UNIVERSIDADE FEDERAL DE ALFENAS
INSTITUTO DE CIÊNCIAS EXATAS
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Rômulo Silva Campos

DEFINIÇÃO DAS DISCIPLINAS DE GESTÃO DE CONFIGURAÇÃO E TESTES PARA O PROCESSO DE DESENVOLVIMENTO DE SOFTWARE DO LP&D

Alfenas, 27 de junho de 2011

UNIVERSIDADE FEDERAL DE ALFENAS
INSTITUTO DE CIÊNCIAS EXATAS
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

DEFINIÇÃO DAS DISCIPLINAS DE GESTÃO DE CONFIGURAÇÃO E TESTES PARA O PROCESSO DE DESENVOLVIMENTO DE SOFTWARE DO LP&D

Rômulo Silva Campos

Monografia apresentada ao Curso de Bacharelado em Ciência da Computação da Universidade Federal de Alfenas como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação.

Orientador: Prof. Rodrigo Martins Pagliares

Alfenas, 27 de junho de 2011.

Rômulo Silva Campos

DEFINIÇÃO DAS DISCIPLINAS DE GESTÃO DE CONFIGURAÇÃO E TESTES PARA O PROCESSO DE DESENVOLVIMENTO DE SOFTWARE DO LP&D

A Banca examinadora abaixo-assinada aprova a monografia apresentada como parte dos requisitos para obtenção do título de Bacharel em Ciência da Computação pela Universidade Federal de Alfenas.

Prof. Humberto César Brandão

Universidade Federal de Alfenas

Profa. Mariane Moreira de Souza

Universidade Federal de Alfenas

Prof. Rodrigo Pagliares (Orientador)

Universidade Federal de Alfenas

Alfenas, 27 de junho de 2011.

AGRADECIMENTO

Agradeço ao professor orientador Rodrigo Pagliares, pelo apoio e encorajamento contínuo à pesquisa e toda equipe do Laboratório de Pesquisa e Desenvolvimento, coordenada pelo professor Humberto César Brandão, pelo apoio e facilidades concedidas na aplicação prática deste trabalho.

RESUMO

Qualquer projeto de desenvolvimento de software, seja ele de porte pequeno, médio ou grande, gera uma grande quantidade de informações e artefatos, sendo estes acessados e modificados diversas vezes ao longo de todo o ciclo de vida deste projeto. Por isso é desejável utilizar a Gestão de Configuração de software é para prover um ambiente controlado, onde as modificações estejam sendo gerenciadas.

As falhas de software são grandes responsáveis por custos e tempo no processo de desenvolvimento de software. Embora não seja possível remover todos os erros existentes em certa aplicação, é possível reduzir consideravelmente o número dos mesmos utilizando uma infra-estrutura de testes mais elaborada, que permita identificar e remover defeitos mais cedo e de forma mais eficaz.

Este trabalho investiga técnicas e ferramentas associadas à princípios de métodos ágeis e define as disciplinas de Gestão de Configuração e de Testes para o processo de desenvolvimento de software da Fábrica de Software do Laboratório de Pesquisa e Desenvolvimento da UNIFAL-MG.

Palavras-Chave: Testes, Gestão de Configuração, Integração Contínua.

ABSTRACT

Any software development project, whether small-sized, medium or large, it generates a lot of information and artifacts, which are accessed and modified several times throughout the life cycle of this project. Therefore it is desirable to use the Software Configuration Management is to provide a controlled environment where changes are being managed.

Software failures are responsible for large costs and time in the process of software development. Although you can not remove all errors in a certain application, it is possible to reduce considerably the number of same using an infrastructure of more elaborate tests, to identify and remove defects earlier and more effectively.

This work investigates techniques and tools associated with the principles of agile methods and defines the disciplines of Configuration Management and Testing in the process of software development Software Factory of the Laboratory of Research and Development UNIFAL-MG.

Keywords: Testing, Configuration Management, Continuous Integration.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIAÇÕES

CMMI	*Capability Maturity Model Integration*
CVS	*Concurrent Version System*
IDE	*Integrated Development Environment*
LP&D	Laboratório de *Pesquisa* e Desenvolvimento
SCM	*Software Configuration Management*
SRS	*Software Requirements Specification*
TDD	*Test Driven Development*
UP	*Unified Process*
VSS	*Visual SourceSafe*
XP	*eXtreme Programming*

LISTA DE TRADUÇÕES

Esta lista apresenta a tradução de alguns termos utilizados no decorrer deste trabalho. Os termos em inglês estão na coluna da esquerda e suas correspondes traduções para a língua portuguesa estão na coluna da direita.

Bug	Defeito
Build	Construção
Branch	Ramificação
Commit	Mandar ou Enviar
Plugin	Complemento
Property	Propriedade
Refactoring	Refatoração
Script	Roteiro
Target	Alvo ou Meta
Tasks	Tarefas
Tracker	Rastreador
Update	Atualização

SUMÁRIO

1
Introdução

Este capítulo apresenta uma visão geral sobre o Manifesto Ágil e o Processo Unificado. As seções 1.1, 1.2 e 1.3 descrevem a justificativa, problematização e objetivos deste trabalho.

Em 2001, um respeitado grupo de profissionais experientes na área de desenvolvimento de software se reuniu em uma estação de *sky* chamada *The Lodge at Snowbird* no estado americano de *Utah*. Este grupo era formado de 17 especialistas na área de desenvolvimento de software que possuíam como objetivo propor uma alternativa aos pesados processos de desenvolvimento de software orientados a documentação. Jim Highsmith (2001) definiu o grupo como : " 'A Aliança Ágil', um grupo de pensadores independentes sobre processo de desenvolvimento de software".

O resultado deste encontro foi um documento mundialmente conhecido como Manifesto Ágil(BECK; BEEDLE, et al. apud SINIAALTO, 2006) para desenvolvimento de software ou simplesmente Manifesto Ágil. O documento foi assinado por todos participantes.

O propósito do Manifesto Ágil se resume às melhores maneiras de desenvolver software valorizando indivíduos e interação entre eles sobre processos e ferramentas, software em funcionamento mais que documentação abrangente, colaboração com o cliente mais que negociação de contratos e responder a mudanças mais que seguir um plano.

Tarefas relacionadas a gestão de configuração e teste de software estão de acordo com o Manisfesto Ágil, pois estão diretamente ligadas com a resposta a mudanças e software funcionando. O objetivo deste trabalho é definir as disciplinas de gestão de configuração e teste baseadas no Processo Unificado.

"O OpenUP é um Processo Unificado que aplica uma abordagem iterativa e incremental dentro de um ciclo de vida estruturado. O OpenUP abraça uma filosofia pragmática e ágil que foca na natureza colaborativa do desenvolvimento de software. É um processo independente de ferramenta e de pouca cerimônia que pode ser estendido para direcionar uma grande variedade de tipos de projeto"(OPENUP, 2004).

O OpenUP é muito útil para pequenas equipes de desenvolvimento de software, *stakeholders* e engenheiros de processos. Pode ser usado em sua forma original, mas se a equipe tiver características significativamente diferentes, o processo deve ser modificado ou estendido para atender a estas necessidades.

O OpenUP está organizado da seguinte maneira segundo seu próprio glossário:

- Processo: Os processos relacionam os elementos de conteúdo em seqüências semi-ordenadas que são personalizadas para determinados tipos de projeto. Sendo assim, um processo é um conjunto de descrições de trabalho parcialmente ordenadas, destinadas a alcançar uma meta maior de desenvolvimento, tal como o lançamento de um software específico.Estas descrições de trabalho estão organizadas em uma estrutura de decomposição hierárquica que um processo foca no ciclo de vida e cria a seqüência do trabalho em uma estrutura de decomposição.

- Elementos de conteúdo: Os elementos de conteúdo fornecem explicações passo-a-passo, descrevendo como as metas de desenvolvimento bem específicas são atingidas independente da colocação destes passos em um ciclo de vida de desenvolvimento. Eles são instanciados e adaptados às situações específicas nas estruturas de processo.

- Fases: OpenUp é dividido em 4 fases: Concepção, Elaboração, Construção e Transição. As Fases são o tempo entre dois grandes marcos de projeto, durante o qual um conjunto bem definido de objetivos é cumprido e a decisão de entrar ou na próxima fase é tomada. Cada fase pode ser realizada utilizando-se um determinado número de iterações.

- Iterações: pequenas divisões do projeto com duração limitada, permitem demonstrar valor incremental e obter feedback rápido e contínuo. As iterações contém um conjunto de atividades.

- Atividades: uma atividade é um elemento de decomposição, que suporta o agrupamento lógico e aninhado de elementos relacionados aos processos, tais como descritor e sub-atividades, formando assim estruturas de decomposição. Cada atividade possui um conjunto de tarefas.

- Tarefas: uma unidade de trabalho que um papel pode ser solicitado a executar. Toda tarefa pertence a uma disciplina, produz/consome produtos de trabalho e é composta por passos.

- Disciplinas: Uma coleção de tarefas relacionadas que definem um grande "área de preocupação". Na engenharia de software, as Disciplinas incluem: Requisitos, Arquitetura, Desenvolvimento, Teste e Gestão de Projeto.

- Papel: uma definição do comportamento e das responsabilidades de um indivíduo, ou um grupo de indivíduos que trabalham juntos como uma equipe.

- Produtos de Trabalho: é um elemento de conteúdo que representa qualquer coisa usada, produzida ou modificada por uma tarefa.

- Passos: um passo é um elemento de conteúdo utilizado para organizar as tarefas em partes ou subunidades de trabalho.

Este trabalho adapta as disciplinas de Gestão de Configuração e Testes descritas no OpenUP para processo de desenvolvimento de software para o LP&D (Laboratório de Pesquisa e Desenvolvimento) na Universidade Federal de Alfenas.

O trabalho está organizado em mais 4 capítulos. No capítulo 2, serão abordadas definições sobre a gestão de configuração e suas aplicações. O capítulo 3 apresenta conceitos de teste de software e seus benefícios. O capítulo 4 define as disciplinas de Gestão de Configuração e de Testes para serem utilizadas no LP&D. O último capítulo é voltado para conclusões e trabalhos futuros.

1.1 Justificativa e Motivação

Atualmente o LP&D não possui um método e um conjunto definido de ferramentas para desenvolvimento de seus projetos.

Acredita-se que a escolha de tais ferramentas em conjunto com a aplicação de princípios ágeis, no contexto de um processo de desenvolvimento de software, mais especificamente dentro de suas disciplinas Gestão de Configuração e Testes podem beneficiar a produtividade do time de desenvolvimento e a qualidade dos produtos desenvolvidos.

1.2 Problematização

O processo de desenvolvimento de software é caracterizado por uma série de problemas que interferem no desenvolvimento em diferentes perspectivas, pode-se relatar como problemas mais relevantes:

> (1) as estimativas de prazo e de custo freqüentemente são imprecisas; (2) a produtividade das pessoas da área de software não tem acompanhado a demanda por seus serviços; e (3) a qualidade de software às vezes é menor que a adequada(PRESSMAN, 2002, p.23).

Como o software hoje em dia afeta o cotidiano da vida das pessoas, a importância da qualidade está aumentando continuamente. As empresas de software, incluindo as pequenas, têm dado atenção às questões de qualidade, afim de garantir a competitividade de seus produtos (HIGHSMITH, 2001, p.137).

Atualmente, o LP&D não utiliza um conjunto definido de ferramentas automatizadas para suporte a princípios ágeis em seu processo de desenvolvimento de software e as disciplinas Gestão de Configuração e Testes não foram definidas. É possível definir tais disciplinas afim de aumentar a produtividade e qualidade da fábrica de software do LP&D?

1.3 Objetivos

Esta seção faz uma abordagem sucinta dos objetivos gerais e específicos deste trabalho.

1.3.1 Gerais

Este trabalho destina-se a um estudo detalhado de princípios e técnicas ágeis, além da investigação de ferramentas que auxiliem o desenvolvimento produtivo de software por um time de desenvolvedores.

1.3.2 Específicos

- Analisar as técnicas e ferramentas usadas atualmente no LP&D para desenvolvimento de software;
- Aplicar os princípios e técnicas ágeis juntamente com ferramentas de apoio no LP&D;
- Treinar os membros do time de desenvolvimento do LP&D em princípios e técnicas ágeis e nas ferramentas investigadas neste trabalho;
- Definir os princípios e ferramentas usadas nas disciplinas de Gestão de Configuração e Testes.

2
Gestão de Configuração de Software

Este capítulo apresenta uma abordagem geral sobre Gestão de Configuração de Software, apresenta uma ferramenta de controle de versões e mudanças, uma ferramenta de builds automatizados e para finalizar descreve a integração contínua e seus benefícios.

Software Configuration Management (*SCM*) ou em português, Gestão de Configuração de Software (GCS), é uma área da Engenharia de Software que visa fornecer o apoio necessário no processo de desenvolvimento.

Roger S. Pressman (2006) define Gestão a de Configuração de Software como :

> Conjunto de atividades projetadas para controlar as mudanças pela identificação dos produtos do trabalho que serão alterados, estabelecendo um relacionamento entre eles, definindo o mecanismo para o gerenciamento de diferentes versões destes produtos, controlando as mudanças impostas, e auditando e relatando as mudanças realizadas .

Pode-se dizer que as tarefas da disciplina de Gerência de Configuração garantem a qualidade dos sistemas produzidos. Seu objetivo é monitorar o produto e os processos, para tentar assegurar que eles estejam em conformidade com os requisitos especificados e os planos estabelecidos. (FIGUEIREDO; SANTOS; ROCHA, 2004, p.3) Sem estas atividades em um ambiente de desenvolvimento de software, diversos tipos de falhas podem vir a ocorrer, entre elas: (1) espaços de trabalho podem ser compartilhados, onde não se tem controle sobre as alterações; (2) sobreposição de informação nos artefatos e alterações perdidas; (3) perda artefatos.

Existe uma dificuldade nas empresas em aderirem corretamente a processos de GCS, principalmente em empresas de pequeno e médio porte, pois não conseguem investir na qualificação de seus processos como deveriam. Outras causas para esta situação, segundo Dias (2007), são o desconhecimento da amplitude e importância da GCS e o desconhecimento das ferramentas de apoio existentes. Além de que, para que este processo seja implementado com sucesso, é necessário ocorrer um esforço de adaptação de toda equipe de desenvolvimento.

A GCS pode ser tanto aplicada por um procedimento manual quanto por ferramentas automatizadas para o controle de revisões. Existem várias ferramentas de controle de versões, comercias ou gratuitas de código aberto, dentre elas pode-se destacar: *Visual SourceSafe* (VISUAL SOURCESAFE, 2005), *Subversion* (COLLINS-SUSSMAN et al., 2002), *Concurrent Version System* (CVS, 2006), *Mercurial* (MERCURIAL, 2005) e Git (GIT, 2008). Todas fazem basicamente a mesma coisa entretanto de maneiras diferentes, o apêndice A descreve algumas funcionalidades do *Subversion*.

Este trabalho surgiu da necessidade de alinhamento entre os conceitos de GCS e as premissas que justifiquem a sua utilização no LP&D. Para a implantação da GCS não existe nenhum tipo de roteiro formalizado, que possa ser aplicado em todos os tipos de organização, uma vez que cada uma delas apresenta cenários diversos umas das outras, tais como necessidades diferentes, além de variados tipos de recursos humanos e computacionais.

2.1 Integração Contínua

Este conceito apresenta a prática de integrar continuamente as mudanças concluídas para reduzir o esforço necessário de mesclar o desenvolvimento paralelo, encontrar erros rapidamente e conduzir um ambiente de trabalho colaborativo(OPENUP, 2009).

O termo integração contínua se originou com o desenvolvimento do método ágil *Extreme Programming(*BECK, 2004*)* como uma de suas doze práticas originais.

Integração contínua é uma prática de desenvolvimento de software em que membros de uma equipe integram seu trabalho freqüentemente. Normalmente, cada membro faz várias integrações diárias. Cada integração é verificada por uma *build* automatizado, incluindo testes para detectar erros de integração o mais rápido possível. Muitas equipes acham que essa abordagem leva a problemas de integração reduzida e permite uma equipe desenvolver software coeso mais rapidamente (FOWLER, 2000).

Anteriormente foram reveladas a importância da gerência de configuração e o uso de uma ferramenta de controle de versão. Agora o foco será mudado para os *builds* e testes automatizados além de uma ferramenta de integração contínua conforme ilustrado na Figura 1.

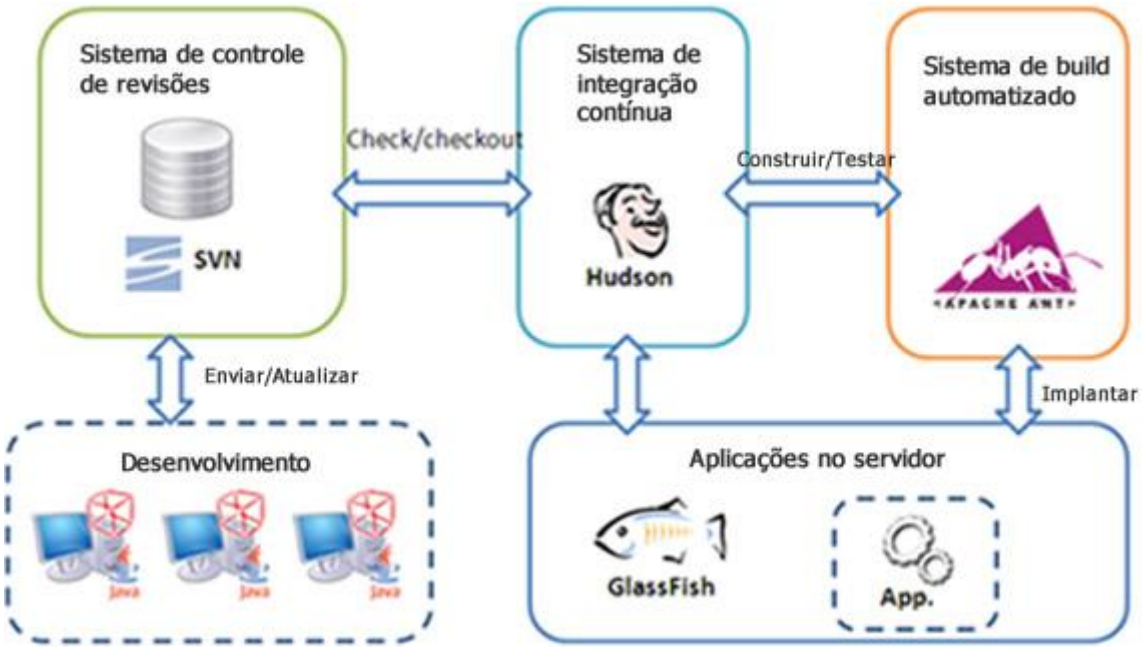

Figura 1- Integração contínua

Fonte: http://samerabdelkafi.files.wordpress.com. Acessado em : 15 jan. 2011(Adaptada).

Apesar da Integração Contínua ser uma prática que não requer nenhuma ferramenta especial para implantar, sugere-se o uso de um servidor de integração contínua. O mais conhecido deles é o *CruiseControl (*CRUISE CONTROL, 2002*)*, uma ferramenta *open source* originalmente construída por várias pessoas na *ThoughtWorks* e hoje mantida por uma vasta comunidade (FOWLER, 2000).

Atualmente o maior concorrente do *CruiseControl* é o Hudson*(*MOSER; PRAKASH, 2011), a ferramenta vem ganhando adeptos pela simplicidade de uso e o número de complemento (*plugins)* disponíveis, neste trabalho iremos adotar o Hudson como ferramenta de integração contínua, por ser uma ferramenta que possui integração com as demais ferramentas utilizadas neste trabalho.

Uma boa maneira de se capturar erros de forma rápida e eficiente é incluir testes automatizados no processo de integração. O teste não é perfeito, mas pode capturar uma ampla gama de erros, o suficiente para ser útil. Em particular, com a ascensão do *Extreme Programming* (XP) e do *Test Driven Development* (TDD), muito se têm feito para popularizar o código de testes automatizados e, como resultado, muitas pessoas viram o valor da técnica (FOWLER, 2000).

2.2 Benefícios da Gestão de Configuração de Software

A adoção de GCS por uma empresa envolve custos e benefícios que devem ser considerados. Os principais benefícios decorrentes da aplicação estão entre a facilidades para acomodar mudanças, o maior controle sobre os produtos, a economia de tempo de desenvolvimento, a facilidades na geração de versões diferentes de um mesmo produto de software (customização), a manutenção de históricos de produtos e a facilidades de se voltar a situações anteriores. Os principais custos, por outro lado, são o treinamento e os investimentos para a implementação, que englobam recursos humanos e computacionais (WHITE, 2001).

A gestão de configuração de software permite minimizar os problemas decorrentes do processo de desenvolvimento, através de um controle sistemático sobre as modificações. Não é objetivo da SCM evitar modificações, mas permitir que elas ocorram sempre que possível, sem que haja falhas inerentes ao processo.

Os benefícios obtidos pela Gestão de Configuração são comprovados pelo fato da mesma estar presente em um reconhecido modelo de maturidade: o CMMI(*Capability Maturity Model Integration*) (SEI, 2007).

> O propósito da área de processo de Gerenciamento de Configuração (*Configuration Management* – CM) é estabelecer e manter a integridade dos produtos, através das atividades de identificação, controle de versão, controle de mudança, relato de status e auditoria de configuração (SEI, 2007, p.114).

Apesar dos debates sobre vantagens e desvantagens do CMMI, ele tem sido usado há mais de 10 anos, tempo suficiente para que muitas companhias possam verificar o aumento da qualidade de seus produtos e a diminuição de seus custos de produção. Numa era de crescente aumento de competitividade, qualquer melhora na produtividade do software não pode ser ignorada.

O próximo capítulo faz uma visão geral sobre teste de software. Tal conteúdo é de fundamental importância para que a disciplina de teste possa ser definida para o LP&D.

3
Teste de Software

Este capítulo aborda os níveis de testes na seção 3.1, os tipos de teste na seção 3.2, já a seção 3.3 incentiva a automação de testes e logo em diante na seção 3.4 é apresentado um processo de desenvolvimento de software baseado em testes.

Este capítulo aborda, de forma sucinta, os níveis e tipos de teste, logo depois apresenta um incentivo aos testes automatizados e descreve o TDD(*Test Driven Development*) e a refatoração.

Uma visão comum do processo de teste é de que ele consiste apenas da fase de execução. Esta, na verdade, é uma parte do mesmo, mas não contempla todas suas atividades .

Existem atividades de teste antes e depois da fase de execução. Por exemplo: planejamento e controle, escolha das condições de teste, modelagem dos casos de teste, checagem dos resultados, avaliação do critério de conclusão, geração de relatórios sobre o processo de teste e sobre sistema alvo e encerramento ou conclusão (exemplo: após a finalização de uma fase de teste). Teste também inclui revisão de documentos (incluindo o código fonte) e análise estática(IASTQB, 2007).

Testes podem possuir objetivos diferentes:

- Encontrar defeitos.
- Ganhar confiança sobre o nível de qualidade e prover informações.
- Prevenir defeitos.

3.1 Níveis de teste

A *International Software Testing Qualifications Board* (ISTQB, 2007) define 4 níveis de teste de software: Teste de componente, teste de integração, teste de sistema e teste de aceitação. Algumas bibliografias definem outros níveis de teste, entretanto este trabalho adota a estrutura da ISTQB por ser internacionalmente aceita.

3.1.1 Teste de Componente

Teste de componentes procura defeitos e verifica o funcionamento do software (ex: módulos, programas, objetos, classes, etc.) que são testáveis separadamente. Pode ser feito isolado do resto do sistema, dependendo do contexto do ciclo de desenvolvimento e do sistema. Controladores (*drivers*) e simuladores (*stubs*) podem ser usados.

Tipicamente, teste de componente ocorre com acesso ao código que está sendo testado e no ambiente de desenvolvimento, assim como o teste de unidade. Na prática, envolve o programador do código.

3.1.2 Teste de Integração

Teste de integração é caracterizado por testar as interfaces entre os componentes, interações de diferentes partes de um sistema, como o sistema operacional, arquivos, *hardware* ou interfaces entre os sistemas.

A cada estágio da integração, os testadores concentram somente na integração propriamente. Por exemplo, o módulo A está sendo integrado com o módulo B o foco é a comunicação entre os módulos, não em suas funcionalidades que são testadas pelo teste de componente.

Idealmente, os testadores devem compreender a arquitetura e influenciar no planejamento da integração. Se o teste de integração for planejado antes que os componentes ou sistemas estejam prontos, eles podem ser preparados visando um teste mais eficiente.

3.1.3 Teste de Sistema

Teste de sistema se refere ao comportamento de todo do sistema/produto definido pelo escopo de um projeto ou programa de desenvolvimento.

Teste de sistema deve tratar requisitos funcionais e não-funcionais do sistema. Os requisitos podem estar como um texto ou diagramas. Os Testadores devem também lidar com requisitos incompletos ou não-documentados. O Teste de sistema em requisitos funcionais deve utilizar a técnica mais apropriada como: caixa-preta ou caixa-branca de acordo com a característica do sistema a ser testado.

Geralmente neste cenário existe uma equipe de testes responsável pelo teste de sistema.

3.1.4 Teste de Aceitação

O objetivo do teste de aceitação é estabelecer a confiança no sistema, parte funcional e não-funcional.. Procurar defeitos não é o principal foco do teste de aceitação. O Teste de aceitação pode avaliar a disponibilidade do sistema para entrar em produção.

Este nível de teste é de responsabilidade do cliente ou do usuário do sistema auxiliados pela equipe de testes. Os interessados (*stakeholders*) também podem ser envolvidos.

3.2 Tipos de Teste

Os testes são direcionados para verificar o sistema (ou uma parte do sistema) com base em um motivo ou finalidade específica. Cada tipo de teste tem foco um objetivo particular, que pode ser o teste de funcionalidades, uma característica da qualidade não-funcional tal como confiabilidade ou usabilidade, uma confirmação de correção de *bug* e/ou uma inspeção de mudanças inesperadas após uma modificação no sistema (ISTQB, 2007). As seções seguinte descrevem os principais tipos de testes de acordo com a ISTQB(2007).

3.2.1 Teste Funcional

Como o próprio nome sugere este tipo de teste valida se as funcionalidades do sistema, subsistema ou módulo do sistema estão de acordo com a especificação de requisitos e/ou documento de caso de uso. As funções especificam "o que" o sistema faz.

O teste funcional considera o comportamento externo do software, geralmente utiliza-se a técnica de caixa-preta em que o analista de teste não tem acesso ao código fonte do sistema.

3.2.2 Teste Não-Funcional

O teste não-funcional verifica as características do produto de software, ele inclui , mas não se limita a : teste de performance, teste de carga, teste de estresse, teste de usabilidade e teste de confiabilidade. É o teste de "como" o sistema trabalha ou se comporta.

O termo teste não-funcional descreve que o teste é executado para medir as características que podem ser quantificadas em uma escala variável, como o tempo de resposta em um teste de performance.

3.2.3 Teste de Confirmação e Regressão

"Quando um defeito é detectado e resolvido, o software pode ser re-testado para confirmar que o defeito original foi realmente removido. Isto é chamado de teste de confirmação" (ISTQB,2007). Já o teste de regressão é o teste repetido de um programa que já foi testado, após sua modificação, para descobrir a existência de algum defeito introduzido ou não coberto originalmente como resultado da mudança. Os testes devem ser repetíveis para suportarem os teste de confirmação e teste de regressão.

3.3 Automação de Testes

Em algum momento, pode-se achar necessário gerenciar o seu esforço de teste através da criação de suítes de teste para os seus recursos de teste. A manutenção de suítes de teste pode assumir várias formas diferentes. Para facilitar o processo de teste, pode-se definir algum nível de automação na suíte de teste. Entretanto, o fato de automatizar as suítes de teste não os tornam necessariamente mais fáceis, a automação pode, de fato, aumentar o peso da manutenção das suítes. O apêndice A descreve algumas ferramentas que podem ser utilizadas na prática de automação de testes.

3.4 Desenvolvimento Dirigido por Testes

Test Driven Development (TDD) ou em português, Desenvolvimento Dirigido por Testes (DDT) é uma abordagem de desenvolvimento de software, em que basicamente escrevemos um teste e na seqüência escrevemos código para fazer o teste passar. Após essas duas etapas, aplicamos a técnica de refatoração para a geração do desenho de nossa solução. Esta abordagem para construção de software encoraja um bom desenho, produz o código testável, e nos mantém longe de excesso de documentação.

Tudo isso é realizado pelo simples ato de evoluir nosso desenho através de testes executáveis que nos direcionam em direção da implementação final (KOSKELA, 2007, p. 4).

O Desenvolvimento Dirigido por Teste é a parte central da abordagem de desenvolvimento ágil derivado do método *Extreme Programming* (XP) (BECK 2004 apud SINIAALTO, 2006 , p.5) e nos princípios do Manifesto Ágil (BECK; BEEDLE, et al. apud SINIAALTO, 2006, p.5).

Segundo a literatura, DDT não é uma técnica tão nova assim. Existem relatos de uso da técnica no Projeto *Mercury* da NASA na década de 1960 (LARMAN; BASILI 2003 apud SINIAALTO, 2006, p.5).

Escrever primeiro o teste e só depois escrever o código para passar no mesmo. Esta regra é controversa para muitos de nós que fomos educados a produzirmos primeiramente o desenho da aplicação, implementar este desenho e finalmente, testar o nosso software (KOSKELA, 2007, p. 15).

A Figura 2 ilustra um comparativo em o desenvolvimento tradicional e o DDT.

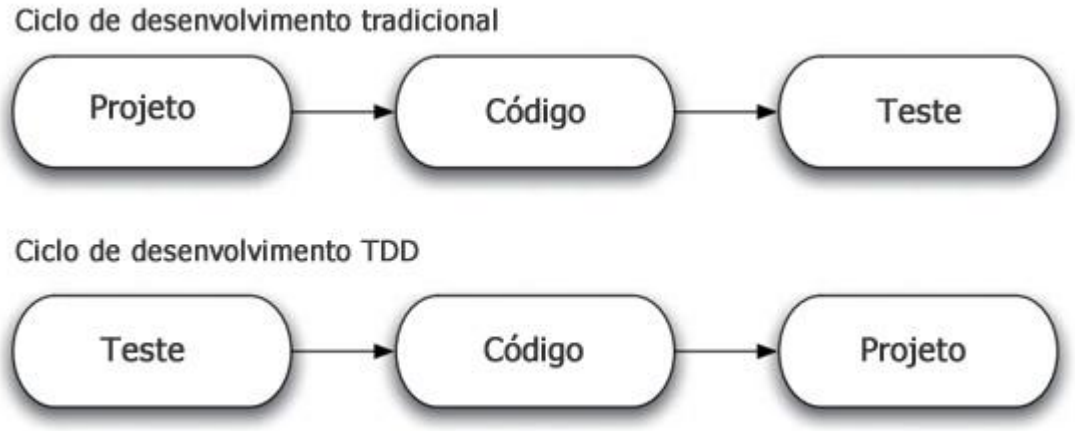

Figura 2 - Comparativo: desenvolvimento tradicional e baseado em testes.

Fonte: KOSKELA (2007,p. 15) (Adaptada)

Apesar do nome, DDT não é uma técnica de teste, mas sim uma técnica de desenvolvimento e desenho em que os testes são escritos antes do código (BECK, 2001).

3.4.1 Desenvolvimento

No DDT, o ciclo incremental é repetido até que toda a funcionalidade seja implementada (SINIAALTO, 2006, p.5 apud ASTELS 2003). Segundo Kent Beck (2002) o ciclo do DDT é composto por 6 etapas fundamentais:

1. Adicionar(criar) os testes;
2. Executar todos os testes e veja se algum deles falha;
3. Escrever código;
4. Executar os testes automatizados e verificar se obteve-se sucesso;
5. Refatorar o código;
6. Repetir tudo.

Este ciclo está ilustrado na Figura 3. A primeira etapa envolve simplesmente escrever um pedaço de código que testa a funcionalidade desejada. A segunda etapa é necessária para validar que o teste está correto, neste caso teste correto é teste que falha, o teste deve falhar pois na há código funcional implementado

ainda. O terceiro passo é a escrita do código. (SINIAALTO,2006,p.5 apud ASTELS,2003).

A cada escrita de código todos os testes devem ser executados, a fim de verificar que as mudanças não introduziram quaisquer problemas em outro lugar no sistema. Depois que todos os testes forem executados , a estrutura interna de o código deve ser melhorada através de refatoração (SINIAALTO, 2006, p.5).

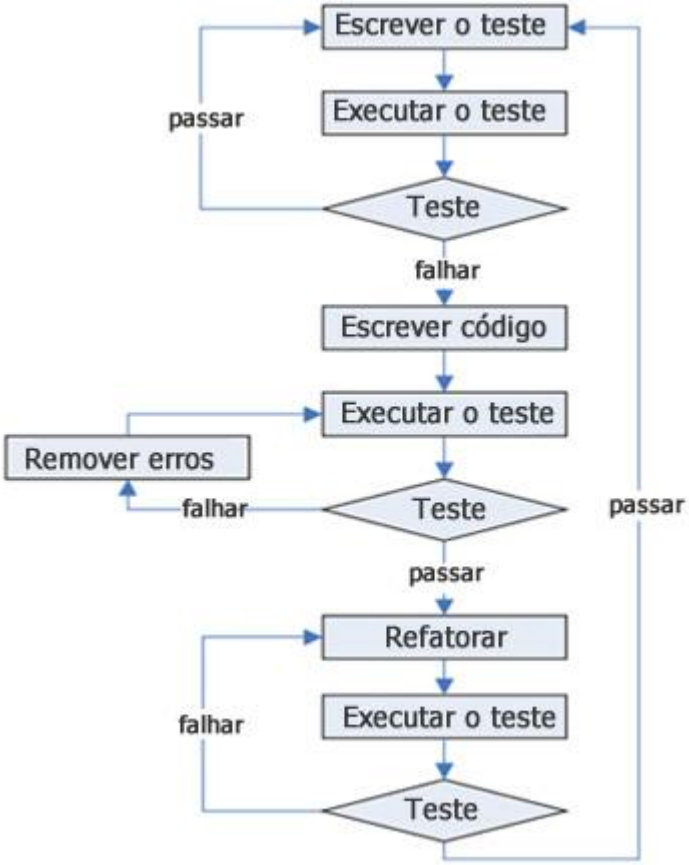

Figura 3- Ciclo TDD

Fonte: SINIAALTO (2006, p.5)(Adaptada).

3.5 Refatoração

Segundo Martin Fowler (2004, p.10), *refactoring* ou refatoração é o processo de alteração de um sistema de software de modo que o comportamento externo do

código não mude, mas que sua estrutura interna seja melhorada. É uma maneira disciplinada de aperfeiçoar o código que minimiza a chance de introdução de falhas. Em essência, quando usa-se a refatoração, melhora-se o desenho do código após este ter sido escrito.

Desenvolver *software* não é simplesmente sentar na frente no computador e iniciar o processo de codificação, o desenvolvimento de software é um processo empírico que requer planejamento. Durante o ciclo de desenvolvimento, as necessidades mudam, novos requisitos surgem outros são melhor compreendidos, fazendo com que o planejamento também mude. Não existe um planejamento fixo em um ambiente de desenvolvimento de *software*, não é como em uma linha de montagem – onde as tarefas são bem definidas e não mudam com frequência. Simples modificações em um sistema de software podem ferir a integridade da aplicação em relação ao modelo proposto inicialmente.

A refatoração possibilita descobrir que o ponto de equilíbrio do trabalho muda. Descobre que o desenho, em vez de acontecer todo no início, ocorre continuamente durante o desenvolvimento. Aprende, com a construção do sistema, a como melhorar o desenho. A resultado de cada passo de refatoraçao leva a um programa que permanece bom à medida que o desenvolvimento continua (FOWLER, p.10, 2004).

Fowler (2004) defende que a refatoração é um elemento-chave no processo de desenvolvimento de software e afirma que um bom ou desenho vem antes mesmo de sua codificação.

3.6 Benefícios do Teste de Software

Williams e Maximilien (2003) apresentam um estudo empírico realizado pela IBM usando desenvolvedores profissionais, que concluiu que a prática do TDD ajuda desenvolvedores a produzir código de alta qualidade. A equipe da IBM produziu uma meticulosa *suite* de casos de testes automatizados após a modelagem UML. O código desenvolvido usando prática TDD mostrou, durante a verificação funcional e testes de regressão, aproximadamente 40% menos defeitos que uma base do produto desenvolvido com uma abordagem mais tradicional. A produtividade da equipe não sofreu impacto pela adição do foco na produção de casos de teste automatizados.

Um experimento estruturado envolvendo 24 desenvolvedores profissionais foi realizado por George e Williams (2003) para investigar a eficácia do TDD. Um

grupo desenvolveu um pequeno programa Java usando TDD enquanto que outro usou uma abordagem baseada em cascata. Os resultados experimentais indicaram que embora os desenvolvedores que utilizam TDD gastaram 16% mais tempo, eles produziram um código de melhor qualidade porque o percentual de casos de testes caixa-preta funcionais que passaram foi 18%, comparando-se com os desenvolvedores que usaram uma abordagem tradicional.

Com base nestes estudos e os dados apresentados na seção 2.2 o próximo capítulo propõe as disciplinas de Gestão de Configuração e Testes para o LP&D.

4
Definição das Disciplinas Gestão de Configuração e Testes

> Uma disciplina é uma coleção de tarefas que se relacionam a uma 'área de interesse' maior em todo o projeto.[...] Embora seja comum executar simultaneamente tarefas que pertençam a várias disciplinas (por exemplo, determinadas tarefas de requisitos são executadas sob a mesma coordenação de tarefas de análise e design), separar estas tarefas em disciplinas distintas é uma forma eficaz de organizar o conteúdo, tornando mais fácil a compreensão(OPENUP, 2004).

Este capítulo descreve as disciplinas "Gestão de Configuração" e "Testes" para o processo de desenvolvimento de software do LP&D. A disciplina de Gestão de Configuração pode aumentar a produtividade do time de desenvolvimento, o foco desta disciplina é gerenciar o ambiente colaborativo de código e o controle de versão de software.

Já a disciplina de Testes propõe tarefas que podem aumentar a qualidade dos softwares desenvolvidos, o desafio é saber o quanto e como testar. Outra questão importante é definir o nível de qualidade necessário para o produto final, lembrando que não é viável testar o sistema em sua totalidade, exceto para sistemas triviais, e quanto maior nível de qualidade maior é o investimento e tempo para finalização do produto.

4.1 Disciplina: Gestão de Configuração

Esta disciplina é composta pelas tarefas "Solicitar Mudança" e "Integrar e Criar a Solução", descritas nas seções 4.1.1.1 e 4.1.1.2 respectivamente.

4.1.1 Atividade: Executar Tarefas Contínuas

Esta atividade ocorre a qualquer momento durante o ciclo de vida em resposta a um defeito observado, uma melhoria desejada ou uma solicitação de mudança. Não é planejada, o que significa que ela não aparece como uma atividade prevista no plano de projeto, no plano da iteração ou na lista de itens de trabalho. Não obstante, é uma atividade crítica que deve ser executada para assegurar o sucesso do projeto, visto que o ambiente não é estático.

As tarefas contidas nesta atividade são freqüentemente executadas pelos desenvolvedores, em resposta a mudanças no projeto ou correções propostas pelos testadores.

4.1.1.1 Tarefa: Solicitar Mudança

Esta tarefa recebe e registra solicitações de mudança e é executada por qualquer papel. A Lista de Itens de Trabalho e a ferramenta *bug tracker* são as entradas. Como saída tem-se a Lista de Itens de Trabalho ou a *bug tracker* atualizada.

Uma solicitação de mudança representa qualquer pedido feito para alterar um produto de trabalho. Isto inclui itens normalmente chamados de informações de *bugs*, solicitações de melhorias, solicitações de mudança de requisitos, solicitações de implementação e solicitações dos *stakeholders*.

Tratando-se de defeitos encontrados no sistema pela equipe de testes as solicitações devem ser relatadas através da ferramenta de *bug tracker*.

Ao submeter uma solicitação de mudança deve-se fornecer o máximo de informação possíveis para permitir uma revisão rápida. No mínimo, todas as solicitações de mudança devem incluir as seguintes informações:

- ID - um identificador único da solicitação de mudança para simplificar o seu rastreamento. Geralmente as ferramenta de *bug tracker* fornecem IDs únicos.

- Descrição Resumida - Uma frase que resuma a solicitação de mudança.
- Descrição Detalhada - Uma descrição detalhada da solicitação de mudança. Para um defeito, se você puder fornecer informações que ajudem a reproduzi-lo, faça-o. Para uma solicitação de melhoria, forneça o raciocínio lógico para a solicitação.
- Item Afetado - O artefato afetado e sua versão.
- Gravidade - O quanto é grave.
-
- Prioridade - Na sua opinião, o quanto é importante esta solicitação.

Estas informações são importantes para validar a solicitação e realizá-la se aprovada.

Os passos desta tarefa são:

- Recolher informações de solicitações de mudança;
- Atualizar a Lista de Itens de Trabalho ou relatar um *bug* na ferramenta de *bug tracker*.

4.1.1.2 Tarefa: Integrar e Criar a Solução

O objetivo desta tarefa é integrar todas as mudanças feitas no código base pelos desenvolvedores e realizar os testes mínimos no incremento de sistema para validar a construção. A meta é identificar problemas na integração, o mais rápido possível de forma que possam ser facilmente corrigidos pela pessoa certa e no momento certo.

O código é mantido em um espaço de trabalho colaborativo chamado repositório, cada desenvolvedor possui uma cópia local do código, mudanças no código local devem ser testadas antes de serem submetidas. Além disso, deve-se atualizar o espaço de trabalho, pois outros desenvolvedores podem ter submetido alterações conflitantes, feito isso os testes devem ser re-executados pelo desenvolvedor e só então o código pode ser submetido ao repositório colaborativo.

Esta tarefa é comumente executada pelo desenvolvedor. Como entrada é necessário o próprio código (implementação), *script* de testes unitários e de sistema.

Os passos desta tarefa são:

1. Integrar elementos implementados;

2. Criar a Construção;
3. Testar elementos integrados;
4. Tornar as mudanças disponíveis;
5. Fechar uma versão base do sistema.

A nomenclatura da construção criada deve seguir as regras descritas no Apêndice C.

4.2 Disciplina: Testes

Está disciplina é composta pelas tarefas: "Criar Casos de Teste", "Implementar os Scripts de Teste" e " Executar os Testes".

4.2.1 Atividade: Identificar e Refinar os Requisitos

Esta atividade descreve as tarefas executadas para especificar, analisar e validar um subconjunto de requisitos do sistema antes de sua implementação. Executa-se esta atividade durante todo o ciclo de vida com os *stakeholders* e toda a equipe de desenvolvimento colaborando para assegurar que um conjunto claro, consistente, correto, verificável e praticável de requisitos esteja disponível para direcionar a implementação.

Inicialmente, o foco está em obter acordo sobre o problema a ser resolvido, recolhendo as necessidades dos *stakeholders* e capturando as características de alto nível do sistema. Logo após, o foco desloca-se para a definição da solução. Isto consiste em encontrar os requisitos que têm o maior valor para os *stakeholders*, que são particularmente desafiadores ou de grande risco, ou que são arquiteturalmente significantes. Deve-se descrever os requisitos com detalhes suficientes para validar a compreensão da equipe de desenvolvimento sobre os requisitos, para assegurar a concordância dos *stakeholders*, e para permitir o início do desenvolvimento do software. Para cada um destes requisitos, defini-se os casos de teste associados para assegurar que os requisitos sejam verificáveis e fornecer a orientação necessária para verificação e validação.

Por fim, o foco desloca-se para o refinamento da definição do sistema. Isto consiste em detalhar os requisitos restantes e os casos de teste associados para direcionar a implementação e a verificação, e gerenciar a mudança nos requisitos.

4.2.1.1 Tarefa: Criar Casos de Teste

Esta tarefa resume em desenvolver os casos de teste e para os requisitos a serem testados. Os casos de testes são escritos pelo Analista de Testes no Plano de Teste. O Documento de Visão, Plano de Iterações, Casos de Uso e/ou SRS (*Software Requirements Specifications*) servem como entrada para a tarefa. A saída são os Casos de Teste, onde se tem os detalhas dos testes a serem realizados preenchidos no Plano de Teste documentado no Apêndice B.

Alguns passos básicos devem ser seguidos:

1. Revisar os requisitos a serem testados: Trabalhar com o Analista e o Desenvolvedor para identificar quais cenários necessitam de novos ou adicionais casos de teste. O Plano de Iteração deve ser revisado para assegurar o escopo de desenvolvimento para a iteração corrente foi entendido.
2. Identificar Casos de Teste relevantes: Identificar os caminhos nos cenários como condições únicas de teste. Considere os caminhos alternativos ou de exceções com perspectivas positivas e negativas.
3. Descrever os Casos de Teste: Para cada caso de teste, escrever uma descrição resumida com um resultado esperado. É desejável que um leitor casual possa entender claramente a diferença entre os casos de teste. Deve-se anotar as pré-condições e pós-condições lógicas que se aplicam a cada caso de teste. Opcionalmente, pode-se descrever os passos para o caso de teste.
4. Identificar os dados de teste necessários : Revisar cada caso de teste e anotar onde os dados de entrada ou saída possam ser necessários. Identificar o tipo, quantidade e singularidade do dado necessário e adicionar essas observações no caso de teste. Deve-se concentrar na articulação dos dados necessários e não na criação de dados específicos.
5. Compartilhar e avaliar os Casos de Teste : Percorrer os casos de teste com o Analista e o Desenvolvedor responsáveis pelo cenário relacionado. Se for possível, os *stakeholders* também devem participar. Deve-se perguntar aos participantes se eles concordam que se os casos teste passarem, eles considerarão os requisitos implementados.

   Durante a avaliação, assegure-se que:

- Os Casos de Uso e a Especificação de Requisitos não funcionais descritos no documento SRS e planejados para a iteração corrente, tenham casos de teste associados.
- Todos os participantes concordam com os resultados esperados dos casos de teste.

4.2.2 Atividade: Testar a Solução

A idéia de testar a solução somente quando a mesma for finalizada é errônea, esta atividade é repetida durante todo o ciclo de vida do projeto. A principal meta desta atividade é validar que a construção atual do sistema satisfaz os requisitos que lhe foram atribuídos (ISTQB, p.12, 2007).

Durante as iterações, a intenção é certificar que os requisitos implementados refletem uma arquitetura robusta, e que os requisitos restantes sejam implementados de forma consistente sob essa arquitetura. À medida que os desenvolvedores implementam a solução, para os requisitos de uma determinada iteração, deve-se submeter o código fonte integrado ao teste de unidade. Em seguida, um testador realiza testes sistema, em paralelo com o desenvolvimento, para certificar que a solução, que está sendo integrada continuamente, satisfaz a intenção especificada nos casos de teste. O testador define quais técnicas serão usadas, quais serão os dados de entrada e quais suítes de teste serão criadas. À medida que os testes forem sendo executados, os defeitos serão identificados e documentados na ferramenta de *bug tracker*, para que possam ser priorizados como parte do trabalho que será feito durante as próximas iterações.

4.2.2.1 Tarefa: Implementar os *Scripts* de Teste

Esta tarefa assegura que o *script* de teste esteja em conformidade com a especificação estabelecida no caso de teste. O caso de teste captura as condições lógicas de satisfação do teste, e o *script* de teste deve implementar esta intenção.

Esta tarefa é executada pelo Analista de Testes como entrada é necessário os casos de testes descritos no artefato Plano de Teste e como saída tem-se os *scripts* implementados.

Para implementar os scripts, deve-se seguir os seguintes passos:

1. Selecionar os Casos de Teste para implementar: Selecionar um conjunto de casos de teste para desenvolver um *script* de teste detalhado e executável. Trabalhar com o gerente de projeto e o desenvolvedor para determinar quais casos de teste precisam de

scripts de teste detalhados durante a iteração atual. No mínimo, casos de teste para os requisitos que estão planejados para a iteração atual ou a próxima devem ser selecionados.

2. Projetar o *Script* de Teste: É interessante rabiscar um esboço do *script* de teste como uma seqüência lógica de passos. Em seguida deve-se revisar os dados dos requisitos do caso de teste, e determinar se os dados existentes são suficientes, ou se é necessário desenvolver novos dados de teste para este *script* de teste. Selecione uma técnica de implementação para este desenho. Ao final, determinar se o *script* de teste será manual ou automático.
3. Implementar o *Script* de Teste executável: Desenvolver um *script* de teste procedural e detalhado baseado no seu projeto. Adotar um estilo que declara uma entrada exata e espera uma saída exata. Explicar as pré-condições que devem ser satisfeitas antes de executar este *script* de teste. Usar dados de teste temporários para os parâmetros do seu *script*. Garantir que cada pós-condição no caso de teste seja avaliada por passos no *script* de teste.
4. Definir dados de testes específicos: Especificar valores reais que sejam específicos para o *script* de teste ou referenciar dados de teste existentes. Por exemplo, ao invés de especificar "um bom número", indicar um valor real, tal como "3". Se o *script* de teste utilizar um conjunto de dados (tal como um arquivo ou banco de dados) acrescentar os novos dados de teste a ele e parametrizar o *script* de teste para recuperar os valores a partir do conjunto de dados. Caso contrário, adicionar valores de dados de teste executável nos passos do *script* de teste. Isto se aplica tanto aos scripts manuais quanto aos automatizados.
5. Organizar os Scripts de Teste em Suítes: Agrupar os testes em grupos relacionados. O agrupamento a ser utilizado vai depender do ambiente de teste. Visto que o sistema que está sendo testado está crescendo em sua própria evolução, crie suas suítes de teste para facilitar os testes de regressão, bem como a identificação da configuração do sistema.
6. Verificar a implementação do *Script* de Teste: Executar o *script* de teste para verificar que o mesmo implementa o caso de teste corretamente. Para testes manuais, deve-se percorrer o *script* de teste. Para testes automatizados, deve-severificar se o *script* de teste executa corretamente e produz o resultado esperado. Adicionar ou atualizar os *scripts* de teste na gestão de configuração.

4.2.2.2 Tarefa: Executar os Testes

Esta tarefa resume-se em executar os scripts de testes apropriados, analisar os resultados, tirar conclusões e comunicar os resultados dos testes para a equipe. A entrada para esta tarefa é a versão de software disponibilizada pela tarefa "Integrar e Criar a Solução" e os scripts de testes. Como saídas tem-se o artefato Registro de Teste, esta tarefa é executada pelo analista de teste.

Os passos desta tarefa são:

1. Revisar os itens trabalhos concluídos na construção;
2. Selecionar os Scripts de Teste;
3. Executar Scripts de Teste na construção;
4. Analisar e comunicar os resultados dos testes.

Deve-se executar todos os testes com a maior freqüência possível. Idealmente, execute-se todos os scripts de teste em cada construção implantada no ambiente de teste. Mesmo os scripts de teste que se espera falha, fornecem resultados valiosos. Entretanto, uma vez que um script de teste passe, ele não deverá falhar em construções subseqüentes da solução.

5
Conclusão

Este trabalho investigou princípios e técnicas ágeis voltadas para desenvolvimento de software que serviram de auxílio em um processo de desenvolvimento mais produtivo da equipe do LP&D.

As disciplinas de Gestão de Configuração e de Testes foram definidas e estão sendo implantadas na Fábrica de Software do LP&D, o processo de implantação é gradativo. Não se pode afirmar que as disciplinas foram perfeitamente definidas para o LP&D conforme o processo evoluir as disciplinas também devem ser adaptadas.

As equipes de testes já detectaram e documentaram erros que provavelmente seriam percebidos pelos usuários finais. O controle de versão e mudanças está sendo utilizado em 3 projetos em que as equipes compartilham código.Quando as disciplinas estiverem implantadas em sua totalidade será possível validar as mesmas e se necessário adaptá-las.

5.1 Trabalhos Futuros

Os conhecimentos obtidos através do desenvolvimento deste trabalho podem ser consideravelmente ampliados através de um trabalho que investiga a implantação das disciplinas de Gestão de Configuração e de Testes.

Conforme o SEI (2007, p.6), a qualidade de um produto ou sistema é altamente influenciada pela qualidade dos processos utilizados para planejar, desenvolver e manter estes produtos ou sistemas. Desta forma deve-se investigar se as disciplinas necessitam de modificações. Além disso será possível comparar se o processo de desenvolvimento de software do LP&D sofreu melhorias significativas após a implantação das disciplinas.

6Referências Bibliográficas

ANT. Apache Ant 1.8.2 Manual, 2004. Disponível em: <http://ant.apache.org/manual/index.html> Acesso em : 20 jun.2011.

ASTELS, D. *Test-Driven Development:* A Practical Guide. Upper Saddle River, New Jersey, 2003.

BECK, KENT. *Test Driven Development:* By Example, Addison Wesley, 2002.

BECK, KENT. *Extreme Programming Explained*: Embrace Change. Boston, 2004.

BECK, K.; BEEDLE, M., et al. Manifesto for Agile Software Development. Disponível em: < http://www.agilemanifesto.org>. Acesso em : 24 mai.2011.

COLLINS-SUSSMAN, Ben; FITZPATRICK , W. Brian; PILATO , C. Michael. *Version Control with Subversion, 2002.*

CRUISE CONTROL. CruiseControl Home, 2002. Disponível em: <http://cruisecontrol.sourceforge.net/>.Acesso em : 24 mai.2011.

CVS. *Concurrent Version System:* Open Source Version Control, 2006. Disponível em: <http://cruisecontrol.sourceforge.net/>.Acesso em : 26 mai.2011.

DIAS, André Felipe. O que é Gerência de Configuração? Disponível em: <http://www.pronus.eng.br/artigos_tutoriais/gerencia_configuracao/gerencia_configuracao.php>. Acesso em: 26 jun.2011.

FIGUEIREDO, Sávio; SANTOS, Gleison; ROCHA, Ana Regina. Gerência de Configuração em Ambientes de Desenvolvimento de Software Orientados a Organização. In: Anais do III Simpósio Brasileiro de Qualidade de Software. Brasília, 2004.

GEORGE, B., WILLIAMS, L. A. Structured Experiment of Test-Driven Development. Information and Software Technology (IST), 46, p. 337-342, 2003

GIT. Fast Version Control System, 2008 Disponível em: < http://git-scm.com/>.Acesso em : 24 mai. 2011.

HIGHSMITH, JIM. History: The Agile Manifesto.2001. Disponível em: <http://agilemanifesto.org/history.html> . Acesso em 26 set. 2010.

MOSER, Manfred; PRAKASH, Winston. *Hudson Book*: This is a placeholder cover.2011.

INTERNATIONAL SOFTWARE TESTING QUALIFICATIONS BOARD. *Base de Conhecimento para Certificação em Teste*: Foundation Level Syllabus., 2007.

LARMAN, G.;BASILI, V. R. *Iterative and Incremental Development:* A Brief History. IEEE Computer, 2003.

MARTIN, FOWLER. Continuous Integration.2000 Disponível em: <http://martinfowler.com/articles/originalContinuousIntegration.html>. Acesso 16 nov. 2010.

MARTIN, FOWLER. *Refatoração*: aperfeiçoando o projeto de código existente. Bookman, 2004.

MERCURIAL. Mercurial SCM, 2005. Disponível em: <http://mercurial.selenic.com/ >. Acesso em : 24 mai.2011.

OPENUP. Open Unified Process. Disponível em: < http://epf.eclipse.org/wikis/openup/>. Acesso em 06 jun.2011.

PILATO , C. Michael; COLLINS-SUSSMAN, Ben; FITZPATRICK , W. Brian. Version Control with Subversion, 2007.

PILONE, Dan; MILES, Russ. Use a Cabeça - Desenvolvimento de Software. Rio de Janeiro: Alta Books, 2008.

PRESSMAN, ROGER S. *Engenharia de Software*: Uma abordagem Prática. 6. ed. McGraw-Hill, 2006.

KOSKELA, LASSE. *Test Driven:* TDD and Acceptance TDD for Java Developers Manning Publications Co. , 2008.

SOFTWARE ENGINEERING INSTITUTE. *Process Maturity Profile:* CMMI® SCAMPI SM Class A Appraisal Results 2007 Mid-Year Update. Pitsburgh: SEI, 2007.

SELENIUMHQ. Selenium web application testing system. 2004. Disponível em: <http://mercurial.selenic.com/ >. Acesso em : 31 mai.2011.

SINIAALTO, MARIA. *Test driven development:* empirical body of evidence. Technical report, ITEA, Information Technology for European Advancement, 2006.

VISUAL SOURCEFACE, 2005. Disponível em : <http://msdn.microsoft.com/pt-br/library/ms181038%28v=vs.80%29.aspx> . Acesso em : 24 mai.2011.

WHITE Brian A. Software Configuration Management Strategies and Rational ClearCase. Addison-Wesley, 2001.

WILLIAMS, L., MAXIMILIEN, E.M., Vouk, M. Test-Driven Development as a DefectReduction Pratice.Proceedings of IEEE International Symposium on Software Reliability Engineering, Denver, CO, p. 34-45, 2003.

7Apêndice

7.1 Apêndice A - Ferramentas de Apoio a Disciplina de Gestão de Configuração e de Testes

Esta seção descreve ferramentas de auxílio que podem ser utilizadas na implantação das disciplina de Gestão de Configuração e Testes. O foco não é ensinar como instalar e utilizar as ferramentas e sim mostrar o propósito da utilização.

7.1.1 O Subversion

Segundo Collins-Sussman et al.(2002, p.18), o *Subversion* é um sistema de controle de versão, livre e de código aberto. Ou seja, o *Subversion* gerencia arquivos e diretórios, e as alterações feitas a eles, ao longo do tempo. Isso permite recuperar versões antigas de arquivos, ou examinar o histórico de como e quando os arquivos foram alterados. Sendo assim, o *SVN* responde quais arquivos mudaram, quando mudou e a autoria da mudança.

É comum, no processo de desenvolvimento de softwares,problemas como: perda de versões de arquivos, diferenciação de versões e controle de autoria. Tais problemas podem ser evitados com utilização do *Subversion*.

Por se tratar de uma aplicação cliente/servidor o *SVN* permite a edição colaborativa (múltiplos usuários) e o compartilhamento de dados. Todos os arquivos compartilhados estão dentro de um diretório referenciado como repositório.

O *Subversion* utiliza o modelo *copy-modify-merge* (copiar-modificar-mesclar). Nesse modelo não é necessário travar (*lock*) um arquivo para executar as modificações, evitando o problema de serialização, em que o usuário tem que esperar para modificar um arquivo que está sendo modificado por outro usuário (COLLINS-SUSSMAN et al., 2002, p.4).

Caso dois usuários alterem o mesmo arquivo, um deles primeiramente, irá salvar as alterações no repositório, o que é chamado de *commit*, quando o segundo

usuário executar o *commit* o *Subversion* mesclará as alterações, o que é chamado de *merge*, esta operação só é possível se não existir conflitos no arquivo, normalmente ocorre quando alterações são realizadas em partes distintas do arquivo.

Se um conflito existir, deverá ser resolvido pelo o usuário. "Note que software não tem como resolver os conflitos automaticamente; apenas pessoas são capazes de compreender e fazer as escolhas inteligentes" (COLLINS-SUSSMAN et al., 2002, p. 5). Além do *commit* e *merge* uma operação fundamental do *SVN* é o *update* em que a versão mais recente do projeto no servidor é copiada na máquina local.

Para possuir total controle sobre os arquivos, o *Subversion* mantém o diretório **.svn**, conhecido como o diretório administrativo da cópia local. Este diretório possui informações como arquivos desatualizados e/ou não publicadas no repositório.

7.1.2 *Builds*[1] Automatizados

Não é suficiente usar o controle de versões para assegurar que o código fique seguro. Deve-se também preocupar-se com a compilação do código e seu encapsulamento em uma unidade implantável (PILONE; MILES, p.183).

No projeto podem existir mais de uma classe executável, qual delas é a principal? Como deve ser executada? O que deve ser configurado? Estes questionamentos são exemplos de dúvidas que surgem para um novo membro da equipe, ou então para um membro que ficou afastado de um determinado projeto e quando regressou encontrou várias mudanças. Tais questionamentos podem ser respondidos pelas ferramentas de build automáticos, assim não será necessário que um membro pare seu produtivo trabalho para responder as dúvidas de um novato ou de uma pessoa que se ausentou por um período de tempo.

7.1.3 O *Ant*

O *Ant (*ANT, 2004*)* é uma ferramenta de *build* para Java que pode compilar código, criar e excluir diretórios e até mesmo empacotar arquivos. Tudo é baseado em um *script*, um arquivo XML, criado pelo especialista em *builds* automatizados, com

lviiilviii———————————

[1] Será utilizado o termo original *builds* ao invés de sua tradução para construções.

instruções para a ferramenta sobre o que fazer quando for preciso construir um programa (PILONE; MILES, p.185).

As etapas para o *build* de seu projeto são armazenadas em um arquivo .XML, geralmente chamado build.xml Tudo que for necessário para o *build* estará neste arquivo, escrito através de *tags* XML.

Uma importante *tag* é a *target*, dentro de cada *target* podem existir várias *tasks*, uma dessas *targets* é definida como padrão e será a primeira a ser executada quando invoca-se o *Ant*.

O arquivo de construção do *Ant* é dividido em quatro partes:

- *Project*
- *Property*
- *Target*
- *Tasks*

As próximas seções definem cada uma delas.

7.1.3.1 Project

É uma *tag* única que define seu projeto, tudo que estiver delimitado por esta *tag* fará parte do *build* de seu projeto. Além de definir o projeto *o target* padrão é definido aqui.

Exemplo:

<project name="MeuProjeto" default="nomeTargetPadrao">

7.1.3.2 Property

Servem basicamente para definir valores reutilizados no decorrer do *script*.

Exemplo:

```
<property name="src" location="src" />
<property name="subpastasrc" location="${src}/subpasta"/>
```

Conforme se vê na segunda linha do exemplo, pode-se utilizar o a diretiva ${nome-propriedade} para referenciar outra propriedade.

7.1.3.3 Target

Aqui são definidas e agrupadas as ações, por exemplo compilar e iniciar.

Exemplo:

<target name="compilar" depends="inicia"/>

Neste caso a *target* "compilar" depende da execução da *target* "inicia".

7.1.3.4 Tasks

Tasks servem para execução de comandos específicos e outros trabalhos básicos de um *script* de *build*:

Exemplo:

<mkdir dir="${src}/novapasta">

<javac srcdir="${src}" destdir="diretorioDestino" />

O *script* acima cria um novo diretório com o nome "novapasta" usando o valor da propriedade "src". Depois compila o conteúdo de " srcdir " e salva em "destdir".

7.1.4 O que deve conter um *Script* de *Build*?

Um bom *script* de *build* captura detalhes que o desenvolvedores podem não saber desde o início sobre como compilar e empacotar um aplicativo. (PILONE; MILES, p.189).

Basicamente um bom *script* de *build* executa 3 atividades :

1. Gerar documentação de si próprio.
2. Compilar e/ou empacotar o projeto.
3. Limpar arquivos e diretórios criados durante o processo de construção.

Além disso, atividades mais avançadas podem ser incluídas no *script*, como:

- Adicionar bibliotecas.
- Executar aplicativos.
- Gerar documentação JavaDoc do seu projeto.
- Executar testes.
- Criptografar arquivos.

Agora que o funcionamento básico do *Ant* e o que devem fazer bons *scripts* de *build* foram explicados, é possível utilizar o *Ant* para escrever bons *scripts* e executá-los em uma ferramenta de integração contínua para construir, empacotar e/ou testar nossa solução.

7.1.5 O Hudson

A escrita, execução e análise de resultados de testes é uma atividade que exige demasiado esforço, além de ser uma atividade periódica e nem sempre se pode contar com o comprometimento da equipe com a sua execução total. O mesmo pode-se dizer sobre scripts de *builds* do *Ant*.

Alguns testes também impõe certas características que dificultam sua execução, tais como testes que manipulam muitas dados. A solução proposta neste trabalho é o Hudson, uma ferramenta de integração contínua usada para automatizar *builds* e testes. A idéia é montar um servidor de testes e *builds* que quando encontra alterações no repositório SVN executa os *builds* e testes automaticamente. A ferramenta vem ganhando adeptos pela sua facilidade de uso e o grande número de complementos (*plugins*) disponíveis.

7.1.6 O Selenium

A SeleniumHQ disponabilza em seu site (http*://seleniumhq.org/projects*) várias ferramentas relacionados a testes as mais importantes para nosso contexto são: O Selenium IDE, Selenium RC e o Selenium Core.

O Selenium IDE é um ambiente integrado de desenvolvimento para *scripts* de testes automatizados. Ele é implementado como uma extensão do Firefox e permite gravar, editar e depurar os testes de forma rápida e fácil(SELENIUMHQ, 2004).

Por ser uma ferramenta gráfica integrada com o Firefox é muito simples criar seus scripts de testes utilizando o Selenium IDE. Além de falicitar a escrita e execução de testes de sistema a ferramenta também simplifica os testes de

regressão, já que a qualquer momento pode-se realizar um mesmo teste nas novas versões do sistema.

O Selenium IDE pode ser instalado como qualquer outro complemento do Firefox, basta acessar o site oficial (http://release.seleniumhq.org/selenium-ide/1.0.11/selenium-ide-1.0.11.xpi) e seguir as instruções.

Com o Selenium IDE aberto basta clicar com o botão direito em cima do elemento na página e adicionar o evento. Originalmente o script é gerado em HTML mas pode ser exportado para C#, PHP, Java e outras linguagens.

O Selenium RC é um servidor escrito em Java que interpreta os scripts criados pelo SeleniumHQ. Ele recebe chamadas http, executa os testes e envia de volta os resultados para o programa chamador. As chamadas vem de frameworks de testes unitários, como por exemplo o JUnit.

" O Selênio RC divide-se em duas partes:

- Um servidor, que executa automaticamente e e atua como um *proxy* HTTP para solicitações web a partir do browser.
- Bibliotecas cliente para a linguagem de programação favorita"(SELENIUMHQ, 2004).

Para instalar o Selenium RC basta fazer o download do mesmo no site oficial (*http://seleniumhq.com/download*), criar o diretório "C:\selenium" e salvar o Selenium RC no mesmo, crie também o subdiretório "C:\selenium\testes" para salvar seus scripts de testes escritos pelo Selenium IDE.

O Selenium *Core* é o motor de execução tanto do Selenium IDE quanto do Selenium RC e já vem embutido em ambos.

7.2 Apêndice B - Plano de Testes

Este apêndice apresenta o plano de testes desenvolvido para ser utilizado nos projetos do LP&D.

Nome do Projeto

Cliente: Nome do Cliente

PLANO DE TESTES

Versão 0.3

Responsável pelo Documento: [NOME]

Analista de Teste: **[NOME]**

7.2.1 Introdução

Este documento tem por finalidade descrever o plano geral das atividades de teste do projeto ***Nome do projeto***, fornecendo aos testadores e desenvolvedores as informações necessárias para a realização dos testes que compõe o sistema em desenvolvimento.

A atividade de teste constituirá uma forma de avaliar e agregar qualidade ao produto, reduzir custos e trabalho, dando maior confiabilidade ao software.

Serão definidas neste documento as estratégias de testes a serem adotadas a cada etapa do desenvolvimento do software, bem como a execução e armazenamento dos resultados dos testes.

7.2.2 Estratégias de teste

Para o projeto ***Nome do projeto***, serão adotadas as seguintes estratégias:

- Teste de Unidade
- Teste de Sistema

Descrição das estratégias:

7.2.2.1 Teste de Unidade

O teste de unidade é de responsabilidade do desenvolvedor , o mesmo deve testar as menores unidades de software desenvolvidas. O universo alvo desse tipo de teste são os métodos dos objetos ou mesmo pequenos trechos de código.

Tabela 1 - Teste de Unidade

Objetivo da Técnica:	Encontrar falhas de funcionamento dentro de uma pequena parte do sistema funcionando independentemente do todo.
Técnica:	O teste unitário consiste em elaborar métodos que testam outros métodos, ou seja, testam pequenas partes do sistema. É um teste automatizado onde a qualquer momento pode-se testar tudo apenas clicando em um botão. Para elaboração dos testes, utiliza-se um framework de testes. Permite desenvolvimento e testes em paralelo.
Técnica de execução:	A técnica exige as seguintes ferramentas:
Ferramentas Necessárias:	- IDE de desenvolvimento em PHP, para implementação dos testes de unidade;
Critérios de Êxito:	- PHPUnit, para execução dos testes criados.
Considerações Especiais:	Não se aplica.

7.2.2.2 Teste de Sistema

Este teste é geralmente executado por uma equipe de testes sob ponto de vista do usuário final, recomenda-se utilizar uma ferramenta de automação.

Tabela 2- Teste de Sistema

Objetivo da Técnica:	*Executar o sistema sob ponto de vista do usuário final, varrendo as funcionalidades em busca de falhas.*
Técnica:	Caixa Preta.
Técnica de execução:	Os testes serão executados em condições similares - de ambiente, interfaces sistêmicas e massas de dados - àquelas que um usuário utilizará no seu dia-a-dia de manipulação do sistema. De acordo com a política de uma organização, podem ser utilizadas condições reais de ambiente, interfaces sistêmicas e massas de dados.
Ferramentas Necessárias:	Sistema funcionando e ferramenta *Selenium IDE plugin Firefox* instalado.
Critérios de Êxito:	O sistema deve se comportar como o esperado (de acordo com o documento de requisitos).
Considerações Especiais:	Não se aplica.

7.2.3 Registro de defeitos

Quando forem identificados erros durante a realização de testes, estes devem ser registrados e classificados na ferramenta *MantisBT* (*http://www.mantisbt.org*), cada projeto possui um espaço na ferramenta com os todas as pessoas envolvidas.

Após o registro do *bug* é possível acompanhar todo o processo de resolução do problema, como os *bugs* ficam registrados é possível consultá-los para auxiliar na solução de novos *bugs* com características parecidas.

A equipe de teste deve estar atenta quanto a duplicação de registro de um mesmo *bug*, isso só atrapalha a correção do mesmo pela equipe de desenvolvimento. Então antes de relatar um erro recomenda-se ler os últimos registrados e ainda não resolvidos.

A equipe de desenvolvimento resolverá os problemas conforme a prioridade e gravidade indicadas no registro do *bug*. Os seguintes detalhes devem ser informados sempre que possível, para simplificar o entendimento da equipe de desenvolvimento:

- Gravidade;
- Prioridade;
- Freqüência em que acontece;
- Plataforma;
- Resumo;
- Descrição;
- Passos para reproduzir;
- Informações Adicionais.

Ainda sendo possível anexar um arquivo como um *Print Screen* da tela com o erro.

7.2.4 Classificação da Prioridade dos Erros

Uma vez identificado algum erro sobre o requisito, deve-se proceder com a classificação do mesmo. Para tanto deve-se utilizar as informações da tabela a seguir, a serem definidas no documento Relatório de Testes.

Tabela 3 - Prioridade de Erros

Prioridade	*Descrição*	**Reação**
1. Urgente	Os defeitos resultam em falhas em todo o sistema ou em partes do sistema.	Resolver imediatamente.
2. Alta	Os defeitos resultam em falhas do sistema, entretanto existem alternativas de processo (manuais, por exemplo) que produzirão os resultados desejados.	Dar alta atenção(assim que possível).
3. Normal	Os defeitos não resultam em falhas, mas produzem resultados inconsistentes, incompletos ou incorretos ou prejudicam a usabilidade do software.	Fila normal.
4. Baixa	Os defeitos não causam uma falha, não prejudicam a usabilidade e os resultados do processamento desejado são obtidos contornando-se o problema.	Baixa Prioridade.
5. Nenhuma	O defeito resulta de uma requisição de alteração ou melhoria, que pode ser indeferida.	Deferir ou não (longo prazo).

7.2.5 Aprovação

Para cada iteração do ciclo de vida do software devem ser definidos os critérios de aceitação para os testes que serão realizados de acordo com medidas de qualidade predefinidas.

Um teste passa quando todos os procedimentos são executados com sucesso, falha quando ocorre uma divergência entre a saída produzida pelo sistema e a saída esperada descrita na verificação do caso de teste e fica bloqueado quando as precondições não podem ser satisfeitas durante a execução.

7.2.6 Critérios de Finalização

Tabela 4 - Critérios de Finalização

Item	*Verificar*	**Aprovar**
Funcionalidade	Deve ser verificada a presença de todas as funções mencionadas; a execução correta destas funções; a ausência de contradições entre a descrição do produto e a documentação do usuário.	Só serão aprovados requisitos **100**% implementados e sem erros graves.
Confiabilidade	O usuário deve manter o controle do produto, sem corromper ou perder dados, mesmo que a capacidade declarada seja explorada até os limites ou fora deles, se uma entrada incorreta é efetuada, ou ainda se instruções explícitas na documentação são violadas.	Só serão aprovados requisitos que respeitem **100**% do descrito ao lado.

7.2.7 Itens a serem testados

7.2.7.1 Nome do Item a ser testado Ex: Visão Gerencial

Descrição do item a ser testado.

Ex: Realizar testes nas consultas das faturas do mês corrente e dos meses anteriores do sistema.

Realizar testes no link do mês/ano que se deseja consultar a fatura.

Tabela 5 - Caso de Teste

Caso de Teste [Nome do Projeto] ou Módulo do Sistema	
Caso de teste:	É um código exclusivo para identificar o caso de teste.
Pré-condições:	Indica o estado inicial do sistema para começar a execução dos testes.
Procedimentos:	1- Passos necessários para chegar aos resultados esperados.
Resultado esperado:	O que se espera.
Dados de entrada:	Dados necessários para executar uma ação no sistema. Ex: usuário e senha.
Critérios especiais:	Se existem critérios especiais.
Ambiente:	Ambiente onde deverá ser executado o caso de teste.
Implementação:	Manual ou automatizado.
Iterações:	Número de iterações.

7.2.8 Pessoas e Papéis

Esta tabela mostra as responsabilidades do perfil da equipe do esforço de teste.

Tabela 6 - Papéis e Responsabilidades

Papel	*Responsabilidades ou Comentários Específicos*
Analista de Testes	Identifica e define os testes específicos a serem conduzidos. Estas são as responsabilidades: • identificar idéias de teste; • definir detalhes dos testes; • definir os resultados esperados pelos testes; • Planejar os testes. • implementar os testes e os conjuntos de testes; • executar os conjuntos de testes; • registrar os resultados; • analisar as falhas dos testes e possibilitar a recuperação posterior; • documentar solicitações de mudança.
Desenvolvedor	Implementa e executa os testes unitários. Estas são as responsabilidades: • implementar os testes unitários utilizando framework de testes;

7.3 Apêndice C - Plano de Gerência de Configuração

Qualquer um que estiver envolvido nas atividades que podem alterar os Itens de Configuração dos projetos e estiver envolvido em algum projeto no Laboratório de Pesquisa e Desenvolvimento (LP&D) deve ler este documento. Ele apresenta procedimentos e padrões para assegurar o controle dos Itens de Configuração o tempo todo.

7.3.1 Propósito

Este documento apresenta procedimentos e padrões relacionados à gerência da configuração de software no LP&D. Qualquer atividade que cause a modificação de Itens de Configuração, artefatos, deve seguir os procedimentos desse documento a fim manter a integridade dos Itens de Configuração.

7.3.2 Escopo

Este documento detalha procedimentos gerais e padrões que devem ser aplicados na Gerência de Configuração dos projetos.

7.3.3 Armazenamento e Controle de Versão

Todos os projetos desenvolvidos no LP&D são armazenados no servidor *SVN Assembla*, que podem ser acessados através da *URL*:

https://subversion.assembla.com/svn/**nome_do_projeto**

Tabela 7 - Detalhes sobre as pastas do projeto

Pasta	*Descrição*
doc	Contêm a documentação disponível do sistema

trunk	Contêm o código vivo do sistema, onde os integrantes desenvolvem as alterações nos ICs (Itens de Configuração). Nesta são implementados correções de *bugs*, customizações e novos projetos.
branches	Contêm as versões que são enviadas aos clientes. São para alterações que não se quer na *trunk* ou para manter o código separado das alterações feitas na *trunk*.
tags	Contêm as versões que foram instaladas nos clientes. (marcos)

7.3.4 *Trunk, Braches* e *Tags*

Todos os padrões de manipulação e nomenclatura das pastas *trunk*, *branches* e *tags* estão definidos nesta seção. O fluxo das relações pode ser vista na ultima seção 12.5.

7.3.4.1 Trunk

Pasta principal do projeto, utilizada para manter a versão completa de desenvolvimento. Todas as implementações do projeto estão na *trunk*.

- **Alteração**

Todo desenvolvimento do projeto ocorre na *trunk*. Antes que as alterações sobre os ICs sejam enviados para o repositório, as alterações devem ser validadas e testadas na estação local do desenvolvedor do projeto.

7.3.4.2 Branches

Branches são usadas para customizar e corrigir *tags* que não foram aprovadas pelo cliente. As *branches* tendem a representar as versões estáveis dos clientes.

- **Criação**

Branches podem ser criadas a partir das *tags* ou da *trunk*, no caso de *branches* para teste. Este procedimento pode ser efetuado por qualquer desenvolvedor do projeto.

- **Alteração**

Toda correção de *bug* e customização feita na *branch* que for aprovada pelo cliente, deverá passar por um procedimento de merge manual da *branch* para a *trunk*. O merge deve ocorrer da *branch* de teste com a *trunk* toda vez que um erro for corrigido na *branch*.

- **Nomenclatura**

As *branches* são identificadas por um conjunto de 3 casas numéricas, como é mostrado na representação a seguir:

Exemplos: 3.2.0, 3.2.1, 3.2.2

Toda vez que uma *branch* for gerada a partir de uma *tag*, é incrementado o valor de n3. Decisões comerciais podem gerar mudanças na primeira casa numérica (n1). O valor de n2 é alterado somente através de modificações relacionadas com *tags*. As *branches* de teste são reconhecidas pelo valor de n2 ser igual a zero.

7.3.4.3 Tags

Tags são versões fechadas e imutáveis que são validadas pelo cliente. Uma *tag* é fechada a partir de uma *branche* ou da trunk.

- **Criação**

Inicialmente toda *tag* gerada é um protótipo a ser validada pelo cliente. Em caso de aprovação, essa *tag* se transforma em uma *tag* final do projeto.

- **Alteração**

Uma *tag* não pode ser modificada. Se a *tag* enviada para o cliente estiver com *bugs*, esta de ser corrigida em uma nova *branch* gerada a partir dessa *tag*.

- **Nomenclatura**

As *tags* são identificadas por um conjunto de 4 casas numéricas, como na representação a seguir:

- *tag_p_3.2.0, tag_f_3.3.0*

O valor de n2 é incrementado toda vez que uma nova *tag* for gerada, seja essa a partir da *trunk* ou de uma *branche*. A variável X pode assumir dois valores distintos:

- Letra p, quando a *tag* for um protótipo a ser validado pelo cliente;
- Letra f, quando a *tag* for aprovada pelo cliente.

7.3.5 Fluxograma do Processo

A imagem abaixo ilustrada as interações do processo de controle de versões do processo de desenvolvimento.

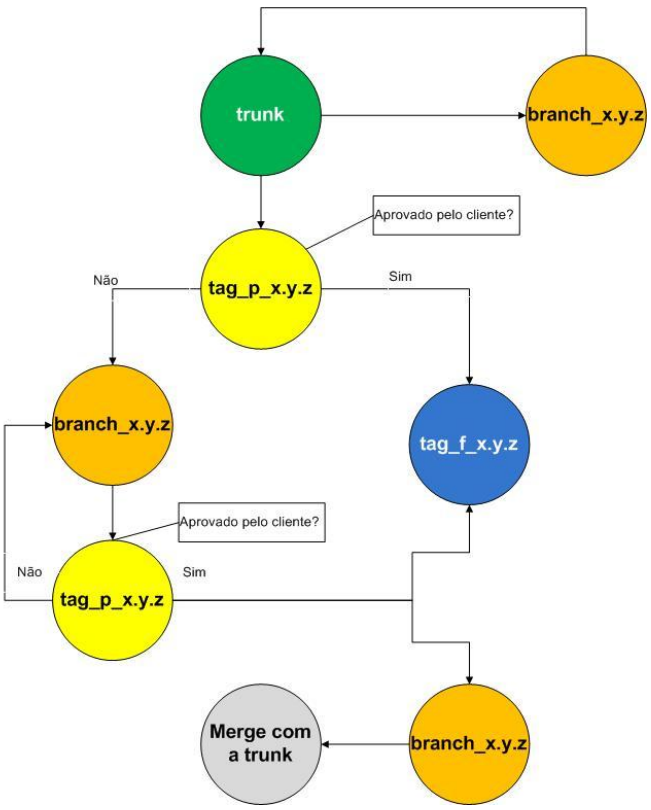

Figura 4 - Fluxograma do Processo

7.3.6 Auditorias

As auditorias podem ocorrer a qualquer hora. O responsável pela auditoria é o Gerente de Projetos.

7.3.7 Procedimentos e verificações

Auditorias devem verificar se o projeto esta de acordo com todos os padrões definidos neste documento.

7.3.8 Relatórios

Toda auditoria deve ser relatada através de um documento de texto.