GGAS
AMBIENTE DE CONTROLE DE QUALIDADE E PROCESSO DE AUDITORIA

Documento Descritivo do Processo

Esse documento representa, de forma técnica, porém resumida, uma parte do processo de auditoria de *check in* e *check out* de código fonte, ferramentas utilizadas neste e dos fluxos dos processos de análise.

Histórico de Alterações

Data	Versão	Descrição	Autor
03/02/2015	1.0	Criação do documento.	Roberto Alencar

Sumário

1 - INTRODUÇÃO

O sistema GGAS é o Sistema de Gestão Comercial de Gás Natural que visa atender, de forma abrangente, as necessidades inerentes à Área comercial de uma empresa distribuidora de gás natural e possibilita a gestão dos cadastros, da medição, dos contratos, do faturamento (incluindo emissão das NF-e's), da cobrança, da arrecadação além de disponibilizar dados para a integração com sistemas da área contábil, financeira, operacional e gerencial.

Esse documento tem por objetivo descrever, de forma técnica, mas resumida, o funcionamento do processo de Auditoria e Controle de versões que funcionará a partir de 01/03/2015 para todas as empresas prestadoras de serviço do sistema GGAS contratadas pelas Companhias Distribuidoras Locais (CDL) de gás natural que possuam contratos de desenvolvimento, implantação e manutenção (preventiva e corretiva). O documento estará disponível no ambiente de gerenciamento de chamados / requisições para todas as distribuidoras.

2 - ALINHAMENTO DE CONCEITOS

Neste capítulo serão apresentados conceitos como sistema de controle de versão, integração contínua, ferramentas para testes e auditoria de código, entre outros, considerados essenciais para o entendimento do processo proposto por esse documento.

2.1 - Engenharia de Software

O software é o conjunto de vários artefatos e não apenas o código fonte. Engenharia de Software é uma área da computação voltada à especificação, desenvolvimento e manutenção de sistemas de software, com aplicação de tecnologias e práticas de gerência de projetos e outras disciplinas, visando organização, produtividade e qualidade. Desta forma, pode-se considerar que engenharia de software é uma abordagem sistemática e disciplinada para o desenvolvimento de software.

Uma das grandes dificuldades da engenharia do software é resolver o problema e deixar o cliente satisfeito com o software. Engenheiros de software devem adotar uma abordagem sistemática e organizada para o seu trabalho e usar técnicas e ferramentas apropriadas, de acordo com o problema a ser resolvido, e com as restrições e recursos disponíveis.

O *Guide to the Software Engineering Body of Knowledge*, conhecido pela sigla SWEBOK, é 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. O SWEBOK apresenta uma classificação hierárquica dos tópicos tratados pela Engenharia de Software, onde o nível mais alto são as Áreas do Conhecimento. Dentre as áreas do conhecimento abordadas no SWEBOK, as que terão enfoque neste documento, serão as áreas de: Gerência de Configuração de Software, Testes de Software e Qualidade de Software.

2.2 - Sistema de Controle de Versão

Um sistema de controle de versão é um software com a finalidade de gerenciar diferentes versões dos códigos-fontes e também da documentação em um determinado projeto. Nele, é possível registrar as mudanças feitas em um arquivo ou um conjunto de arquivos ao longo do tempo de forma que você possa recuperar versões específicas. Com isso, ele te oferece uma maneira muito mais inteligente e eficaz de organizar seu projeto, pois é possível acompanhar um histórico de desenvolvimento, desenvolver paralelamente e ainda te oferecer outras vantagens, como exemplo, customizar uma versão, incluir outros requisitos, finalidades específicas, layout e afins sem mexer no projeto principal ou resgatar o sistema em um ponto que estava estável, isso tudo sem mexer na versão principal.

Como funciona?

Basicamente, os arquivos do projeto ficam armazenados em um repositório (um servidor em outras palavras) e o histórico de suas versões é salvo nele. Os desenvolvedores podem acessar e resgatar a última versão disponível e fazer uma cópia local, na qual poderão trabalhar em cima dela e continuar o processo de desenvolvimento. A cada alteração feita, é possível enviar novamente ao servidor e atualizar a sua versão a partir outras feitas pelos demais desenvolvedores.

Figura 1 – Esquema de funcionamento dos controles de versão

Atualmente, os sistemas de controle de versão são classificados em dois tipos: Centralizados e distribuídos.

O **centralizado** trabalha apenas com um servidor central e diversas áreas de trabalho, baseados na arquitetura cliente-servidor. Por ser centralizado, as áreas de trabalho precisam primeiro passar pelo servidor para poderem comunicar-se. Essa versão atende muito bem a maioria das equipes de desenvolvimento que não sejam enormes e trabalhem em uma rede local, além de não ter problemas de velocidade para enviar e receber os dados e ter um bom tempo de resposta do servidor. Um dos principais sistemas com o tipo de controle de versão centralizado é o Subversion, ou simplesmente SVN.

O **distribuído** vai mais além. Ele é recomendado para equipes com muitos desenvolvedores e que se encontram em diferentes filiais. Esta versão funciona da seguinte maneira: cada área de trabalho tem seu próprio "servidor", ou seja, as operações de *check-in* e *check-out* são feitas na própria máquina. Porém diferentemente do centralizado, as áreas de trabalho podem comunicar-se entre si, recomenda-se usar um servidor como centro do envio dos arquivos para centralizar o fluxo e evitar ramificações do projeto e a perda do controle sobre o mesmo, geralmente o sistema te dá essa opção, oferecendo um servidor remoto para hospedar o projeto. A comunicação entre o servidor principal e as áreas de trabalho funciona com outras duas operações, para atualizar e mesclar o projeto, chamadas de *pull* e *push* (puxar e empurrar).

- *pull*: Com esta operação é possível pegar a versão de outra área de trabalho e mesclar com a sua.
- *push*: Com esta operação temos o processo inverso do *pull*, ou seja, enviando para outra área a sua versão do projeto.

Por ser na própria máquina, o sistema de controle distribuído acaba sendo mais rápido, porém exige maior conhecimento da ferramenta e de início podem atrapalhar o desenvolvedor. Como exemplo, o sistema de mesclagem em edições concorrentes, se torna diferente por trabalhar em um sistema de arquivos binários (sequenciais de bits compostos por zero e um) que em determinadas situações não permite a comparação entre atualizações concorrentes. O sistema centralizado trabalha com arquivos de texto, que permite a comparação em atualizações concorrentes e da opção ao desenvolvedor para escolher a melhor solução.

Portanto, por esse tratamento de mesclagem ser diferente, podem ocorrer situações onde o trabalho de alguém possa ser sobreposto e gerando tormento para os desenvolvedores. Para isso existe uma função chamada *lock*, que bloqueia o arquivo para que não seja modificado por outros enquanto estiver com você. Os sistemas distribuídos mais conhecidos são o Git e o Mercurial. **O sistema que será utilizado para gestão e controle de versão do projeto GGAS será o Git.**

2.2.1 - Git

Git é um sistema de controle de versão distribuído com ênfase em velocidade. Apesar de possuir uma interface parecida, o Git armazena e pensa sobre informação de uma forma totalmente diferente desses outros sistemas; entender essas diferenças lhe ajudará a não ficar confuso ao utilizá-lo.

Normalmente a maioria dos controles de versão guardam as mudanças do código como alterações de um determinado arquivo. Ou seja, a cada mudança no arquivo, o sistema guarda essa mudança apenas e não o arquivo inteiro. O Git pensa um pouco diferente: ele trata os dados como snapshots. Cada vez que commitamos (commitar é enviar alterações para o controle de versão) ou salva o estado do projeto no Git, ele basicamente guarda um snapshot de como todos os arquivos estão naquele momento e guarda a referência desse estado. Para os arquivos que não foram modificados, ele

não guarda uma nova versão, ele apenas faz um link para a versão anterior idêntica que já foi guardada em outro momento.

Para maiores informações, ver:
http://git-scm.com/book/pt-br/v1/Primeiros-passos-No%C3%A7%C3%B5es-B%C3%A1sicas-de-Git

2.3 - Integração Contínua

Muito se fala sobre metodologias ágeis, mas há necessidade de utilização de algumas técnicas para que estas metodologias tenham sucesso, uma das técnicas é a integração contínua.

A integração contínua é um termo originado na metodologia ágil XP e utilizado em diversas metodologias, consistindo em algo simples: o desenvolvedor integra o código alterado e/ou desenvolvido ao projeto principal na mesma frequência com que as funcionalidades são desenvolvidas, sendo feito muitas vezes ao dia em vez de apenas uma vez. O objetivo principal de utilizar a integração contínua é verificar se as alterações ou novas funcionalidades não criaram novos defeitos no projeto já existente. A prática da integração contínua pode ser feita através de processos manuais ou automatizados, utilizando ferramentas como o **Jenkins**, Hudson entre outros.

"Integração Contínua é uma pratica de desenvolvimento de software onde os membros de um time integram seu trabalho frequentemente, geralmente cada pessoa integra pelo menos diariamente – podendo haver múltiplas integrações por dia. Cada integração é verificada por um build automatizado (incluindo testes) para detectar erros de integração o mais rápido possível. Muitos times acham que essa abordagem leva a uma significante redução nos problemas de integração e permite que um time desenvolva software coeso mais rapidamente."

Martin Fowler

Para a utilização contínua, é importante que a equipe possa trabalhar em grupo, para isso é importante que se tenha um sistema de controle centralizado de versão.

Para maiores informações, ver:
http://www.devmedia.com.br/integracao-continua-uma-introducao-ao-assunto/28002
http://en.wikipedia.org/wiki/Continuous_integration

2.3.1 - Jenkins

A ferramenta Jenkins é uma forma de prover integração contínua, uma das práticas mais importantes do desenvolvimento ágil. Através dela, é possível agilizar tarefas demoradas como a compilação de um projeto e a execução dos seus testes automatizados. Com um servidor de integração contínua bem configurado, essas tarefas são executadas a cada mudança no repositório de código e, em caso de erros de compilação ou falhas nos testes automatizados, todos os desenvolvedores são alertados rapidamente. Dessa forma, se o servidor de integração não aponta problemas no projeto, a equipe tem a segurança de que as mudanças no código estão de acordo com a bateria de testes.

Jenkins é um sistema de Integração Contínua (CI), projetado para fazer builds automáticos de um projeto a partir de gatilhos pré-definidos (periódico, a cada push no repositório, ao acessar uma URL, etc).

A ideia é garantir que apenas builds de sucesso possam ser publicados em produção, usando para isso um cartridge Jenkins, disponível na plataforma.

Para maiores informações, ver:
https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins
http://blog.caelum.com.br/integracao-continua-de-projeto-java-com-jenkins/
https://getupcloud.com/blog/integracao-continua-com-jenkins

2.3.2 - Sonar

O Sonar é uma plataforma aberta para gerenciar a qualidade de código e extremamente útil quando usado dentro de um ambiente de integração contínua. O Sonar é uma ferramenta *open source* construída para avaliar a qualidade do código que está sendo desenvolvido. Questões de arquitetura, código duplicado, pontos potenciais de bugs, nível de complexidade e cobertura dos testes são avaliadas pelo Sonar, de modo a dar para a equipe um feedback da qualidade do seu código.

Ele cobre os sete eixos da qualidade de código: arquitetura e desenho, duplicações de código, testes unitários, comentários de código, complexidade de código, bugs potenciais e análise estática baseada em regras.

Através do Sonar é possível fazer o acompanhamento da qualidade do código de seus projetos. Ele possui uma interface web e gráficos onde é possível ver nitidamente a evolução da qualidade ao longo das versões geradas

Para maiores informações, ver:
http://www.sonarqube.org/

2.4 - Teste de Sistemas

O teste do software é a investigação do software a fim de fornecer informações sobre sua qualidade em relação ao contexto em que ele deve operar. Isso inclui o processo de utilizar o produto para encontrar seus defeitos. O teste de software pode ser visto como uma parcela do processo de qualidade de software.

De forma geral, mensurar o bom funcionamento de um software envolve compará-lo com elementos como especificações, outros softwares da mesma linha, versões anteriores do mesmo produto, inferências pessoais, expectativas do cliente, normas relevantes, leis aplicáveis, entre outros. Enquanto a especificação do software diz

respeito ao processo de verificação do software, a expectativa do cliente diz respeito ao processo de validação do software. Por meio da verificação será analisado se o produto foi feito corretamente, se ele está de acordo com os requisitos especificados. Por meio da validação será analisado se foi feito o produto correto, se ele está de acordo com as necessidades e expectativas do cliente.

Um desenvolvimento de software organizado tem como premissa uma metodologia de trabalho. Esta deve ter como base conceitos que visem a construção de um produto de software de forma eficaz. Dentro desta metodologia estão definidos os passos necessários para chegar ao produto final esperado.

Assim, quando se segue uma metodologia para o desenvolvimento de um produto de software, espera-se um produto final que melhor agrade tanto aos clientes quanto ao próprio fornecedor, ou seja, a empresa de desenvolvimento. Observando este aspecto, não faz sentido iniciar a construção de um produto de software sem ter uma metodologia de trabalho bem solidificada e que seja do conhecimento de todos os envolvidos no processo. Porém, além de uma crescente demanda por softwares de qualidade, as empresas de desenvolvimento de software sofrem cada vez mais pressão por parte dos clientes para que o produto seja entregue num curto período de tempo. Este fato pode fazer com que uma sólida metodologia de trabalho acabe por se desequilibrar.

Independentemente da metodologia de trabalho empregada no desenvolvimento de um software, para que se obtenha um produto final com um certo nível de qualidade é imprescindível a melhoria dos processos de engenharia de software.

Uma maneira viável para se assegurar a melhoria de tais processos seria tomar como base modelos sugeridos por entidades internacionais respeitadas no assunto. Dentro de uma gama de modelos, sejam eles para situações e ambientes específicos ou para soluções genéricas, existem alguns que são mais utilizados e tidos como eficientes, como por exemplo os SW-CMM, SE-CMM, ISO/IEC 15504 e o mais conhecido CMMI.

Outro fator com grande influência sobre a qualidade do software a ser produzido é o que diz respeito aos testes que serão executados sobre tal produto. Todas as metodologias de desenvolvimento de software têm uma disciplina dedicada aos testes. Atualmente esta é uma tarefa indispensável, porém muitas vezes efetuada de maneira ineficiente, seja pelo subestimar dos que desenvolvem, pela falta de tempo ou mesmo pela falta de recursos humanos e financeiros.

De acordo com um estudo conduzido pelo NIST em 2002, os defeitos resultam num custo anual de 59,5 bilhões de dólares à economia dos Estados Unidos. Mais de um terço do custo poderia ser evitado com melhorias na infraestrutura do teste de software.

Para maiores informações, ver:
http://pt.wikipedia.org/wiki/Teste_de_software

2.4.1 - Testes Automatizados

Automação de teste é o uso de software para controlar a execução do teste de software, a comparação dos resultados esperados com os resultados reais, a configuração das pré-condições de teste e outras funções de controle e relatório de teste. De forma geral, a automação de teste pode iniciar a partir de um processo manual de teste já estabelecido e formalizado.

Apesar do teste manual de software permitir encontrar vários erros em uma aplicação, é um trabalho maçante e que demanda um grande esforço em tempo. Também, pode não ser efetivo na procura de classes específicas de defeitos. A automação é o processo de escrita de um programa de computador para realizar o teste. Uma vez automatizado, um grande número de casos de teste podem ser validados rapidamente. As vantagens da automação tornam-se mais evidentes para os casos de softwares que possuem longa vida no mercado, devido ao fato de que até mesmo

pequenas correções no código da aplicação podem causar a quebra de funcionalidades que antes funcionavam.

A automação envolve testes de caixa-preta, em que o desenvolvedor não possui conhecimento sobre a estrutura interna do sistema, ou caixa branca, em que há pleno conhecimento da estrutura interna. Para os dois casos, a cobertura de teste é determinada respectivamente pela experiência do desenvolvedor ou pela métrica de cobertura de código.

Para maiores informações, ver:
http://pt.wikipedia.org/wiki/Automa%C3%A7%C3%A3o_de_teste

2.4.1.1 - JUnit

O JUnit é um framework open-source, criado por Erich Gamma e Kent Beck, com suporte à criação de testes automatizados na linguagem de programação Java.

Esse framework facilita a criação de código para a automação de testes com apresentação dos resultados. Com ele, pode ser verificado se cada método de uma classe funciona da forma esperada, exibindo possíveis erros ou falhas podendo ser utilizado tanto para a execução de baterias de testes como para extensão.

Com JUnit, o programador tem a possibilidade de usar esta ferramenta para criar um modelo padrão de testes, muitas vezes de forma automatizada.

O teste de unidade testa o menor dos componentes de um sistema de maneira isolada. Cada uma dessas unidades define um conjunto de estímulos (chamada de métodos), e de dados de entrada e saída associados a cada estímulo. As entradas são parâmetros e as saídas são o valor de retorno, exceções ou o estado do objeto. Tipicamente um teste unitário executa um método individualmente e compara uma saída conhecida após o processamento da mesma.

Para maiores informações, ver:

http://junit.org/

http://pt.wikipedia.org/wiki/JUnit

2.4.1.2 - Selenium

Selenium é uma ferramenta para testar aplicações web pelo *browser* de forma automatizada. Selenium se refere ao *Acceptance Testing* (ou functional testing) que envolve rodar testes num sistema finalizado. Os testes rodam diretamente num *browser*, exatamente como o usuário faria. Desta forma, a ferramenta promove testes funcionais no estilo ClickAndReplay, onde é permitido gravar cliques e inserção de dados em sua aplicação web e repeti – los depois.

Imagine um formulário com 50 campos a serem preenchidos, e toda ves que formos executar o teste deste formulário devemos preenche os 50 campos. Inviável!

O Selenium é uma ferramenta de testes funcionais para aplicações Web. De forma geral, o Selenium oferece funcionalidades de:

- Implementação de casos de teste;
- Execução automática de casos de teste;
- Geração de relatórios de teste;

Dentro do processo de testes automatizados proposto para o GGAS, o JUnit ficará encarregado pela estruturação dos casos de teste, verificação das saídas e pelos testes propriamente ditos. A grosso modo, comparando valores esperados com valores retornados pelo software, nos informando se os testes passaram ou falharam. Já o Selenium, ficará responsável por automatizar as entradas de dados no navegador, executando ações como clicks, submissão de formulários, seleção de checkboxes, inserção de dados em inputs, etc.

Para maiores informações, ver:
http://www.devmedia.com.br/introducao-aos-testes-funcionais-automatizados-com-junit-e-selenium-webdriver/28037

2.4.2 - Testes Manuais

Além dos testes automatizados descritos anteriormente, serão executados eventualmente testes manuais do tipo exploratório e de performance dentro do processo de controle de qualidade e auditoria do sistema GGAS.

2.4.2.1 - Testes Funcionais Exploratórios

Teste exploratório é uma abordagem de testes que enfatiza as habilidades do testador em tomar decisões sobre o que será testado durante a execução do teste ao invés de seguir um roteiro previamente planejado. No entanto, o conceito não é novo. Glenford Myers em 1979, no seu livro chamado *"The Art of Software Testing"*, já preconizava conceitos embrionários de testes de natureza exploratória, quando discutia técnicas de testes de suposição de erros (do inglês: *"Error Guessing Testing"*).

Teste exploratório é muitas vezes confundido ou usado como sinônimo de teste ad hoc. No entanto, podemos afirmar que teste exploratório é um tipo de ad hoc, já que ambos são testes não sistemáticos que dependem da experiência e intuição do testador. No entanto, as diferenças param por aqui, já que no teste exploratório temos uma estrutura mais sofisticada em comparação com o teste puramente ad hoc.

Para maiores informações, ver:
http://www.qualister.com.br/blog/testes-exploratorios-parte-1-introducao

2.4.2.2 - Testes de Performance

Testes de performance ou desempenho são essenciais por avaliarem a capacidade de resposta de um sistema em determinados cenários e configurações e por permitirem planejar melhorias no ambiente para atender a demandas atuais e futuras.

Tipos de testes de desempenho:

- **Teste de Carga:** Teste realizado para verificar se um sistema suporta uma determinada carga desejada.
- **Teste de estresse:** Teste feito para determinar a capacidade máxima do sistema.
- **Teste de estabilidade:** Teste realizado para verificar se o sistema degrada o desempenho com o tempo.

Existem muitas razões para o teste de carga em aplicações Web. O tipo mais básico de teste de carga é usado para determinar o comportamento da aplicação Web através de condições normais e altos picos de carga. À medida que se começa o teste de carga, é recomendável se começar com um pequeno número de usuários virtuais(Virtual Users) e então incrementar gradualmente a carga do normal até o pico.

Para maiores informações, ver:
http://crowdtest.me/testes-desempenho-conceitos/
http://www.linhadecodigo.com.br/artigo/3259/testes-de-performance-testes-de-carga-stress-e-virtualizacao-parte-3.aspx

2.5 - Auditoria de Código

Mesmo depois de entregue, o trabalho em cima do código fonte de uma aplicação não para. A fim de validar e certificar pontos relacionados a segurança e otimização de performance uma auditoria de código pode ser feita.

O principal intuito da auditoria é verificar se técnicas de programação foram utilizadas de forma correta, além de identificar possíveis pontos falhos ou até mesmo códigos que não sejam pertinentes a aplicação.

Depois de feita a análise um relatório de resultados é entregue e com base nele uma série de medidas são implementadas, garantindo assim a segurança e confiabilidade do código da aplicação. Empresas que trabalham com dados importantes de seus clientes como no caso dos bancos tem na auditoria de código fonte uma alternativa altamente confiável para validar suas aplicações antes de colocá-las em ambiente de produção.

2.5.1 - Auditoria Automatizada

Dentro do processo de controle de qualidade e auditoria automática do sistema GGAS, serão utilizadas algumas ferramentas das quais são descritas a seguir.

2.5.1.1 - Cobertura

As principais métricas de um teste incluem a cobertura e a qualidade.

A cobertura é a métrica da abrangência do teste e é expressa pela cobertura dos requisitos e casos de teste ou pela cobertura do código executado. A qualidade é uma medida de confiabilidade, de estabilidade e de desempenho do objetivo do teste (sistema ou aplicativo em teste). Ela se baseia na avaliação dos resultados do teste e na análise das solicitações de mudança (defeitos) identificadas durante o teste.

As métricas de cobertura fornecem respostas à pergunta “Qual é a abrangência do teste?”. As medidas de cobertura usadas com mais frequência são a cobertura de teste baseada em requisitos e em códigos. Em resumo, a cobertura de teste é qualquer medida de abrangência relacionada a um requisito (baseada em requisitos) ou a um critério de design/implementação do código (baseada em códigos), como a

verificação de casos de uso (baseada em requisitos) ou a execução de todas as linhas de código (baseada em códigos).

Se a cobertura baseada em códigos for aplicada, as estratégias de teste serão formuladas em termos da quantidade do código-fonte que foi executada pelos testes. Esse tipo de estratégia de cobertura de teste é muito importante para sistemas de segurança crítica.

No processo de controle do GGAS, será utilizada apenas a métrica de cobertura de código. A cobertura de código é uma métrica meramente quantitativa e não deve ser utilizada para aferir a qualidade de um conjunto de testes, tem como objetivo apenas demonstrar, percentualmente, quanto do código foi efetivamente executado durante o teste. Deve ser levada em consideração para identificar pontos com nenhum (ou quase nenhum) teste executado.

Para aferir a métrica de cobertura do código, será utilizada a ferramenta **EclEmma** que é uma ferramenta que verifica o quanto seu código está sendo testado. Ao final da execução dos testes unitários, o EclEmma gera um relatório contendo todas as classes do projeto e a porcentagem das linhas de código que foram testadas.

Para maiores informações, ver:
http://www.wthreex.com/rup/portugues/process/workflow/test/co_keyme.htm
http://www.aquiejava.com/2012/11/verificar-cobertura-de-teste-com-plugin-do-maven.html
http://www.eclemma.org

2.5.1.2 - PMD, FindBugs e CheckStyle

As ferramentas PMD, FindBugs e CheckStyle vêm sendo largamente utilizadas por programadores em ambientes de codificação automatizados como suporte ao controle de código-fonte. Elas analisam estaticamente o código para prover informações a respeito de pontos de risco e pontos de melhoria que ele possua.

A prática de inspeção/auditoria/revisão de código-fonte auxilia na: 1) organização; 2) clareza; 3) facilidade de reúso e manutenção; 4) consistência, entre outros aspectos, do software em desenvolvimento. Ambas as técnicas, PMD, FindBugs e CheckStyle podem ser combinadas para obter maior cobertura do código na inspeção.

São atividades desempenhadas por elas:

PMD = Analisa o código-fonte à procura de problemas potenciais (bugs e warnings), código não utilizado, código de baixa qualidade, expressões complicadas ou ilegíveis e duplicidade de código. Exemplos:

- Variáveis e importações não utilizadas;
- Expressões muito longas;
- Uso do método equals() ao invés do sinal '==';
- Laços e declarações desnecessários.

FindBugs = Possui função similar ao PMD, no entanto, ele trabalha sobre bytecode, ao invés do código-fonte. Exemplos:

- uso inadequado dos métodos .equals() e .hashCode();
- casts inseguros/impróprios;
- Nulidade de variáveis;
- Possíveis estouros de memória;
- Possíveis exceções ignoradas.

CheckStyle = Analisa o estilo e convenções do código-fonte. Não analisa erros e possibilidade de erros no código, como o PMD e FindBugs, mas verifica a conformidade dele em relação aos padrões estabelecidos (documentação, comentários, sintaxe). Exemplos:

- JavaDoc ausente/impróprio;
- Espaços em branco;

- Ausência de chaves e parênteses nas variáveis e condições;
- Extensão da expressão;
- Outras convenções a respeito de nomenclatura.

Para maiores informações, ver:
http://www.qualidadedesoftware.com.br/7185/pmd-findbugs-e-checkstyle

2.5.2 - Auditoria Manual

As auditorias manuais, dentro do processo de qualidade e controle do GGAS, serão realizadas através de técnicas de inspeção de código.

Inspeções representam um tipo de revisões formais, as quais são técnicas de análise para avaliação de forma, estrutura e conteúdo de um documento, código fonte ou outro produto de trabalho. Em outras palavras, inspeção é um método formal de revisão, onde um grupo de pessoas, incluindo o autor, se reúne para examinar um determinado produto de trabalho.

O foco da inspeção é a identificação de defeitos, com base em preparação prévia dos participantes. Vale salientar que métricas devem ser coletadas e utilizadas para determinar critérios de entrada para a reunião de inspeção, assim como para serem consideradas no processo de melhoria.

A inspeção de software pode ser utilizada em especificações, arquiteturas, projetos, código, procedimentos de teste e outros artefatos. O processo de inspeção de software é geralmente considerado umas das *best practices* em engenharia de software.

2.5.2.1 - Técnica de Revisão Formal (TRF)

A Técnica de Revisão Formal (TRF), criada por FAGAN em 1976, é o processo mais conhecido e utilizado dentre as técnicas de inspeção de software. Ela descreve uma forma detalhada de se realizar uma revisão. Neste processo, existem seis atividades principais:

- Planejamento: Um usuário, desempenhando o papel de moderador da inspeção, define o contexto da inspeção (descrição da inspeção, técnica a ser utilizada na detecção de defeitos, documento a ser inspecionado, autor do documento, entre outros), seleciona os inspetores e distribui o material a ser inspecionado.

- Apresentação: Os autores dos artefatos a serem inspecionados apresentam as características destes. Esta fase pode ser omitida se os inspetores possuem conhecimento sobre o projeto e os artefatos que devem ser inspecionados.

- Preparação: Os inspetores estudam os artefatos individualmente, e eventualmente fazem anotações sobre estes produzindo uma lista de discrepâncias. O fornecimento de técnicas de leitura pode facilitar a execução desta tarefa.

- Reunião: Uma reunião em equipe ocorre, envolvendo o moderador, os inspetores e os autores do documento. Discrepâncias são discutidas, e classificadas como defeito ou falso positivos. A decisão final sobre a classificação de uma discrepância sendo discutida é do moderador. A solução dos defeitos não é discutida durante a reunião, que não deve exceder duas horas, uma vez que após este tempo a concentração e a capacidade de análise dos inspetores costuma reduzir drasticamente. No caso em que uma reunião precisar de mais de duas horas, é sugerido que o trabalho de inspeção continue no próximo dia.

- Retrabalho: O autor corrige os defeitos encontrados pelos inspetores e confirmados pelo moderador.

- Continuação: O material corrigido pelos autores é repassado para o moderador, que faz uma análise da inspeção como um todo e reavalia a qualidade do artefato inspecionado. Ele tem a liberdade de decidir se uma nova inspeção deve ocorrer ou não.

Para o processo de inspeção manual de código do GGAS, será utilizada a TRF de Fagan, com algumas modificações.

Para maiores informações, ver:
http://www.devmedia.com.br/artigo-engenharia-de-software-introducao-a-inspecao-de-software/8037

3 - SOLUÇÃO PROPOSTA

Analisando resumidamente o cenário em 2014, tem-se o formato no qual inúmeras distribuidoras firmaram contratos com um fornecedor para a prestação de serviços de implantação, manutenção evolutiva e manutenção corretiva do sistema GGAS.

Após essas implementações, é feita a unificação dos códigos-fonte em uma única versão para posterior publicação dessa versão no Portal de Software Público Brasileiro para que as outras distribuidoras possam baixar as novas versões e utilizar o sistema, conforme ilustrado a título de exemplo na Figura 2.

Figura 2 – Cenário atual de desenvolvimento do sistema GGAS (imagem meramente ilustrativa).

Em função do cenário onde empresas utilizadoras do sistema GGAS (CDLs), estando essas em diferentes estágios de maturidade no processo de implantação e manutenção deste sistema, e ainda diante da possibilidade de outros *players* do segmento de desenvolvimento de sistemas (fornecedores) serem indicados, a partir

dos processos de contratação que avizinham, como sendo vencedores destes, tem-se um ambiente extremamente propício para que a unidade de código-fonte, padronização de processos e manutenção da base de dados deste sistema mantidos até então seja perdido.

Dentro do contexto apresentado na análise do cenário atual, pode-se levantar os seguintes questionamentos:

1. Quem audita a qualidade do código produzido pelos fornecedores?
2. Quem controla a evolução de um único produto desenvolvido por vários fornecedores simultaneamente?
3. Qual é a periodicidade para disponibilização de versões no Portal de Software Público?
4. Qual o processo necessário para submeter códigos do GGAS para a versão única do produto?
5. Como as LDCs podem contribuir no desenvolvimento de alguma Funcionalidade e/ou Bug?

Motivado pelos questionamentos acima, as distribuidoras, através da implementação de um ambiente centralizado de desenvolvimento distribuído, denominado de "Ambiente de Controle de Qualidade e Processo de Auditoria" (disponível em ggas.com.br), visa alcançar, de forma mais efetiva do que o vem sido praticado atualmente, o controle de versionamento, a auditoria e a inspeção de código-fonte do sistema de faturamento denominado GGAS, hospedado no Portal do Software Público, visando assim, aumentar o desempenho/performance no processamento das rotinas do sistema, a atualização tecnológica dos componentes utilizados por este software, bem como contribuir para o processo e disseminação do conhecimento das tecnologias utilizadas no seu desenvolvimento e de outras relacionadas no processo de desenvolvimento distribuído de sistemas/versões.

Figura 3 – Inclusão da empresa mantenedora do processo de desenvolvimento do GGAS.

A partir dessa implementação pode-se verificar que o projeto de desenvolvimento do GGAS será mantido em um ambiente gerenciado e centralizado de desenvolvimento distribuído de sistemas, podendo, a partir desse, efetuar todos os controles já mencionados anteriormente.

Visando a melhoria contínua dos serviços e eficiência atualmente implementados no GGAS, será estabelecido um processo completo e bem definido para dar suporte a evolução funcional do sistema.

3.1 - O processo de Testes e Auditoria do Sistema GGAS

Através do processo de controle de versão implementado nesse ambiente, as CDLs continuarão firmando contratos de implantação, manutenção evolutiva e corretiva com os diferentes fornecedores, porém, ao invés dos fornecedores trabalharem isoladamente e cada um evoluindo uma versão especifica do sistema GGAS, o que acabaria por ter-se posteriormente diferentes versões disponíveis no portal do software público e, consequentemente, gerando custos duplicados para as CDLs uma vez que

as mesmas poderão estar custeando o desenvolvimento de uma funcionalidade já desenvolvida por outra CDL com outro fornecedor em uma outra versão.

Assim sendo, o desenvolvimento de todos os fornecedores deverá, quando concluído, ser submetido na infraestrutura do Git central do GGAS, mantido no endereço ggas.com.br, e cada submissão passará por um processo de controle de qualidade e auditoria.

Caso todas as etapas do processo sejam concluídas com êxito, as implementações feitas por esses fornecedores serão promovidas para a versão única do sistema e esta versão será disponibilizada no portal de software público para acesso das demais CDLs, conforme pode ser visualizado na Figura 4.

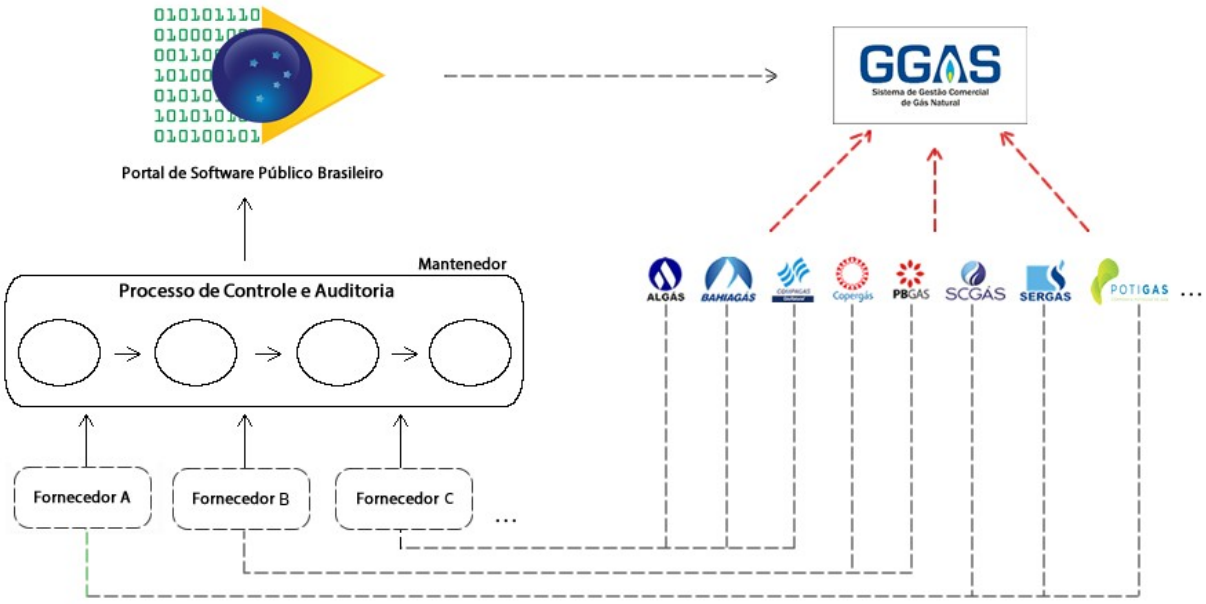


Figura 4 – Processo de Controle e Auditoria Proposto para o Sistema GGAS.

Sendo assim, o processo de testes e auditoria estabelecido para a evolução funcional do sistema GGAS seguira o estabelecido na Figura 5.

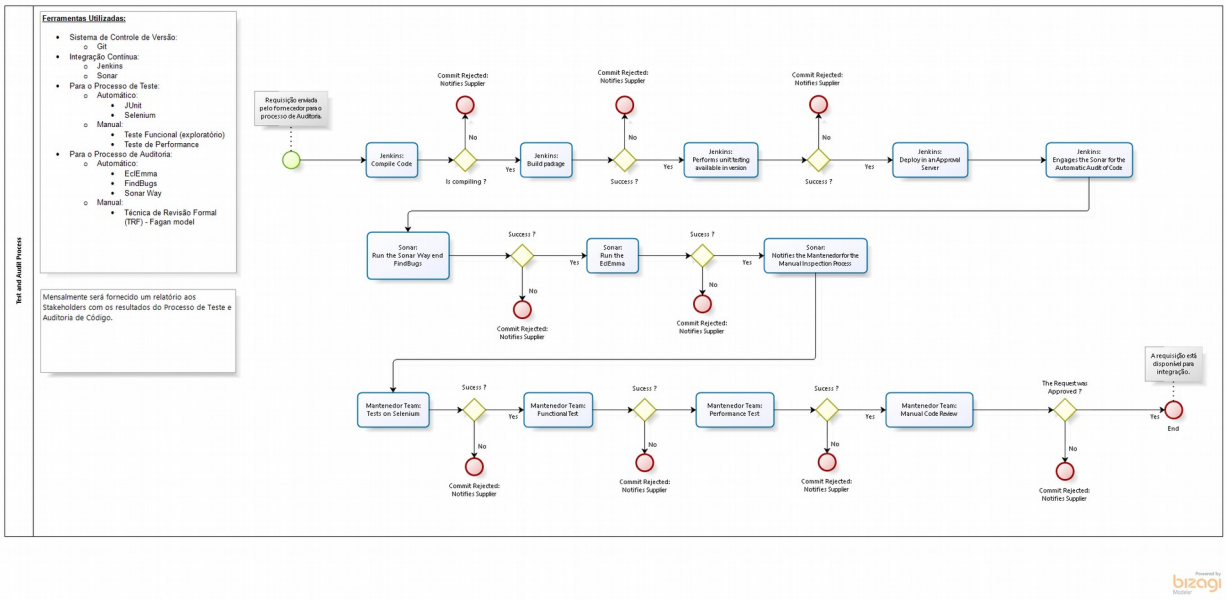


Figura 5 – Workflow do processo de Controle e Auditoria do sistema GGAS.

De acordo com a figura 5, as ferramentas utilizadas no processo são (*apresentadas na sessão 2 – Alinhamento de Conceitos*):

- Sistemas de Controle de Versão
  - Git (v. 7.5.1);
- Integração Contínua
  - Jenkins (v. 1.596);
  - Sonar (v. 5.0.0);
- Para o processo de testes:
  - Junit (v. 4.11);
  - Selenium (v. 2.8.0);
  - Testes Exploratórios;
  - Testes de Performance;
- Para o processo de auditoria de código:
  - EclEmma (v. 2.3.2);
  - Sonar Way (v. 5.0.0);

- FindBugs (v. 3.0.0);
- Técnica de Revisão Formal (TRF);

Os arquivos (incluindo código fonte) do sistema GGAS serão armazenados num sistema de controle de versão remoto Git, disponível em: http://www.ggas.com.br.

O processo será executado a cada submissão de código ao servidor do Git e passará pelas seguintes etapas:

1. Jenkins: Compila o código fonte;
2. Jenkins: Constrói o pacote de trabalho;
3. Jenkins: Executa os testes unitários disponíveis na versão;
4. Jenkins: Implanta em um servidor de homologação;
5. Jenkins: Aciona o sonar para a auditoria automática de código;
6. Sonar: Executa a ferramenta de inspeção automática com o *profile* Sonar Way e o FindBugs;
7. Sonar: Executa a ferramenta EclEmma para aferir os testes automáticos;
8. Sonar: Notifica a finalização dos testes automatizados e notifica a equipe de análise de conformidade de código fonte para início do processo de inspeção manual de código;
9. Equipe de Conformidade: Executa os testes do Selenium;
10. Equipe de Conformidade: Executa os testes funcionais;
11. Equipe de Conformidade: Executa os testes de performance;
12. Equipe de Conformidade: Executa o processo de auditoria manual do código;

Caso alguma das etapas descritas até o passo 7 indicadas acima falhem, a submissão será rejeitada e o fornecedor que está enviado as implementações será notificado para que ele promova as correções.

Caso todas as etapas descritas acima executem com sucesso, a submissão será promovida a versão principal do sistema e disponibilizada na próxima versão do GGAS.

Vale salientar que as etapas 9, 10, 11 e 12 do processo descrito acima, podem não ser executas em uma determinada de submissão, ou seja, essas etapas são opcionais dentro do processo estabelecido.

3.2 - Gerenciamento e Acompanhamento do Processo

Para gerenciamento do fluxo estabelecido pelo processo de evolução funcional do sistema GGAS, será utilizada a ferramenta Redmine.

Redmine é um software livre e de código aberto, licenciado sob os termos da GNU *General Public License v2* (GPL). Foi desenvolvido na linguagem Ruby utilizando framework *Ruby on Rails*. É uma ferramenta multiplataforma que suporta vários bancos de dados, extensões de plugins e sistema de controle de versão. Abaixo segue a relação das fortes características dessa ferramenta:

- Vários projetos de apoio;
- Controle de acesso baseado em papel flexível (Controle de acesso);
- Flexibilidade no sistema de monitoramento;
- Gráfico e calendários;
- Gerenciamento de notícias, arquivos e documentos;
- Fórum, wiki do projeto;
- Gerenciamento de tempo (projetos e usuário);
- Integração ao sistema de controle de versões (svn, git, cvs);
- Suporte a autenticação LDAP;
- Suporte a multilinguagem;
- Vários bancos de dados.

Para mais informações sobre o Redmine, recomenda-se o estudo do guia oficial através do link: http://www.redmine.org/guide.

Para acessar a ferramenta Redmine utilizada para controlar o projeto GGAS, deve-se acessar o endereço: http://redmine.ggas.com.br; realizar o cadastro da empresa no link Cadastre-se, e aguardar a ativação do seu usuário pela Equipe de Conformidade.

As requisições serão cadastradas no menu Nova Tarefa do Redmine, e seguirão os seguintes estados dentro do processo de controle e auditoria do código:

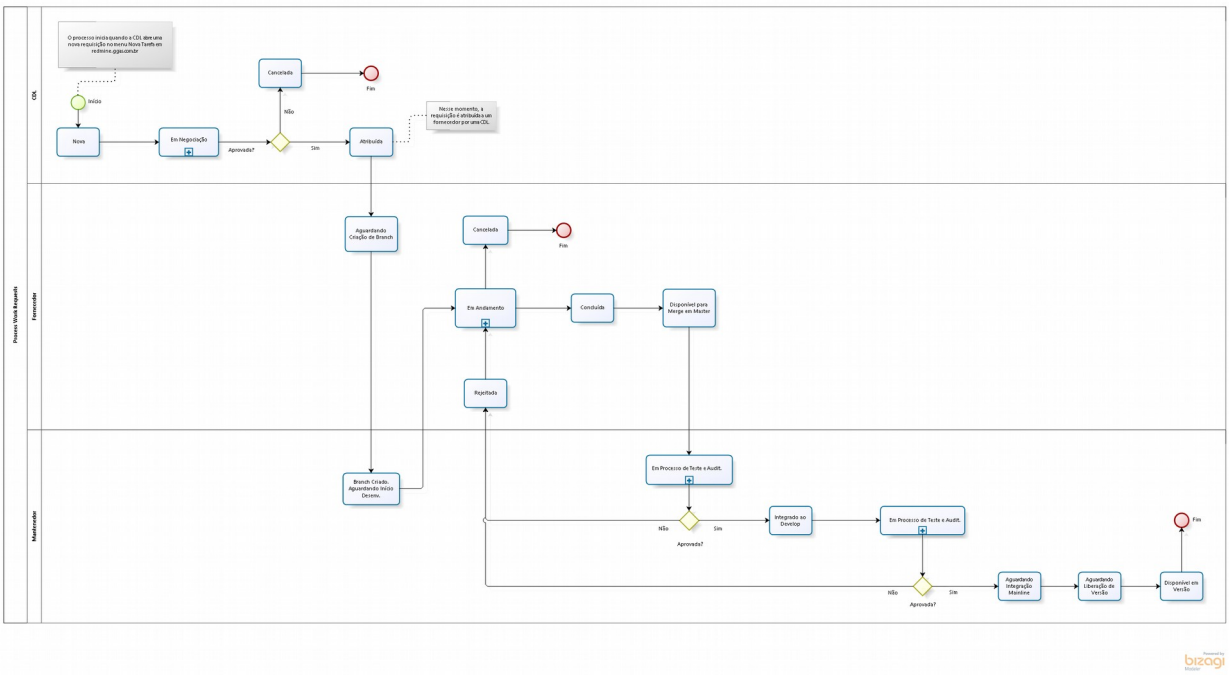


Figura 6 – Estados das requisições no processo de Controle e Auditoria do sistema GGAS.

De acordo com a Figura 6, os usuários da ferramenta poderão pertencer a três categorias de perfis de usuário:

- **CDL:** Esse perfil de acesso será dado aos usuários pertencentes as Companhias Distribuidoras Locais (CDLs – Bahiagás, PBGás, Alǵas, e etc.). São estes usuários que iniciarão o processo ao criar uma tarefa na ferramenta. Essa tarefa pode representar qualquer necessidade de implementação no sistema GGAS e pode variar desde a solicitação de implementação de um simples botão, à implementação de um novo módulo. A granularidade de cada tarefa, será definido pela própria CDL e conjunto com o fornecedor. Salienta-se que, toda e qualquer entrega de código por parte dos fornecedores, deverão estar vinculadas a pelo menos uma tarefa no sistema Redmine. Além de criar tarefas, esse perfil de acesso é o único que poderá modificar a tarefa para os seguintes status: Nova, Em Negociação, Cancelada e Atribuída. Este último status, é quando uma CDL conclui o processo de negociação (custo, tempo,

escopo e etc) com o fornecedor e atribui a atividade ao mesmo, a partir disso, apenas usuários com o perfil de acesso Fornecedor, poderão modificar a tarefa.

- **Fornecedor:** Esse perfil de acesso será dado aos usuários pertencentes aos fornecedores que tiverem contratos de implantação e/ou manutenção evolutiva e corretiva do sistema GGAS, para com as CDLs. Estes usuários receberão as tarefas atribuídas pelas CDLs e poderão modificar o status para Aguardando Criação de Branch, na qual o fornecedor indicará para o mantenedor que está aguardando a criação de um novo branch de trabalho; Em Andamento, nesse momento cada fornecedor trabalhará dentro do seu processo específico de desenvolvimento e poderá modificar o status da tarefa para Concluída ou Cancelada. Para o caso das tarefas Concluídas, os fornecedores poderão colocá-las no status Disponível para Merge em Master e isso indicará que eles mesclaram o código produzido na Branch Master do fornecedor e este código está disponível para avaliação da qualidade no processo de testes e auditoria realizado pelos usuários com perfil Mantenedor.

- **Mantenedor:** Esse perfil de acesso será dado aos usuários pertencentes à Equipe de Conformidade, que serão responsáveis por iniciar, após receberem as submissões de código enviada pelos fornecedores, o processo de testes e auditoria de código. O processo de testes e auditoria seguira o descrito na Figura 5 e, caso todas as etapas sejam concluídas com sucesso, os status das tarefas poderão ser modificados para: Integrado ao Develop; Aguardando Integração na Mainline; Integrada na MainLine; Aguardando Liberação de Versão e Disponível em Versão.

As tarefas cadastradas no Redmine, poderão estar, em um dado momento, nos status descritos abaixo:

- **Nova:** Indica que a tarefa foi criada pelo usuário de perfil CDL e que ainda não iniciou o processo de negociação com o fornecedor. Desta forma, todo o *backlog* do sistema que ainda não tenha iniciado o desenvolvimento, estará com esse status.

- **Em Negociação:** Indica que deu-se inicio ao processo de negociação para desenvolvimento da demanda especificada na tarefa. Essa negociação, em alguns fornecedores, envolve atividades como reuniões para definição e detalhamento do escopo, estimativas de prazos e custos, entre outras.

- **Cancelada:** Indica que a demanda especificada pela tarefa não será mais atendida, ou seja, ela está cancelada. A tarefa poderá ir para esse status em dois momentos, ou após a conclusão do status Em Negociação, quando uma CDL não aprova o orçamento estabelecido pela negociação da tarefa, ou após o início do desenvolvimento da tarefa pelo Fornecedor (status Em Andamento), em virtude de algum problema técnico e devidamente alinhado com a CDL proprietária da tarefa.

- **Atribuída**: Indica que a negociação para desenvolvimento da tarefa foi concluída e o fornecedor poderá iniciar o desenvolvimento da mesma.

- **Aguardando Criação de Branch:** Indica que o Fornecedor está aguardando que o Mantenedor crie uma Branch para o Fornecedor começar o desenvolvimento do escopo indicado na Requisição;

- **Branch Criado. Aguardando Início do Desenvolvimento:** Indica que o Mantenedor já criou a Branch de trabalho e que o fornecedor pode iniciar o desenvolvimento da Requisição.

- **Em Andamento:** Indica que o fornecedor iniciou o desenvolvimento da tarefa.

- **Concluída:** Indica que o fornecedor concluiu o desenvolvimento da tarefa, porém, ainda não disponibilizou a mesma para o processo de auditoria e controle de qualidade. Nesse status a tarefa ainda está sob responsabilidade do fornecedor e a mesma está aguardando algum processo interno para disponibilização ao mantenedor.

- **Disponível para Merge:** Indica que o fornecedor concluiu a tarefa e submeteu para o servidor Git remoto para ser avaliada no processo de controle de qualidade e auditoria do código desenvolvido.

- **Em Processo de Testes e Auditoria:** Indica que a tarefa está passando por todas as etapas do processo de controle de qualidade e auditoria do código, demonstradas na Figura 5.

- **Rejeitada:** Indica que tarefa não passou por alguma etapa do controle de qualidade e auditoria e está sendo devolvida ao fornecedor para as devidas correções.

- **Integrado ao Develop:** Indica que o processo de testes e auditoria do código foi executado com sucesso no Branch Master do Fornecedor e que o código foi mesclado ao Branch Develop para dar início aos testes de integração.

- **Aguardando Integração na Mainline:** Indica que a tarefa foi aprovada em todas as etapas do processo de controle de qualidade e auditoria do código e está aguardando a Equipe de Conformidade realizar o *merge* a versão principal do sistema.

- **Aguardando Liberação de Versão:** Indica que a tarefa foi integrada a versão principal do sistema e está aguardando a próxima liberação de versão por parte

da Equipe de Conformidade.

- **Disponível em Versão:** Indica que a tarefa está disponível na versão para distribuição através do Portal de Software Público Brasileiro.

Vale ressaltar que o Redmine não substituirá a ferramenta de *bug tracking*[1] dos fornecedores, ou seja, os bugs encontrados pelas CDLs continuarão sendo cadastrados nas ferramentas de cada fornecedor para controle dos seus respectivos SLAs estabelecidos em cada contrato. O que deverá ocorrer é que, como todas as requisições para serem incorporadas a versão principal do sistema, devem estar necessariamente vinculadas a uma tarefa no Redmine, consequente esta tarefa deverá ser criada no Redmine para passar pelo processo de auditoria e controle de qualidade como qualquer requisição normal.

3.3 - Regra para Aceitação das Submissões

Para que o código produzido pelo Fornecedor/CDL seja aceito, e consequentemente mesclado para o *Branch* Develop pelo mantenedor, ele deverá antes passar por todas as etapas descritas na Figura 5. Desta forma, os critérios para aceitação do código seguirá a regra abaixo:

1 O Jenkins deve conseguir compilar o código fonte com sucesso;

2 O Jenkins deve conseguir construir o pacote de trabalho com sucesso;

3 O Jenkins deve executar os testes unitários disponíveis na versão e TODOS os testes devem retornar sucesso em sua execução;

4 O Sonar deve executar a ferramenta de inspeção automática do código com o *profile* Sonar Way e FindBugs e TODOS os níveis de não conformidades NÃO

1 Sistema para registro e controle de ocorrências de bugs. Ver: https://qualidadebr.wordpress.com/2008/12/31/ferramentas-bug-tracking/

poderão, em hipótese alguma, subir as quantidades anteriormente indicadas no Sonar. Os níveis de não conformidade configurados no Sonar são:

- Blocker
- Critical
- Major
- Minor
- Info

Por exemplo, se a quantidade de não conformidades do tipo Critical for 445, em um dado momento do projeto, e após a submissão do fornecedor ela aumentar para 446 (ou qualquer número superior) o código será classificado com rejeitado, ou seja, ele não atendeu ao critério 4 deste conjunto de regras e o fornecedor deverá corrigi-lo até que ele seja menor ou igual a 445.

5 O Sonar deve executar a ferramenta EclEmma para aferir a cobertura dos testes automáticos e, se o percentual de cobertura do código for igual ou inferior ao percentual anteriormente indicado no Sonar, o código será classificado com rejeitado, ou seja, ele não atendeu ao critério 5 deste conjunto de regras e o fornecedor deverá corrigi-lo até que ele seja obrigatoriamente superior ao percentual de testes automáticos anteriormente indicado na ferramenta.

6 A Equipe de Conformidade do Mantenedor, deve executar uma bateria de testes realizando testes automatizados no Selenium, testes funcionais exploratórios e testes de performance. Caso algum Bug crítico seja encontrado e a Equipe de Conformidade considerar este suficientemente relevante, esta Equipe deverá registrar uma nota técnica na própria tarefa de requisição da ferramenta Redmine, e a CDL proprietária da tarefa, será convidada a decidir pela rejeição ou não da submissão. Caso a CDL opte pela não aprovação do código, o mesmo será classificado com rejeitado, ou seja, ele não atendeu ao critério 6

deste conjunto de regras e o fornecedor deverá corrigi-lo para que o código possa ser mesclado ao *Branch* Develop.

7 A Equipe de Conformidade do Mantenedor, deve realizar o processo de inspeção manual do código e NÃO deve encontrar não conformidades no código submetido pelo Fornecedor/CDL. Caso a não conformidade seja encontrada, e a Equipe de Conformidade considerar esta suficientemente relevante, esta Equipe deverá registrar uma nota técnica na própria tarefa de requisição da ferramenta Redmine, e a CDL proprietária da tarefa, será convidada a decidir pela rejeição ou não da submissão. Caso a CDL opte pela não aprovação do código, o mesmo será classificado com rejeitado, ou seja, ele não atendeu ao critério 7 deste conjunto de regras e o fornecedor deverá corrigi-lo para que o código possa ser mesclado ao *Branch* Develop.

Caso todas as 7 regras descritas anteriormente passem com sucesso, o código será classificado como apto para ser mesclado ao Branch Develop pelo Mantenedor.

4 - RESULTADOS ESPERADOS

Com a implementação do processo proposto, espera-se obter os seguintes resultados:

- Melhorar a qualidade e, consequentemente, a confiabilidade das versões disponibilizadas pelos fornecedores contratados pelas CDLs;
- Proporcionar um repositório centralizado com o código fonte do projeto para que diferentes fornecedores e CDLs, posam ter acesso e contribuir para a evolução do sistema;
- Evitar que, quando da existência de mais de um fornecedor, sejam criadas várias versões do sistema GGAS. A existência de diferentes versões do GGAS, poderia trazer os seguintes problemas:
  - Perca da padronização de processos e boas práticas entre as CDLs, uma vez que as mesmas começariam a trabalhar de forma isolada com seus respectivos fornecedores;
  - Perca da uniformização da base de dados, o que tornaria mais complicado um processo futuro de extração de informações estratégicas por parte dos acionistas;
  - Não aproveitar a oportunidade de interação e troca de conhecimentos entre as CDLs;
  - Possibilidade de uma CDL pagar por uma funcionalidade na versão x, sendo que esta já está desenvolvida na versão y, ocasionando custos duplicados para companhias do mesmo grupo.
- Contribuir e orientar os fornecedores com boas práticas de engenharia de software, de forma a maximizar a produtividade e qualidade do código fonte produzido no sistema GGAS.

5 - RELATÓRIOS DE MÉTRICAS E DIVULGAÇÃO DE RESULTADOS

Qualidade é um conceito que se refere a atributos e características positivas, diz respeito ao grau de perfeição e de conformidade a certo padrão. A percepção da qualidade depende de como ela é observada por um indivíduo, que usa como base diversos indicadores para medi-la, como por exemplo, quesitos como durabilidade e utilidade. Apesar da subjetividade deste conceito, em se tratando de software, existem vários atributos e indicadores aceitos universalmente como sendo características de um bom software e de um código fonte bem escrito.

Conceitualmente falando, qualidade de software e de código são duas disciplinas bem diferentes, cada uma delas está relacionada a diferentes aspectos e indicadores qualitativos do produto de software.

A qualidade de software está em um nível mais elevado, faz parte da visão dos usuários e gestores, é o que eles vêm e obtêm a partir da utilização de um software. Alguns indicadores utilizados para medir a qualidade de um software incluem usabilidade e confiabilidade.

Já a qualidade de código está em um nível mais baixo, faz parte da visão dos desenvolvedores, engenheiros, arquitetos e, em alguns casos, analistas e gerentes. Os indicadores da qualidade de código incluem, por exemplo, complexidade do código, duplicações de código, tamanho do código, entre outros.

De forma a possibilitar a mensuração de algumas das métricas mais utilizadas para medir o nível de adequação do código fonte a determinadas características de qualidade, será utilizado por este processo a ferramenta Sonar que mapeia estes conceitos e métricas em uma plataforma de monitoramento de qualidade.

Quando um código que não possua muitas características positivas – como baixa complexidade, pouca duplicação de código e alta cobertura de testes unitários – diz-se que ele tem uma elevada dívida técnica (Technical Debt). Dívida técnica é uma metáfora que se refere às consequências de uma arquitetura de software de baixa qualidade, o esforço necessário para resolver problemas provenientes dessa arquitetura e à fragilidade do código em relação a diversas características técnicas, como duplicações e complexidade.

A prática de gerenciar os débitos de um projeto vem ganhando espaço, especialmente em processos ágeis de desenvolvimento, e está associada, intrinsecamente, à qualidade da arquitetura do software. Para medir a dívida técnica, geralmente é utilizada alguma unidade de tempo, representando assim o tempo necessário para resolver os débitos do projeto. Ultimamente, muito tem se discutido sobre a importância de gerenciar os débitos de um projeto. A preocupação com a dívida técnica está relacionada ao fato que quanto maior o débito de um projeto, menor é a produtividade da equipe de desenvolvimento e, consequentemente, maiores serão os gastos financeiros. Um dos principais motivadores para melhorar a qualidade do código e diminuir os débitos do projeto é exatamente o fato da dívida técnica ter impacto direto nas questões financeiras. É daí, também, que vem o nome desse termo.

Para diminuir a dívida técnica durante o desenvolvimento e aumentar a qualidade do código, há muitas abordagens indicadas. Entre elas estão a programação em pares, a refatoração e o desenvolvimento guiado por testes (TDD). Essas técnicas são úteis quando utilizadas desde o início do projeto, fazendo com que o mesmo seja desenvolvido dentro de padrões aceitáveis de qualidade e com uma dívida técnica baixa. Para códigos já existentes, a solução é a refatoração. Mas antes de varrer o código aplicando diversas técnicas a fim de melhorar a estrutura do mesmo, é preciso entender e medir o código.

Nesse contexto, as métricas extraídas por este processo seguirá um período de coleta inicialmente de 3 meses, podendo este período ser modificado para mais ou para menos no futuro e devidamente informado a todos os *stakeholders*. A cada período de coleta, será disponibilizado na página: http://redmine.ggas.com.br/projects/ggas-evolucao-sistema/documents ; um conjunto de relatórios com métricas coletadas durante processo de controle de qualidade e auditoria do código.

Desta forma, serão disponibilizadas as seguintes informações:

- Informações extraídas do Redmine e/ou Sonar:
  - **Total de requisições abertas por CDL**. Exemplo:
    - Algás: 08
    - Compagas: 05
    - Bahiagás: 03
    - PBGás: 01
    - ...
  - **Total de requisições em cada estado.** Exemplo:
    - Nova: 25
    - Em Negociação: 04
    - Em Desenvolvimento: 10
    - Em Processo de Testes e Auditoria: 03
    - Rejeitada: 05
    - …
  - **Total de requisições atribuídas no momento.** Exemplo:
    - Em CDLs: 35
    - Em Fornecedores: 18
    - Em Mantenedor: 6
  - **Qualidade de submissões de fornecedores por período de coleta.** Por exemplo**:**
    - Jan/15 – Mar/15:
      - Fornecedor X:

        - QTD Requisições: 20
        - QTD Rejeitadas: 12
        - QTD Aprovadas: 8
    - Fornecedor Y:
        - QTD Requisições: 11
        - QTD Rejeitadas: 2
        - QTD Aprovadas: 9
  - Abr/15 – Jun/15:
    - Fornecedor X:
        - QTD Requisições: 8
        - QTD Rejeitadas: 6
        - QTD Aprovadas: 2
    - Fornecedor Y:
        - QTD Requisições: 9
        - QTD Rejeitadas: 1
        - QTD Aprovadas: 8
  - Jul/15 - Set/15:
    - …
- **Quantidade de linhas de código produzidas por fornecedor no último período de coleta.** Exemplo:
  - Fornecedor x: 22.536 linhas de código
  - Fornecedor y: 7.229 linhas de código
  - ...
- **Percentual de código duplicado no projeto.** Por exemplo:
  - Jan/15 – Mar/15: 58%
  - Abr/15 – Jun/15: 38%
  - Jul/15 - Set/15: 45%
  - …

- **Percentual de cobertura dos testes unitários.** Por exemplo:
  - Jan/15 – Mar/15: 14%
  - Abr/15 – Jun/15: 20%
  - Jul/15 - Set/15: 25%
  - …
- **Número de não conformidades do código.** Por exemplo:
  - Jan/15 – Mar/15:
    - Severidade Muito alta: 0
    - Severidade Alta: 0
    - Severidade Média: 1450
    - Severidade Baixa: 8
    - Severidade Muito baixa: 140
  - Abr/15 – Jun/15:
    - Severidade Muito alta: 0
    - Severidade Alta: 0
    - Severidade Média: 1237
    - Severidade Baixa: 12
    - Severidade Muito baixa: 87
  - …

6 - FLUXO DE DESENVOLVIMENTO DO PROJETO

6.1 - Acesso ao Projeto

Para acessar o projeto GGAS, é necessário ter uma credencial de acesso (usuário e senha) na ferramenta Gitlab, disponível em: http://www.ggas.com.br. Essa conta será fornecida mediante a prévia solicitação do fornecedor ou CDL ao endereço de e-mail: contato@ggas.com.br.

A solicitação será analisada pelo mantenedor do sistema e, caso aprovada, serão enviadas as instruções de acesso ao Gitlab do sistema GGAS, bem como, a URL do repositório para que seja possível baixar sua respectiva *branch*, e assim poder desenvolver para o GGAS.

6.2 - Fluxo de Desenvolvimento do Projeto GGAS

Conforme dito em sessões anteriores, o processo de trabalho proposto utilizará o sistema de controle de versão Git. Mas antes de apresentar o fluxo, faz-se necessário um bom entendimento do conceito de *Branch*.

Quase todos os Sistemas de Controle de Versão têm alguma forma de suporte a ramificação (*branching*). Criar um *branch* significa que você vai clonar a última versão da linha principal de desenvolvimento e continuar a trabalhar em possíveis bugs e novas funcionalidades sem bagunçar a linha principal de produção que provavelmente está no ar e não pode receber falhas. Desta forma, pode-se dizer que um *branch* é uma cópia do projeto. Cada *branch* pode ser editado e evoluído independentemente em linhas paralelas a linha principal do projeto.

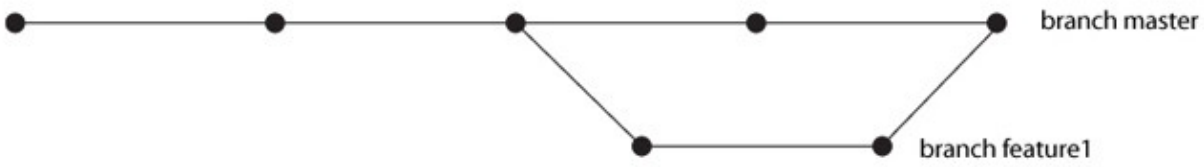


Figura 7 – Exemplo de um *Branch*.

Dentro do processo proposto, haverá um fluxo de trabalho baseado no desenvolvimento com várias linhas de desenvolvimento (*branches*) podendo o *branch* ser o Master principal (caixa verde da Figura 8), Develop (caixa vermelha), Master do Fornecedor/CDL (caixas em azul-claro), de Features do Fornecedor/CDL (círculos azul-escuro) ou de Bugfixes (círculos rosas), conforme pode ser visualizado na Figura 8.

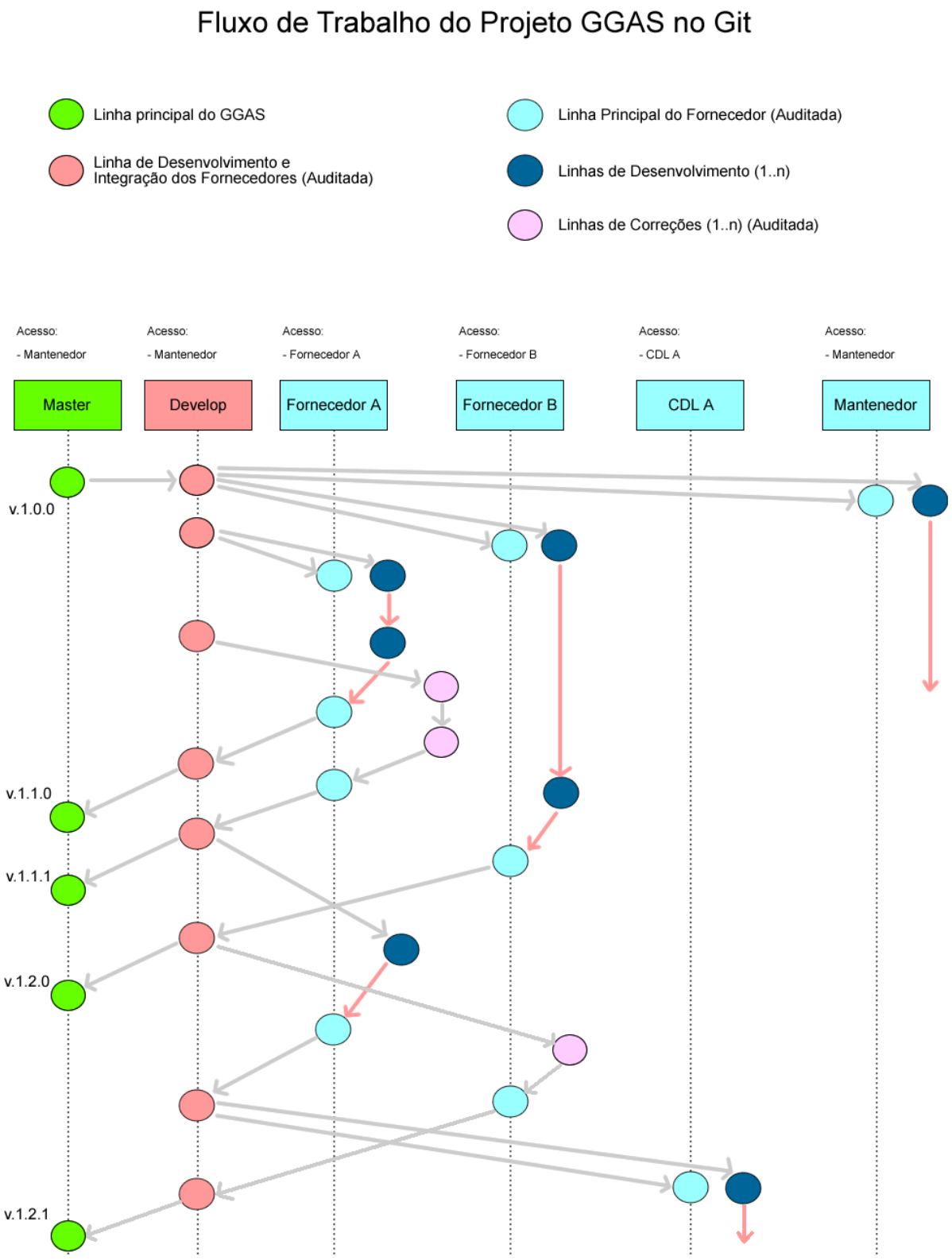


Figura 8 – Exemplo do Fluxo de Trabalho do GGAS no Git.

Neste fluxo, os *branches* Master e Develop são considerados históricos – isso porque eles guardarão a história do projeto. As tags de marcação de release são feitas no Master e o Develop serve como *branch* de integração para *branches* de features de fornecedores e/ou CDLs.

A versão disponibilizada na linha principal do projeto (Master) receberá o número de versão 1.0.0 e será evoluída conforme descrito na sessão 6.4 deste documento.

A Figura 8 mostra um exemplo hipotético de um determinado momento do ciclo de desenvolvimento do GGAS, na ilustração, é clonado o projeto na versão 1.0.0 do *branch* Master para o *Branch* Develop e depois para *Branch* do Fornecedor B. Isso significa que o Fornecedor B iniciará o desenvolvimento de alguma Requisição de trabalho de uma CDL. Desta forma, será disponibilizada o *Branch* para o Fornecedor B que poderá cloná-la para um repositório local nas estações de trabalho da sua própria empresa, o Fornecedor B trabalhará por algum tempo naquela Requisição e, durante esse tempo, ele poderá dar vários *commits* no seu repositório local. Quando a requisição estiver totalmente pronta, o fornecedor dará um Push (envia as modificações para o servidor) para o *Branch* do servidor remoto (representadas nas bolas azul escuras) e o fornecedor fará o merge para a sua Branch principal (representada na bola azul clara), assim que o *Branch* principal do fornecedor for atualizado, será dado automaticamente o *start* para o processo de auditoria automática do código e logo em seguida para todas as etapas previstas no processo de Controle e Auditoria do sistema GGAS (apresentado na sessão 3.1 e ilustrado na Figura 5). Caso o código avaliado no *branch* principal do Fornecedor B passe pelos critérios de qualidade estabelecidos no processo e seja aprovado, o Mantenedor será notificado e ele será mesclado no *Branch* Develop pelo Mantenedor.

O *branch* Develop será o *Branch* que conterá os merges de todos os fornecedores/CDLs antes de irem para a versão oficial do sistema, desta forma, quando chegar um código no *Branch* Develop proveniente do merge dos *Branches* dos fornecedores/CDLs, esse código também será auditado no processo de Controle e

Auditoria do sistema. Caso todas as etapas do processo de controle de qualidade passem com sucesso, o Mantenedor fará um merge para o Master e será gerada uma versão para a disponibilização no Portal de Software Público.

Em virtude do exposto, tem-se que o processo de Controle e Auditoria do sistema será executado em dois momentos dentro do ciclo de vida de uma requisição de trabalho. O primeiro momento será quando do merge do fornecedor para o seu respectivo *Branch* principal e o segundo momento será quando do merge do código do *Branch* do fornecedor para o *Branch* Develop que terá o código concluído de todos os fornecedores. Passados por esses dois momentos de auditoria, a requisição do fornecedor será incluída na versão disponibilizada no Master.

Outro ponto importante a ser explicado na figura, é o funcionamento do *Branch* Hotfix (representados pelas bolas rosa claro). Cada fornecedor terá nenhum ou vários *Branches* de Hotfix que serão criados pelo Mantenedor no momento que for aberta uma requisição de correção no Redmine. Ao concluir a correção, o fornecedor dará o *push* no código e, assim que o servidor receber as modificações, o processo de Controle e Auditoria do sistema será executado, ou seja, todos os *Branches* de Hotfix criados pelo Mantenedor para os fornecedores, também serão monitorados pelo processo de auditoria automática do código. Caso todas as etapas passem com sucesso, o fornecedor será notificado e ele poderá gerar uma versão emergencial e disponibilizá-la para seu respectivo cliente (CDL). Após entregar a versão emergencial para a CDL, o fornecedor deverá obrigatoriamente fazer um merge para o seu *Branch* principal. Quando o código chegar na linha principal, será rodado o processo de auditoria e o Mantenedor será notificado quando da conclusão com sucesso, este fará o merge para a *Branch* Develp e posteriormente para o Master, onde o código de correção realizado pelo fornecedor poderá ser disponibilizado na versão principal do sistema.

Em síntese, o processo de correção de bugs seguirá as seguintes etapas:

1. É aberto uma requisição de trabalho no Redmine para correção do(s) problema(s);
2. O mantenedor cria um *Branch* de Hotfix para o fornecedor com a última versão do código disponível no Portal de Software Público;
3. O Fornecedor atua e conclui as correções especificadas na requisição aberta;
4. O Fornecedor dá um *Push* no código do *Branch* de Hotfix;
5. O servidor reconhece a atualização do código do *Branch* de Hotfix e roda o processo de auditoria automática do código;
6. Após o sucesso do processo de auditoria, o Fornecedor é notificado e o mesmo gera uma versão emergencial a partir do *Branch* de Hotfix e disponibiliza a CDL;
7. O Fornecedor faz o merge do código do *Branch* de Hotfix para a seu *Branch* principal;
8. O servidor reconhece a atualização do código do *Branch* principal do fornecedor e roda o processo de auditoria automática do código;
9. Após o sucesso do processo de auditoria, o mantenedor será notificado e este fará o merge para o *Branch* Develop e posteriormente para o Master.

Vale salientar a importância do passo 7 no processo descrito acima, essa atividade é obrigatória por parte do fornecedor e caso o mesmo não faça o merge para o seu *Branch* principal, a correção ficará armazenada apenas no *Branch* de Hotfix, isso resultará na perca do código de correção quando o fornecedor atualizar a versão do sistema a partir do Portal de Software Público.

O objetivo deste fluxo de trabalho é possibilitar o desenvolvimento em paralelo de vários fornecedores simultaneamente através da utilização boas práticas de controle de versionamento de sistemas e de técnicas e ferramentas eficientes e consolidados na indústria de engenharia de software.

6.3 - Regra de Nomenclatura dos Branches

O projeto possuirá Branches fixos e Branches criados sobre demanda. Os Branches fixos são:

- GGAS
  - MASTER – utilizado pelo mantenedor.
  - DEVELOP – utilizado pelo mantenedor.
- MANTENEDOR
  - MASTER – linha principal do mantenedor.
- FORNECEDOR A
  - MASTER – linha principal do fornecedor A.
- FORNECEDOR B
  - MASTER – linha principal do fornecedor B.
- CDL A
  - MASTER – linha principal da CDL A.

Desta forma, cada ator terá no mínimo uma linha principal de desenvolvimento.

Os Branches sob demanda serão criados para atendimento de requisições de trabalho do Redmine (Tarefas cadastradas pelas CDLs), para atendimento de novas funcionalidades e/ou correção de Bugs. Esses Branches sob demanda seguirão o seguinte padrão de nomenclatura:

<nome fornecedor> _ <tipo do branch> _ <número da tarefa do redmine>.

O Branch poderá ser de dois tipos:

- DEV: para indicar branches de novas funcionalidades;
- HOTFIX: para indicar branches de correções;

Desta forma, a estrutura de Branches existentes no projeto poderá ser representada pelo exemplo abaixo:

- GGAS
  - MASTER – utilizado pelo mantenedor.
  - DEVELOP – utilizado pelo mantenedor.
- MANTENEDOR
  - MASTER – linha principal do mantenedor.
  - MANTENEDOR_DEV_7764
  - MANTENEDOR_DEV_8810
  - MANTENEDOR_HOTFIX_0909
- FORNECEDOR A
  - MASTER – linha principal do fornecedor A.
  - FORNECEDOR A_DEV_1122
  - FORNECEDOR A_DEV_1414
  - FORNECEDOR A_DEV_9851
- FORNECEDOR B
  - MASTER – linha principal do fornecedor B.
  - FORNECEDOR A_DEV_8712
  - FORNECEDOR A_HOTFIX_0010
  - FORNECEDOR A_HOTFIX_0011
  - FORNECEDOR A_HOTFIX_0801
- CDL A
  - MASTER – linha principal da CDL A.
  - CDL A_HOTFIX_7410

6.4 - Regra da Numeração do Versionamento do Sistema GGAS

A versão disponibilizada na linha principal do projeto (Master) receberá o número de versão 1.0.0 e evoluirá de acordo com a seguinte regra:

- O primeiro número, antes do primeiro ponto (**3**.8.2), representará a versão principal do sistema e será incrementado apenas no caso de alterações significativas. Por exemplo, quando da mudança da arquitetura interna do sistema, quando da mudança da interface gráfica, quando da inclusão de um novo módulo com representação significativa dentro do sistema e etc. Desta forma, se a versão do GGAS for, por exemplo, 3.8.2, pode se dizer que os sistema está na sua 3ª. Versão.

- O segundo número, após o primeiro ponto (3.**8**.2), representará a disponibilização de novas funcionalidades significativas ou não dentro da versão atual do sistema. Por exemplo, quando da inclusão de novas informações em alguma tela, ou a inclusão de alguma nova regra de negócio de algum processo interno. Desta forma, se a versão do GGAS for, por exemplo, 3.8.2, pode se dizer que já foram disponibilizadas 8 versões com novas funcionalidades para a 3ª. versão do sistema.

- O terceiro número, após o segundo ponto (3.8.**2**), representará a disponibilização de versões de correção para a versão atual do sistema. Por exemplo, quando da necessidade de se corrigir algum bug em uma tela, ou alguma regra de negócio implementada incorretamente. Desta forma, se a versão do GGAS for, por exemplo, 3.8.2, pode se dizer que já foram disponibilizadas 2 versões de correção para a versão 3.8 do sistema.

7 - REFERÊNCIAS

Primeiros passos - Noções Básicas de Git. Disponível em: http://git-scm.com/book/pt-br/v1/Primeiros-passos-No%C3%A7%C3%B5es-B%C3%A1sicas-de-Git.

Integração Contínua com Jenkins. Disponível em: https://getupcloud.com/blog/integracao-continua-com-jenkins.

JUnit. Disponível em: http://pt.wikipedia.org/wiki/Junit.

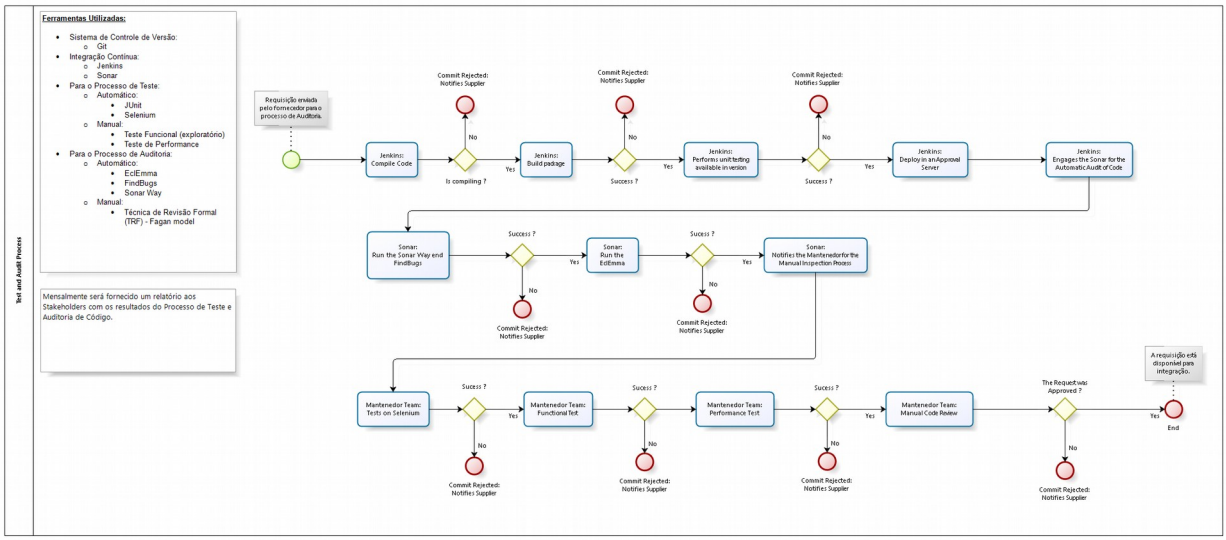
JUnit. Disponível em: http://junit.org.

Sonar Java: avaliando o código através de métricas. Disponível em: http://www.devmedia.com.br/sonar-java-avaliando-o-codigo-atraves-de-metricas/31278.

Quatro workflows para trabalhar com Git. Disponível em: http://imasters.com.br/desenvolvimento/quatro-workflows-para-trabalhar-com-git-melhores-2013 .

8 - ANEXOS

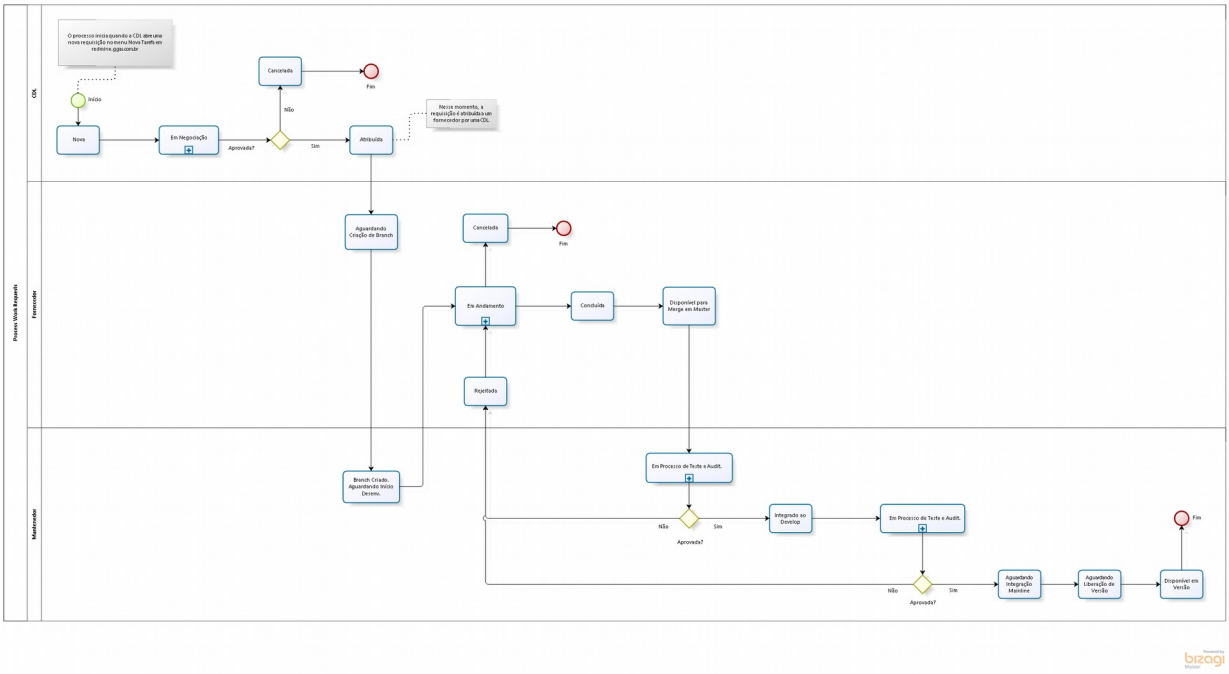
8.1 - Anexo I – Workflow do Processo de Controle e Auditoria do sistema GGAS.



Disponível em:

http://redmine.ggas.com.br/attachments/download/28/Desenho%20do%20Processo%20-%20Processo%20de%20Teste%20e%20Auditoria.png

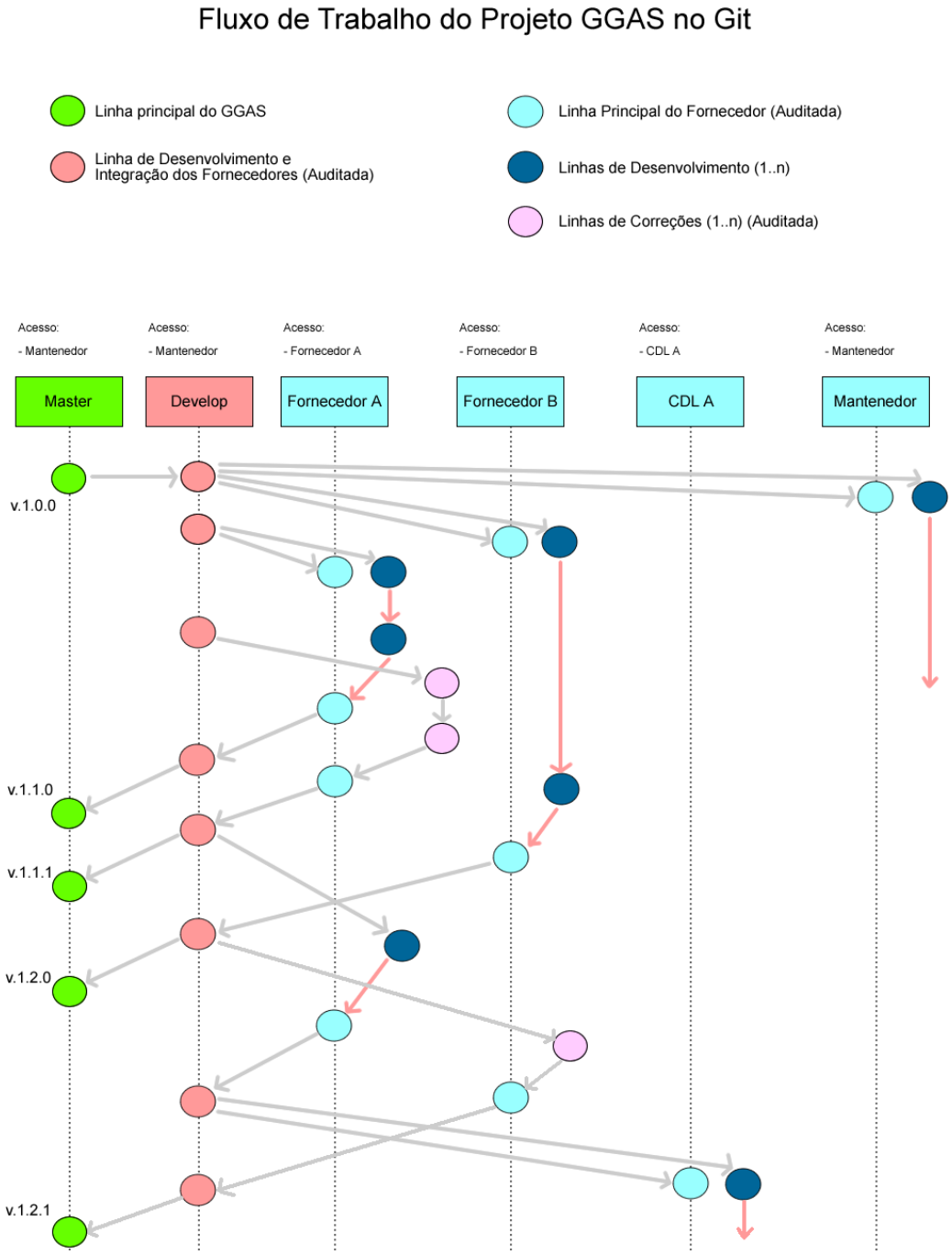
8.2 - Anexo II – Estados das Requisições no Processo de Controle e Auditoria do Sistema GGAS.



Disponível em:

http://redmine.ggas.com.br/attachments/download/30/Desenho%20do%20Processo%20-%20Estados%20da%20Requisi%C3%A7%C3%A3o.png

8.3 - Anexo III – Exemplo do Fluxo de Trabalho do Projeto GGAS no Git.



Disponível em:

http://redmine.ggas.com.br/attachments/download/29/Fluxo%20de%20Trabalho%20no%20Git.png

8.4 - Anexo IV – Regras e Critérios Utilizados nas Ferramentas de Auditoria.

Sonar Way:

- Formato de Instalação:
  - Fazer o download do Sonar em:
    - http://dist.sonar.codehaus.org/sonarqube-5.0.zip
  - Descompactar o arquivo, e executar o arquivo *sonar.bat* ou *sonar.sh* respectivamente para Window e Linux que se encontra na pasta *bin*;
  - Após o Sonar está devidamente instalado, é possível criar um profile de auditoria pela URL http://<servidor>:9000/profile. Obs. porta 9000 é a porta *default*, podendo ser configurada para uma outra qualquer. As regras de auditoria podem ser reproduzidas pelos links:
    - http://sonar.ggas.com.br/profiles/export?language=java&name=GGAS
    - http://sonar.ggas.com.br/profiles/export?format=findbugs&language=java&name=GGAS
  - Utilizando o eclipse, é possível instalar o *plugin* do sonar através do Eclipse Market Place. Desta forma, instale o plugin **SonarQube 3.4.0**.
  - Após o *plugin* instalado, o usuário deverá clicar no Projeto com o botão direito do mouse e acionar opção:
    - >> Configuração >> Sonar >> Analisys >> Remote
- Arquivo de regras:
  - http://sonar.ggas.com.br/profiles/export?language=java&name=GGAS

FindBugs:

- Formato de Instalação:
  - Instalar o plugin **FindBugs Eclipse Plugin 3.0.0** do eclipse, disponível no **Eclipse Marketplace**.
- Arquivo de regras:

- http://sonar.ggas.com.br/profiles/export?format=findbugs&language=java&name=GGAS

EclEmma:

- Formato de Instalação:
  - Instalar o plugin **EclEmma Java Code Coverage 2.3.2** do eclipse, disponível no **Eclipse Marketplace**.
- Arquivo de regras:
  - Não se aplica.