Inspeção:** as pessoas do time devem, frequentemente, inspecionar os artefatos do *Scrum* e o progresso, com objetivo de detectar variações não desejadas. Esta inspeção não deve ser tão frequente a ponto de atrapalhar a própria execução das tarefas. As inspeções são mais benéficas quando desempenhadas por inspetores habilidosos no local do trabalho.
- **Adaptação:** se um inspetor determina que um ou mais aspectos do processo desviou para fora dos limites aceitáveis, o resultado do produto não é aceitável, portanto, o processo ou o material que está sendo produzido deve ser ajustado. Um ajuste deve ser feito tão logo quanto possível, para minimizar os desvios futuros. O *Scrum* prescreve quatro oportunidades formais para inspeção e adaptação:
  - Reunião de planejamento do *Sprint* (*planning*);
  - Reunião diária (*daily scrum* ou *daily meeting*);
  - Reunião de revisão do *Sprint* (*review*);
  - Retrospectiva da *Sprint*.

SCHWABER e SUTHERLAND (2011) explicam que *Scrum* consiste em equipes, ou times, associadas a seus papéis, eventos, artefatos e regras. Cada componente dentro do *framework* serve a um propósito específico e é essencial para o uso e o sucesso do *Scrum*.

Para SCHWABER e SUTHERLAND (2011), o coração do *Scrum* é o *Sprint*, um *time-box* (período fixo) de um mês, ou menos, durante o qual uma versão potencialmente utilizável de um incremento do produto é criada. *Sprints* tem durações consistentes ao longo do esforço de desenvolvimento. Um novo *Sprint* começa imediatamente depois da conclusão do *Sprint* anterior. *Sprints* consistem em uma reunião de planejamento do *Sprint* (*planning*), reuniões diárias, o trabalho de desenvolvimento, a reunião de revisão para fechamento do *Sprint* e a retrospectiva de como foi o *Sprint*.

Como o *Scrum* é formado por um conjunto de regras, etapas, papéis e atividades, a proponente da pesquisa desenvolveu a figura 2.7. a seguir, com o objetivo de ajudar a esclarecer a dinâmica desta metodologia ágil. Apesar da possibilidade de utilizar o *Scrum* para desenvolver qualquer tipo de produto, com a intenção de manter a proximidade com o objeto desta pesquisa de mestrado, optou-se por descrever o *Scrum* como se fosse utilizado para desenvolver um produto de mídia digital, como um *website*, por exemplo.

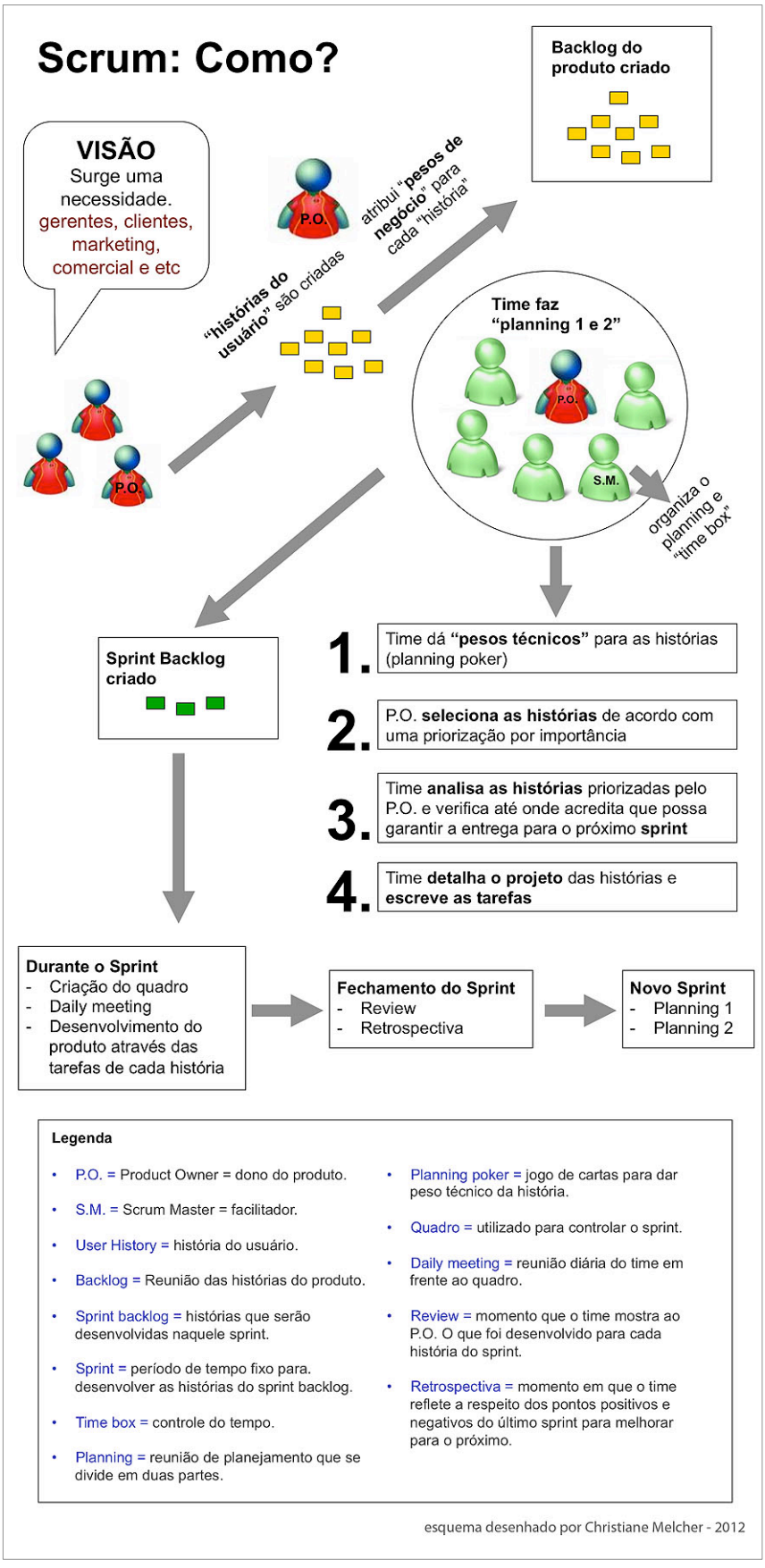

Figura 2.7. - Esquema para explicar as etapas e atividades que envolvem o *Scrum*. Desenvolvido por Christiane Melcher (2012).

Descreve-se brevemente, nos tópicos abaixo, o esquema da figura 2.7. que retrata a dinâmica do *Scrum*:

- Tudo começa com a visão de um produto novo, seja vinda da área de marketing, negócios, de clientes internos ou externos. Portanto, há uma demanda a ser desenvolvida com a metodologia ágil *Scrum*. Então, um time é formado, ou aproveita-se um time que já existe na empresa. A ideia é fazer com que a empresa sempre trabalhe com a mesma metodologia e que já tenha os times disponíveis. Desta forma, o desenvolvimento do produto é delegado para o time que ficará responsável pelo projeto.
- Após a definição do produto, o *Product Owner* deve escrever as histórias do usuário (*user stories*), que representam todos os requisitos e funcionalidades que o produto deve ter. Essas histórias são agrupadas em uma lista, que é chamada de *Backlog* (que é constantemente alimentado por novas histórias, que surgem ao longo dos ciclos de desenvolvimento do produto). Além disso, o P.O. deve atribuir um peso de negócio para cada uma destas histórias, com o objetivo de ajudar o time (e a ele mesmo) a diferenciar quando uma história é importante e outra é secundária. Geralmente, as histórias mais importantes, que têm valor de negócio maior, são as primeiras a entrar nos ciclos de desenvolvimento.
- Uma vez formado o *Backlog*, começa o planejamento (*Planning*) do primeiro *Sprint.* Isso acontece, geralmente, em uma reunião com o time inteiro, durante um período que, normalmente, não deve demorar muito mais do que um ou dois dias, tempo recomendado para o caso de empresas que trabalham com um *Sprint* de duas semanas, em média (um *Sprint* deve durar de duas semanas até um mês). É muito importante que, uma vez estabelecido o tempo do *Sprint*, este período não seja alterado, permitindo que o time possa estimar a dificuldade técnica da produção das histórias de forma cada vez mais eficaz e eficiente, pois o aprendizado de um *Sprint* serve como base para melhorar a estimativa do próximo. Com o passar do tempo, o time também consegue estimar melhor quantos pontos, relativos à dificuldade técnica para o desenvolvimento de uma história, ele consegue entregar a cada novo *Sprint*. Desta forma, o time é capaz de estabelecer metas para melhorar o seu desempenho, além de ter mais segurança para fazer suas estimativas.

- O *Planning* começa com uma rodada de *Planning Poker*, ou seja, uma espécie de baralho com números, que cada integrante do time (com exceção do *Product Owner* e do *Scrum Master*) recebe para "votar" na complexidade técnica de uma história. O *Product Owner* apresenta algumas histórias, consideradas mais importantes para desenvolvimento no próximo *Sprint*. O time conversa sobre a solução técnica e começa a sessão de *Planning Poker*. Cada pessoa apresenta uma carta com a pontuação que acredita que aquela história deva ter, de acordo com a sua complexidade de desenvolvimento. Então, as pessoas do time que "votaram" na maior e na menor pontuação apresentam seus argumentos, explicando o motivo de terem escolhido aquele número e como imaginam a solução. Essa etapa serve para que todos tenham em mente o caminho e as dificuldades que podem enfrentar, além de alinhar as expectativas de cada integrante da equipe. O time joga as cartas novamente por três vezes, até chegar a uma pontuação de complexidade técnica em comum.
- Há situações em que uma história é muito complexa para ser desenvolvida em apenas um *Sprint*. Neste caso, o time apresenta a carta que corresponde à eliminação dessa história na rodada, obrigando o *Product Owner* a reescrever essa história, de forma a separa-la em histórias menores. O time decide a linha de corte, ou seja, quais histórias os integrantes da equipe se comprometem a entregar no final do tempo determinado para o *Sprint*. Desta maneira, forma-se o *Sprint Backlog*, que nada mais é do que as histórias que serão desenvolvidas naquele período.
- Uma vez que os pontos de complexidade técnica para o desenvolvimento de cada história estejam acordados entre todos do time, o *Product Owner* determina uma ordem de preferência para o desenvolvimento dessas histórias.
- O time passa a detalhar como será a solução de projeto para desenvolver cada história. Pode-se utilizar *sketches* (para mais informações sobre *sketches*, consulte o capítulo 3), que são rascunhos de tela desenhados no papel, permitindo que todos tenham o mesmo entendimento das soluções de projeto propostas. Nesta etapa, as regras de negócio e os aspectos sobre a implementação das soluções de projeto também são detalhadas.
- A partir daí, encerra-se a etapa de *Planning*, dando início à etapa dois, onde o time utiliza *post-its* para descrever as tarefas que precisa executar

para concluir cada história. No *Scrum*, o conceito de "concluído", normalmente, aplica-se a um *software* executável e pronto para lançamento. Mas, o *Product Owner* também pode utilizar alguns *Sprints* para desenvolver um conjunto de funcionalidades para, somente depois, lançar uma primeira versão do produto (essa primeira versão é resultado da reunião do conjunto de funcionalidades que foram desenvolvidas ao longo de alguns *Sprints*). O *Product Owner* é quem decide quando o produto será lançado (e o que será lançado).

- Após descrever todas as tarefas que precisa executar em cada história, o time monta o quadro de acompanhamento, conforme a figura 2.8. Nesse quadro, os *post-its* de cada tarefa são agrupados em uma primeira coluna, classificada como *Sprint Backlog*. Quando uma tarefa é iniciada, o *post-it* muda para a segunda coluna, classificada como *In Progress*. No momento em que uma tarefa é finalizada, o *post-it* muda para a terceira coluna, classificada como *Done*. Além dos *post-its*, também é possível incluir um *Burndown* no quadro de acompanhamento. Esse gráfico é utilizado para mostrar a entrega das tarefas no *Sprint* e uma lista de impedimentos do time ou da organização. Impedimentos são acontecimentos que prejudicam ou impedem o desenvolvimento de alguma tarefa ou história. Se o impedimento for relacionado ao time, a equipe deve se empenhar, junto com o *Scrum Master*, para resolvê-lo. Se o impedimento for da organização, o mesmo pode ser resolvido pelo *Scrum Master* ou pelo *Product Owner*. Caso esse impedimento não seja resolvido, o *Sprint* atual deve ser cancelado e o time deve iniciar o planejamento de um novo *Sprint*. Neste caso, inicia-se todo o processo novamente, pois nunca deve-se aproveitar o tempo de um *Sprint* em outro. O período de duração do *Sprint* deve ser sempre fixo.
- Ao longo de todos os dias do *Sprint*, o time deve se reunir em frente ao quadro de acompanhamento para realizar o *Daily Meeting*, ou *Daily Scrum*, que é um encontro de curta duração, sem ultrapassar mais do que quinze minutos. Durante o *Daily Meeting*, cada pessoa do time escolhe uma tarefa, que deve ser executada em um dia ou menos. Se o período de duração for maior do que isso, a tarefa deve ser quebrada em mais de uma parte. No segundo dia de *Daily Meeting*, cada pessoa relata se terminou a tarefa escolhida no dia anterior. Caso não tenha conseguido fazer isso, ela explica o motivo. Caso a tarefa escolhida esteja concluída, o integrante da equipe escolhe uma nova tarefa e move

o *post-it* correspondente a essa nova tarefa para a coluna *In Progress*. Esse procedimento se repete, diariamente, até o final do *Sprint*.

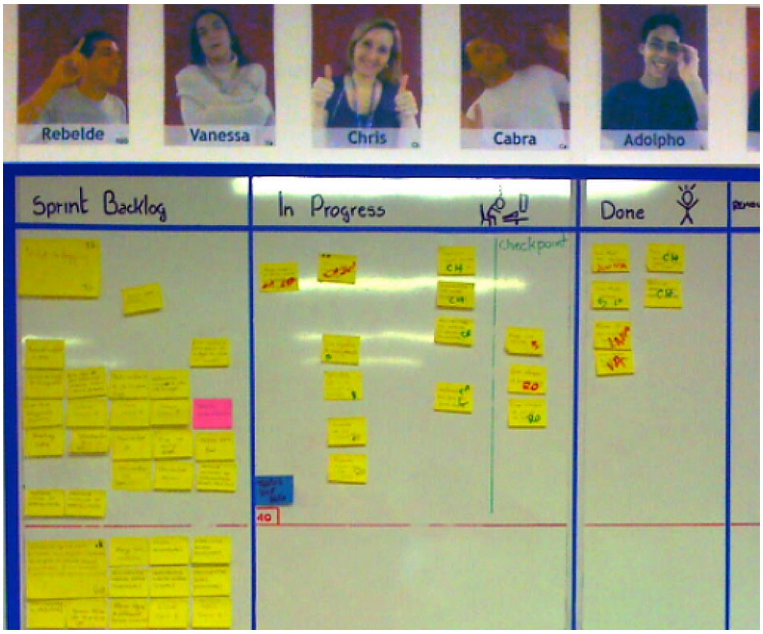

Figura 2.8. - Quadro de acompanhamento de um time de *Scrum* na Globo.com, em 2009. Foto: Christiane Melcher.

- O evento que conclui o *Sprint* é a revisão do produto (*review*), onde o time apresenta, para o *Product Owner*, tudo o que foi desenvolvido para finalizar uma história. Em alguns casos, essa apresentação também pode ser realizada durante o *Sprint*, caso o *Product Owner* tenha a intenção de colocar alguma funcionalidade no ar antes do término desse *Sprint*. Ao terminar uma história, o time inteiro deve repetir o mesmo procedimento e apresentar tudo o que foi desenvolvido para concluir a segunda história da fila do *Sprint Backlog*, e assim sucessivamente, até terminar a lista.
- Após o *Product Owner* dar a sua opinião sobre o que foi desenvolvido, considerando, ou não, todas as histórias como prontas, encerra-se o *Sprint*. O *Scrum* determina que a equipe sempre entregue um *software* de qualidade, ou parte de um *software*, no final de cada ciclo de desenvolvimento. Caso essa situação não aconteça, deve ser tratada como exceção, fazendo o time refletir sobre os problemas ocorridos.
- Após o encerramento do *Sprint*, o próximo passo é a retrospectiva, uma reunião com o objetivo de causar uma reflexão no time sobre o que foi positivo e o que pode melhorar no processo de trabalho. Dessa forma, todos os integrantes da equipe apresentam suas opiniões e criam uma lista, indicando o que devem continuar fazendo e o que devem modificar

para que o próximo *Sprint* seja ainda melhor. Como é possível perceber, a filosofia do *Scrum* é pautada na comunicação direta entre as pessoas.

- Uma vez encerrando o *Sprint*, inicia-se outro *Planning*, repetindo todos os procedimentos descritos anteriormente. Não há limite de ciclos, ou seja, deve-se realizar quantos *Sprints* forem necessários para a entrega do produto final.

Descreve-se, de forma geral, o papel de alguns membros do time:

- **P.O. - *Product Owner* ou Dono do Produto:** se a metodologia antiga for utilizada para efeitos de comparação, é possível dizer que o *Product Owner* tem atribuições de cliente do time e também de gerente. Mas, diferente do cenário antigo, se o time afirmar que só é capaz de concluir duas histórias no final do *Sprint*, o *Product Owner* tem a obrigação de aceitar esse fato, mesmo que o seu desejo seja diferente disso (como, por exemplo, querer que o time entregue quatro histórias). O *Product Owner* também é responsável por escrever histórias, casos de aceitação e priorizar o que deve ser desenvolvido no próximo *Sprint*. Resolver impedimentos da organização também é sua função, além de negociar com clientes internos e externos.
- **S.M. - Scrum Master ou Facilitador:** o *Scrum Master* não é um coordenador, pois o time não necessita da sua permissão para executar suas tarefas. O *Scrum Master* está ali para facilitar o trabalho do time, organizar e proteger das interferências, sejam de outros times, gerentes ou do próprio *Product Owner*. Portanto, ele age como uma espécie de defensor das regras do *Scrum*, sendo responsável também por resolver impedimentos da organização e tentar melhorar a comunicação do time. O *Scrum Master* participa do *Planning* e procura ouvir os problemas descritos na retrospectiva, para ajudar a melhorar as condições para o próximo *sprint*.
- **Time ou Equipe de Desenvolvimento:** deve ser multidisciplinar, composto por desenvolvedores, profissionais que projetam a experiência do usuário (UX - *User eXperience*), como arquitetos de informação e designers de interação, um *Scrum Master* (S.M.) e um *Product Owner* (P.O.). Apesar de existir a indicação para o time ser multidisciplinar, não há uma obrigatoriedade desse time contar com todos esses perfis de profissionais. Porém, é importante ter o número adequado de pessoas, permitindo que o time tenha autonomia para tomar todas as decisões

necessárias para o desenvolvimento do produto. O *Product Owner* e o *Scrum Master* definem os membros do time, procurando selecionar pessoas capazes de fazer uma entrega ágil e ao mesmo tempo com qualidade. Recomenda-se entre três e nove pessoas, sem contar o *Scrum Master* e o *Product Owner*. O time é responsável por determinar pesos técnicos, resolver impedimentos que dizem respeito às tarefas do próprio time e, principalmente, executar as tarefas do *Sprint*. O time se compromete a entregar o que foi acordado no *Planning*, definindo a solução técnica mais adequada para ser implementada. O *Product Owner* explica a visão de negócio que ele pretende incorporar ao produto, mas a solução é de responsabilidade do time.

Após a descrição da dinâmica do *Scrum*, vale ressaltar uma de suas maiores vantagens: como é possível determinar um número fixo de dias para um *Sprint* e, com o passar do tempo, o time começa a ter ideia de quantos pontos de complexidade técnica, em média, ele consegue entregar em cada *Sprint*, somado ao fato de que, com o passar do tempo, o *Product Owner* pode pedir para o time fazer algumas rodadas adiantadas de *Planning Poker* para ter uma ideia da pontuação das histórias do *Backlog*, é possível ter uma noção mais clara de quantos *Sprints* o time precisa para concluir uma versão inicial do produto, que é composta por um conjunto de histórias. Além disso, caso o cliente queira alguma mudança no direcionamento do produto, o *Product Owner* consegue, rapidamente, trocar as histórias que ele priorizou o desenvolvimento no próximo *Sprint*. Desta forma, muitas empresas grandes da área de mídias digitais conseguiram justificar a adoção desta metodologia ágil no projeto dos seus produtos.

Apesar das vantagens da adoção desta metodologia, restam algumas lacunas sobre a forma de trabalho do profissional de UX - *User eXperience* neste cenário, uma vez que, segundo as regras do *Scrum*, o produto deve ser planejado no *Planning*. Como essa fase de planejamento tem um curto período de duração, e só deve ser detalhado o que será desenvolvido no próximo *Sprint*, pode ser que esse (pouco) tempo não seja adequado para um projeto consistente da experiência de uso do produto. Além disso, o *Scrum* não explica como é possível realizar pesquisas com usuários, abrindo espaço para algumas discussões, que são apresentadas abaixo e também no capítulo 3 desta dissertação de mestrado, cujo tema está relacionado com o projeto e avaliação de usabilidade de interfaces desenvolvidas em métodos ágeis.

2.4. Discussão sobre a mudança: de Cascata para Ágil

Como descrito nos tópicos anteriores, os produtos cujos requisitos não podem ser estáticos, e que estão inseridos em um cenário cuja velocidade de mudança é grande, começaram a buscar novas formas de desenvolvimento que pudessem lhes permitir uma rápida adequação ao mercado. Desta forma, empresas como Apple, Yahoo, Google, Globo.com, UOL, Terra, Nokia e etc., que lidam com tecnologia, inovação e criam produtos para as mídias digitais, adotaram os métodos ágeis de desenvolvimento de *software*, como forma de criar produtos que possam ser projetados em pouco tempo.

O conceito de produto "pronto" mudou. Hoje, os produtos de mídias digitais são lançados como "beta", o que significa que ainda estão em testes e sendo analisados para que melhorias possam ser realizadas. Desta forma, os usuários destes produtos podem testá-los enquanto ainda estão em desenvolvimento, informando suas opiniões às empresas, além de proporem melhorias. Essas empresas, inclusive, podem colocar algo no mercado em bem menos tempo, além de serem beneficiadas por essa proximidade com os consumidores.

A grande questão que ainda deve ser discutida, tanto no mercado quanto na academia, tem relação com o projeto e a avaliação da experiência de uso de um produto que ainda está em desenvolvimento. Como o designer pode projetar a interface, e a interação que o usuário terá com este sistema, se somente uma parte será desenvolvida e detalhada com maior profundidade naquele momento? A chance do produto se parecer com uma colcha de retalhos é grande, à medida que não se tem a visão completa do projeto na hora do seu desenvolvimento.

Também é preciso levar em conta como e quando o designer, enquanto pesquisador, deve recorrer a avaliação com usuários durante a fase de projeto, que, na metodologia ágil, é muito curta e particionada em ciclos. Esta é a questão fundamental desta pesquisa de mestrado, cujas características serão descritas com mais detalhes nos próximos capítulos.