PONTIFÍCIA UNIVERSIDADE CATÓLICA DE SÃO PAULO

Kleber Aoki Gava

Adaptação do Processo OpenUP para o Desenvolvimento de Sistemas Seguros

SÃO PAULO

2009

Kleber Aoki Gava

Adaptação do Processo OpenUP para o Desenvolvimento de Sistemas Seguros

Monografia apresentada como exigência parcial para a conclusão do curso de especialização em Engenharia de Software na Pontifícia Universidade Católica de São Paulo.

SÃO PAULO - 2009

APRECIAÇÃO

Resumo

Características de segurança são muitas vezes negligenciadas no ciclo de vida de desenvolvimento de software. Freqüentemente estas características não são sequer consideradas e, quando o são, tendem a ser ações reativas. Entretanto, assim como a qualidade, é preciso incorporar a segurança em todas as fases de um processo de desenvolvimento. Para isso, existem diversas práticas e técnicas que propõem a inclusão da segurança no software, no entanto, poucos trabalhos abordam estas atividades nos métodos ágeis de desenvolvimento. Neste trabalho será utilizado um destes métodos (OpenUP) como base para enumerar quais os padrões e as melhores práticas de segurança podem ser incorporadas a um processo de desenvolvimento que adota a filosofia ágil.

Abstract

Security features are often neglected on software development lifecycle. Usually they are not considered at all and, when they are, they tend to be reactive actions. As well as quality, the security must be merged in all phases of a development process. To do so, there are several practices and techniques that propose the inclusion of security into the software; however, few works address security in agile methods. In this work one of these methods (OpenUP) is used as a base for enumerating which security patterns and best practices can be incorporated in a development process that embrace the agile philosophy.

A melhor maneira de ficar em segurança é nunca se sentir seguro.

Benjamin Franklin

Agradecimentos

Ao Senhor Deus por tudo.

Ao professor Carlos Eduardo de Barros Paes pelas idéias.

À Clarissa pela paciência.

À FAPESP pela ajuda financeira.

Sumário

1. Introdução

1.1. Segurança no Software

O ser humano do século 21 tem se tornado dependente dos *softwares*. Isso pode ser tanto uma dádiva quanto um grande problema (Kennet et al., 1999). Vários setores da sociedade, públicos ou privados, têm o *software* como uma ferramenta vital para os negócios e para o acesso à informação. Dados críticos são armazenados, transportados e processados utilizando sistemas que podem estar expostos à Internet, e que, conseqüentemente, poderão ser acessados e utilizados por outros sistemas ou pessoas. Não é difícil deduzir que esta exposição de informações importantes é algo alarmante. O uso do *software* se estendeu a níveis antes não imaginados, e tem abarcado as mais diversas funções, aumentando a lista de riscos que antes não eram considerados.

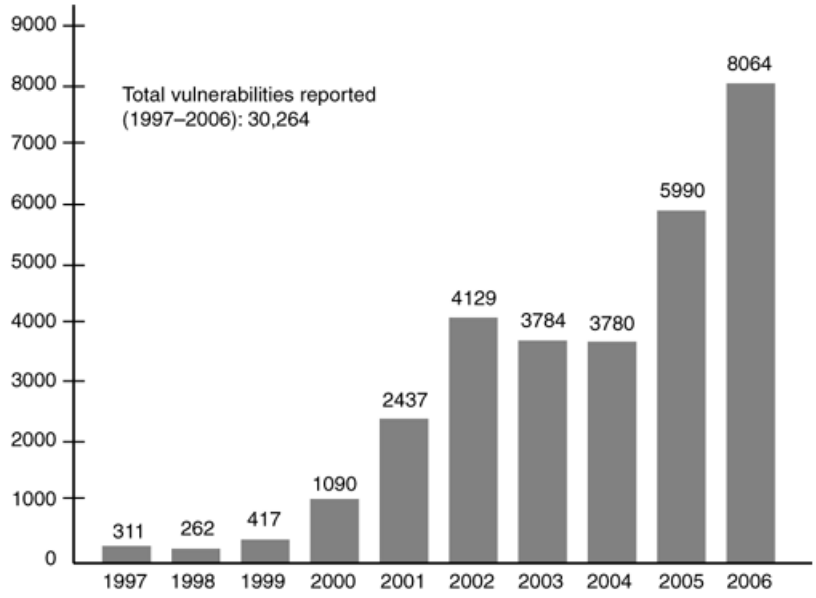

Figura 1.1 - Quantidade de vulnerabilidades reportadas pela CERT/CC (Computer Emergency Readiness Team Coordination Center) no período de 1997 a 2006 (http://www.cert.org).

Historicamente os desenvolvedores de *software* criavam sistemas apenas para uso interno ou *ad-hoc*, e havia pouca ou nenhuma preocupação com exposição dos dados e acesso não autorizado. Mas, o advento das redes de computadores (Internet), os sistemas distribuídos, e, mais recentemente, o conceito de *Cloud Computing*, tem evidenciado a necessidade de se produzir *softwares* mais confiáveis, para que a exposição a ataques e

fraudes seja minimizada. Essa preocupação é fundamentada, pois o número de problemas relacionados com a segurança da informação vem crescendo rapidamente (Figura 1.1). A noção de segurança de *software* sempre existiu, mas apenas recentemente os profissionais vêm estudando sistematicamente como construir *softwares* seguros (McGraw, 2006).

Embora a tecnologia presente proporcione a criação de aplicações que eram consideradas difíceis ou impossíveis anteriormente, ela trouxe uma complexidade muito grande. Atualmente é necessário o conhecimento não apenas de uma linguagem de programação, mas também das tecnologias de suporte, isso cria vários pontos de falha a serem relevados. Além da exposição à Internet e da crescente complexidade dos sistemas, a constante adição de funcionalidades e extensões tornam o *software* gradualmente mais vulnerável, o que representa mais desafios para os engenheiros de *software*. A realidade mostra que quanto mais o *software* cresce em tamanho, complexidade e conectividade, maior é a probabilidade de problemas e vulnerabilidades surgirem (McGraw et al., 2008).

Segurança da informação é uma das preocupações emergentes na comunidade de TI, pois tem se tornado cada vez mais pessoal. Uma vez que o *software* é um componente crítico em muitas aplicações cotidianas, o seu mau uso ou funcionamento inadequado pode trazer potenciais perdas financeiras, vazamento de informações sigilosas ou pessoais, e até mesmo danos materiais. E neste mundo onde o *software* manipula informações preciosas, desenvolvê-lo com qualidade é algo de primordial importância, tanto para os negócios quanto para o próprio ser humano. Conseqüentemente, tão necessário quanto garantir a qualidade, produzir um sistema seguro e capaz de funcionar apropriadamente sob condições inóspitas deve ser um dos principais focos de um projeto de *software*.

O *software* é, atualmente, quase onipresente na sociedade, e isso afeta muitos aspectos da vida de muitas pessoas. Este feito inevitavelmente traz à tona questões sobre segurança de sistemas, e levanta dúvidas sobre se um *software* é seguro, como verificar isso, e quais as implicações das falhas de segurança no *software*. Essas propriedades só podem ser observadas se mensuradas por meio de um processo de desenvolvimento de *software* com ênfase na segurança. No entanto, é muito comum confundir os mecanismos que provêem segurança, com a segurança em si (Mead et al., 2005). Novos equipamentos (como *firewalls*) e novas tecnologias (como criptografias fortes) avançaram consideravelmente nos últimos anos, disponibilizando soluções sofisticadas para tratar do problema; ferramentas de detecção de vírus ou autenticação via *passwords*, por exemplo, não são segurança, mas mecanismos para alcançá-la. Entretanto, apenas estas soluções não são suficientes para garantir um sistema seguro se a camada de aplicação ainda é a mais vulnerável e suscetível a ataques (Lanowitz,

2005, e McGraw et al., 2008).

O desenvolvimento de uma aplicação segura não se inicia pela codificação ou compra de soluções tecnológicas, mas, antes, através da escolha de um processo de desenvolvimento adequado. Pesquisas mostram que as diferenças entre um *software* seguro e um *software* inseguro residem na natureza dos processos e práticas usados para especificar, projetar e codificar o *software* (Goertzel et al., 2006). A segurança, portanto, deve ser vista como um processo, e não como um produto. Ao se iniciar o projeto, porém, isso dificilmente é levado em conta. Inicialmente mais atenção é dada à compreensão do domínio, e a segurança é adiada para a fase de codificação. Isso é perigoso, porque ao ignorá-la nas fases iniciais ficará evidente, nas fases posteriores, que adicionar características de segurança é muito mais difícil (e caro[1]) do que apenas adotar uma solução tecnológica (Yoder, 1998).

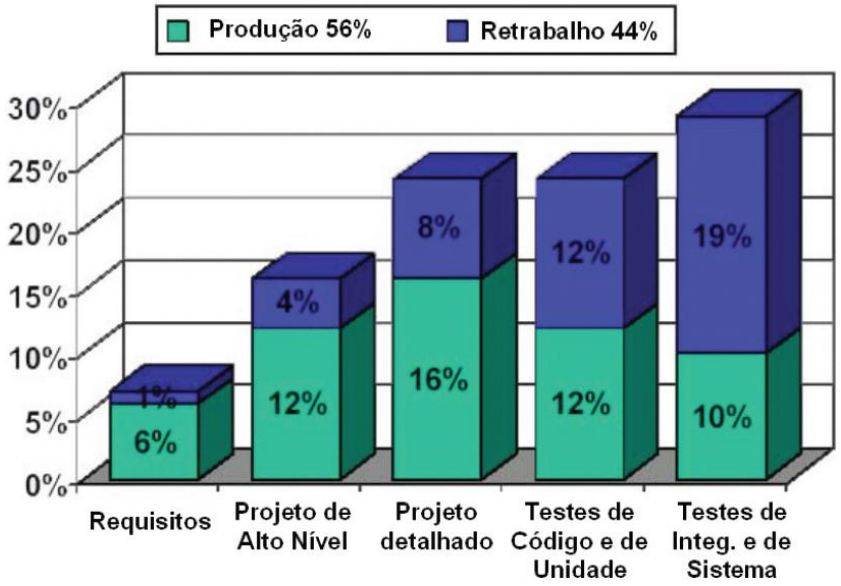

Figura 1.2 - Distribuição do retrabalho pelas atividades de desenvolvimento de software. Adaptado de Wheeler (WHEELER, D.A., BRYKEZYNSKI, B., MEESON, R.N., 1996, Software Inspections: An Industry Best Practice, IEEE Computer Society).

Para que a segurança seja incorporada em um sistema, é necessário adotar uma série de políticas e mecanismos (Paes, 2007). Mas, além disso, é preciso considerá-la em todo o

[1] Segundo Boehm (Boehm, B. W. & Papaccio, P. N. "Understanding and Controlling Software Costs." IEEE Transactions on Software Engineering SE-4, 10 (Oct.1988): 1462-77) encontrar e corrigir um problema de *software* após sua entrega chega a ser 100 vezes mais caro do que encontrar e corrigir um problema durante a fase de requisitos e *design*.

processo de desenvolvimento. O tema é controverso, pois o ciclo de vida tradicional[2] não trata desta disciplina diretamente, e, geralmente, as questões de segurança são relegadas ao segundo plano. Dentre os fatores que usualmente levam a isso se destacam: a inexperiência dos analistas, a falta de uma cultura de segurança, e a escassez de recursos necessários ao desenvolvimento seguro. Normalmente, quando a segurança é considerada dentro do processo, ela se encontra em seções à parte e em uma lista de requisitos genéricos (Mead et al., 2005).

Apesar de ser um dos pilares para construção de um sistema de boa qualidade, a segurança é equivocadamente resumida a soluções tecnológicas. É necessário, no entanto, enfatizá-la em todo o ciclo de vida de desenvolvimento, afinal, as preocupações com a segurança se iniciam logo no levantamento de requisitos. Quando se define segurança dentro do ciclo de vida é preciso compreender o processo de desenvolvimento e as atividades associadas. Em cada uma das etapas, atividades específicas devem ser executadas para garantir que a segurança seja incorporada ao produto em construção (Purcell, 2007). Se estas atividades forem integradas desde cedo no ciclo de vida, decisões posteriores serão feitas baseadas nas necessidades dos envolvidos no projeto, e não como manutenções custosas. A Figura 1.2 ilustra a proporção produção/retrabalho por fases de um projeto de *software*, como se percebe, quase metade do tempo é gasto em retrabalho, que tende a aumentar conforme o projeto avança.

1.2. Motivações para este Trabalho

Atualmente, os padrões e as práticas de desenvolvimento de *software* resolvem de forma limitada a questão da segurança nos processos de desenvolvimento. Organizações e grandes empresas, como ISO, IBM e Microsoft, têm trabalhado em padrões, técnicas e ferramentas que visam o desenvolvimento seguro. Muitas idéias estão amadurecendo, mas na maior parte das vezes a segurança ainda é vista superficialmente ou possui alguma deficiência quando considerada no ciclo de vida de desenvolvimento.

De maneira simplificada, um processo de desenvolvimento é uma formalização das atividades envolvidas no desenvolvimento do *software*. Esta formalização implica em um seqüenciamento destas atividades em etapas distintas e bem definidas. Como existem muitas propostas de processos de desenvolvimento, as atividades mais comumente aceitas foram

[2] Neste contexto, ciclo de vida tradicional é aquele usualmente aceito pelo SWEBOK (SWEBOK, 2004), que, embora bastante semelhante ao *waterfall*, não é o recomendado pelo mesmo.

reunidas em um documento criado sob o patrocínio da IEEE com a finalidade de servir de referência em assuntos considerados, de forma generalizada pela comunidade, como pertinentes a área de Engenharia de *Software*. Este guia é conhecido como *Guide to the Software Engineering Body of Knowledge* (SWEBOK), e tem, dentre outros, o objetivo de caracterizar o conteúdo e classificar em tópicos as áreas de conhecimento da disciplina de Engenharia de *Software*. A segurança, no entanto, é coberta de forma pouco profunda na área de Qualidade de *Software*[3]. O RUP, um processo que é um refinamento do *Unified Process*[4], dispõe o assunto da segurança de maneira genérica, uma vez que o desenvolvimento seguro é abordado através de *plugins* e extensões.

Os padrões SSE-CMM (*Systems Security Engineering Capability Maturity Model*) e o *Common Criteria* (ISO/IEC 15408), apesar de feitos especificamente para tratar da segurança no desenvolvimento de *softwares*, apresentam deficiências. O SSE-CMM é uma extensão do CMM para tratar do desenvolvimento seguro. Este modelo descreve características essenciais para que o processo de Engenharia de Segurança tenha êxito, mas não prescreve um processo em particular, apenas reúne melhores práticas geralmente observadas na indústria. O SSE-CMM também serve como uma métrica padrão para práticas de Engenharia de Segurança, no entanto, o modelo dá pouca ênfase à tecnologia e a parte técnica do processo de desenvolvimento, mas não especifica um processo de *design*. Por sua vez, o *Common Criteria* (ISO/IEC 15408) é um *framework* que fornece a garantia de que o processo de especificação, implementação e avaliação de um produto de *software* seguro tenha sido conduzido de forma rigorosa e padronizada. A norma, porém, é muito genérica, e não provê diretamente uma lista de requisitos de segurança de produtos ou características para classes específicas de produtos, concentrando-se no levantamento e avaliação de requisitos.

Além destes esforços, existem muitos livros didáticos e artigos que analisam pragmaticamente a segurança no desenvolvimento de *software*. Os trabalhos começaram surgir na década de 2000[5], e vêm consolidando termos e conceitos desde então. A literatura é farta, mas vários aspectos ainda não foram abrangidos (McGraw, 2006). Os métodos ágeis, por exemplo, são negligenciados em algumas áreas. Por tratarem de matéria nova, ainda são recorrentes as discussões que envolvam a aplicabilidade e a adequação destes métodos a certos ambientes e domínios (Wäyrynen, 2006). No campo da Engenharia de Segurança tais

[3] *Software Quality*, uma de suas *Knowledge Areas* (KAs)
[4] O *Unified Process* é um *framework* geral descrito por Jacobson, Booch e Rumbaugh (JACOBSON, Ivar; BOOCH, Grady; RUMBAUGH, James. The Unified Software Development Process. Addison-Wesley Professional, 1999. ISBN-10: 0201571692, ISBN-13: 9780201571691).
[5] Os primeiros livros específicos sobre segurança de *software* foram publicados em 2001, Anderson (2001) e Viega (2001).

métodos têm recebido um grande número de críticas. Isso porque, a princípio, segurança e métodos ágeis parecem entrar em conflito.

Os críticos destes métodos alegam que os mesmos não fornecem uma documentação de especificação sólida, e ressaltam a importância de uma maior formalidade para atingir melhores resultados de segurança. Seus defensores rebatem argumentando que os métodos possuem práticas que são extensivamente conhecidas e aplicadas no desenvolvimento de *software* seguro, portanto, benéficos e compatíveis com a segurança (Wäyrynen, 2006). É evidente que, por ser um assunto novo, não há um consenso entre os estudiosos, e as informações disponíveis vêm de conhecimento tácito ou estudos de caso. Há um caminho a percorrer para consolidação destes conhecimentos. De qualquer forma, é preciso verificar a capacidade destes métodos de se adequarem à engenharia de segurança.

1.3. Objetivos

O objetivo principal deste trabalho é abordar o *Open Unified Process*, que será utilizado como base, de uma perspectiva do desenvolvimento seguro. Pretende-se, além de apresentar os vários aspectos diferentes envolvidos no projeto de sistemas seguros, enumerar as técnicas, os processos e os padrões de Engenharia de *Software* que auxiliam os participantes de um projeto a *incorporar* e não *adicionar* a segurança no processo de desenvolvimento. A finalidade é verificar o que é atualmente utilizado e o que é aplicável a cada uma das etapas de um projeto, relacionando os padrões com estas etapas.

O OpenUP foi escolhido por ser uma proposta ágil do *Unified Process*, fundamentado em práticas amplamente aceitas de Engenharia de *Software*, além de possuir documentação acessível e adotar um padrão aberto que é apoiado pela comunidade *open source*. Sua natureza iterativa e incremental é adequada para o controle da qualidade do desenvolvimento, pois a cada iteração as definições e perspectivas de qualidade podem ser refinadas e ajustadas. O OpenUP também não depende de ferramentas, e pode ser usado *as is* ou desdobrado para abarcar diversos tipos de projetos (Balduino).

Por se preocupar com a mitigação de riscos antecipadamente no projeto, incluir a questão da segurança no processo se torna menos intrusivo, uma vez que o processo é configurável, e pode ser adaptado facilmente para as particularidades de um projeto. Destaca-se também sua natureza disciplinada e formal, que reúne o melhor de dois mundos: alguns dos ativos do RUP (papéis, atividades, artefatos, etc) e a busca constante por *software* valoroso e de alta qualidade do desenvolvimento ágil. Como as críticas aos métodos ágeis se voltam para

a falta de documentação e formalidade, o OpenUP supre esta demanda com artefatos bem definidos.

1.3.1. Atividades e Resultados Esperados

Para a realização deste trabalho, serão analisadas práticas relacionadas com a Engenharia de Segurança e os padrões propostos para que suas características sejam embutidas nas etapas do ciclo de vida de desenvolvimento. É importante examinar o que significa ser seguro, e o que os diversos padrões afirmam sobre este aspecto com relação ao processo de desenvolvimento. E tão importante quanto verificar esta definição, é preciso contextualizá-la nos processos de desenvolvimento. Assim, os padrões *Common Criteria* e SSE-CMM servirão como ponto de partida para a listagem dos tópicos a serem verificados.

Além destes padrões, artigos recentes sobre segurança no método ágil serão analisados para observar como os especialistas têm abordado o tópico, o porquê das críticas feitas e como as críticas são tratadas por seus proponentes, utilizando as conclusões destes estudos para fundamentar o trabalho.

Espera-se que a contribuição seja um guia fundamentado nas melhores práticas de Engenharia de *Software* e em modelos bem definidos pelas diversas instituições comerciais e de pesquisa. Importante ressaltar que, apesar de o trabalho se basear no OpenUP, o processo é extensível, e pode ser aplicado a qualquer método ágil.

1.3.2. Escopo

O foco principal deste trabalho é o processo, e não as diversas tecnologias relacionadas. Assim sendo, não fará parte do escopo deste texto questões de configuração de *firewalls*, detecção de intrusos, ou como resistir a ataques de DDoS. Bem como, não serão relacionados os diversos tipos de ameaças e como preveni-las.

É importante diferenciar projeto de *software* seguro e segurança no projeto de *software*. Este trabalho não visa apenas projetos que têm a segurança como principal meta, mas embutir a segurança em qualquer projeto de *software*.

1.4. Organização da Monografia

O objetivo do **Capítulo 2** é apresentar conceitos e princípios para fundamentar a segurança no processo de desenvolvimento. Este capítulo fará uma síntese das idéias mais

importantes que serão estudadas, as motivações para utilização destes conceitos. Neste capítulo será edificado o alicerce sobre o qual o Capítulo 3 será erguido. Três pilares serão construídos: padrões e práticas de qualidade, padrões e práticas de segurança e processo de desenvolvimento e método ágil.

O **Capítulo 3** se inicia com uma discussão sobre a segurança no processo de desenvolvimento, e na seqüência lista as quatro fases do OpenUP (*Inception*, *Elaboration*, *Construction* e *Transition*), para cada uma delas procura obter e explicitar características importantes sobre segurança.

A conclusão do trabalho será dada no **Capítulo 4**, com uma breve discussão do que foi analisado e sugestões para futuros trabalhos.

Finalmente, o **Capítulo 5** apresentará o material consultado na confecção dos textos, contendo o conhecimento de especialistas e estudiosos no assunto.

2. Conceitos Fundamentais

2.1. Relação entre Qualidade, Segurança e Processo

Três conceitos importantes são necessários para o desenvolvimento de uma aplicação segura: qualidade, segurança e processo. Segurança é, geralmente, considerada como um atributo de qualidade, e para que qualidade seja alcançada é necessário adotar atividades e tarefas com objetivos claros e determinados, para isso, um processo adequado deve ser escolhido (Figura 2.1). Somente práticas consistentes e padronizadas suportam a avaliação e o progresso das atividades no ciclo de vida de desenvolvimento. É por meio destas avaliações que se pode afirmar, ou seja, mensurar, se um *software* tem qualidade ou segurança.

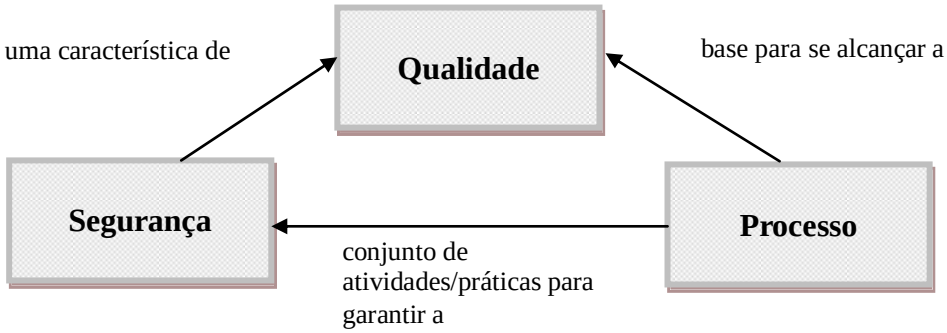

Figura 2.1 – Relação entre qualidade, segurança e processo de desenvolvimento.

O SWEBOK dispõe que os requisitos de *software* definem as características de qualidade requeridas, e influenciam os métodos de medidas e os critérios de aceitação para avaliar estas características. É importante normatizar estes métodos de medidas e definir processos para avaliar as características de qualidade. Há uma diversidade de padrões que propõem modelos de processo para alcançar qualidades específicas, como, por exemplo, a norma ISO/IEC 9126 e o CMMI. Da mesma forma, a segurança deve ser caracterizada e avaliada no ciclo de vida. Assim como para a qualidade, existem padrões e práticas de segurança, como o *Common Criteria* e o SSE-CMM, que sugerem práticas para as fases de um processo. Estes modelos podem ser aplicados em um processo de desenvolvimento, no caso, o OpenUP foi utilizado.

2.2. Qualidade de *Software*

Existem muitas definições propostas para qualidade, mas, para o objetivo deste trabalho será adotada a do padrão ISO 9000, que propõe o seguinte: "é o grau em que um conjunto de características inerentes a um produto, processo ou sistema, cumpre os requisitos inicialmente estipulados para estes." E prossegue afirmando que "a qualidade de alguma coisa pode ser determinada pela comparação de um conjunto de características inerentes com um conjunto de requisitos; se estas características inerentes estão em conformidade com todos os requisitos, então alta ou excelente qualidade é atingida; se estas características não estão em conformidade com todos os requisitos, uma qualidade baixa ou pobre é atingida". Esta definição pode ser estendida ao se afirmar que além da conformidade com os requisitos, devem existir especificações precisas e maneiras de se avaliar um produto (métricas). No desenvolvimento de *software*, a qualidade do produto está diretamente relacionada à qualidade do processo de desenvolvimento, desta forma, é comum que a busca por um *software* (produto) de maior qualidade passe necessariamente por uma melhoria no processo de desenvolvimento.

2.2.1. Padrões de Qualidade

A terminologia para as características de qualidade de *software* difere de um modelo de qualidade para outro, cada modelo possui suas próprias características, métricas e classificações (SWEBOK, 2004). Existem vários modelos de atributos de qualidade de *software*, e podem ser úteis para planejar, discutir e priorizar a qualidade dos produtos de *software*. Dois padrões utilizados para avaliar a qualidade do produto e do processo são: o padrão ISO/IEC 9126, que é uma norma sobre características de qualidade de *software*, e a ISO/IEC 14598, que define um processo de avaliação da qualidade do *software*. Estes dois padrões estão sendo gradualmente substituídos pela norma ISO/IEC 25000:2005 (SQuaRE, ou *Software Product Quality Requirements and Evaluation*), que é uma evolução destas duas. A ISO/IEC 25000 não substitui estas normas, mas reorganiza os materiais destas sem realizar muitas mudanças, assim, a ISO/IEC 9126 continua válida, do mesmo modo que vários aspectos da ISO/IEC 14598 (Koscianski, 2007). Seus documentos possuem um caráter didático, pois há uma grande preocupação em fornecer uma extensa lista de exemplos de métricas. A norma é genérica o bastante para que outros modelos de qualidade elaborados por pesquisadores de maneira pontual possam ser mapeados para o modelo SQuaRE. Por fim, esta norma estabelece uma linguagem ou vocabulário comum e válido internacionalmente

(Koscianski, 2007). O SQuaRE divide os assuntos em cinco tópicos (Tabela 2.1).

Tópico	***Descrição***
Gerenciamento	Documentos voltados para todos os possíveis usuários da norma. É uma introdução geral a todo o conjunto de normas.
Modelo de Qualidade	É o modelo apresentado pela norma ISO/IEC 9126.
Medição	Define o que é uma medição, descrevendo os diversos aspectos relacionados à realização da tarefa; e propõe uma série de métricas que podem ser utilizadas ou adaptadas.
Requisitos de Qualidade	Outro aspecto que veio da norma ISO/IEC 9126. Além de realizar medidas, é preciso que valores-alvo, especificados nos requisitos de software, tenham sido previamente especificados.
Avaliação	Concretiza a norma na realização de uma avaliação de qualidade a partir de medições cujos resultados devem ser confrontados contra um modelo definido pelo usuário.

Tabela 2.1 – Tópicos do SQuaRE (Koscianski, 2007).

O uso de processos bem estabelecidos e boas práticas da Engenharia de *Software* auxiliam tanto na definição das características desejáveis de qualidade quanto no correto processo para desenvolvê-las.

2.2.2. Qualidade no Ciclo de Vida de Desenvolvimento

Segundo o SWEBOK a garantia de qualidade é uma preocupação que deve estar presente em todas as etapas de um projeto de *software*, e abrange muitas de suas *Knowledge Areas* (SWEBOK, 2004). Kenett et al. (1999) e Pressman (2006) complementam afirmando que deve ser uma preocupação de todas as pessoas envolvidas no projeto.

É importante enfatizar que a qualidade deve ser definida no início de um projeto, pois apenas bem compreendida e delimitada ela pode ser alcançada (Kenett et al., 1999). Nos estágios iniciais do desenvolvimento, porém, apenas recursos e processos podem ser mensurados. Quando produtos intermediários ficam disponíveis eles podem ser avaliados pelos níveis das métricas internas escolhidas. Estas, por sua vez, podem ser usadas para se prever valores para as métricas externas.

A gestão da qualidade monitora e gerencia a qualidade do *software* quando este está em desenvolvimento. É sua responsabilidade identificar e sugerir práticas para que o processo seja sempre melhorado. Estas práticas auxiliam no aumento da qualidade do *software*, fornecendo informações gerais para a gerência do projeto, incluindo indicações da qualidade de todo o processo de Engenharia de *Software*. As atividades envolvidas consistem de tarefas e de técnicas para indicar se os planos do *software* estão sendo seguidos e quão bem os produtos intermediários e finais estão atingindo os requisitos especificados. É seu papel também assegurar que os problemas e as necessidades dos *stakeholders* sejam abordados claramente e de maneira precisa, e que os requisitos para a solução sejam definidos, expressados e compreendidos adequadamente.

Na gestão de qualidade há três componentes principais (Pressman, 2006): (1) controle de qualidade; (2) garantia de qualidade; e (3) melhoria da qualidade. O **controle de qualidade** envolve uma série de inspeções, revisões e testes. Para que isso seja possível, é necessário haver especificações definidas e mensuráveis que possam ser utilizadas como parâmetros. A **garantia de qualidade** consiste de um conjunto de funções para auditar e relatar, e são utilizadas para avaliar a efetividade e completeza das atividades de controle de qualidade. Estes dados identificam problemas, e devem ser fornecidos à gerência para que esta resolva as questões de qualidade. Finalmente, a **melhoria da qualidade** se diferencia do controle de qualidade por mudar um processo de maneira proposital (iterativamente) para melhorar a confiabilidade em se alcançar um resultado.

Enquanto o controle de qualidade dá ênfase aos testes e ao bloqueio da liberação de produtos defeituosos, a garantia de qualidade se concentra no melhoramento e estabilização da produção, e nos processos associados para evitar (ou minimizar) características que levam aos defeitos. Ambos são necessários dentro do projeto, uma vez que os parâmetros definidos para qualidade devem ser testados e verificados. Em resumo, o controle de qualidade é um controle de produtos de *software* (verificação e validação), e a garantia de qualidade é um controle de processos, fazendo a conformidade entre o que está sendo feito e os parâmetros normatizados.

2.3. Segurança de *Software*

A segurança de *software* diz respeito ao entendimento dos riscos de segurança induzidos por *software* e como gerenciá-los (McGraw, 2006). O SWEBOK classifica segurança como um requisito não funcional. Requisitos não funcionais são aqueles que agem

como restrições para a solução (SWEBOK, 2004) e muitas vezes são chamados de requisitos de qualidade. Nesta seção será investigado o que significa um *software* ser seguro, e quais as práticas para que a segurança seja alcançada no processo de desenvolvimento. Não há práticas definitivas, e, de fato, segurança é um conjunto de características, melhores práticas e padrões que devem ser executadas para tornar um *software* seguro.

As principais propriedades que tornam um *software* seguro são (Bass, 2003, Dowd, 2006, e McGraw et al., 2008):

- **Confidencialidade**: é a propriedade que protege os dados ou serviços de acesso não autorizado. O *software* deve garantir que qualquer uma de suas características, ativos gerenciados, e/ou conteúdos, sejam escondidos de entidades não autorizadas.
- **Integridade**: propriedade na qual os dados ou serviços são entregues conforme o planejado. O *software* e seus ativos gerenciados devem ser resistentes e resilientes a modificações não autorizadas no código, nos ativos geridos, nas configurações, ou no comportamento de entidades autorizadas, ou quaisquer modificações por entidades não autorizadas. Esta propriedade deve ser preservada tanto no desenvolvimento quanto na execução do *software*.
- **Disponibilidade**: propriedade pela qual o sistema estará disponível para uso legítimo. O *software* deve estar operacional e acessível para seus propósitos para usuários autorizados sempre que requisitado. Da mesma forma, suas funcionalidades e privilégios devem estar inacessíveis para usuários não autorizados todas as vezes.
- **Não-rejeição**: propriedade em que a transação (acesso ao dado ou serviço) não pode ser negada para nenhuma das partes. O *software* não pode refutar ou negar a responsabilidade pelas ações realizadas. Esta propriedade garante que a responsabilidade não possa ser subvertida ou enganada.
- **Garantia**: propriedade em que as partes envolvidas em uma transação são quem eles pretendem ser.
- **Auditoria / responsabilidade**: propriedade em que o sistema rastreia atividades internas em níveis suficientes para reconstruí-las. Todas as ações relacionadas à segurança devem ser registradas, com a atribuição de responsabilidade. O rastreamento deve ser possível enquanto e após a ação registrada ocorrer.

2.3.1. Padrões de Segurança

2.3.1.1. ISO/IEC 15408: Common Criteria

O *Common Criteria* (*Common Criteria for Information Technology Security Evaluation* ou CC) é um padrão internacional (ISO/IEC 15408) voltado para a segurança lógica das aplicações e para o desenvolvimento de aplicações seguras, também utilizado para seleção de métricas apropriadas de TI.

O CC estabelece critérios a serem usados como base para avaliação das propriedades de segurança de sistemas e produtos de TI, chamados de *Target of Evaluation*, ou TOE, que podem ser *hardware*, *software* ou *firmware*. As avaliações da norma são realizadas sobre segurança de produtos e sistemas, desta maneira, é preciso identificar o TOE. Estas avaliações servem para validar as asserções feitas sobre este TOE. Para que seja de uso prático a avaliação deve verificar as características-alvo de segurança. Alguns artefatos são essenciais para este propósito (todos devem ser criados baseados em modelos específicos fornecidos pelo CC):

- ***Protection Profile* (PP)**: é um documento, criado pelos consumidores, que identifica requisitos de segurança para uma classe de dispositivos de segurança (ex: *smart cards* ou *firewalls*) relevantes para um propósito particular dos mesmos. Por ser de propósito mais geral ele é independente de implementação.
- ***Security Target* (ST)**: é um documento que identifica as propriedades de segurança do alvo a ser avaliado. Pode se referir a um ou mais PPs. Por ser mais específico, ele é dependente da implementação.
- ***Security Functional Requirements* (SFR)**: especifica funções individuais de segurança as quais podem ser fornecidas pelo produto. O CC apresenta um catálogo padrão de tais funções. Embora o CC não prescreva qualquer SFR a ser incluso em um ST, ele identifica dependências onde a correta operação de uma função (tal como a habilidade de limitar o acesso de acordo com o papel) é dependente de outra (tal como a habilidade de identificar papéis individuais).

O processo de avaliação também tenta estabelecer o nível de confiança que pode ser colocado nas características de segurança do produto através de processos de garantia de qualidade. Isto é feito pelo *Evaluation Assurance Level* (EAL). O EAL se trata do índice numérico que descreve a profundidade e o rigor de uma avaliação. Cada EAL cobre todo o desenvolvimento de um produto, com um dado nível de exatidão. Os requisitos funcionais e

os requisitos de garantia são a base para o CC. Há sete níveis listados no padrão, com o EAL 1 sendo o mais básico (e portanto o mais barato de se implementar e avaliar) e o EAL 7 sendo o mais rigoroso (e o mais caro). Normalmente, um autor de um ST ou PP não selecionará garantia de requisitos individualmente, mas escolherá um destes pacotes, possivelmente aumentando os requisitos em algumas áreas com requisitos de maior nível. EALs de maior nível não necessariamente implicam em melhor segurança, estes níveis apenas significam que a garantia de segurança reivindicada do TOE foi validada mais extensivamente. Os sete níveis de avaliação (definidos na parte 3 da norma) são[1]:

1. **EAL 1**: **Testado funcionalmente**. Aplicável quando se requer confiança na correta operação de um produto, mas ameaças não são levadas a sério. Uma avaliação neste nível deve fornecer evidência de que o TOE funciona de uma maneira consistente com sua documentação, e que fornece proteção útil contra ameaças identificadas.
2. **EAL 2: Testado estruturalmente**. Aplicável quando desenvolvedores e usuários requerem de baixa a média segurança garantida independentemente, mas o completo registro do desenvolvimento não é disponível imediatamente. Esta situação pode surgir quando há limitado acesso do desenvolvedor ou quando há um esforço para garantir segurança para sistemas legados.
3. **EAL 3: Testado e checado metodicamente**. Aplicável quando desenvolvedores ou usuários requerem um nível moderado de segurança garantida independentemente, e requer uma investigação completa do TOE e seu desenvolvimento, sem re-engenharia substancial.
4. **EAL 4: Testado, projeto e revisado metodicamente**. Aplicável quando desenvolvedores e usuários requerem de moderada a alta segurança garantida independentemente em produtos de comodidade convencional, e estão preparados para incorrer em custos adicionais de engenharia de segurança específica.
5. **EAL 5: Testado e projeto semi-formalmente**. Aplicável quando desenvolvedores e usuários requerem alta segurança garantida independentemente, em um desenvolvimento planejado e requer uma abordagem rigorosa de desenvolvimento que não incorra em custos exorbitantes de técnicas especialistas em engenharia de segurança.
6. **EAL 6: Testado e com *design* verificado semi-formalmente**. Aplicável quando se desenvolve TOEs para aplicações em situações de alto risco onde o valor dos ativos

[1] Uma lista de produtos validados e seus níveis de EAL associados é mantida atualizada em http://niap.nist.gov/cc-scheme/vpl/vpl_type.html.

protegidos justifica custos adicionais.

7. **EAL 7: Testado e com *design* verificado formalmente**. Aplicável para o desenvolvimento de TOEs para aplicações em situações de alto risco extremo, bem como quando o alto valor dos ativos justifica os altos custos.

Cada EAL consiste de um conjunto comum de classes de garantias (*assurances*) que são as mesmas para cada nível, com a diferença de que os requisitos incrementam gradativamente em número a medida que o nível sobe. O objetivo é avaliar até que ponto o processo de desenvolvimento envolve testes, *design*, e verificação de um sistema ou produto, e se isto é feito de maneira funcional, estrutural e/ou formal. Os EALs são, por esta razão, úteis como um complemento aos PAs do SSE-CMM, quando se analisa a extensão a qual um processo de desenvolvimento preenche as atividades de segurança prescritas, para garantir que as necessidades de segurança são satisfeitas.

2.3.1.2. System Security Engineering Capability Maturity Model (SSE-CMM)

O modelo de referência SSE-CMM foi desenvolvido tendo como base o SE-CMM (*System Engineering CMM*). Este modelo, entretanto, aumenta as *Process Areas* (PAs) organizacionais e de projeto do SE-CMM, ao acrescentar PAs de Engenharia de Segurança (Goertzel et al., 2007). Isso visa melhorar e avaliar a maturidade dos processos de Engenharia de Segurança usados para produzir produtos de segurança da informação, sistemas confiáveis, e capacidades de segurança em sistemas de informação. O escopo dos processos abordados pelo SSE-CMM abarca todas as atividades da segurança do ciclo de vida da Engenharia de Segurança do sistema, incluindo definições de conceitos, análises de requisitos, *design*, desenvolvimento de integrações, instalações, operações, etc. O modelo de referência inclui requisitos para desenvolvedores de produtos, desenvolvedores e integradores de sistemas seguros, e organizações que forneçam serviços de segurança de computadores e/ou Engenharia de Segurança de computadores, incluindo organizações comerciais, governamentais, e acadêmicas. Uma tabela com as PAs de Engenharia de Segurança do SSE-CMM pode ser vista no Anexo C.

Um dos conceitos fundamentais do SSE-CMM afirma que, para realizar uma atividade em particular de maneira correta e repetível, certos processos devem estar presentes. Quão bem uma atividade é realizada é dependente, pelo menos em parte, da maturidade do processo usado para realizar a atividade. Entretanto, o como do processo não é crítico, há muitas

formas de atingir um objetivo em particular ou realizar uma atividade em particular. O que é importante são os objetivos da atividade e o que é alcançado. Não importam os detalhes de uma metodologia em particular, mas a maturidade do processo. É esta maturidade que o SSE-CMM tenta mensurar.

2.4. Qualidade VS Segurança

O objetivo de um desenvolvimento seguro tem muitas características em comum com o domínio do desenvolvimento de um *software* com qualidade (Figura 2.5). Desta forma, enumerar as características e as propriedades que permitem desenvolver um *software* com alta qualidade é uma forma de orientar o desenvolvimento com ênfase na segurança. No entanto, nem todas as propriedades que levam à produção de um *software* com alta qualidade produzem um *software* seguro. O processo de qualidade do produto auxilia de forma limitada a introdução de segurança no produto de *software*.

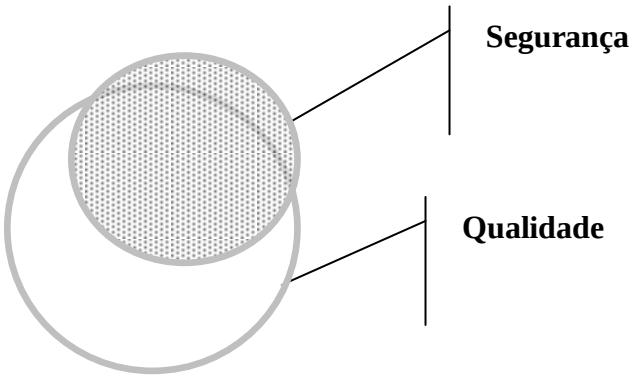

Figura 2.5 – Relação entre qualidade e segurança. Adaptado de Howard (2006).

Há ocasiões em que qualidade e segurança se excluem, conforme afirmam McGraw (2006) e Howard (2006). O desenvolvimento seguro, por exemplo, deve levar em conta ações maliciosas de agentes externos ao sistema (pessoas ou sistemas), isto não tem relação alguma com qualidade de *software*. Da mesma forma, fazer *exploits* no *software* não é, essencialmente, uma tarefa de qualidade. As atividades de qualidade, por sua vez, se preocupam com o tratamento de defeitos, mas não necessariamente este tratamento garante que defeitos relacionados à segurança sejam eliminados completamente, pois vulnerabilidades podem ser introduzidas não intencionalmente. Explorar estas vulnerabilidades com o objetivo de utilizar o *software* de outra maneira que não àquela especificada, no entanto, é intencional,

e isso é um problema primordial de segurança.

Práticas de qualidade podem não apontar questões de segurança, mas podem ser usadas por um revisor experiente para identificar complexidade, coesão, acoplamento, e outros pontos de partida para identificar possíveis vulnerabilidades relacionadas à segurança. Em muitos casos, erros de segurança que advêm de um código mal escrito não aparecem de forma explícita no código. Eles normalmente são conseqüências de uma falta de cuidado maior e mais difundida que podem às vezes ser vistas de uma perspectiva maior de erros de qualidade. É preciso dar uma atenção maior em módulos que possuam pontos fracos na qualidade (McGraw et al., 2008).

O objetivo principal de garantia de qualidade é prover uma confiança justificável de que o *software* é livre de vulnerabilidades, e que funcione não apenas da maneira planejada, mas que esta não comprometa a segurança e outros requisitos de qualidade enumerados (Goertzel et al., 2006). Assim, para que a segurança seja atingida é preciso que as atividades da qualidade se preocupem com a integridade do *software*, com sua execução previsível e a conformidade com os requisitos. Estas características fazem parte do conjunto de atributos que um *software* deve possuir para ser considerado seguro. Para que o projeto de *software* construa um produto com estes atributos, é preciso primeiramente compreender as propriedades de um *software* e quais destas fazem com que o *software* seja mais ou menos seguro.

2.5. Processo de Desenvolvimento de *Software*

O IEEE define um processo como sendo "uma seqüência de passos executados para um dado propósito" [2]. No âmbito da Engenharia de *Software*, "um processo é o conjunto total de atividades de engenharia necessárias para transformar requisitos do usuário em software" [3]. O processo é uma parte integrante da Engenharia de Software.

Na composição de um processo, um modelo é utilizado como fundamento para organizar, estruturar e seqüenciar o processo. Este modelo propõe uma coleção de padrões que definem um conjunto de atividades, ações, marcos, tarefas e produtos de trabalho (artefatos), necessários ao desenvolvimento do software com qualidade. Independente do modelo, Pressman (2006) afirma que, de maneira geral, os engenheiros de software adotam um

[2] IEEE Standards Coordinating Committee. **IEEE Standard Glossary of Software Engineering Terminology** (IEEE Std 610.12-1990). Los Alamitos, CA: IEEE Computer Society, 1990 (ISBN 0738103918).

[3] HUMPHREY, Watts S. **Managing the Software Process**. Addison-Wesley Professional, 1989 (ISBN-10: 0201180952, ISBN-13: 9780201180954).

processo genérico que inclui as seguintes atividades: comunicação, planejamento, modelagem, construção e implantação. Isso também pode ser observado no SWEBOK que, dentre outras disciplinas, inclui: requisitos, *design*, construção, teste e manutenção (SWEBOK, 2004).

Um modelo de processo fornece uma referência de melhores práticas que podem ser usadas para avaliação e melhoria do processo. Modelos não definem processos, antes definem suas características. As melhores práticas que visam metas em comum são agrupadas em Áreas de Processo (*Process Areas*), e áreas similares podem ser organizadas em categorias (Davis, 2006).

Existem inúmeros modelos propostos, dentre eles destacam-se: *waterfall*, espiral, prototipação, baseado em componentes, incrementais, etc. Não é escopo deste trabalho discutir os méritos de cada um, mas um modelo em particular será estudado: iterativo e incremental.

2.5.1. Processos Iterativos e Incrementais

Neste modelo são praticadas iterações (repetições) que aumentam as funcionalidades de um *software* em desenvolvimento conforme o projeto progride (Figura 2.6). Uma iteração é um micro projeto, que pode passar por todas as fases de um processo (*waterfall*, por exemplo), com menor ou maior grau de intensidade para cada fase, dependendo do andamento do projeto. Diferentemente do modelo espiral, neste todas as fases do processo podem ocorrer ao mesmo tempo de forma concorrente.

Cada iteração produz incrementos de *software* passíveis de serem entregues (*deliverables*), contendo um número maior de funcionalidades que o incremento anterior, ou correções de defeitos.

Quando este modelo é usado, o primeiro incremento é freqüentemente chamado de *núcleo do produto* (Pressman, 2006). Mas, diferentemente do modelo de prototipagem, por exemplo, este tem o objetivo de apresentar um produto operacional a cada incremento. Os primeiros incrementos são versões simplificadas do produto final, mas oferecem capacidades que têm alguma utilidade ao usuário, além de uma plataforma de avaliação para o mesmo. Esta filosofia incremental é aplicada pelos métodos ágeis que serão discutidos na próxima seção.

A gerência de riscos é auxiliada pelo modelo iterativo, pois é muito complexo analisar requisitos de forma seqüencial (*waterfall*). Ao dividir o projeto em iterações é possível

entregar capacidades incrementais passíveis de avaliação. Isto fornece um rápido *feedback*, permitindo que riscos sejam abordados e mitigados constantemente. Desta forma, a gerência de riscos evita custos adicionais e retrabalho.

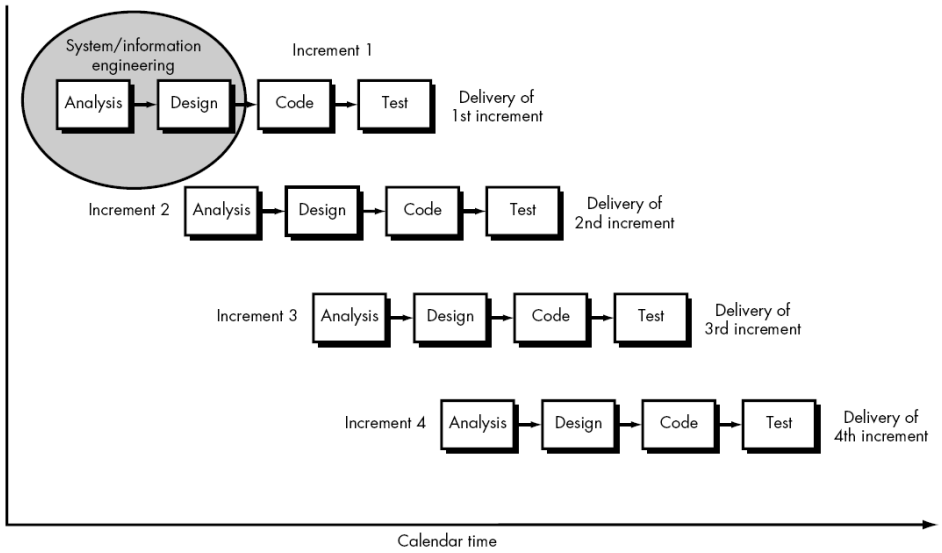

Figura 2.6 – Modelo de Desenvolvimento Incremental (Pressman, 2006)

2.5.2. Métodos Ágeis de Desenvolvimento

Métodos ágeis não são formalmente considerados processos de desenvolvimento[4]. Enquanto um processo impõe sistematicamente uma seqüência de ações que alteram / produzem artefatos do sistema em um período de tempo, um método é um conjunto de princípios e regras que direcionam uma disciplina. Na verdade, a agilidade vem de melhores práticas de Engenharia de *Software* aplicadas de forma iterativa e que podem ser aplicáveis a um processo. O OpenUP é um exemplo disso, pois é um processo (possui ações seqüenciais) e incorpora o método ágil (conjunto de princípios e práticas). Uma tabela comparativa entre o RUP, o OpenUP e o XP, é apresentada no Apêndice A.

Estes métodos se baseiam em uma abordagem pragmática, e a idéia fundamental é simplificar o desenvolvimento de *software*. Apesar disso, os métodos não são simples, e podem ser de difícil adoção e aplicação, pois demandam uma grande mudança de pensamento e na forma de a equipe trabalhar.

[4] O termo *Metodologias Ágeis* é também muito utilizado, mas de forma incorreta. *Metodologia* significa “estudo dos métodos”. Método ágil é um método de desenvolvimento de *software*, ou seja, uma "receita" técnica para a produção de *software*.

Os métodos reúnem um conjunto de práticas, princípios e valores interconectados e dependentes. Para ser considerado ágil, o desenvolvimento deve seguir os princípios enumerados pelo *Agile Manifesto*[5]. Os conceitos chave destes manifesto são (Ambler, 2006):

- **Indivíduos e interações** ao invés de processos e ferramentas: são as pessoas (programadores, testadores, gerentes, arquitetos, clientes, etc) que constroem sistemas de *software*, e para fazer isso elas precisam trabalhar em equipe de forma efetiva. O fator mais importante necessário a ser considerado são as pessoas e como elas trabalham juntas, pois se isso não for feito da maneira correta as melhores ferramentas e os melhores processos não terão utilidade. Evidentemente que ferramentas e processos são importantes, mas não tão importantes quanto trabalhar em equipe de forma efetiva[6].
- ***Software* executável** ao invés de documentação: grande importância é dada à entrega freqüente de *software* funcional para que os *stakeholders* avaliem diretamente e aprovem os resultados antes da entrega do produto final. O *design* deve sempre ser simples, pois gastar muito tempo em *designs* complexos e sofisticados pode ser maléfico, a idéia é enfatizar o trabalho apenas em requisitos conhecidos e implementar apenas o que é realmente necessário na situação.
- **Colaboração do cliente** ao invés de negociação de contratos: é dada ênfase na comunicação e no envolvimento dos *stakeholders* com a equipe de desenvolvimento. A geração rápida de *software* permite um *feedback* contínuo dos *stakeholders*, isso é feito por meio de mecanismos de testes contínuos.
- **Respostas rápidas a mudanças** ao invés de seguir planos: com o progresso do trabalho, o entendimento dos *stakeholders* acerca do domínio do problema e do que deve ser feito se altera. Bem como o ambiente de negócio muda e as tecnologias. Mudança é uma realidade no desenvolvimento de *software*, e que deve ser refletida no processo de desenvolvimento. Um plano de projeto deve ser maleável, e deve haver espaço para mudanças conforme a situação se altera, caso contrário o plano rapidamente se torna irrelevante.

[5] Declaração de princípios em que se assenta o desenvolvimento ágil de *software*. Ver Anexo A.
[6] "A fool with a tool is still a fool" (BROOKS, Frederick P. **The Mythical Man-Month: Essays on Software Engineering**, 2nd Edition. Addison-Wesley Professional, August 12, 1995. (ISBN-10: 0201835959, ISBN-13: 978-0201835953))

Estes métodos vêm ganhando notório destaque na comunidade de desenvolvimento de *software*. Em grande parte porque são alternativas aos processos considerados "pesados", orientados principalmente ao planejamento e à documentação e menos ao código. Quando se desenvolve um *software* é importante lembrar que o objetivo principal é gerar um produto funcional. Mas, existem dois extremos que normalmente entram em atrito: desenvolvimento de *software* versus desenvolvimento de documentação[7]. A balança não deve pender somente para a documentação, uma vez que o tempo gasto com documentação é tempo gasto em não desenvolver novas funcionalidades no sistema. Assim como não deve pender somente para a codificação, pois é preciso que o produto final esteja preparado para evoluir, tenha manutenção fácil e seja passível de medições e avaliações, e isso não é possível sem uma documentação adequada.

No decorrer desta década vários métodos ágeis têm surgido, tendo como motivação a criação de uma alternativa aos processos "pesados". A necessidade de um processo mais "leve" veio da pressão exercida pelo *Agile Manifesto*, que no primeiro de seus princípios prega: "nossa prioridade mais alta é satisfazer o cliente através de entregas rápidas e contínuas de *software* de valor" (Agile Manifesto) (livre tradução). Um dos processos que adotou este princípio é o OpenUP, sugerindo uma abordagem ágil, mas com um certo grau de formalidade herdado do *Unified Process*.

2.5.3. OpenUP – Open Unified Process

O OpenUP (*Open Unified Process*) foi desenvolvido para ser uma versão mais leve do *Unified Process*, mas preservando suas características essenciais. Estas características são: desenvolvimento iterativo, dirigido por cenários e casos de uso, gerenciamento de riscos e a abordagem centralizada na arquitetura; e mantendo a estrutura de ciclo de vida em fases. Neste processo, o ciclo de vida provê uma maior visibilidade aos *stakeholders*, além de fazer um histórico dos pontos de decisão no decorrer de um projeto. A filosofia ágil incorpora-se em sua idéia principal, que é permitir que o *software* seja constantemente alterado e de forma rápida, com foco direcionado em sua qualidade. Segurança, sendo um de seus atributos, está inclusa neste objetivo e, portanto, não é intrusivo ao processo.

2.5.3.1. Conceituação

O OpenUP é definido como sendo um *Unified Process* "enxuto" que aplica a

[7] Esta documentação inclui os modelos, as especificações, os requisitos, os manuais, os casos de teste, etc.

abordagem iterativa e incremental dentro de um ciclo de vida estruturado. É considerado, também, uma revisão do RUP que é mínimo, completo e extensível. É mínimo, pois foca-se apenas no conteúdo fundamental do RUP, fornecendo um conjunto simplificado de produtos de trabalho (*work products*), papéis (*roles*), tarefas (*tasks*) e orientações (*guidance*). É completo, pois, apesar de ser um *framework* de processo de desenvolvimento, ele pode ser manifestado como um processo inteiro para construir um sistema. É extensível, pois pode ser usado como uma fundação no qual o conteúdo do processo pode ser relacionado ou feito sob medida quando necessário (Wang, 2007). É também um método de desenvolvimento ágil e iterativo baseado no *Unified Process*.

2.5.3.2. Produtos de Trabalho: Artefatos

Um produto de trabalho (*work product*) é o resultado de uma tarefa, ou seja, **o que** é produzido. Estes podem ser classificados em artefatos (se são concretos), resultados (se não são concretos) e entregáveis (se são pacotes de artefatos). Um artefato é algo que é produzido, modificado, ou usado por uma tarefa. Os papéis são responsáveis por criar ou atualizar estes artefatos, que estão sujeitos a controles de versão através do ciclo de vida do projeto.

No OpenUP há um total de 17 artefatos (OpenUP, 2008). Estes são considerados essenciais a um projeto, que deve usá-los para capturar informações relativas ao produto e ao próprio projeto. A captura de informações de maneira formal em artefatos não é incisiva, esta informação pode ser informal, capturada em *whiteboards*, atas de reunião, etc. Modelos, não obstante, fornecem uma maneira padrão de capturar informação.

Um produto pode ser um documento, um modelo ou o próprio *software*. O OpenUP fornece um conjunto mínimo de artefatos (produtos concretos), considerados suficientes para o uso do processo. Suas descrições são apresentadas na tabela B.1 do Anexo B.

2.5.3.3. Disciplinas e Tarefas

O OpenUP possui as seguintes disciplinas: Arquitetura, Desenvolvimento, Gerência de Projeto, Requisitos, Testes e Gerência de Configuração e Mudança. As demais disciplinas e áreas de interesse do RUP são omitidas, tais como Modelagem e Negócios, Ambiente, e Gerência Avançada de Requisitos e Gerência de Configuração. Estas disciplinas são consideradas desnecessárias para equipes pequenas ou são manipuladas por outras áreas da organização, portanto, não concernentes à equipe de desenvolvimento. Para uma comparação entre as disciplinas do RUP e do OpenUP ver Apêndice B.

Cada disciplina consiste de um número de tarefas (*tasks*). Uma tarefa indica **como** um

papel deve realizar o trabalho. Normalmente é definido como uma série de passos que envolvem a criação e atualização de um ou mais produtos de trabalho. No OpenUP há um número de tarefas que os papéis realizam como executores primários (ou seja, são responsáveis pela execução da tarefa) ou como executores suplementares (suportando ou fornecendo informações usadas na execução da tarefa).

2.5.3.4. Papéis

Um papel (*role*) indica **quem** realiza o trabalho. Um papel define o comportamento e as responsabilidades de um indivíduo ou um conjunto de indivíduos da equipe. Papéis não são individuais, pelo contrário, papéis descrevem responsabilidades. Um indivíduo tipicamente assume vários papéis de cada vez, e freqüentemente irá trocar de papéis durante a evolução do projeto.

As habilidades essenciais necessárias para um time de desenvolvimento que pretende utilizar o OpenUP são representadas por estes papéis. A Figura 2.7 ilustra bem a relação entre estes papéis e os sub-processos associados. No círculo exterior os papéis são definidos, e no círculo interior os alvos de cada papel são colocados: *Intent* (a intenção, isto é, o que deve ser produzido), *Management* (gerenciamento do projeto) e *Solution* (a solução de *software* proposta). Cada papel tem um foco em um ou dois alvos, a comunicação e a colaboração são parte central do processo de ênfase nestas tarefas.

Os sete papéis propostos pelo OpenUP são descritos a seguir:

- ***Stakeholder***: pode ser exercido por qualquer um que é, ou pode vir a ser, afetado pelo produto final. Equivale ao *Product Owner* do SCRUM. O *Stakeholder* se comunica e colabora tanto com o Gerente de Projeto (*Management*) quanto com os analistas (*Intent*).
- **Analista (*Analyst*)**: coleta informações dos *stakeholders* para entender o problema e capturar as prioridades dos requisitos. O foco principal do Analista é a *intenção (Intent*), ou seja, o que é o produto, centralizando-se no entendimento dos requisitos. Pode-se dizer que o Analista representa o cliente dentro da equipe de desenvolvimento. Este papel lida diretamente com os *stakeholders* para capturar requisitos, e com o Testador para elaboração/planejamento de testes.
- **Arquiteto (*Architect*)**: toma decisões chave de arquitetura que dirigem o *design* e a codificação do projeto. Deve ser exercido por um técnico sênior. O Arquiteto se comunica e colabora tanto com o Gerente de Projeto (*Management*) quanto com os desenvolvedores (*Solution*).

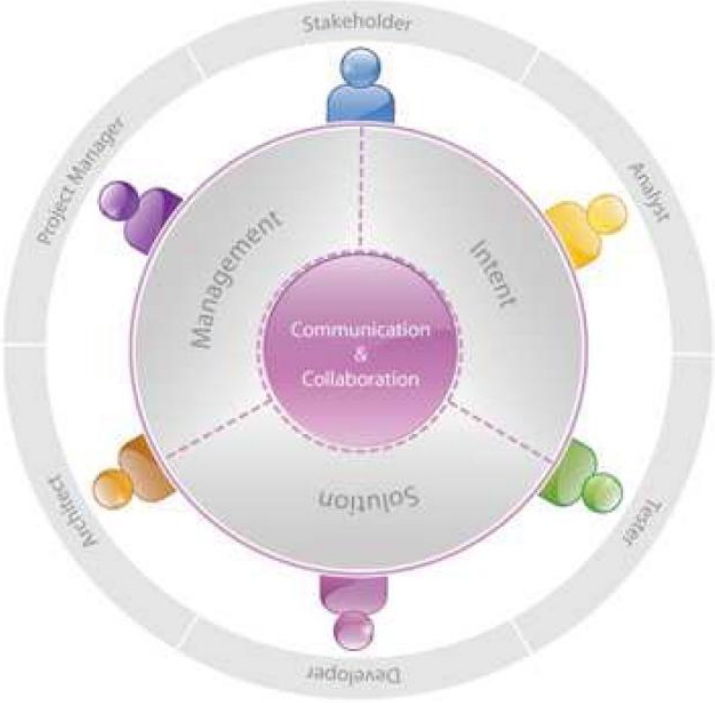

Figura 2.7 – Papéis no OpenUP (OPENUP, 2008).

- **Desenvolvedor (*Developer*)**: faz a codificação, testes unitários e a integração dos componentes que compõem a solução, isto de acordo com a arquitetura. Seu foco principal é a solução (*Solution*), ou seja, a codificação.
- **Testador (*Tester*)**: atividades de teste, tais como identificar, definir, implementar, e conduzir os testes necessários, assim como analisar os resultados. O Testador se comunica e colabora tanto com o Analista (*Intent*) quanto com os desenvolvedores (*Solution*).
- **Gerente de Projeto (*Project Manager*)**: lidera o planejamento em colaboração com os *stakeholders* e o time, criando um plano macro, coordenando a comunicação com os *stakeholders*, avaliando constantemente as iterações e o projeto, e mantendo o time focado em atingir seus objetivos. Esse plano descreve os tamanhos e os objetivos das quatro fases e das iterações de cada fase.
- **Qualquer papel (*Any Role*)**: representa qualquer membro do time que pode efetuar tarefas gerais.

2.5.3.5. Processo: Visão Geral

Um processo reúne tarefas, produtos de trabalho, e papéis, e adiciona estrutura e

seqüencialmente de informações. Tarefas ou produtos de trabalho podem ser agrupados em níveis mais altos de atividades, chamados WBS (*Work Breadkdown Structure*). Atividades ou tarefas podem ser marcadas como "planejadas" para identificar trabalho que se espera que seja designado. Diagramas podem ser adicionados para fornecer o seqüenciamento das informações.

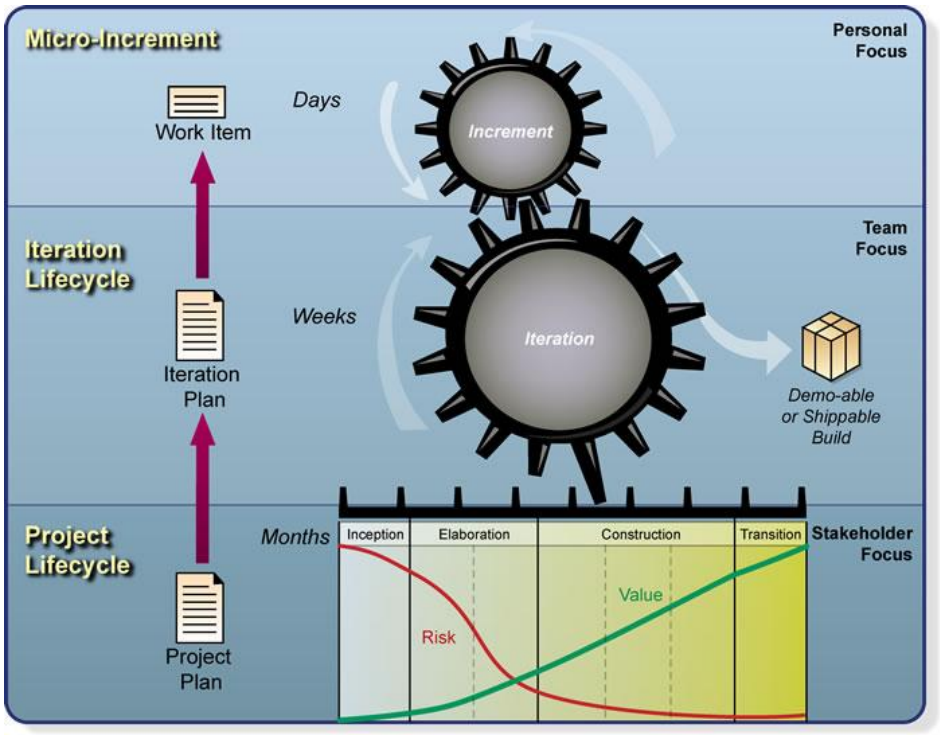

Figura 2.8. Visão geral do gerenciamento do OpenUP (OPENUP, 2008).

OpenUP é adequado para times pequenos que trabalham junto e no mesmo local, as pessoas envolvidas precisam se engajar em várias interações diárias face a face. Os membros da equipe são representados pelos papéis descritos anteriormente. Cada um toma suas próprias decisões sobre em que precisam trabalhar, e quais são as prioridades, e a melhor maneira de direcionar as necessidades dos *stakeholders*.

O conteúdo do OpenUP organiza o trabalho em três níveis, cada um com um enfoque em particular. Estes níveis são: pessoal, time e *stakeholder* (Figura 2.8). No nível pessoal (*Personal Focus*), cada indivíduo contribui com micro-incrementos[8], que devem auxiliar o time a atingir os objetivos estabelecidos. O time de desenvolvimento (*Team Focus*), liderado pelo gerente, faz um plano de iteração tendo como base os objetivos e as prioridades

[8]Isso equivale aos *baby steps* do XP.

definidos. Os *stakeholders* (*Stakeholder Focus*), finalmente, contribuem ao fazer a avaliação dos resultados de um release para fornecer o *feedback* sobre o produto de *software* funcional.

O projeto é dividido em iterações, e os micro-incrementos são planejados para que definam o que deve ser entregue em cada iteração. Esta abordagem faz com que o desenvolvimento seja focado na entrega incremental de algo útil aos *stakeholders* de uma maneira bastante previsível. O time de desenvolvimento é auto-organizável de tal modo que chegue a uma conclusão de como atingir seus objetivos e como os produtos serão entregues.

O ciclo de vida do projeto é estruturado em quatro fases: *Inception*, *Elaboration*, *Construction* e *Transition*. Esta estrutura disponibiliza uma boa visão aos *stakeholders* e aos membros do grupo. Isto permite uma administração superior efetiva, além de auxiliar na tomada de decisões nas situações apropriadas. As decisões devem acontecer rapidamente, não deixando espaço para debates. Este tipo de desenvolvimento foca na produção de código funcional, reduzindo o risco de uma análise truncada. A freqüente demonstração de código funcional provê um mecanismo de *feedback* que permite que correções sejam feitas no andamento do projeto.

2.6. Segurança no Ciclo de Vida de Desenvolvimento

Um processo seguro é aquele em que há um conjunto de atividades que devem ser executadas para desenvolver, manter, e entregar uma solução segura de *software* (Davis, 2006). Muitas vulnerabilidades poderiam ser evitadas se simplesmente os desenvolvedores fossem treinados para lidar sistemática e consistentemente com estas vulnerabilidades. Geralmente, práticas para reconhecer e remover defeitos de vulnerabilidades não são compreendidas. Assim, é importante que todos os participantes do projeto saibam conduzir a qualidade de forma que testes sejam direcionados para encontrar erros de insegurança através do uso de técnicas de desenvolvimento. Os desenvolvedores devem ter conhecimento das implicações da segurança na arquitetura, no *design*, na codificação, no *deploy* e na operação.

Os engenheiros de *software* devem tratar todos os defeitos e fraquezas como *exploits* em potencial. A tarefa de reduzir fraquezas que podem ser exploradas se inicia com as especificações de requisitos de segurança, também se deve considerar requisitos que possam ter sido negligenciados. Um *software* que inclui requisitos de segurança é menos suscetível a ataques, pois a segurança está embutida em seu interior, e o processo auxilia na detecção de problemas e nos testes e verificações de segurança. E, novamente, é menos trabalhoso e custoso projetar e desenvolver um *software* desde o início com segurança em mente, do que tentar demonstrar que o *software* é seguro posteriormente.

A análise e a modelagem podem servir para melhor proteger o *software* contra padrões de ataque (*attack patterns*) complexos, e que envolvam interações entre componentes externos e não previstos na arquitetura. Isto também permite reforçar a segurança do *software* em suas interfaces, e aumentar a tolerância a falhas.

Embora implicitamente os *stakeholders* esperem que a segurança seja entregue junto com o produto encomendado, eles não solicitam isso a princípio. Com prazos apertados e orçamentos estourados, os desenvolvedores só pensarão em desenvolver o que foi solicitado. Um processo que tenha um ciclo de vida com ênfase em segurança pode compensar inadequações na segurança dos requisitos. Isso é feito ao se adicionar práticas dirigidas por riscos, e checar pela adequação destas práticas durante todas as fases do ciclo de vida. A Figura 2.9 ilustra como incorporar a segurança no ciclo de vida usando o conceito de práticas proposto em McGraw et al. (2008). As melhores práticas de segurança são aplicadas ao conjunto de artefatos que são criados durante o processo de desenvolvimento do *software*.

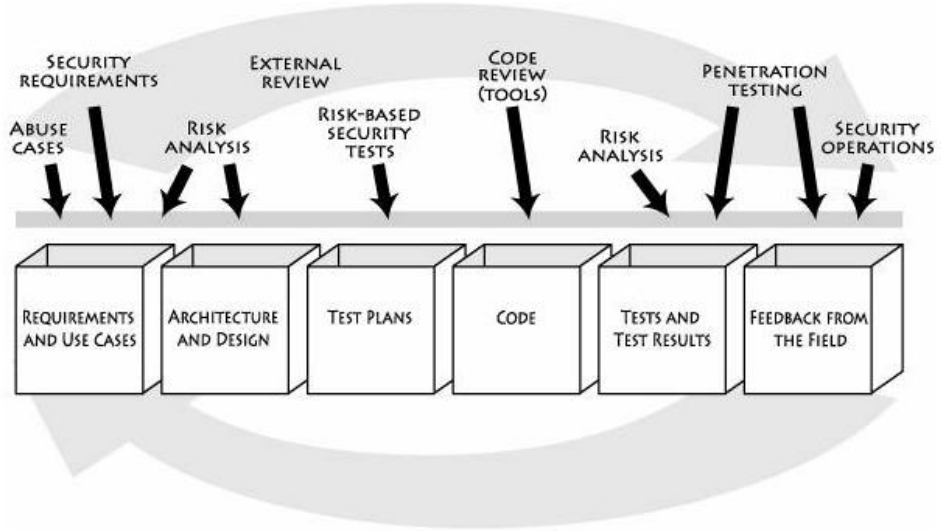

Figura 2.9 – Abordagem proposta por McGraw et al .(2008) para incorporar a segurança no ciclo de vida de desenvolvimento.

Os controles de segurança no ciclo de vida, porém, não devem se limitar aos requisitos, *design*, codificação e testes. É importante realizar continuamente revisões de código, testes de segurança, controle estrito de configurações, e garantia de qualidade durante o *deploy* e durante a operação do *software.* Essas atividades visam assegurar que atualizações e *patches* não adicionem novas fraquezas ou lógicas maliciosas para o *software* em produção. Todo este esforço, porém, é muito menos custoso do que corrigir defeitos após a entrega.

3. Segurança nas Fases do OpenUP

3.1. Conceituação

No capítulo anterior foi visto quais características podem ser associadas à qualidade e segurança do *software*, e *o quê* deve ser feito para garantir e influenciar estas características no processo de desenvolvimento. Neste capítulo será visto *como* estes padrões podem ser aplicados em cada fase de desenvolvimento para que a segurança seja embutida no ciclo de vida do *software*, tendo como base o modelo proposto pelo OpenUP.

Paes (2007) afirma que existem duas alternativas para considerar a segurança no ciclo de vida de desenvolvimento: como uma extensão (incluir uma nova disciplina ao processo) ou como uma customização (incorporar atividades, tarefas e papéis relacionados com a segurança nas disciplinas existentes).

3.2. Fontes de Insegurança no Software

Atualmente tem-se construído sistemas extremamente grandes e complexos. Estes sistemas podem passar por um vasto número de diferentes estados. Estas características fazem com que seja particularmente difícil de desenvolver e operar *softwares* que sejam consistentemente corretos, e muito menos seguros. A presença inevitável das ameaças e riscos significa que os desenvolvedores precisam prestar atenção à segurança do *software* mesmo quando ela não é um requisito explícito (McGraw et al., 2008, e Goertzel et al., 2006).

Tamanho e complexidade não deveriam ser apenas propriedades da codificação do *software*, mas também do *design*, pois desta forma seria mais fácil descobrir falhas no *design* que poderiam ser manifestas como fraquezas exploráveis na implementação. A rastreabilidade deve ser feita pelos mesmos revisores para garantir que o modelo satisfaz os requisitos de segurança especificados e que a execução não se desvie da concepção segura. Além disso, rastreabilidade provê uma base formal na qual se define os casos de teste de segurança.

Normalmente, as fraquezas exploradas pelos ataques vêm das seguintes fontes de insegurança:

- Complexidades, inadequações e/ou mudanças no modelo de processo de desenvolvimento.
- Suposições erradas dos engenheiros de *software*, entre elas, sobre a capacidade, saídas

e comportamentos da execução do *software*, ou ainda entradas de entidades externas.

- Especificações ou *design* falhos, ou implementações defeituosas de interfaces com entidades externas e de componentes do ambiente de execução do *software*.
- Iterações não intencionais nos componentes de *software*, inclusive aqueles fornecidas por terceiros.

Defeitos podem ser evitados nas fases de requisitos e de *design* (métodos formais), e na fase de desenvolvimento (testes e revisões no código), mas vulnerabilidades podem ser introduzidas durante a montagem, integração, entrega, e na operação. Mesmo que haja um ciclo de vida confiável e com ênfase na segurança, se o *software* crescer em tamanho e complexidade será inevitável a introdução de defeitos e vulnerabilidades. Portanto, continuamente avaliar e melhorar o processo é atividade gerencial importante para que a segurança seja mantida no *software* mesmo após sua evolução e/ou modernização.

3.3. Gerenciamento da Segurança no Desenvolvimento

Geralmente, as defesas de uma aplicação seguem a abordagem reativa (esperam o problema aparecer para reagir) (McGraw et al., 2008), no entanto, as práticas associadas com a segurança no *software* e seu papel nos processos de engenharia de segurança focam em prevenir que fraquezas entrem no *software*, ou, se isso for inevitável, ao menos remover estas fraquezas tão logo seja possível no início do ciclo de vida de desenvolvimento, e antes que do *deploy* do *software*. Estas fraquezas, não intencionais ou inseridas de forma maliciosa, podem entrar no *software* em qualquer ponto do desenvolvimento do *software* por meio de requisitos inadequados ou incorretos; por meio de arquiteturas ou *designs* ambíguos, incompletos, instáveis, ou impróprios; por meio de erros na implementação; por meio de testes incompletos ou inapropriados; ou decisões inseguras de configuração e entrega.

Ao contrário do que se pensa a segurança no *software* não pode ser de responsabilidade exclusiva dos desenvolvedores que escrevem o código, pelo contrário, requer o envolvimento de todo o time de desenvolvimento com o suporte da organização. Existem várias práticas para gerentes de projeto que focam na segurança e que podem ser embutidas em qualquer processo de desenvolvimento. Entre outras coisas, estas práticas incluem requisitos de engenharia de segurança com *Abuse Cases*, análise de riscos arquitetural, revisão de código seguro, testes de segurança baseados em riscos, e testes de

penetração de *software*. Este é o objetivo central deste trabalho: enumerar estas práticas, aplicáveis a todos os envolvidos no projeto, no decorrer do ciclo de vida de desenvolvimento, e, no caso, utilizando o OpenUP como base para isso.

As atividades de segurança devem se iniciar na concepção, com levantamento e análise dos requisitos de segurança. Nas fases de elaboração e construção se concentram a maior parte das atividades, primeiro para definição de uma arquitetura adequada, e depois para realização desta arquitetura. Finalmente, na fase de transição é preciso verificar e validar os requisitos para avaliar a conformidade com os requisitos de segurança.

3.4. Antes de Iniciar um Projeto de Software Seguro (Fase Zero)

O material humano do projeto tem um peso maior nos desenvolvimentos ágeis. Assim, os participantes do projeto devem ter conhecimento sobre o desenvolvimento seguro. Isso pode ser feito através de treinamentos externos e por meio da inclusão na equipe de um especialista em Engenharia de Segurança. Como destacado por Howard (2006), um projeto de *software* seguro deve contar com um especialista, geralmente um técnico sênior, que pode assumir um papel interno do OpenUP (Desenvolvedor ou Arquiteto) ou atuar como um consultor de segurança. Isto tem como objetivo disseminar a cultura de segurança entre os participantes do projeto.

Finalmente, é vital que haja apoio da organização para que as atividades sejam feitas de forma efetiva. Esta conscientização e cultura de segurança devem fazer parte das metas da área de desenvolvimento.

3.5. Customização do Processo

O SSE-CMM fornece um conjunto de PAs e metas para essas PAs (a Tabela D.1, Anexo D, enumera as PAs e suas metas). Estas metas podem ser utilizadas como entradas para avaliação dos resultados de uma iteração. O papel da Gerência de Projeto tem como atividades primárias realizar o gerenciamento e planejamento de uma iteração, e avaliando os resultados da mesma. Como este planejamento é atividade a ser executada a cada iteração, algumas metas destas PAs auxiliam na avaliação dos riscos relativos a segurança, e são bastante adequadas para isso, uma vez que sua natureza é iterativa. Desta forma, as seguintes PAs podem auxiliar neste processo:

- A PA05 (Avaliar o Risco Operacional da Segurança) tem como propósito identificar

riscos de segurança envolvidos com a confiança operacional do sistema em um ambiente definido. Esta PA foca na apuração destes riscos baseada no entendimento estabelecido de como as capacidades operacionais e os ativos são vulneráveis a ameaças. Isto inclui atividades que avaliam o impacto operacional que resulta de um *exploit* de uma vulnerabilidade feito com sucesso. Este conjunto de atividades é realizado em qualquer momento durante o ciclo de vida para suportar decisões relacionadas ao desenvolvimento, manutenção, ou operação do sistema dentro de um ambiente conhecido.

- A PA08 (Administrar os Controles de Segurança) deve ser executada a cada iteração. Tem como propósito assegurar que a segurança pretendida para o sistema, integrada no *design*, esteja de fato sendo alcançada pelo sistema resultante em seu estado operacional.
- A PA09 (Coordenar a Segurança) tem como propósito garantir que os participantes do projeto estejam cientes e envolvidos com as atividades de Engenharia de Segurança. Esta atividade é crítica, pois a Engenharia de Segurança não pode ocorrer isoladamente. Esta coordenação envolve a manutenção de comunicações abertas entre os grupos de segurança, outros grupos de engenharia, e grupos externos (se houver). Vários mecanismos podem ser usados para coordenador e comunicar as decisões e recomendações entre os envolvidos, incluindo memorandos, documentos, e-mail, reuniões, e grupos de trabalho.

3.5.1. Estruturação da Segurança nas Fases do OpenUP

Nesta seção as práticas, atividades e padrões concernentes à segurança serão relacionadas com as quatro fases do OpenUP. Como cada fase tem ênfase maior em determinadas atividades, e será dado maior destaque àquelas pertinentes a segurança que se encaixam nas atividades da fase em questão. De maneira geral, para cada fase os seguintes temas serão discutidos:

- Quais as perspectivas de segurança dos *stakeholders* e da equipe de desenvolvimento. Como os padrões de segurança influenciam a segurança da fase em estudo.
- O que cada papel sugerido pelo OpenUP deve realizar, quais artefatos devem ser gerados e qual a importância destes na segurança e na qualidade do *software*.

- Quais os *milestones* de segurança que devem ser atingidos ao final de cada fase.

3.5.2. Segurança nas Fases do OpenUP

Como o OpenUP é um processo iterativo, todas as suas disciplinas são executadas em cada iteração, mas com quantidades de atividades distintas dependendo da fase em que o projeto está. Nas seções seguintes as quatro fases do processo serão descritas em termos de segurança, ou seja, como os padrões e as práticas podem auxiliar nas atividades da fase em questão.

INCEPTION (Concepção)

No início do projeto o escopo do sistema é estabelecido ao se alcançar um alto nível de entendimento dos requisitos, incluindo uma compreensão consistente de qual tipo de sistema será construído. Como se observa na Figura 3.1, grande esforço nesta fase se concentra na disciplina de Requisitos. É nesta fase que muitos dos riscos de negócio são mitigados, e onde se produz um documento de Visão. Este documento auxilia na decisão dos *stakeholders* de proceder ou não com o projeto. Isto é similar ao que os demais métodos ágeis se referem como Iteração 0 (zero). É nesta fase inicial que os erros devem ser eliminados da solução e os defeitos devem ser prevenidos.

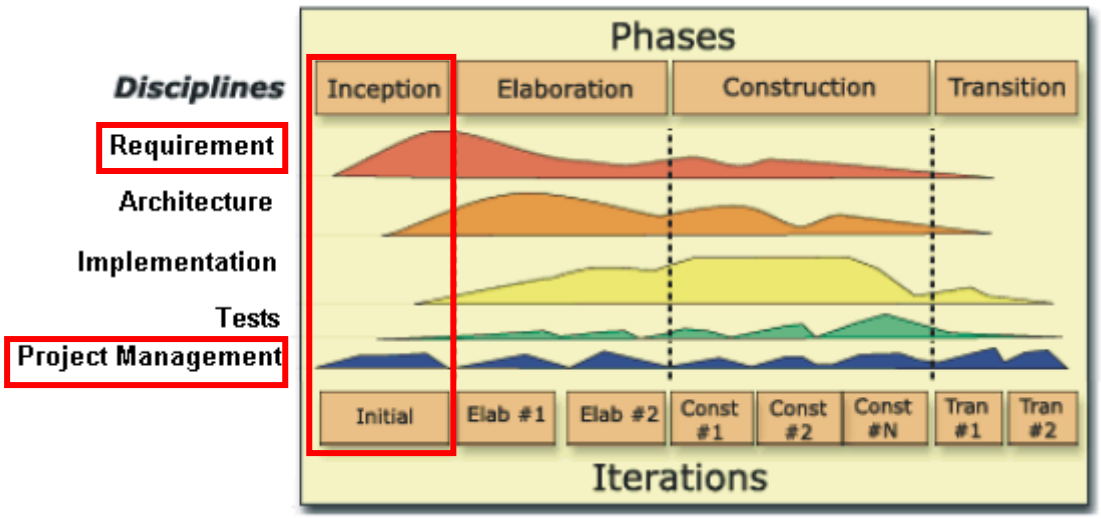

Figura 3.1 – Esforço das atividades da fase de Concepção.

Durante as definições iniciais do projeto de *software* o foco é somente nas

características e funções de negócio. Isto deixa requisitos não funcionais importantes, como desempenho, usabilidade, e segurança, para serem resolvidos mais tarde no ciclo de vida, causando muitos atrasos e embaraços no projeto. As considerações de segurança incluem um número de aspectos tais como controle de acesso e autorização, manipulação apropriada de dados sensíveis, uso apropriado de dados e de acesso a armazenamento, e métodos de criptografia. Requisitos de segurança são requisitos não funcionais. Por outro lado, muitos requisitos de segurança são orientados a casos de uso, e precisam da definição de um cenário principal, somados à definição de caminhos alternativos e de exceção. Se requisitos funcionais e não funcionais não forem definidos e incorporados ao *software* de maneira correta, erros na codificação e falhas no *design* podem colocar informações críticas e operações em risco. Requisitos de segurança devem ser tratados como quaisquer outros requisitos, isto é, devem ser priorizados, colocados em um escopo, e gerenciados como parte dos casos de uso e requisitos funcionais (Dent, 2008).

Durante a fase de concepção, requisitos não funcionais devem ser capturados e definidos como atributos do *software*. Os requisitos não funcionais que envolvam segurança precisam ser mapeados para requisitos funcionais para que possam ser construídos no *software* e testados apropriadamente. Ao se mapear os requisitos não funcionais para requisitos funcionais, os requisitos de segurança tornam-se uma parte do processo de análise dos requisitos. Requisitos de segurança podem expressar tanto restrições para solução quanto declarações de propriedades ou atributos requisitados.

O esforço maior nesta fase é despendido nas atividades de levantamento e análise dos requisitos (Figura 3.1). Assim, o entendimento dos requisitos de segurança deve ser claro e, inicialmente, levantado junto com os *stakeholders*. É importante que os requisitos sejam definidos efetivamente, para que não haja retrabalho, custos extras, e para que seja possível avaliar e mensurar a segurança no decorrer do ciclo de vida de desenvolvimento. Isso evitará que as ações contra vulnerabilidades de segurança sejam pró-ativas e não reativas, ou seja, é preciso antecipar comportamentos anormais.

É vital que exista um especialista em segurança para agir como consultor para a equipe de desenvolvimento (Howard, 2006 e Goertzel et al., 2006). Requisitos de segurança e as funcionalidades de segurança resultantes são atividades de especialistas em segurança. Ambos são definidos e construídos como defesas explícitas contra agentes maliciosos. Desta forma, a noção de requisitos de segurança é tarefa principal dos Analistas de Segurança. O OpenUP é adequado para equipes pequenas e co-alocadas e, talvez, seja um luxo ter um especialista em segurança na equipe. Mas, como mencionado, ele pode agir apenas como um consultor,

auxiliando nas atividades de planejamento, avaliação e comunicação entre os demais papéis.

Levantamento de Requisitos

Requisitos podem ser derivados de políticas de segurança e padrões de segurança que descrevem a proteção requerida pelo sistema, como, por exemplo, as descrições dos TOEs produzidos pelas avaliações do CC. É preciso considerar tanto as perspectivas internas (desenvolvedores) e externas (atacantes), o que inclui as necessidades dos *stakeholders* (os ativos a serem protegidos) e como os atacantes agem para explorar vulnerabilidades do *software* (padrões de ataque).

Como abordagens úteis para a Engenharia de Requisitos de Segurança destacam-se algumas técnicas utilizadas para auxiliar gerentes de projeto a garantirem que o produto resultante alcance, de forma efetiva, estes requisitos: a norma SQuaRE , *Attack Trees*, e os *Abuse Cases*. Como o OpenUP é dirigido por casos de uso, apenas os *Abuse Cases* serão mostrados aqui, uma breve descrição sobre as árvores de ataque é dada no Apêndice C.

Abuse Cases

Abuse[1] (ou *Misuse*[2]) *Case* é uma extensão dos diagramas de Caso de Uso, e pode auxiliar na elaboração da perspectiva do atacante. Contemplar eventos negativos ou inesperados ajuda no melhor entendimento de como criar *softwares* mais confiáveis e seguros. Esta técnica representa as ações que o sistema deve prevenir em conjunto com aquelas que o *software* deve suportar para que seja feita a análise de requisitos de segurança.

Os *Abuse Cases* aplicam o conceito de cenário negativo, isto é, uma situação que os *stakeholders* não desejam que ocorra, em um contexto de Caso de Uso. Um Caso de Uso, em contraste, geralmente descreve o que o sistema deve apresentar aos *stakholders.* Modelos de Casos de Uso e seus diagramas associados são úteis para especificação de requisitos.

Um dos objetivos dos *Abuse Cases* é decidir e documentar a priori como o *software* deve reagir contra o uso ilegítimo. O método mais simples e prático para criar *Abuse Cases* é normalmente por meio de um processo de *brainstorming*. Métodos formais podem ser utilizados, mas consomem muito tempo e recursos. A prática do *brainstorming* é mais

[1] John McDermott and Chris Fox. "Using Abuse Case Models for Security Requirements Analysis," Proceedings of the 15th Annual Computer Security Applications Conference, Scottsdale, AZ, IEEE Computer Society Press, 1999, p. 55.

[2] Sindre, Guttorm, & Opdahl, Andreas L. "Eliciting Security Requirements by Misuse Cases," 120–131. Proceedings of the 37th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS-37 '00). New York: IEEE Press, 2000.

pragmática e adequada para métodos ágeis, pois cobre muitas possibilidades rapidamente, apesar de requerer um especialista experiente.

Nesta técnica, o especialista realiza várias questões para os *designers* para ajudar na identificação de lugares onde o sistema está mais propenso a fraquezas. Esta atividade procura espelhar a maneira como o atacante pensa, envolve um olhar cuidadoso sobre todas as interfaces de usuário (fatores ambientais) e considera todos os eventos que os desenvolvedores assumem que uma pessoa irá ou não fazer. Os atacantes tentarão, certamente, explorar o que não se pode fazer.

O processo de especificar os *Abuse Cases* deve permitir que o *designer* faça uma distinção muito clara do uso apropriado do uso inapropriado da aplicação. Para chegar a este ponto, no entanto, os *designers* devem levantar as questões corretas. Tentar responder a estas questões ajuda os *designers* a questionar explicitamente os pressupostos de arquitetura e *design*, e coloca o *designer* a frente do atacante ao identificar e corrigir um problema antes que este seja criado.

Priorização de Requisitos

A segurança deve ser explicitamente trabalhada no nível de requisitos. Bons requisitos de segurança cobrem tanto requisitos de segurança evidentes (exemplo: criptografia) quanto características emergentes (capturadas em *Abuse Cases* ou *patterns* de ataque). Uma vez bem definidos, é necessário priorizá-los para que possam ser corretamente implementados.

Uma vez identificados os requisitos de segurança, é necessários criar uma lista priorizada de requisitos (artefato Lista de Riscos). Isso é necessário porque existem restrições de tempo e orçamento, o que torna difícil implementar todos os requisitos levantados. Bem como, requisitos de segurança são, normalmente, implementados em estágios, e a priorização ajuda na determinação de quais devem ser implementados primeiro.

Na priorização de requisitos é preciso aplicar a técnica mais adequada. Vários modelos se baseiam em técnicas tradicionais de Engenharia de Requisitos, e podem ser usadas no desenvolvimento de requisitos de segurança. Algumas destas são: árvore de busca binária, técnica de designação numérica, *planning game*, etc (mais podem ser vistas em McGraw et al., 2008).

Customização da Fase de Concepção

Os papéis, atividades e artefatos do OpenUP devem ser estendidos e/ou incluídos para englobar as práticas de segurança. A Tabela 3.1 propõe as tarefas que devem ser executadas, e

os artefatos que devem ser produzidos para que tornar possível a inclusão da segurança no ciclo de vida.

Papel (executor primário)	***Tarefas***	***Saídas (artefatos)***
Analista	Desenvolver a visão técnica (fornece subsídios para identificação de ameaças); identificar e esboçar requisitos; detalhar cenários de casos de uso (pode ser estendido para detalhar *Abuse Cases*); detalhar requisitos *system-wide* (pode ser customizado para identificação dos requisitos de segurança)	Glossário, Visão (*Attack Trees*), Casos de Uso (*Abuse Cases*), Modelos de Casos de Uso (Modelos de *Abuse Cases*), Requisitos não expressos por casos de uso (Requisitos de Segurança)
Gerência de Projeto	Planejar o projeto (pode incluir as atividades de segurança); planejar e gerenciar a iteração, e avaliar resultados.	Plano de Projeto (Avaliação de Riscos), Plano de Iteração, Lista de Riscos e Lista de Itens de Trabalho
Arquiteto	Esboçar a arquitetura (identificação de *patterns* de ataque)	Arquitetura (Modelo de Ameaças)

Tabela 3.1 – Customização das atividades da fase de concepção para contemplar artefatos e atividades de segurança.

Geralmente esta fase possui poucas iterações (uma ou duas). Com a inclusão de atividades de segurança é preciso que estas sejam prolongadas, pois artefatos há um número maior de atividades e artefatos. Uma proposta de mapeamento dos objetivos da fase com as práticas de segurança é apresentada na Tabela 3.2.

Objetivos da fase	***Atividades que abordam os objetivos***	***Práticas de segurança***
Entender o que será construído e Identificar funcionalidades chave do sistema	• Iniciar o Projeto • Identificar e Refinar Requisitos Detalhar um conjunto de requisitos (um ou mais casos de uso, cenários	Juntamente com os requisitos de negócio detalhados, devem ser levantados os requisitos de segurança (PPs). Os Casos de Uso são estendidos para os *Abuse Case* correspondentes, e isso produz cenários para implementação de testes de segurança. As metas da PA01 (SSE-CMM) podem ser

	ou requisitos *system-wide*). *Kick off* do projeto, entrar em acordo com os *stakeholders* com relação ao escopo do projeto, e um plano inicial para alcançá-lo. Esta atividade reúne tarefas requeridas para definição da Visão e criação do Plano de Projeto	satisfeitas na fase de concepção. A Parte 1 do *Common Criteria* pode ajudar no levantamento de requisitos de segurança. Requisitos de segurança também podem ser levantados por meio de técnicas de *attack trees.*
Determinar pelo menos uma solução possível	• Concordar com a Abordagem Técnica Alcançar um acordo de uma abordagem técnica viável para o desenvolvimento da solução.	Com a arquitetura esboçada já é possível identificar um ou mais *patterns* de ataque e árvores de ataque. As metas da PA02 auxiliam na disseminação da cultura de segurança e na abordagem técnica a ser utilizada para garantir a segurança.
Entender o custo, estimativas, e riscos associados como projeto	• Iniciar o Projeto • Planejar e Gerenciar a Iteração Iniciar a iteração e permitir que os membros da equipe iniciem as atividades de desenvolvimento. Monitorar e comunicar o status do projeto para *stakeholders* externos. Identificar e manipular exceções e problemas.	A iteração deve prever as atividades de segurança. McGraw et al. (2008) sugere que um processo de Engenharia de Segurança seja definido no início do projeto.

Tabela 3.2 – Mapeamento entre os objetivos da fase de Concepção e as práticas de segurança.

***Common Criteria* e Requisitos de Segurança**

A fase de concepção é auxiliada pela Parte 1 da norma, mas como referência inicial, uma vez que os componentes do CC serão melhores aplicados na fase de Elaboração (Piattini, 2007). Os Analistas podem se beneficiar com as informações de apoio e com as referências da parte 1, isso permite que as especificações de segurança dos TOEs sejam desenvolvidas. Da mesma forma, os Testadores poderão utilizar estas mesmas informações para orientar a estruturação dos PPs e dos STs. Este padrão ajuda, além da especificação dos requisitos, na

especificação de atributos de segurança dos produtos de *software*.

SSE-CMM e Requisitos de Segurança

Algumas PAs do SSE-CMM podem fornecer metas de segurança para a fase de concepção. A PA01 (Especificar as Necessidades de Segurança) pode ser aplicada para especificação de requisitos. O propósito desta PA é identificar explicitamente as necessidades relacionadas com a segurança do sistema. Estas necessidades são integradas com base no contexto operacional da segurança do sistema. A segurança atual (se existir) e o ambiente dos sistemas da organização, e o conjunto de objetivos de segurança são identificados. Um conjunto de requisitos relacionados à segurança é definido para o sistema que, se aprovado, torna-se a linha base para a segurança dentro do sistema.

A PA02 (Fornecer Entrada para Segurança) deve ser feita pela Gerência do Projeto para disseminar a cultura de segurança. Seu propósito é fornecer aos arquitetos, *designers*, desenvolvedores, ou mesmo usuários, com informações de segurança de que precisam. Esta informação inclui segurança na arquitetura, no *design*, ou alternativas de implementação, e orientações para segurança. A entrada é desenvolvida, analisada, fornecida e coordenada pelo time de desenvolvimento com base nas necessidades de segurança identificadas pela PA01.

***Milestones* de Segurança**

Se a segurança for considerada na Concepção, os seguintes *milestones* são sugeridos (Dent, 2008):

- Requisitos de segurança de alto nível devem estar bem compreendidos, documentados (Políticas de Segurança), e priorizados em relação aos objetivos de negócio (Lista de Riscos).
- Se for possível, quaisquer requisitos de segurança arquiteturalmente significativos que possam apresentar riscos ao planejamento devem ser priorizados durante a próxima fase (Elaboração)

ELABORATION (Elaboração)

É na elaboração que os riscos mais importantes são reduzidos de modo que seja possível estimar custos e atualizar cronogramas. Nesta fase os riscos técnicos mais importantes devem ser mitigados ao se cuidar da maior parte das tarefas técnicas mais difíceis. *Design*, codificação e testes já começam a ser feitos, mas a atividade mais importante é delinear uma arquitetura executável, incluindo subsistemas, suas interfaces, componentes chave, e

mecanismos arquiteturais tais como: persistência e comunicação entre processos. Ao definir, fazer o *design*, implementar, e testar capacidades chave, os riscos de negócio de maior impacto são abordados e validados com o *stakeholder*. Nem todos os requisitos devem ser analisados e definidos, se isso for feito, corre-se o risco de o desenvolvimento se transformar em cascata. Como conseqüência disso, apenas os riscos levantados devem ser detalhados e analisados.

Nesta fase as atividades se concentram principalmente na elaboração de uma arquitetura estável e executável (Figura 3.2). É possível que as implementações se iniciem, atacando requisitos mais críticos ou como formas de validar a arquitetura. Yoder (1998) afirma que muitas das vulnerabilidades dos *softwares* se originam de escolhas feitas nesta fase.

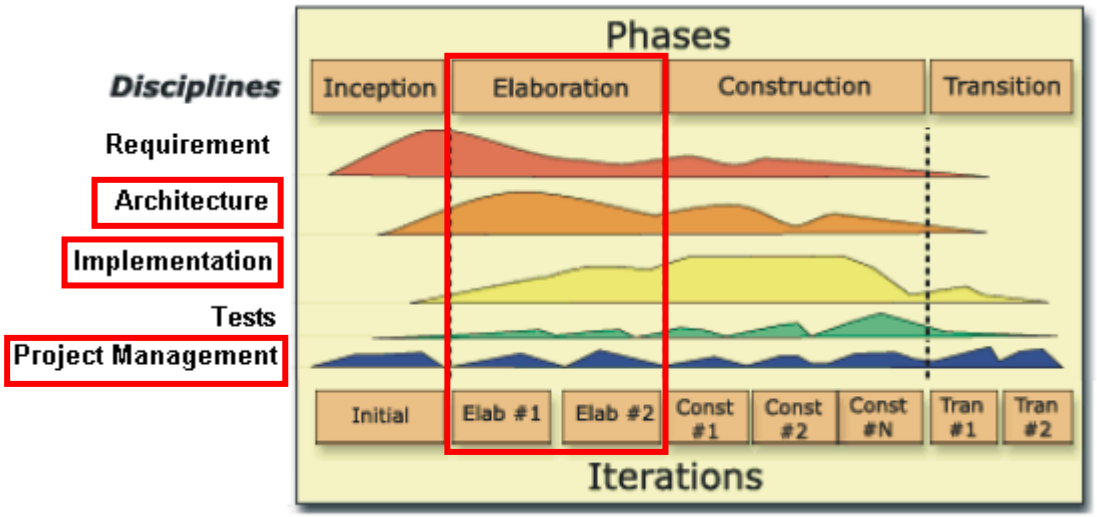

Figura 3.2 – Esforço das atividades da fase de Elaboração.

As Políticas de Segurança são estabelecidas nas iterações iniciais desta fase (Paes, 2007), e devem ser definidas de forma muito clara e consistente. O documento de Políticas de Segurança deve trazer definições do que significa ser seguro, práticas de segurança, princípios, etc. Este documento pode ser atualizado no decorrer do projeto. Estas políticas se referem ao projeto, e não à organização, mas fazem parte da mesma.

Durante a elaboração, o plano de projeto para a aplicação começa a tomar forma. Os requisitos têm seu *design* mais detalhado e as estimativas para a fase de Construção são esboçadas. A arquitetura do sistema está bem entendida. Qualquer requisito, incluindo aqueles pertinentes à segurança, que foram identificados como potenciais riscos durantes a fase de

Concepção, requerem uma atenção antecipada durante esta fase. Um esqueleto do sistema funcional e apresentável aos *stakeholders* que enderece os riscos técnicos mais críticos serão desenvolvidos durante esta fase para provar que a estratégia arquitetural funciona na prática, e não apenas na teoria, reduzindo, assim, todos os riscos técnicos no projeto.

Assim que o *design* estiver totalmente elaborado e os planos do projeto estiverem esclarecidos, novos riscos de projeto podem provavelmente ser identificados. Um elemento chave neste planejamento é o conjunto de iterações do projeto; cada iteração possui um conjunto de metas e critérios de saída, normalmente para provar um conjunto de requisitos está completo. As primeiras iterações devem focar na eliminação de fatores de riscos identificados durante as fases de Concepção e Elaboração. Planos de teste e validações de requisitos de segurança devem estar maduros e incorporados às metas da iteração. Espera-se que ao final da fase a arquitetura dos requisitos de segurança esteja completa. Em adição, a construção destes requisitos deve ser planejada, e as validações de qualquer requisito deve ser incorporado aos critérios para iterações que ocorram durante a fase de Construção.

Ferramentas de análise de código são úteis a cada iteração para provar certos *milestones* de segurança. Isto garante que erros no código e falhas no *design* sejam encontrados e resolvidos regularmente, reduzindo dramaticamente os riscos de segurança do sistema. Nesta fase podem ser testados algoritmos de criptografia, validação de rotinas, uso inapropriado de APIs, eliminação de senhas *harcoded*, etc. Outro fator importante é a modelagem, que pode usar as facilidades do UMLSec[3], uma extensão da UML para expressar informações relevantes sobre segurança, dentro de diagramas em uma especificação do sistema.

Customização da Fase de Elaboração

Na fase de Elaboração, os princípios de segurança são primordiais, pois descrevem perspectivas e práticas de alto nível para que uma orientação prescritiva da arquitetura e *design* seja elaborada. Os *attack patterns* são capturados formalmente por meio de métodos comuns de ataque e servem como orientações para melhorar a resistência a ataques da arquitetura do *software*. Finalmente, a análise de riscos arquiteturais realiza uma avaliação detalhada dos riscos da arquitetura e do *design* do *software*, e sua habilidade para suportar de maneira segura os requisitos do *software*. Estas três práticas são tarefas exercidas primariamente pelo Arquiteto e pelo Desenvolvedor (*Designer*). A Tabela 3.3 ilustra as atividades mais relevantes executadas por estes papéis do OpenUP nesta fase. A Tabela 3.4

[3] Jurjens, J., Secure Systems Development With UML. Springer, 2005.

mapeia as práticas de segurança com os objetivos desta fase.

Papel (executor primário)	***Tarefas***	***Saídas (artefatos)***
Desenvolvedor	*Design* da Solução (seguindo orientações dos princípios de segurança); implementar a solução; criar e integrar os *builds;* implementar e executar testes do desenvolvedor (unitários)	*Design, Build,* Testes do Desenvolvedor (unitários) e Implementação
Arquiteto	Refinar a arquitetura (Análise de Resistência a Ataques, Análise de Ambigüidade e Análise de Fraquezas); captura dos padrões de ataque; avaliação da análise de risco; orientações prescritivas específicas da tecnologia em uso para integrar a segurança na arquitetura e no *design*	Arquitetura (*Design* da Análise de Riscos, *Attack Patterns* e Orientações de Segurança)

Tabela 3.3 – Customização das atividades da fase de Elaboração para contemplar artefatos e atividades de segurança.

Objetivos da fase	***Atividades que abordam os objetivos***	***Práticas de segurança***
Obter um entendimento mais detalhado dos requisitos e Mitigar riscos essenciais, e produzir um planejamento preciso e estimativas de custos	• Identificar e refinar requisitos • Planejar e gerenciar a Iteração Detalhar um conjunto de requisitos (um ou mais casos de uso, cenários ou requisitos *system-wide*). Iniciar a iteração e permitir que os membros da equipe iniciem as atividades de desenvolvimento. Monitorar e comunicar o status do projeto para *stakeholders* externos. Identificar e manipular exceções e	Princípios e Orientações de segurança, (McGraw et al., 2008). Estabelecimento do documento de Políticas de Segurança (Paes, 2007).

	problemas.	
Projetar, implementar, validar e delinear uma arquitetura	• Desenvolver a arquitetura • Desenvolver Incremento para a Solução • Testar a solução Desenvolver requisitos arquiteturalmente significativos e que foram priorizados para a iteração. Projetar, codificar, testar, e integrar a solução para um requisito dentro de um dado contexto. De uma perspectiva do sistema, testar a avaliar os requisitos desenvolvidos.	Análise de riscos arquiteturais e levantamento dos padrões de ataque (McGraw et al., 2008).

Tabela 3.4 – Mapeamento entre os objetivos da fase de Elaboração e as práticas de segurança.

Common Criteria na Fase de Elaboração

Desenvolvedores podem usar a Parte 2 da norma como referência para interpretação de declarações dos requisitos funcionais e formulação de especificações formais para os TOEs. Os Testadores podem usar como referência para interpretação das declarações dos requisitos funcionais. De maneira geral, o CC auxilia no refinamento dos requisitos de segurança e na elaboração dos PPs.

Os PPs são documentos que expressam um conjunto de requisitos de segurança para um produto de *software* (TOE) que esteja em conformidade com necessidades específicas dos *stakeholders*, e podem ser utilizados como fontes de requisitos de segurança. Para desenvolver um PP os *stakeholders* devem elucidar, definir, e validar seus requisitos de segurança. Além de servir como fonte de requisitos de segurança para os desenvolvedores, servem como uma fundação na qual os STs podem ser desenvolvidos. Este documento deve apresentar uma declaração concisa dos requisitos de segurança do produto.

Os PPs devem ser constantemente atualizados após certos eventos, tais como alterações nas políticas de segurança, identificação de novas ameaças, novas tecnologias, etc. O OpenUP possui tarefas bastante convenientes para esta atividade. O planejamento da iteração pode ser estendido para incluir avaliações de segurança em cada iteração. O artefato

de Lista de Riscos deve ser revisado a cada iteração, priorizando riscos de segurança.

SSE-CMM na Fase de Elaboração

Ao final da elaboração a arquitetura deve estar estável. Identificados os padrões de ataque, estes devem ser incorporados a esta arquitetura. Estes padrões auxiliam na enumeração e mitigação das vulnerabilidades. No SSE-CMM, a PA10 (Determinar as Vulnerabilidades de Segurança) tem como propósito determinar analiticamente as vulnerabilidades de segurança associadas com o sistema. Esta PA inclui atividades como analisar ativos do sistema, definir vulnerabilidades e suscetibilidades específicas, e fornecer uma avaliação geral das vulnerabilidades do sistema. Os termos associados com os riscos de segurança e avaliações de vulnerabilidades são usados diferentemente em muitos contextos. Para o propósito deste modelo, suscetibilidades referem-se às vulnerabilidades exploráveis, buracos de segurança, ou implementação de defeitos dentro de um sistema que são mais prováveis de serem atacados por uma ameaça. Estas suscetibilidades são independentes de qualquer instanciação de ameaça ou ataque. Uma vez que estas suscetibilidades estejam associadas com uma ameaça específica e uma probabilidade de ser explorada, elas são referidas como vulnerabilidades. Este conjunto de atividades é realizado a qualquer tempo no ciclo de vida de desenvolvimento para suportar a decisão para desenvolver, manter, ou operar o sistema dentro do ambiente conhecido.

***Milestones* de Segurança**

Ao final da fase de Elaboração e início da fase de Construção, os seguintes objetivos de segurança devem ser alcançados (Dent, 2008):

- O *design* dos *Abuse Cases* e dos requisitos deve estar completo.
- Definidos o Plano de Projeto e as iterações para a fase de Construção.
- Estratégias de planos de testes de segurança em elaboração, e devem ser finalizados na próxima fase.
- Se necessário, customizar testes para Políticas de Segurança.
- Construir um sistema funcional o mais cedo possível.

CONSTRUCTION (Construção)

Na construção, é preciso ocupar-se com a maioria das implementações, conforme o projeto avança, de uma arquitetura executável para a primeira versão operacional do sistema. O esforço maior nesta fase (Figura 3.3) é empregado nas atividades de codificação e testes.

Vários *deploys* internos e liberações de versões ALFA são feitos para assegurar que o sistema é utilizável e aborda as necessidades do usuário. Ao final desta fase, entrega-se uma versão BETA totalmente funcional do sistema, incluindo instalação, documentação de suporte e material para treinamento (embora o sistema ainda precise de refinamentos de funcionalidades, desempenho e de qualidade geral). Nesta fase o uso de ferramentas é essencial, tanto para análise e verificação de código quanto para automatização de testes. Uma prática dos métodos ágeis é o TDD (*Test Driven Development*).

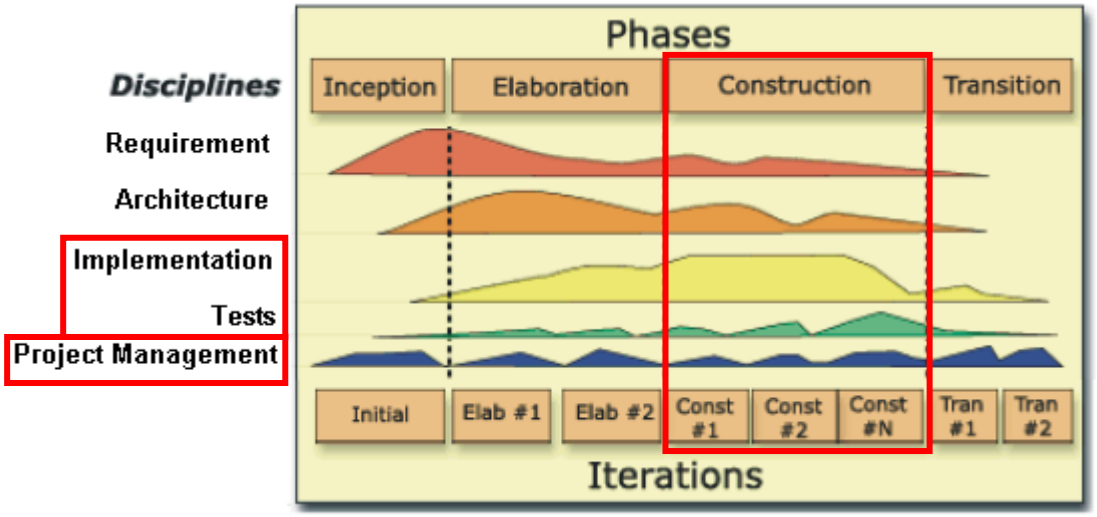

Figura 3.3 – Esforço das atividades da fase de Construção.

O termo construção de *software* não diz respeito apenas à codificação, antes, se refere a uma criação detalhada de um *software* funcional, por meio de uma combinação de codificação, verificação, testes unitários, teste de integração, e atividades de *debugging* (SWEBOK, 2004). Esta atividade se relaciona fortemente com as atividades de *design* e testes, porque, além de o processo de construção envolver muitas destas atividades, utiliza as saídas do *design*, e fornece entradas para os testes.

Para o início desta fase a arquitetura deve estar estável, o que permite que os requisitos restantes possam ser implementados em cima desta arquitetura. Outra vantagem de validar a arquitetura, e eliminar tantos riscos quanto possíveis durante a elaboração, é que isso fornece maior previsibilidade na Construção, o que permite que o gerente de projetos foque na eficiência do time e na redução de custos.

A fase de Construção é onde a maior parte do código é, de fato, desenvolvida. A fase é dividida em uma série de iterações onde certas metas são definidas a cada uma, tais como

validação de características de casos de uso, validação de casos de uso arquiteturalmente significativos, etc. Estes *milestones* são definidos na fase de Elaboração. Casos de uso de segurança, assim como qualquer outro caso de uso, devem ser validados por meio de *milestones* das iterações durante o curso da fase de Construção. Com isso, é possível fazer medições para verificar, além do progresso do projeto, a segurança do sistema.

Além da validação dos casos de uso, durante a fase de Construção a equipe de desenvolvimento deverá, também, assegurar que não sejam introduzidos defeitos no *software*. Isto é alcançado por meio de testes contínuos no código fonte e introdução de melhores práticas de codificação. Funcionalidades são continuamente implementadas, testadas e integradas, resultando em *builds* que são mais e mais completos e estáveis. Uma versão BETA pode ser liberada ao final desta fase. Entregar a versão atual é foco da próxima fase (Transição).

Ferramentas podem auxiliar em continuamente verificar a segurança conforme o desenvolvimento é feito. Desta forma o código se torna tão seguro quanto possível antes que seja checado para ser colocado no *build*. Uma das metas desta fase é produzir um sistema *build-able* tão cedo seja possível. Quanto mais cedo o sistema possa ser construído, mais cedo os riscos de integração são eliminados e o sistema pode ser testado de maneira holísticas de forma regular. Da perspectiva e segurança isto é crítico porque a integração de diferentes componentes pode criar diferentes vulnerabilidades de segurança em oposição à separação de componentes agindo independentemente.

Práticas de Codificação Segura

Todos os projetos de *software* garantem que haja um artefato em comum: o código fonte. Devido a esta garantia básica, faz sentido centralizar a atividade garantia *software* ao redor do código. Além disso, um grande número de problemas de segurança é causado por defeitos simples que podem ser localizados no código (exemplo: *buffer overflows* e exceções não tratadas). Em termos de defeitos e falhas, a revisão do código se refere a encontrar e corrigir defeitos. Juntos com a análise de riscos arquiteturais, a revisão de código para segurança estão no topo da lista das melhores práticas de segurança de *software* (McGraw, 2006).

Ferramentas são importantes para realização destas análises e testes. Realizar testes de regressão, por exemplo, pode ser tedioso, difícil e maçante. Analistas que praticam revisão de código geralmente cometem muitos erros (McGraw, 2006). Isso é auxiliado por ferramentas

que permitam automatização de testes. O método ágil incentiva esta prática por meio do TDD (discutido mais adiante).

Dentre as práticas seguras desta fase, destacam-se:

- Código seguro: utilizar práticas comprovadamente seguras para auxiliar na redução de defeitos de software introduzidos durante a codificação.
- Revisão de código fonte para deficiências de segurança: realizar revisões de código fonte usando ferramentas de análise estática, análises métricas, e revisão manual para minimizar defeitos de segurança no nível da codificação.
- Aspectos únicos de testes de segurança de software: entender as diferenças entre testes de segurança de software e testes de software tradicionais, e planejar como melhor abordar estas diferenças (incluindo pensar como um atacante e enfatizar como exercitar o que o software não deve fazer).
- Casos de testes funcionais para segurança: construir casos de testes funcionais (utilizando várias técnicas) que demonstram a aderência do software com seus requisitos funcionais, incluindo requisitos de segurança (requisitos positivos).
- Casos de teste de segurança baseados em riscos: desenvolver casos de teste baseados em riscos (utilizando, por exemplo, abuse cases, padrões de ataque, ou modelos de ameaças) que exercitam erros comuns, deficiências suspeitas no software, e mitigações que têm a intenção de reduzir ou eliminar riscos, visando assegurar que eles não possam ser frustrados (requisitos negativos).
- Casos de testes utilizando várias estratégias de testes de segurança: usar um complemento das estratégias de testes incluindo testes *White Box* (baseado no conhecimento avançado do código fonte), testes *Black Box* (focando no comportamento do software visível externamente), e testes de penetração (identificando e objetivando vulnerabilidades específicas no nível do sistema).

TDD e *Pair Programming*

O TDD é uma prática encorajada pelos métodos ágeis (particularmente pelo XP). Trata-se de uma técnica de desenvolvimento de *software* que usa iterações curtas de desenvolvimento baseadas em casos de testes previamente implementados, estes casos de teste definem melhorias desejadas ou novas funcionalidades. Cada iteração produz um código necessário para passar nos testes da iteração. Finalmente, o programador ou o time fazem *refactoring* no código para acomodar as mudanças. Um conceito chave no TDD é que preparar os testes antes da codificação facilita as alterações feitas com base nos *feedbacks*. O

TDD não se trata de um método de teste, e sim de um método de *design* de *software*.

Embora de forma limitada, o TDD pode auxiliar na avaliação de vulnerabilidade, mas testes não são suficientes, uma vez que testes não podem garantir que todos os problemas foram removidos. A propriedade coletiva do código e padronizações na codificação tornam mais pessoas aptas para fiscalizar todas as partes do sistema em busca de vulnerabilidades, que também são facilitadas pelas práticas do XP de design simples e *refactoring* que auxiliam a manter o código sob controle. Com os métodos ágeis, fornecer estes argumentos de garantia pode ser difícil, mas no caso do OpenUP isso é facilitado se os artefatos corretos forem fornecidos. Modificações no processo são necessárias, como o OpenUP pode ser estendido para incorporar mais artefatos, mas é preciso cuidado para que o processo não seja onerado para que não perca a agilidade. Isso irá depender que qual nível de garantia foi requisitado. Os níveis mais baixos de garantia, como EAL 1 e EAL 2 do *Common Criteria*, se encaixam bem nas práticas sugeridas pelos métodos ágeis (atividades de testar-codificar-testar), mas para níveis mais altos de EAL é preciso um grau maior de verificação, que geralmente os métodos ágeis, como o XP, não incluem extensões de segurança.

O *pair programming* é uma técnica de desenvolvimento na qual dois programadores trabalham juntos em um mesmo computador. Enquanto um digita o código, o outro o revisa conforme é digitado. A pessoa que digita é chamada de *driver*, e a pessoa que revisa é chamada de *observer* ou *navigator*. Os dois programadores devem trocar de papéis freqüentemente.

Durante a codificação, a prática do *pair programming* pode ser utilizada como uma atividade de garantia, uma vez que um de seus propósitos é remover a necessidade de revisões. No entanto, mais importante do que práticas de segurança, é preciso que os desenvolvedores tenham bom conhecimento sobre vulnerabilidades (exemplo: *buffer overflows* e SQL-injection). Caso contrário, eles não irão perceber problemas ao utilizar esta técnica. A cultura da segurança, porém, é mais bem disseminada na equipe se um especialista executar esta atividade com desenvolvedores diferentes.

Customização da Fase de Construção

Tanto o SSE-CMM quanto o CC não fornecem práticas técnicas para codificação de *softwares* seguros, no entanto, diversas técnicas são propostas por especialistas, e algumas práticas dos métodos ágeis podem ser adaptadas para a codificação e o *design* seguros, conforme foi discutido anteriormente. A Tabela 3.5 propõe uma customização das tarefas e

artefatos de construção associadas com segurança, enquanto a Tabela 3.6 faz um mapeamento entre as atividades da fase e as práticas seguras associadas.

Papel (executor primário)	***Tarefas***	***Saídas (artefatos)***
Desenvolvedor	*Design* da solução; implementar testes do desenvolvedor (TDD); implementar a solução (*Pair programming*); integrar e criar o *build*; executar testes do desenvolvedor	*Build*, *design*, testes do desenvolvedor (automatização de testes) e codificação
Testador	Criar casos de teste; implementar testes; executar testes (incluir testes automatizados)	Casos de teste, Scripts de teste (incluir testes automatizados) e registro de execução de testes

Tabela 3.5 – Customização das atividades da fase de Construção para contemplar artefatos e atividades de segurança.

Objetivos da fase	***Atividades que abordam os objetivos***	***Práticas de segurança***
Desenvolver iterativamente um produto completo que esteja pronto para a transição para os usuários finais Minimizar os custos de desenvolvimento e alcançar um algum grau de paralelismo	• Identificar e refinar requisitos • Desenvolver um incremento para a solução • Testar a solução • Planejar e gerenciar a iteração	TDD com casos de teste de segurança. *Pair programming* para revisão constante do *software*.

Tabela 3.6 – Mapeamento entre os objetivos da fase de Construção e as práticas de segurança.

Milestones

Os seguintes objetivos de segurança devem ter sido alcançados ao final da fase de Construção (Dent, 2008):

- Arquitetura e *design* seguros estão totalmente realizados.

- Todos os requisitos de segurança são validados durante o curso das iterações de construção.
- Análise contínua do código fonte é automatizada para assegurar que vulnerabilidades resultantes de práticas pobres de codificação tenham sido introduzidas.
- Desenvolvedores devem estar melhor educados sobre melhores práticas de codificação segura.

TRANSITION (Transição)

Na fase de transição deve-se assegurar que o *software* atenda as necessidades de seus usuários, isso é feito testando-se o produto em preparação para liberação, e fazendo ajustes e correções baseados em *feedbacks* dos *stakeholders*. Neste ponto do ciclo de vida, este *feedback* foca principalmente em ajustes finos, configuração, instalação, e aspectos de usabilidade, uma vez que todos aspectos mais importantes da estrutura foram trabalhados nas fases iniciais do ciclo de vida de desenvolvimento. Na fase operacional do *software* é preciso realizar atividades de contenção de defeitos, minimizando as conseqüências das falhas de uma aplicação.

Nas iterações finais desta fase, os *stakeholders* devem verificar se o sistema realiza aquilo que foi prometido, e se todas as funcionalidades estão presentes. Antes da entrega completa, o *software* deve ser validado (todas as funcionalidades presentes) e verificado (conformidade com os requisitos) o máximo possível e testes finais de aceitação devem ser conduzidos. Estas são atividades da Garantia de Qualidade, e podem ser aproveitadas também para a segurança.

As atividades desta fase se concentram nos testes de aceitação e nas implementações de correções (ajustes) (Figura 3.4). Seu propósito principal é entregar o *software* com todas as funcionalidades pretendidas, e tem como *milestone* principal a liberação do produto. Como parte dos testes de aceitação, deve-se considerar os testes de segurança.

Os testes propostos para a garantia de qualidade são atividades com o objetivo de assegurar que o *software* cumpre seus requisitos funcionais de negócio. Geralmente, são os casos de uso e os requisitos que guiam estes testes, portanto, a ênfase maior está nas características e funcionalidades de negócio da aplicação. As atividades de testes propostas pelo OpenUP, para a fase de transição, não se aplicam diretamente aos testes de segurança. É importante verificar se as características de segurança levantadas por requisitos de segurança estão implementadas corretamente por meio de testes de conformidade (atividades de *White*

hat hackers). Mas, mais importante que isso, é preciso garantir que ataques intencionais não possam facilmente comprometer o funcionamento do *software* (atividades de *Black hat hackers*).

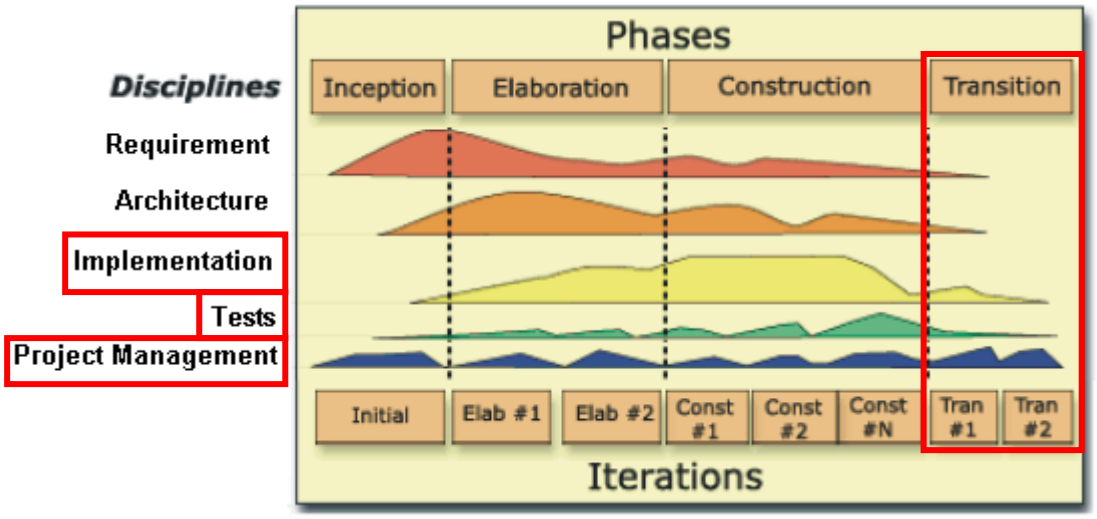

Figura 3.4 – Esforço das atividades da fase de Transição.

Grande parte dos defeitos e vulnerabilidades relacionados com a segurança não estão diretamente relacionados com as funcionalidades de segurança explicitadas nas iterações iniciais do projeto. Pelo contrário, aspectos de segurança envolvem o mau uso intencional de uma aplicação descoberto por um atacante. Da mesma forma que testes funcionais são utilizados para cobrir requisitos positivos, os testes de segurança (ou de penetração) devem cobrir requisitos negativos. Isto significa que o Testador deve sondar direta e profundamente os riscos de segurança para determinar como o sistema se comporta sob ataque.

Testes para verificar requisitos negativos são complicados. É fácil testar se uma característica funciona ou não, mas é muito difícil demonstrar se um sistema é seguro ou não sob ataque malicioso. Se os testes não descobrem falhas, isso apenas prova que algumas falhas não ocorrem nas condições particulares do teste, e não que falhas não existem. O mesmo se pode afirmar dos testes de penetração: se um teste de penetração não encontrar vulnerabilidades isso não significa que a aplicação esteja imune a ataques.

Testes de penetração suportados por ferramentas auxiliam na produção de ataques à superfície e na análise de código. O rigor das avaliações e dos testes de aceitação irá depender dos padrões aplicados.

Defesa da Aplicação

Empregar práticas focadas em detectar e mitigar fraquezas no *software* depois que ele é entregue é, geralmente, referido como defesa de aplicação. As técnicas seguem tipicamente os seguintes aspectos:

- Estabelecer um limite de proteção ao redor da aplicação que reforça as regras, define entradas válidas, reconhece, bloqueia ou filtra entradas que contenham padrões de ataque reconhecíveis.
- Limitar a extensão e o impacto do dano que pode ser resultado de um *exploit* de vulnerabilidade na aplicação.
- Descobrir pontos de vulnerabilidade na aplicação desenvolvida através de testes *Black box* para auxiliar desenvolvedores e administradores a identificarem contramedidas necessárias

Medidas reativas de defesa da aplicação têm mais a intenção de reforçar as fronteiras ao redor da aplicação do que abordar as verdadeiras vulnerabilidades dentro da aplicação. Em alguns casos, estas medidas são aplicadas provisoriamente para aplicações até que um *patch* de segurança ou nova versão sejam liberados. Em *softwares* que incluem componentes adquiridos ou reutilizados, as técnicas de defesa e as ferramentas podem ser apenas contramedidas de custo efetivo para mitigar vulnerabilidades nestes componentes.

As técnicas utilizadas para defender uma aplicação apenas ajudarão a mitigar vulnerabilidades óbvias. Fraquezas mais críticas, incluindo falhas de implementação e *design*, não são normalmente detectáveis desta maneira. Uma perspectiva de segurança do *software*, porém, não apenas incorpora técnicas de proteção, mas também aborda as necessidades de especificar, projetar, e codificar uma aplicação co uma superfície de ataque mínima.

O que se deve levar em consideração é que um processo de desenvolvimento disciplinado, repetível e com a segurança intensificada, deve ser instituído para assegurar que as medidas de defesa do *software* sejam usadas apenas porque são determinadas na fase de *design* para serem as melhores abordagens para resolução de problemas de segurança, não porque são as únicas abordagens possíveis depois que o *software* é entregue. Assim, usar práticas de Engenharia de Segurança pode ser útil para proteger *softwares* projetados com segurança no *deploy* ao reduzir sua exposição a ameaças em vários ambientes operacionais.

Customização da Fase de Transição

A elaboração de testes de segurança iniciada na fase de construção deve ser terminada na fase de transição. A Tabela 3.6 propõe uma customização da fase de Transição para incluir práticas de segurança.

Papel (executor primário)	***Tarefas***	***Saídas (artefatos)***
Testador	Implementar testes (incluir testes de penetração e/ou de *fuzzing*); executar testes (automatização de testes)	Script de Testes (Scripts para Testes de Penetração e/ou de *Fuzzing*), Registro de Testes (Inspeção de Requisitos de Segurança)
Desenvolvedor	Fazer o *design* da solução; implementar testes do do desenvolvedor; implementar a solução (*Pair programming*); executar testes do desenvolvedor; integrar e criar o *build* (incluir na implementação dos testes os testes de integração)	*Design*, Testes do Desenvolvedor, implementação (Verificação dos *Builds*), Registro de Testes; *Build*

Tabela 3.7 – Customização das atividades da fase de transição para contemplar artefatos e atividades de segurança.

***Common Criteria* na Fase de Transição**

Desenvolvedores podem utilizar o CC como referência para interpretação das declarações de requisitos de garantia e determinação de garantias dos TOEs. Testadores podem utilizar como referência para interpretar as declarações de requisitos de garantia. O CC ajuda a determinar se, de fato, os requisitos de segurança levantados nas iterações iniciais do projeto atingem seus objetivos.

Segundo o CC, a avaliação do TOE é colocada em prática ao ser verificada sua conformidade com os critérios de avaliação contidos na Parte 3 (EALs). Para isso, utiliza-se um ST avaliado como base. A meta desta avaliação é demonstrar que o TOE atinge os requisitos de segurança contidos no ST. Apenas lembrando que o ST é um documento que aponta propriedades de segurança do TOE a ser avaliado. Este documento contém uma lista de requisitos funcionais de segurança os quais o TOE deve satisfazer.

O resultado da avaliação de um TOE será uma demonstração do grau de conformidade do TOE com os requisitos. Este grau pode indicar se os requisitos foram atingidos de forma

satisfatória ou não. Um TOE que satisfez os requisitos torna-se elegível para entrar em produção.

SSE-CMM na Fase de Transição

O SSE-CMM possui metas bastante claras que podem ser adaptadas facilmente para a fase de Transição.

A PA03 (Verificar e Validar a Segurança) tem como propósito garantir que a solução seja verificada e validada com respeito à segurança. Soluções são validadas em comparação com as necessidades operacionais de segurança dos *stakeholders*. Isso é feito por meio dos testes de aceitação que foram elaborados tendo como entrada os requisitos de segurança (derivados dos requisitos não funcionais de segurança).

A PA04 (Atacar a Segurança) é também chamada de Testes de Penetração, seu propósito é identificar vulnerabilidades existentes do sistema e validar o potencial para *exploits* destas vulnerabilidades. Vulnerabilidades são descobertas por meio de ataques ativos contra o sistema. Estes testes também são elaborados na fase de requisitos, e têm como entrada os requisitos não funcionais de segurança (negativos), mas também são feitos através dos modelos de ataque.

Objetivos da fase	***Atividades que abordam os objetivos***	***Práticas de segurança***
Testes BETA para validar se as expectativas dos stakeholders são atingidas *e* *Alcançar conformidade dos stakeholders para assegurar que a entrega esteja completa*	• Tarefas em curso • Desenvolver incremento para a solução • Planejar e Gerenciar a Iteração • Testar a Solução Executar tarefas em curso que não fazem, necessariamente, parte do cronograma do projeto. Projetar, implementar, testar, e integrar a solução para um requisito dentro de um determinado contexto. De uma perspectiva do sistema, testar e avaliar os requisitos desenvolvidos.	Teste de Penetração (PA04) para verificar a conformidade dos requisitos de segurança levantados. Avaliar resultados das métricas de qualidade e segurança. O grau de conformidade da segurança é desenvolvido ao se conduzir as avaliações do CC (EALs) e do SSE-CMM (PA03). Esta atividade é colocada em prática pela garantia de qualidade, com a participação dos *stakeholders* e do especialista em segurança (Piattini, 2006).

	Iniciar a iteração e permitir que os membros do time obtenham suas tarefas de desenvolvimento. Monitorar e comunicar o status do projeto para *stakeholders* externos. Identificar e manipular exceções e problemas.
Melhorar o desempenho de futuros projetos por meio de lições aprendidas	• Planejar e Gerenciar a Iteração Iniciar a iteração e permitir que os membros do time obtenham suas tarefas de desenvolvimento. Monitorar e comunicar o status do projeto para *stakeholders* externos. Identificar e manipular exceções e problemas.

Tabela 3.8 – Mapeamento entre os objetivos da fase de Concepção e as práticas de segurança.

Milestones

Durante a fase de Transição, os seguintes *milestones* de segurança devem ser finalizados (Dent, 2008):

- Testes de aceitação final completados antes do *deploy*.
- Estabelecida a preparação para evoluções e manutenções do sistema.
- Riscos da aplicação são gerenciados como parte de um portfólio maior das aplicações entregues.

4. Conclusões

4.1. A Falácia da Segurança

Absolutamente nenhum processo garante que um *software* esteja livre de falhas e que seja seguro (McGraw et al., 2008). Da mesma forma, como mencionado na introdução, apenas mecanismos de segurança colocados isoladamente não garantem a segurança. Os mecanismos de segurança devem ser utilizados para se atingir os objetivos de segurança, e não podem limitar ou prender o desenvolvimento. Abordagens técnicas devem ir além das características genéricas e óbvias, elas devem fazer parte da estrutura do *software* para que forneça segurança suficiente. No entanto, geralmente esta é a única abordagem dos desenvolvedores para inclusão de segurança no *software*.

Segurança é uma propriedade emergente de um sistema e não pode ser adicionada quando todas as outras propriedades foram desenvolvidas, muito menos ser colocada como um *patch* após um ataque ter ocorrido no ambiente operacional. Ao invés disso, segurança deve ser construída dentro do produto desde a sua fundação, como parte crítica do *design* desde o início do projeto (levantamento de requisitos) e incluída em cada fase subseqüente, até sua conclusão e entrega.

Pensar na segurança como algo além das características normativas, e incorporar estas características em todo o processo de desenvolvimento, é a solução com melhor custo-benefício para incorporar a segurança no *software*. Cada vez que um novo requisito, característica, ou caso de uso é criado, o desenvolvedor, auxiliado por um especialista, deve gastar tempo pensando em como esta característica pode ser usada de forma errada ou explorada de forma maliciosa intencionalmente. Profissionais que sabem como características são atacadas e como proteger o *software* devem executar papéis neste tipo de análise.

4.2. Segurança e Métodos Ágeis

Pelo que se observa das práticas e atividades concernentes à segurança, e à qualidade, o processo de desenvolvimento exige disciplina e formalidade. Disciplina, pois é necessário que uma série de validações e verificações sejam constantemente feitas, de forma repetitiva e contínua, para garantir resultados mensuráveis. E formalidade, pois é necessário que estas validações e verificações sejam analíticas, rastreáveis e formalizadas em artefatos.

Considerando os padrões estudados (*Common Criteria* e SSE-CMM) percebe-se que para atingir um nível de confiança e maturidade é preciso formalidade, mas, além de consumir muito tempo, maior foco é dado ao processo. Por outro lado, como ressaltado por Wäyrynen (2006), a documentação de segurança deve ser feita no início do processo, quando o domínio ainda não está completamente entendido, o que pode tornar o documento obsoleto. A questão a ser respondida, portanto, é como o desenvolvimento ágil resolve estas duas questões.

4.2.1. Disciplina

Os métodos ágeis são bastante disciplinados. Todos os métodos propostos (SCRUM, XP, e mesmo o OpenUP) exigem interações freqüentes (geralmente diárias) entre os participantes do projeto (equipe de desenvolvimento e *stakeholders*) para priorização de requisitos, gerenciamento do progresso, *feedbacks*, etc. Os micro-incrementos do OpenUP são exemplos disso: um micro-incremento é planejado para um período curto de tempo (alguns dias, e no máximo uma semana). Mesmo que os objetivos de uma iteração não tenham sido plenamente atingidos, alguma coisa deve ser entregue, mensurada e avaliada, permitindo que a próxima iteração seja planejada de acordo. A produção contínua de *software* e o *feedback* constante dos *stakeholders* garante que validações e verificações sejam realizadas de forma efetiva, e, conseqüentemente, torna mais fácil a medição do progresso do desenvolvimento. Esta questão parece estar bem resolvida no desenvolvimento ágil.

4.2.2. Formalidade

Alguns dos princípios dos métodos ágeis parecem divergir da formalidade exigida pelo processo de desenvolvimento seguro. Por exemplo, um *design* baseado em princípios seguros que trata de riscos de segurança identificados durante uma atividade inicial, tal como modelagem de ameaças, é uma parte essencial de processos preocupados com a segurança, mas entra em conflito com os princípios de requisitos e *design* emergentes dos métodos ágeis. O RUP, por exemplo, é considerado um processo bastante formal, pois possui alta cerimônia (número alto de artefatos), isso fornece a formalidade requerida para um desenvolvimento seguro. Os métodos ágeis, por outro lado, focam muito mais nas pessoas e nas interações do que nos processos e ferramentas. Um de seus princípios define que o *software* funcionando é mais importante do que uma documentação completa e detalhada. No entanto, existe o entendimento errado do conceito de desenvolvimento ágil no qual se pensa que nenhuma documentação é criada, o que, como conseqüência, comprometeria a segurança do projeto.

Isso não é verdade. O método ágil diz que a documentação deve ser suficiente, ou seja, deve possuir cerimônia suficiente. Por suficiente entende-se: fazer a documentação se e quando for de fato necessária (Ambler). É possível que em projetos muito pequenos não seja feita documentação alguma, mas isto ocorre em casos raros, sendo que a regra é a geração de modelos e documentos. A documentação também deve ser emergente, em outras palavras, deve surgir apenas quando realmente necessária ou requisitada explicitamente pelos *stakeholders*.

O problema com os métodos ágeis é que os seus conselhos e orientações podem ser difíceis de serem traduzidos para uma determinada situação de um projeto, uma vez que são baseadas somente em conhecimento tácito e em livros didáticos. É essencial enumerar quais artefatos e práticas da Engenharia de Segurança são necessários e suficientes para que o desenvolvimento ágil seja beneficiado sem que seus princípios sejam desrespeitados. Por esta razão os padrões de segurança foram estudados, e agora devem ser relacionados com as regras e princípios dos métodos ágeis.

4.2.3. Padrões de Segurança e o OpenUP

A meta de se especificar requisitos de segurança é avaliar os riscos de segurança e definir características de segurança necessárias para que seja estabelecido um nível de risco aceitável. Ambos os padrões estudados requerem avaliações de riscos. Esta é a visão predominante no campo da Engenharia de Segurança, e vários especialistas argumentam que não é possível adicionar segurança após o desenvolvimento, ou seja, considerá-la após o levantamento de requisitos e modelagem (Wäyrynen, 2006), mas ela tem de ser construída e especificada desde as fases iniciais.

O método ágil XP não fornece um processo que suporte esta área (requisitos não funcionais ou requisitos *system wide*), no OpenUP isto é previsto no artefato *Requisitos não expressos por casos de uso*, executado pelo papel do Analista nas atividades: *Detalhar Requisitos não Expressos por Casos de Uso* e *Identificar e Esboçar Requisitos.* No entanto, enumerar requisitos não funcionais é complicado, porque: os *stakeholders* focam-se primariamente nos requisitos funcionais, e, geralmente, estes requisitos são difíceis de serem quebrados em porções passíveis de testes e estimativas. O papel do Analista é importante para identificação e estruturação destes requisitos.

No estudo do OpenUP verifica-se que o processo *as is* pode facilmente preencher os requisitos de um EAL 3, o XP chega a preencher os requisitos dos EAL 1 e EAL 2

(Wäyrynen, 2006). No entanto, para aumentar o nível de garantia é necessário adaptar e/ou adicionar práticas e artefatos, como visto no Capítulo 6. É possível também que haja a necessidade de inclusão de novas iterações ao projeto. Independente da abordagem, o processo precisa de modificações para atender às atividades da Engenharia de Segurança.

OpenUP e o SSE-CMM

O SSE-CMM é projetado como uma ferramenta para medir a capacidade de uma organização de desenvolver sistemas seguros. As PAs são úteis como *checklists* quando se analisa métodos de desenvolvimento de *software* de uma perspectiva de Engenharia de Segurança. O SSE-CMM também é útil na especificação do nível de maturidade, ou seja, o nível que uma organização realiza as PAs.

O SSE-CMM traça perfis de uma organização baseados na premissa de que diferentes tipos de organizações não requerem o mesmo nível de maturidade em todo o processo que esta organização usa, e ainda podem entregar produtos e serviços os quais os consumidores tenham um alto nível de confiança. Obviamente, em um processo de desenvolvimento seguro nem todas as PAs serão utilizadas, é preciso analisar quais são as necessidades requeridas e como o padrão pode auxiliar em cada caso.

As PAs fornecidas pelo SSE-CMM podem ser mapeadas diretamente por elementos do processo OpenUP. Todos os PAs podem ser exercidos por um ou mais papéis, onde atividades específicas do processo podem ser adaptadas para inclusão de aspectos de segurança, e o mesmo serve para os artefatos. O padrão tem maior enfoque na análise e avaliação dos riscos, por está razão muitas atividades se encaixam no papel da Gerência de Projeto e do Arquiteto.

Todos os papéis estão envolvidos de alguma forma. A gerência deve sempre avaliar os riscos de segurança envolvidos para planejar todo o projeto, ou após avaliar os resultados de segurança, planejar uma iteração. A lista de riscos deve ser revisada constantemente, e serve como demonstração dos aspectos de segurança verificados. O Analista deve levantar os cenários de *Abuse Cases*. O arquiteto deve, além de compor a arquitetura com elementos seguros (mecanismos), rever e incluir aspectos de segurança durante os refinamentos da arquitetura. O desenvolvedor projeta a solução considerando a codificação segura e *patterns* de ataque, e planejando testes de segurança. O testador deve verificar e validar a segurança ao executar os casos de teste. Finalmente, o *stakeholder* deve fornecer as informações necessárias para os requisitos funcionais de segurança e para que demais aspectos sejam abstraídos.

O SSE-CMM fornece um método de avaliação das capacidades em paralelo com o

ciclo de vida de um produto, sistemas ou serviços, ou seja, não em um ponto específico, mas no processo como um todo. Isso o diferencia de outras abordagens de avaliação, mas não é um substituto para as outras abordagens. Neste contexto, ele deve ser usado com técnicas e práticas pontuais (como o *Common Criteria*).

OpenUP e o Common Criteria

O *Common Criteria* pode ser descrito como uma linguagem para expressar requisitos de segurança em um sistema (requisitos funcionais), e seu processo de desenvolvimento (requisitos de garantia). Os requisitos funcionais no CC são usados como um repositório padrão de requisitos, e podem ser combinados, adaptados e estendidos para formar uma especificação do total de requisitos de segurança de um sistema. Juntamente com a documentação relativa a ameaças, riscos e o ambiente do sistema, os requisitos são colhidos no documento de ST, e serve como uma especificação dos requisitos relativos à segurança. O CC foca mais nas atividades de requisitos da Engenharia de Segurança.

O CC pode ser adaptado dentro das iterações do processo. Desta forma, os TOEs são os artefatos (manifestações físicas dos requisitos) produzidos em uma iteração e que devem ser avaliados: *builds*, lista de riscos, arquitetura, etc. Estes elementos devem constar no artefato Lista de Itens de Trabalho do OpenUP. Mas, a ênfase maior é na elaboração dos requisitos de segurança. Como dito anteriormente, o papel do Analista é importante para identificação e detalhamento dos requisitos não funcionais. O CC pode auxiliar nesta atividade, pois sugere a quebra de um requisito muito vago ou genérico demais (exemplo: “o sistema deve proteger informações sensíveis”) em requisitos separados e mais concretos. O PP expressa as necessidades de segurança e é fornecido pelos *stakeholders*, cabe ao analista extrair os requisitos de segurança (o que deve ser protegido) e elaborar o ST (propriedades de segurança a serem avaliadas, ou seja, o que será produzido pelo desenvolvimento) juntamente com o Testador. Finalmente, é papel do Gerente de Projeto estabelecer qual o nível de EAL a ser alcançado por meio de elaboração e execução dos testes de segurança de sistema.

A meta da construção de um argumento de garantia é conduzir de maneira clara as necessidades de segurança para que estas sejam satisfeitas. No CC isso significa produzir grandes quantidades de documentos, e realizar e documentar revisões de vários tipos. Quanto maior o EAL, maiores são os requisitos de documentação. Isso pode entrar em conflito com os métodos ágeis que dão maior ênfase ao código. Se a documentação é necessária, isto é, documentação considerada crítica para o trabalho sendo desenvolvido, esta documentação pode ser adicionada como um novo item na Lista de Trabalho a ser considerada no Plano de

Iteração. No entanto, do ponto de vista da segurança, a documentação de garantia é considerada de importância primária, mesmo se não pareça importante aos desenvolvedores. É preciso, então, adaptar o processo para que evidências de conformidade com os requisitos de segurança sejam desenvolvidas, ou fornecer uma maneira alternativa para demonstrar que a segurança foi atingida. No OpenUP, isso pode ser demonstrado através da execução dos casos de teste. Além de demonstrarem as funcionalidades, os testes podem ser elaborados e executados com ênfase em segurança para que se possa mensurar a conformidade do produto com os requisitos de segurança.

4.3. Conclusões

É importante entender o processo de desenvolvimento utilizado para construir um *software* seguro porque, caso contrário, torna-se difícil determinar os pontos fortes e fracos do produto. É também de grande ajuda utilizar *frameworks* comuns para guiar as melhorias no processo, e avaliar a conformidade do processo com um modelo em comum para determinar áreas a serem melhoradas. Modelos de processos promovem medições em comum para processos através do ciclo de vida de desenvolvimento. Estes modelos identificam muitas práticas técnicas e gerenciais. Embora poucos modelos tenham sido projetados desde o início para abordar a segurança, há substanciais evidências de que estes modelos endereçam boas práticas de Engenharia de *Software* para gerenciar e construir *softwares* (McGraw et al., 2008).

Mesmo que um modelo de processo esteja em conformidade com o processo em uso, não há garantia de que o *software* construído esteja livre de vulnerabilidades de segurança não intencionais, ou de códigos maliciosos intencionais. Entretanto, há provavelmente um melhor indício da construção de *software* seguro quando práticas de Engenharia de *Software* são seguidas de maneira sólida com ênfase em bom *design*, práticas de qualidade (tais como inspeções e revisões), uso métodos de testes completos, uso apropriado de ferramentas, gerenciamento de riscos, e gerenciamento de pessoas.

Adotar segurança no *software* em uma equipe de desenvolvimento pode ser um desafio, e por isso deve ser muito bem planejado. É necessária uma mudança cultural que pode ser de difícil adoção, pois são necessárias decisões técnicas importantes e a adoção de práticas e padrões. Isso também pode impactar nos planos de uma organização que dependem de uma aplicação para os negócios e não pretendem ter gastos extras na implantação dos aspectos de segurança.

Iniciativas de segurança no *software* são possíveis e estão caminhando para um grande número de organizações (como mencionado na introdução). As práticas mencionadas têm se mostrado benéficas para implementação da segurança em sistemas. Estas práticas se baseiam em conhecimento comum, e auxiliam nos passos para melhorar a segurança do *software* de modo que as metas de negócio sejam também melhor alcançadas.

Com relação ao método ágil de desenvolvimento, pode-se concluir que não é possível adotar todas as atividades e artefatos sugeridos pelos padrões, e por vezes segurança e agilidade parecem divergir. O excesso de artefatos exigidos pelos padrões e práticas de segurança pode fazer com que a agilidade seja comprometida. O que foi estudado serve para trazer a segurança aos métodos ágeis, no entanto, é necessário que os métodos se adaptem aos aspectos seguros. Equilibrar a formalidade necessária para medir e avaliar a segurança no desenvolvimento é fator importante.

No entanto, é importante que estes métodos sejam considerados. O *Agile Manifesto* pressionou a comunidade de TI para que software de alta qualidade fosse produzido e de forma rápida, a segurança, agora, pressiona os métodos ágeis para que práticas de desenvolvimento seguro sejam incorporadas.

As seguintes observações podem ser feitas:

- Somente a garantia de qualidade de *software* não garante a segurança, pois esta característica requer certas atividades não relacionadas com a qualidade.
- É possível incorporar as práticas de segurança nos métodos ágeis, mas de forma limitada. É necessário, para isso, introduzir rigor e formalidade que podem tirar um pouco da agilidade.
- O processo precisa de um nível de maturidade 3 ou superior para que um grau de segurança adequado seja atingido.
- Padrões ajudam de forma limitada, pois não sugerem práticas técnicas e genéricas, e podem ser difíceis de adaptar para um dado projeto.

4.4. Sugestões para trabalhos futuros

A análise feita neste estudo não foi exaustiva, e o trabalho serve somente como introdução e orientação aos assuntos abordados. Algumas sugestões podem ser feitas para o desenvolvimento de futuros trabalhos.

4.4.1. Evolução e/ou modernização de sistemas

O ciclo de vida de desenvolvimento aqui estudado trata essencialmente de sistemas construídos do zero, mas a inclusão dos conceitos de segurança em sistemas legados também é possível, uma vez que os processos de desenvolvimento também são utilizados para evolução e/ou modernização de sistemas. As técnicas de *refactoring* e reengenharia de sistemas podem ser exploradas para este intuito.

4.4.2. Ênfase em Determinada Fase ou Disciplina

A etapa de levantamento de requisitos é a mais importante e foi pouco explorada neste trabalho. É interessante aprofundar esta questão, porque se trata da fundamentação da segurança no processo, e deve ser levantada corretamente e de forma eficaz para que o desenvolvimento seja seguro.

4.4.3. Segurança Utilizando outro Método Ágil

O OpenUP é considerado um método ágil, e foi utilizado por ser mais formal que os demais. O assunto foi pouco discutido com relação ao método ágil de uma forma geral. Seria interessante verificar as adequações de outras propostas ágeis, como o XP, por exemplo, e como estas podem ser customizadas para incorporar práticas seguras.

5. Referências

5.1. Referências Bibliográficas

PRESSMAN, Roger S. **Engenharia de Software.** 6ª edição, McGrall Hill, 2006 (ISBN-13 9788586804571)

KOSCIANSKI, André; SOARES, Michel dos Santos. **Qualidade de *Software* – Aprenda as metodologias e técnicas mais modernas para o desenvolvimento de *software*.** 2ª edição. Novatec, São Paulo, 2007 (ISBN 9788575221129).

KROLL, Per; MCISAAC, Bruce. **Agility and Discipline Made Easy: Practices from OpenUP and RUP**. Addison Wesley Professional, Boston, MA, 2006 (ISBN-10: 0-321-32130-8, Print ISBN-13: 978-0-321-32130-5)

BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software Architecture in Practice**. Second edition, Addison Wesley, 2003 *(*ISBN 0321154959)

Guide to the Software Engineering Body of Knowledge - A project of the IEEE Computer Society Professional Practices Committee, 2004 version (ISBN 0769523307).

KENETT, Ron S.; BAKER, Emanuel R. **Software Process Quality – Management and Control**. First Edition. Marcel Dekker, Inc., Los Angeles, CA, 1999 (ISBN 0824717333).

MCGRAW, Gary et al. **Software Security Engineering: A Guide for Project Managers.** Addison Wesley Professional, 2008 (ISBN-13 9780321509178)

MCGRAW, Gary. **Software Security: Building Security In**. Boston, MA: Addison-Wesley, 2006.

HOWARD, Michael; LIPNER, Steve. **The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software**. Redmond, WA: Microsoft Press, 2006 (ISBN-10: 0735622140, ISBN-13: 9780735622142).

VIEGA, John; MCGRAW, Gary. **Building Secure Software: How to Avoid Security Problems the Right Way**. Addison-Wesley: 2001

PAES, Carlos E. B.; HIRATA, Celso M. "RUP Extension for the Development of Secure Systems". International Conference on Information Technology (ITNG'07), 2007.

WANG, Q., PFAHL, D., RAFFO, D. M. "Software Process Dynamics and Agility: International Conference on Software Process". Proceedings, ICSP 2007 Minneapolis, MN, USA, May 19-20, 2007.

PIATTINI, Mario; FERNÁNDEZ-MEDINA, Eduardo; MELLADO, Daniel. 2007. "A common criteria based security requirements engineering process for the development of secure information system". Computer standards & interfaces 29 (2007), pp 244-253.

INTERNATIONAL STANDARD - ISO/IEC 9126-1:2001. Software engineering — Product quality — Part 1: Quality model.

INTERNATIONAL STANDARD - ISO/IEC 14598-1:1999. Information technology - Software product evaluation - Part 1: General overview.

Common Criteria. INTERNATIONAL STANDARD - ISO/IEC 15408-1. Information technology — Security techniques — Evaluation criteria for IT security — Part 1: Introduction and general model. First edition 1999-12-01.

5.2. Referências da WEB

DENT, Claudia. "Secure at the Source: Implementing source code analysis in the IBM Rational Software Development Lifecycle". IBM, 2008. Disponível em <www.ibm.com/developerworks/rational/library/edge/08/jan08/dent/index.html>

MEAD, Nancy R. "The Common Criteria". Software Engineering Institute, 2008 Carnegie Mellon University. Disponível em <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/requirements/239-BSI.html>. Acesso em: 26 abr. 2009.

PETEANU, Razvan. “Best Practices for Secure Development”, 2001. Disponível em <http://www.arcert.gov.ar/webs/textos/best_prac_for_sec_dev4.pdf>. Acesso em: 1 fev. 2009.

PURCELL, James. “Defining and Understanding Security in the Software Development Life Cycle”, abr. 2007. Disponível em <http://www.giac.org/resources/whitepaper/application/342.php>. Acesso em: 25 fev. 2009.

VIEGA, John. “Security in the software development lifecycle”. 2004. Disponível em <http://www.ibm.com/developerworks/rational/library/content/RationalEdge/oct04/viega/>. Acesso em: 28 fev. 2009.

LANOWITZ, Theresa. ”Now Is the Time for Security at the Application Level”. Gartner, 2005. Disponível em <http://www.sela.co.il/_Uploads/dbsAttachedFiles/GartnerNowIsTheTimeForSecurity.pdf>. Acesso em: 5 mar. 2009.

AGILE MANIFESTO. “Manifesto for Agile Software Development”. Disponível em <http://agilemanifesto.org>. Acesso em: 7 mar. 2009.

DAVIS, Noopur. “Secure Software Development Life Cycle Processes”. Carnegie Mellon University, 2006. Disponível em <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/sdlc/326-BSI.html>. Acesso em: 10 mar. 2009.

LIPNER, Steve; HOWARD, Michael. “The Trustworthy Computing Security Development Lifecycle”. Microsoft, 2005. Disponível em <http://msdn.microsoft.com/en-us/library/ms995349.aspx>. Acesso em: 5 mar. 2009.

YODER, Joseph; BARCALOW, Jeffrey. “Architectural Patterns for Enabling Application Security”. PLoP, 1997. Disponível em <http://st-www.cs.uiuc.edu/~plop/plop97/Proceedings/yoder.pdf>. Acesso em: 10 mar. 2009.

GOERTZEL, Karen Mercedes et al. “Security in the Software Lifecycle: Making Software Development Processes — and Software Produced by Them — More Secure”. U.S.

Department of Homeland Security, 2006 (Draft version 1.2). Disponível em <http://www.cert.org/books/secureswe/SecuritySL.pdf>. Acesso em: 11 mar. 2009.

GOERTZEL, Karen Mercedes et al. “State of the Art of Software Security Assurance”. IATAC, 2007. Disponível em <http://iac.dtic.mil/iatac/download/security.pdf>. Acesso em: 12 mar. 2009.

WÄYRYNEN, Jaana; BODÉN, Marine; BOSTRÖM, Gustav. “Security Engineering and eXtreme Programming: an Impossible Marriage?”. 2006 Disponível em <http://is.dsv.su.se/PubsFilesFolder/612.pdf>. Acesso em: 13 mar. 2009.

SSE-CMM, “Systems Security Engineering Capability Maturity Model, Model Description” Document Version 3.0. Disponível em <http://www.sse-cmm.org/docs/sse-cmm.pdf>. Acesso em:16 mar. 2009.

BEZNOSOV, Konstantin. “Extreme Security Engineering: On Employing XP Practices to Achieve 'Good Enough Security' without Defining It”. First ACM Workshop on Business Driven Security Engineering. Disponível em <http://hct.ece.ubc.ca/publications/pdf/beznosov-etal-2003.pdf>. Acesso em: 15 mar. 2009.

PEETERS, Johan. “Agile Security Requirements Engineering”. Disponível em <http://johanpeeters.com/papers/abuser%20stories.pdf>. Acesso em: 15 mar. 2009.

AMBLER, Scott W. “Examining the Agile Manifesto”. 2006. Disponível em <http://www.ambysoft.com/essays/agileManifesto.html>. Acesso em: 16 mar. 2009.

OpenUP version 1.5.0.1. EPF 2008. Disponível em <http://epf.eclipse.org/wikis/openup/index.htm>. Acesso em: 9 mar. 2009.

BALDUINO, Ricardo. “Introduction to OpenUP (Open Unified Process)”. Disponível em <http://www.eclipse.org/epf/general/OpenUP.pdf>, Acesso em: 11 mar. 2009.

GUSTAFSSON, B., “OpenUP – The Best of Two Worlds”. Disponível em <http://www.methodsandtools.com/archive/archive.php?id=69 >. Acesso em: 16 abr. 2009.

KROLL, P. “OpenUP In a Nutshell”. IBM, 2007. Disponível em <http://www.ibm.com/developerworks/rational/library/sep07/kroll/index.html>. Acesso em: 20 out. 2008.

AMBLER, S.W. “The Open Unified Process (OpenUP): Agile, Open Source, and Straightforward”. IBM, 2006. Disponível em <www.cincomsmalltalk.com/presentations/userConf06/Keynote_Ambler_Open-Unified-Process.pdf>. Acesso em: 21 out. 2008.

AMBLER, S. W., “Agile/Lean Documentation: Strategies for Agile Software Development”. Disponível em <http://www.agilemodeling.com/essays/agileDocumentation.htm>. Acesso em: 20 out. 2008.

APÊNDICE A - Tabela comparativa entre RUP, OpenUP e XP

Para mostrar que o OpenUP comporta características do RUP e dos métodos ágeis, a Tabela A.1 apresenta alguns dados de cada processo para efeito de comparação. Esta tabela foi adaptada, com inclusão de conteúdo, de (Runeson, P. e Greberg, P., "Extreme Programming and Rational Unified Process - Contrasts or Synonums?". Disponível em <http://serg.telecom.lth.se/research/publications/docs/282_runeson_XP_RUP.pdf>, acesso em: 21 out. 2008).

Elemento de comparação	***RUP***	***OpenUP***	***XP***
Número de Artefatos	80	17	Pelo menos 2 (*User stories* e teste de aceitação e de unidade.)
Número de Papéis (Roles)	40	7	10
Ferramentas	Necessárias (geralmente comerciais)	Opcionais	Opcionais
Práticas	Desenvolvimento iterativo	Micro-incrementos	*Small releases*
Planejamento	Mudanças constantes nos planejamentos	Mudanças constantes nos planejamentos	Planejar em detalhes apenas o futuro próximo
Definição de release	Uma versão estável e executável de um produto e seus artefatos necessários	Produto de valor para o *stakeholder* totalmente testado, demonstrável ou um build que pode ser disponibilizado	Conjunto de estórias de usuários criando um valor de negócio
Foco	Foco no processo	Foco nos indivíduos	Foco nos indivíduos

Guias do projeto	Casos de uso	Casos de uso	*User stories*
Defensores	Defendido pela gerência (*top-down*)	Defendido pelo pessoal técnico (*bottom-up*)	Defendido pelo pessoal técnico (*bottom-up*)

Tabela A.1. Comparação entre RUP, OpenUP e XP.

APÊNDICE B - Disciplinas do RUP e do OpenUP

Apesar de os processos RUP e OpenUP compartilharem muitas coisas em comum (afinal, ambos derivam do *Unified Process*) as disciplinas de cada um variam. O OpenUP possui um conjunto menor de disciplinas, isto é, possui um subconjunto de disciplinas do RUP. A Figura B.1 mostra as disciplinas do RUP 2007, enquanto a Figura B.2 destaca aquelas presentes do OpenUP.

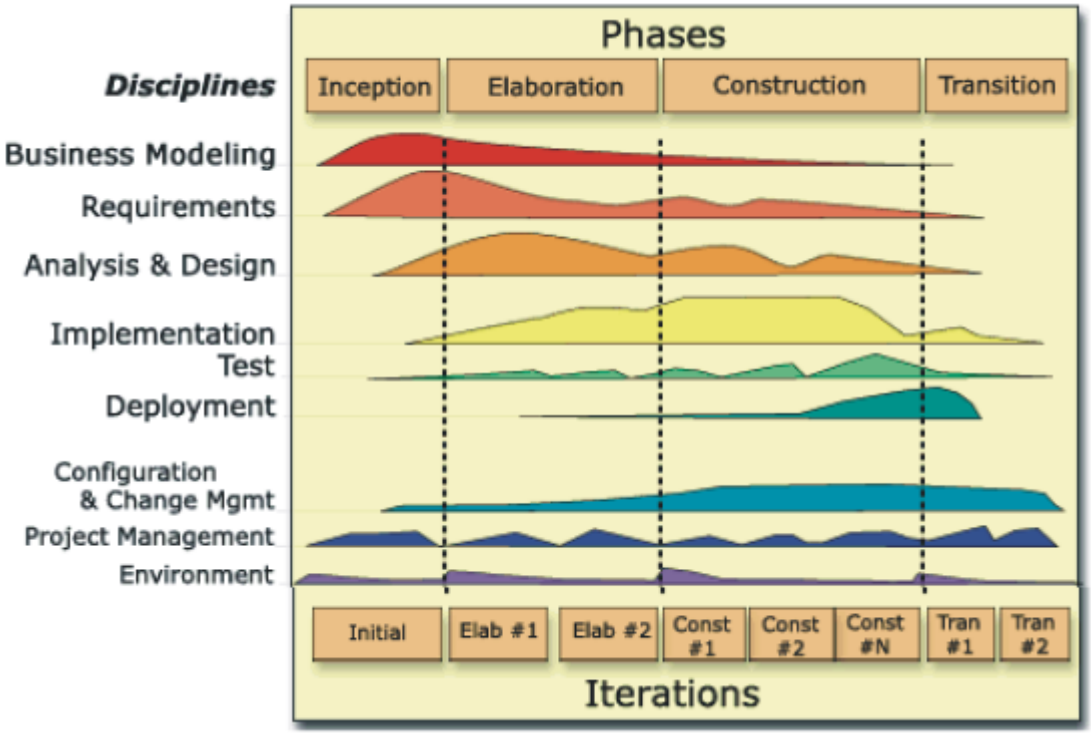

Figura B.1 - Disciplinas do RUP.

O OpenUP não possui as disciplinas *Business Modeling, Configuration & Change Management* e *Environment.* Além disso, a disciplina *Analysis & Design* é chamada de *Architecture* no OpenUP.

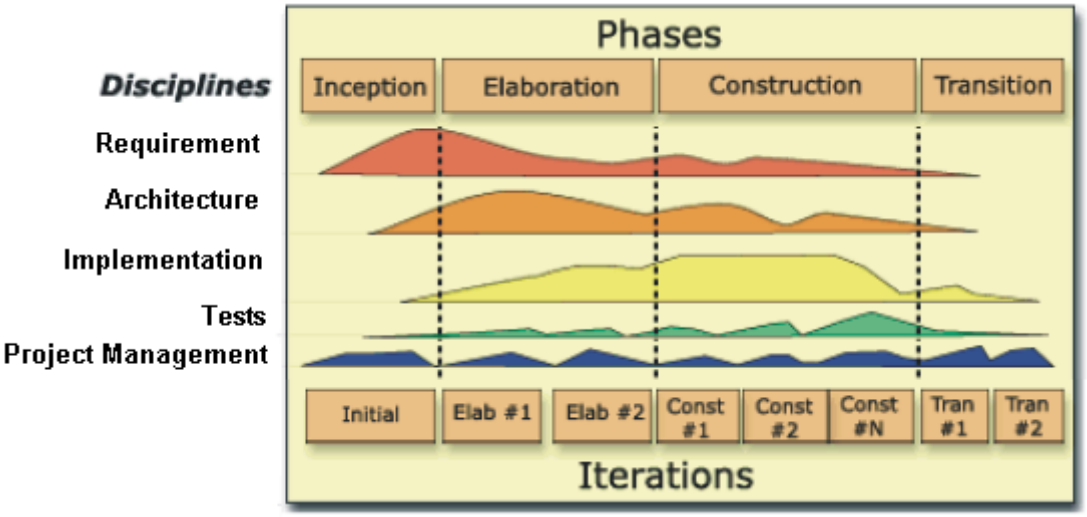

Figura B.2 – Disciplinas do OpenUP.

APÊNDICE C – *Attack Trees*

Árvore de Ataque para segurança (também referida como Árvore de Ameaças ou Análise de Árvore de Faltas) é uma abordagem *top-down* para identificação de vulnerabilidades. Em uma árvore de ataque, a meta do atacante é colocada no topo da árvore (nó raiz). Então, o Analista documenta possíveis alternativas para se atingir a meta do atacante. Para cada alternativa, o Analista pode, recursivamente, adicionar alternativas precursoras para alcançar sub-metas que compõem a meta principal do atacante. Este processo é repetido para cada meta do atacante. Ao examinar os nós de nível mais baixo (nós folha) da árvore de ataque resultante, o Analista pode então identificar todas as possíveis técnicas para violação da segurança do sistema; prevenções para estas técnicas podem então ser especificadas como requisitos de segurança para o sistema (Gortzel et al., 2006).

ANEXO A – Agile Manifesto

Principles behind the Agile Manifesto (Agile Manifesto)

We follow these principles:

Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Business people and developers must work together daily throughout the project.

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Working software is the primary measure of progress.

Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Continuous attention to technical excellence and good design enhances agility.

Simplicity--the art of maximizing the amount of work not done--is essential.

The best architectures, requirements, and designs emerge from self-organizing teams.

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

ANEXO B – Artefatos do OpenUP

Lista dos artefatos do processo OpenUP (OPENUP, 2008).

Discplina / Papel	Artefato	Descrição
Arquitetura	*Arquitetura*	Descreve a análise racional, as suposições, os esclarecimentos, e as implicações das decisões feitas para formar a arquitetura. Tem como objetivo capturar e fazer decisões arquiteturais e explicar aos desenvolvedores tais decisões.
Desenvolvimento	*Modelo de Implementação*	Códigos fonte, arquivos de dados, e arquivos de suporte que representam as matérias-prima para construção do sistema. Representa as partes físicas que compõem o sistema e organizam estas partes de tal maneira que tudo seja compreensível e gerenciável.
	Build	Versão operacional do sistema (fonte compilado) ou parte dele que demonstra um subconjunto de capacidades a serem fornecidas no final. Entrega valor incremental ao usuário ou cliente, e fornece um artefato testável para verificação.
	Modelo de Design	Descreve a realização de funcionalidades requeridas do sistema e serve como uma abstração do código fonte. Esta descrição mostra os elementos do sistema para que possam ser examinados e compreendidos de maneiras que o código fonte não permite.
	Testes do Desenvolvedor	Valida e especifica aspectos da implementação de um elemento. Este artefato é usado para avaliar se um *build* se comporta da maneira como foi especificado.
Gerência de Projeto	*Lista de Riscos*	Este artefato é uma lista de riscos conhecidos e abertos do projeto, esta lista é ordenada por importância e associada com ações de mitigação e contingência. Tem como propósito capturar os riscos percebidos para aumentar as chances de sucesso do projeto.

	Lista de Itens de Trabalho[17]	Contém uma lista de todos os trabalhos agendados a serem feitos no projeto, assim como o trabalho proposto que pode afetar o produto neste ou em futuros projetos. Cada item de trabalho pode conter referências a informações relevantes para efetuar o trabalho descrito dentro do item de trabalho. Este artefato tem como propósito coletar todas as requisições para o trabalho que será potencialmente feito em um projeto, e priorizar, estimar esforços e medir o progresso deste trabalho.
	Plano de Iteração	Um plano de granularidade fina que descreve os objetivos, designação de trabalho, e critérios de avaliação para a iteração. Os objetivos principais deste planejamento são fornecer ao time: um local central para informações sobre objetivos da iteração; um plano detalhado com designação de tarefas; e resultados de avaliação. Este artefato também ajuda o time a monitorar o progresso da iteração, e mantém os resultados desta avaliação que podem ser úteis para melhorar a próxima iteração.
	Plano de Projeto	Colhe todas as informações necessárias para gerenciar o projeto em um nível estratégico. A parte mais importante consiste de um plano de grossa granularidade, identificando iterações e seus objetivos. É um planejamento macro e um planejamento das fases. Tem como propósito fornecer um documento central onde qualquer participante do projeto possa encontrar informações de como o projeto será conduzido.
Requisitos	*Glossário*	Define termos importantes e os reúne para clarificar o vocabulário usado no projeto. Tem como finalidade registrar os termos que são usados para que todos tenham um entendimento comum deles; atingir consistência ao promover o uso de uma terminologia comum no decorrer do projeto; fazer com que diferentes *stakeholders* usem os mesmos termos para dar significado a coisas diferentes, ou diferentes termos para dar significado à mesma coisa; e fornecer termos importantes para os analistas e para os *designers.*
	Visão	Define a visão dos *stakeholders* da solução técnica a ser

[17] Itens de trabalho podem ser: requisitos, diagramas, fontes, defeitos, etc.

		desenvolvida. Esta definição é especificada em termos de necessidades chave e características dos *stakeholders*. Ela contém um esboço dos requisitos centrais previstos para o sistema. Este artefato fornece uma base de alto nível para requisitos técnicos mais detalhados. Fornece um contexto para os membros da equipe tomarem decisões na área de requisitos. Este artefato deve ser visível para todos da equipe.
	Requisitos não Expressos por Casos de Uso (System-wide)	É usado para os seguintes propósitos: descrever os atributos de qualidade do sistema, e os limites que as opções de *design* devem satisfazer para entrega de metas de negócio, objetivos, ou capacidades; capturar requisitos funcionais que não são expressos por casos de uso (*system-wide*); avaliar o tamanho, custo, e viabilidade do sistema proposto; e entender os requisitos em nível de serviço para o gerenciamento operacional da solução.
	Modelo de Caso de Uso	Apresenta uma visão geral do comportamento pretendido pelo sistema. É a base para o contrato entre os *stakeholders* e a equipe de projeto em relação às funcionalidades pretendidas do sistema. O artefato também guia várias tarefas no ciclo de vida do desenvolvimento do *software*.
	Casos de Uso	Captura o comportamento do sistema para revelar um resultado observável do valor para aqueles que interagem com o sistema. Este artefato tem os seguintes propósitos: atingir um entendimento comum do comportamento do sistema; projetar elementos que suportem o comportamento requerido; identificar casos de teste; planejar e avaliar o trabalho; e escrever documentação de usuário.
Testes	*Casos de Teste*	Especificar um conjunto de entradas para os testes, condições de execução, e resultados esperados que são identificados para avaliar um aspecto em particular do cenário.
	Script de Teste	Contém instruções passo a passo que compõem um teste e o torna executável. Estes *scripts* podem estar no formato de instruções textuais legíveis por seres humanos, ou na forma de *scripts* executáveis por computador possibilitando os testes automatizáveis.

	Registros de Execução dos Testes	Coleta as saídas "cruas" (*raw*) capturadas durante a execução de um ou mais testes unitários para um único ciclo de teste.

Tabela B.1 – Artefatos do OpenUP.

ANEXO C – Práticas do OpenUP

Transcrição das práticas gerenciais e das práticas técnicas do OpenUP.

Prática gerencial	***Descrição / Propósito***
Desenvolvimento iterativo	Criar uma solução em incrementos. Cada incremento é completado em um período fixo de tempo, chamado "iteração".
Ciclo de vida Risco-Valor	Esta prática suplementa as práticas do desenvolvimento iterativo e do planejamento em dois níveis com um ciclo de vida de processo unificado. Este ciclo de vida identifica quatro fases, cada uma procura balancear o valor fornecido contra a mitigação de riscos apropriado para cada fase.
Planejamento de projeto em dois níveis	Esta prática personifica o conceito de planejamento de alto nível para o escopo completo do projeto (macro) e o planejamento de baixo nível (micro) para incrementos ou iterações imediatos e futuros.
Toda a equipe	Descreve como um time de desenvolvimento se organiza para capacitar o trabalho efetivo.
Gerência de mudanças na equipe	Esta prática captura requisições de mudanças que são gerenciadas como parte da gerência do item de trabalho.

Tabela C.1 – Práticas gerenciais do OpenUP.

Prática técnica	***Descrição***
Testes concorrentes	Esta prática descreve como juntar os testes ao desenvolvimento ágil.
Integração contínua	Nesta prática, os membros da equipe integram seus trabalhos frequentemente (pelo menos diariamente).
Arquitetura evolucionária	Analisar as preocupações técnicas mais relevantes que afetam a solução, e capturar as decisões arquiteturais para garantir que estas decisões sejam avaliadas e comunicadas.
Design evolucionário	Descreve uma abordagem para fazer o *design* que assume que o mesmo possa evoluir no decorrer do tempo, minimizando a documentação enquanto ainda provê orientações para

	fazer decisões de *design* e comunicando estas decisões.
Visão compartilhada	Suporta a definição e comunicação de uma visão geral para o projeto.
TDD (Test Driven Development)	Esta prática descreve uma abordagem para o desenvolvimento no qual os casos de teste são primeiro definidos, e então o código é desenvolvido para que passe nestes testes.
Desenvolvimento guiado por casos de uso	Descreve como capturar requisitos com uma combinação de casos de uso e requisitos em nível de sistema (*system-wide*, são requisitos não expressos por casos de uso), e então dirige o desenvolvimento e os testes destes casos de uso.

Tabela C.2 – Práticas técnicas do OpenUP.

ANEXO D – *Process Areas* do SSE-CMM

Lista das PAs (Process Areas) e suas metas do SSE-CMM.

Engenharia de Segurança das PAs	***Metas das PAs***
PA01 - Especificar as Necessidades de Segurança	Alcançar um entendimento comum das necessidades de segurança entre todos os interessados aplicáveis, incluindo o cliente.
PA02 - Fornecer Entrada para Segurança	Revisar todas as discussões sobre implicações de segurança e resolver estas discussões de acordo com metas de segurança; assegurar que todos os membros do time de projeto entendam a segurança para que possam realizar suas funções; assegurar que a solução reflita a entrada de segurança fornecida.
PA03 - Verificar e Validar a Segurança	Garantir que as soluções satisfaçam todos os requisitos de segurança e encontrem as necessidades operacionais de segurança do cliente.
PA04 – Atacar a Segurança	Identificar vulnerabilidades do sistema e determinar os *exploits* em potencial destas vulnerabilidades.
PA05 - Avaliar o Risco Operacional da Segurança	Alcançar um entendimento dos riscos de segurança associados com a operação do sistema dentro de um ambiente definido.
PA06 - Construir Argumentos de Garantia	Assegurar que os artefatos de trabalho e os processos forneçam claramente uma evidência de que as necessidades de segurança dos clientes foram atingidas.
PA07 - Monitorar a Postura de Segurança do Sistema	Detectar e rastrear eventos internos e externos relacionados com a segurança; responder a incidentes de acordo com as políticas; identificar e manipular mudanças na postura de segurança operacional em conformidade com os objetivos de segurança.
PA08 - Administrar os Controles de Segurança	Assegurar que os controles de segurança são configurados e usados de maneira apropriada.
PA09 - Coordenar a Segurança	Garantir que todos os membros do time de projeto estejam cientes e envolvidos com as atividades de Engenharia de Segurança, até a extensão necessária para realizar suas funções; coordenar e comunicar todas as decisões

	e recomendações relacionadas à segurança.
PA10 - Determinar as Vulnerabilidades de Segurança	Alcançar um entendimento das vulnerabilidades de segurança do sistema dentro de um ambiente definido.

Tabela D.1 – PAs e Metas de Engenharia de Segurança do SSE-CMM (SSE-CMM).