Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Processo de Desenvolvimento de Software Ágil com adição de práticas para melhoria da qualidade dos produtos finais

Pryscilla M. Dóra Selister

Campina Grande – PB
Dezembro de 2011.

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Processo de Desenvolvimento de Software Ágil com adição de práticas para melhoria da qualidade dos produtos finais

Pryscilla M. Dóra Selister

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande – Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Gerencia de Projetos

Orientadores

José Antão B. Moura
Jacques Philippe Sauvé

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

S465p Selister, Pryscilla Marcili Dóra.

Processo de desenvolvimento de software ágil com adição de práticas para melhoria da qualidade dos produtos finais / Pryscilla Marcili Dóra Selister. – Campina Grande, 2011.

118 f. : il.

Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.

Orientadores: Prof. Dr. José Antão Beltrão Moura e Prof. Dr. Jacques Philippe Sauvé.

Referências.

1. Processos de Desenvolvimento de Software. 2. Qualidade de Software. I. Título.

CDU 004.4(043)

"PROCESSO DE DESENVOLVIMENTO DE SOFTWARE ÁGIL COM ADIÇÃO DE PRÁTICAS PARA MELHORIA DA QUALIDADE DOS PRODUTOS FINAIS"

PRYSCILLA MARCILI DORA SELISTER

DISSERTAÇÃO APROVADA EM 12/12/2011

JOSÉ ANTÃO BELTRÃO MOURA, Ph.D
Orientador(a)

JACQUES PHILIPPE SAUVÉ, Ph.D
Orientador(a)

DALTON DARIO SEREY GUERRERO, D.Sc
Examinador(a)

AYLA DÉBORA DANTAS DE SOUZA REBOUÇAS, D.Sc
Examinador(a)

JOÃO PAULO FREITAS DE OLIVEIRA, M.Sc
Examinador(a)

CAMPINA GRANDE - PB

A minha cara metade, meu marido
Gustavo, que quando eu resolvi pular ele
esteve ao meu lado para pularmos juntos.
À minha mãe por toda a sua dedicação,
confiança e exemplo.
À minha filha, porque inúmeras foram as
vezes que busquei forças em ti e por ti
para seguir nesta jornada.

Resumo

As atividades realizadas há mais de 10 anos no LSD (Laboratório de Sistemas Distribuídos) apontam para uma consolidação de resultados na área de Sistemas Distribuídos, principalmente, nas pesquisas em Grades Computacionais, Computação nas nuvens, Sistemas *Peer-to-Peer* (P2P), Tolerância a Falhas, Desenvolvimento de Software Distribuído e Concorrente e, mais recentemente, Aplicações Industriais e Qualificação de Software.

Um dos principais projetos do laboratório é o *OurGrid*, uma grade aberta, *free-to-join*, que agrega serviços computacionais e dá suporte a uma gama de aplicações chamadas de *bag-of-tasks*. O OurGrid é um software livre e existe um número expressivo de instituições, dentro e fora do país, que usaram ou estão usando essa tecnologia.

Dentro do escopo do laboratório também desenvolveram-se outros produtos como o *OurBackup* [Oliveira, 2007], um sistema de *backup* P2P baseado em redes sociais, o JIC, protocolo de comunicação do *OurGrid*, entre outros. Em todos os casos, o LSD buscou problemas reais no contexto do desenvolvimento e utilização desses produtos de software por clientes externos ao laboratório para desenvolver sua pesquisa. Pela importância desses projetos e por ser uma das iniciativas em grades computacionais de grande sucesso no Brasil, o LSD tem tido a preocupação em oferecer produtos de qualidade para a comunidade de usuários de grades.

Em função do cenário do LSD, isto é, equipe composta, em sua maioria, por alunos de graduação em Ciência da Computação; histórico da universidade no desenvolvimento de software; experiência em elaboração de processo de desenvolvimento como XP1 [XP1, 2002] e *OurProcess* [*OurProcess*] e o tipo de software desenvolvido, optou-se por solucionar o problema de qualidade baixa nos produtos iniciando pela elaboração de um novo processo de desenvolvimento, isso porque a qualidade dos produtos desenvolvidos não estava nos níveis idealizados. Essa preocupação com a qualidade resultou no *OurQualityProcess* (OQP) um processo baseado na metodologia ágil XP [Beck, 2001], com a incorporação de práticas da metodologia não ágil e evitando a intrusividade excessiva no trabalho dos desenvolvedores.

Os resultados obtidos com o OQP mostraram fortes indícios de melhoria na qualidade dos produtos desenvolvidos através de comparações estatísticas utilizando a metodologia científica no estudo de alguns casos.

Palavras Chave: Processos de Desenvolvimento de Software, Qualidade de Software.

Abstract

The tasks which have been executed for more than 10 years on LSD (Distributed System Laboratory) show a consolidation of results in Distributed Systems area, mainly in research on Grids Computing, Cloud computing, Peer-to-Peer (P2P) Systems, Fault Tolerance, Distributed Systems Development and Concurrent Systems development and more recently, Industrial Applications and Software Qualification.

One of the main projects of the laboratory is the *OurGrid*, an open grid, free-to-join, which combines computational services and supports a range of applications called bag-of-tasks. *OurGrid* is free software and there are a significant number of institutions, inside and outside the country, which have used or are still using this technology.

In the laboratory scope other products, were also developed such as *OurBackup* [Oliveira, 2007], a P2P backup system based on social networks, the JIC, *OurGrid* communication protocol, among others. In all cases, LSD looked for real problems in the development context and use of these software products by external customers of the laboratory to develop its research. According to the importance of these projects and for it being one of the initiatives in the great Grid success in Brazil, LSD has been concerned about offering quality products to the community of grid users.

Due to the LSD scenario (team composed mostly by graduate students in Computer Science, university history in software development, experience in defining processes as XP1 [XP1, 2002] and *OurProcess* and the type of software developed, we chose to pursue quality in the products starting with the drafting of a new development process, which is why the quality of the products developed was less than what was expected. This concern for quality has resulted in *OurQualityProcess* (OQP), a process based on Agile XP [Beck, 2001], with the incorporation of practices from traditional methodology and avoiding excessive intrusiveness in the work of developers.

The results obtained with the OQP showed strong signs of improvement in the quality of the products developed through statistical comparisons using a scientific methodology in the study of some cases.

Keywords: Software Development Processes, Software Quality.

Agradecimentos

Esta sessão de agradecimentos é pequena para agradecer a todos que me ajudaram e me apoiaram durante a realização deste trabalho, mas o primeiro agradecimento é para o Mestre maior, Obrigada Meu Deus!

Agradeço a toda minha família em especial ao meu marido Gustavo pela compreensão da ausência e por todos os momentos que precisei do teu ombro para me apoiar; a minha mãe por sempre acreditar e mim e, principalmente, pelos ensinamentos não só com suas doces palavras, mas com a dignidade dos seus atos. A minha princesinha Maria Eduarda. Filha! Obrigada por existir na minha vida, és a inspiração e a força que me faz seguir em frente.

Obrigada aos meus orientadores, prof. Dr. Antão, por aceitar a minha ideia e embarcar nessa comigo, por me estender a mão quando mais precisei e ao prof. Dr. Jacques pela sua sapiência e ensinamentos. Agradeço ainda aos meus orientadores extra-oficiais, prof. Dr. Dalton, que coordenou o projeto Auto Teste, foi amigo e mestre e muito me ensinou; ao prof. Dr. Fubica, que me trouxe pra Campina Grande e com quem aprendi muito sobre como ser um bom pesquisador e à minha amiga a prof. Dr. Raquel, que além de professora, conselheira e coordenadora é uma amiga para todas as horas.

Obrigada aos "meus meninos" o meu time. Formamos uma grande família! Guguito, Érick, Mel, Sabrina, Gil, Ciçu, Japão, Rômulo, Dimas, João, Ayla, Matheus, Bah, JP, Iury, Ramon, Paf, Laurinha, Tiago, Carlinha, Alcione e a todos os colegas do LSD por serem o alicerce dessa construção.

Agradeço aos colegas da pós, Aninha, Magna, Ianna, Geraldo pelo companheirismo e as amigas Vivi, pelo cuidado e carinho de sempre e Lili, pela sua amizade que ultrapassa as barreiras da distância.

Agradeço a COPIN e ao CNPq, pelo apoio financeiro, e a todos que de alguma forma direto ou indireta contribuíram para a realização não deste trabalho, mas de um grande sonho.

Conteúdo

Lista de Símbolos

ASD – *Adaptative Software Development*
C2 – *Cunningham Cunningham Ind.*
CMM – *Capability Maturity Model*
DBC – *Design By Contract*
DSDM – *Dynamic Systems Development Method*
FDD – *Feature Driven Development*
GQM – *Goal Question Metrics*
HP – *Hewlett-Packard*
IDE – *Integrated Development Environment*
ISO – *International Organization for Standardization*
LSD – *Laboratório de Sistemas Distribuídos*
OOPSLA –*Object-Oriented Programing System, Linguages and Applications*
OP – OurProcess
OQP – *OurQualityProcess*
P2P – *Peer-to-Peer*
PSP – *Person Software Process*
PU – *Processo Unificado*
QIP – *Quality Improvement Paradigm*
RAD – *Rapid Application Development*
RH – Recursos Humanos
ROP – *Rational Objectory Process*
RUP – *Rational Unified Process*
SEI – *Software Engineering Institute*
TDD – *Test Driven Development*
TSP – *Team Software Process*
UML – *Unified Modeling Language*
US – *User Story*
XP – *eXtreme Programing*
XP1 – *eXtreme Programing*

Lista de Figuras

Lista de Tabelas

Capítulo 1

1. Introdução

1.1. Contexto e justificativa

As atividades realizadas no LSD (Laboratório de Sistemas Distribuídos) há mais de 10 anos apontam para uma consolidação de resultados na área de Sistemas Distribuídos principalmente nas pesquisas em Grades Computacionais, Computação nas nuvens, Sistemas *Peer-to-Peer* (P2P), Tolerância a Falhas, Desenvolvimento de *Software* Distribuído e Concorrente e, mais recentemente, Aplicações Industriais e Qualificação de *Software*.

Um dos principais projetos do laboratório é o *OurGrid*, uma grade aberta, *free-to-join*, que agrega serviços computacionais e dá suporte a uma gama de aplicações chamadas de *bag-of-tasks*. O *OurGrid* é um *software* livre e existe um número expressivo de instituições, dentro e fora do país, que usaram ou estão usando essa tecnologia.

Dentro do escopo do laboratório também se desenvolveu o *OurBackup* [Oliveira, 2007], um sistema de *backup* P2P baseado em redes sociais. Em ambos os casos, o LSD buscou problemas reais no contexto do desenvolvimento e utilização desses produtos de *software* por clientes externos ao laboratório para desenvolver sua pesquisa. Pela importância desse projeto e por ser uma das iniciativas em grades computacionais de grande sucesso no Brasil, o LSD tem tido a preocupação de oferecer produtos de qualidade para a comunidade de usuários de grades.

Em função do cenário do LSD, isto é, equipe composta, em sua maioria, por alunos de graduação em Ciência da Computação; histórico da universidade no desenvolvimento de *software;* experiência em elaboração de processos de desenvolvimento como XP1 [XP1, 2002] e o tipo de *software* desenvolvido, optou-se por buscar a qualidade dos produtos

iniciando pela elaboração de um processo de desenvolvimento, baseado na metodologia ágil XP proposta por Beck [Beck, 2000], denominado *OurProcess [OurProcess]*.

A ideia do *OurProcess* era seguir a metodologia XP e adaptá-la de forma a minimizar os formalismos dos processos tradicionais e aumentar a flexibilidade dos projetos. Os processos ágeis surgiram como uma opção para desenvolver *software* de forma mais direta e menos burocrática que os métodos tradicionais, sem exagerar no controle e na documentação.

Nos cenários com mudanças constantes, como é o caso do ambiente utilizado como estudo, isto é, o ambiente do LSD, esse estilo de desenvolvimento mostrou-se vantajoso quando comparado aos tradicionais, principalmente devido à característica que os métodos ágeis tem de abordar diretamente os problemas ocasionados por mudanças frequentes e/ou repentinas ao desenvolver *software*.

Outra motivação para a adoção de um processo com pouco formalismo deve-se à natureza acadêmica, onde esse tipo de metodologia facilita o desenvolvimento e estimula o estudante a: reconsiderar constantemente, cada hipótese e premissa; fazer experimentos; construir e aprimorar seus conhecimentos através de tentativas e erros, sem que isso acarrete grandes riscos à organização e aos objetivos de desenvolvimento sem perder o foco na qualidade dos produtos desenvolvidos.

Durante o desenvolvimento de vários produtos, seguindo o processo *OurProcess,* notou-se que a qualidade dos produtos não estava nos níveis idealizados. Um fato que desencadeou a preocupação com a qualidade dos produtos foi a quantidade de defeitos simples reportados em seguida do lançamento de novas versões (novos *releases*), tanto pela comunidade do *OurGrid* como também por membros de outros projetos desenvolvidos no laboratório que se utilizavam da tecnologia *OurGrid*. Após o lançamento da versão *3.0* do *OurGrid* foi decidido que novas estratégias deveriam ser adotadas para melhorar a qualidade dos produtos.

A caracterização desta preocupação resultou em uma investigação para identificar os possíveis problemas. Dos problemas encontrados separaram-se os que foram considerados mais críticos, quais sejam:

- Violação de regras de integridade (perda da consistência dos dados);
- Violação de regras de negócios;

- Falhas de integração de unidades;
- Grandes desvios nas estimativas de esforço;
- Grandes desvios no cronograma;
- Falta de infraestrutura para execução de testes distribuídos;
- Falsos positivos em testes;
- Baixa cobertura e testes do código.

Após a identificação dos problemas, decidiu-se relacionar cada problema com suas possíveis causas:

1. XP não trata sobre mecanismos para testar regras de integridade ou mesmo regras de negócio, apenas de validação com o cliente das regras de negócio;
2. A atividade de integração deve ser rápida (5 a 10 minutos) o suficiente para não interferir na produtividade dos desenvolvedores e fornecer um *feedback* rápido assegurando assim a saúde do código (código sem falhas). Porém o tempo de execução da bateria de testes por módulo pode variar de 42 minutos a 68 minutos quando não há a ocorrência de erros de compilação. A bateria de teste é composta somente por testes de unidade e parte dos testes de integração, isto é, os testes de integração referentes a um módulo;
3. XP incentiva a criação de testes, porém não define como deve ser a estrutura de comunicação para a elaboração de testes para sistemas distribuídos, o que reduz a testabilidade (a probabilidade de um pedaço de *software* falhar na próxima execução dos testes assumindo que uma entrada do *software* inclui uma falha);
4. A prática de *pair programming*, quando executada em um ambiente não propício, pode não prevenir os vícios de programação que ocorrem quando os desenvolvedores são os mesmos responsáveis pela atividade de elaboração de testes e focados no comportamento esperado e não nas possíveis falhas, isto porque na academia não é possível garantir que os pares (na sua maioria alunos) terão horários compatíveis para desenvolverem seus trabalhos; em muitas situações os alunos cursam períodos diferentes e estão disponíveis para o trabalho em horários diferentes;
5. Muitos defeitos encontrados após a execução da bateria de testes eram falsos defeitos (falsos positivos), isso porque os mecanismos de *rollback* (para limpeza

do ambiente) não foram implementados, visando a minimização do tempo de execução da bateria de testes pré-integração;

6. Para criar mecanismos de *rollback* para cada teste executado, o tempo do *build* aumentava significativamente, pois para cada teste executado um mecanismo era implementado. Por exemplo, a bateria completa do *OurBackup* era composta por mais de 1.600 casos de teste com tempo aproximado de duração de 1 hora. Com o mecanismo de *rollback* para cada caso de teste, esse tempo aumentava e não era possível executar a bateria a cada integração sem comprometer o rendimento dos desenvolvedores.

Para aumentar a qualidade dos produtos desenvolvidos foi proposto um novo conjunto de princípios e práticas que auxiliassem os desenvolvedores a buscarem produtos com maior qualidade. Esses princípios e práticas estão organizados na forma de um novo processo, denominado *OurQualityProcess* (*OQP*) [*OurQualityProcess*]. Sendo que, o *OurQualityProcess* é um aperfeiçoamento do *OurProcess*.

A elaboração do *OQP* iniciou a partir dos requisitos de qualidade desejados pelos líderes de desenvolvimento do LSD. Como a necessidade de elevar a qualidade dos produtos desenvolvidos era alta e urgente, não foi possível testar cada um dos princípios e práticas propostos e aferir o impacto na qualidade desses separadamente. Foi preciso adotar uma estratégia emergencial para que o resultado na qualidade fosse obtido no menor tempo possível.

Para constatar que os requisitos de qualidade identificados pelos líderes de desenvolvimento são condizentes com as expectativas e prioridades do mercado, desenvolveu-se ainda uma pesquisa junto a setenta e sete gerentes de desenvolvimento e de controle de qualidade de produtos de *software* com experiência em gestão por mais de três anos. A *survey* foi disponibilizada na internet através de um dos recursos do *Google[Google]*, o *Google Docs Forms [GoogleForms]*. A descrição da pesquisa e os resultados corresponderam às mesmas necessidades reportadas pelos líderes de desenvolvimento do nosso estudo de caso.

Tendo todos os problemas identificados no LSD, confirmados através da *survey*, iniciou-se a elaboração e implementação do novo processo buscando a maximização da

qualidade dos produtos desenvolvidos e auxiliar desenvolvedores de *software* a aprimorarem a qualidade de seus produtos através de um conjunto de princípios e práticas utilizadas conjuntamente com os princípios da programação extrema.

Os resultados obtidos com o *OQP* após a inserção do conjunto de princípios e práticas, mostraram fortes indícios de que a qualidade dos produtos foi elevada conforme demonstra o estudo de alguns casos.

1.2. Problemas

Dos problemas identificados no desenvolvimento dos produtos *OurGrid*, selecionamos quatro deles por serem os mais críticos e com maior impacto na qualidade do produto final de acordo com os líderes de desenvolvimento. Cada problema será detalhado nas subseções seguintes.

1.2.1. Regras de Integridade

Durante o desenvolvimento do produto *OurBackup*, aconteceram muitas falhas relativas a parâmetros de métodos que ou estavam nulos ou não estavam respeitando regras de integridade entre classes e módulos. Em relação aos parâmetros, havia problemas com objetos sendo construídos sem receber todas as informações necessárias para o funcionamento (pré-condições). Por exemplo, existiam os parâmetros no construtor, mas era passado um valor *null* na instanciação, além de outros casos em que ocorria uma falha, mas a detecção era muito custosa.

Em relação às regras de integridade, elas não eram facilmente testáveis; para que fosse possível perceber uma violação dessas regras era necessária uma revisão de todo o código, o que se tornava impraticável para um *software* com quase 35 mil linhas, por exemplo.

A maneira com que a prática de TDD [Astels, 2003; Back, 2003] estava sendo aplicada no ambiente de estudo não era suficiente para detectar as falhas no *software*; muitas falhas ocorriam na interação entre os módulos e não na integração nas unidades do código ou na integração de classes e módulos. Onde somente teste dependentes da execução poderiam detectar as falhas.

1.2.2. Execução do Build de Integração

No desenvolvimento dos produtos *OurGrid* [OurGrid], *OurBackup* [Oliveira, 2007] e *JIC* [Lima, 2006] (protocolo de comunicação do *OurGrid* 3.2) foi adotada a prática XP de integração contínua do código, isto é, a cada pequena implementação realizada, uma bateria de testes era executada no novo código desenvolvido, o código era integrado e iniciava-se uma nova implementação. Portanto toda a implementação, por menor que seja, deverá ser integrada ao código (apenas se) não houver falhas durante a execução nos testes.

Porém, a execução da bateria completa de testes do *OurBackup*, por exemplo, demorava em média 1 hora, o que tornava impraticável a execução da bateria de testes cada vez que fosse necessário integrar uma alteração no código (um *commit*). O desenvolvedor não poderia esperar um tempo tão grande para continuar suas atividades, visto que cada desenvolvedor, – por ser aluno tipicamente - trabalha em períodos que variam de 2 a 4 horas diárias. Em média ele desenvolve duas a três implementações por hora e se a cada implementação se devesse esperar 1 hora para integrá-la, não seria viável. No caso, ele desenvolveria por 1 hora e esperaria 3 horas pelos *builds*.

Conforme afirma [Back, 2003], a atividade de integração deve ser rápida (5 a 10 minutos), isto é, o suficiente para não interferir na produtividade dos desenvolvedores e fornecer um *feedback* rápido assegurando assim a saúde do código (código sem falhas).

1.2.3. Testabilidade

Produtos de *software* modernos, essencialmente concorrentes e distribuídos, impõem desafios para avanços e melhorias no processo de qualificação [Flanagan, 2005; Rutherford, 2006]. A natureza concorrente dos programas tende a torná-los intrinsecamente não determinísticos, dificultando o controle de sua execução e, conseqüentemente, reduzindo sua testabilidade. Sua natureza distribuída, por outro lado, impõe a necessidade de realizar testes de sistema sobre infraestruturas de comunicação distribuídas. As dificuldades em controlar a infraestrutura de comunicação, bem como a de detectar falhas, tendem a influenciar nos resultados dos testes e no processo de qualificação.

XP incentiva a criação de testes, mas não define como deve ser feito a variabilidade dos casos de testes ou mesmo a variabilidade do ambiente de testes, o que é fundamental para elevar a qualidade do *software*.

A prática de programação em pares nem sempre é favorecida no ambiente acadêmico por dois motivos:

1. Nem sempre há compatibilidade de horários entre os pares. Os pares, na sua maioria alunos, tem horários diferentes e se um par consegue estabelecer horários compatíveis, a prática de rodízio de pares inviabiliza esse tipo de planejamento para um único par. Com isso, adotar a prática de programação em pares nem sempre é favorecida num ambiente onde não é possível planejar os pares entre os desenvolvedores devido a incertezas de horários;
2. Alunos com maiores afinidades tendem a planejar um pareamento contínuo, prática que também interfere na boa prática de rodízio entre pares estimulada por XP.

1.2.4. Falsos Positivos

A detecção de falsos positivos (falsas falhas), isto é, falhas que sinalizam que a execução do teste não retornou o valor esperado, quando na verdade o ocorrido foi um erro na elaboração do testes ou erro do ambiente de teste ou ainda pelo uso incorreto de operações. Tem-se vários exemplos detectados no ambiente de estudo: atrasos explícitos (por exemplo, a utilização *Thread.sleep* para que uma determinada *thread* do teste exercitado espere por um tempo pré-definido pelo testador), espera ocupada (atrasos explícitos em laços quando uma condição geral está sendo verificada), ou ainda a remanescência de lixo de outros casos de testes que provocavam quebra no código pela falta de limpeza do ambiente de teste, (por exemplo, após a execução de um teste, algumas variáveis eram incrementadas ou rotinas eram executadas de modo a alterarem o estado do sistema e como não havia mecanismos de *rollback*, o estado alcançado não era o inicial e as variáveis permaneciam com valores não esperados para a execução de um novo caso de teste).

No exemplo da remanescência de lixo, foram detectados alguns falsos positivos. Imaginava-se que uma falha ocorrera, mas na investigação notava-se que era lixo de testes anteriores. Como o teste estava associado a uma bateria de testes, era difícil prever essas situações, isso porque não há controle em tempo de execução das variáveis que estão sendo manipuladas, por exemplo. Para controle da execução seria necessário criar um mecanismo

de monitoramento para a execução de testes, o que poderia trazer mais malefícios por incrementar o tempo de execução dos testes.

Além disso, na criação de mecanismos de *rollback* para cada teste executado, o tempo do *build* aumentava significativamente, pois para cada teste executado, um mecanismo era implementado. Por exemplo, a bateria de um módulo com 230 casos de testes era executada em aproximadamente 8 minutos, com o mecanismo de *rollback* a execução levava 12 minutos. Para a bateria completa composta por mais de 1.600 casos de testes a adição do mecanismo de *rollback* aumentaria significativamente o tempo de execução.

1.3. Categorização dos Problemas

Os problemas mencionados anteriormente são relativos a três grandes categorias de problemas pertinentes ao desenvolvimento de *software*, que são:

- Produtividade – Tempo de desenvolvimento;
- Confiabilidade – Ausência de defeitos e falhas;
- Manutenibilidade – Facilidade para modificar, aperfeiçoar, corrigir e testar o *software*.

Essas categorias são fortemente relacionadas, porque o problema da confiabilidade (ocasionado principalmente por defeitos e falhas no *software*) é o motivador do problema da manutenção. Para cada defeito detectado é necessária uma manutenção no código para a correção do programa, consumindo recursos de infraestrutura e de pessoal e, consequentemente, interferindo na produtividade, pois aumenta o tempo de desenvolvimento. Estabelece-se um ciclo vicioso visto que se os recursos são alocados para manutenção, as modificações e aperfeiçoamentos serão então postergados, afetando por sua vez, a produtividade.

1.4. A relevância dos Problemas

Foi desenvolvido um *survey* para identificar os principais fatores para avaliar a qualidade de um produto de *software* (e seu desenvolvimento) e o grau de importância de cada um deles na visão de gerentes de desenvolvimento e gestores de qualidade. Os resultados serviram para estabelecer requisitos a serem atendidos pelos processos de desenvolvimento.

Os fatores levantados conforme grau de importância foram:

Defeitos no código (confiabilidade): São erros de funcionamento no *software* que ocasionam divergências entre as funcionalidades esperadas e as executadas. Há várias categorias de defeitos (*bugs*). De acordo com a ferramenta *Jira* da *Atlassian* (repositório de *issues*) os defeitos podem pertencer a uma, dentre cinco classes de impacto, são elas: bloqueantes, críticos, major, minor ou triviais, além da possibilidade de configurar outras categorias que se queiram.

Cobertura de testes (confiabilidade): Linhas de código verificadas através de testes (rotinas que verificam a execução do código). Um código estará 100% coberto se a bateria de testes executada percorrer todas as linhas do código.

Atraso no cronograma (produtividade): Entrega do produto após a data prevista no planejamento (prazo de entrega subestimado).

Erro na estimativa de esforço (produtividade): (esforço = tempo necessário para o desenvolvimento de uma tarefa), quando os desenvolvedores subestimam ou sobreestimam a execução de uma tarefa/atividade.

***Issues* em aberto (manutenibilidade):** (*issues* = requisições solicitadas por desenvolvedores/clientes/usuários) quando o cliente/usuário requisita alguma *issue* para o desenvolvimento, a *issue* é cadastrada e fica em aberto até que seja resolvida e fechada. As *issues* podem ser classificadas como: novas funcionalidades, melhorias, detecção de defeitos (*bugs*) ou tarefas (*task*).

Falta de qualidade (conjunção de todos os problemas): Segundo a norma ISO 9000 (versão 2000), a qualidade é o grau em que um conjunto de características inerentes a um produto, processo ou sistema cumpre os requisitos inicialmente estipulados para estes sem falhas. A falta de qualidade é o não cumprimento dos requisitos estipulados pelos clientes/usuários ou falhas nos requisitos funcionais do *software*.

Falta de ambiente para testes distribuídos (manutenibilidade + testabilidade): Escrever testes concorrentes é difícil porque as infraestruturas existentes de teste não proporcionam mecanismos para expressar, observar e controlar linhas de execução (*threads*) apropriadamente; em particular, é muito difícil identificar quando as assertivas podem ser executadas de forma segura, ou, de forma equivalente, quando a atividade do *software* sob teste cessou. No caso de testes distribuídos a tarefa é ainda mais difícil: os

cenários incluem todos os problemas mencionados para *software* concorrente e adicionam problemas específicos como: *deployment*, latência na comunicação, falhas de rede, dificuldade de injeções de falhas, entre outros.

1.5. Objetivos e Contribuições

O objetivo geral deste trabalho é identificar o impacto da adoção de estratégias de qualidade de *software* dos processos tradicionais em um processo que segue a metodologia ágil baseada em eXtremme Programming (XP). Este impacto é aferido através de uma comparação, utilizando o teste de wilcoxon pareado para comparar a qualidade dos processos *OurProcess* e *OurQualityProcess* e identificar se são equivalentes.

Portanto, os objetivos primários são:

- Realizar um estudo bibliográfico para identificar o estado da arte dos processos desenvolvidos sob a perspectiva da metodologia ágil;
- Identificar lacunas do *OurProcess.*
- Identificar estratégias amplamente difundidas na literatura para aquisição e controle de qualidade em *software.*
- Propor um conjunto de estratégias de qualidade para maximizar a qualidade dos produtos;
- Implantar as estratégias elaboradas em um processo de desenvolvimento de *software* existente;
- Comparar, utilizando metodologia científica, o processo existente com o mesmo processo acrescido do conjunto de estratégias de qualidade e aferir a equivalência dos processos.

E os objetivos secundários são:

- Identificar as lacunas não cobertas pelos processos;
- Pontos falhos do processo proposto.

1.6. Delimitações do Estudo

Em função da amplitude do tema escolhido (elaboração e implantação de um processo de desenvolvimento com foco na qualidade dos produtos desenvolvidos), foi necessário delimitar o escopo do trabalho. O primeiro ponto a ser delimitado foi o conjunto

de fatores de qualidade avaliados, para isso foi necessária uma pesquisa para identificar os principais problemas de qualidade enfrentados. Foi usado como estudo de caso o desenvolvimento no LSD dos produtos *OurGrid*, em especial o *OurBackup Home* por ser um produto de porte médio e que seguia o processo *OurProcess* na íntegra. Além disso, foi elaborado um *survey* com gerentes de desenvolvimento com mais de três anos de experiência na área para confirmar que os problemas identificados pelos líderes de desenvolvimento do LSD eram compatíveis com os problemas identificados por experientes gestores de desenvolvimento de *software*.

Entende-se que com os resultados apresentados neste trabalho não é possível generalizar automaticamente para processos distintos dos aqui apresentados, porém acredita-se também que os resultados apresentados geram subsídios suficientes para a realização de trabalhos futuros que utilizarem parte ou mesmo a totalidade das estratégias propostas para elevar a qualidade dos produtos desenvolvidos.

1.7. Estrutura do Trabalho

uuuA estrutura desse trabalho é dividida em capítulos. Os capítulos são:

1. Introdução, que apresenta a contextualização e justificativa do trabalho, além da apresentação dos problemas identificados, os objetivos e contribuições do trabalho e ainda a delimitação do estudo; **2. O processo proposto, o *OurQualityProcess***, contemplando os princípios, o ciclo de vida, o contrato, as atividades e os papéis. **3. Avaliação do *OurQualityProcess***, capítulo que apresenta duas avaliações, a primeira, seguindo a metodologia científica através da definição do estudo experimental, passando pelo planejamento, preparação, execução, refutação das hipóteses formuladas, a análise de qualidade dos produtos e respectivas ameaças à validade e considerações finais do experimento. **4 . Avaliação através do Baseline,** a segunda avaliação realizada através de um estudo comparativo entre os dois processos sob análise com um *baseline*, também seguindo a metodologia científica, bem como as fases de elaboração do *baseline* e **5. Conclusões e Trabalhos Futuros**, contendo as conclusões obtidas do resultado geral do trabalho e possibilidades de trabalhos futuros. No **Apêndice A - Revisão Bibliográfica**, capítulo com as contribuições da literatura, o estado da arte da área de processos de desenvolvimento de software, as metodologias ágeis, alguns tipos de processos.

Capítulo 2

2. Proposta de Melhoria para o OurProcess: *OurQualityProcess*

2.1. Principais assuntos do capítulo

Princípios

Ciclo de Vida

Contrato

Papeis

Tipos de testes

2.2. Introdução

Nos últimos anos constatou-se que a qualidade de um *software* está intimamente ligada ao processo de desenvolvimento utilizado [Guerra, 2002], e por esta razão, a busca por um processo de desenvolvimento ideal tem aumentado. Entretanto, adotar um processo de desenvolvimento não é uma tarefa simples, principalmente quando as equipes são compostas por um número restrito de integrantes [Crispin 2009].

Neste contexto, por ser um modelo de processo extremamente aderente ao ambiente de pequenas equipes [Beck 2000], foi escolhido como ponto de partida o XP para elaboração do *OurProcess*. Porém, após a implantação em alguns produtos desenvolvidos, foi constatado que a qualidade dos produtos desenvolvido estava abaixo do esperado.

Seguindo a filosofia de melhoria contínua, foi constatado que seria necessária a adoção de estratégias para maximizar a qualidade dos produtos. Foi essa adaptação no processo *OurProcess* resultou em um novo processo, o *OurQualityProcess* (*OQP*).

O *OurQualityProcess* é uma extensão do *OurProcess,* um processo que adiciona poucas obrigações à equipe de desenvolvimento com metodologia ágil, de forma a ser minimamente intrusivo. Ele foca na elaboração de requisitos claros e testes automáticos e reproduzíveis e documentados em um *wiki* coletivo. Novos cenários de teste são, continuamente, identificados através de testes manuais, os quais são revisados e

automatizados tanto quanto possível. Além disso, *OQP* propõem a inserção de uma equipe de qualificação para atuar, juntamente, com a equipe de desenvolvimento, promovendo assim, um esforço duplicado com foco na detecção de defeitos e busca pela satisfação dos clientes.

Uma das prática do *OurQualityProcess*, diferentemente do processo XP e inexistente no *OurProcess* é o foco na validação da descrição dos requisitos, com crítica direta a cada sentença. Com isso, espera-se que aspectos como completude, corretude e ambiguidade sejam avaliados. Esta prática minimiza problemas de escrita e interpretação, conduzindo assim a uma documentação executável na forma de testes automáticos coesos e corretos que durem "a vida toda" do *software*.

Além da análise e validação dos requisitos, adotamos princípios que direcionam o processo e práticas na busca da qualidade do *software*.

Quanto às práticas, pode-se ressaltar a adoção de *Test-Driven Development* – TDD [Beck, 2003] tanto em testes de unidade como em testes de integração adotadas tanto pela equipe de desenvolvimento quanto pela equipe de qualificação. Destacam-se também que as práticas não são limitadas à equipe de qualificação, as mesmas afetam também a forma de atuação dos desenvolvedores através da descrição detalhada dos requisitos, da elaboração dos testes que validem cada um dos requisitos, revisões constantes, refatoramentos, desenvolvimentos por contratos e a execução da bateria de testes, diariamente, contendo todos os testes elaborados para avaliar a saúde do código. A seguir os princípios que regem o *OQP*.

2.3. Princípios

Inserção de uma equipe de qualificação – A equipe de qualificação atua paralelamente à equipe de desenvolvimento, para cada fase do desenvolvimento, uma atividade de qualificação também é desenvolvida. Com isso espera-se desenvolver uma bateria externa de testes da qualificação que deve ser incorporada à bateria de teste do desenvolvimento no final de um dia de trabalho. Além disso, a equipe de qualificação tem seus esforços direcionados para a validação dos requisitos funcionais do produto em desenvolvimento, isto é, para cada requisito um ou mais testes deve ser elaborado garantindo uma documentação de validação executável.

Qualificação Gradativa – De acordo com a filosofia do XP "Faça a coisa mais simples que pode funcionar" [Beck 2004], ela se aplica também para a qualificação. Inicialmente, validam-se os requisitos, através da crítica direta a cada sentença da documentação conduzindo assim, a uma documentação executável na forma de testes automáticos coesos e corretos que durem "a vida toda" do *software* e possam ser re-executados a um custo mínimo.

Após a elaboração dos testes de aceitação básicos, inicia-se a expansão dos testes, denominado fase de explosão dos testes, com o objetivo de estressar o código, onde para cada teste de aceitação produzido pelos desenvolvedores, um ou mais testes, quando possível, devem ser desenvolvidos pela qualificação. Esta prática identifica defeitos provocados por vícios de programação e erros lógicos dos desenvolvedores.

Por fim, executa-se a bateria de testes manuais e valida-se a correção dos defeitos. É importante ressaltar que a condição de parada dos testes é a obtenção de uma cobertura mínima de 80% para cada funcionalidade, visto a inviabilidade de cobrir 100% do código devido a classes e arquivos não testáveis, como, por exemplo, arquivos de configuração.

Manutenabilidade da Saúde do Código – Todo novo código deve passar por uma bateria de testes automatizados para ser integrado ao repositório. Esta integração permite a verificação da "saúde" do código. Entretanto, são definidas diferentes baterias de testes, cada uma com uma especificidade particular. Durante a codificação, a bateria executada é simplificada. Nela estão somente os testes de unidade referentes ao módulo em desenvolvimento. A bateria de testes aumenta conforme a evolução do desenvolvimento do *software*. Por conseguinte, todas as noites a bateria completa dos testes é executada gerando um status diário da "saúde" do código.

Além disso, a integração de um novo teste desenvolvido deve ocorrer o mais breve possível para que todos os membros das equipes tenham acesso ao novo teste e, consequentemente, aumente a verificação dos novos códigos desenvolvidos.

Outra prática adotada para manter a "saúde" do código foi a adoção de *Design by Contract* (DBC) para mapear as responsabilidades de objetos e classes, tornando a implementação mais robusta e segura. As regras do negócio são verificadas na forma de

asserções lógicas, para verificar se os dados de entrada e saída foram processados ou construídos corretamente.

Revisão do código – A revisão do código deve ser realizada por uma pessoa que não participou do desenvolvimento, preferencialmente, pelo líder da equipe, seja do desenvolvimento ou da qualificação. A programação em par também aumenta a confiabilidade do código desenvolvido. A adoção do princípio de revisão refletirá na redução dos erros, quebras de contratos, interpretações equivocadas, aumento da legibilidade do código e melhora do *design*.

Seguindo os princípios da qualificação, inseridos no processo de desenvolvimento, a cada fase do processo de desenvolvimento, uma atividade correspondente deve ser executada pela equipe de qualificação, conforme demonstrado na Tabela 1.

2.4. Ciclo de Vida

O desenvolvimento dos requisitos, bem como a aplicabilidade dos princípios ocorre através de pequenas liberações (*releases*) durante as iterações. Cada *release* é definido como uma ou mais funcionalidades selecionadas de acordo com a necessidade do cliente. O desenvolvimento integral do *software* é uma sucessão de *releases* codificados e testados. Isso porque o ciclo de vida do *software* será a junção de todos os *releases* desenvolvidos. As atividades executadas durante o *release* são identificadas através das linhas que cortam a Figura 1 e são descritas mais detalhadamente na Tabela 1.

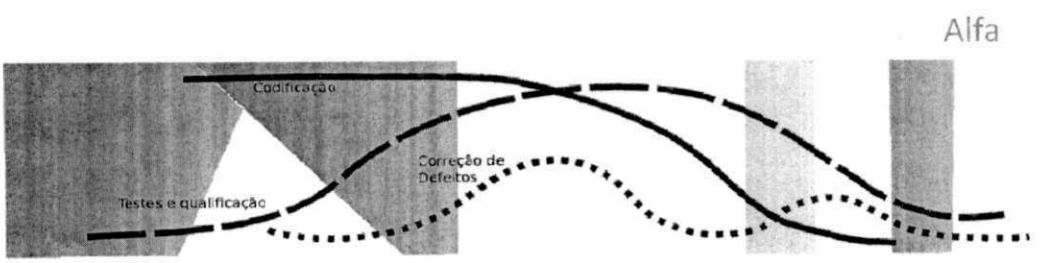


Figura 1: Ciclo de Vida do Release

Os *releases* são divididos em quatro fases: Elicitação de Requisitos, Desenvolvimento, Testes Exploratórios e Alfa, conforme ilustra a Figura 1. Durante essas fases, as atividades das equipes de desenvolvimento e qualificação são paralelamente realizadas. Ao final do *release*, tem-se uma versão desenvolvida e qualificada.

Tabela 1 – Relação entre atividades e fases do ciclo de vida da release.

Equipes	Elicitação de Requisitos	Desenvolvimento	Testes Exploratórios	Alfa
Desenvolvimento	Escrever requisitos	Implementação de código	Correção de defeitos	Correção de defeitos
	Definir design	Implementação de testes de unidade e integração		
	Elaborar testes de aceitação	Correção de defeitos		
Qualificação	Verificar e validar requisitos	Implementação de novos casos de testes de integração e sistema.	Testes de aceitação automáticos; Execução de testes manuais e exploratórios	Validação da correção dos defeitos
	Elaborar novos testes de aceitação (estender casos de teste do desenvolvimento)			

Cada uma das quatro fases, vistas na Figura 1, representa o estado em que se encontra o desenvolvimento do *release*, representado através de diferentes cores e três linhas diferenciadas que representam as atividades de: codificação; testes e qualificação; e correção de defeitos. Cada fase tem um objetivo bem definido, que são:

ELICITAÇÃO DOS REQUISITOS:

Nessa fase todas as funcionalidades de um *release* serão descritas textualmente para direcionar o desenvolvimento e para que se tenha uma visão geral das necessidades do desenvolvimento.

A função da equipe de desenvolvimento é, juntamente, com o cliente e o *coach* do projeto, formalizar os requisitos, enquanto que a equipe de qualificação analisa cada requisito, sugere alterações, novas descrições e os transforma em documentação executável, isto é, testes de aceitação automáticos.

Entre as fases de Elicitação dos Requisitos e Desenvolvimento não há um marco separando-as, isso porque o trabalho dos desenvolvedores não é paralisado para a validação dos requisitos, essa transição é contínua e progressiva e finalizada no instante que: todos os requisitos forem validados, textualmente, e que a equipe de qualificação tenha finalizado a elaboração do conjunto de testes de aceitação (manuais e automáticos) que mapeiam todos os requisitos do *release*.

DESENVOLVIMENTO

A fase de desenvolvimento é fundamentada nos princípios do XP, antes de qualquer implementação de unidades de código a equipe de desenvolvimento realiza o TDD. Portanto, o desenvolvimento é a conjunção da elaboração do teste de unidade e a codificação da unidade em questão. Como complemento do desenvolvimento ainda existe a atividade de análise/solução das *issues* (tickets/requisições) reportadas, isto é, requisições feitas pelos *stakeholders* do *release* como: (*reports* de defeitos, solicitações de novas funcionalidades, solicitações de novas tarefas, de novos testes, entre outras solicitações).

Paralelamente a elaboração dos testes de unidade, a equipe de qualificação se responsabiliza pela elaboração de testes com o mesmo cenário, mas diferentes situações, essa fase é denominada explosão dos casos de testes.

Quando um novo defeito é descoberto, imediatamente o qualificador ou desenvolvedor que o encontrou o reporta. Nesse momento, o responsável pelo cadastro é o detentor da análise de validação da solução destinada ao seu cadastro. Ele pode aceitar a solução ou reabrir o cadastro.

Em relação à atividade de testes automáticos, os testes de aceitação *Happy Day* (fluxo mais simples) devem ser elaborados pela equipe de desenvolvimento. A partir do *Happy Day* e de acordo com a descrição das funcionalidades, novos casos de testes serão desenvolvidos pela equipe de qualificação. O maior esforço da equipe de qualificação deve ser na execução e desenvolvimento de testes automáticos.

De acordo com a evolução do desenvolvimento, a equipe de qualificação se dedica a diversificar os casos de testes. O final da fase de Desenvolvimento é marcado pela liberação do *release* contendo todas as funcionalidades desenvolvidas e associadas a pelo menos um teste de unidade, elaborados pela equipe de desenvolvimento. Nesse instante a equipe de desenvolvimento congela a implementação e oficializa a entrega para a equipe de qualificação através da criação de um *branch*.

Idealmente a equipe de qualificação deve também ter desenvolvido todos, ou a maioria, dos testes automatizados, caso contrário, enquanto a qualificação finaliza a elaboração dos testes automáticos, a equipe de desenvolvimento, no *trunk* do projeto, foca

seus esforços na correção dos defeitos e/ou na elicitação dos requisitos do próximo *release* a ser desenvolvido e/ou nos casos de teste de aceitação *Happy Day*.

Ao final da automatização e explosão dos casos de testes, pela equipe de qualificação, ocorre a sincronização entre as versões com a finalidade de sincronizar o trabalho de ambas as equipes em uma solução consolidada.

TESTES EXPLORATÓRIOS

A fase de Testes Exploratórios inicia com execução da bateria de testes manuais (descritas no plano de testes elaborado pela equipe de desenvolvimento) na versão congelada do *release*. Os testes manuais são executados, pela equipe de qualificação, para validarem os casos que:

- Não foram possíveis de serem automatizados ou
- O esforço para automatizar é mais custoso que o esforço de executá-lo manualmente.

O marco entre a fase de Testes Exploratórios e a Alfa é identificado pela validação da solução das *issues* pela equipe de qualificação. O ideal é que todas as *issues* reportadas tenham uma solução elaborada pela equipe de desenvolvimento

Muitas vezes uma *issue*, do tipo defeito, pode ser uma nova funcionalidade requerida ou então, pode ser impossível ou não desejável a sua correção. Alguns defeitos encontrados, para serem corrigidos, demandam um tempo e custo maior que o benefício que eles proporcionariam. Portanto, após negociações com o cliente, este defeito poderá não ser corrigido e uma versão Alfa lançada.

ALFA

Na fase Alfa o *software* é lançado para os *stakeholders* do projeto, isto inclui os participantes do desenvolvimento, qualificação, clientes, além de integrantes de equipes de outros projetos e alguns convidados. Situações complexas são testadas, testes exploratórios dirigidos e não dirigidos são executados.

Caso alguma *issue* seja cadastrada durante a execução Alfa, uma nova reunião com o cliente é realizada para definir o que deve ser incorporado, melhorado, corrigido, retirado refatorado na versão e, consequentemente, a versão é finalizada. A avaliação feita pelo

cliente pode resultar em novos pacotes com alterações, modificações e/ou correções do *release*.

Para que todas as fases do desenvolvimento apresentadas possam ocorrer paralelamente entre as equipes de desenvolvimento e qualificação existe um conjunto de regras definidas na forma de contrato entre as duas equipes.

2.5. O Contrato

É fundamental para o desenvolvimento do trabalho paralelo entre as equipes de desenvolvimento e qualificação que todas as regras do contrato sejam cumpridas. Esse contrato deve ser elaborado, posteriormente, a elicitação dos requisitos.

Cláusula 01: É dever da equipe de desenvolvimento implementar e manter uma fachada para testes, contendo todos os métodos necessários para a validação e verificação das funcionalidades do produto (através de *release*), bem como mecanismos de injeção de falhas para testar situações difíceis ou até mesmo impossíveis de testá-las sem o uso desse tipo de recurso.

Cláusula 02: É dever da equipe de desenvolvimento descrever os requisitos do *software*, identificados como *User Stories* (US). Cada US deve ser qualificada e mantida ao longo do desenvolvimento. No entanto, elas podem sofrer alterações, complementações ou mesmo a criação de uma nova funcionalidade durante todo o desenvolvimento. Depois de escritas, as USs passam por uma fase de homologação, isto é, a validação da descrição da(s) funcionalidade(s) pelo cliente e só então a equipe de qualificação começa a validação das USs.

Cláusula 03: Para uma US estar aceitável, ela precisa conter:

1) Identificação do tipo da US:
    - Funcionais – Representam funcionalidades do sistema, O que ele deve fazer. A maioria dos requisitos de um projeto são requisitos funcionais.
    - Desempenho – quão rápido o sistema deve ser.
    - Usabilidade – facilidade de uso da interface, de acordo com o perfil dos usuários.

- Segurança – quão seguro o sistema deve ser. Exige criptografia, *firewall*, login com senha.
- Confiabilidade – certeza que as transações foram concluídas e que os dados retornados pelo sistema estão corretos e atualizados.
- Disponibilidade – qual a porcentagem de tempo que o sistema deve estar no ar. (Por exemplo: 99% - Baixa; 99,9% Média; 99,999% Alta).
- Localização – Identificação das linguagens que a interface do sistema deve disponibilizar.
- Manutenabilidade – quão fácil deve ser para manter o sistema. Legibilidade do código.
- Extensibilidade – o sistema deve prever a inclusão de novas funcionalidades.
- Testabilidade – o sistema deve ser capaz de ser testado por outra equipe.

2) Descrição da funcionalidade – devem ser escritos em sentenças simples e curtas, que delimitam o escopo da solução. É importante ressaltar que os requisitos não amarram "como" o sistema deve ser feito, e sim "o quê" ele deve fazer. Evitando ambigüidade, através de sentenças claras e concisas.
3) Descrição Comportamental das funcionalidades – as saídas esperadas. Por exemplo, a descrição do que é esperando quando houver uma entrada vazia ou inválida?
4) Um teste de aceitação, denominado *Happy Day*, desenvolvido pela equipe de desenvolvimento.

Cláusula 04: É de responsabilidade de ambas as equipes manter as USs atualizadas. Sempre que uma modificação for feita no *software* deve-se modificar a US, uma vez que grande parte dos testes de aceitação automáticos são desenvolvidos de acordo com a descrição das funcionalidades.

Cláusula 05: Todo o planejamento e realização das atividades de projeto devem estar condizentes com a priorização feita pelo cliente tanto para a equipe de desenvolvimento quanto para a equipe de qualificação.

O planejamento de ambas as equipes é quinzenal intercalado com reuniões de acompanhamento. Nas reuniões de planejamento ficam acordadas as priorizações, metodologias, problemas encontrados anteriormente, refatoramentos, definição e alocação das tarefas para cada integrante, bem como o tempo necessário para o cumprimento das atividades.

Na reunião de acompanhamento é monitorado o progresso das atividades das equipes, ocorre a garantia da comunicação interna, de forma que toda a equipe conheça como um todo o que está sendo desenvolvido e os problemas e/ou dificuldades também devem ser reportadas para toda a equipe, visto que algumas soluções são encontradas em reuniões do projeto.

2.6. Atividades do Processo:

Incrementalmente ao processo de desenvolvimento *OurProcess,* algumas novas atividades, referentes às atividades de qualificação foram inseridas ou adaptadas. A seguir a listagem de cada uma delas com suas definições:

2.6.1. Planejamento das Iterações

O desenvolvimento é dividido em iterações, cada iteração tem a duração de duas semanas. Esta periodicidade pode ser alterada, caso haja necessidade, porém todos os membros da equipe devem ser comunicados da antecipação ou da prorrogação da iteração.

Nas iterações fica acordada a responsabilidade/atribuições de todos os membros da equipe. Cada atividade planejada é cadastrada em uma ferramenta de gerenciamento e alocada para um membro da equipe, muito embora, outros membros possam auxiliar na resolução da atividade.

Em muitos casos as atividades são realizadas em pares. Esta prática ajuda a evitar erros de codificação, auxilia o par a manter um foco e enquanto um membro do par implementa o outro pensa na solução como um todo. Sempre que uma atividade não puder ser desenvolvida em par, obrigatoriamente um revisor de testes (*teste reviewer*) deverá ser alocado para revisar o código implementado.

As atividades planejadas devem ser estimadas de acordo com a percepção do executor e membros da equipe. Além disso, elas também devem ser associadas a um

tracker do projeto, o qual tem a função de acompanhar o andamento das atividades e garantir o cumprimento do planejamento.

As iterações são identificadas por um índice. Este índice tem por finalidade identificar e organizar a execução do projeto, conforme a priorização feita pelo cliente. Este índice é composto pela letra "R", de *Release*, seguida de numerais seqüenciais iniciando em "01", espaço, hífen, espaço e um nome. Por exemplo: R01 – Nome significativo para a *Release.*

Em cada iteração são criadas as *User Stories* (USs) – histórias do cliente. Cada US por sua vez deve ser considerada como uma macro atividade, que pode ser uma ou um conjunto de funcionalidades do desenvolvimento. Existem ainda USs internas de cada linha que não, necessariamente, representam uma atividade funcional do *software.* Os casos mais comuns de USs internas são: reuniões, preenchimento de relatórios; configuração de ambiente; atualização de máquinas; instalação de programas; entre outras. O padrão de identificação das USs devem ser as letras "US", seguidas de numerais seqüenciais, iniciando por "01", espaço, hífen, espaço e um nome significativo. Para cada US é definido um conjunto de atributos, conforme o exemplo:

Nome: US01 – Nome para identificação da US que expresse o tipo de atividade.

Disposição: pode ser planejada (uma nova US), transferida (iniciou em alguma iteração passada) ou adicionada (após o planejamento houve a necessidade de criação de uma nova US).

Cliente: Nome do Cliente (o gerente do desenvolvimento pode assumir o papel de cliente ou o coordenador do projeto).

***Tracker*:** Nome do *Tracker* do projeto. (O líder de linha pode assumir este papel, caso necessário).

Status: Pode assumir sete valores – Rascunho (não foi consolidada a US), Definido (US a ser executada sem previsões de término), Estimado (definida e estimada), Planejado (definida, estimada e planejada), Implementado (já foi desenvolvida), Verificado (em fase de verificação), Aceito (Completa).

Prioridade: crescente de 1 até 4.

Tempo estimado: em horas.

Existem três USs em uma iteração padrão, são elas:

US00a – Reuniões, onde devem ser reportadas todas as reuniões tanto de planejamento e acompanhamento, como também reuniões menores entre membros, com os clientes, reuniões técnicas e outras que se façam necessárias para o progresso do trabalho.

O marco para identificação de um nova iteração deve ser a realização de uma reunião de planejamento.

US00b – Atividades Gerais, são atividades que fazem parte da iteração, mas não dos objetivos principais dela e sim atividades necessárias para a evolução do trabalho, como o preenchimento de relatórios, configurações de máquinas, instalações de ferramentas e ou programas, entre outras.

US00c – Estudos e Pesquisas, em grande parte das iterações, alguma atividade de leitura de material, artigos, documentações ou inovações são necessárias visto que os projetos são desenvolvidos em um ambiente acadêmico e de pesquisa.

2.6.2. Reuniões de Planejamento

O líder da equipe tem como função cadastrar a iteração e suas respectivas USs, preferencialmente, antes da reunião. Durante a reunião serão definidas as atividades de cada US, denominadas de *Tasks* e identificadas pela letra "T" seguida por numerais seqüenciais iniciando por "01", espaço, hífen espaço, e um nome significativo para a atividade. Por exemplo: T01 – Nome significativo para a tarefa.

Cada atividade deve possuir um responsável e uma estimativa associada de quantas horas será preciso para completá-la. Mas outros membros da equipe também podem trabalhar na atividade e reportar as horas trabalhadas. Toda a atividade que for desenvolvida em par deve conter o nome dos dois participantes no mesmo período.

O ideal é que cada atividade não exceda um dia de trabalho. Se para a realização de uma atividade forem necessárias muitas horas, o ideal é particioná-la. Cada atividade deve ser planejada pelo executor e conferida pelo líder da linha.

O preenchimento das horas trabalhadas em cada *Task* é de responsabilidade do executor que deve registrar com o maior grau de precisão possível as horas trabalhadas nela. Quanto maior for a precisão no registro das horas trabalhadas na atividade, melhores serão as próximas estimativas, visto que um dos fatores relevantes para estimar o tempo de execução de uma *task* é no histórico de execuções passadas.

2.6.3. Reuniões de Acompanhamento

Essas reuniões ocorrem, normalmente, uma semana depois da reunião de planejamento. Desta forma, o processo conta com reuniões semanais para garantir o bom andamento das atividades. As reuniões servem também para rever as estimativas em relação a cada atividade, ou qualquer outra alteração.

2.6.4. Atividades de Acompanhamento

O acompanhamento das atividades é feito através do gerenciamento de vários fatores como Recursos Humanos (RH), Custos e Riscos. Essas atividades devem ser planejadas durante a reunião de planejamento da iteração, bem como, a estimativa de horas trabalhadas, o controle de aquisições.

Além disso, outra atividade gerencial fundamental é o monitoramento do progresso das tarefas da equipe fundamental para garantir a comunicação interna, proporcionar que todos os membros da equipe conheçam a evolução do trabalho, promover reunião técnicas onde todos os membros de todas as linhas tenham a oportunidade de conhecer o trabalho das outras linhas (artigos da área, pesquisas científicas, relatos de experiência, novas técnicas, e outros). Essas reuniões proporcionam também a integração entre as equipes e a possibilidade de contato com diversas tecnologias.

2.6.5. Atividades de Teste

As atividades de testes são realizadas por ambas as equipes, desenvolvimento e qualificação. As práticas adotadas não se restringem às atividades da equipe de qualificação, pois alguns tipos de testes são de responsabilidade exclusiva dos desenvolvedores, como por exemplo, os testes de unidade.

2.6.5.1. Teste de Aceitação

O teste de aceitação divide-se em duas fases, uma de responsabilidade da equipe de desenvolvimento e outra pela equipe de qualificação.

Primeira Fase

Responsável: Definidos pelo Cliente e implementados pela equipe de desenvolvimento.

Objetivo: Transcrever as necessidades do cliente para requisitos técnicos através das *Users Stories* e transformá-los em testes de aceitação automáticos (*Happy Day*).

Início: Após a identificação dos requisitos solicitados pelo cliente, e a descrição das *Users Stories.* É o primeiro teste que a equipe de desenvolvimento realizará.

Término: Após a identificação de todas as funcionalidades descritas e necessárias de acordo com os requisitos do sistema.

Segunda Fase

Responsável: Equipe de qualificação.

Objetivo: Explodir o teste de aceitação desenvolvido pela equipe de desenvolvimento para cobrir pontos ainda não identificados e/ou testados. O esforço da equipe de qualificação deve ser feito para que cada teste *Happy Day* desenvolvido possa ser explodido para tantos quanto forem possíveis/viáveis de serem desenvolvidos.

Início: Após a entrega dos testes *Happy Day* pela equipe de desenvolvimento.

Término: Após a explosão dos testes de aceitação.

2.6.5.2. TESTE DE UNIDADE

Responsável: Desenvolvedores.

Objetivo: Verificar e validar as condições internas do código referentes a um determinado módulo/programa, isto é, descobrir possíveis erros de codificação lógica, funcional, estrutural nas classes utilizando a ferramenta *JUnit* [JUnit].

Início: Antes da implementação de um determinado módulo/programa.

Término: Quando todos os testes de unidade forem executados sem falhas.

2.6.5.3. TESTE DE INTEGRAÇÃO

Primeira Fase:

Responsável: Equipe de desenvolvimento.

Objetivo: Submeter o código a uma bateria maior de teste para verificar a corretude e completude do código. Compará-los com os requisitos do sistema.

Início: Após o termino dos testes de unidade, no momento em que dois ou mais módulos forem integrados.

*Término:*Após a integração de todas as funcionalidades descritas e necessárias para atender a necessidade do cliente.

Segunda Fase

Responsável: Equipe de qualificação.

Objetivo: Submeter o código a uma bateria maior de testes para verificar a corretude e completude do código. Compará-los com os requisitos do sistema.

Início: Ao término da 1ª fase, aplica-se a mesma prática dos testes de aceitação. Idealmente, para cada teste de integração elaborado pela equipe de desenvolvimento, a equipe de qualificação deve implementar tantos quanto forem os testes de integração possíveis/viáveis devem ser desenvolvidos.

Término: Após a integração de todas as funcionalidades descritas e necessárias para atender a necessidade do cliente.

2.6.5.4. TESTE DE SISTEMAS

Responsável: Ambas as equipes

Objetivo: Qualificar o *software* comparando os requisitos com as funcionalidades associadas.

Início: Após avaliação Final do líder de desenvolvimento.

Término: *Software* atendendo aos requisitos funcionais.

2.6.5.5. TESTE DE REGRESSÃO

Responsável: Idealmente as duas equipes. Caso haja uma bateria de testes de regressão da equipe de desenvolvimento, ela deverá ser executada antes da bateria de testes da equipe de qualificação.

Objetivo: Verificar a integridade do *software*, elaborar novos testes para verificar/validar as modificações ocorridas após um defeito descoberto.

Início: Após a parte do código que apresentou defeito ser corrigida. Caso o teste de regressão já tenha sido utilizado anteriormente, esse pacote de testes deverá ser revisado antes de iniciar o novo teste de regressão.

Término: Quando nenhuma inconformidade (defeito) for detectada (o).

2.6.5.6. TESTE PARA LANÇAMENTO DE VERSÃO (ALFA, FINAL, PATCHS)

Responsável: Dependendo da versão segue a ordem dos responsáveis.

Objetivo: Avaliação do *software* por diferentes usuários, com a finalidade de verificar se todas as suas necessidades estão cobertas pelas funcionalidades implementadas.

Início: Após aprovação do teste de sistema pela equipe de qualificação.

Término: Lançamento da próxima versão na ordem que se segue (Alfa, Final e *Patchs*) conforme Tabela 2.

Tabela 2: Fases do desenvolvimento e seus responsáveis

Fase	Responsável
Elicitação de requisitos	Equipes de desenvolvimento, qualificação e cliente.
Desenvolvimento	Equipes de desenvolvimento, qualificação e cliente.
Testes Exploratórios	*Stakeholders* + convidados.
Alfa	Usuários externos com garantia de uso.

2.7. Os papeis

2.7.1. Cliente

Representante da empresa financiadora dos projetos de natureza acadêmica.

- *Atividades:*
  - Idealizar o produto através da descrição de suas necessidades.
  - Descrever as funcionalidades/requisitos do sistema que são identificadas no processo como USs.
  - Definir requisitos não funcionais.
  - Priorizar as funcionalidades.
  - Descrever, validar e executar os testes de aceitação.
  - Estar disponível para refinar a especificação e realizar reuniões com a equipe de desenvolvimento e *coach*.

O papel do cliente em alguns projetos é difícil de definir. Por isso, algumas vezes um professor, um pesquisador, coordenador ou outra pessoa poderá assumi-lo.

2.7.2. Gerente de Projeto

Pessoa interna ao projeto e a instituição responsável pela gerência de RH, de custos, de aquisições, riscos, garantia de comunicação com o cliente.

- *Atividades:*
    - Acompanhar a execução das atividades e o andamento do plano de trabalho.
    - Revisar o planejamento quando necessário.
    - Emitir relatórios de acompanhamento.
    - Avaliar riscos e lidar com riscos descobertos.
    - Garantir comunicação entre o cliente e os grupos de pesquisa.
    - Manter o cliente informado sobre o andamento do projeto.
    - Acompanhar e gerenciar a execução do projeto e evitar atrasos no planejamento.
    - Negociar ajustes quando necessário.

O Gerente de Projeto ainda assume algumas atividades que poderiam ser de responsabilidade de um Gerente Financeiro.

2.7.3. Pesquisador

Professores, alunos de pós-graduação ou profissionais responsáveis pelo estudo e pesquisa em áreas afins conforme sincronização com as necessidades do cliente.

- *Atividades:*
    - Fundamentar teoricamente as suas áreas de pesquisa.
    - Propor novas soluções, ferramentas, métodos, modelos que possam aprimorar o desenvolvimento e concepção dos projetos.

2.7.4. Consultor

É um especialista da área de projeto que auxilia as equipes com orientações, proposições de novos modelos, ferramentas, técnicas e soluções.

- *Atividades:*
    - Debater junto com líderes das linhas sobre questões que exijam uma fundamentação teórica nas suas áreas de pesquisa.
    - Auxiliar na definição de visões do cliente e tarefas nas linhas de pesquisa.

- Quando requisitado, produzir ou revisar relatórios para a linha.

Em muitos casos os Pesquisadores podem assumir o papel de Consultor.

2.7.5. Coach

Pessoa responsável por coordenar, avaliar, acompanhar o plano de trabalho das equipes, bem como promover melhorias no processo de desenvolvimento.

- *Atividades:*
  - Refinar o processo.
  - Monitorar e reforçar o processo.
  - Participar de reuniões de projeto.
  - Fazer *pair programing*.
  - Organizar seminários sobre estudos relacionados ao desenvolvimento.
  - Conduzir atividades de planejamento e acompanhamento.
  - Fazer levantamento/avaliação das ferramentas necessárias para o desenvolvimento e acompanhamento do projeto.
  - Organizar reuniões rápidas (*Stand Up Meetings*) para desenvolver a comunicação entre membros de equipes diferentes.

O coach ainda assume algumas atividades que poderiam ser de responsabilidade de um Gerente de Produto.

- Elaborar o plano de *releases* do produto, junto ao cliente.
- Controlar as solicitações de mudanças.
- Definir os critérios de qualidade do produto (documentação, usabilidade, robustez).
- Levantar novas características para o produto.
- Fazer a prospecção de novos clientes para o produto.

2.7.6. Líder de Linha

Pessoa responsável por coordenar, avaliar, acompanhar o trabalho da sua equipe.

- *Atividades:*
  - Responder acerca das tarefas que estão sendo desenvolvidas por todos os membros da equipe.

- Verificar se os dados de acompanhamento estão sendo corretamente preenchidos.
- Fazer *pair programing*.
- Fazer *test review*
- Garantir/incentivar a criação de testes de unidade para cada classe criada.

O Líder de linha ainda assume atividades que poderiam ser de responsabilidade de um Gerente de Configuração.

- Manter controle de versão de artefatos para projetos que necessitam.
- Garantir o uso do repositório.
- Controlar a criação de novos *branchs*.
- Gerar *tags* para cada *release* (interna ou externa).
- Definir um *workspace* para o desenvolvimento.
- Coordenar a realização de integração de itens ao repositório.

O Líder de linha pode também assumir as atividades que poderiam ser de responsabilidade de um Gerente de Qualidade.

- Garantir que padrões de codificação e boas práticas de desenvolvimento estão sendo seguidas.
- Garantir que revisões de código e refatoramentos aconteçam periodicamente (idealmente uma vez por semana).

2.7.7. Desenvolvedor:

Pessoas responsáveis pela implementação do produto, bem como pelo desenvolvimento de testes e interfaces.

- *Atividades:*
  - Debater junto com líderes das linhas sobre questões que exijam uma fundamentação teórica nas suas áreas de pesquisa.
  - Auxiliar na definição de visões do cliente e tarefas nas linhas de pesquisa.
  - Quando requisitado, produzir ou revisar relatórios para a linha.
  - Estimar o tempo de desenvolvimento das atividades.
  - Dividir as atividades (visões do cliente) em tarefas.
  - Escolher as tarefas para desenvolvimento de acordo com a priorização do cliente.

- Atualizar e registrar o status de suas atividades diariamente.
- Implementar os produtos de *software*.
- Elaborar a arquitetura e esquema lógico dos dados quando necessário.
- Implementar testes de unidade, aceitação, integração.
- Refatorar o código constantemente.
- Documentar o código conforme a evolução da implementação.
- Desenvolver a fachada para que a equipe de testes possa desenvolver os novos testes e casos de testes.
- Integrar diariamente com o repositório, seguindo os cuidados mínimos exigidos.
    - Ex executar bateria de testes.
- Revisar o código.
    - Fazer *pair programming*.
    - Fazer *test review*
    - Solicitar revisão de seus códigos.

- *Restrições:* Desenvolvedores não devem
    - Inventar "coisas legais" para colocar no *software*. A definição de novas funcionalidades é de responsabilidade do cliente.
    - Decidir a ordem em que as atividades são implementadas, exceto quanto houver dependências entre atividades.
    - Excluir funcionalidades por limitação de tempo. A definição de cortes de funcionalidades é de responsabilidade do cliente.

2.7.8. Testadores:

Pessoas responsáveis por validação e verificação dos produtos de *software* em desenvolvimento e consequentemente responsáveis por garantir a qualidade do *software* produzido.

Existem muitos níveis de testadores, a contratação de pessoas internas para esse papel normalmente têm o perfil de desenvolvedores na condição de testadores.

- Níveis de testadores:

- Testadores internos: Normalmente alunos de graduação contratados como desenvolvedores, em virtude da necessidade de implementação de código na elaboração dos testes automáticos.
- Testadores exploratórios: Pessoas normalmente ligadas ao projeto que em período de validação de versões atuam como testadores com o objetivo de estressarem o produto de *software* desenvolvido.
- Testadores externos: Pode ser o próprio cliente, ou representante, executando os testes de aceitação, como também outros usuários do produto.

- *Atividades:*
  - Revisar/validar a escrita das USs.
  - Revisar/validar os testes de aceitação básicos (*Happy Day*).
  - Elaborar e implementar novos casos de testes automáticos para os produtos em desenvolvimento (atividade de explosão dos testes de aceitação).
  - Refatorar constantemente os testes.
  - Analisar/verificar as assertivas do código.
  - Elaborar testes manuais ou aplicações que verifiquem e validem o *software*.
  - Quando requisitado, produzir ou revisar relatórios para a linha.
  - Estimar o tempo para testar os requisitos do *software*.
    - Disseminar fundamentos, conceitos, experiências, ferramentas sobre as melhores práticas de teste de *software* a serem aplicadas no ciclo de desenvolvimento com objetivo de obter qualidade no *software* produzido.

2.7.9. Tracker

Pessoa responsável por acompanhar o andamento das atividades da equipe.

- *Atividades:*
  - Acompanhar o andamento das atividades garantindo o cumprimento do plano de liberação.

O coach ou o líder de linha podem assumir este papel.

2.7.10. *Test Reviewer;*

Pessoa responsável por acompanhar e revisar o código desenvolvido caso a atividade de pair programming não tenha sido praticada ou em situações onde a criticidade ou complexidade do código é muito alta.

- *Atividades:*
    - Revisar o desenvolvimento de funcionalidades ou de testes com o intuito de validar a implementação.

Pode também assumir o papel de *Design Reviewer* e *Code Reviewer* se desconfiar da qualidade do que foi produzido.

Ciclo de Vida do *Software*

O ciclo de vida do *software* é a junção de todos os ciclos de vida dos *releases*. Para cada *release* o processo de desenvolvimento e qualificação deve ser mantido, todas as atividades dos processos realizadas, ou negociadas com o cliente, todos os artefatos de saída (conjunto de funcionalidades testadas e homologadas) desenvolvidos.

O *software* só será finalizado quando o cliente aprovar a versão integrada de todas as funcionalidades desenvolvidas.

2.8. As Mudanças

O processo proposto, o *OurQualityProcess*, consistiu em um conjunto de mudanças buscando a melhoria do processo e, consequentemente, a melhoria dos produtos finais.

Para simplificar o entendimento e possibilitar uma avaliação efetiva, um resumo com todas as alterações propostas estão organizadas nessa seção através de uma categorização, são elas:

ATIVIDADES:

Incluídas:

- Revisões/validações dos requisitos (USs);
- Todas as atividades relacionadas à equipe de teste;
- Validação das soluções/correções das *issues*;
- Revisões de código que não foram desenvolvidos em *pair programming*;
- Maximizar a automatização de testes manuais.

Alteradas:

- desenvolvimento de testes – anteriormente a responsabilidade era exclusivamente dos desenvolvedores e, no novo processo, a responsabilidade também se aplica aos testadores;
- Divisão do *build*, ao invés de um build adotou-se três tipos de *builds* (unidade, integração e *night build*);
- Para cada teste executado pela equipe de desenvolvimento (*HappyDay*) um ou mais casos de teste deveriam ser desenvolvidos;

ARTEFATOS:

Incluídos:

- testes de aceitação explodidos;
- testes de integração explodidos;
- testes de sistema explodidos;
- Relatórios automáticos diários com laudo sobre a saúde do código (código saudável = código sem falhas).

Alterados:

- Padronização da escrita dos requisitos (USs);
- Bateria de testes com duas modalidades, testes de desenvolvimento e testes de qualificação.
- Geração de versões Alfa, Final e *Paths*.

FERRAMENTAS:

Incluídas/Adotadas:

- Bamboo (integração);
- Jira (repositório de *issues* sobre o projeto);
- Metrics (ferramenta para coleta de métricas);
- JDepend (ferramenta de análise de dependência de pacotes e gerador de métricas sobre a qualidade do "Design" para cada package Java)

- CodeProActiveX (ferramenta de análise de código, auxilia a geração automática de testes unidade, recursos de coleta de métricas);
- Clover (verifica a cobertura dos testes);
- FindBugs (análise estática de código);

Alterados:

- Padronização da escrita dos requisitos (USs);
- Bateria de testes com duas modalidades, testes de desenvolvimento e testes de qualificação.
- Geração de versões Alfa, Final e *Paths*.

MÉTODOS:

Incluídos:

- testes de aceitação explodidos;
- testes de integração explodidos;
- testes de sistema explodidos;
- adoção de DBC;
- Relatórios automáticos diários com laudo sobre a saúde do código (código saudável = código sem falhas).

Alterados:

- Padronização da escrita dos requisitos (USs);
- Bateria de testes com duas modalidades, testes de desenvolvimento e testes de qualificação.
- Geração de versões Alfa, Final e *Paths*.

Travassos [Travassos, 2002] orienta que sempre que novos métodos, técnicas, linguagens e ferramentas são sugeridas e/ou inventadas deve-se utilizar o processo científico experimental. Com esse processo é possível obter um modo sistemático, disciplinado, computável e controlado para avaliação da atividade humana.

Sua afirmação vai além, ele diz que : "*é preciso avaliar novas inversões e sugestões em comparação com as existentes para que se tenha validade nas produções*". [Travassos,

2002]. Complementando essa ideia, Zelkowitz garante que "*adotar novas tecnologias sem evidências relacionadas à efetividade pode resultar no retrocesso da qualidade*" [Zelkowitz, 2003].

Capítulo 3

3. Avaliação do processo proposto - *OurQualityProcess*

3.1. Principais assuntos do capítulo

Estudos Experimentais

Roteiro do Estudo

Preparação

Execução

Avaliação de acordo com o *Baseline*

Análise

Empacotamento

Ameaças à Validade

3.2. Estudos Experimentais

A área da Ciência da Computação está inserida em dois universos, o da ciência e o da engenharia. Nessa junção de áreas alguns princípios de uma sobrepõem os da outra, com isso muitos pesquisadores se questionam sobre os métodos que cientistas da computação fazem ciência.

Um trabalho que reflete essa questão foi apresentado por Peter Denning em que ele questiona a ciência que os cientistas da computação estão fazendo e afirma ainda que exista uma incredibilidade na área, inclusive perante cientistas da própria computação. Uma das questões levantadas no artigo é: "*como é possível fazer ciência sem seguir um método científico para validar os desenvolvimentos*?" [Denning, 2005].

Mais crítico que a questão se a ciência da computação é ciência é definir se a Engenharia de *Software* é ciência. Nesse paradigma um movimento vem ganhando espaço

para validar os resultados dos engenheiros de *software* no meio acadêmico-científico, esse movimento é a Engenharia de *software* Experimental.

Para definir a Engenharia de *Software* Experimental necessita-se definir o que é um experimento. Segundo Kerlinger "*Um experimento é um tipo de pesquisa científica na qual um pesquisador manipula e controla uma ou mais variáveis independentes e observa a variação na variável ou variáveis dependentes concomitantemente à manipulação das variáveis independentes*". [Kerlinger, 1979].

Portanto a Engenharia Experimental associa o processo de criação de produtos de *software* com a melhoria contínua através da manipulação e controle das variáveis do processo.

Os processos de desenvolvimento são de natureza não-determinística e possuem muitas variáveis que precisam ser observadas. Assim, recomenda-se caracterizar o contexto do problema através da observação de variáveis e do registro de possíveis explicações para os comportamentos observados; testar as hipóteses na prática, registrando probabilidades, novos valores e grau de importância de variáveis, incluindo novas variáveis e excluindo outras; e, por fim, realizar análises com o objetivo de reduzir incertezas. [Pfeeger, 1999].

Dependendo do propósito da análise e das condições para a investigação, quatro tipos de estratégias podem ser aplicadas: Análise das características, *Surveys*, Estudos de Caso e Experimentos [Pfeeger, 1999; Wohlin, 2000].

Análise das características: é o método de investigação mais simples, que através de uma abordagem subjetiva se propõem a agrupar, selecionar e sumarizar as características de um conjunto de fatores com o objetivo de classificar e definir qual fator utilizar. Uma importante consideração sobre essa abordagem de investigação é que ela corresponde ao estudo de acontecimentos passados, onde não há a possibilidade de alteração dos dados, o que ocorre é uma observação de acontecimentos passados e, portanto, não é possível considerar o comportamento de causa e efeito.

Survey: também conhecida como pesquisa de opinião, é um instrumento abrangente para coletar informação, descrever, comparar ou elucidar conhecimento, atitudes e comportamento [Pfleerger, 1999; Kitchenham, 2002a]. A investigação do tipo *survey* tem o

objetivo de descrever um fenômeno de interesse ou avaliar o impacto de alguma intervenção. Quando o propósito do *survey* se destina a descrever um fenômeno ele é utilizado para determinar a distribuição de características, especificamente, identificar qual é a distribuição e não se preocupar com o porquê. Quando o *survey* é utilizado para avaliar um determinado impacto de uma intervenção, o propósito é avaliar o efeito de tecnologias antes de serem usadas ou na implantação [Kitchenham, 2002a; Wohlin, 2000]. Uma importante constatação é que semelhante a análise de características, não há manipulação dos dados, apenas análise dos resultados obtidos.

Estudo de Caso: esta abordagem não prevê manipulação, é uma técnica observacional, onde há uma monitoração e dados são coletados durante a execução do caso sob estudo. Uma desvantagem dessa abordagem é a dificuldade que há em: generalizar os resultados obtidos [Wohlin, 2000]; determinar tendências e prover uma validade estatística devido à especificidade de cada projeto [Zelkowitz, 1998].

No trabalho de Kitchenham, [Kitchenham, 1995], ele fornece um guia que consiste na descrição de sete passos para a realização de estudos de caso, com esse guia é possível identificar em quais condições os estudos de caso são mais adequados e como a abordagem do estudo de casos deve ser conduzida de forma a prover mais rigor no decorrer da investigação. Os passos que Kitchenham propõem são:

1. Definir as hipóteses do estudo;
2. Selecionar projetos-piloto;
3. Identificar o método de comparação;
4. Minimizar os efeitos dos fatores de confusão;
5. Planejar o estudo de caso;
6. Monitorar o estudo de caso conforme o plano;
7. Analisar e reportar os resultados.

Em relação ao 4º passo é que no estudo de caso é grande a possibilidade de existir fatores de confusão (*confounding factors*), onde o analisador terá dificuldade de distinguir a influências de determinados fatores no resultado. Em muitos casos o estudo de caso é utilizado em parte da investigação e outros métodos são incorporados para maximizar a generalização dos dados encontrados.

Segundo Travassos existem três maneiras de elaborar o estudo de caso, são elas:

1) **Comparação dos resultados**: Utilizar um método novo e comparar com um *baseline* da mesma área;
2) **Projetos semelhantes**: Comparação dos resultados de projetos similares;
3) **Método de atribuição**: Atribuir um método a um projeto da empresa e não atribuir aos outros. [Travassos, 2002].

Experimento: De acordo com [Wohlin, 2000] "*Um experimento é uma investigação formal, rigorosa e controlada*". Por ser controlado, normalmente, ocorrem em ambiente que proporcionam uma estrutura para possibilitar a manipulação de um ou um conjunto de fatores e o controle de outros, de acordo com os propósitos do estudo. O controle deve-se estender para a coleta e análise dos dados para obter um resultado estatisticamente significativo [Zelkowitz, 2003].

Os principais elementos de um experimento são:

Atributos: Fatores manipulados ou variáveis independentes do estudo;

Tratamentos: A manipulação efetuada em um determinado atributo;

Variáveis Ambientais: Influência exercida pelos atributos em um determinado tratamento;

Fatores de Confusão: Influência dos outros atributos;

Variáveis dependentes: variáveis de resposta, responsáveis por representar os efeitos obtidos com a experiência.

Para realizar um experimento Wohlin propõem um processo de execução que, posteriormente, foi estendido por Amaral. [Wohlin, 2000; Amaral, 2003]. Na versão final o processo é composto por cinco fases seqüenciais e uma fase paralela de empacotamento, são elas:

1) **Definição:** Definir o contexto, os objetivos o problema e hipóteses.
2) **Planejamento e Design:** Definir o contexto, hipóteses, variáveis, tratamentos, instrumentação e validade.

3) **Preparação:** Preparar participantes, treinamentos, ferramental, configurações, ambiente.
4) **Execução:** Seguir o design, realizar as medições, coletar resultados.
5) **Análise e Interpretação:** análise e avaliação dos resultados obtidos
6) **Empacotamento:** Apresentação, empacotamento visando a repetição.

Independente do estudo experimental aplicado, o que se pretende é identificar/definir os fatores de causa e efeito, isto é, tratamentos e resultados. Para a aplicação de um estudo experimental é fundamental que se defina as variáveis do estudo (dependentes e independentes), os objetos sob estudo, os participantes, o contexto, as hipóteses (nula e alternativas) e o projeto de estudo.

Durante a realização de um estudo experimental é necessário também respeitar alguns princípios que são: aleatoriedade, agrupamento e balanceamento, além de outros definidos de acordo com a característica do estudo.

Para validar um do experimento é necessário medir e validar os resultados. Em relação às métricas existem quatro tipos de escalas: ordinal, nominal, intervalar e razão. Essas escalas são necessárias para medir os parâmetros e, consequentemente, possibilitar a verificação das hipóteses. Em relação à validação dos resultados existem quatro tipos considerados de importância científica, são eles: validação de conclusão (análise estática), interna (tratamento versus resultado), externa (generalização dos resultados) e de construção (teoria versus observação).

3.3. Roteiro do Estudo Experimental:

3.3.1. Definição

1) **Tema:** Processo de desenvolvimento de *software*.
2) **Áreas:** Engenharia de *Software*, Processo de Desenvolvimento de *Software*.
3) **O problema:** Definir qual processo resulta em produtos com maior qualidade.
4) **A importância do problema:** Aperfeiçoar os produtos desenvolvidos, aumentar o entendimento sobre os produtos, melhorar o processo de desenvolvimento. De forma geral, garantir o controle de qualidade na avaliação de produtividade da equipe de desenvolvimento e garantir a satisfação dos clientes.

5) **O objetivo:** Analisar dois processos de desenvolvimento de *software* baseados em metodologia ágil, estabelecer índices de qualidade e, baseando-se nestes índices, identificar o processo que resulta em produtos com o maior nível de qualidade.
6) **Hipótese Nula:** Os processos geram produtos de qualidade semelhantes.
7) **Hipótese Alternativa:** Os processos não geram produtos de qualidade semelhantes, isso é um processo é melhor que outro.

3.3.2. Planejamento e Design

- Definir dois processos de desenvolvimento de *software*:
  - *OurProcess*;
  - *OurQualityProcess*;
- Estabelecer fases durante o processo de desenvolvimento para a coleta dos dados relacionados à qualidade do *software*;
  - Fase 1: Elaboração de testes automáticos;
  - Fase 2: Verificação das correções;
  - Fase 3: Elaboração de testes exploratórios;
  - Fase 4: Alfa
- Coletar os dados - instrumentação;
- Analisar os resultados obtidos;
  - Validade de conclusão
  - Validade interna
  - Validade Externa
  - Validade de construção
- Validação Comparativa
  - Validação comparativa com *baseline* desenvolvido.

3.3.2.1. INSTRUMENTAÇÃO

Para a coleta, os dados de tempo foram obtidos através da ferramenta *XPlanner* [XPlanner], uma ferramenta para gerenciamento de projetos baseados em XP. As medidas de tempo são contabilizadas através do número de dias decorridos entre a data de início das tarefas, (de uma atividade ou de todo o projeto) até a data na qual todas as tarefas da *US*

foram fechadas. Em relação ao esforço, a medida coletada foi de *homens/hora*, isso é, um *homem/hora* é o tempo de uma hora empregado por um membro da equipe do projeto. Assim, o esforço é o somatório do número total de horas empregadas por todos os membros da equipe na execução de uma tarefa ou de todo o projeto [Carleton,1992].

Para a coleta dos *bugs* detectados usou-se a ferramenta *Jira* [Jira], uma ferramenta para acompanhamento e gestão de problemas desenvolvida pela *Atlassian Software Systems* [Atlassian]. A cobertura do código foi coletada através da ferramenta *Clover* [Clover], também desenvolvido pela *Atlassian Software Systems*. Os testes automáticos foram coletados através do *Metrics* [Metrics], uma ferramenta para coleta de métricas, *CodePro AnalytiX* [CodePro], *JUnit* [JUnit] e para análise da Versão foi utilizada *Integrated Development Environment* (IDE) – Eclipse [Eclipse].

3.3.2.2. FORMULAÇÃO DE HIPÓTESES

H_0 – **A hipótese nula:** A qualidade do *software* que segue o processo *OurProcess* é igual à qualidade do *software* que segue o processo *OurQualityProcess*.

H_1 – **A hipótese alternativa:** A qualidade de um *software* com o *OurProcess* é diferente do *OurQualityProcess*.

Para que seja possível avaliar a melhoria do processo um conjunto de atributos de qualidade foi definido. Esses atributos serão mensurados separadamente e a hipótese nula definida, anteriormente, será subdividida em dez hipóteses secundárias agrupadas de acordo com os objetivos definidos de acordo com a metodologia GQM.

O padrão para a nomenclatura das hipóteses secundárias será:

(H_{sn}) = Enésima hipótese secundária

(obj) = J-ésimo objetivo analisado

"0" e "1" = hipótese nula e a alternativa, respectivamente.

HIPÓTESES SECUNDÁRIAS

Objetivo 1:

$H_{S1\text{-}ob1\_0}$: A precisão total do cronograma do *OurProcess* é igual à precisão total do cronograma do *OurQualityProcess*.

$H_{S1\text{-}ob1\_1}$: A precisão total do cronograma do *OurProcess* é menor que a precisão total do cronograma do *OurQualityProcess*.

$H_{S2\text{-}ob1\_0}$: A precisão total do cronograma de teste do *OurProcess* é igual à precisão total do cronograma de teste do *OurQualityProcess*.

$H_{S2\text{-}ob1\_1}$: A precisão total do cronograma de teste do *OurProcess* é menor que a precisão total do cronograma de teste do *OurQualityProcess*.

$H_{S3\text{-}ob1\_0}$: A precisão total do esforço do *OurProcess* é igual à precisão total do esforço do *OurQualityProcess*.

$H_{S3\text{-}ob1\_1}$: A precisão total do esforço do *OurProcess* é menor que a precisão total do esforço do *OurQualityProcess*.

$H_{S4\text{-}ob1\_0}$: A precisão total do esforço de teste do *OurProcess* é igual à precisão total do esforço de teste do *OurQualityProcess*.

$H_{S4\text{-}ob1\_1}$: A precisão total do esforço de teste do *OurProcess* é menor que a precisão total do esforço de teste do *OurQualityProcess*.

Objetivo 2:

$H_{S1\text{-}ob2\_0}$: O número de *bugs* reportados no *OurProcess* é igual ao número de *bugs* reportados no *OurQualityProcess*.

$H_{S1\text{-}ob2\_1}$: O número de *bugs* reportados no *OurProcess* é menor que o número de *bugs* reportados no *OurQualityProcess*.

$H_{S2\text{-}ob2\_0}$: A cobertura dos testes no *OurProcess* é igual à cobertura dos testes no *OurQualityProcess*.

$H_{S2\text{-}ob2\_1}$: A cobertura dos testes no *OurProcess* é menor que cobertura dos testes no *OurQualityProcess*.

$H_{S3\text{-}ob2\_0}$: O número de testes desenvolvidos no *OurProcess* é igual ao número de testes desenvolvidos no *OurQualityProcess*.

$H_{S3\text{-}ob2\_1}$: O número de testes desenvolvidos no *OurProcess* é menor que o número de testes desenvolvidos no *OurQualityProcess*.

Objetivo 3:

$H_{S1\text{-}ob3\_0}$: O número de defeitos detectados por linha de código no *OurProcess* é igual ao número de defeitos detectados por linha de código no *OurQualityProcess*.

$H_{S1\text{-}ob3\_1}$: O número de defeitos detectados por linha de código no *OurProcess* é menor que o número de defeitos detectados por linha de código no *OurQualityProcess*.

$H_{S2\text{-}ob3\_0}$: O número de *issues* reportadas por linha de código no *OurProcess* é igual ao número de defeitos detectados por linha de código no *OurQualityProcess*.

$H_{S2\text{-}ob3\_1}$: O número de *issues* reportadas por linha de código no *OurProcess* é menor que o número de defeitos detectados por linha de código no *OurQualityProcess*.

$H_{S3\text{-}ob3\_0}$: O percentual de *issues* solucionadas no *OurProcess* é igual ao percentual de *issues* solucionadas no *OurQualityProcess*.

$H_{S3\text{-}ob3\_1}$: O percentual de *issues* solucionadas no *OurProcess* é menor que o percentual de *issues* solucionadas no *OurQualityProcess*.

3.3.2.3. SELEÇÃO DE VARIÁVEIS

Seleção das variáveis independentes: *OurProcess* (OP) e *OurQualityProcess* (OQP).

Seleção das variáveis dependentes:

Precisão total do cronograma do *OP*, Precisão total do cronograma do *OQP*, Precisão total do esforço do *OP*, Precisão total do esforço do *OQP*, Precisão total do cronograma na fase de testes do *OP*, Precisão total do cronograma na fase de testes do *OQP*, Precisão total do esforço na fase de testes do *OP*, Precisão total do esforço na fase de testes do *OQP*, Qualidade da detecção de falhas por linhas de código no *OP*, Qualidade da detecção de falhas por linhas de código no *OQP*, Qualidade na resolução de problemas no *OP*, Qualidade na resolução de problemas no *OQP*.

3.3.2.4. SELEÇÃO DE UNIDADES EXPERIMENTAIS

OurProcess e *OurQualityPocess*.

3.3.2.5. DESIGN DOS EXPERIMENTOS

Design Fatorial Completo

Tabela 3: Número de Experimentos x Fase x Tipo de Processo.

# Experimento	Fase do Desenvolvimento	Processo de Qualificação
1	1	*OP*
2	1	*OQP*
3	2	*OP*
4	2	*OQP*
5	3	*OP*
6	3	*OQP*
7	4	*OP*
8	4	*OQP*

Fases do processo de desenvolvimento:

- Fase 1: Elaboração de testes automáticos;
- Fase 2: Verificação das correções;
- Fase 3: Elaboração de testes exploratórios;
- Fase 4: Alfa

3.3.3. Preparação

Para realizar um experimento alguns preparativos devem ser cautelosamente analisados antes e durante a execução do experimento:

- Estar ciente do estudo e possuir ou receber treinamentos adequados para execução das atividades;
    - Todos os envolvidos no estudo estavam cientes da implantação do novo processo, bem como preparados para executar todas as atividades do processo, não houve a necessidade de treinamentos.

- Estudo sobre ferramentas, métodos, técnicas, práticas definidas no planejamento e instrumentação;
    - Todos os envolvidos no estudo estavam preparados e familiarizados com as ferramentas, os métodos, as técnicas e as práticas para execução do novo processo.
- Configuração do ambiente;
    - Houve a necessidade de preparar o ambiente com a instalação das ferramentas definidas na instrumentação tanto nas máquinas utilizada pelos desenvolvedores, quando nas máquinas dos testadores. Foi necessário também configurar o servidor.
- Preparar o cenário com a maior fidelidade possível para os dois processos. Manter a mesma equipe de desenvolvimento e testadores.

3.3.4. Execução

3.3.4.1. METODOLOGIA

Para a execução de um processo experimental válido, alguns princípios são fundamentais: organização, controle, acompanhamento, medições, análise e interpretação dos dados. Visando facilitar a elaboração dos experimentos, várias metodologias foram desenvolvidas, cada uma com suas peculiaridades em relação às fases, aos objetivos, às ferramentas de empacotamento, às métricas, entre outras diferenças [Travassos, 2002].

Um exemplo de metodologia de experimentação avançada é o *Quality Improvement Paradigm* (QIP) onde o foco é na melhoria dos processos de desenvolvimento de *software* [Brasili, 1994].

Uma abordagem que estende a QIP através de um modelo baseado em camadas para oferecer o processo da melhoria do desenvolvimento através de um modelo *top-down* de medição é a metodologia *Goal Question Metrics* (GQM). A essência dessa abordagem pode ser descrita através de um processo com três fases - Figura 2 - onde é necessário definir os **objetivos** (1) do desenvolvimento, identificados através de **questões** (2) que possam se mensuradas através de um conjunto de **métricas** (3). Para realizar a interpretação segue-se o inverso do modelo, isto é, a abordagem *botton-up* onde a partir das **métricas** (3) de

software é possível responder as **questões** (2) para cada **objetivo** (1) definido, resultando na demonstração do grau de sucesso alcançado.

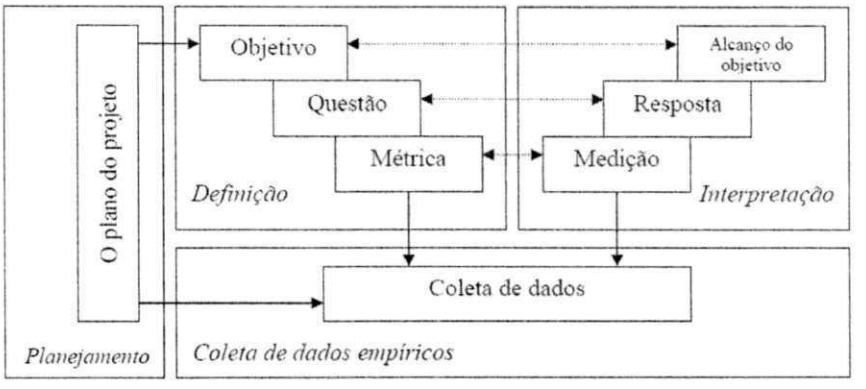


Figura 2: Processo GQM [Travassos, 2002].

A metodologia foi desenvolvida de acordo com os seguintes passos:

1. Identificação dos problemas enfrentados pela equipe de desenvolvimento sob a ótica dos clientes e gerentes de desenvolvimento;
2. Identificação dos objetivos a partir da metodologia GQM, onde a solução de cada problema identificado foi descrita na forma de objetivos;
3. Identificação das questões que englobem a totalidade dos objetivos;
4. Identificação das métricas para cada questão (salientando que uma questão pode necessitar de uma ou mais métricas);
5. Instrumentação e coleta de métricas para o Processo *OurProcess* e para o Processo *OurQualityProcess*;
6. Avaliação e Interpretação dos resultados;
7. Conclusões.

3.3.4.2. Identificação dos Problemas, Questões e Métricas

Para utilizar a abordagem GQM foi necessário identificar quais os principais problemas enfrentados no desenvolvimento dos nossos produtos e relacionar cada problema com um objetivo a ser alcançado.

A partir de entrevistas realizadas com os líderes de cada linha de desenvolvimento e análise do repositório de *issues* foi possível constatar que a falta de precisão nas estimativas, o alto índice de defeitos encontrados após o lançamento das versões e a baixa

qualidade observada nos produtos liberados ocasionada pela baixa cobertura dos testes eram os principais problemas encontrados no ambiente utilizado no estudo de caso.

Desta forma, foram definidos três objetivos para melhorar a qualidade do processo de desenvolvimento como sendo: (1) melhorar a precisão das estimativas de projeto, (2) aumentar a detecção de defeitos e/ou falha (*bug*) antes da liberação e, finalmente, (3) aumentar a qualidade dos produtos desenvolvidos.

Para cada objetivo traçado, foram definidas também as questões que, ao serem respondidas, direcionariam a uma análise que identificaria se as metas estipuladas foram devidamente alcançadas. As métricas de cada uma das questões consideram tanto os valores de todo o projeto quanto os valores de cada *User Story* – US (nomenclatura utilizada conforme a proposta adotada pela metodologia XP). De acordo com a Tabela 4 é possível visualizar as respectivas questões e métricas para cada objetivo definido.

De acordo com Gomes [Gomes, 2001], as questões relativas à melhoria das estimativas que caracterizam o problema são saber (1) "qual a precisão da estimativa de cronograma" e (2) "qual a precisão da estimativa de esforço". Neste experimento foram utilizadas as mesmas questões propostas.

Tabela 4: Primeiro objetivo: Questões e métricas segundo abordagem GQM.

<table>
<tr><td colspan="4">Objetivo 1: Melhorar precisão das estimativas de projeto</td></tr>
<tr><td colspan="4">Questão 1.1: Qual a precisão das estimativas de cronograma dos projetos de desenvolvimento e testes por processo?</td></tr>
<tr><td>Métrica 1.1.a.a
$P_{Total\_OP} = \frac{T_{E\_OP}}{T_{R\_OP}}$</td><td>Métrica 1.1.a.b
$P_{Total\_OQP} = \frac{T_{E\_OQP}}{T_{R\_OQP}}$</td><td>Métrica 1.1.b.a
$P_{TotalT\_OP} = \frac{T_{TE\_OP}}{T_{TR\_OP}}$</td><td>Métrica 1.1.b.b
$P_{TotalT\_OQP} = \frac{T_{TE\_OQP}}{T_{TR\_OQP}}$</td></tr>
<tr><td colspan="4">Questão 1.2: Qual a precisão das estimativas de esforço dos projetos de desenvolvimento e testes por processo?</td></tr>
<tr><td>Métrica 1.2.a.a
$P_{TotalEsf\_OP} = \frac{Esf_{E\_OP}}{Esf_{R\_OP}}$</td><td>Métrica 1.2.a.b
$P_{TotalEsf\_OQP} = \frac{Esf_{E\_OQP}}{Esf_{R\_OQP}}$</td><td>Métrica 1.2.b.a
$P_{EsfT\_OP} = \frac{Esf_{TE\_OP}}{Esf_{TR\_OP}}$</td><td>Métrica 1.2.b.b
$P_{EsfT\_OQP} = \frac{Esf_{TE\_OQP}}{Esf_{TR\_OQP}}$</td></tr>
</table>

Onde:
Para métrica 1.1.a.a tem-se:

- P_{Total_OP} = precisão total do cronograma *OurProcess*;

- T_{E_OP} = tempo estimado *OurProcess*;
- T_{R_OP} = tempo real *OurProcess*.

Para métrica 1.1.a.b tem-se:

- P_{Total_OQP} = precisão total do cronograma do *OQP;*
- T_{E_OQP} = tempo estimado do *OQP*;
- T_{R_OQP} = tempo real do *OQP*.

Para métrica 1.1.b.a tem-se:

- P_{TotalT_OP} = precisão do cronograma do projeto de testes do *OurProcess*;
- T_{TE_OP} = tempo estimado do projeto de testes do *OurProcess*;
- T_{TR_OP} = tempo real do projeto de testes do *OurProcess*.

Para métrica 1.1.b.b depois se tem:

- P_{TotalT_OQP} =precisão do cronograma do projeto de testes do *OQP*;
- T_{TE_OQP} = tempo estimado do projeto de testes do *OQP*;
- T_{TR_OQP} = tempo real do projeto de testes do *OQP*.

Para métrica 1.2.a.a tem-se:

- $P_{TotalEsf\_OP}$ = precisão total do esforço do *OurProcess*;
- Esf_{E_OP} = esforço estimado *OurProcess*;
- Esf_{R_OP} = esforço real *OurProcess*.

Para métrica 1.2.a.b tem-se:

- $P_{TotalEsf\_OQP}$ = precisão total do esforço do *OQP*;
- Esf_{E_OQP} = esforço estimado do *OQP*;
- Esf_{R_OQP} = esforço real do *OQP*.

Para a métrica 1.2.b.a tem-se:

- P_{EsfT_OP} = precisão total do esforço estimada do projeto de testes do *OurProcess;*
- Esf_{TE_OP} = esforço estimado do projeto de testes do *OurProcess*;
- Esf_{TR_OP} = esforço real do projeto de testes do *OurProcess*.

Para a métrica 1.2.b.b tem-se:

- P_{EsfT_OQP} = precisão total do esforço estimada do projeto de testes do *OQP*;
- Esf_{TE_OQP} = esforço estimado do projeto de testes do *OQP*;
- Esf_{TR_OQP} = esforço real do projeto de testes do *OQP*.

Para o segundo objetivo, foram propostas três questões, (1) "qual o aumento na detecção de *bugs* (*bugs* será considerado neste trabalho como defeitos e/ou falhas)?", (2) "qual o aumento na cobertura dos testes?" e (3) "qual o aumento na quantidade de testes desenvolvidos?", conforme Tabela 5.

Tabela 5 – Segundo objetivo: Questões e métricas segundo abordagem GQM.

Objetivo 2: Aumentar a detecção de *bugs* antes da liberação	
Questão 2.1: Qual o aumento na detecção de *bugs*? (%)	
Métrica 2.1.a OP_{Bugs}	**Métrica 2.1.b** OQP_{Bugs}
Questão 2.2: Qual o aumento na cobertura dos testes? (%)	
Métrica 2.2.a OP_{Cob}	**Métrica 2.2.b** OQP_{Cob}
Questão 2.3: Qual o aumento na quantidade de testes desenvolvidos? (%)	
Métrica 2.3.a OP_{Test}	**Métrica 2.3.b** OQP_{Test}

Onde:
Para métrica 2.1.a e 2.1.b tem-se:

- OP_{Bugs} = número de *bugs* reportados no *OurProcess*;
- OQP_{Bugs} = número de *bugs* reportados no *OQP*.

Para métrica 2.2.a e 2.2.b tem-se:

- OP_{Cob} = cobertura dos testes no *OurProcess*;
- OQP_{Cob} = cobertura dos teste no *OQP*.

Para métrica 2.3.a e 2.3.b tem-se:

- OP_{Test} = número de testes produzidos no *OurProcess*;
- OQP_{Test} = número de testes produzidos no *OQP*.

O terceiro objetivo é o mais complexo de ser avaliado, pois medir qualidade envolve diferentes aspectos e caso algum desses aspectos for desprezado ou supervalorizado, toda a avaliação pode ser comprometida. Neste contexto teremos uma ameaça à validade do experimento. De acordo com o trabalho desenvolvido por [Brasili, 2002] para avaliar a qualidade de um determinado produto é necessário identificar quais os aspectos de qualidade que estamos interessados. Fenton [Fenton, 2009] defende a teoria de qualidade no software é atender os requisitos para obtenção da satisfação do cliente enquanto Brady [Brady, 2010] complementa que para um produto de *software* ter qualidade

é preciso minimizar a soma das vaiáveis esforço, tempo de desenvolvimento e quantidade de defeitos.

Através da combinação das abordagens de Brasili, Fenton e Brady [Basili, 2002; Fenton, 2009; Brady, 2010] utilizou-se como variáveis para medir a satisfação: o número de *issues* resolvidas, em relação às métricas de tempo e esforço foram consideradas anteriormente, no objetivo 1 e o número de *bugs* detectados também foi considerado de modo expandido no objetivo 3. Portanto para mensurar a qualidade dos objetivos é necessário avaliar todos os objetivos conjuntamente. No objetivo 3 serão considerados os complementos das variáveis necessárias para avaliar o aumento da qualidade.

Os objetivos complementares para avaliação da qualidade foram: (1) quantos defeitos foram detectados por linhas de código antes e depois do processo de qualificação? e (2) qual a qualidade de *issues* fechadas antes e depois do processo de qualificação? (Tabela 6).

Tabela 6 - Terceiro objetivo: Questões e métricas segundo abordagem GQM.

Objetivo 3: Aumentar a qualidade dos produtos desenvolvidos	
Questão 3.1: Quantos defeitos foram detectados por linhas de código por processo?	
Métrica 3.1.a $Q_{Total\_OP} = \frac{Issues_{OP}}{LOC_{OP}}$	**Métrica 3.1.b** $Q_{Total\_OQP} = \frac{Issues_{OQP}}{LOC_{OQP}}$
Questão 3.2: Qual a qualidade de *issues* fechadas por processo?	
Métrica 3.2.a $P_{resolv\_OP} = \frac{Issues_{Fehadas\_OP}}{Issues_{Abertas\_OP}}$	**Métrica 3.2.b** $P_{resolv\_OQP} = \frac{Issues_{Fehadas\_OQP}}{Issues_{Abertas\_OQP}}$

Onde:
Para métrica 3.1.a tem-se:

- Q_{Total_OP} = quantidade de defeitos reportados no *OurProcess* por linha de código;
- $Issues_{OP}$ = número de *issues* detectadas no *OurProcess*;
- LOC_{OP} = número de linhas de código no *OurProcess*.

Para métrica 3.1.b tem-se:

- Q_{Total_OQP} = quantidade de defeitos reportados no *OQP* por linha de código;
- $Issues_{OQP}$ = número de *issues* detectadas no *OQP*;
- LOC_{OQP} = número de linhas de código no *OQP*.

Para métrica 3.2.a tem-se:

- P_{resolv_OP} = percentual de *issues* no *OurProcess* solucionadas;
- $Issues_{Fechadas\_OP}$ = número de *issues* fechadas no *OurProcess*;
- $Issues_{Abertas\_OP}$ = número de *issues* detectadas no *OurProcess*.

Para métrica 3.2.a tem-se:

- P_{resolv_OQP} = percentual de *issues* no *OQP* solucionadas;
- $Issues_{Fechadas\_OQP}$ = número de *issues* fechadas no *OQP*;
- $Issues_{Abertas\_OQP}$ = número de *issues* detectadas no *OQP*.

Em relação ao tamanho do sistema, representada através do número de linhas de código, serão desconsideradas as linhas em branco e linhas contendo apenas comentários, conforme sugerido por Carleton [Carleton, 1992]. Além disso, para garantir a consistência entre diferentes medições e evitar que divergências apareçam devido a formas distintas de contagem, foi adotado como ferramenta para coleta de métricas a ferramenta *CodeProAnalytiX* em todas as medições referentes ao tamanho do sistema. Para que não haja divergência no método de coleta da métrica, visto que há diferença na contagem entre a ferramenta *CodeProAnalytiX* e a ferramenta *Metrics*, também utilizada para coleta de outras métricas.

3.3.4.3. RESULTADOS

Todos os objetivos foram avaliados de acordo com a abordagem proposta na seção anterior, para cada métrica consideramos a média dos valores obtidos nas quatro fases analisadas (Elaborar testes automáticos, verificar correções, elaborar testes exploratórios e Alfa). Para cada objetivo, será apresentada uma tabela com os valores médios obtidos e, sequencialmente, os testes estatísticos referentes para cada par de valores.

A comparação será realizada para verificar se há evidências de diferenças entre os processos *OurProcess* e *OurQualityProcess* através da comparação de cada par de métricas.

Para isso será utilizado um teste não paramétrico para comparar amostras pareadas. Aplicaremos o teste de Wilcoxon, isso porque não é possível garantir que as amostras

tenham uma distribuição normal e como o número de amostras não é superior a 30 (onde a distribuição aproxima-se de uma normal) o Teste T (paramétrico) não pode ser realizado.

Com o teste de Wilcoxon é possível verificar a magnitude da diferença dos pares, desde que a amostra tenha uma distribuição simétrica. Não será possível utilizar o teste para todos os atributos de qualidade definidos, isso porque é necessário utilizar uma amostra de, no mínimo, três repetições, o que não foi possível coletar no *OurProcess*.

O ambiente dos dois processos foi o mais semelhante possível. Em relação a equipe, foi introduzidas três pessoas para comporem o time de qualificação. O *software* desenvolvido com o OP foi o *OurBackup Home* (Sistema de backup P2P através de uma rede social) e com o OQP foi o *OurBachup Enterprise* (Sistema de backup P2P para uma organização) isso porque ambos *software* tem os mesmos requisitos funcionais aplicados a diferentes alvos (rede social x organização).

A complexidade do OurBackup Enterprise é relativamente maior que a do Home. Porém, como o expertise da equipe tende a ser maior no segundo desenvolvimento, essas diferenças foram consideradas como equivalentes.

Para cada um dos nove atributos será apresentada uma tabela com as seguintes informações:

- Definição das hipóteses;
- Identificação do tamanho das amostras;
- Definição do nível de significância;
- Ranking e atribuição dos postos;
- Cálculo da soma dos postos;
- Definição do valor crítico de acordo com a tabela de Wilcoxon [Barbetta, 2004].
- Refutação das hipóteses secundárias H0s.

3.4.1.3.1. Hipótese $H_{S1\text{-}ob1}$: Precisão Total de Cronograma:

a) Definição das Hipóteses, onde:

μ_1 = precisão total do cronograma do *OurProcess* e

μ_2 = precisão total do cronograma do *OurQualityProcess*.

Utilizando a abordagem unilateral, visto que espera que com o *OurQualityProcess* tenha maior precisão de cronograma:

$H_{S1\text{-}ob1\_0} : \mu_1 = \mu_2$

$H_{S1\text{-}ob1\_1}: \mu_1 < \mu_2.$

b) Identificação do tamanho das amostras (fases do processo):

n_1= amostra do *OurProcess* = 4 e n_2= amostra do *OurQualityProcess* = 4

c) Nível de significância: será considerado um nível de significância de 5%.

d) Ranking e atribuição dos postos:

Tabela 7: Atribuição dos pesos da precisão total do cronograma.

Grupo	Valores	Ranking
1	58	1
1	75	2
1	84	3
1	87	4
2	96	5
2	98	6
2	99	7,5
2	99	7,5

A Tabela 7 apresenta o ranking para os dois projetos, OP considerado como valores do grupo 1 e OQP, considerado como grupo 2. Nas quatro diferentes fases dos processos foram coletados os valores, descritos na 2ª coluna e, a partir dos valores observados, o ranking foi composto, conforme pode ser visto na 3ª coluna.

Para todas as hipóteses onde foi possível coletar mais de três amostras por processo, o teste de Wilcoxon foi utilizado.

e) Cálculo da soma dos postos $w_1 = 10$. Logo, aplicando a fórmula para encontrar o valor crítico para "*p*", *tem-se*:

$$u_1 = pesos - \left[\frac{N \cdot (N+1)}{2}\right] = 10 - [10] = 0$$

f) Encontrar o valor crítico u_c para o qual $P(U \leq u_c)$ □ 1 – 0,05 = 0,95. Procurando na tabela de Wilcoxon para n_1 e $n_2 = 4$ tem-se um $u_c = 14$.

g) Refutação da hipótese $H_{S1\text{-}ob1\_0}$:

Como a amostra produziu o valor $u_1 = 0$ que é menor que o valor crítico $u_c = 14$, o teste rejeita $H_{S1\text{-}ob1\_0}$ ao nível de significância de 5%. Então, o teste de Wilcoxon encontrou evidências estatísticas de que a precisão total do cronograma do *OurProcess* é inferior ao do *OurQualityProcess*.

3.4.1.3.2. Hipótese $H_{S2\text{-}ob1}$: Precisão do Cronograma na fase de teste:

a) Definição das Hipóteses, onde:

μ_1 = precisão total do cronograma para as USs de teste do *OurProcess* e

μ_2 = precisão total do cronograma para as USs de teste do OQP.

$H_{S2\text{-}ob1\_0} : \mu_1 = \mu_2$

$H_{S2\text{-}ob1\_1}: \mu_1 < \mu_2$.

b) Identificação do tamanho das amostras: $n_1 = n_2 = 4$

c) Nível de significância: 5%.

d) Ranking e atribuição dos postos:

Tabela 8: Atribuição dos pesos da precisão do cronograma de teste.

Grupo	Valores	Ranking
1	73	1,5
1	73	1,5
1	82	3
1	84	4
2	92	5
2	94	6
2	96	7
2	98	8

e) Cálculo da soma dos postos $w_1 = 10$. Logo, aplicando a fórmula para encontrar o valor crítico para "*p*", *tem-se*:

$$u_1 = pesos - \left[\frac{N \cdot (N+1)}{2}\right] = 10 - [10] = 0$$

f) Valor crítico u_c para o qual $P(U \leq u_c)$ □ 1 – 0,05 = 0,95. Procurando na tabela de Wilcoxon para n_1 e $n_2 = 4$ tem-se um $u_c = 14$

g) Refutação da hipótese $H_{S2\text{-}ob1\_0}$:

Como a amostra produziu o valor $u_1 = 0$ que é menor que o valor crítico $u_c = 14$, o teste rejeita $H_{S2\text{-}ob1\_0}$ ao nível de significância de 5%. Então, o teste de Wilcoxon encontrou evidências estatísticas de que a precisão do cronograma para as USs de teste do *OurProcess* é inferior ao do *OurQualityProcess*.

3.4.1.3.3. Hipótese $H_{S3\text{-}ob1}$: Precisão Total do Esforço:

a) Definição das Hipóteses, onde:

μ_1 = precisão total do esforço do *OP* e μ_2 = precisão total do esforço do *OQP*.

$H_{S3\text{-}ob1\_0} : \mu_1 = \mu_2$

$H_{S3\text{-}ob1\_1}: \mu_1 < \mu_2$.

b) Identificação do tamanho das amostras: n_1 e $n_2 = 4$

c) Nível de significância = 5%.

d) Ranking e atribuição dos postos:

Tabela 9: Atribuição dos pesos da precisão do esforço.

Grupo	Valores	Ranking
1	71	1
1	75	2,5
1	75	2,5
1	75	2,5
2	97	5
2	98	6,5
2	98	6,5
2	99	8

e) Cálculo da soma dos postos $w_1 = 8,5$. Logo, aplicando a fórmula para encontrar o valor crítico para "*p*", *tem-se*:

$$u_1 = pesos - \left[\frac{N \cdot (N+1)}{2}\right] = 8,5 - [10] = -1,5$$

f) Encontrar o valor crítico u_c para o qual $P(U \leq u_c)$ □ 1 – 0,05 = 0,95. Procurando na tabela de Wilcoxon para n_1 e $n_2 = 4$ tem-se um $u_c = 14$.

g) Refutação da hipótese $H_{S3\text{-}ob1\_0}$:

Como a amostra produziu o valor $u_1 = -1,5$ que é menor que o valor crítico $u_c = 14$, o teste rejeita $H_{S3\text{-}ob1\_0}$ ao nível de significância de 5%. Então, o teste de Wilcoxon encontrou evidências estatísticas de que a precisão total de esforço do *OurProcess* é inferior ao do *OurQualityProcess*.

3.4.1.3.4. Hipótese $H_{S4\text{-}ob1}$: Precisão do Esforço para as USs de teste:

a) Definição das Hipóteses, onde:

μ_1 = precisão do esforço para as USs do *OurProcess* e

μ_2 = precisão do esforço para as USs do *OurQualityProcess*.

Utilizando a abordagem unilateral, visto que espera que com o *OurQualityProcess* tenha maior precisão de esforço para USs:

$H_{S4\text{-}ob1\_0}: \mu_1 = \mu_2$

$H_{S4\text{-}ob1\_1}: \mu_1 < \mu_2$.

b) Identificação do tamanho das amostras: n_1 e $n_2 = 4$

c) Nível de significância = 5%.

d) Ranking e atribuição dos postos:

Tabela 10: Atribuição dos pesos da precisão do esforço para as USs.

Grupo	Valores	Ranking
1	64	1
1	77	2.5
1	77	2.5
1	78	4
2	94	5
2	98	6,5
2	98	6,5
2	98	6,5

e) Cálculo da soma dos postos $w_1 = 10$. Logo, aplicando a fórmula para encontrar o valor crítico para "*p*", *tem-se*:

$$u_1 = pesos - \left[\frac{N \cdot (N+1)}{2}\right] = 9 - [10] = -1$$

f) Encontrar o valor crítico u_c para o qual $P(U \leq u_c)$ □ 1 – 0,05 = 0,95. Procurando na tabela de Wilcoxon para n_1 e $n_2 = 4$ tem-se um $u_c = 14$.

g) Refutação da hipótese $H_{S4\text{-}ob1\_0}$:

Como a amostra produziu o valor $u_1 = -1$ que é menor que o valor crítico $u_c = 14$, o teste rejeita $H_{S4\text{-}ob1\_0}$ ao nível de significância de 5%. Então, o teste de Wilcoxon encontrou evidências estatísticas de que a precisão total de esforço do *OurProcess* é inferior ao do *OurQualityProcess*.

3.4.1.3.5. Hipótese $H_{S1\text{-}ob2}$: Número de *bugs* reportados:

a) Definição das Hipóteses, onde:

μ_1 = Número de *bugs* do *OurProcess* e

μ_2 = Número de *bugs* do *OurQualityProcess*.

Utilizando a abordagem unilateral, visto que espera que com o *OurQualityProcess* tenha maior número de *bugs* reportados:

$H_{S1\text{-}ob2\_0}: \mu_1 = \mu_2$

$H_{S1\text{-}ob2\_1}: \mu_1 < \mu_2$.

b) Identificação do tamanho das amostras: n_1 e $n_2 = 4$

c) Nível de significância = 5%.

d) 4 Ranking e atribuição dos postos:

Tabela 11: Atribuição dos pesos para *bugs* reportados.

Grupo	Valores	Ranking
2	251	1
1	38	2
2	36	3
1	12	4,5
2	12	4,5
1	8	6
1	4	7
2	3	8

e) 5 – Cálculo da soma dos postos w_1 = 10. Logo, aplicando a fórmula para encontrar o valor crítico para "*p*", *tem-se*:

$$u_1 = pesos - \left[\frac{N \cdot (N+1)}{2}\right] = 19{,}5 - [10] = 9{,}5$$

f) Encontrar o valor crítico u_c para o qual $P(U \leq u_c)$ □ 1 – 0,05 = 0,95. Procurando na tabela de Wilcoxon para n_1 e $n_2 = 4$ tem-se um $u_c = 14$.

g) Refutação da hipótese $H_{S1\text{-}ob2\_0}$:

Como a amostra produziu o valor $u_1 = 9{,}5$ que é menor que o valor crítico $u_c = 14$, o teste rejeita $H_{S1\text{-}ob2\_0}$ ao nível de significância de 5%. Então, o teste de Wilcoxon encontrou evidências estatísticas de que o número de *bugs* detectados no *OurProcess* é inferior ao do *OurQualityProcess*.

3.4.1.3.6. Hipótese $H_{S2\text{-}ob2}$: Cobertura de testes:

a) Definição das Hipóteses, onde:

μ_1 = Cobertura de testes do *OurProcess* e

μ_2 = Cobertura de testes do *OurQualityProcess*.

Utilizando a abordagem unilateral, visto que espera que com o *OurQualityProcess* tenha maior cobertura de testes:

$H_{S2\text{-}ob2\_0}: \mu_1 = \mu_2$

$H_{S2\text{-}ob2\_1}: \mu_1 < \mu_2$.

b) Identificação do tamanho das amostras: n_1 e $n_2 = 1$

Como não houve possibilidade de coletar a cobertura de testes em diferentes fases do processo *OurProcess* não foi possível aplicar um teste estatístico, apenas a comparação direta entre os valores obtidos, conforme Tabela 12. Apenas lembrando que os testes elaborados durante o OP não foram considerados no OQP.

Tabela 12: Valores da cobertura de testes para *OurProcess* e OQP.

Métrica 2.2.a	**Métrica 2.2.b**
$OP_{Cob} = 43{,}3\%$	$OQP_{Cob} = 92{,}4\%$

c) Refutação da hipótese $H_{S2\text{-}ob2\_0}$:

Como não foi possível coletar amostras para a cobertura de testes não é possível refutar a $H_{S2\text{-}ob2\_0}$. Porém, de acordo com os valores obtidos é notória a melhora da cobertura de testes no *OurQualityProcess,* aproximadamente 47% a mais de cobertura.

3.4.1.3.7. Hipótese $H_{S3\text{-}ob2}$: Número de testes produzidos:

a) Definição das Hipóteses, onde:

μ_1 = Número de testes produzidos no *OurProcess* e

μ_2 = Número de testes produzidos no *OurQualityProcess*.

Utilizando a abordagem unilateral, visto que espera que com o *OurQualityProcess* tenha maior número de testes produzidos:

$H_{S3\text{-}ob2\_0}: \mu_1 = \mu_2$

$H_{S3\text{-}ob2\_1}: \mu_1 < \mu_2$.

b) Identificação do tamanho das amostras: n_1 e $n_2 = 4$

c) Nível de significância = 5%.

d) Ranking e atribuição dos postos:

Tabela 13: Atribuição dos pesos para testes produzidos.

Grupo	Valores	Ranking
2	431	1
1	193	2
2	88	3
1	65	4
1	39	5
2	19	6
2	4	7
1	0	8

e) Cálculo da soma dos postos $w_1 = 10$. Logo, aplicando a fórmula para encontrar o valor crítico para "*p*", *tem-se*:

$$u_1 = pesos - \left[\frac{N \cdot (N+1)}{2}\right] = 19 - [10] = 9$$

f) Encontrar o valor crítico u_c para o qual $P(U \leq u_c)$ □ 1 – 0,05 = 0,95. Procurando na tabela de Wilcoxon para n_1 e $n_2 = 4$ tem-se um $u_c = 14$.

g) Refutação da hipótese $H_{S3\text{-}ob2\_0}$:

Como a amostra produziu o valor $u_1 = 9$ que é menor que o valor crítico $u_c = 14$, o teste rejeita $H_{S3\text{-}ob2\_0}$ ao nível de significância de 5%. Então, o teste de Wilcoxon encontrou evidências estatísticas de que o número de testes produzidos no *OurProcess* é inferior ao do *OurQualityProcess*.

3.4.1.3.8. Hipótese $H_{S1\text{-}ob3}$: Detecção de defeitos por linha de código:

a) Definição das Hipóteses, onde:

μ_1 = Número de defeitos encontrados no *OurProcess* e

μ_2 = Número de defeitos encontrados no *OurQualityProcess*.

Utilizando a abordagem unilateral, visto que espera que com o *OurQualityProcess* tenha maior número de defeitos encontrados:

$H_{S1\text{-}ob3\_0}: \mu_1 = \mu_2$

$H_{S1\text{-}ob3\_1}: \mu_1 < \mu_2$.

b) Identificação do tamanho das amostras: n_1 e $n_2 = 1$

Como não houve possibilidade de coletar o número de defeitos por linha de código em diferentes fases do processo *OurProcess* não foi possível aplicar um teste estatístico, apenas a comparação direta entre os valores obtidos, conforme Tabela 14.

Tabela 14: Número de defeitos encontrados por linha de código para *OurProcess* e OQP.

Métrica 3.1.a	**Métrica 3.1.b**
$Q_{Total\_OP} = 0{,}006$	$Q_{Total\_OQP} = 0{,}008$

c) Refutação da hipótese $H_{S1\text{-}ob3\_0}$:

Como não foi possível coletar amostras do número de defeitos por linha de código não é possível refutar a $H_{S1\text{-}ob3\_0}$. Porém, de acordo com os valores obtidos é possível identificar que a detecção de defeitos por linha de código no *OurProcess* é inferior ao do *OurQualityProcess*.

3.4.1.3.9. Hipótese $H_{S2\text{-}ob3}$: Detecção de defeitos por linha de código:

a) Definição das Hipóteses, onde:

μ_1 = Número de *issues* fechadas no *OurProcess* e

μ_2 = Número de *issues* fechadas no *OurQualityProcess*.

Utilizando a abordagem unilateral, visto que espera que com o *OurQualityProcess* tenha maior número de *issues* fechadas:

$H_{S2\text{-}ob3\_0}: \mu_1 = \mu_2$

$H_{S2\text{-}ob3\_1}$: $\mu_1 < \mu_2$.

b) Identificação do tamanho das amostras: n_1 e $n_2 = 1$

Como não houve possibilidade de coletar o número de *issues* fechadas em diferentes fases do processo *OurProcess* não foi possível aplicar um teste estatístico, apenas a comparação direta entre os valores obtidos, conforme Tabela 15.

Tabela 15: Percentual de *Issues* resolvidas para *OurProcess* e OQP.

Métrica 3.2.a	**Métrica 3.2.b**
$P_{resolvA} = 48\%$	$P_{resolvD} = 87\%$
$Issues_{Abertas} = 128$	$Issues_{Abertas} = 302$
$Issues_{Fechadas} = 62$	$Issues_{Fechadas} = 263$

c) Refutação da hipótese $H_{S2\text{-}ob3\_0}$:

Como não foi possível coletar amostras do número de *issues* fechadas não é possível refutar a $H_{S2\text{-}ob3\_0}$. Porém, de acordo com os valores obtidos é possível identificar que a porcentagem de *issues* fechadas no *OurProcess* é inferior ao do *OurQualityProcess*. Um acréscimo de mais de 55%.

Capítulo 4

4. Avaliação de Produtos do Processo Proposto *OurQualityProcess* segundo *Baseline* levantado junto a praticantes de Engenharia de Software.

4.1. Principais assuntos do capítulo

Estudos Experimentais

Roteiro do Estudo

Preparação

Execução

Avaliação

4.2. Introdução

A elaboração de uma segunda fase de avaliação foi motivada devido à impossibilidade de aplicar a avaliação estatística completa. Essa falta de completude foi ocasionada em virtude da insuficiência de amostras de alguns atributos de qualidade estabelecidos no processo *OurProcess*.

Na implantação do *OurProcess* não era previsto a obtenção da baixa qualidade dos produtos desenvolvidos e, desta forma, o controle e coletas sistemáticas não foi aplicado. Na implantação do novo processo a situação era diferente, esperava-se identificar se com a incorporação de novas estratégias seria possível maximizar a qualidade dos produtos desenvolvidos e, portanto, era previsto uma comparação para aferir a alteração da qualidade e, consequentemente, utilizou-se um processo de controle e coleta mais rigoroso.

Essa diferença na aquisição das informações relativas aos processos resultou que alguns atributos só foram mensurados no final do *OurProcess*, inviabilizando assim a comparação pareada entre os processos.

Para expandir a validade dos resultados foi desenvolvido um questionário (*survey*) aplicado a gestores de desenvolvimento com mais de um ano de experiência na área de desenvolvimento para definir os valores aceitáveis como métricas de avaliação da qualidade de *software* e o utilizar para analisar a qualidade dos dois processos de desenvolvimento de software utilizados como estudo de caso.

O conjunto de métricas foi definido seguindo a metodologia GQM. Para definir a qualidade dos produtos, os objetivos procurados são:

- Precisão das estimativas de tempo do projeto;
- Detecção de defeitos antes da liberação do produto;
- Cobertura de testes.

4.3. Definição do Baseline

Para definir o *baseline* foi desenvolvido um questionário, cujo objetivo foi identificar aspectos de qualidade de modo qualitativo, seguindo um processo de experimentação em quatro etapas de acordo com a metodologia apresentada por Travassos [Travassos, 2002]:

- Definição dos objetivos;
- Planejamento do experimento;
- Execução do experimento e coleta de dados;
- Análise e interpretação dos dados obtidos.

4.4. Definição dos Objetivos

Objeto de estudo: Exigências dos participantes em relação às métricas definidas para a qualidade de software.

Objetivo do estudo: Com o propósito de definir o *baseline* para métricas de qualidade de software e, com isso, avaliar a qualidade dos processos de desenvolvimento de software utilizados como estudo de caso.

O foco da qualidade: Confiabilidade dos processos de desenvolvimento de software.

Perspectiva: Do ponto de vista de gerentes de desenvolvimento de software e/ou gerentes de qualidade de software com mais de um ano de experiência.

Contexto: Ambientes de desenvolvimento de software.

Em resumo, estão apresentados na Tabela 16 os objetivos e suas relativas questões e métricas. As métricas (terceira coluna) significam como os objetivos (primeira coluna)

serão atendidos. Isto é, quais serão os dados coletados que irão responder às questões (segunda coluna) e, assim, representar os objetivos quantitativamente e qualitativamente.

Tabela 16: Objetivos, questões e métricas definidas através da abordagem GQM

Objetivos	Questões	Métricas
Objetivo 1: Precisão das estimativas de projeto	**Questão 1:** Que taxa de erro é aceitável para a divergência entre os prazos previstos e os realizados em um cronograma de desenvolvimento de software?	T_E - **tempo** estimado do desenvolvimento
		T_R - tempo **real** do desenvolvimento
	Questão 2: Que taxa de erro é aceitável para a divergência entre o esforço planejado (homem/hora) e o realizado?	Esf_E - esforço **estimado** do desenvolvimento
		Esf_R - esforço **real** do desenvolvimento
Objetivo 2: Detecção de defeitos antes da liberação do produto	**Questão 3:** Que percentual mínimo de aumento no número de falhas detectadas após o desenvolvimento justifica uma nova etapa de desenvolvimento?	$Def_{PósProcessA}$ - Defeitos por linhas de código no desenvolvimento
		$Def_{PósProcessB}$ - Defeitos por linhas de código após o desenvolvimento
	Questão 4: Considerando o número total de problemas em aberto (*issues*), qual seria um percentual aceitável para a resolução dos mesmos antes da homologação do produto de software?	$Issues_{Total}$ - Total de ***issues***
		$Issues_{Fechadas}$ - Total de ***issues*** **fechadas**
Objetivo 3: Cobertura de teste	**Questão 5:** Qual o percentual mínimo aceitável para a cobertura dos testes em um produto de software?	$Cobertura_{Cod}$ - Cobertura do código

4.5. Planejamento

A primeira etapa do planejamento de experimentos é a definição das hipóteses assumidas. Neste estudo, as hipóteses são as seguintes:

H_0 – **A hipótese nula:** Todos os gerentes optam pelos mesmos níveis de exigências para as métricas de qualidade de software.

H_0:(Nível de exigência do gerente$_1$ ∩ Nível de exigência do gerente$_2$ ∩ ...Nível de exigência do gerente$_n$ = Nível de exigência do gerente$_1$), gerentes.

H_1 – **A hipótese alternativa:** Os gerentes divergem em relação aos níveis de exigência do cronograma.

H_1: ($\exists$x,y | Nível do cronograma do gerente$_x$ ∩ Nível do cronograma do gerente$_y$ = Ø)

H_2 – **A hipótese alternativa:** Os gerentes divergem em relação aos níveis de exigência do esforço.

H_2: ($\exists x,y$ | Nível do esforço do gerente$_x$ ∩ Nível do esforço do gerente$_y$ = Ø)

H_3 – **A hipótese alternativa:** Os gerentes divergem em relação aos níveis de exigência do percentual de falhas que justifiquem uma nova etapa do desenvolvimento.

H_3: ($\exists x,y$ | Nível do percentual de falhas do gerente$_x$ ∩ Nível do percentual de falhas do gerente$_y$ = Ø)

H_4 – **A hipótese alternativa:** Os gerentes divergem em relação aos níveis de exigência do percentual de *issues*. Que devem ser fechadas antes da homologação do produto final.

H_4: ($\exists x,y$ | Nível do percentual de *issues* fechadas do gerente$_x$ ∩ Nível do percentual de *issues* fechadas do gerente$_y$ = Ø)

H_5 – **A hipótese alternativa:** Os gerentes divergem em relação aos níveis de exigência da cobertura de testes.

H_5: ($\exists x,y$ | Nível de cobertura do gerente$_x$ ∩ Nível de cobertura do gerente$_y$ = Ø)

Para testar as hipóteses foram definidas questões de múltipla escolha mostradas na Tabela 17.

Tabela 17 - Questionário submetido aos gerentes e suas respectivas alternativas

Questão 1: Que taxa de erro é aceitável para a divergência entre os prazos previstos e os realizados em um cronograma de desenvolvimento de software?	Inferior a 5%
	Entre 5 e 15%
	Entre 15 e 35%
	Entre 35 e 50%
	Acima de 50%
Questão 2: Que taxa de erro é aceitável para a divergência entre o esforço planejado (homem/hora) e o realizado?	Inferior a 5%
	Entre 5 e 15%
	Entre 15 e 35%
	Entre 35 e 50%
	Acima de 50%
Questão 3: Que percentual mínimo de aumento no número de falhas detectadas após o desenvolvimento justifica uma nova etapa de desenvolvimento?	Até 1%
	Entre 1 e 5%
	Entre 5 e 15%

	Entre 15 e 30%
	Superior a 30%
Questão 4: Considerando o número total de problemas em aberto (*issues*), qual seria um percentual aceitável para a resolução dos mesmos antes da homologação do produto de software?	Até 50%
	Entre 50 e 70%
	Entre 70 e 90%
	Entre 90 e 95%
	Superior a 95%
Questão 5: Qual o percentual mínimo aceitável para a cobertura dos testes em um produto de software?	Até 50%
	Entre 50 e 70%
	Entre 70 e 90%
	Entre 90 e 95%
	Superior a 95%

Outra fase de instrumentação deste trabalho foi elaborada para coletar as métricas de qualidade relacionadas aos processos *OurProcess* e *OurQualityProcess*. Para tanto foi utilizado o mesmo conjunto de ferramentas de análise de código, acompanhamento de *builds* diários (execução de toda a bateria de testes), repositório de erros, ferramenta de gerenciamento de atividades, entre outras.

A avaliação da confiabilidade do questionário foi feita através da técnica de *Inter-observer Reliability* e a validação foi através do método subjetivo *Content Validity*. Além disso, aplicamos as técnicas propostas por Travassos [Travassos, 2002] como padrão para a elaboração do estudo empírico para *Survey*.

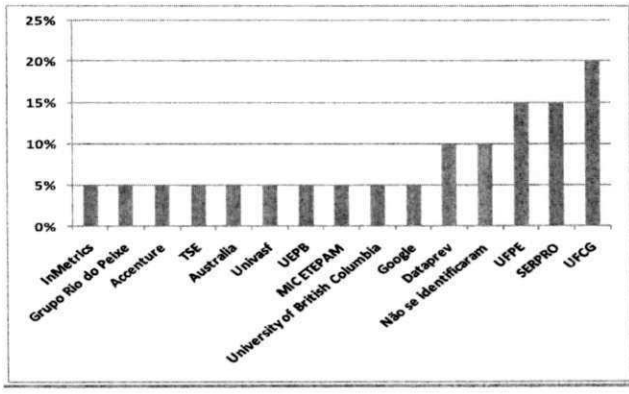
4.6. Coleta de Dados

O questionário foi submetido aos respondentes utilizando um recurso disponível nas aplicações do *Google* conhecido como *Google Docs Forms*. Com ele é possível responder à pesquisa diretamente no e-mail recebido ou acessar o formulário de modo *on-line*.

Além das questões apresentadas na Tabela 17, foram elaboradas também questões para mapear o perfil dos participantes, permitindo a preservação do anonimato. Os perfis obtidos tanto em relação às instituições de origem dos participantes quanto à experiência profissional e os resultados encontram-se em detalhes na Tabela 18.

Conforme a Tabela 18 é possível identificar que 96% dos participantes possui mais de um ano de experiência como gestor de projetos de desenvolvimento de software, sendo 76% deles com mais de três anos. Além disso, é possível também identificar que o questionário foi respondido por profissionais oriundos de diversas instituições, tanto privadas quanto públicas, e de pequeno, médio e grande porte, desenvolvendo aplicações de diversas naturezas, essa diversidade expande a utilização do *baseline* para desenvolvimento de software de diferentes naturezas.

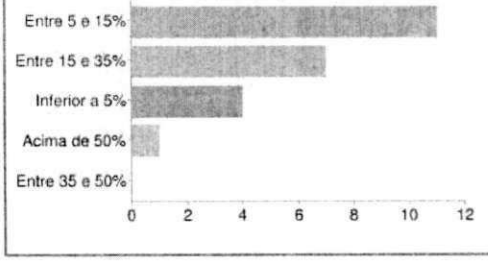
Tabela 18 - Dados dos participantes



Experiência	Quant.	%
Inferior a 1 ano	2	4%
Entre 1 e 3 anos	14	30%
Entre 3 e 5 anos	20	43%
Entre 5 e 10 anos	2	4%
Superior a 10 anos	8	17%

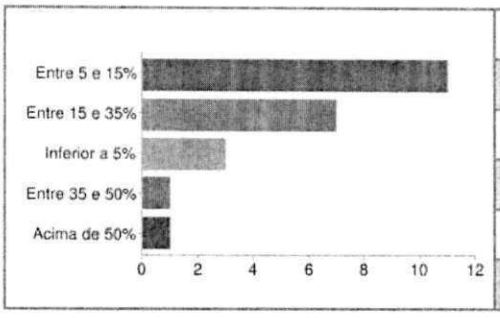
4.7. Resultado, Análise e Interpretação dos Dados Obtidos

Como os valores-respostas para as questões propostas estão da escala ordinal, somente é possível escolher como medida de tendência central a mediana e a moda. No contexto sob estudo será utilizada a moda, pois com a mediana não será possível representar a opção da maioria.



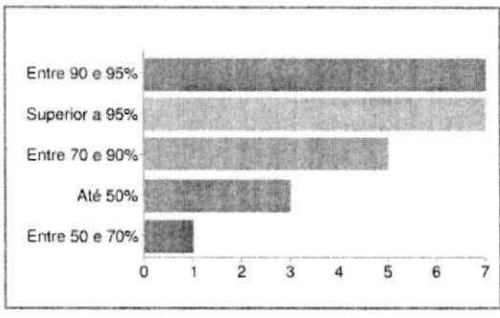
Opção	Número de Pessoas	%
Inferior a 5%	8	17%
Entre 5 e 15%	22	48%
Entre 15 e 35%	14	30%
Entre 35 e 50%	0	0%
Acima de 50%	2	4%

Figura 3 - Diagrama de Pareto e estatísticas para a Questão 1 referente ao erro de cronograma aceitável.



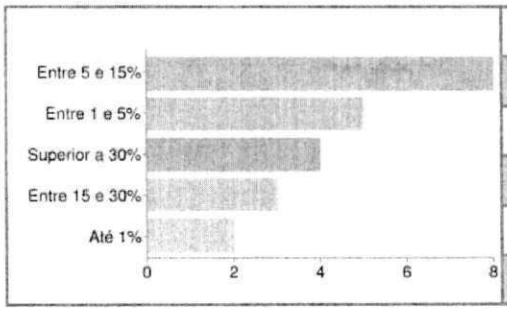
Opção	Número de Pessoas	%
Inferior a 5%	6	13%
Entre 5 e 15%	22	48%
Entre 15 e 35%	14	30%
Entre 35 e 50%	2	4%
Acima de 50%	2	4%

Figura 4 - Diagrama de Pareto e estatísticas para a Questão 2 referente ao erro de estimativa de esforço aceitável.



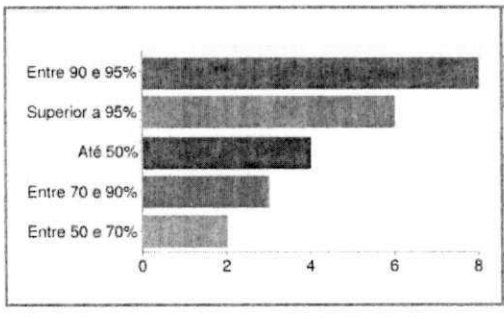
Opção	Número de Pessoas	%
Até 50%	6	13%
Entre 50 e 70%	2	4%
Entre 70 e 90%	10	22%
Entre 90 e 95%	14	30%
Superior à 95%	14	30%

Figura 5 - Diagrama de Pareto e estatísticas para a Questão 3 referente ao percentual de falhas detectadas que justificam uma nova fase de desenvolvimento.



Opção	Número de Pessoas	%
Até 1%	4	9%
Entre 1 e 5%	10	22%
Entre 5 e 15%	16	35%
Entre 15 e 30%	6	13%
Superior à 30%	8	17%

Figura 6 - Diagrama de Pareto e estatísticas para a Questão 4 referente ao percentual mínimo aceitável para soluções de *issues* detectadas.



Opção	Número de Pessoas	%
Até 50%	8	17%
Entre 50 e 70%	4	9%
Entre 70 e 90%	6	13%
Entre 90 e 95%	16	35%
Superior a 95%	12	26%

Figura 7 - Diagrama de Pareto e estatísticas para a Questão 5 referente ao mínimo aceitável para a cobertura de testes.

De acordo com os gráficos e tabelas apresentados, conseguimos definir os valores que compõem o *baseline* conforme apresentado na Tabela 19.

Tabela 19 - Moda dos resultados - dados da estatística descritiva.

Questões	Opção	Pessoas	%
Questão 1	Entre 5 e 15%	22	48%
Questão 2	Entre 5 e 15%	22	48%
Questão 3	Entre 90 e 95%	14	30%
	Superior a 95%	14	30%
Questão 4	Entre 5 e 15%	16	35%
Questão 5	Entre 90 e 95%	16	35%

Houve duas modas para a questão 3. Duas opções foram escolhidas por quatorze pessoas. Note, porém, que as escalas são seqüenciais. Dessa forma, aumentaremos a abrangência da escala para que seja possível agrupar as duas alternativas empatadas, visto que o intervalo destas opções é menor que o intervalo entre as demais. Esta prática de agrupamento de repostas é possível devido ao tipo de medida de tendência central. De acordo com as respostas selecionadas e os resultados da estatística descritiva, sumarizamos as conclusões na Tabela 20, identificando o percentual de escolha para as opções que representaram a moda.

Tabela 20 - Conclusões do teste descritivo.

Questões	Múltipla escolha	Características
Questão 1	Entre 5 e 15%	Para 48% dos respondentes a taxa de erro entre o cronograma planejado e o cronograma realizado não pode passar de 15% de erro.
Questão 2	Entre 5 e 15%	Para 48% dos respondentes a taxa de erro entre o esforço planejado (homem/hora) e o esforço realizado não pode passar de 15% de erro.
Questão 3	Superior a 90%	Como houve um agrupamento envolvendo dois intervalos que representaram a moda, com 28 participantes optaram por esta medida. Portanto, para 60% dos respondentes se for feito um levantamento dos erros existentes e após a fase de desenvolvimento houver um aumento superior a 5% em relação aos erros encontrados durante o desenvolvimento, então é necessária uma nova fase de desenvolvimento para aumentar a qualidade dos produtos.

Questão 4	Entre 5 e 15%	Para 35% dos respondentes é necessário resolver quase a totalidade das *issues* abertas durante o desenvolvimento para se obter um software de qualidade. Porém, os respondentes não descartaram a possibilidade de deixar algumas *issues* conhecidas sem solução, designando uma margem de 5% para esses casos.
Questão 5	Entre 90 e 95%	Para 35% dos respondentes, a cobertura de testes deve ser superior a 90%.

4.8. Verificação das Hipóteses

A hipótese nula H_0: Para verificar a hipótese nula precisamos analisar as respostas de todos os questionários e todas as alternativas selecionadas para garantir que não há diferença entre as respostas dos participantes. Como em todas as questões houve divergências nas respostas, a hipótese nula foi refutada.

A hipótese alternativa H_1: Para verificar a hipótese alternativa H_1 precisamos analisar qualquer caso em que haja divergência para confirmar a alternativa. Analisando os gráficos apresentados na Figura 3, nota-se que existem divergências entre as respostas coletadas.

A hipótese alternativa H_2: Para verificar a hipótese alternativa H_2 precisamos analisar qualquer caso em que haja divergência para confirmar a alternativa. Analisando os gráficos apresentados na Figura 4, nota-se que existem divergências entre as respostas coletadas.

A hipótese alternativa H_3: Para verificar a hipótese alternativa H_3 precisamos analisar qualquer caso em que haja divergência para confirmar a alternativa. Analisando os gráficos apresentados na Figura 5, nota-se que existem divergências entre as respostas coletadas.

A hipótese alternativa H_4: Para verificar a hipótese alternativa H_4 precisamos analisar qualquer caso em que haja divergência para confirmar a alternativa. Analisando os gráficos apresentados na Figura 6, nota-se que existem divergências entre as respostas coletadas.

A hipótese alternativa H_5: Para verificar a hipótese alternativa H_5 precisamos analisar qualquer caso em que haja divergência para confirmar a alternativa. Analisando os gráficos apresentados na Figura 7, nota-se que existem divergências entre as respostas coletadas.

4.9. Avaliação dos Softwares de Acordo com o Baseline Definido

Todos os objetivos foram avaliados de acordo com a abordagem proposta na seção anterior. Analisando os valores coletados, de modo geral, foi visto que os objetivos

apresentados foram alcançados de forma satisfatória. Para cada objetivo, os resultados serão apresentados através das tabelas abaixo e uma avaliação dos mesmos.

Tabela 21 – Primeiro objetivo - com os resultados de cada métrica.

Objetivo 1 : Precisão das estimativas de projeto			
Questão 1: Que taxa de erro é aceitável para a divergência entre os prazos previstos e os realizados em um cronograma de desenvolvimento de software? (em horas)			
$Crono\_Previstos_{OP}$ = 605,1	$Crono\_Realizados_{OP}$ = 918	$Crono\_Previstos_{OQP}$ = 412,1	$Crono\_Realizados_{OQP}$ = 44,2
$Crono\_Precisão_{OP}$ =65 e Erro = 35%		$Crono\_Precisão_{OP}$ =93 e Erro = 7%	
Questão 2: Que taxa de erro é aceitável para a divergência entre o esforço planejado (homem/hora) e o realizado?			
$Esf\_Previsto_{OP}$ = 238,8	$Esf\_Realizado_{OP}$ = 341,5	$Esf\_Previsto_{OQP}$ = 92,5	$Esf\_Realizado_{OQP}$= 100,9
$Esf\_Precisão_{OP}$ = 69 e Erro = 31% e		$Esf\_Precisão_{OP}$ = 92 e Erro = 8%	

Com relação ao primeiro objetivo, foi verificado que a precisão das estimativas melhorou significativamente no *OQP*. Essa melhora foi resultado da mudança no planejamento das atividades, criou-se uma regra para que toda a atividade planejada tivesse o tempo máximo de duração de quatro horas. Com esta prática, as atividades com maior complexidade foram particionadas, obtendo-se uma redução nos erros das estimativas. Pela análise da precisão das Questões 1 e 2 nos Processos *OP* e *OQP*, mostradas na Tabela 21, notamos que em ambas as comparações de precisão das estimativas obtivemos mais de 90% de precisão com o *OQP*, o que facilita a tomada de decisões de projeto relativas principalmente à alocação de recursos, de tempo e de pessoal.

Tabela 22 – Segundo objetivo, com os resultados de cada métrica.

Objetivo 2: Detecção de defeitos antes da liberação do produto
Questão 3: Que percentual mínimo de aumento no número de falhas detectadas após o desenvolvimento justifica uma nova etapa de desenvolvimento? Para identificar o percentual de falhas do projeto aplicamos as métricas em dois momentos: 1 - durante o desenvolvimento. 2 - Após o término do desenvolvimento.

Defeitos$_{OP\text{-}01}$ = 64 Defeitos$_{OP\text{-}02}$ = 228 Mais de 78% dos defeitos foram detectados a após o término do desenvolvimento.	Defeitos$_{OP\text{-}01}$ = 302 Defeitos$_{OP\text{-}01}$ = 11 Apenas 3,5% dos defeitos foram detectados após o término do desenvolvimento.
Questão 4: Considerando o número total de problemas em aberto (*issues*), qual seria um percentual aceitável para a resolução dos mesmos antes da homologação do produto de software?	
Issues-Detectadas$_{OP}$ = 128 *Issues*-Fechadas$_{OP}$ = 62 Fechamento de 48% das *issues*	*Issues*-Detectadas$_{OQP}$ = 302 *Issues*-Fechadas$_{OQP}$ = 263 Fechamento de 87% das *issues*

No que diz respeito ao objetivo 2, analisando os dados da Tabela 22 pode-se obter duas análises, uma em relação às falhas descobertas após a liberação do software pela equipe de desenvolvimento e outra em relação ao número de *issues* fechadas para a liberação do produto.

Na primeira, nota-se que o esforço empregado na detecção de defeitos na fase de desenvolvimento do *OurProcess* foi insuficiente, devido ao aumento de 356% na detecção dos defeitos após a liberação do produto.

Em relação ao *OQP*, nota-se que houve um aumento no esforço empregado na detecção de defeitos durante a fase de desenvolvimento, visto que a detecção de defeitos foi menor que 4% após a finalização do desenvolvimento.

Na análise do fechamento das *issues*, comparando os dois processos, nota-se que no *OP* houve um fechamento menor que 50% do total de *issues,* no entanto 20% das *issues* abertas eram referentes a requisição de novas funcionalidades para o software e 28% eram defeitos de código.

No *OQP* o fechamento resultou em 87%. Embora não tenhamos obtido um valor ideal para o fechamento das *issues* concluí-se que as *issues* remanescentes não representam instabilidades do código, isto porque 83,6% das abertas são requisições de melhorias de usabilidade e apenas duas *issues* (0,6%) fazem referência aos defeitos.

Tabela 23 - Terceiro objetivo, com os resultados de cada métrica.

Objetivo 3: Cobertura de teste	
Questão 5: Qual o percentual mínimo aceitável para a cobertura dos testes em um produto de software?	
Cobertura$_{OP}$ = 43%	Cobertura$_{OQP}$ = 92%

A Tabela 23 apresenta os valores coletados relacionados ao terceiro objetivo. Esta análise é bastante significativa e muito utilizada para avaliar a qualidade dos softwares. No *OurProcess*, a cobertura dos testes não chegou a 50%, um valor relativamente baixo para os padrões de qualidade almejados. No *OurQualityProces*, notou-se que houve um aumento na detecção de falhas durante a fase de desenvolvimento evitando que defeitos se manifestassem após a liberação do software.

4.10. Análise Utilizando o Baseline

O intuito do questionário não é restrito a definição e coleta das métricas, mas também avaliar os dois processos de desenvolvimento de software. De acordo com Gomes [Gomes 2001], podem-se identificar métricas relacionando-as com questões e as identificando com objetivos a serem alcançados. Essas métricas não garantem a melhoria do processo, apenas indicam se as metas foram atingidas e possíveis falhas.

Para que a análise tenha uma abrangência maior, os processos serão avaliados separadamente a partir dos resultados obtidos fazendo uma análise da conformidade entre os valores obtidos nos processos e os definidos no *baseline*.

A Tabela 24 identifica cada questão (primeira coluna), os valores definidos no *baseline* (segunda coluna) e valores obtidos no *OurProcess* e *OurQualityProcess* (colunas 3 e 4, respectivamente).

Tabela 24 - Avaliação dos processos de desenvolvimento de acordo com o *baseline* definido.

Questões	*Baseline*	*OurProcess*	*OurQualityProcess*
Questão 1	Entre 5 e 15%	35%	7%
Questão 2	Entre 5 e 15%	38%	8%
Questão 3	Entre 1 e 5%	356%	3%
Questão 4	Superior a 90%	48%	87%
Questão 5	Entre 90 e 95%	43%	92%

Questões 1 e 2: a diferença entre os processos é significativa. Enquanto o *OQP* está dentro dos padrões aceitáveis, o *OP* é mais que 100% pior dos valores máximos aceitáveis para o percentual de erros na estimativa de cronograma do projeto.

Questão 3: O *OQP*, também na questão 3 tem seus valores dentro do intervalo aceitável definido no *baseline,* enquanto que no *OP* houve uma alta detecção de falhas após a liberação do produto de software, com um valor muito elevado para a detecção de erros após o término do desenvolvimento.

Questão 4: ambos os processos não obtiveram valores dentro do padrão estipulado no *baseline*. Muito embora o *OQP* tenha ficado muito próximo do intervalo ideal de valores maiores ou iguais a 90%, enquanto que o *OP* obteve um percentual inferior a 50% do esperado.

Questão 5: a avaliação dos processos foi satisfatória para o *OQP* e insatisfatória para o *OP* e novamente os percentuais obtidos foram abaixo da metade esperada.

4.11. Empacotamento

O *survey* que produziu o *baseline* foi disponibilizado através da ferramenta Google Docs Form [Dora, 2010].

O *OurProcess* pode ser obtido no site do LSD [*OurProcess*].

O *OurQualityProcess* também pode ser obtido no site do LSD [*OurQualityProcess*].

4.12. Ameaças à validade

Validade de Conclusão:

A validade de conclusão será feita através de dois aspectos: métricas de qualidade de software e o resultado da comparação entre os dois processos, o *OurProcess* e o *OurQualityProcess*.

As ameaças à validade de conclusão deste trabalho são:

- O tamanho das equipes. Se o número de desenvolvedores estimado não foi o ideal para o tamanho do projeto ou então se o tamanho da equipe de qualificação não foi suficiente ou ainda superestimada para o tamanho do projeto;
- perfil da equipe de desenvolvimento (equipes são compostas tanto por alunos com pouca experiência como por profissionais de desenvolvimento, portanto a qualidade do código desenvolvido e avaliado pode ter divergências dependendo do responsável (eis) pela implementação);

- tempo requisitado pelo cliente para a entrega do software (se o prazo de entrega for apertado os desenvolvedores priorizam o desenvolvimento do à elaboração de testes, por exemplo. Neste caso a qualidade é preterida em relação ao término de desenvolvimento das funcionalidades requisitadas);
- orçamento do projeto (projetos com maiores orçamentos permitem maior distribuição de papéis, melhores equipamentos e estrutura para o desenvolvimento, contratação de profissionais especializados, treinamentos, dentre outros benefícios para facilitar o desenvolvimento do software).

Validade Interna:

A amostra utilizada na realização deste experimento é significativa por representar produtos difundidos mundialmente e seus usuários não se restringem ao ambiente de desenvolvimento ou mesmo com finalidades acadêmicas. Desta forma, a amostra em estudo permite que sejam observados comportamento e utilização diversificados.

Com relação à validade interna deste estudo, existem algumas ameaças, como:

- a maturidade dos desenvolvedores;
- tempo no desenvolvimento de software de natureza distribuída;
- tipo de atividade atribuída a cada desenvolvedor, enfim todas as peculiaridades em relação à experiência de um desenvolvedor na atividade realizada;
- o tipo de software sendo desenvolvido (produtos de software com maior integração de módulos, muito grandes, que são executados em diferentes plataformas, distribuídos, entre outras características tendem a serem mais complexos e exigir mais dos desenvolvedores. Com isso, a comparação entre software utilizando diferentes processos pode ser injusta).

Validade de Construção:

O processo de qualificação foi desenvolvido emergencialmente e todos os princípios e práticas propostos foram agrupados vislumbrando a elevação da qualidade dos produtos imediata. A seleção dos princípios foi obtida através de estudos de casos de sucesso no desenvolvimento de software, os problemas foram detectados de acordo com a experiência dos líderes de desenvolvimento e confirmados através de um estudo realizado com gerentes de desenvolvimento, com mais de um ano de experiência, em diferentes tipos de

desenvolvimento. Os instrumentos de coleta de dados foram ferramentas consolidadas e de ampla utilização no meio científico/acadêmico.

Todavia, a forma como o processo foi produzido, isto é, inserindo um conjunto de princípios e práticas conjuntamente não se pode aferir qual/quais os princípios e práticas se mostraram mais eficientes para este contexto. Para este fim, seria necessário um novo estudo para comparar separadamente cada estratégia inserida no novo processo.

Validade Externa:

O processo *OurQualityProcess* foi aplicado no LSD em diferentes tipos de software de natureza distribuída com diferentes equipes de desenvolvimento e variabilidade nos perfis de desenvolvedores.

Dentre as ameaças à validade externa, tem-se que o processo não foi aplicado externamente ao ambiente supracitado onde a característica do desenvolvimento tende a ser semelhante, portanto mesmo se tratando de diferentes equipes de desenvolvimento, todas estão inseridas no mesmo ambiente. Situações externas com outros clientes, não vivenciadas no LSD, podem ter influência significativa no resultado da aplicação do processo.

Capítulo 5

5. Conclusões e Trabalhos Futuros

5.1. Principais assuntos do capítulo

Conclusões

Trabalhos Futuros

5.2. Conclusões

O conceito de qualidade de *software* está no núcleo do desafio do desenvolvimento de *software*, isso porque qualidade é um atributo subjetivo e dependente das necessidades do cliente. Então o primeiro problema de pesquisa neste trabalho foi identificar quais os atributos no *software* que caracterizam a qualidade.

Através de entrevistas diretas com os líderes de desenvolvimento do Laboratório de Sistemas Distribuídos - LSD da UFCG e análise dos problemas reportados pelos usuários do *OurGrid* (grade computacional de grande expressividade em âmbito nacional e alguma internacional) foi possível identificar os principais problemas que resultavam em uma qualidade dos produtos abaixo do esperado.

Para cada problema identificado, uma solução foi proposta e incorporada no processo *OurProcess* (processo de desenvolvimento precursor ao *OurQualityProcess*). Após inserções e adaptações foi desenvolvido o *OurQualityProcess* com o objetivo de direcionar o desenvolvimento para produzir *software* com qualidade satisfatória.

A elaboração de um novo processo de desenvolvimento de software ágil foi fundamentada nos principais problemas de desenvolvimento detectadas com o *OurProcess* e devido a inúmeras peculiaridades do ambiente de desenvolvimento de sistemas distribuídos e o desenvolvimento de softwares na academia. Com isso, foi necessário integrar dois mundos e analisar a peculiaridade desses dois paradigmas para propiciar uma fusão que resultasse no desenvolvimento de *software* distribuído de forma ágil e com maior

controle de qualidade e respeitando a natureza do desenvolvimento acadêmico buscando a minimização dos problemas identificados.

Após a elaboração do *OurQualityProcess* foi selecionado um produto do *OurGrid* para ser utilizado como projeto piloto do novo processo, o *OurBackup*. Esse software foi selecionado pelos seguintes motivos: ter uma equipe de desenvolvimento que seguia o *OurProcess*; ser um software distribuído, com alta complexidade, porém, bem menor que o *OurGrid*; ter uma equipe de desenvolvimento fixa; ter um ciclo de vida bem definido; ter métricas de software que possibilitavam uma comparação direta entre os desenvolvimentos seguindo os dois processos.

Para a fase de análise e avaliação do *OurQualityProcess* foram utilizadas as técnicas da Engenharia de Software Experimental para comparação dos dois processos com o objetivo de refutar a hipótese nula de que ambos os processos, *OurProcess* e *OurQualityProcess* geravam produtos de *software* tinham qualidade equivalente. Embora não tenha sido possível analisar todos os sub-atributos de qualidade através de testes estatísticos (*Mann- Whitney Wilcoxon*) utilizando para comparar pareadamente dois tratamentos, onde não há possibilidade de garantia da normalidade dos dados, no caso deste trabalho, a comparação pareada dos dois processos foi utilizada para a maioria dos atributos analisados. A análise foi dividida em nove hipóteses, isto é, nove hipóteses nulas e para cada hipótese nula uma hipótese alternativa correspondente. Para seis das 9 hipóteses foi possível aplicar o teste de estatístico pareado e, em todos os casos as hipóteses nulas foram refutadas, comprovando estatisticamente que os processos *OurProcess* e *OurQualityProcess* não podem ser considerados equivalente e, além disso, como os testes foram unilaterais, foi possível comprovar também que para a qualidade (de acordo com os atributos de qualidade pré-estabelecidos) do *OurProcess* é inferior à qualidade do *OurQualityProcess*. Para os três atributos - cobertura de teste, defeitos por linha de código e percentual de *issues* resolvidas – onde não foi possível coletar mais que uma amostra e, consequentemente, não foi possível aplicar o teste estatístico, houve uma comparação direta e os resultados obtidos indicam uma significativa melhora nos índices de cada um dos três atributos.

Com o intuito de melhorar a avaliação dos processos, foi também desenvolvida uma avaliação comparativa com um *baseline* desenvolvido de acordo com a metodologia

científica, isto é: definição dos objetivos, planejamento, elaboração, execução, coleta de dados, análise, interpretação e empacotamento. Na avaliação o *OurQualityProcess* teve uma avaliação melhor que o *OurProcess* porém, na quarta questão, referente ao percentual ideal de *issues* resolvidas para a homologação do *software* nenhum dos dois processos obteve um percentual dentro do intervalo aceitável. O *OQP* ficou a três pontos percentuais de distância do mínimo aceitável, enquanto que o *OP* ficou a quarenta e dois pontos percentuais do mínino.

Com os valores obtidos nas duas análises é possível identificar uma significativa melhora incorporada no *OuQualityProcess* porém, como se trata de uma área muito subjetiva, existem muitas ameaças a validade desde estudo, como o tipo do software desenvolvido, a possibilidade de discrepâncias em relação a habilidade dos desenvolvedores, a alocação de tarefas, o tempo de desenvolvimento, a inserção dos princípios e prática agrupadamente (possibilitando a incorporação de confusões de fatores), a homogeneidade dos produtos desenvolvidos, entre outros.

5.3. Trabalhos Futuros

O estudo de caso utilizado neste trabalho comparou apenas o *OurQualityProcess* com um processo de natureza semelhante e destinado para a mesma natureza de softwares. Seria interessante comparar o *OQP* com outros processos ágeis não necessariamente desenvolvidos para a natureza distribuída e realizar novas comparações estatísticas. Além disso, é necessário também identificar separadamente cada uma das estratégias incorporadas para aferir o grau de importância de cada técnica ou princípio incorporado no *OP*.

Outra possibilidade seria estudar melhor os processos de desenvolvimento existentes para identificar os pontos comuns e divergentes, tentando assim adaptar o *OQP* para ser utilizado por diferentes tipos de desenvolvimento.

Por fim, em relação a análise apresentada houve alguns atributos de qualidade que não foram testados estatisticamente por falta de dados no processo precursor, com isso a validação estatística não abrange a totalidades dos atributos que se pretendeu avaliar, portanto, outra atividade importante a ser realizada é o acompanhamento de um novo desenvolvimento, com um controle sistemático apropriado para a coleta de todas as informações necessárias para uma avaliação estatística integral.

Bibliografia

[AgileCoop] Corbucci, H., *AgileCoop: curso de verão 2010.* 2010 Disponível em: http://www.Agilcoop.org.br/files/AgilCoop-Verao10-Lean.pdf. Último acesso em maio/ 2011.

[Abath, 2007] Abath O., Sauvé, J., Dantas, A. *Patterns for Scripted Acceptance Test-Driven Development.* Proceedings of EuroPLoP, 7, 2007.

[Abrahamsson, 2002] Abrahamsson, P.,Salo, O., Ronkainen, J., Warsta, J. *Agile software development methods –Review and Analysis.* Relatório Técninco, 478, p 107. VTT Publication. Disponível em: http://www.vtt.fi/inf/pdf/publications/2002/P478.pdf. Último acesso em: setembro de 2011.

[Amaral, 2003] Amaral, E. *Empacotamento de Estudos Experimentais em Engenharia*

de Software. Dissertação de Mestrado. Programa de Engenharia de Sistema e Computação, COPPE/UFRJ, Rio de Janeiro, Brasil, 2003.

[Anderson, 1998] Anderson, A., Beattie, R. Beck, K. et al. *Chrysler goes to "extremes".* In: Distributed Computing, 1998. Disponível em: http://www.xprogramming.com/ publications/dc9810cs.pdf. Último acesso em: setembro/2010.

[Astels, 2003] Astels, D., *test driven development – A practical Guide.* Prentice Hall, 2003.

[Atlassian] *Software development and collaboration tools for teams that get stuff done.* Dsponível em: http://www.atlassian.com/ Último acesso em: junho de 2011.

[Barbetta, 2004] Barbetta, P., Reis, M., Bornia, A. *Estatística para Cursos de Engenharia e Informática.* Editora Atlas, 2004 pp 390.

[Basili, 1994] Basili, V., Caldiera, G., Rombach, H. *The Experience Factory.* In: MARCINIAK, John J. Encyclopedia of Software Engineering. New York: John Wiley & Sons, 1994.

[Basili, 2002] Basili, R. et al. *Lessons learned from 25 years of process improvement:The Rise and Fall of the NASA Software Engineering Laboratory.* In: Proceedings of the ICSE-International Conference on Software Engineering, Orlando. USA, 2002.

[Beck, 2000] Beck, K., Fowler, M. *Planning Extreme Programming.* Addison Wesley Professional, The XP Series, 2000.

[Beck, 2002] Beck, K., Miller, R. *Extreme Programming Applied: Playing to Win.* Addison Wesley Professional, The XP Series, 2002.

[Beck, 2003] Beck, K., *Test-DrivenDevelopment By Exemple.* The Addison Wesley Signature Series, 2003.

[Beck, 2004] Beck, K., Andres, C. *Extreme Programming Explained: Embrace Change.* Addison Wesley Professional, 2nd edition. 2004.

[Booch, 2000] Booch, G., Rumbaugh, J., Jacobson, I. *UML – Guia do Usuário.* Editora Campus, 2000.

[Brady, 2010] Brady, A., Menzies, T., Keung, J., El-Rawas, O., Kocaguneli, E. *Case-based reasoning for reducing software development effort.* Journal of Software Engineering and Applications, 2010.

[Carleton 1992] Carleton A., Park R., Goethert W., Florac W., Bailey E., Pfleeger S.L. *Software Measurement for DoD Systems: Recommendation for Initial Core Measures.* CMU/SEI-92-TR-19, ESC-TR-92-19, Pittsburgh, PA, Software Engineering Institute, Carnegie Mellon University, September 1992.

[Chrissis, 2003] Chrissis, B. et al. *CMMI®: Guidelines for Process Integration and Product Improvement.* Boston, Addison Wesley, 2003.

[Clover] Atlassian, "CLOVER – Coverage code". Em: http://www.atlassian.com/software/clover/. Último acesso em: março de 2011.

[Coad, 1999] Coad, P., Lefebvre, E., De Luca, J. Java Modeling In Color With UML: *Enterprise Components and Process.* Prentice Hall, 1999.

[Cockburn, 2001] Cockburn, A., Highsmith, J. *Agile software Development: The business of Innovation.* In IEEEComputer Magazine, 2001.

[Cockburn, 2004] Cockburn, A. *Crystal clear: a human-poweredmethodology for small teams.* Boston: Addison-Wesley, 2004.

[CodePro] *CodePro Analytix: Automated Java software quality and testing tools for Eclipse developers.* CodePro Software Products. Disponívem em: http://www.instantiations.com/

[Coffin, 2011] Coffin, R., Lane, D. *A Practical Guide to Seven Agile Methodologies – Part 2,* 2011 Disponível em: http://www.devx.com/architect/article/32836/1954. Último acesso em: setembro/2011.

[Cohen, 2003] Cohen, D., Lindvall, M., Costa, P. *Agile Software Development,* DACS Stateof–the–Art. 2003. Disponível em: http://www.dacs.dtic.mil/techs/agile/ agile.pdf Último acesso em: julho/2010.

[Crispin, 2009] Crispin, L., Gregory, J. *Agile Testing: A practical guide for testing agile teams.* Addison-Wesley, 2009.

[Denning, 2005] Denning, P. J., *Is Computer Science Science?* In: ACM Communication April, 2005.

[Dora, 2010]Dóra, P., Oliveira, A. *Avaliação da Qualidade de um Processo de Desenvolvimento de Software.* Publicado: Google Docs – Forms. Disponível em: https://docs.google.com/spreadsheet/viewform?formkey=dEpZYm9MSC1raUZCSkpWUkc2TjEweVE6MQ. Último acesso em outubro de 2011.

[DSDM, 2009] Site Oficial. Disponível em: http://www.dsdm.org. Acesso em janeiro de 2011.

[Eclipse] Site Oficial. Disponível em: HTTP://www.eclipse.org. Acesso em: fevereiro, 2010.

[Fenton, 2009] Fenton, N., Neil, M., Marsh, W., Hearty, P., Radlinski, L., Krause P. *Project data incorporating qualitative factors for improved software defect prediction.* In PROMISE'09. Vancouver, Canadá, 2009.

[Flanagan, 2005] Flanagan, C., Godefroid, P. *Dynamic partial-order reduction for model checking software. SIGPLAN Not.* 40, 1, 110-121, 2005. DOI= http://doi.acm.org/10.1145/1047659.1040315.

[Fuggetta, 2000] Fuggetta, A. *Software Process: A Roadmap.* In: FINKELSTEIN, Anthony, (Ed.) The Future of Software Engineering, New York, ACM Press, 2000.

[Gomes 2001] Gomes, A., Oliveira K., Rocha, A.R. *Avaliação de Processos de Software Baseada em Medições.* Em: Simpósio Brasileiro de Engenharia de Software. Rio de Janeiro, RJ, 2001.

[Google] Site Oficial. Disponível em: www.google.com. Último acesso em: outubro, 2011.

[Google Forms] Site Oficial. Disponível em: www.docs.google.com. Último acesso em: agosto, 2010.

[Guerra, 2002] Guerra, A., S'antana, M. *Quality of Software Process or Quality of Software Product?*, In: International Conference on Software Quality, Canada, 2002.

[Highsmith, 2002] Highsmith, J. *Agile Software Development Ecosystems.* Boston: Addison Wesley, 2002.

[Humppherey, 1989] Humphrey, W. *Managing the Software Process*, Addison-Wesley, 1989.

[Humppherey, 1994] Humphrey, W. *A Personal Commitment to Software Quality.* Pittsburgh, PA: The Software Engineering Institute, 1994. Disponível em: http://www.sei.cmu.edu.

[IEEE, 2010] Mellor, S. *To Discuss Advanced Agile.* Press Room. IEEE. 31 March 2010. Disponível em: http://www.computer.org/portal/web/pressroom/2010/mellor. Último acesso: setembro de 2011.

[ISO/IEC, 1995] ISO/IEC 12207: *Information Technology - Software Life-Cycle Processes*, 1995.

[ISO/IEC, 2003] ISO/IEC 15504-1: *Information Technology – Process Assessment, Part 1: Concepts and Vocabulary, 2003.*

[Jacobson, 1999] Jacobson, I., Rumbaugh, J., Booch, G. *The Unified Software Development Process*. Addison-Wesley, 1999.

[Jira] Atlassian, *JIRA - Bug tracking, issue tracking and project management software.* Disponível em: http://www.atlassian.com/software/jira/docs/latest/. Último acesso em

[JUnit] *JUnit: Resources for Test Driven Development.* JUnit.org. Disponível em: http://www.junit.org/.

[Kerlinger, 1979] Kerlinger, C., Taylor, R. *Marketing research: an applied approach.* Tóquio: McGraw-Hill Kogakusha, 1979.

[Kitchenham, 1995] Kitchenham, B., Pickard, L., Pfleeger, L. *Case studies for method and tool evaluation*, In: IEEE Software, V. 12, 1995.

[Kitchenham, 2002a] Kitchenham, A., Pfleeger, L. *Principles of Survey Research Part 2: Designing a Survey*. In: *Software* Engineering Notes, 2002.

[Kitchenham, 2002b] Kitchenham. et al. *Preliminary Guidelines for Empirical Research in Software Engineering.* In: IEEE Transactions on *Software* Engineering, 2002.

[Larman, 2004] Larman, C. *Utilizando UML e Padrões: uma introdução à análise e ao projeto orientado a objetos e ao Processo Unificado*. Trad. Luiz Augusto Meirelles Salgado e João Tortello. 2 ed. Porto Alegre: Bookman, 2004.

[Lean] *Lean Enterprise Institute*. Lean.org. Disponível em: http://www.lean.org/.

[Lima, 2006] Lima, A., Cirne, W., Brasileiro, F., Fireman, D., *Case for Event-Driven Distributed Objects*. In: OTM Confederated International Conferences, CoopIS, DOA, GADA, and ODBASE, 2006.

[Manifesto Ágil, 2001] Beck, K.; Beedle, M.; van Bennekum, A.; Cockburn, A.; Cunningham, W.; Fowler, M.; Grenning, J.; Highsmith, J.; Hunt, A.; Jeffries, R.; Kern, J.; Marick, B.; Martin, R. C.; Mellor, S.; Schwaber, K.; Sutherland, J.; and Thomas, D. *Manifesto for agile software development.* http://www.agilemanifesto.org. Último Acessado em: dezembro de 2010.

[Marchenko, 2008] Marchenko, A., Abrahamsson, P. *Scrum in a Multiproject Environment: An Ethnographically-Inspired Case Study on the Adoption Challenges*, Toronto, Canada, IEEE Computer Society, 2008.

[Mellor, 2002] Mellor, S., Balcer, M. *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley, 2002.

[Mellor, 2004] Mellor, S. Scott, K., Uhl, A., Weise, D. *MDA Distilled*. Addison-Wesley, 2004.

[Mellor, 2011] Blog Mellor, S. Disponível em: http://www.stephenmellor.com/bio.html. Último acesso em: outubro de 2010.

[Metrics] *Eclipse Metrics plugin* Em: http://sourceforge.net/projects/metrics.

[Nebulon, 2004] Neubulon Pty. Ltd. *Feature Driven Development(FDD)*, 2002-2004, Disponível em: http://www.featuredrivendevelopment.com

[Oliveira, 2007] Oliveira, M. *OurBackup: Uma Solução P2P de Backup Baseada em Redes Sociais* In: Coordenação de Pós–Graduação em Informática, Universidade Federal de Campina Grande, 2007.

[OpenView] Site official. Disponível em: http://openviewpartners.com/ Último acesso em: maio de 2001.

[OurGrid] *The OurGrid Community.*OurGrid.org. Diponível em: http://www.ourgrid.org/.

[*OurProcess*] *OurProcess*. Disponível em: http://twiki-public.lsd.ufcg.edu.br/twiki-public/bin/view/LSD/LSDProcessoDesenvolvimentoAntigo. Último acesso em: outubro de 2011.

[*OurQualityProcess*] *OurQualityProcess*. Disponível em: http://twiki-public.lsd.ufcg.edu.br/twiki-public/bin/view/LSD/LSDProcessoDesenvolvimento. Último acesso em: outubro de 2011.

[Palmer, 2001] Palmer, R., *A practical guide to feature-driven development.* Pearson Education, 2001.

[Palmer, 2002] Palmer, S. R., Felsing, J. M. *A Practical Guide to Feature-Driven Development*. Prentice Hall, 2002.

[Poppendieck, 2003] Poppendieck, M., Poppendieck, T. *Lean software development: An agile toolkit*. Boston, MA, USA, Addison-Wesley Longman Publishing, 2003.

[Poppendieck, 2006] Poppendieck M., Poppendieck, T. *Implementing Lean Software Development: From Concept to Cash*. Addison-Wesley Professional, 2006.

[Pressman, 2006] Pressman, R. *Engenharia de Software*. 6º ed. Rio de Janeiro: McGraw-Hill, 2006.

[Pfleeger, 1999] Pfleeger, L., *Albert Einstein and Empirical Software Engineering*. In: IEEE Computer, V. 32, 1999.

[Pfleeger, 2001] Pfleeger, L., Kitchenham, *A. Principles of Survey Research - Part 1: Turning Lemons into Lemonade*, Software Engineering Notes, 2001.

[Reis, 2002] Reis, R., Reis, C., Nunes, D. *Automação no Gerenciamento do Processo de Engenharia de Software*. EIN'2002 – Escola de Informática Norte. Belém: 2002. Disponível em: http://www.labes.ufpa.br/quites/teaching/2006/TopicosES/ProcessoEngenhariaSoftware.pdf. Último Acesso em maio de 2010.

[RSC, 1998] Development Teams. Rational *Software Corporation White Paper TP026B*, Rational Rose. 1998. Rev. 11/2001.

[RUP, 1998] Rational *Software, Rational Unified Process: Best Practices for Software.* 1998.

[Rutherford, 2006] Rutherford, M. J., Carzaniga, A., Wolf, A. L. *Simulation-based test adequacy criteria for distributed systems*. In Proceedings of the 14th ACM SIGSOFT international Symposium on Foundations of Software Engineering (Portland, Oregon, USA, November 05 - 11, 2006). SIGSOFT '06/FSE-14. ACM, New York, NY, 231-241. DOI= http://doi.acm.org/10.1145/1181775.1181804.

[Scrum] *Scrum*. Disponível em: http://www.scrum.org/storage/Scrum%20Update%202011.pdf. Último acesso em: junho de 2011.

[Shwauber, 1995] Shwaber, K. *Scrum development process.* In: Workshop on Business Object Design and Implementation, Tenth Annual Conference on Object-Oriented Programming System, Linguages and Applications (OOPSLA'95). Austin, Texas, 1995.

[SOFFTEX, 2005] SOFTEX, *MPS.BR - Melhoria de Processo do Software Brasileiro, Guia Geral Versão 1.0*. Campinas, SP, 2005.

[Sato, 2007] Sato, D. *Uso eficaz de métricas em métodos ágeis de desenvolvimento de software.* Dissertação de Mestrado. Orientador: Prof. Dr. Alfredo Goldman vel Lejbman, São Paulo, Brasil, 2007.

[Sommerville, 2007] Sommerville, I. *Engenharia de Software*. PEARSON EDUCATION DO BRASIL. 8ª edição, 2007.

[Strode, 2005] Strode, E. *The Agile Methods: An analytical comparison of five agile methods and an investigation of their target environment.* Dissertação de Mestrado Massey University, New Zeland. Orientador: Professor Dr.Alexei, Tretiakov, 2005. Disponível em: http://mro.massey.ac.nz/bitstream/handle/10179/515/02whole.pdf?sequence=1. Último acesso em: maio de 2010.

[Travassos, 2002] Travassos, G., Gurov, D., Amaral, E., *Introdução à Engenharia de Software Experimental*. Relatório Técnico ES 590/02. Rio de Janeiro, PESC/COPPE/UFRJ, 2002.

[Thomas, 2006] Thomas, S., *An agile comparison*. 2006. Disponível em: http://www.itsadeliverything.com/agile-comparison.htm. Último acesso em: fevereiro, 2010.

[Udo, 2003] Udo, N., Koppensteiner, S. Will. *Agile changesthe way we manage software projects?* In: Agile from a PMBox guide perspective. Projectway. 2003.

[Vicente, 2010] Vicente A., Delamaro, M., Maldonado, C., *Uma revisão sistemática sobre a Atividade de Teste de software em Métodos Ágeis.* In: XXXV Conferencia Lationamericana de Informática, (XXXV CLEI), Pelotas – RS, Brasil, 2009.

[Voight, 2004] Voight, J. *Dynamic system development method.* Seminar of Department of Information Technology, University of Zurich, 2004. Disponível em: https://files.ifi.uzh.ch/rerg/amadeus/teaching/seminars/seminar_ws0304/14_Voigt_DSMD_Ausarbeitung.pdf. Último acesso em: junho de 2011.

[Wohlin, 2000] Wohlin, C. et. al., *Experimentation in Software Engineering – An Introduction. Massachusets*. Kluwer Academic Publishers, 2000.

[XP1, 2002] XP1: Um Processo de Desenvolvimento. Em: http://www.dsc.ufcg.edu.br/~jacques/cursos/2002.2/projii/xp1/xp1.html. Último acesso em: dezembro de 2011.

[XPlanner] XPlanner Organization. Em: http://www.xplanner.org/. Último acesso em: novembro de 2009.

[Zelkowitz, 1998] Zelkowitz, V., Wallace, R. *Experimental Models for Validating Technology*. IEEE Computer, 1998.

[Zelkowitz, 2003] Zelkowitz, V., Wallace, R., Binkley, W. *Experimental Validation of New Software Technology*, Lecture Notes on Empirical Software Engineering, World Scientific Publishing, 2003.

APÊNDICE A

A. Processo de desenvolvimento de *Software*

A.1. Principais assuntos do capítulo

Processo de Desenvolvimento de *Software*

Visão geral sobre os processos

Tipos de processos

Metodologias Ágeis - Seus fundadores

Exemplos de processos ágeis

OurProcess

A.2. Os Processo de Desenvolvimento de Software

Existem diversas definições para processos de desenvolvimento de *software*, a definição adotada nesse trabalho foi exposta por [Fuggetta, 2000] onde processo é definido como: "Um conjunto coerente de políticas, estruturas organizacionais, tecnologias, procedimentos e artefatos que é necessário para conceber, desenvolver, implantar e manter um produto de *software*".

Outra definição, mais simplificada, apresentada em 2006, por Pressman, é:

"Processo é um arcabouço de tarefas que são necessárias para a construção de *software* de qualidade" [Pressman, 2006].

Mas há um ponto que todos os pesquisadores desta área convergem é que um fator decisivo para a construção de um *software* confiável e de qualidade é a obtenção de um processo de desenvolvimento de *software*. [Pfleeger, 2001].

Para auxiliar a elaboração de processos existem modelos e normas em prol da qualidade dos processos de *software* como ISO/IEC 12207 [ISO/IEC, 1995], ISO/IEC 15504 [ISO/IEC, 2003], CMMI [Chrissis, 2003] e MR-MPS [SOFTEX, 2005], que servem como modelo para auxiliar a definição de processos.

Embora as normas sirvam como guia para a elaboração dos processos, não há uma definição exata de como deve ser um processo de desenvolvimento, eles sugerem atividades, artefatos de saída como: documentação, produtos, relatórios, entre outros indicativos. Porém, a definição dos processos é de competência das organizações para que as peculiaridades e particularidades possam ser modeladas através do processo (por exemplo, a estrutura, o tipo e o tamanho da organização, o domínio do produto, a complexidade dos projetos).

De acordo com Chrissis "... um processo deve ser moldado de acordo com as peculiaridades de cada organização e dificilmente um processo desenvolvido para uma determinada organização poderá ser aplicado, sem alterações, em outra organização". [Chrissis,2003].

Além dos modelos e normas como facilitadores à elaboração dos processos de desenvolvimento, é necessário identificar a natureza dos processos, o tamanho das equipes, o grau de complexidade dos produtos, além de outros fatores que serão decisivos para definir o tipo de desenvolvimento e, consequentemente, o modo como o processo será definido e aplicado, por exemplo: processos tradicionais ou processos ágeis?

A.3. Uma visão geral sobre tipos de desenvolvimento

Essa seção tem a função de apresentar uma visão geral e superficial, apenas para contextualizar o leitor sob o estado da arte da Engenharia de *Software* e mais especificamente na evolução do desenvolvimento de *software*.

O termo "processo de desenvolvimento de *software*" surgiu inicialmente com a definição de Gimenes, em 1994, a partir da definição de processo, como: "*a realização sistemática de uma ou mais tarefas, com o objetivo de se criar um produto*".

Posteriormente, Reis em 2002 define o termo processo de desenvolvimento de *software* ou processo de engenharia de *software* como: "*como o conjunto de atividades necessárias à transformação dos requisitos do usuário em um produto de software*". [Reis, 2002].

E Sommerville, simplifica a definição para: *"conjunto de atividades que leva à produção de um produto de software"*. [Sommerville, 2007].

Os processos devem ser definidos de acordo com o tipo de *software* desenvolvido. Há basicamente uma divisão entre processos tradicionais e processos ágeis.

Uma visão geral de processos tradicionais e processos ágeis.

Processos Tradicionais

Os processos de desenvolvimento tradicionais se caracterizam por estruturarem o desenvolvimento em fases distintas, onde cada fase tem um propósito único, seja especificação, projeto, análise, verificação, evolução ou outras de acordo com o método utilizado pela organização. As fases do desenvolvimento proporcionam um fechamento e uma visão direcionada e o pré-requisito para a inicialização de uma fase é que sua antecessora deverá ter sido finalizada e, consequentemente, um conjunto de artefatos de saída produzidos. Normalmente os processos ágeis não são regidos por fases bem definidas.

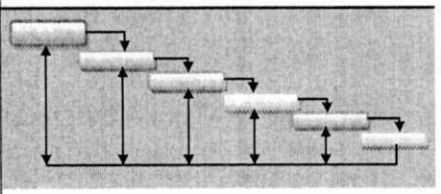
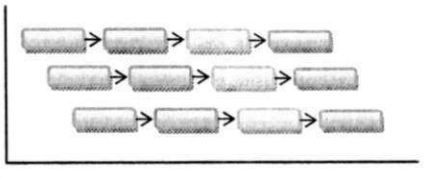
As principais características dos processos tradicionais são sua linearidade, sequencialidade, dependência entre as fases, inflexibilidade e rigidez tanto nos requisitos quanto no processo. De modo geral, essas características dos processos tradicionais destinam-se a projetos onde os requisitos são bem definidos e as mudanças pouco frequentes ou onde a complexidade é muito alta.

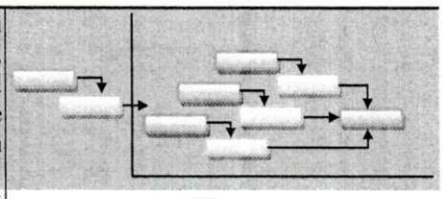
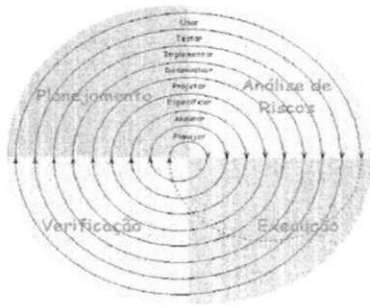
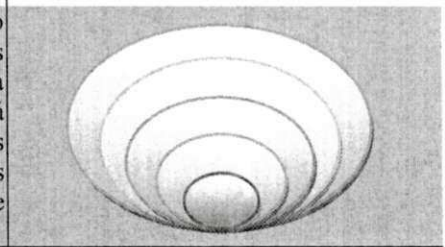
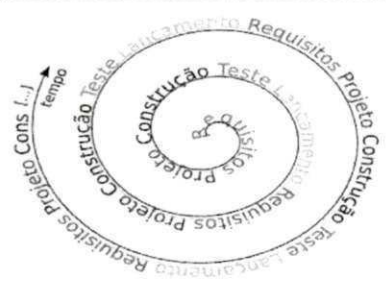
Processos Ágeis

A natureza ágil proporciona maior flexibilidade, adaptação a mudanças, foco na saúde do código e interação constante com o cliente. Com isso as metodologias ágeis prevêem processos mais flexíveis, com foco no indivíduo. Porém essa flexibilidade pode ser um ponto suscetível a falhas, visto que interpretações equivocadas ou mal adaptadas à realidade da organização não garantirão a qualidade dos produtos desenvolvidos.

Alguns modelos de processos consagrados podem ser vistos na Tabela 25.

Tabela 25: Exemplos de Modelos de Processos

Nome do Modelo	Descrição Simplificada	Representação do Modelo
Cascata	O mais antigo, segue a abordagem iterativa para o desenvolvimento de software através de um modelo seqüencial, através das fases de análise de requisito, projeto, implementação, teste (validação), integração, e manutenção de software.	
Incremental	Abordagem incremental, criado para complementar o modelo cascata, é considerado o mais tradicional em vários ciclos, cada ciclo, ou fase, pode ser implementado paralelamente e, no final da implementação de todas as funcionalidades, há a ocorrência de uma integração única.	

RAD	Desenvolvimento rápido de aplicação também se enquadra na filosofia iterativo e incremental, não tem um ciclo de vida padrão. Sua principal característica é ser rápido, isto é, os ciclos de desenvolvimento devem ser curtos e podem gerar protótipos das aplicações.	
Espiral	Modelo iterativo e incremental que foi desenvolvido com o objetivo de integrar todos os modelos existentes, ele combina elementos da prototipação, incorpora uma nova fase de análise de riscos, as fases de planejamento, verificação e execução. Cada fase deve contemplar o conjunto de oito atividades: planejar, analisar, especificar, projetar, desenvolver, implementar, testar e usar.	
Interativo e Incremental	É um desenvolvimento cíclico e cada ciclo representa uma interação, as partes são criadas separadamente e a cada entrega uma nova revisão ocorre, a última entrega contemplará todo o software e terá sido revisada tantas vezes quanto forem as interações. Dividido nas fases de concepção, elaboração, construção e transição.	
Ágil	Também é um desenvolvimento cíclico, interativo e incremental. A grande diferença está no envolvimento do cliente para validação das funcionalidades desenvolvidas. Os métodos ágeis compartilham algumas características comuns, incluindo o desenvolvimento iterativo, e um foco na comunicação interativa e na redução do esforço empregado em artefatos intermediários.	

A.4. Tipos de Processos

A.4.1. Metodologias Ágeis

A utilização dos Métodos Ágeis iniciou-se nos Estados Unidos e Europa e a maioria das suas características, como refatoração, programação em pares, adaptação a mudanças, a integração contínua, o desenvolvimento iterativo e a ênfase à atividade de testes surgiu a partir da cultura da comunidade *SmallTalk* desde a década de 80 [Sato, 2007].

No ano de 2001 dezessete pesquisadores (Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland e Dave Thomas) criaram o Manifesto para o

Desenvolvimento Ágil de *Software* (simplificadamente conhecido como Manifesto Ágil). Embora cada pesquisador tivesse seus métodos de trabalho todos idealizavam a simplificação do processo de desenvolvimento. Com isso foram definidos os princípios do Desenvolvimento Ágil:

"Estamos descobrindo maneiras melhores de desenvolver software, fazendo-o nós mesmos e ajudando outros a fazerem o mesmo. Através deste trabalho, passamos a valorizar:

Indivíduos e interações mais que processos e ferramentas.
Software em funcionamento mais que documentação abrangente
Colaboração com o cliente mais que negociação de contratos
Responder a mudanças mais que seguir um plano.

Ou seja, mesmo havendo valor nos itens à direita, valorizamos mais os itens à esquerda". [Manifesto Ágil – 2001].

A criação do Manifesto Ágil serviu como o guia para todos os métodos de desenvolvimento considerados ágeis [Beck, 2000], mas em 2006, Pressman afirma que:

"*a agilidade pode ser incorporada em qualquer processo de software*". [Pressman, 2006].

Com o movimento ágil em alta, alguns pesquisadores [Abrahamsson, 2002; Strode, 2005; Thomas, 2006] se dedicaram a comparar os métodos de desenvolvimento, categorizar os tipos de métodos de acordo com a natureza e complexidade dos produtos desenvolvidos e identificar semelhanças e diferenças entre eles.

A.4.2. Os fundadores do Manifesto Ágil

Cada um dos envolvidos no Manifesto Ágil continuou a disseminar esse novo método de desenvolvimento. Alguns desenvolveram metodologias próprias, como é o caso de Kent Beck com *eXtreme Programing* (XP), Alistair Cockburn (Família Crystal) e Ken Schwaber com *Scrum*.

Kent não atuou sozinho para fundar o XP, ele teve o auxílio de Ron Jeffries e Ward Cunningham. Ron é um dos autores de *Extreme Programming Installed*, o segundo livro o publicado sobre XP, e *Adventures Extreme Programming em C #*. Ward teve seu foco de

atuação na sua empresa *Cunningham & Cunningham, Inc.* (c2) e ainda participa ativamente do mundo *wiki*, isso porque foi o primeiro a desenvolver um *wiki* utilizando os preceitos ágeis para esse e outros dos seus projetos.

Já no caso de Mike Beedle, ele trabalha os princípios do desenvolvimento ágil utilizando *Scrum, XBreed* e XP, enquanto Arie van Bennekum desde 1994 se dedica ao *Dynamic Systems Development Method* (DSDM).

Martin Fowler participou das primeiras experiências com XP, foi co-autor do Manifesto Ágil e atua em uma grande empresa que utiliza métodos ágeis desde 2000. Participou também da escrita do livro *Planning Extreme Programming* [Beck, 2000] com Kent Beck e foi também autor de diversos livros na área do desenvolvimento de *software*, formalização da *Unified Modeling Language* (UML), entre outras [Beck, 2000; Beck, 2002; Beck, 2004].

Jim Highsmith é co-autor do Manifesto Ágil, membro-fundador da *Agille Alliance*, co-autor da Declaração de Interdependência para líderes de projeto e co-fundador e primeiro presidente da *Agile Project Leadership Network*. Jim é autor de diversos livros na área do gerenciamento de projetos e ganhador do prestigioso prêmio *Jolt Award* e do prêmio internacional *Stevens Award* 2005 por suas contribuições excepcionais para o desenvolvimento de sistemas. Destacou-se também pelas inúmeras consultorias para organizações de TI e de desenvolvimento de produtos e empresas de *software* em diversos países.

James Grenning desde 2001 se dedica ao tema *Test Driven Development* (TDD), atualmente no desenvolvimento ágil de sistemas embarcados desenvolvidos em C, tem uma empresa de consultoria chamada em português de "A Renascença em Desenvolvimento de *Software*".

Andrew Hunt e Dave Thomas trabalham juntos, fundaram a *Pragmatic Programmers* uma empresa ágil para treinamento de programadores, já escreveram mais de 100 livros, além das mais diversas publicações como: vídeos, áudios, revistas, fóruns.

Jon Kern, um dos co-autores do Manifesto Ágil, tem uma atuação em diversas empresas, centenas de projetos de diversas áreas de vários domínios, foi também co-autor do *Java Design*. Desde 1994 após se associar com Peter Coad participa ativamente da Metodologia Coad/C++.

Brian Marick atua efetivamente na área de teste de *software*, um dos precursores do Teste dirigido por Contexto, escreveu três livros: *The Craft of Software Testing, Everyday Scripting with Ruby*, e *Programming Cocoa with Ruby* e se dedica difundir a área de *Ágile Test*.

Robert C. Martin é o autor do premiado livro sobre design orientado a objetos e desenvolvimento Ágil: *Designing Object Oriented C++ Applications* usando o Método de Booch, além de fundador do CEO e presidente da *Object Mentor Incorporated* (uma empresa de consultores altamente experientes que trabalham com processos XP e ágil, design de *software*, treinamento e serviços de desenvolvimento de grandes corporações mundiais). Nos últimos 35 anos trabalhou com centenas de projetos de *software.* Ele é um ex-editor do Relatório de C++ e mensalemente escreve para a revista *Software Development Magazine* na coluna sobre o movimento de *Software Craftsmanship.*

Steve Mellor, hoje se denomina um "agente livre" (*freeter*) na área de desenvolvimento dirigido a modelos para sistemas embarcados. Foi por muitos anos cientista chefe da divisão de Sistemas Embarcados em *Mentor Graphics Corporation*. Sua participação no manifesto ágil foi motivada por ter fundado uma empresa e trabalhar com o objetivo de "*tornar o desenvolvimento de software racional, através de um processo controlável previsível.*" [Mellor, 2011].

Ele tem contribuído para o *Object Management Group*, que preside o consórcio que acrescentou ações executáveis para a UML e especificação do *Model Driven Architecture*. Ele também preside o conselho consultivo da revista IEEE *Software* [IEEE, 2010] e participa ativamente de conferências acadêmicas, cursos e diversas publicações importantes na área de sistemas embarcados. Dentre os livros que escreveu os mais recentes são: *Executable UML: A Foundation for Model Driven Architecture* [Mellor, 2002] e *MDA Distilled* [Mellor, 2004].

Ken Schwaber e Jeff Sutherland trabalharam juntos no desenvolvimento das primeiras versões do *framework Scrum* e em 1995 eles apresentaram, formalmente, o *Scrum* como processo formal para a comunidade científica no OOPSLA. Nos últimos anos eles tem se dedicado a implantar *Scrum* nas organizações, consultorias, treinamentos, avaliações, certificações para Scrum Masters, Desenvolvedores Scrum, Scrum Product Owners, além da formalização do Guia definitivo *Scrum* [Scrum].

Ken é um dos líderes do movimento ágil de desenvolvimento de *software*. Ele é fundador da *Agile Alliance*, e ele é responsável pela fundação da *Scrum Alliance* e criando o *Certified Scrum Master* programas e seus derivados, enquanto Jeff além de diretor executivo chefe da Scrum, Inc, em Boston, Massachusetts e Conselheiro Sênior do *OpenView Venture Partners* [OpenView].

A.4.3. Exemplos de processos seguindo a metodologia ágil

A.4.3.1. eXTREME PROGRAMMING – XP

O primeiro trabalho com programação extrema iniciou-se em 1996 por Kent Beck, Ron Jeffries e mais tarde com a participação de Ward Cuninghan, em discussões no wiki, isso porque Cuninghan foi o desenvolvedor do primeiro wiki e utilizou a sua criação para disseminar o XP. [Anderson, 1998].

A evolução do método ganhou forças com o manifesto ágil e hoje XP é um dos mais populares métodos ágeis. Os idealizadores queriam prover um método que pudesse ser facilmente aderido por pequenas e médias equipes de desenvolvimento de maneira simples, ágil, com garantia de qualidade e satisfação do cliente.

Muitos dos conceitos do XP foram fundamentados a partir dos elementos centrais da comunidade de *SmallTalk* que aderiam as práticas de: adaptação à mudanças; *feedback* constante do cliente, refatoramento de código; programação em pares; integração contínua; desenvolvimento iterativo e desenvolvimento focado na elaboração de testes automáticos.

As doze práticas do XP incluindo as derivadas do *SmallTalk* são:

Jogo de planejamento (*planning game*): inicia com a priorização do conjunto de funcionalidades definidas pelo cliente (mapeadas como estórias – *User Stories* - US), e, consequentemente, planejamento das iterações, tarefas, estimativa de esforço e tempo para implementar cada tarefa e a definição de um programador responsável para cada tarefa [Beck, 2002].

Testes (*testing*): Todo o desenvolvimento XP é guiado por teste, isto é, primeiro se testa e, posteriormente, se desenvolve código. Essa prática ocorre inclusive com as USs, no jogo de planejamento, o cliente elabora as USs e um teste automático (teste de aceitação)

também deve ser elaborado para verificar/validar a funcionalidade descrita na US implementada. A implementação da funcionalidade é finalizada quando o teste associado a ela for executado com sucesso. Os testes são incorporados na bateria de testes e executados frequentemente. Na metodologia XP os testes são a documentação executável do *software*, proporcionam maior segurança e coragem de mudar [Beck, 2000].

Programação em pares (*pair programming*): Todo o desenvolvimento em XP é feito em pares, isto é, uma estação de trabalho e dois programadores, onde o manuseio do mouse e teclado é alternado, frequentemente, e os pares revezados, periodicamente. A ideia é que enquanto um programador esteja atento à solução local o outro foque na solução global. Com isso espera-se um código revisado, com melhores práticas incorporadas, nivelamento da equipe, favorecimento à comunicação e a posse coletiva do código.

Refinamento do design (*refactoring*): Não existe uma etapa isolada de *design* em XP, o código é o *design* e deve ser melhorado continuamente através de refatoramentos. O refatoramento é um processo formal realizado para melhorar a estrutura interna do código e evitar a deterioração do código. Existem muitas técnicas de refatoramente que podem ser utilizadas sempre que indícios como duplicidade de código, queda de desempenho, classes muito longas, métodos muito longos ou difíceis de entender, uso excessivo de heranças, muitos parâmetros, acontecem. A existência prévia de testes é essencial porque elimina o medo de que o sistema irá deixar de funcionar por causa da mudança.

***Design* simples (*simple design*)**: O design está presente em todas as etapas do XP, o projeto começa simples e se mantém simples através de testes e refatoramento do *design*. A filosofia XP adota a prática do *design* simples a níveis extremos, isto é, não é permitido a implementação de função adicional, a qual não será usada na atual iteração. Além disso, o código deve ser o mais limpo possível, sem duplicidades, tem somente as classes e métodos necessários, primando pela organização e de acordo com a filosofia da simplicidade e reutilização.

Posse coletiva do código (*collective code ownership*): Toda a equipe é responsável por todo o código do sistema, portanto, qualquer dupla de programadores pode melhorar o sistema a qualquer momento. Com isso obtêm-se maior qualidade, menos riscos e menos dependência de indivíduos.

Integração contínua (*continuous integration*): Projetos XP mantem o sistema integrado o tempo todo e a integração ocorre várias vezes ao dia. Todos os testes (unidade e integração) devem ser executados antes da integração de código. Um código deve ser integrado quando houver sucesso na execução da bateria de testes. A prática estimula design simples, tarefas curtas, agilidade e oferece *feedback* sobre todo o sistema, permite encontrar problemas de *design* rapidamente.

Participação ativa do Cliente (On-Site Customer): O cliente tem um papel fundamental na metodologia XP, ele deve, continuamente, se reunir com a equipe de desenvolvimento para esclarecer questões, priorizar funcionalidades, utilizar o sistema e fornecer *feedback* preciso para os desenvolvedores.

Pequenos lançamentos (*small releases*): Denomina-se *release* a um conjunto de funcionalidades a serem desenvolvidas em um intervalo de tempo pré-definido. A cada finalização da implementação de um *release*, uma entrega deve ser efetuada. Consequentemente, há uma redução do risco, mudanças não geram um grande impacto no desenvolvimento, o *feedback* do cliente permitirá que problemas sejam detectados cedo, promoção da comunicação entre o cliente e o desenvolvedor, entre outros benefícios.

40 horas semanais (40-Hour Week): Projetos com cronogramas apertados na metodologia XP resultam em baixa produtividade. A ideia da semana de 40 horas foi elaborada para que não haja sobrecargas de trabalho e, com isso, os membros da equipe trabalham pelo período em que possam manter a qualidade. Desenvolvedores cansados causam mais erros e mais problemas para o sistema. O que XP prega é que muitas horas de trabalho desmotivam, cansam, resultam em baixa produtividade e, consequentemente,

produtividade baixa leva a código ruim, relaxamento da disciplina, dificuldade de comunicação, provocam irritabilidade e stress na equipe.

Padrões de codificação (*Coding Standards*): O código que tem um padrão de codificação evita que "argumentos estúpidos" sejam incorporados no código e fornece suporte a todas as outras práticas, promovendo a organização no código.

Metáfora (*System Metaphor*): Equipes XP devem preservar a comunicação para melhor interpretarem o *software* em desenvolvimento, de modo que como XP não possui uma arquitetura macro do sistema, os desenvolvedores devam fazer analogias, rabiscos, metáforas para facilitar a comunicação entre os membros da equipe e o cliente. [Beck, 2002; Beck, 2003].

Na versão 2004 alguns entendimentos foram aprimorados para uma visão mais positiva e inclusiva ao invés de somente práticas claras específicas para a programação. No aprimoramento da nova abordagem Kent Back afirma que: *"XP é sobre mudança social"* [Beck, 2004], e expande as 12 práticas iniciais para 24 práticas, 13 práticas primárias e 11 práticas corolárias para enquadrar a metodologia na nova filosofia de valorização de interações sociais e confiáveis.

Inicialmente os princípios (ou valores) do XP, eram: Comunicação, Simplicidade, Feedback e Coragem, após 5 anos de trabalho XP evolui e um novo princípio é incluído, o Respeito, muito embora Beck afirme que outros princípios são possíveis como: a equipe, o projeto como um todo (organização), e outros valores que podem ser definidos desde que toda a equipe esteja comprometida e de acordo com os novos princípios, os mais comuns de existirem são: segurança, manutenibilidade, previsibilidade, qualidade de vida, e tantos outros que se achar necessários. [Beck, 2004].

A visão de Beck é que entre os valores e as práticas existe um caminho a ser preenchido, que são os princípios, a Figura 8 mostra essa visão.

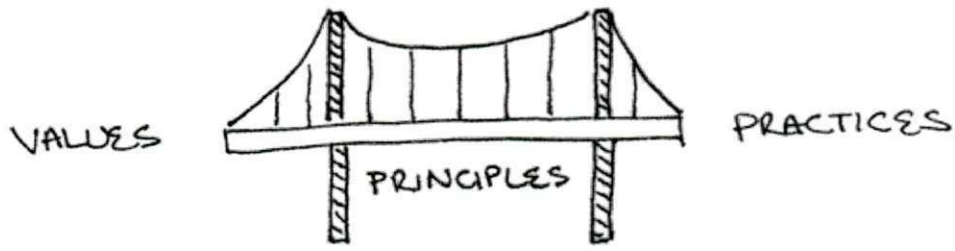


Figura 8: Valores, princípios e práticas XP – [Beck, 2004].

Uma consideração importante que Beck faz sobre a Figura 8 é que não basta conhecer os valores, princípios e práticas para ser um programador extremo, é necessário um envolvimento maior, interagir com praticantes. A aprendizagem é a utilização de técnicas intelectuais para traduzir valores em praticas.

Os princípios:

Humanidade: Analisar o que as pessoas precisam para serem bons desenvolvedores;

Economia: O desenvolvimento de *software* que não reconhece os riscos da economia pode alcançar somente o "Sucesso Técnico". Certifique-se de que o que você está fazendo tem valor comercial, atende os objetivos de negócio e serve às necessidades do negócio;

Benefício Mútuo: É o princípio mais importante do XP e o mais difícil de ser aderido, pois o benefício mútuo só é adquirido quando qualquer atividade executada deva beneficiar toda a equipe;

Auto-Semelhança: Faz comparações, reflete sobre similaridades de soluções e resulta em um amadurecimento na compreensão dos problemas;

Melhoria: Não existe projeto perfeito, sempre é possível melhorá-lo ao longo do tempo. A regra é fazer o melhor hoje e buscar sempre a continuidade da melhora;

Diversidade: Equipes XP precisam ter variadas habilidades, atitudes e perspectivas;

Reflexão: Equipes XP não somente fazem o trabalho, elas pensam sobre como estão fazendo e porque estão fazendo;

Fluxo: Fluxos são partes do todo. Para que o todo agregue valor os fluxos precisam agregar valor;

Oportunidade: Veja os problemas como oportunidade para mudanças, atitudes são o início da solução;

Redundância: Para um problema crítico e difícil onde existem várias formas de solução;

Falha: As falhas são inevitáveis, procure a solução mas, se não conseguir encontrar, falhe;

Qualidade: Sacrificar a qualidade nunca é um meio efetivo de controle, qualidade não deve ser considerada como uma variável de controle;

Passos Pequenos: Somente pessoas podem mudar rapidamente. Pequenos passos, pequenos riscos podem resultar em grandes saltos;

Aceitação da Responsabilidade: Responsabilidade não pode ser imposta, deve ser aceita. [Beck, 2004].

Outra alteração impactante na metodologia XP foi a retirada da restrição que XP é uma metodologia para pequenas e médias equipes. A nova abordagem afirma que XP pode ser usado em qualquer tamanho de equipe, preservando sempre a sua adaptabilidade a sistemas com requisitos vagos e mutáveis.

A.4.3.2. SCRUM

Para alguns autores Scrum não é só um processo, ele também é conhecido como *framework* de gerenciamento de projetos de *software* inclusive para outros tipos de projetos. A filosofia segue os princípios do Manifesto Ágil (**Pessoas e interações** são mais importantes que processos e ferramentas; ***Software* funcionando** é mais importante do que documentação extensa; **Colaboração com o cliente** é mais importante que contratos; **responder a mudanças** é mais importante que seguir um plano). O nome derivou do jogo de *Rugby*, onde um grupo de indivíduos trabalham para o bem comum, isto é, objetivo único.

Entre os anos de 80 e 90 alguns co-autores do manifesto trabalharam no desenvolvimento da metodologia Scrum. Foram eles: Ken Shwaber, Jeff Sutherland e posteriormente Mike Beedle e a apresentaram oficialmente no OOPSLA'95, *International Conference on Object-oriented Programing System, Linguages and Applications*. [Shwaber, 1995].

A popularidade do Scrum vem ganhando espaço no mercado de desenvolvimento de *software* e ganhou maior notoriedade depois que grandes empresas como Google (com o Gmail), Nokia e Microsoft (com o Xbox), Yahoo! (com o *Eletronic Arts*) entre outras como Intel, Toyota, Boeing, Toght Works, HP, Xerox e Globo adotaram a metodologia. [Marchenko, 2008].

O foco na satisfação do cliente parece que está ganhando adeptos e confiança, uma vez que o cliente pode acompanhar o projeto a cada passo (*Sprint*) além de ter a possibilidade de alterar/modificar os requisitos. Anteriormente as mudanças não eram bem vistas, os requisitos iniciais não eram alterados e, se houvesse a necessidade de alteração, os custos eram expressivos dado o foco no planejamento e não nas reais necessidades do cliente.

O gerenciamento é muito semelhante ao da metodologia XP, dividido em três etapas: Planejamento, *Sprints* e o encerramento. No Planejamento será definido a arquitetura inicial do sistema, macro estimativas e custos, além da criação do *backlog* (descrição dos requisitos e priorizações), da equipe e dos papeis (*Product Owner* – representante dos *Stakeholders* e *Scrum Master* – representante do sucesso da implantação do *Scrum*). Nos *Sprints,* que podem ocorrer uma ou duas vezes no mês, a equipe recebe parte do *backlog* a ser feito e este deverá ser mantido até o final do *sprint*, o qual deve gerar um conjunto de requisitos executáveis. Durante as iterações, diariamente, ocorre o monitoramento do andamento do desenvolvimento com as chamadas *stand-up meetings,* revisões (onde cortes podem ser aderidos e mudanças incorporadas no *backlog*) e retrospectivas (identificação dos pontos fortes e fracos da iteração). Na última fase ocorre a integração de todos os executáveis gerados, elaboração da documentação do usuário, preparação de treinamentos e material de marketing [Scrum].

Ciclo de vida Scrum

Cada atividade do scrum pode ser analisada na Figura 9. O processo inicia com a visão do produto (os requisitos do cliente), com o conjunto com todos os requisitos, com os seus graus de prioridades e com um tempo estipulado para realização do produto é gerado o *backlog* do produto. A cada nova iteração (*sprint*) o *backlog* é avaliado e alterações regulares ocorrem [Scrum].

A cada "rodada" um *sprint planning* 1 ocorre, onde o *Product Owner,* juntamente, com a equipe de desenvolvimento define quais serão os requisitos desenvolvidos e no *sprint planning* 2 há a definição da arquitetura, alocação de tarefas e estimativa de tempo. Após todas as definições, o sprint é desenvolvido e, obrigatoriamente, uma versão executável será gerada a partir de cada *sprint*. A partir do segundo *sprint,* uma fase de integração de *sprints* ocorre e o processo recomeça. Após as integrações de todos os *sprints* desenvolvidos, o produto final é gerado.

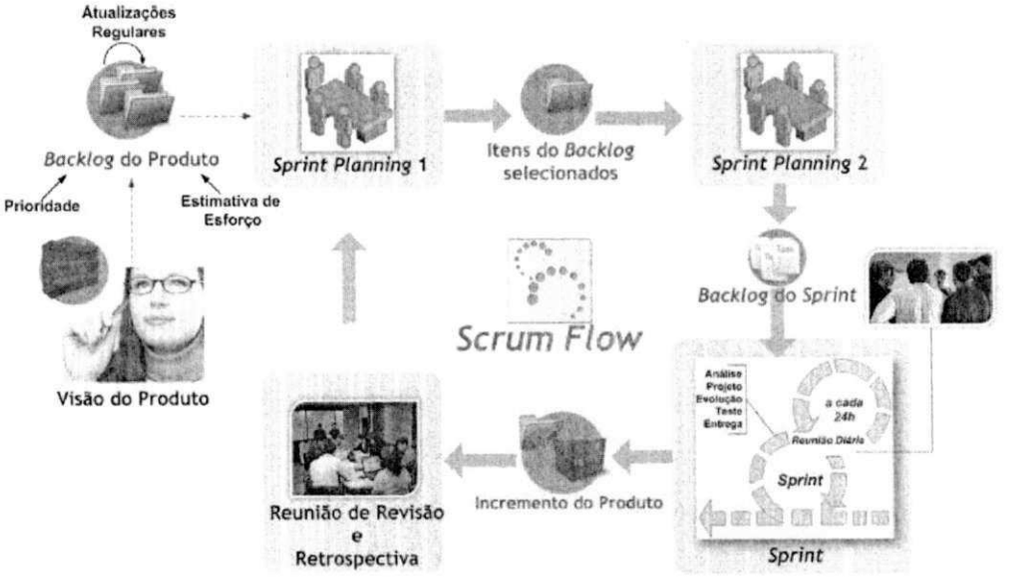


Figura 9: Ciclo de vida Scrum [Vicente, 2010].

A.4.3.3. LEAN SOFTWARE DEVELOPMENT

O *Lean Software Development* foi um processo fundamentado a partir dos moldes de desenvolvimento da TOYOTA [Poppendieck, 2003; Poppendieck, 2006]. Uma definição do processo é:

"*É uma estratégia de negócios para aumentar a satisfação dos clientes através da melhor utilização dos recursos. A gestão Lean procura fornecer constantemente valor aos clientes com o custo mais baixo (PROPÓSITO) através da identificação da melhoria dos fluxos de valores primários e de suporte (PROCESSO) por meio do envolvimento das pessoas qualificadas, motivadas e com iniciativas (PESSOAS)*". [Lean]

A mesma filosofia de XP e Scrum o Lean obedece: simplicidade, com foco nas necessidades do negócio. Tanto para o desenvolvimento de automotores como para o

desenvolvimento do *software* os processos são definidos de acordo com sete princípios [Poppendieck, 2006]:

Eliminar o desperdício: finalizar implementações, refatorar código, reuso de funções, testar para não refazer, priorizar *feedback* à documentação.

Amplificar o aprendizado: Dificuldades passadas devem ser fonte de conhecimento e devem incorporar ações, comportamentos para melhoria do processo.

Adiar comprometimentos e manter a flexibilidade: Quanto mais tarde uma decisão for tomada, maior será o entendimento sobre ela.

Entregar rápido: O cliente define melhor suas necessidade e gera maior compreensão sobre as reais necessidades, gera menores mudanças por insatisfações e/ou incompreensões.

Tornar a equipe responsável: Se os desenvolvedores são os responsáveis pela satisfação do cliente, eles devem ser inseridos nas tomadas de decisão. Cada desenvolvedor deve ter sua responsabilidade definida por tarefa.

Construir com qualidade - integridade: O desenvolvimento feito com responsabilidade deve gerar código de qualidade, íntegro, coerente, manutenível, adaptável e extensível.

Visualizar e otimizar o todo: Todo o desenvolvimento deve ser guiado visando a satisfação do cliente. Todo o aprendizado adquirido deve ser incorporado para futuros desenvolvimentos.

Figura 10: Ciclo de vida, visão alto nível [AgileCoop].

A Figura 10 representa o ciclo de vida do processo *Lean*, conforme pode ser visto existem duas visões. Na primeira sabe-se que todo o *release* desenvolvido tem um início, um fim e uma manutenção planejada. Antigos *releases* não são abandonados no processo. A cada novo release uma atividade de manutenção dos releases anteriores é planejada.

Figura 11: Ciclo de vida do Lean [AgileCoop].

Em outras metodologias ágeis apresentadas, normalmente, cada *release* dura quatro semanas, conforme pode ser percebido na Figura 11. Todas as atividades são guiadas pelos princípios apresentados anteriormente e, a partir da segunda tarefa executável, todas as outras são integradas para gerar a versão final.

A.4.3.4. Dynamic Systems Development Methodology – DSDM

O DSDM também segue a abordagem interativa e incremental familiar aos métodos ágeis, com a filosofia RAD (*Rapid Application Development*). O desenvolvimento do DSDM teve início nos 90 e foi lançado em 1995, na Inglaterra. Desde sua criação vem sendo aperfeiçoado e novas versões constantemente são lançadas.

O diferencial dessa metodologia é o foco nos projetos com prazos apertados e orçamentos restritos.

O DSDM é regido por nove princípios, que seguem:

- **Envolvimento**: foco do DSDM, a filosofia é focada em que quanto maior o envolvimento entre o desenvolvimento e o cliente maior o sucesso.
- **Autonomia**: desenvolvedores devem estar aptos a tomar decisões que sejam importantes para o progresso do desenvolvimento.
- **Entregas**: o princípio se sustenta na afirmação que quanto mais cedo o cliente receber uma parte do sistema, mais rápido será o feedback, menores falhas ou insatisfação serão incorporadas.
- **Eficácia**: é a priorização das funcionalidades mais importantes e também a garantia de que cada função desenvolvida corresponde com a expectativa/necessidade do cliente.
- **Feedback**: é o controlador do desenvolvimento, de acordo com os *feedbacks* que o desenvolvimento evolui.
- **Reversibilidade**: nenhuma ação deve ser irreversível.

- **Previsibilidade**: todo o desenvolvimento deve iniciar com um planejamento inicial, onde o escopo e os requisitos de alto nível serão definidos.
- **Ausência de Testes no escopo**: testes não fazem parte do escopo do desenvolvimento, são executados a parte do projeto.
- **Comunicação**: é um princípio essencial para o sucesso do projeto, através da comunicação que a cooperação é favorecida. [Voigt, 2004; DSDM, 2009].

É composto por 3 ciclos diferentes e iterativos (Figura 12): o primeiro é a divisão do projeto em pequenas "porções" com orçamento e prazo fixos; o segundo ciclo é a geração de protótipos utilizados como pré-testes de desempenho, usabilidade e evoluções, por exemplo: o terceiro ciclo representa os *workshops* que possibilitam uma melhor interação entre equipe de desenvolvimento e o cliente.

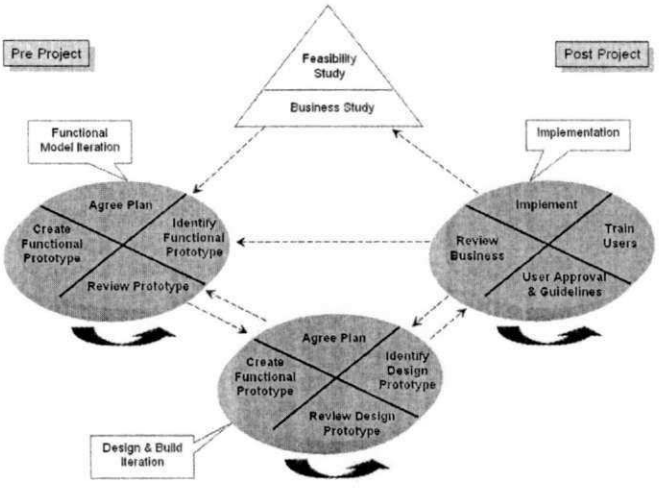


Figura 12: Ciclo de Vida dos projetos DSDM [Coffin,2011].

A.4.3.5. FEATURE DRIVEN DEVELOPMENT – FDD

O início do desenvolvimento guiado por funcionalidades foi motivado a partir das técnicas de desenvolvimento guiado por modelos (*Model-driven)* quando Peter Coad e Jeff De Luca, nos anos 90, tentando incorporar *Model-driven* no desenvolvimento ágil elaboraram o FDD e publicaram em 1999 a metodologia em um capítulo do livro "Java Modelling in Color with UML [Coad, 1999].

Como a intenção do FDD não é guiar todo o desenvolvimento, ele não é um processo completo [Palner, 2001]. Seus princípios e práticas mesclam entre processos ágeis e a

abordagem tradicional. O lema de FDD é: "Resultados frequentes, tangíveis e funcionais" [Palmer, 2002] e para isso todo o processo é guiado por um conjunto de cinco princípios: Desenvolvimento de um modelo geral; Construção da lista de características; Planejamento por característica; Projeto por característica e Construção por característica.

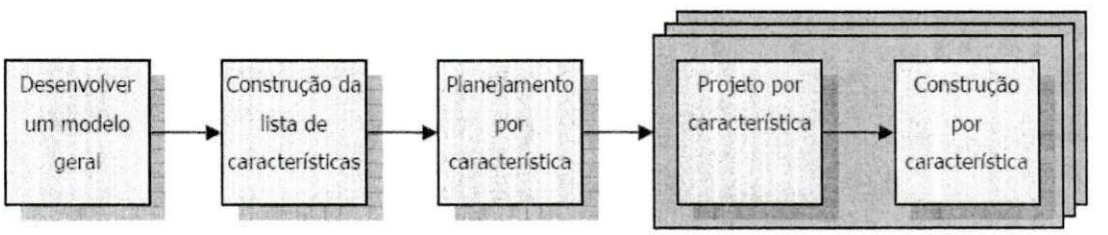


Figura 13: Os processos de FDD [Coad, 1999].

Mais detalhes de cada processo apresentado na Figura 13 podem ser obtidos no trabalho de Nebulon, [Nebulon, 2004], onde foi elaborada uma descrição completa com todos os critérios de entrada, descrição das tarefas associadas, verificação e validação e critérios de saída.

A.4.3.6. FAMAMÍLIA CRYSTAL

Quando Cockburn propôs a metodologia Crystal [Cockburn, 2001; 2004] ele não definiu um único método. Na verdade, ele fez uma junção de vários métodos, um para cada tipo de projeto. Os métodos são diferenciados pelas cores (*Color Method*) e dependendo da análise dos fatores de definição de método, a cor do método será definida. Os fatores de definição são: **Carga de comunicação** – definida de acordo com o número de participantes do projeto; **Criticidade do sistema** – que pode ser avaliado em relação ao conforto, dinheiro, dinheiro essencial ou à vida; e **Prioridade do projeto** – que pode ser avaliado por prioridade ou por tolerância.

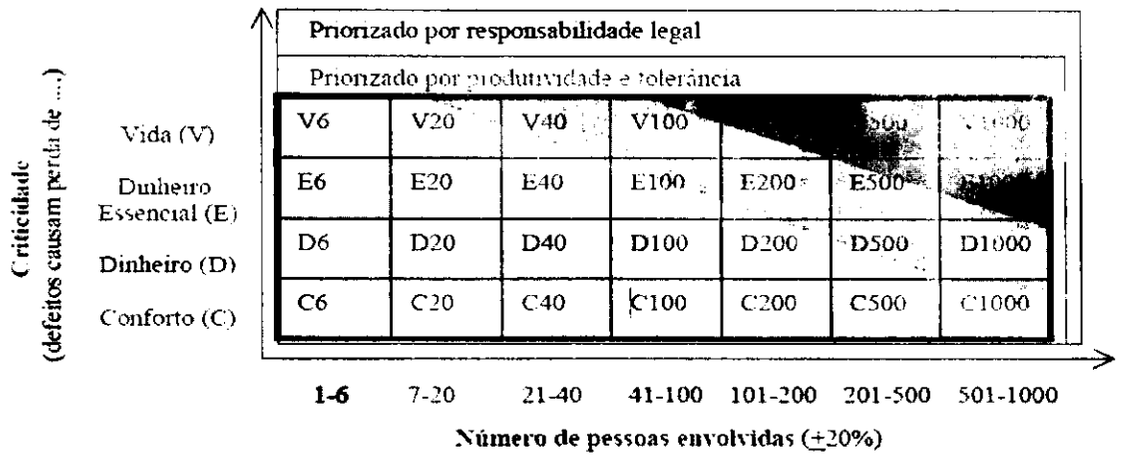


Figura 14: Esquema para definição da cor do método [Cohen, 2003].

Cohen [Cohen, 2007], orienta que para definir a cor do método é necessário analisar características como: tamanho da equipe (isso porque Crystal é escalável), duração das iterações (duração máxima para grandes projetos e/ou críticos), se existem equipes distribuídas (aplicação de práticas para equipes distribuídas) e o grau de criticidade dos sistemas (por exemplo, sistemas que lidam com risco de vidas ou valores monetários). Para facilitar a avaliação dos fatores Cohen elaborou uma tabela, Figura 14, para auxiliar a definição da cor do método.

Embora Crystal ofereça um aporte para a definição do tipo e necessidades do projeto, a definição do processo como a definição das atividades como executar o processo é de responsabilidade da equipe de desenvolvimento. Por isso, é muito comum equipes utilizarem a filosofia do Crystal com as práticas de outros métodos ágeis como XP e Scrum.

Independente de Crystal não apresentar um processo final, duas regras devem ser sempre respeitadas para a adoção da metodologia Crystal, são elas: ciclos incrementais (como a maioria dos métodos ágeis) e reuniões de reflexão para estimular a comunicação entre a equipe (de acordo com um dos princípios do manifesto ágil e aderido também pela maioria dos métodos) [Cockburn, 2004; Cohen, 2003; Udo, 2003].

A.4.3.7. Adaptative Software Development – ASD

Assim como DSDM, ASD também foi originada a partir de RAD por Highsmith em 1992. A criação do método ASD foi motivada para ser utilizado em projetos onde a

principal característica eram incertezas e alta complexidade, ou seja, requisitos não bem definidos.

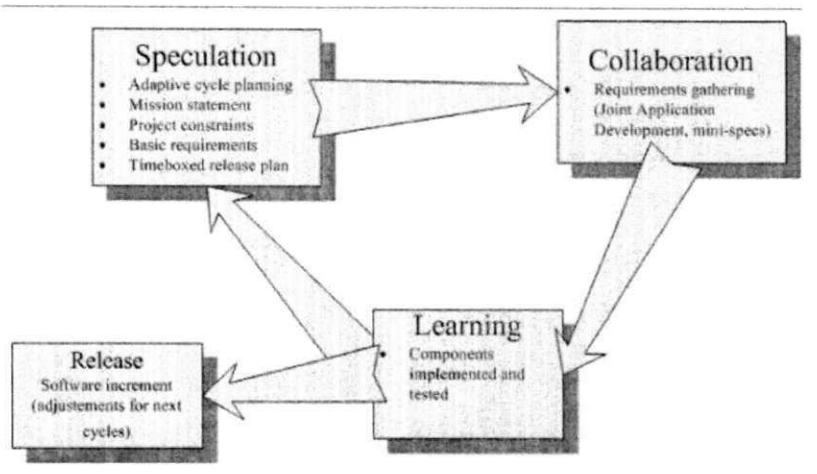


Figura 15: Ciclo de vida do médodo ASD [Highsmith, 2002].

Conforme visto na Figura 15, o ciclo de vida do ASD é composto por três princípios que geram um *release*, são eles:

Especulação: O planejamento é adaptativo e vai sendo construído de acordo com a evolução do desenvolvimento;

Colaboração: uma vez que o processo não tem fase de planejamento é essencial que haja colaboração para definir o que desenvolver, como e quando. O processo é fundamentado na criatividade de cada membro da equipe e na confiança mútua entre os membros;

Aprendizado: Como os requisitos não são bem definidos inevitavelmente falhas ocorrerão e as falhas no método ASD são esperadas e fazem parte do aprendizado sobre o sistema. Consequentemente, ações corretivas são constantes e necessárias. Além disso cada aprendizado adquirido em um ciclo é usado para melhorar o desenvolvimento dos próximos *releases*.

Embora não seja descrito como princípio do processo ASD, existe uma importante regra que é o tamanho de cada *release*, quanto menores forem os *releases*, menores serão as incidências de falhas e, consequentemente, menores os atrasos e maior o aprendizado.

Não diferente dos outros métodos, ASD também estimula o desenvolvimento incremental e interativo com entregas frequentes de um conjunto de requisitos executáveis para "colaboração" com o cliente. Para que seja possível prover organização no

desenvolvimento, o ASD fornece uma estrutura e um controle para garantir a colaboração entre a equipe e cliente. O gerente de projetos é o responsável por favorecer a colaboração e o aprendizado [Highsmith, 2002].

A.5. Procesos de Desenvolvimento

A.5.1. Person Software Process – PSP

O Processo Pessoal de *Software* nasceu com Humppherey em parceria com o SEI (*Software Engineering Institute*) [Humppherey, 1989]. A principal influência do PSP foi a partir da filosofia de maturidade de processos, isso porque essa parceria entre Humppherey e o SEI gerou também o CMM e o *Team Software Process* (TSP), processos que normatizam o desenvolvimento de *software*.

As principais diferenças são que:

CMM tem o foco na melhoria da capacidade organizacional e afeta em muito pouco as práticas pessoais de engenharia de *software* dos desenvolvedores.

PSP tem seu foco na melhoria contínua do processo de desenvolvimento do *software* e a difícil aplicação em pequenas equipes de desenvolvimento ou no nível individual.

O objetivo do PSP é condizente com a proposta do manifesto ágil "*as pessoas são mais importantes*" [Manifesto Ágil, 2001]. Por essa razão, PSP objetiva a transformação das pessoas em "melhores engenheiros de *software*", maximizando suas capacidades de planejamento, acompanhamento e qualidade nos resultados produzidos. Por focar nas pessoas, PSP pode ser utilizado tanto como ferramenta de uso geral para gerenciar as atividades pessoais particulares, quanto para o gerenciamento das atividades profissionais.

Todos os processos e métodos de desenvolvimento PSP são regidos por um conjunto de princípios que norteiam o desenvolvimento, quais sejam:

- Melhorar a estimativa de prazo e esforço;
- Melhorar o planejamento e o acompanhamento;
- Evitar o excesso de compromissos;
- Criar um comprometimento pessoal com a qualidade e com a melhoria contínua do processo.

O direcionamento fornecido pelos princípios do PSP provêem uma estrutura para o amadurecimento pessoal. Isso ocorre através da introdução de sete passos fundamentados na abordagem do CMM, descritos na Figura 16:

Figura 16: Processo PSP (imagem inspirada em [Humppherey, 1989]).

Abaixo segue uma descrição sumária dos níveis do PSP [Humpphrey, 1994]:

No nível de **Medição Pessoal**, a função principal é focada na melhoria das estimativas de tempo das atividades em cada etapa do ciclo do desenvolvimento e os defeitos encontrados. Para que essas atividades possam ser realizadas, o PSP indica a utilização de formulários adequados. A continuação do processo inicial (PSP0.1) especializa outros registros e acrescenta a utilização de um padrão para: codificação, medidas e preenchimento de formulário com proposta de melhoramento no processo.

No nível de **Planejamento Pessoal**, o foco é nas estimativas e no planejamento. O PSP1 foca em relatório de testes e estimativas de tamanho além da utilização de dados históricos para mensurar e comparar tarefas semelhantes. No 4º passo (PSP1.1) o foco é no planejamento de tarefas e na elaboração de cronogramas.

Os passos 5 e 6 são referentes à **Qualidade Pessoal** e em (PSP2) ocorrem as revisões pessoais de projeto e de código. A filosofia de PSP em relação aos erros/defeitos do código é: "*melhor prevenir do que remediar*". No 6º passo (PSP2.1) são tratados os critérios de melhoria do projeto, avaliação de técnicas de verificação e consistência de projeto.

O último nível do PSP é o **Processo Pessoal Cíclico** onde o foco é no desenvolvimento de grandes projetos, embora ainda em nível pessoal. A idéia é dividir os grandes programas em pedaços que possam ser tratados no PSP2. Neste caso, o desenvolvimento acontece em etapas incrementais.

A.5.2. Processo Unificado - PU

O processo de desenvolvimento PU é a junção de processos e dentre os seus criadores está Ivar Jacobson, que se desligou da ERICSON para se dedicar a sua empresa *Objectory* AB, onde desenvolveu o *Objectory* (uma mistura de processo e produto). Após um período *Rational* comprou a empresa de Jacobison e o contratou. Nessa nova junção de empresas foi desenvolvido o Processo *Objectory* da *Rational* (ROP) em paralelo com o Método Unificado, posteriormente denominado de UML. [RUP, 1998; Jacobson, 1999; Booch, 2000]

Buscando a melhoria do processo, uma nova versão foi desenvolvida, o conhecido *Rational Unified Process* (RUP), uma evolução continua do processo e, de acordo com as últimas publicações da *Rational* e, principalmente, de Jacobson, o fim do processo de desenvolvimento tradicional está próximo. Essa afirmação pode ser constatada com os novos produtos da RUP, o *EssencialUP* (com o desacoplamento entre disciplinas, atividades, artefatos e papéis) e a versão ágil de RUP, o *OpenUP* (que parece muito com as metodologias ágeis atuais). Enfatizando essa tendência surge recentemente a ferramenta *EssWork*, para ser usada em equipes "sem processos", apenas com um gerenciador do repositório das práticas, para organizar a forma de trabalho da equipe e para integrá-lo com as ferramentas de desenvolvimento da equipe.

Embora a introdução demonstre a evolução do PU, ele e todos os seus filhos foram fundamentados em três princípios:

Os princípios:

- **Incremental e iterativo**: dividido em mini-projetos, cada mini-projeto é uma interação e cada interação gera um incremento, com isso os idealizadores do processo pretendem reduzir riscos, obter robustez na arquitetura, prover iterações e aprendizado [Larman, 2004].

- **Direcionado a casos de usos**: os requisitos do sistema são representados por casos de uso. Cada caso de uso é a representação de uma sequência de ações que produzem um resultado de utilidade para um ator.
- **Entrada na arquitetura**: esse princípio pretende preservar a descrição das partes essenciais do sistema. É a partir da arquitetura que as decisões de organização do sistema como um todo, elementos estruturais, interfaces, comportamento, composição de elementos estruturais e comportamentais nos subsistemas são definidos.

O PU também é definido através de fases bem definidas, que são:

- **Inicialização**:
  - Planejamento;
  - Identificação dos riscos;
  - Definir subconjuntos e arquitetura candidata;
  - Estimar custos;
  - Estimar esforços;
  - Alocação de pessoal;
  - Definir padrões de qualidade;
  - Iniciar o *business case.*
- **Elaboração**
  - Identificar e reduzir riscos de construção;
  - Especificar, se não todos, a maioria dos *use-cases*;
  - Fixar a arquitetura em proporções executáveis;
  - Preparar o plano de projeto da próxima fase;
  - Estimar e justificar o orçamento;
  - Finalizar o *business case*;
- **Construção**
  - Desenvolvimento com foco na viabilidade incremental executável;
    - Análise; Projeto; Implementação; Testes; Integração.
  - Estender o modelo de *use case* para toda a aplicação;
  - Manter a integridade da arquitetura;
  - Monitorar os riscos (principalmente os críticos).

- **Transição**
    - Eliminar problemas e erros descobertos durante o desenvolvimento;
    - Geração do beta-teste;
    - Execução da versão beta-teste por alguns usuários definidos;
    - Treinamentos;
    - Documentação.

As quatro fases do processo PU também foram descritas por Pressman [Pressman, 2006] ressaltando que uma importante atividade do PU é o *feedback* recebido após o teste-beta.

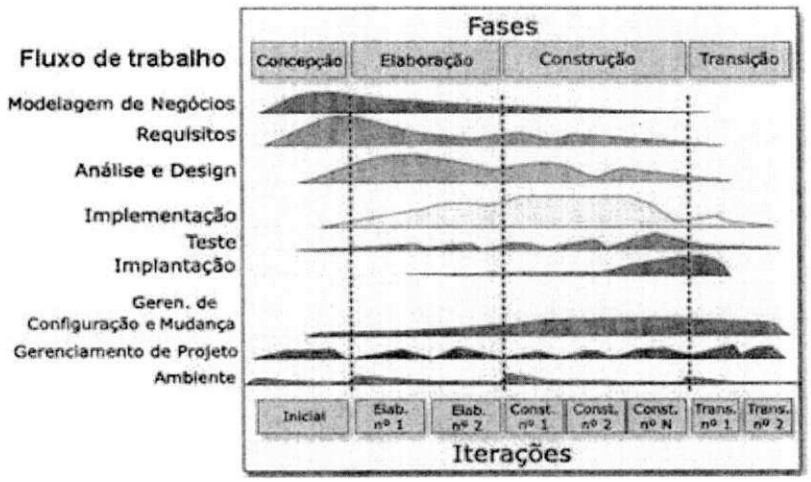


Figura 17: Ciclo de vida do processo PU [Jacobson, 1999].

O ciclo de vida do PU (Figura 17) reflete bem as quatro fases e a intensidade do fluxo de trabalho para cada atividade. A seguir é mostrada a descrição de cada atividade identificada na legenda à esquerda da Figura 17.

Modelagem de Negócios – pico no início do projeto e decai progressivamente após a fase de concepção;

Requisitos – O ponto máximo da elicitação de requisitos ocorre na transição das fases de concepção e elaboração, porém há uma atividade de revisão dos requisitos na fase de construção;

Análise e Design – ocorre com mais intensidade na fase de elaboração, embora na fase de construção também ocorra alguma atividade, isso porque revisões de design e análise são realizadas;

Implementação – durante as fases de elaboração e construção a implementação é desenvolvida, na elaboração o foco é organização e estrutura. Na fase de construção essa atividade tem uma intensidade maior e é possível notar também que a atividade de implementação na fase de construção é contínua e decai somente no final da fase de construção;

Teste – a atividade de teste é contínua com pequenos picos de atividade, sendo a maior no final da fase de construção;

Implantação – é construída do meio da fase de elaboração até a metade da fase de transição com um pico único que inicia no meio da fase de construção e começa a decair na transição das duas últimas fases;

Gerenciamento de Configuração e Mudanças – é a atividade mais constante de todo o processo PU, ela inicia no meio da fase de Concepção e finda no término do desenvolvimento e um leve aumento no esforço da atividade ocorre nas duas últimas fases do ciclo de vida do processo PU;

Gerenciamento de Projeto – semelhante a fase de Gerenciamento de configuração também é constante durante todo o ciclo de vida do processo, porém com sucessivos picos com intervalos semelhantes e intensidade de atividade similar;

Ambiente – Também é uma atividade constante com um pico no início de cada fase com intensidade e frequencia equivalente.

A.5.3. OurProcess (OP)

O *OP* é um processo baseado na metodologia XP1 [XP1, 2002]. Foi desenvolvido no cerne do ambiente acadêmico para direcionar, principalmente, trabalhos de mestrado e doutorado, muito embora diversos alunos de graduação utilizem o processo.

O processo é baseado nos princípios propostos por Beck [Beck, 2000] com algumas modificações necessárias realizadas com o intuito de simplificar a adoção de um processo em ambiente acadêmico. As principais adaptações realizadas foram adição de novos papéis ao processo de desenvolvimento: o gerente de produto, o líder técnico, o consultor, e líder de time.

Além disso, este processo inclui novas atividades para cada um destes papéis e realocação de atividades que antes pertenciam ao gerente:

2. **Alocar *reviewers* de testes**: Passa a ser de responsabilidade do líder de time juntamente com o *coach.*
3. **Manter o processo**: Esta tarefa fica sob responsabilidade do *coach.*

O objetivo principal do *OurProcess* foi proporcionar aos usuários um processo mais coeso para uso em equipes mais maduras, buscando continuamente a evolução com o objetivo de melhor atender às necessidades de projetos de *software* realizados em ambientes acadêmicos primando sempre pela simplicidade na incorporação das atividades do processo.

A.5.4. As diferenças entre OurProcess e XP1

Por ser baseado no XP1, o *OurProcess* preserva a agilidade, porém, faz-se necessário remover todas as regras e práticas que possam dificultar a adoção do processo. Ele foi concebido de acordo com a premissa que: "*fazer um aluno adotar um processo com 5 práticas é mais fácil do que fazê-lo adotar um processo com 10 práticas*". Essa remoção de práticas foi possível visto que algumas práticas de XP não se aplicam totalmente ao ambiente acadêmico. Em outras práticas foram necessários alterações em suas concepções iniciais, como o exemplo do *pair programming.*

O *pair programming* requer que a equipe esteja trabalhando sempre no mesmo lugar (pelo menos dois membros), o tempo todo para possibilitar a formação de pares, preferencialmente, com alternância entre os pares. Na maioria das vezes, não é possível reunir todos os alunos no mesmo horário. Como *pair programming* é opcional no processo, o *Review* de Testes foi colocado no seu lugar para não perder de todo os benefícios de *pair programming.*

A prática é recomendada quando uma funcionalidade com alta criticidade ou alta complexidade está sendo desenvolvida. A regra básica para o *pair programming* é: se uma tarefa foi desenvolvida com *pair programming*, não faça *test review* nela.

Em outras palavras, se alguém programou sozinho, alguma parte de uma tarefa, faça *test review.* Se for possível fazer *pair programming*, melhor, pois facilita a comunicação e, além do mais, remove o stress de "ver minhas coisas passarem por um *review*". *Pair programming* estressa a cooperação e treinamento ao invés da crítica.

Outra remoção foi o *design review* e o *code review*, isso porque o *OurProcess* amarra a elaboração de testes e, acredita-se que o *test review* seja suficiente para manter a qualidade do código. Se o *test reviewer* desconfiar da qualidade do que ele está vendo, pode mergulhar mais fundo e fazer um *review* de *design* e código.

Finalizando a descrição das remoções e iniciando a descrição das adições feitas no *OurProcess* tem-se uma substituição do método de planejamento das atividades. Há a exclusão do planejamento por escopo e voltando a planejar por tempo, conforme recomenda XP. Essa substituição além de motivar a vantagem de se encaixar nas restrições universitárias em que o semestre tem um fim fixo, outros benefícios foram pretendidos, os quais seriam:

- XP (e XP1) acredita em planejamento com tempo fixo por quê?
    - É muito mais fácil controlar o botão "escopo" do que controlar o botão "tempo". Então deixe o tempo fixo e controle o que é feito neste tempo.
    - Não tem planejamento para atrasos, já que o tempo é fixo. A negociação com o cliente não será por tempo. Idealmente, a data da entrega não muda, o que deve alterar é o plano de *release.* Porém, atrasos são factíveis caso o conjunto de funcionalidade mínima não possa ser entregue.

Para que essa alteração não afete a satisfação do cliente, sempre que uma entrega necessitar de ajustes, o cliente é o detentor do poder de corte das funcionalidades.

Além da remoção das práticas e ou a substituição há também a inserção de algumas práticas como:

- **Projeto arquitetural:** foi introduzido porque identificamos a necessidade de uma arquitetura para facilitar o planejamento do projeto e a elicitação dos requisitos.
- **Projeto do esquema lógico de dados:** foi introduzido como atividade destacada haja vista a dificuldade de alterar esquemas lógicos em bancos de dados populados. Com um planejamento antecipado, pretende-se minimizar erros lógicos.
- ***Test Review***: foi introduzido para garantir algum dos benefícios de *pair programming*, muito embora sempre que a prática de *pair programming* for realizada o papel de *test review* será sucumbido.