RENATO AFONSO COTA SILVA

PSP E MÉTODOS ÁGEIS NA MELHORIA DA QUALIDADE EM PRODUÇÃO DE SOFTWARE: UM ESTUDO DE CASO

Tese apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de "Magister Scientiae".

VIÇOSA
MINAS GERAIS - BRASIL
2006

Ficha catalográfica preparada pela Seção de Catalogação e Classificação da Biblioteca Central da UFV

T
S586p 2006

Silva, Renato Afonso Cota, 1981-
PSP e métodos ágeis na melhoria da qualidade em produção de software : um estudo de caso / Renato Afonso Cota Silva. – Viçosa : UFV, 2006.
xvi, 128f. : il. ; 29cm.

Inclui apêndice.

Orientador: José Luis Braga.
Dissertação (mestrado) - Universidade Federal de Viçosa.
Referências bibliográficas: f. 121-128.

1. Engenharia de software. 2. Software - Desenvolvimento - Administração. 3. Programação eXtreme. I. Universidade Federal de Viçosa. II.Título.

CDD 22.ed. 005.10684

RENATO AFONSO COTA SILVA

PSP E MÉTODOS ÁGEIS NA MELHORIA DA QUALIDADE EM PRODUÇÃO DE SOFTWARE: UM ESTUDO DE CASO

Tese apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de "Magister Scientiae".

APROVADA: 05 de abril de 2006

_____	_____
Alcione de Paiva Oliveira	Carlos Henrique Osório Silva
(Conselheiro)	(Conselheiro)
_____	_____
José Carlos Maldonado	Mauro Nacif Rocha

_____
José Luis Braga
(Orientador)

Feliz é o homem que acha sabedoria, e o homem que adquire entendimento.

Provérbios 3:13

Agradecimentos

- Agradeço em primeiro lugar a Deus, que me deu o dom da vida e a oportunidade de realização deste sonho. Graças a Deus pela sua graça, fidelidade e amor, os quais pude sentir durante todo o tempo dessa minha caminhada.

- Agradeço a minha família pelo apoio que me foi dado, em especial a minha mãe Helena, a qual em todo tempo esteve comigo, sofrendo e se alegrando junto comigo e me incentivando em todos os momentos a continuar firme no meu propósito. Ao meu pai Geraldo, pelo apoio financeiro, pelo incentivo e pelo carrão que me emprestou pra eu não ter que andar a pé. Valeu pai! Também não posso me esquecer da minha querida irmãzinha Viviane, que me deu muitas dicas de como se comportar em um mestrado, me passando suas experiências e incentivando a manter firme meu propósito. Também agradeço ao meu irmão Júnio, com o qual dividi quarto durante toda minha vida, e aqui não poderia ser diferente. Amo muito todos vocês, e essa conquista é nossa!

- Agradeço em especial ao Prof. José Luis Braga, por ter acreditado em mim, ter dedicado parte de sua vida a me orientar e por se tornar o meu pai aqui em Viçosa. Além de orientador você é um grande amigo e foi fundamental no meu processo de amadurecimento, me ensinando a enxergar a floresta, e não apenas as árvores isoladamente.

- Minha querida princesinha Juliana, você também é parte dessa conquista. Na verdade você é uma outra conquista que faz parte da minha vida. Você foi e é muito importante para mim. Não sei se agradeço a você por me ajudar a chegar ao fim do mestrado ou se agradeço ao mestrado por me fazer conhecer você. De

toda forma, agradeço o seu companherismo e carinho que tem dedicado a mim durante todo esse tempo.

- Ao meu grande amigo Bruno Rodrigues Silva. Companheiro desde a graduação ... Compartilhamos sonhos ... Agora compartilhamos vitórias.

- Agradeço também aos conselheiros, em especial ao Prof. Carlos Henrique Osório Silva, pelo apoio e empenho demonstrado para chegarmos aos resultados.

- À toda galera da Igreja Presbiteriana de Viçosa, pela receptividade que me ofereceram desde o primeiro dia de Viçosa. Vocês moram em meu coração.

- À galera da república, que me fez esquecer em alguns momentos os problemas do mestrado.

- Ao Ulisses Leitão, grande amigo desde os tempos de graduação. Obrigado por acreditar em mim e me apoiar para conseguir chegar até aqui. Valeu!

- Ao Departamento de Informática da Universidade Federal de Viçosa, aos professores e funcionários por me receber tão bem e pela amizade conquistada durante esse período e por permitir que eu utilizasse um gabinete para desenvolvimento desse trabalho. Em especial à Eliana e ao Altino, que sempre foram gentis, amigáveis e dipostos a ajudar.

- À DoctumTec e a CAPES pelo apoio financeiro.

- À Universidade Federal de Viçosa.

Biografia

Renato Afonso Cota Silva, filho de Geraldo Liberato da Silva e Helena Cota da Silva, brasileiro nascido em 20 de novembro de 1981 no município de Ipatinga, no Estado de Minas Gerais.

No ano 2000, após conculir o curso Técnico em Processamento de Dados na cidade de Ipatinga, ingressou-se no curso de graduação em Ciência da Computação nas Faculdades Integradas de Caratinga, onde se graduou no ano 2003. Em 2004 mudou-se para a cidade de Viçosa para cursar o mestrado em Ciência da Computação na Universidade Federal de Viçosa - UFV, onde se tornou o primeiro mestre do recém-criado curso de mestrado do Departamento de Informática - DPI, defendendo tese em abril de 2006.

Conteúdo

Lista de Tabelas x

Lista de Figuras xi

Resumo xiii

Abstract xv

1 Introdução **1**

1.1 O problema e sua Importância 4

1.2 Objetivos do Trabalho 5

1.3 Trabalhos Relacionados 6

1.4 Organização deste Documento 6

2 Revisão Bibliográfica **8**

2.1 Introdução 8

2.2 Qualidade de Software 9

2.2.1 Tratamento de Defeitos 13

2.3 Competência Organizacional 14

2.3.1 A Norma NBR ISO/IEC 12207 16

2.3.2 A Norma ISO/IEC 15504 17

2.3.3 CMM/CMMI 19

2.3.4 MPS.BR 24

2.4 Personal Software Process (PSP) 28

2.4.1 Fases do PSP 31

2.4.2 PSP e CMM 33

2.4.3 O Curso PSP 33

2.4.4 PSP no Ambiente Acadêmico 36

2.4.5 PSP no Ambiente Industrial 37
2.5 Test Driven Development (TDD) 39
2.6 Métodos Híbridos 42
2.7 Engenharia de Software Experimental 43

3 PSPm - PSP modificado **48**
3.1 Introdução 48
3.2 PSPm 49
3.2.1 Projetando o Software 50
3.3 Possíveis Melhorias no Processo 52

4 Metodologia, Resultados e Discussão **56**
4.1 Introdução 56
4.2 Protocolo Experimental 56
4.3 Metodologia 58
4.3.1 Treinamento dos Alunos 60
4.3.2 Coleta dos Dados 60
4.4 Análise e Interpretação 63
4.4.1 Notação e Definições 63
4.4.2 Hipóteses Estatísticas 65
4.4.3 *Bootstrap* - Teste por Reamostragem 66
4.4.4 P-valores e Intervalos de Confiança 67
4.4.5 Resultados e Discussão 68
4.4.6 Alunos com CA $\geq$ 80 72
4.4.7 Alunos com CA<80 73
4.5 Considerações Finais 75
4.5.1 PSPm e MPS.BR 75

5 Conclusões **78**
5.1 Trabalhos Futuros 81

A Exercícios do PSP - Série A **82**
A.1 PROGRAMA 1A 82

A.2 R1 - PADRÃO DE CONTAGEM DE LINHAS DE CÓDIGO 83
A.3 R2 - PADRÃO DE CODIFICAÇÃO 84
A.4 PROGRAMA 2A 84
A.5 PROGRAMA 3A 85
A.6 R3 - O RELATÓRIO DE ANÁLISE DE DEFEITO 85

B Dados Brutos **87**

B.1 Dados Coletados em 2004 - PSP 88
B.1.1 Tempo - Programa 1A - PSP 88
B.1.2 Tempo - Programa 2A - PSP 89
B.1.3 Tempo - Programa 3A - PSP 90
B.1.4 Defeitos Injetados - Programa 1A - PSP 91
B.1.5 Defeitos Injetados - Programa 2A - PSP 92
B.1.6 Defeitos Injetados - Programa 3A - PSP 93
B.1.7 Defeitos Removidos - Programa 1A - PSP 94
B.1.8 Defeitos Removidos - Programa 2A - PSP 95
B.1.9 Defeitos Removidos - Programa 3A - PSP 96
B.2 Dados Coletados em 2005 - PSPm 97
B.2.1 Tempo - Programa 1A - PSPm 97
B.2.2 Tempo - Programa 2A - PSPm 98
B.2.3 Tempo - Programa 3A - PSPm 99
B.2.4 Defeitos Injetados - Programa 1A - PSPm 100
B.2.5 Defeitos Injetados - Programa 2A - PSPm 101
B.2.6 Defeitos Injetados - Programa 3A - PSPm 102
B.2.7 Defeitos Removidos - Programa 1A - PSPm 103
B.2.8 Defeitos Removidos - Programa 2A - PSPm 104
B.2.9 Defeitos Removidos - Programa 3A - PSPm 105

C Scripts de Execução **106**

C.1 Análise de Tempo - Fase Teste 106
C.1.1 Programa 2A 106
C.1.2 Programa 3A 108

C.2 Análise de Defeitos Injetados - Fase Projeto 110
C.2.1 Programa 2A 110
C.2.2 Programa 3A 111

D Empacotamento do Experimento **114**

D.1 Comunidade de Engenharia de Software 114
D.2 Organização do Experimento 115
D.3 Instrumentos 116
D.4 Roteiro de Re-experimentação 117

Referências Bibliográficas **121**

Lista de Tabelas

3.1 Script de execução do PSPm 51

4.1 Estrutura dos trabalhos 61
4.2 Variáveis resposta e respectivas hipóteses alternativas (H_1). 66
4.3 Resumo dos resultados das análises para a variável defeitos injetados com PSP e PSPm. 69
4.4 Resumo dos resultados das análises para a variável tempo com PSP e PSPm. 70
4.5 Resumo dos resultados das análises para a variável defeitos injetados com PSP e PSPm - CA $\geq$ 80. 72
4.6 Resumo dos resultados das análises para a variável tempo com PSP e PSPm - CA $\geq$ 80. 73
4.7 Resumo dos resultados das análises para a variável defeitos injetados com PSP e PSPm - CA<80. 74
4.8 Resumo dos resultados das análises para a variável tempo com PSP e PSPm - CA<80. 74

D.1 Plano de treinamento do PSP 119
D.2 Roteiro de Re-experimentação 120

Lista de Figuras

2.1 Características de qualidade de software ISO/IEC9126 (1991) 10
2.2 Níveis de Maturidade do CMM 21
2.3 MPS.BR 25
2.4 Estrutura do MPS.BR 26
2.5 Níveis de Maturidade do MPS.BR 27
2.6 Estrutura do PSP 30
2.7 Níveis de Competência do PSP 31
2.8 KPA's do CMM cobertas pelo PSP 34
2.9 Ciclo de execução do TDD 40
2.10 Os conceitos de um experimento (Wohlin, 2000) 46

3.1 *Framework* da fase Projeto do PSPm 52
3.2 Diagrama de Influência das principais variáveis do PSPm 53

4.1 Exemplo da planilha *Time Recording Log* 62
4.2 Exemplo da planilha *Defect Recording Log* 63
4.3 Exemplo da planilha *Project Plan Summary* 64
4.4 Áreas de Processo do MPS.BR cobertas pelo PSPm 77

B.1 Planilha de coleta de dados - Tempo 1A 88
B.2 Planilha de coleta de dados - Tempo 2A 89
B.3 Planilha de coleta de dados - Tempo 3A 90
B.4 Planilha de coleta de dados - Defeitos Injetados 1A 91
B.5 Planilha de coleta de dados - Defeitos Injetados 2A 92
B.6 Planilha de coleta de dados - Defeitos Injetados 3A 93

B.7 Planilha de coleta de dados - Defeitos Removidos 1A 94
B.8 Planilha de coleta de dados - Defeitos Removidos 2A 95
B.9 Planilha de coleta de dados - Defeitos Removidos 3A 96
B.10 Planilha de coleta de dados - Tempo 1A 97
B.11 Planilha de coleta de dados - Tempo 2A 98
B.12 Planilha de coleta de dados - Tempo 3A 99
B.13 Planilha de coleta de dados - Defeitos Injetados 1A 100
B.14 Planilha de coleta de dados - Defeitos Injetados 2A 101
B.15 Planilha de coleta de dados - Defeitos Injetados 3A 102
B.16 Planilha de coleta de dados - Defeitos Removidos 1A 103
B.17 Planilha de coleta de dados - Defeitos Removidos 2A 104
B.18 Planilha de coleta de dados - Defeitos Removidos 3A 105

D.1 Exemplo de planilha para coleta dos dados 118

Resumo

SILVA, Renato Afonso Cota, M.S.,Universidade Federal de Viçosa, abril de 2006. **PSP e métodos ágeis na melhoria da qualidade em produção de software: Um estudo de caso**. Orientador: José Luis Braga. Conselheiros: Alcione de Paiva Oliveira e Carlos Henrique Osório Silva.

Há vários problemas com a Engenharia de Software que é realizada atualmente, principalmente em se tratando de pequenas e médias empresas, que não possuem capital suficiente para implantar grandes processos de produção de software e menos ainda para se submeterem à avaliação de maturidade segundo padrões internacionais. Isso reforça a idéia de que é necessário investir em processos de capacitação pessoais. Parte dos problemas com a qualidade final de produtos de software derivam do fato de se confiar ou delegar aos testes finais a responsabilidade pela qualidade. Conforme comprovado na literatura, detectar e remover defeitos a partir dos testes é mais caro, não efetivo e imprevisível. A forma mais efetiva de gerenciar a qualidade do software é focar na remoção de defeitos antes de eles serem inseridos, se possível no mesmo passo de desenvolvimento em que essa inserção aconteceria. Um dos grandes problemas da qualidade do software é o não entendimento dos requisitos de usuário por parte dos programadores, os quais na codificação desenvolvem códigos que não satisfazem aos interesses dos usuários a esses requisitos. Neste trabalho descreve-se a obtenção de um processo híbrido para a produção de software de qualidade, aplicável no nível pessoal, a partir da interseção de dois processos de origens diferentes, mas que têm como foco principal o desenvolvimento de software no nível de competência individual. São eles o PSP (*Personal Software Process*), que é baseado nos Métodos Dirigidos por Planejamento, e o TDD (*Test Driven Development*) que é baseado nos Métodos

Ágeis, sendo que os dois são aplicáveis ao nível individual ou dos programadores. A combinação desses dois processos gerou um processo híbrido, chamado de PSPm. A principal característica do PSPm é a adição da técnica *test-first* oriunda do TDD na fase de projeto do PSP. Ou seja, os programadores escrevem os testes na fase de projeto e não na fase de testes, como propõe o PSP original. O PSPm foi aplicado e testado em sala de aula por alunos do curso de Ciência da Computação da Universidade Federal de Viçosa que cursavam a disciplina Engenharia de Software I. Os resultados da aplicação do PSPm foi comparado aos resultados de aplicação do PSP que havia sido aplicado em turma semelhante no ano anterior. Resultados da análise desenvolvida indicam que o uso do PSPm possibilitou a injeção de um menor número médio de erros na fase de projeto e resultou em um menor tempo médio para se testar o software. Com o intuito de validar os resultados, foi utilizada a técnica estatística de Testes por Reamostragem conhecida como *bootstrap* aplicada aos dados coletados. Ficou evidenciada uma redução na quantidade de defeitos no projeto com a utilização do PSPm.

Abstract

SILVA, Renato Afonso Cota, M.S.,Universidade Federal de Viçosa, April, 2006. **Improving the quality of the software production using PSP and agile methods: a Case Study**. Advisor: José Luis Braga. Committe members: Alcione de Paiva Oliveira and Carlos Henrique Osório Silva.

Small and medium sized software development organizations face challenges to develop quality software. Costs to adhere to international quality standards are very high, making it almost impossible to adopt and acquire maturity evaluations based on those standards. One plausible and less expensive way to increase product quality is investing in personal software development processes. Part of quality problems are a direct consequence of relying too much on tests as a quality assurance mechanism. That makes defect detection and removing an expensive, non effective and unpredictable step in software development. Previous results from the literature show that tests must be associated with error prevention procedures in early development steps, blocking error propagation to later steps where their removal is much more expensive and prone to more error insertion. Requirements elicitation, analysis and understanding seem to be a crucial point in the whole process. This is an error prone step, particularly when requirements have to be extracted from human actors that communicate with analysts and software engineers using everyday language. This work describes the design of a hybrid personal process to be used in quality software production. It is derived from a tiny intersection of PSP (Personal Software Process) that is an instance of Plan Based Methods, with TDD (Test Driven Development) that is an instance of Agile Methods and is also focused on the personal level. The hybrid process so obtained is called PSPm, where the **m** stands for **modified**. It

was obtained by introducing the *test-first* technique from TDD, in the PSP Design phase. This means programmers should design tests before coding even small slices of software, and they should use those test cases in testing software after they are coded. PSPm was tested by Computer Science students enrolled in Software Engineering I classes at Universidade Federal de Viçosa, in the years 2004 and 2005. Data gathered in 2005 were then compared to data gathered in 2004 in the same discipline. Results showed that using PSPm lead to a smaller error injection rate in PSP Design phase, thus leading to a smaller testing mean time. Data were processed and analyzed using the Resampling Tests *bootstrap* technique, applied to the data gathered in both years.

Capítulo 1

Introdução

"Se os engenheiros construíssem prédios
como os programadores escrevem programas,
um único pica-pau seria capaz de destruir a civilização".
Segunda Lei de Weinberg.

Hoje em dia o software assume um duplo papel. Ele é o produto e, ao mesmo tempo, o veículo para entrega do produto. Como produto ele disponibiliza o potencial de computação presente no computador ou, amplamente, numa rede de computadores acessível pelo hardware local. Quer resida em um telefone celular, quer opere em um computador de grande porte, o software é um transformador de informações - produzindo, gerando, adquirindo, modificando, exibindo, ou transmitindo informação, que pode ser tão simples como um único bit ou tão complexa como uma apresentação em multimídia (Pressman, 2001).

O software entrega o mais importante produto de nossa época - a informação. Portanto é truísmo dizer que os sistemas de software se tornaram ubíquos na sociedade em que vivemos. Em menos de quatro décadas eles evoluíram do suporte às atividades administrativas básicas das empresas (folha de pagamento, contabilidade, etc.) para estarem presentes em todos os instantes de nossas vidas (Pressman, 2001). Para nossa sociedade, portanto, é de suma importância devotarmos atenção à construção de software e à disciplina relacionada, que é a Engenharia de Software.

Pfleeger (2001) e Sommerville (2004) afirmam que a Engenharia de Software

é uma disciplina cujo foco é desenvolver sistemas de software de alta qualidade com eficiência relativa ao custo, para solucionar problemas cuja análise demonstrou a necessidade do uso de um sistema de software.

Apesar dos inúmeros avanços recentes nessa área, muito ainda é discutido acerca da baixa qualidade e produtividade da indústria mundial de software, refletindo-se na insatisfação de seus usuários e em prejuízos financeiros de enormes proporções. Segundo *The Standish Group*, foram gastos U$250 bilhões com o desenvolvimento de software nos Estados Unidos no ano de 2002, sendo que destes, U$38 bilhões foram perdidos em projetos mal sucedidos e U$17 bilhões foram gastos extra, acima do previsto nos projetos (Group, 2004). Ainda de acordo com a mesma estatística:

- 15% dos projetos foram cancelados;
- 43% dos projetos que foram completados apresentaram custos diferentes da previsão orçamentária;
- 66% dos projetos não atenderam às necessidades dos usuários;
- 52% dos requisitos funcionais são entregues no produto;
- 34% dos projetos são bem sucedidos.

Os fatos descritos acima são evidências da denominada "aflição crônica"[1] ou "crise do software", que correspondem ao conjunto de problemas encontrados no desenvolvimento de software de computador e não se restringe apenas ao “software que não funciona". Essa aflição inclui problemas associados com a maneira como se desenvolve software, como se dá suporte ao crescente volume de software existente e como enfrentar uma demanda crescente por mais software, sendo um reflexo da incapacidade da indústria de software em atender plenamente às exigências de qualidade de um mercado consumidor cada vez mais sofisticado em termos de aplicações.

Visando a auxiliar no saneamento desses problemas de qualidade, têm surgido ao longo do tempo processos e modelos de avaliação de maturidade de processos de

[1]Essa terminologia foi sugerida pelo professor Daniel Teichrow, da Universidade de Michigan, numa palestra apresentada em Genebra, Suíça, em abril de 1989.

software que vêm sendo experimentados no apoio ao ciclo de vida do software, com o objetivo de gerar produtos de maior qualidade.

Para o nível de maturidade organizacional, há o *Capability Maturity Model* (CMM), seu sucessor *Capability Maturity Model Integration* (CMMI) (Paulk et al., 1993; SEI, 2002) e o Modelo de Melhoria do Processo de Software Brasileiro (MPS.BR) (SOFTEX, 2005). Para o nível organizacional e de equipes, existem os métodos baseados em planejamento derivados do *Unified Process* (UP) (Jacobson et al., 1999), como o *Rational Unified Process* (RUP) Kruchten (2000) e o PRAXIS (Paula Filho, 2003). Mais recentemente, surgiram a partir da indústria de desenvolvimento de software outras propostas classificadas como métodos ágeis, tais como *eXtreme Programming* (XP) (Beck, 2000; Ambler, 2002) e SCRUM (Ambler, 2002). Já no nível pessoal, temos o *Personal Software Process* (PSP) (Humphrey, 1995, 1996, 2000) e técnicas como o *Test-Driven Development* (TDD) (Beck, 2003; Williams et al., 2003; Erdogmus, 2005), dentre inúmeras outras.

Devido aos altos custos de implantação, muitas dessas soluções não se enquadram no perfil das pequenas e médias empresas. Para ser avaliado no padrão CMMI, por exemplo, é necessário um alto investimento, tanto financeiro, quanto de pessoal. As pequenas e médias empresas de modo geral não dispõem de recursos para obtenção dessa certificação. A adoção e implantação de processos de desenvolvimento derivados do UP, como o RUP e o PRAXIS, podem auxiliar na forma organizacional do processo de desenvolvimento de software, o que pode contribuir para a melhoria da qualidade do processo interno de desenvolvimento, com impacto direto na qualidade do produto. Essas soluções, embora possam contribuir para o objetivo final da certificação organizacional, também introduzem nas empresas um aumento de custos, devido às necessidades de treinamento e adoção de plataformas de software para desenvolvimento que apóie todos os passos do processo.

Segundo Boehm and Turner (2004), os processos de software podem ser adaptados, combinando características dos Métodos Ágeis com as dos Métodos Dirigidos por Planejamento, dando origem às abordagens híbridas. A grande motivação para a proposição e adoção de abordagens híbridas é o aumento do espectro de proble-

mas resolvíveis pelos híbridos, quando comparados com os Métodos Ágeis e Métodos Dirigidos por Planejamento isoladamente.

O presente trabalho apresenta uma abordagem híbrida dirigida ao nível de maturidade pessoal, baseada na combinação do PSP (Humphrey, 1995) com técnicas originadas do TDD (Beck, 2000). O PSP é um processo de nível pessoal que tem como objetivo disciplinar os programadores de software a partir do auto-conhecimento em termos de desenvolvimento de software com qualidade, com foco na minimização dos erros cometidos durante o desenvolvimento. Essa disciplina permite melhorar a qualidade no nível pessoal, o que inclui previsibilidade de tempo e competência para estimar custos. A técnica TDD tem como objetivo ajudar o programador a entender melhor os requisitos de problemas antes de implementá-lo, por intermédio da elaboração de testes antes da codificação, procedimento denominado *Test-First*.

1.1 O problema e sua Importância

Um caminho promissor para obtenção de qualidade na produção de software é investir na melhoria do nível de qualidade pessoal de cada desenvolvedor. O PSP tem o objetivo de auxiliar nesse processo de auto-conhecimento e posterior melhoria de competência em produção de software, fornecendo mecanismos para que o desenvolvedor alcance a disciplina e a organização necessárias.

Segundo Balbino and Braga (2004), programadores inserem uma parte significativa dos defeitos nas fases de Análise e Projeto do software, o que é considerado muito negativo do ponto de vista de qualidade, pois são erros difíceis de serem descobertos em fases posteriores do desenvolvimento. O custo da remoção desses erros tende a ser baixo quando eles são descobertos nas próprias fases de Análise e Projeto em que foram inseridos. Quanto mais tarde no processo eles forem descobertos, mais cara será a sua remoção, e maior será o volume de re-trabalho, com o risco adicional de introdução de novos erros.

Uma das possíveis causas para a introdução de erros nessas fases do processo é o pouco entendimento do problema que está sendo resolvido. A análise do problema

tende a ser deficiente e conduzida em pouco tempo, e os programadores têm a tendência de querer partir logo para a fase de Codificação, deixando os erros para serem descobertos na fase de Testes. O ciclo "Codificação $\Longleftrightarrow$ Testes" tende a ser o padrão usado pelos programadores para a solução da maioria dos problemas. A conseqüência é um aumento significativo do tempo total gasto na produção do software, com uma diminuição no padrão de qualidade do produto final, que ficará mais propenso a apresentar defeitos e a não satisfazer às necessidades do usuário final.

No método XP, da Modelagem Ágil, o projeto do teste do software nas fases iniciais do processo é utilizado (Ambler, 2002) para melhorar a qualidade do produto final, diminuindo consideravelmente a quantidade de erros. Nessa perspectiva o programador desenvolve seus testes antes de desenvolver o código. Isso o força a pensar melhor no problema e na solução, com resultados positivos na qualidade do produto final (Ambler, 2002).

Seguindo essa abordagem voltada aos testes, os desenvolvedores descobrem rapidamente se suas idéias realmente funcionam, pois os testes irão validar ou não seus modelos, fornecendo retorno rápido com relação às idéias capturadas (Ambler, 2002).

1.2 Objetivos do Trabalho

O objetivo geral do trabalho é adaptar a fase de projeto do PSP com a técnica de elaboração de testes chamada de *test-first* oriunda do TDD, verificando o impacto dessa interseção no processo de desenvolvimento de software a nível pessoal. Especificamente, pretense-se:

- Adaptar o PSP com a técnica *test-first* - o qual denominamos PSPm (PSP modificado);
- Aplicar o PSPm em sala de aula;
- Comparar os resultados obtidos com os alunos que utilizaram o PSPm com os dos alunos que utilizaram o PSP;

- Verificar se houve redução dos valores médios de defeitos injetados na fase de projeto;
- Verificar se houve redução dos tempos médios gastos na fase de testes;
- Utilizar métodos estatísticos para validar os resultados.

1.3 Trabalhos Relacionados

Apesar de ter sido realizada uma intensa busca nos principais sites de busca disponíveis, tais como o scholar.google.com, google.com, periodicos.capes.gov.br, etc. não foi encontrado nenhum trabalho realizado semelhante a este, enxertando o PSP com técnicas de TDD.

Foram encontrados vários trabalhos que fazem a comparação entre as técnicas de elaboração de testes antes da codificação (*test-first*) e de elaboração de testes após a codificação (*test-last*), dos quais cita-se: Erdogmus (2005), Pancur et al. (2003) e Williams et al. (2003).

1.4 Organização deste Documento

- No Capítulo 2 são apresentados as diretrizes metodológicas que contextualizam o desenvolvimento deste trabalho.
- No Capítulo 3 é apresentado o PSPm, um processo de desenvolvimento de software a nível pessoal que une técnicas do TDD com o PSP.
- No Capítulo 4 são descritos a metodologia utilizada, as hipóteses, os resultados e a interpretação desses resultados de comparação entre o PSP e o PSPm.
- No Capítulo 5 são apresentadas as principais conclusões e contribuições observadas ao longo do trabalho, são apontadas possíveis melhorias e relatados futuros trabalhos que poderão ser realizados para aprimoramento deste.

Ao final do documento ainda são apresentados os Apêndices e as Referências Bibliográficas utilizadas, onde:

- No Apêndice A são apresentados todos os exercícios aplicados durante o desenvolvimento deste trabalho.
- No Apêndice B são apresentados todos os dados brutos resultantes da aplicação do experimento.
- No Apêndice C são apresentados os códigos utilizados pelo programa *resampling stats* para análise estatística.
- No Apêndice D é apresentado o empacotamento do experimento, com os procedimentos necessários para realização de um novo experimento.

Capítulo 2

Revisão Bibliográfica

"It is not enough to do your best: you must know what to do, and THEN do your best".
W. Edwards Deming[1]

2.1 Introdução

Historicamente, poucas organizações de software mantiveram o custo e prazo no desenvolvimento de softwares conforme haviam planejado. Essas falhas não apenas criaram uma má fama para a categoria como também causaram sérios problemas nos negócios. Há muitos casos de negócios fracassados, processos judiciais e insatisfação de clientes, sendo que em alguns casos, defeitos em software tiraram vidas humanas (Peterson, 1996).

Em nossos dias os softwares são uma força motora que impulsiona os processos de negócio em todos os segmentos da sociedade, seja em pequenas, médias ou grandes organizações. Isso faz com que sejam movimentados trilhões de dólares na construção e manutenção de softwares.

O termo Engenharia de Software ainda é um pouco obscuro, ainda não tão bem estabelecido como a Engenharia Civil ou Engenharia Elétrica. O desenvolvimento de software nos nossos dias se parece mais com Arte de Software do que com Engenharia

[1]Extraído de *Software Engineering Proverbs* - Tom Van Vleck
http://www.multicians.org/thvv/proverbs.html (03/11/2005)

de Software. Muito tem se falado e proposto, mais pouco se tem avançado no rumo de uma disciplina de engenharia.

2.2 Qualidade de Software

Pressman (2001) faz uma analogia interessante e fácil de entender a respeito de Gestão de Qualidade: "*O problema da gestão de qualidade não é o que as pessoas não sabem a respeito dela. O problema é o que as pessoas pensam que sabem... Neste sentido, a qualidade tem muito a ver com sexo. Todo mundo é a favor. (sob certas circunstâncias, certamente.) Todo mundo se considera um entendido no assunto. (mesmo que não queiram explicá-lo.) Todo mundo pensa que a execução é apenas uma questão de seguir as inclinações naturais. (apesar de tudo, conseguimos de alguma forma.) E, certamente, a maioria das pessoas acha que problemas nessas áreas são causados pelos outros. (se apenas pudessem usar o tempo para fazer as coisas direito.)*".

Alguns dos desenvolvedores de software continuam a acreditar que qualidade de software é algo com que se começa a preocupar depois que o código foi gerado. Nada poderia estar mais longe da verdade! Qualidade de software é uma atividade que deve ser aplicada ao longo do processo de software (Pressman, 2001).

A forma mais efetiva e eficiente para gerenciar a qualiade de software é focar na compreensão dos requisitos de usuário e na eliminação de defeitos o mais cedo quanto possível (Hayes and Over, 1997).

Mas o que é qualidade de software? Segundo Pressman (2001), a qualidade de software é definida como: "Conformidade com requisitos funcionais e de desempenho explicitamente documentados e características implícitas, que são esperadas em todo software desenvolvido profissionalmente".

Baseado nessa definição, resta pouca dúvida se ela poderia ser modificada ou estendida. Entretando, uma definição definitiva para qualidade de software poderia ser debatida indefinidamente. Mas, para a finalidade deste trabalho, essa definição serve para enfatizar três pontos importantes:

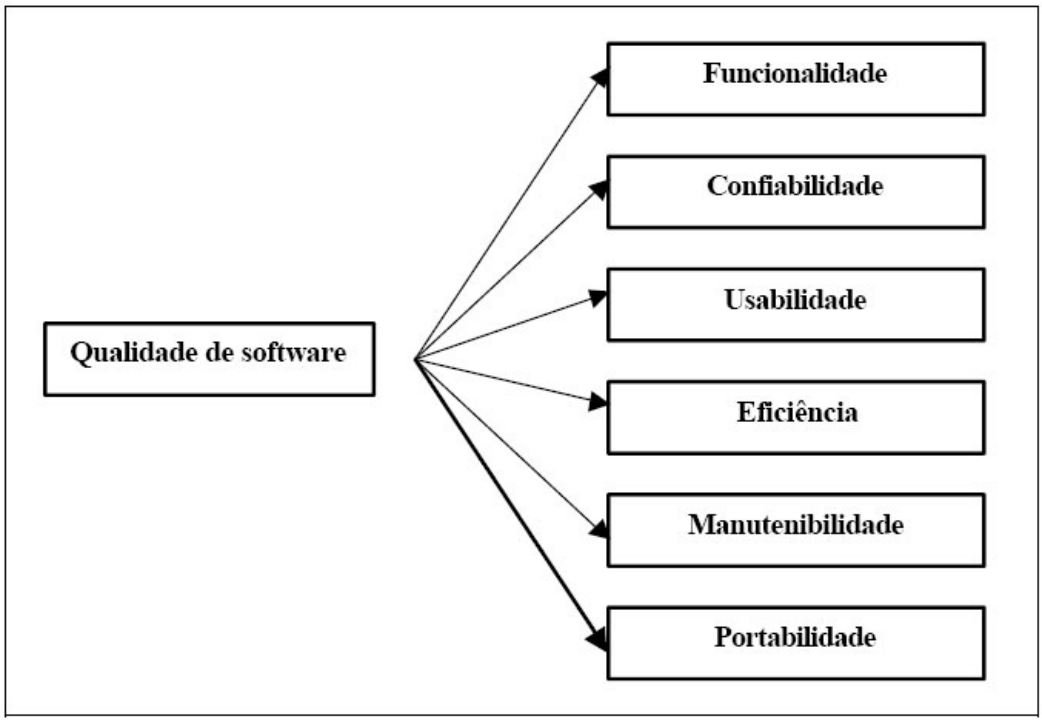
1. Os requisitos de software são a fundação a partir da qual a qualidade é medida.

A falta de conformidade com os requisitos é falta de qualidade.

2. Os padrões especificados definem um conjunto de critérios de desenvolvimento que guia o modo pelo qual o software é submetido à engenharia. Se os critérios não existem, haverá falta de qualidade.

3. Um conjunto de requisitos implícitos frequentemente não é mencionado, como por exemplo, a facilidade de uso e uma boa manutenibilidade. Se o software satisfaz seus requisitos explícitos mas deixa de satisfazer os requisitos implícitos, a qualidade do software é suspeita.

Existem na literatura várias definições para qualidade de software, mas o fundamental é que o foco de qualquer definição seja o atendimento das necessidades dos usuários. Segundo a norma ISO/IEC9126 (1991), essas necessidades podem ser sintetizadas em 6 características principais, as quais são apresentadas na Figura 2.1.



Fonte: Extraído de Balbino (2004)

Figura 2.1: Características de qualidade de software ISO/IEC9126 (1991)

Ainda de acordo com a Figura 2.1, definem-se os seis atributos-chave de qua-

lidade:

- *Funcionalidade*: Grau em que o software satisfaz as necessidades declaradas, conforme indicado pelos seguintes subatributos: adequabilidade, precisão, interoperabilidade, atendibilidade e segurança.

- *Confiabilidade*: Período de tempo em que o software está disponível para uso, conforme indicado pelos sub-atributos: maturidade, tolerância a falha e recuperabilidade.

- *Usabilidade*: Grau em que o software é fácil de usar, conforme indicado pelos seguintes sub-atributos: inteligibilidade, adestrabilidade, operabilidade.

- *Eficiência*: Grau em que o software faz uso otimizado dos recursos do sistema, conforme indicado pelos seguintes sub-atributos: comportamento em relação ao tempo, comportamento em relação aos recursos.

- *Manutenabilidade*: Facilidade na qual podem ser feitos reparos no software, conforme indicado pelos seguintes sub-atributos: analisabilidade, mutabilidade, estabilidade e testabilidade.

- *Portabilidade*: Facilidade com a qual o software pode ser transposto de um ambiente para o outro, conforme indicado pelos seguintes sub-atributos: adaptabilidade, instabilidade, comformidade e permutabilidade.

Para desenvolver um software de qualidade, atendendo às necessidades citadas, várias são as práticas necessárias, dentre as quais estão um processo de gerência bem definido, um processo de recursos humanos de qualidade e um processo de desenvolvimento de software adequado (Pressman, 2001).

Apesar de a comunidade de engenharia de software constantemente se dedicar ao desenvolvimento de técnicas mais eficazes, muitos destes métodos e técnicas, usados na indústria de software, ainda não estão plenamente efetivos. Isso se deve ao fato de esses métodos e técnicas serem aplicadas de forma errada, tendo como foco principal de qualidade o processo, ao invés de se preocupar com a qualidade a nível individual.

A qualidade de software pode ser vista sobre duas diferentes abordagens: a qualidade dos produtos de software e a qualidade dos processos de desenvolvimento e manutenção de softwre (Pressman, 2001).

A qualidade de um produto de software é resultante das atividades realizadas no processo de desenvolvimento do mesmo. Avaliar a qualidade de um produto de software é verificar, através de técnicas e atividades operacionais o quanto os requisitos são atendidos. Tais requisitos, de uma maneira geral, são a expressão das necessidades, explicitados em termos quantitativos ou qualitativos, e têm por objetivo definir as características de um software, a fim de permitir o exame de seu atendimento (Tsukumo et al., 1997).

A preocupação com a qualidade do processo de desenvolvimento de software tornou-se tão importante quanto a qualidade dos produtos entregues ao cliente. Isto se deve em função da importância estratégica que os softwares vêm assumindo na vida das organizações. Partindo do pressuposto que a qualidade dos produtos de software está diretamente relacionada com a qualidade do seu processo de desenvolvimento, muitas empresas passaram a investir na melhoria deste processo. Neste contexto, surgem modelos de maturidade, como, por exemplo, o CMM (Paulk et al., 1993; SEI, 2002) e o MPS.BR (SOFTEX, 2005), os quais serão abordados em seções posteriores. Estes modelos são usados pelas organizações como guias para examinar as práticas dos seus processos de software e continuamente melhorá-las, buscando atender os padrões especificados.

Organizações produtoras de software bem sucedidas, geralmente adotam práticas de desenvolvimento de software bem definidadas e fazem com que seus funcionários utilizam essas práticas. Com isso, os profissionais se consideram responsáveis pelo seu próprio controle de qualidade e se esforçam para melhorar sua performance, desenvolvendo a qualidade.

Um primeiro passo para a melhoria do processo de desenvolvimento de software é investir na capacitação individual. Para se desenvolver a qualidade individual é preciso encarar o desenvolvimento de software como um trabalho de engenharia, o que implica em manter a disciplina em todas as fases do desenvolvimento de software.

No entanto, a forma como os desenvolvedores de software trabalham atualmente mais se parece com arte que com engenharia. Cada um tem seus próprios métodos e técnicas de desenvolvimento (Humphrey, 2000).

2.2.1 Tratamento de Defeitos

Um *defeito* ocorre quando uma pessoa comete um engano, chamado de *erro*, na realização de alguma atividade relacionada a um software. Por exemplo, um projetista pode não compreender corretamente um requisito e criar um projeto que não condiz com a verdadeira intenção do analista de requisitos e do usuário. Esse defeito do projeto é uma codificação do erro e pode levar a outros defeitos, tais como código incorreto e descrição incorreta do manual do usuário. Portanto, um simples erro pode gerar muitos defeitos, e um defeito pode estar presente em qualquer produto do desenvolvimento ou da manutenção do software (Pfleeger, 2001).

Em geral, metas de melhorias na área de qualidade se iniciam com algum tipo de medição de defeitos. Tanto defeitos no produto quanto no processo são registrados para identificar as áreas no processo de desenvolvimento que possuem maiores necessidades de melhorias (Solingen and Berghout, 1999).

Segundo Paula Filho (2003), o tempo de desenvolvimento do software é, em geral, reduzido com o aumento da qualidade no processo de desenvolvimento de software, pois, se o trabalho é desenvolvido com qualidade, o processo de prevenção, detecção e remoção de defeitos é mais eficiente. Levando-se em conta que todo defeito inserido tem um custo de remoção e que quanto mais tarde o defeito é corrigido mais cara é sua correção, o aumento da qualidade traz benefícios diretos na questão do cumprimento de custos e prazos, que atualmente é um dos maiores problemas na área de desenvolvimento de software.

Com o aumento da complexidade e tamanho das aplicações, o potencial de dano de algum defeito cresce na mesma proporção. Com isso, a indústria de software utilizou como estratégia inicial o aumento do rigor e tempo com os testes. Porém, comprovadamente esta não é uma estratégia totalmente efetiva (Humphrey, 1995).

Encontrar problemas de requisitos utilizando testes é caro. A identificação e

remoção desses defeitos antes da codificação, diminuem de forma significativa o custo da remoção. O ideal é que defeitos não sejam injetados, porém caso sejam, o objetivo é removê-los o mais cedo possível. Neste sentido, revisões e inspeções minimizam o número de defeitos no produto em todos os estágios, reduzindo também a quantidade e o custo do re-trabalho. É provável que se reduza também o custo para encontrar um defeito (Humphrey, 1995).

Enfim, está claro que os custos de identificação são mais altos durante o teste e uso. Desta forma qualquer estratégia que deseje reduzir os custos de desenvolvimento ou tempo deveria focar na prevenção ou remoção de erros antes dos testes (Humphrey, 1995).

2.3 Competência Organizacional

Após várias décadas de promessas não cumpridas sobre ganhos de produtividade e qualidade na aplicação de novas metodologias e tecnologias, organizações da indústria e do governo estão concluindo que o problema fundamental é a falta de habilidade em gerenciar processos de software.

Mesmo com a utilização dos melhores métodos, técnicas e ferramentas, os benefícios proporcionados por eles não podem ser alcançados em ambientes caóticos e indisciplinados. Entretanto, algumas organizações sem disciplina alcançam excelentes resultados em alguns projetos de software. Porém, o sucesso obtido nesses projetos, é em geral resultado de esforços heróicos de uma equipe dedicada.

Na falta de um processo de software definido e aplicável para a organização, a repetição dos resultados depende da disponibilidade de ter as mesmas pessoas para o próximo projeto e ainda "contar com a sorte". O sucesso, que depende unicamente da equipe, não fornece condições para a melhoria da produtividade e da qualidade da organização por um longo período. Melhorias contínuas ocorrem apenas por intermédio de esforços focados no processo de desenvolvimento de software e sustentados por práticas de gestão.

É interessante notar algumas diferenças existentes entre as organizações madu-

ras e as imaturas na produção de software. Nas organizações imaturas, os processos de software são improvisados por pessoas experientes, em conjunto com seus gerentes, durante o desenrolar do projeto. Mesmo que o processo tenha sido especificado, ele não é seguido de forma rigorosa ou não é obrigatório. Essas organizações são reacionárias, onde os gerentes estão focados na solução de problemas imediatos, ação mais conhecida como "apagar incêndios". Os cronogramas e orçamentos são rotineiramente excedidos por razões de estimativas irreais. Quando são impostos prazos críticos, a qualidade e funcionalidade do produto são reduzidas na tentativa de manter o cronograma dentro do prazo.

Por outro lado, as organizações de software maduras possuem uma habilidade organizacional ampla para gerenciar o desenvolvimento e manutenção de software. O processo de software é cuidadosamente explicitado à equipe já existente e aos novos funcionários, sendo que as atividades são realizadas de acordo com processos definidos. Esses processos são atualizados sempre que necessário e as melhorias são implementadas através de testes-piloto e/ou análise de custo-benefício. As regras e as responsabilidades no processo definido são claras em toda a parte do projeto e da organização. Os gerentes monitoram a qualidade dos produtos de software e a satisfação dos clientes. Cronogramas e orçamentos são baseados em históricos de desempenho e são reais; os resultados esperados para custo, cronograma, funcionalidade e qualidade do produto são quase sempre alcançados. Em geral, um processo disciplinado é seguido de forma consistente porque todos os participantes compreendem a importância disso. Além disso, existe infra-estrutura necessária para dar suporte ao processo.

Para tirar proveito dessas observações sobre as organizações de software maduras e imaturas, é necessário a construção de uma estrutura de maturidade de processo de software. Essa estrutura descreve um caminho evolutivo, desde os processos caóticos, *ad hoc*, até os processos maduros, disciplinados. Sem essa estrutura, os programas de melhoria podem se mostrar ineficientes, porque não são estabelecidos os fundamentos necessários para dar suporte às sucessivas melhorias.

A maturidade do processo de software é a extensão para a qual um processo específico é explicitamente definido, gerenciado, medido, controlado e efetivado. A

maturidade representa o potencial de crescimento da capacidade e indica a riqueza do processo de software da organização e a consistência com que o mesmo é aplicado em todos os seus projetos. Em uma organização madura, o processo de software é bem compreendido - o que geralmente é feito por meio de documentação e treinamento - e está sendo continuamente monitorado e melhorado pelos seus usuários. A capacidade de um processo de software maduro é conhecida. A maturidade de processo de software implica que a produtividade e a qualidade resultantes do processo possam ser continuamente melhoradas através de ganhos consistentes na disciplina alcançada com a sua utilização.

Quando uma organização obtém ganhos na maturidade de um processo de software, ela o institucionaliza por meio de políticas, padrões e estruturas organizacionais. A institucionalização exige a construção de uma infra-estrutura e de uma cultura corporativa que possa dar suporte aos métodos, práticas e procedimentos de negócio que perdurem após possíveis afastamentos daqueles que originalmente os definiram.

2.3.1 A Norma NBR ISO/IEC 12207

A Norma NBR ISO/IEC 12207 - Tecnologia da Informação - Processos de Ciclo de Vida de Software (ISO/IEC12207, 2004) estabelece uma estrutura comum para os processos de ciclo de vida de software, com terminologia bem definida, que é tomada como referência pela indústria de software. A estrutura da norma contém processos, atividades e tarefas que devem ser aplicados na aquisição, fornecimento, desenvolvimento, operação e manutenção de produtos de software. Esse conjunto de processos, atividades e tarefas foi projetado para ser adaptado de acordo com as características de cada projeto de software, o que pode envolver o detalhamento, a adição e a supressão de processos, atividades e tarefas não aplicáveis ao mesmo.

Em outubro de 2002 e de 2004, a ISO/IEC 12207 recebeu duas emendas (emendas 1 e 2, respectivamente) para tratar a evolução da Engenharia de Software e, também, para harmonizá-la com a norma ISO/IEC15504 (2004). Basicamente, essas melhorias criaram novos processos ou expandiram o escopo de alguns processos, além de serem adicionados, a cada processo, o seu propósito e resultados. Agora, cada

processo tem um propósito e um resultado associados, além de atividades e tarefas.

Os processos da ISO/IEC 12207 (ISO/IEC12207, 2004) são agrupados em três categorias, a saber: Processos Fundamentais, Processos de Apoio e Processos Organizacionais. Além desses, há um processo de adaptação, que, como o nome indica, trata da adaptação da norma à cultura e aos aspectos específicos de uma organização.

Os Processos Fundamentais constituem um conjunto de cinco processos que atendem aos adquirentes, fornecedores, desenvolvedores, operadores e mantenedores do software, durante o seu ciclo de vida. São eles os processos de Aquisição, Fornecimento, Desenvolvimento, Operação e Manutenção.

Um Processo de Apoio auxilia um outro processo, como parte integrante do mesmo, com um propósito distinto, contribuindo para o sucesso e qualidade do projeto de software. Em outras palavras, é um processo empregado ou executado por outro processo, quando necessário. São processos de apoio: Documentação, Gerência de Configuração, Garantia da Qualidade, Verificação, Validação, Revisão Conjunta, Auditoria, Resolução de Problemas, Usabilidade e Avaliação do Produto.

Os Processos Organizacionais são empregados por uma organização para melhorar continuamente a sua estrutura e os seus processos. Eles são tipicamente empregados fora do domínio de projetos e contratos específicos. Entretanto, os ensinamentos desses projetos e contratos contribuem para a melhoria da organização. São processos organizacionais os processos de Gerência, Infra-estrutura, Melhoria, Recursos Humanos, Gerência de Ativos, Gerência de Programa de Reuso e Engenharia de Domínio.

2.3.2 A Norma ISO/IEC 15504

Desenvolvida pela comunidade internacional em um projeto denominado SPICE (*Software Process Improvement and Capability dEtermination*), a Norma ISO/IEC 15504 - *Information Technology - Process Assessment* (ISO/IEC15504, 2004) (Tecnologia da Informação - Avaliação de Processos) é um padrão internacional ISO para avaliação de processos de software (ISO/IEC15504, 2004).

A ISO/IEC 15504 provê uma abordagem estruturada para avaliação de processos de software com os seguintes objetivos:

i) permitir o entendimento, por ou em favor de uma organização, do estado dos seus processos, visando estabelecer melhorias;

ii) determinar a adequação dos processos de uma organização para atender a um requisito particular ou classe de requisitos;

iii) determinar a adequação de processos da organização para um contrato ou classe de contratos.

Essa norma pode ser usada tanto por adquirentes de software para determinar a capacidade dos processos de software de seus fornecedores, quanto por fornecedores para determinar a capacidade de seus próprios processos ou para identificar oportunidades de melhoria.

É composta de cinco partes:

1. Conceitos e Vocabulário (*Concepts and Vocabulary*): provê uma introdução geral dos conceitos de avaliação de processos e um glossário de termos relacionados;

2. Realizando uma avaliação (*Performing an Assessment*): estabelece os requisitos mínimos para se realizar uma avaliação. Essa parte é a única que tem caráter normativo;

3. Orientações para se Realizar uma Avaliação (*Guidance on Performing an Assessment*): provê orientações para se interpretar os requisitos para realização de uma avaliação;

4. Orientações para Uso em Melhoria de Processo e Determinação da Capacidade de Processo (*Guidance on use for Process Improvement and Process Capability Determination*): fornece orientações para utilização dos resultados de uma avaliação nos contextos melhoria de processo e determinação da capacidade de processo;

5. Um Exemplo de Modelo de Avaliação de Processo (*An exemplar Process Assessment Model*): contém um exemplo de modelo de avaliação de processo baseado no modelo de referência da ISO/IEC 12207.

2.3.3 CMM/CMMI

O *Capability Maturity Model* - CMM foi criado pelo *Software Engineering Institute* - SEI em conjunto com o Departamento de Defesa dos Estados Unidos (*Department of Defense* - DoD) (Paulk et al., 1993; SEI, 2002) e tem sido considerado o modelo mais consolidado e aplicável de qualidade de software (Pressman, 2001). O DoD, após observar que a situação dos seus contratos de desenvolvimento de software tornara-se insustentável, patrocinou a criação do SEI, em 1984, visando a proporcionar condições para a evolução das boas práticas de Engenharia de Software. O objetivo era que fosse alcançado, nos projetos de desenvolvimento de software dos fornecedores do DoD, o mesmo nível de repetibilidade e controle encontrado em outros setores da atividade industrial, tais como a manufatura e a construção civil.

O CMM fornece às organizações de software um guia de como obter controle em seus processos para desenvolver e manter software e como evoluir em direção a uma cultura de engenharia de software com excelência de gestão. O CMM foi projetado para guiar as organizações de software no processo de seleção das estratégias de melhoria, determinando a maturidade atual do processo e identificando as questões mais críticas para a qualidade e melhoria do processo de software.

A estrutura em estágios do CMM é baseada em princípios de qualidade de produto surgidos nos últimos sessenta anos. Nos anos 30, Walter Shewhart promulgou os princípios de controle estatístico da qualidade. Seus princípios foram desenvolvidos e demonstrados com sucesso no trabalho de Deming (1986) e Juran and Gryna (1988). Esses princípios foram adaptados pelo SEI dentro da estrutura de maturidade que estabelece a gestão de projeto e os fundamentos de engenharia para o controle quantitativo do processo de software, que é a base para a contínua melhoria do processo.

O modelo é construído a partir do conceito de processo, o qual é integrado por pessoas, ferramentas e métodos para executar uma seqüência de passos com o objetivo definido de transformar determinadas entradas em determinadas saídas. Uma premissa implícita no modelo é que a qualidade de um produto é determinada em grande medida pela qualidade dos processos utilizados na sua produção e manutenção. Na medida em que a maturidade dos processos de uma empresa evolui, estes passam

a ser mais bem definidos e institucionalizados nas organizações e com maior equilíbrio entre os seus três componentes (pessoas, ferramentas e métodos).

A melhoria contínua de processo é baseada em muitas etapas evolutivas pequenas ao invés de fundamentar-se em inovações revolucionárias. O CMM fornece uma estrutura para organizar essas etapas evolutivas em cinco níveis crescentes de maturidade que estabelecem fundamentos sucessivos para a contínua melhoria do processo. Cada nível define o grau de maturidade dos processos de uma organização e é composto por áreas-chave do processo. Cada área-chave permite alcançar um conjunto de metas ou objetivos.

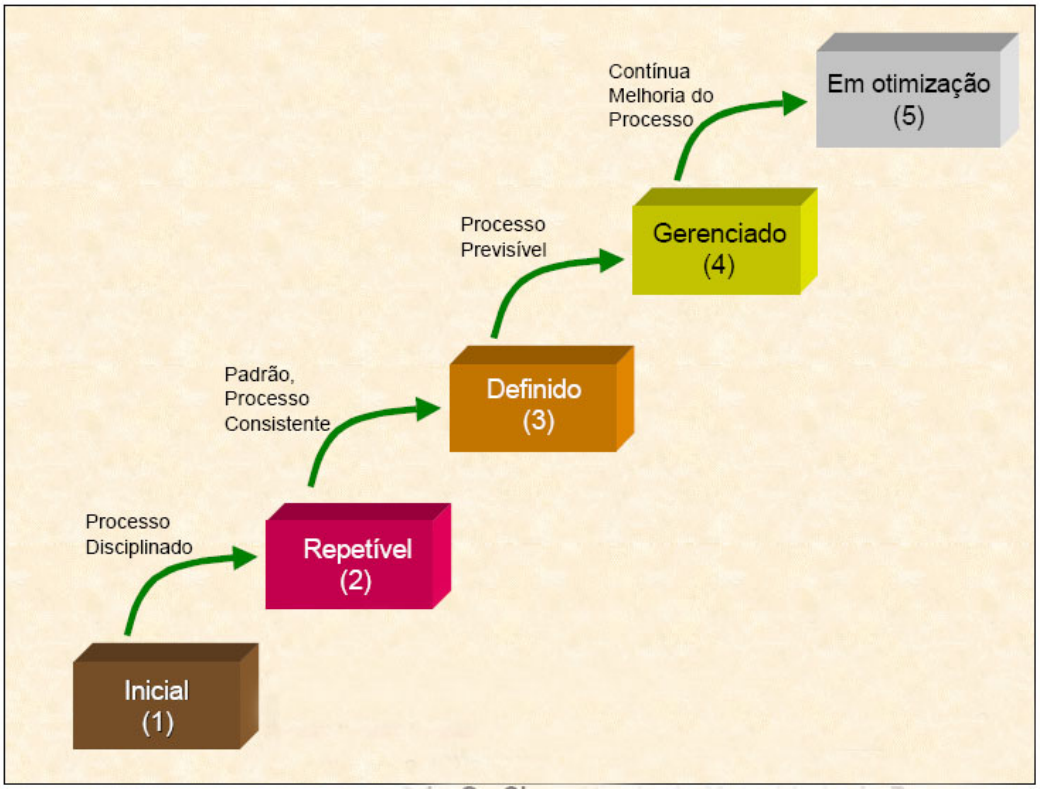
Nível de maturidade é um estágio evolutivo bem definido em busca de um processo de software maduro. Cada nível compreende um conjunto de objetivos de processos que, quando satisfeitos, estabilizam um componente importante do processo de software. Alcançando cada nível da estrutura de maturidade, estabelecem-se diferentes componentes no processo de software, resultando em um crescimento na capacidade de processo da organização, conforme mostra a Figura 2.2.

Baseados nesta estrutura geral foram criados modelos específicos para atender diferentes áreas e aspectos de uma organização, tais como a área de recursos humanos (*People* CMM), de sistemas em geral (SE-CMM), de software (SW-CMM), entre outras. Além disto, existem ainda modelos para conduzir avaliações (SCE, SCAMPI) e planejar a melhoria dos processos (IDEAL). O SW-CMM abrange práticas de planejamento, engenharia e gerência para o desenvolvimento e manutenção de software, auxiliando uma organização a transformar seus processos de software *ad hoc* e imaturos em processos maduros e disciplinados.

De acordo com a Figura 2.2, o SW-CMM prevê os seguintes níveis:

1. **Inicial:** O processo de software é caracterizado como "*ad hoc*" e até mesmo ocasionalmente caótico. Poucos processos são definidos e o sucesso depende de esforço individual.

2. **Repetível:** Os processos básicos de gestão de projeto são estabelecidos para acompanhar custo, cronograma e funcionalidade. A necessária disciplina do



Fonte: Adaptado de Paulk et al. (1993)

Figura 2.2: Níveis de Maturidade do CMM

processo existe para repetir sucessos anteriores em projetos com aplicações similares.

3. **Definido:** O processo de software para as atividades de gestão e engenharia é documentado, padronizado e integrado em um processo de software padrão para a organização. Todos os projetos utilizam uma versão aprovada do processo de software padrão para desenvolver e manter software.

4. **Gerenciado:** Medidas detalhadas do processo de software e da qualidade do produto são realizadas. O processo e os produtos de software são quantitativamente compreendidos e controlados.

5. **Em Otimização:** A melhoria contínua do processo é propiciada pelo *feedback* quantitativo do processo e pelas idéias e tecnologias inovadoras.

No CMM as áreas-chave do processo as chamadas KPA's (*Key Proccess Area*),

descrevem os atributos essenciais que seriam esperados para caracterizar uma organização em um nível particular de maturidade. A intenção é que o CMM tenha um nível suficiente de abstração que não restrinja desnecessariamente a maneira como o processo de software é implementado pela organização. Ele simplesmente descreve o que normalmente seria esperado dos atributos essenciais do processo de software.

As KPA's descrevem as funções de engenharia de software que devem estar presentes para satisfazer uma boa prática em um nível específico. Cada KPA é descrita identificando-se as seguintes características (Pressman, 2001):

- *Metas*: os objetivos gerais que a KPA deve atingir;
- *Empenhos*: requisitos que devem ser satisfeitos para atingir as metas ou fornecer prova de intenção de alcançar as metas;
- *Capacidades*: procedimentos que devem ser estabelecidos para permitir à organização cumprir com os compromissos ou empenhos;
- *Atividades*: tarefas específicas necessárias para obter a função da KPA;
- *Métodos para monitorar a implementação*: maneira pela qual a prática adequada da KPA pode ser verificada.

Dezoito KPA's são definidas para todo o modelo de maturidade e é feita sua correspondência com os diferentes níveis de maturidade do processo. As seguintes KPA's devem ser conseguidas em cada nível:

Nível 2

- Gestão de configuração de software;
- Garantia de qualidade de software;
- Gestão de subcontratos de software;
- Acompanhamento e supervisão de projeto de software;
- Planejamento de projeto de software;

- Gestão de requisitos.

Nível 3

- Revisões pelos pares;
- Coordenação intergrupal;
- Engenharia de produto de software;
- Gestão integrada de software;
- Programa de treinamento;
- Definição do processo da organização;
- Enfoque do processo da organização.

Nível 4

- Gestão de qualidade de software;
- Gestão quantitativa do processo.

Nível 5

- Gestão de modificação de processo;
- Gestão de mudança de tecnologia;
- Prevenção de defeitos.

O CMM não diz à organização como melhorar, simplesmente descreve a organização em cada nível de maturidade sem especificar os meios para conseguí-lo. Pode levar vários anos para se passar do Nível 1 para o Nível 2, e assim sucessivamente.

A melhoria de processo de software ocorre dentro do contexto dos planos estratégicos e dos objetivos de negócio da organização, da sua estrutura organizacional, das tecnologias em uso, da sua cultura social e sistema de gestão. O CMM está voltado para os aspectos de processo da Gestão da Qualidade Total. A melhoria de processo

bem sucedida implica que os aspectos fora do escopo de processo de software também sejam encaminhados, como por exemplo as questões pessoais envolvidas nas mudanças da cultura organizacional que possibilitem a implementação e a institucionalização das melhorias de processo.

Recentemente surgiu o *Capability Maturity Model Integration* - CMMI, que é considerado a nova versão do modelo CMM e tem como objetivo fornecer um guia para melhorar os processos e a habilidade de gerenciar o desenvolvimento, aquisição e manutenção de produtos e serviços da organização. O modelo coloca as melhores práticas dentro de uma estrutura que ajuda na avaliação da maturidade organizacional ou da capacidade da área do processo, estabelecendo prioridades para a melhoria e implementando estas melhorias (SEI, 2002). O projeto também se preocupou em tornar o CMM compatível com a norma ISO 15504, de modo que avaliações em um modelo pudessem ser reconhecidas como equivalentes às do outro. Vale destacar que, apesar de prover um novo modelo, o CMMI procura preservar ao máximo os investimentos feitos em melhoria de processos baseadas no CMM.

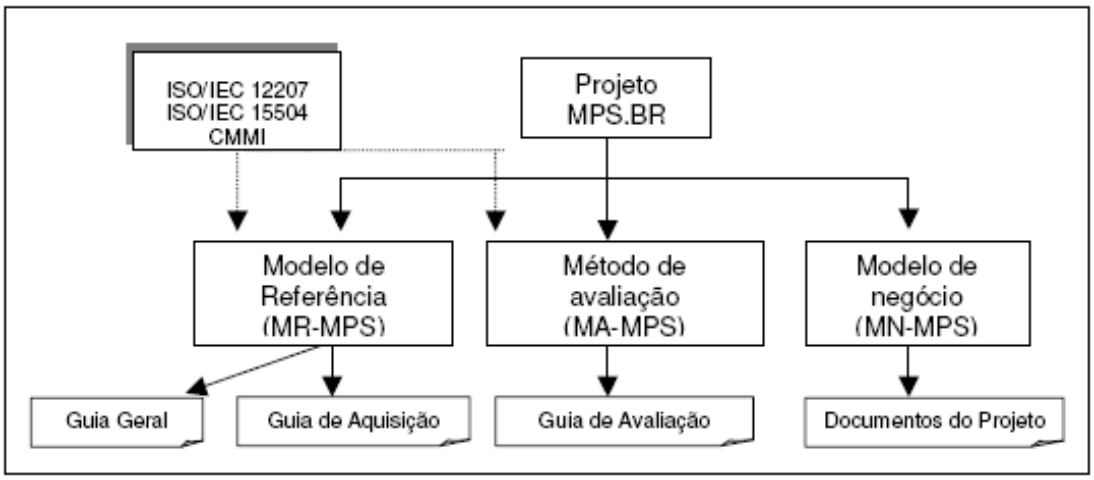
O CMMI oferece duas abordagens diferentes para a melhoria de processos, conhecidas como o *modelo contínuo* e o *modelo em estágios*. A representação em estágio, já presente no CMM, visa a uma abordagem relativa à maturidade da organização. Já a representação contínua mapeia níveis de capacitação de cada área de processo, procurando se alinhar com a norma ISO/IEC 15504. Ambas as representações contém Áreas de Processos (antigas Áreas-Chave de Processo do CMM) comuns. Porém, na representação em estágios, as Áreas de Processos são agrupadas em níveis de maturidade, enquanto na representação contínua as mesmas Áreas de Processo estão divididas em categorias.

2.3.4 MPS.BR

O MPS.BR (SOFTEX, 2005) tem como objetivo definir um modelo de melhoria e avaliação de processo de software, preferencialmente para as micro, pequenas e médias empresas, de forma a atender as suas necessidades de negócio e a ser reconhecido nacional e internacionalmente como um modelo aplicável à indústria de software. Este

é o motivo pelo qual ele está aderente a modelos e normas internacionais. O MPS.BR também define regras para sua implementação e avaliação, dando sustentação e garantia de que o MPS.BR está sendo empregado de forma coerente com as suas definições.

A base técnica utilizada para a construção do MPS.BR é composta pelas normas NBR ISO/IEC 12207 (Seção 2.3.1) - Processo de Ciclo de Vida de Software e suas emendas 1 e 2 e a ISO/IEC 15504 (Seção 2.3.2)- Avaliação de Processo (também conhecida por SPICE: *Software Process Improvement and Capability dEtermination*) e seu Modelo de Avaliação de Processo de Software ISO/IEC 15504-5, portanto o modelo é aderente a essas normas. O MPS.BR também cobre o conteúdo do CMMI, através da inclusão de processos e resultados de processos em relação aos processos da Norma NBR ISO/IEC 12207, conforme Figura 2.3.



Fonte: SOFTEX (2005)

Figura 2.3: MPS.BR

O MPS.BR está dividido em três (3) componentes: Modelo de Referência (MR-MPS), Método de Avaliação (MA-MPS) e Modelo de Negócio (MN-MPS). Cada componente do modelo foi descrito através de Guias e dos Documentos do Projeto.

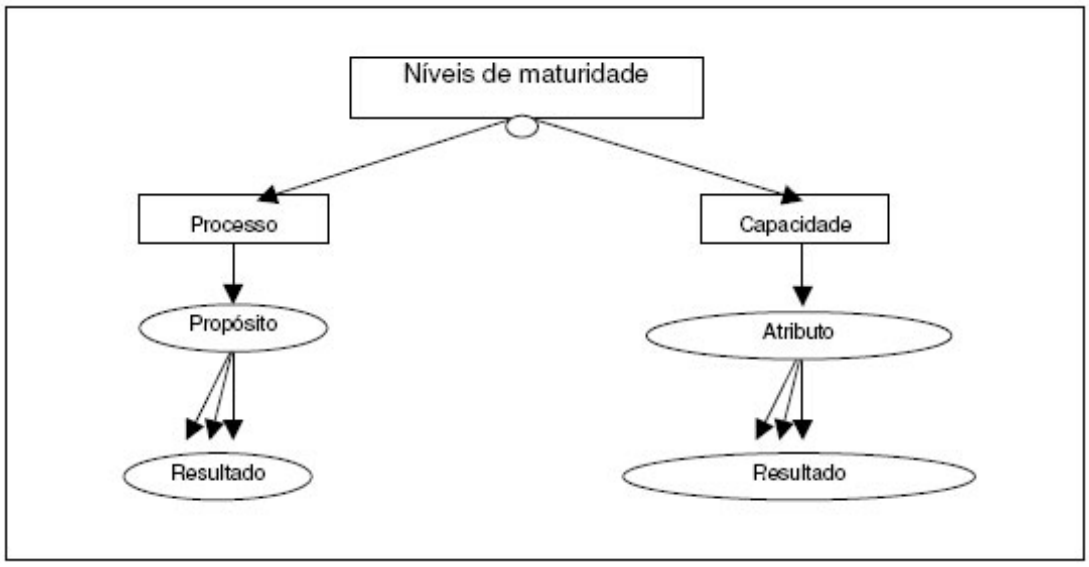
O Modelo de Referência de Melhoria de Processo de Software (MR-MPS) contém os requisitos que as organizações deverão atender para estar em conformidade com o MR-MPS. Ele contém as definições dos níveis de maturidade, da capacidade de processos e dos processos em si. Ele foi baseado nas normas NBR ISO/IEC 12207

e suas emendas 1 e 2, ISO/IEC 15504 e adequado ao CMMI.

Adicionalmente, foi escrito o Guia de Aquisição, que é um documento complementar para organizações que pretendam adquirir software e serviços correlatos. O Guia de Aquisição não contém requisitos do MR-MPS, mas boas práticas de aquisição de software e serviços correlatos.

O Método de Avaliação (MA-MPS) contém o processo de avaliação, os requisitos para os avaliadores e os requisitos para averiguação da conformidade ao modelo MR-MPS. Ele está descrito de forma detalhada no Guia de Avaliação e foi baseado na norma ISO/IEC 15504.

O Modelo de Referência MR-MPS define níveis de maturidade que são uma combinação entre processos e capacidade de processos, conforme a estrutura apresentada na Figura 2.4.

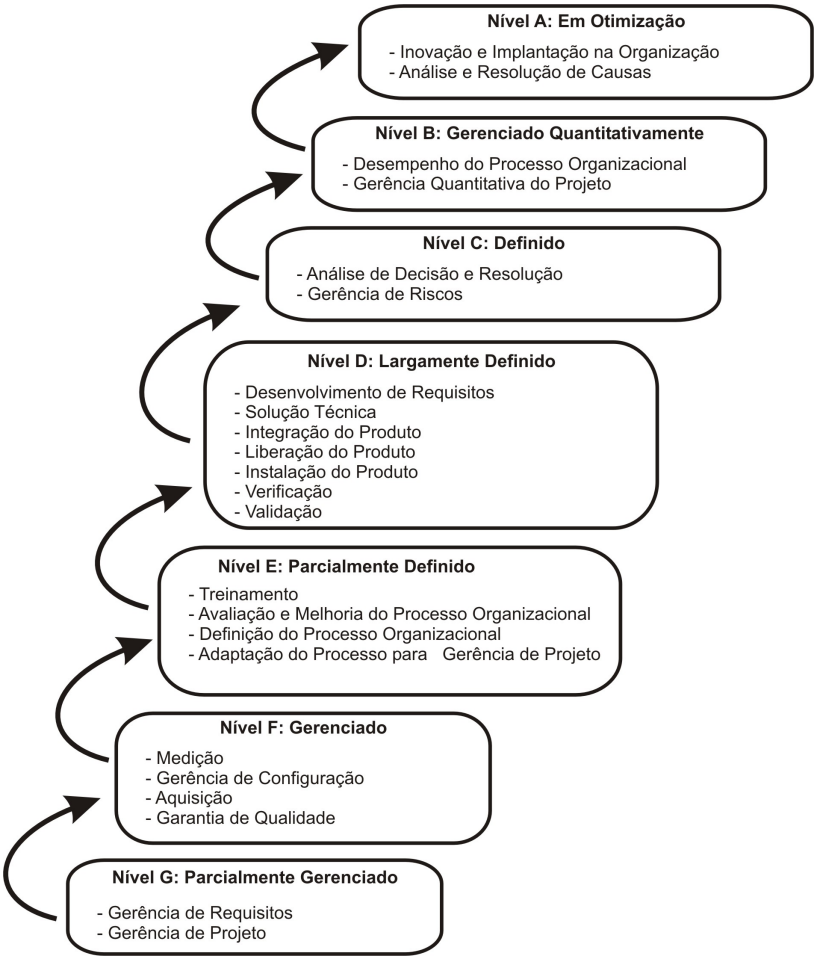


Fonte: SOFTEX (2005)

Figura 2.4: Estrutura do MPS.BR

Os níveis de maturidade estabelecem patamares de evolução de processos, caracterizando estágios de melhoria de implementação de processos na organização. O nível de maturidade em que se encontra uma organização permite prever seu desempenho futuro em uma ou mais disciplinas. O MR-MPS define sete níveis de maturidade: A (Em Otimização), B (Gerenciado Quantitativamente), C (Definido), D (Largamente

Definido), E (Parcialmente Definido), F (Gerenciado) e G (Parcialmente Gerenciado). A escala de maturidade se inicia no nível G e progride até o nível A. Para cada um destes sete níveis de maturidade foi atribuído um perfil de processos e de capacidade de processos que indicam onde a organização tem que colocar esforço para melhoria de forma a atender os objetivos de negócio, conforme Figura 2.5.



Fonte: Elaborado pelo Autor

Figura 2.5: Níveis de Maturidade do MPS.BR

O progresso e o atingimento do nível de maturidade se obtêm quando são atendidos todos os resultados e propósito do processo, e os atributos de processo relacionados àquele nível e aos anteriores.

A divisão em estágios, embora baseada nos níveis de maturidade do CMMI, tem uma graduação diferente, com o objetivo de possibilitar uma implementação e

avaliação mais gradual e adequada às pequenas e médias empresas. A possibilidade de se realizar avaliações considerando mais níveis permite uma visibilidade dos resultados de melhoria de processos com prazos mais curtos.

A capacidade do processo é um conjunto de atributos de processo descrito em termos de resultados os quais proporcionam o atendimento dos atributos de processo. A capacidade estabelece o grau de refinamento e institucionalização com que o processo é executado na organização. À medida que evolui nos níveis, um maior ganho de capacidade para desempenhar o processo é atingido pela organização.

O atendimento dos atributos do processo e dos resultados dos atributos do processo é requerido para todos os processos correspondentes ao nível de maturidade, embora eles não sejam detalhados dentro de cada processo. A sua execução é cumulativa, ou seja, se a organização está no nível F, tem que atender ao nível de capacidade do nível G e do nível F para todos os processos relacionados no nível de maturidade, e assim sucessivamente.

2.4 Personal Software Process (PSP)

O PSP (Humphrey, 1995) é um processo pessoal de melhoria projetado para ajudar os desenvolvedores a controlar, administrar e aperfeiçoar sua competência para produzir software de qualidade. O PSP é uma estrutura organizada de formulários, diretrizes e procedimentos para desenvolvimento de software. Corretamente usado, o PSP provê os dados históricos que o desenvolvedor de software precisa para fazer e conhecer melhor seus compromissos e torna os elementos rotineiros de seu trabalho mais previsíveis e mais eficientes.

A filosofia por trás do PSP é que a competência de uma organização para construir softwares de determinado tamanho e grau de complexidade decorre, em parte, da habilidade individual de seus engenheiros de software para desenvolver, com alta qualidade, pequenos programas de maneira disciplinada e efetiva. O PSP se baseia no princípio do conhecimento, avaliação e melhorias contínuas do processo individual de desenvolvimento de software, com foco no perfil de erros cometidos individualmente

com mais frequência e em sua minimização (Humphrey, 1995).

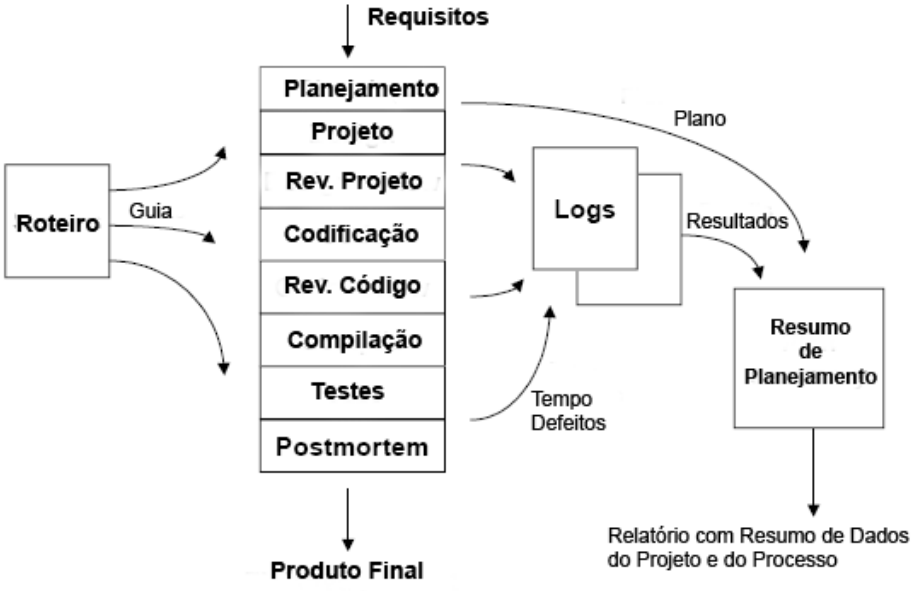
O propósito do PSP é ajudar o desenvolvedor a melhorar sua engenharia de software a partir do instante em que este disciplina sua forma de trabalho e passa a conhecê-la melhor, entendendo sua própria performance e sabendo onde e como melhorá-la (Humphrey, 1995). Mais especificamente as metas do PSP são (Balbino, 2004):

- Como estimar precisamente, planejar, traçar e re-planejar o tempo necessário para o esforço individual no desenvolvimento de software;
- Como trabalhar de acordo com um processo bem definido;
- Como definir e refinar o processo;
- Como usar revisões efetivamente e eficientemente para desenvolver a qualidade do software (pela remoção imediata dos defeitos);
- Como evitar erros;
- Como analisar os dados medidos para desenvolver as estimativas e a remoção e prevenção de erros;
- Como identificar e obstruir as deficiências do processo.

Dados são coletados em formulários específicos em cada um dos níveis do PSP, constituindo os *logs* de registro que vão formar uma base pessoal de dados sobre os programas e sistemas desenvolvidos até determinado instante. Esses dados são sobre erros cometidos tanto em termos absolutos quanto em termos relativos, tempo gasto em cada fase do processo, registro de tamanho baseado em número de linhas de código produzidas, alteradas, incluidas ou reutilizadas no software em desenvolvimento, estimativas de tempo e tamanho, etc. Medidas relativas, tais como número de erros por *Kloc* (mil linhas de código), tempo gasto por *Kloc* e outras são obtidas e atualizadas ao longo da aplicação do processo. O PSP ainda propõe que o desenvolvedor estabeleça um padrão de codificação para organizar visualmente o texto dos programas, e

com isso favorecer a sua manutenibilidade e transferência para outros programadores. Associado a esse padrão, há um conjunto de regras para contagem de linhas de código estabelecidas pelo próprio desenvolvedor, que vão permitir a aquisição de dados sobre volume de código em *Kloc*.

O PSP fornece técnicas estatísticas para análise desses dados e geração de curvas e gráficos, que registram o perfil de cada desenvolvedor. Com o passar do tempo, o desenvolvedor adquire competência para fazer boas estimativas de tempo de desenvolvimento e tamanho para novos projetos, fazer o planejamento para o desenvolvimento de software e avaliar custos.



Fonte: Adaptado de Humphrey (1995)

Figura 2.6: Estrutura do PSP

Conforme mostra a Figura 2.6, o processo inicial associado ao PSP consiste de roteiros, fases, *logs* e relatórios. O principal item de entrada no PSP são os requisitos de usuário e o principal item de saída é o software. Os roteiros servem de guias para a execução do processo, indicando quais atividades devem ser realizadas e em que ordem, e quais os insumos de entrada e saída para cada uma das fases do processo. Os *logs* servem para registrar os dados referentes a tempo e a defeitos injetados e removidos em cada fase do PSP. Após a conclusão da fase Testes, o desenvolvedor passa

para a fase PostMortem, na qual é preenchido o formulário de Resumo do Projeto (*Project Plan Summary*), onde são utilizados os dados provenientes dos *logs* de tempo e defeitos injetados e removidos. Na fase PostMortem o desenvolvedor tem disponíveis o histórico de execução do projeto e ainda a base acumulada, constituindo uma linha de base (*baseline*) pessoal que será utilizada nos sucessivos níveis de competência.

2.4.1 Fases do PSP

O PSP é composto por quatro níveis de competência verticais, sendo que os três primeiros são compostos por dois níveis horizontais cada um. São ao todo sete níveis de competência, que são atingidos em sequência. Cada nível adiciona novas competências aos níveis já implantados. A aplicação do processo dessa forma minimiza o impacto da mudança no desenvolvedor, que gradativamente vai adquirindo a nova cultura, conforme mostra a Figura 2.7.

Fonte: Adaptado de Humphrey (1995)

Figura 2.7: Níveis de Competência do PSP

De forma mais detalhada, a Figura 2.7 descreve:

- PSP0 - O processo base: o primeiro estágio do PSP é estabelecer uma base (*baseline*), que inclui um relatório e algumas medições básicas desenvolvidos

sobre o próprio processo atual do engenheiro de software. Este estágio provê uma base consistente para medir o progresso e uma fundação definida sobre a qual melhorar. Para o desenvolvedor que não possui um processo regular Humphrey sugere que o PSP seja desenvolvido incluindo as fases de projeto, codificação, compilação e teste, em um processo seqüencial;

- PSP0.1: do PSP0 evolui-se para o PSP0.1 adicionando um padrão de codificação, medições de tamanho e uma proposta de desenvolvimento de processo (*Process Improvement Proposal* - PIP). O PIP é um formulário que provê uma forma estruturada de registrar problemas no processo, experiências e sugestões de melhoria;

- PSP1 - O processo de planejamento pessoal: o PSP1 adiciona passos de planejamento ao PSP0. O incremento inicial adiciona um relatório de teste e estimativas de tamanho e recursos;

- PSP1.1: são introduzidos planejamentos de tempo e tarefas;

- PSP2 - o processo de administração da qualidade pessoal: para administrar os seus defeitos, o desenvolvedor tem que saber quantos faz. O PSP2 acrescenta técnicas de revisão ao PSP1, para ajudar a achá-los no início, quando forem menos caros para resolver. Faz-se isto juntando e analisando os defeitos achados na compilação e nos testes dos primeiros programas. Com estes dados, o desenvolvedor pode estabelecer listas de conferência de revisão e fazer suas próprias avaliações de qualidade de processo;

- PSP2.1: estabelece critérios de verificação e de finalização do projeto e examina várias técnicas de verificação e consistência de projeto;

- PSP3: Processo pessoal cíclico: até a fase anterior o PSP considera um processo cascata. É possível trabalhar com processo em cascata para programas pequenos, no entanto, este processo não é adequado para grandes projetos. Assim, a estratégia do PSP3 é subdividir um programa maior em pedaços do tamanho requerido pelo PSP2. A primeira construção é um módulo básico ou núcleo, que

aumenta em ciclos de iteração. Em cada iteração, faz-se um PSP2 completo, incluindo projeto, codificação, revisões, compilação e testes.

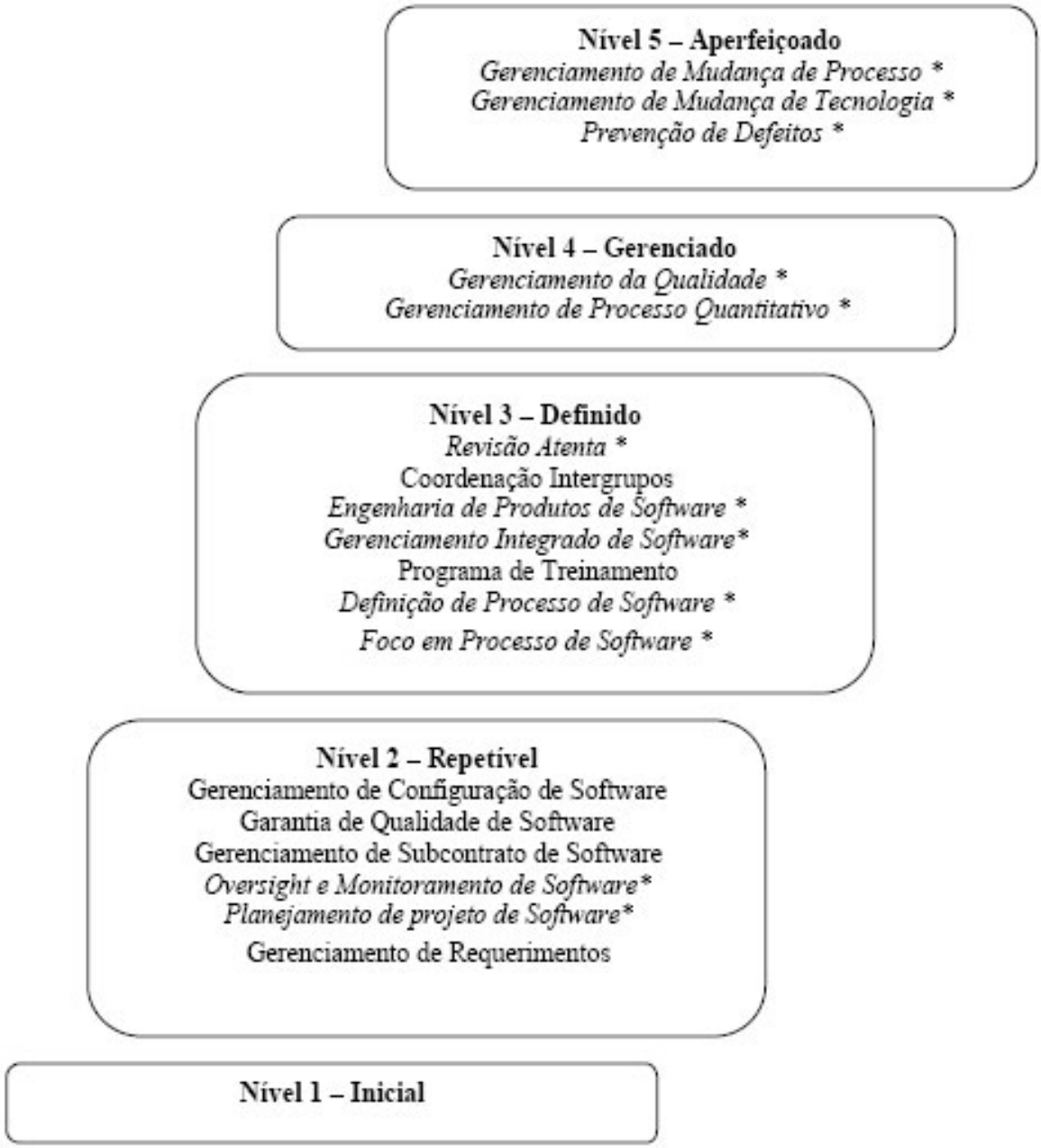
2.4.2 PSP e CMM

Como o PSP foi criado pelo mesmo autor do CMM , apresentado na Seção 2.3.3, Watts Humphrey, ele possui uma estrutura de maturidade semelhante ao CMM. O PSP surgiu de uma necessidade de simplificação das idéias pregadas pelo CMM, de proporcionar a pequenas empresas e desenvolvedores a oportunidade de crescerem profissionalmente em termos de disciplina e qualidade no desenvolvimento de software, dando a eles uma preparação para uma futura avaliação organizacional no CMM (Humphrey, 2000). O PSP e o CMM se baseiam no princípio da melhoria do processo de desenvolvimento. Enquanto o CMM se concentra na melhoria da capacidade organizacional, o PSP objetiva a melhoria do desenvolvedor de um ponto de vista individual (Humphrey, 1995).

Conforme mostra a Figura 2.8, o PSP permite atender, pelo menos de forma parcial, a doze das dezoito KPA's do CMM (KPA's em *itálico* seguidas de *), deixando clara a estreita relação existente entre os dois (Humphrey, 1995, 2000), sendo que as seis áreas restantes não podem ser acomodadas dentro de projetos de pequena escala (Humphrey, 1996) e dizem respeito a questões organizacionais. Portanto o PSP, se usado de forma correta, capacita os desenvolvedores dentro dos padrões exigidos pelo modelo CMM (Humphrey, 1995) e é uma alternativa de custo mais baixo que a implantação do CMM, portanto mais viável para pequenas empresas. Mesmo tendo seu foco dirigido ao desenvolvedor e não à empresa, há vantagens comprovadas na sua adoção em uma etapa preparatória para uma avaliação CMM (Humphrey, 1995).

2.4.3 O Curso PSP

O PSP assumiu uma forma mais didática em 1995, quando Humphrey publicou suas idéias sobre o processo pessoal em seu livro *A Discipline for Software Engineering* (Humphrey, 1995), voltado principalmente para alunos de graduação e pós-graduação.



Fonte: Adaptado de Humphrey (1995)

Figura 2.8: KPA's do CMM cobertas pelo PSP

O livro foi estruturado em forma de um curso, contendo todos os formulários, diretrizes e exercícios necessários para assimilação da estratégia do PSP. O livro traz consigo um programa com os passos a se seguir para a aplicação de cada exercício. Antes de uma tarefa se faz necessário a apresentação de um conjunto de capítulos do livro que contém toda a base teórica correspondente ao exercício em questão.

Em 1997 Humphrey lançou outro livro chamado *Introduction to the Personal Software Process* (Humphrey, 1997), que possui uma linguagem mais simples e é voltado para alunos de curso médio. Recentemente foi lançado outro livro chamado *PSP: A Self-Improvement Process for Engineers* (Humphrey, 2005) que cita vários exemplos bem sucedidos de empresas que adotaram o PSP.

Após Humphrey publicar seu primeiro livro em 1995, muitos outros registros de experiências com o curso de PSP foram relatados, tanto no ambiente acadêmico (Paula Filho, 2003; Prechelt and Unger, 2001; Albuquerque et al., 1999; Maletic et al., 2001; Towhidnejad and Hilburn, 1997) quanto no ambiente empresarial (Kelly and Culleton, 1999; Ferguson et al., 1997; Hayes and Over, 1997), e obtiveram resultados similares. Estes dados mostram que após a experiência com o PSP, a precisão das estimativas dos desenvolvedores cresceu consideravelmente, o número de defeitos por linha de código (LOC) diminuiu em um fator de três ou mais e a produtividade não diminuiu apesar do *overhead* para registros e contabilização dos dados.

No aprimoramento de qualidade dos desenvolvedores a eficácia do PSP também tem sido amplamente registrada. Prechelt and Unger (2001) aplicaram a bateria de exercícios de programação em dois grupos: um grupo que havia sido submetido ao curso de PSP e um grupo que havia recebido outras técnicas de treinamento. O resultado da pesquisa mostrou que o grupo do PSP estimou sua produtividade muito mais precisamente que o outro grupo, cometeu poucos erros triviais, executou uma checagem de erros bem mais cuidadosa e, além disso, a variabilidade de performance entre os membros do grupo do PSP foi menor em vários aspectos.

2.4.4 PSP no Ambiente Acadêmico

A visão de produção de software como um trabalho de engenharia implica, em primeiro lugar, em manter a disciplina em todas as fases do desenvolvimento do software. No entanto disciplinar um profissional, que geralmente não está acostumado com este tipo de prática, pode ser extremamente complicado. Além da resistência inerente a qualquer processo de mudança, a introdução de um maior rigor na forma de trabalho do desenvolvedor pode, em um primeiro momento, trazer alguma queda de produtividade, embora este não seja o caso das experiências com PSP citadas na literatura, o que aumenta ainda mais a resistência a esta mudança. Além disso, quanto mais tardio for este processo de mudança de perfil maior será a resistência imposta pelo indivíduo (Balbino, 2004).

No processo de aprendizagem dos alunos, o ideal seria que as instituições formadoras de profissionais trabalhassem o perfil de seus estudantes para que estes ingressassem no mercado com as melhores práticas de desenvolvimento de software já fixadas. No entanto, a existência de uma lacuna entre as necessidades da indústria e a educação acadêmica de Engenharia de Software tem sido amplamente registrada (Dion, 1993; Hilburn et al., 1995). A indústria requer engenheiros de software, mas as universidades têm formado cientistas da computação (Hilburn et al., 1995). Estudos recentes (Towhidnejad and Hilburn, 1997) de um instituto de Engenharia de Software registrou sucintamente o estado do problema: "Os estudantes não estão preparados para fazer o salto da ciência da computação ou a engenharia de software na escola para a engenharia no mundo profissional".

Um dos principais motivos desta lacuna entre o que os estudantes aprendem nas universidades e o que a indústria de software espera dos profissionais recém formados é a carência de programas acadêmicos voltados para a competência necessária nas tarefas da vida profissional. Dentre estas competências estão o desenvolvimento de grandes projetos, a habilidade de trabalhar em equipe, o conhecimento de processo, a qualidade e habilidade de planejar, estimar tempo e elaborar custos (Towhidnejad and Hilburn, 1997).

A inclusão do PSP nas atividades das universidades tem sido freqüentemente

registrada (Albuquerque et al., 1999; Prechelt and Unger, 2001; Towhidnejad and Hilburn, 1997; Maletic et al., 2001). Tais atividades incluem além do aprendizado do próprio PSP, o desenvolvimento dos exercícios propostos por Humphrey (1995). Estes estudos demonstram claramente o progresso trazido pelo PSP, justamente nas questões onde a indústria mais tem necessidade (Balbino, 2004).

As experiências de aplicação do PSP permitem aos instrutores armazenar uma grande quantidade de dados que podem ser utilizadas posteriormente de forma benéfica. Estes dados podem constituir a base para uma detalhada análise a respeito da efetividade, produtividade e qualidade do trabalho dos estudantes (Towhidnejad and Hilburn, 1997).

2.4.5 PSP no Ambiente Industrial

Várias organizações que aplicaram os princípios do PSP mostraram uma melhoria substancial de performance de seus desenvolvedores, um conseqüente aumento de qualidade dos produtos, redução do tempo de desenvolvimento e uma melhoria na comunicação da gerência com as equipes de software (Kelly and Culleton, 1999; Ferguson et al., 1997).

Estudos realizados por Hayes and Over (1997) mostram a contribuição do PSP no aprimoramento contínuo da performance dos engenheiros de software que fazem uso da estratégia. Os estudos procuraram analisar o impacto do PSP em cinco fatores ligados ao desenvolvimento de software: precisão nas estimativas de tamanho e esforço, qualidade do produto, qualidade do processo e produtividade pessoal. Ficou comprovado que o PSP melhorou a performance nos quatro primeiros fatores, sem qualquer perda de produtividade. Estes resultados têm importantes implicações para qualquer organização de desenvolvimento de software, pois indicam que o PSP pode ajudar os desenvolvedores a atingir melhorias em quatro áreas de importância crítica na perspectiva do negócio: custo e cronograma, qualidade do produto, tempo de ciclo do produto e melhoria do processo de desenvolvimento. Resultados de estudos realizados por Ferguson et al. (1997) também apoiam esta conclusão.

Conforme mostrado na Seção 2.1, uma necessidade crítica do processo de negó-

cio para todas as organizações que desenvolvem software é melhorar o gerenciamento de custo e cronograma. Problemas de custos e cronogramas geralmente começam quando os projetos assumem compromissos que são baseados em estimativas inadequadas de tamanho e recursos de desenvolvimento. O PSP ajuda a solucionar este problema mostrando para os engenheiros como melhorar suas estimativas usando técnicas estatísticas e dados históricos (Hayes and Over, 1997).

Outra necessidade crítica é a qualidade do software. Um gerenciamento pobre de qualidade limita a habilidade do desenvolvedor, aumenta o custo de desenvolvimento do software e torna o desenvolvimento de cronogramas muito mais difíceis de predizer. Muitos destes problemas de gerenciamento de qualidade derivam do fato de se delegar aos testes a responsabilidade da qualidade do software. No entanto, detectar e remover defeitos no teste é muito mais custoso, não efetivo e imprevisível (Hayes and Over, 1997; Humphrey, 1995). A forma mais eficiente e efetiva de gerenciar a qualidade do software é pela adoção de um programa abrangente focado na remoção de defeitos o mais cedo possível, ou seja, na sua própria fonte. O PSP ajuda o desenvolvedor a encontrar e remover defeitos onde eles são injetados, antes da compilação e dos testes. Com poucos defeitos para remover nos testes os custos são reduzidos drasticamente, os cronogramas são mais previsíveis, gastos com manutenção são reduzidos e a satisfação do usuário será maior (Hayes and Over, 1997).

Outra meta de uma organização deve ser reduzir o tempo de ciclo do produto. O tempo de ciclo pode ser reduzido pelo uso de um melhor planejamento e eliminação de retrabalho através do desenvolvimento da qualidade do produto. Planejamentos precisos permitem desenvolver cronogramas mais ajustados e aumentam a concorrência entre as atividades planejadas. A melhoria da qualidade através da remoção rápida dos defeitos, além de reduzir os gastos, favorece o aumento da precisão do planejamento na medida que reduz a principal fonte de variação, que é a descoberta e correção de defeitos apenas nos testes (Hayes and Over, 1997). Com o PSP, os desenvolvedores aprendem como reunir os dados do processo necessários para minimizar o tempo de ciclo. Esses dados ajudam a construir planejamentos mais exatos, eliminando re-trabalho e reduzindo os testes em até quatro ou cinco vezes (Ferguson

et al., 1997).

É bem entendido por todos que a melhoria do processo de desenvolvimento de software aumenta a vantagem competitiva, mas é difícil envolver os engenheiros de software nesta melhoria de processo. Geralmente estes vêem esta responsabilidade como uma atividade exclusiva da equipe de gerenciamento. Com o PSP, os desenvolvedores ganham experiência com a melhoria do processo e tornam-se os "donos" do processo, envolvidos diretamente na medição, gerenciamento e melhoria do processo de desenvolvimento (Hayes and Over, 1997).

Como foi exposto, muitas vezes existe uma grande resistência, principalmente no ambiente industrial, a todo processo que impõe um certo nível de disciplina. Por este motivo, o PSP foi desenvolvido para ser um processo de auto-convencimento.

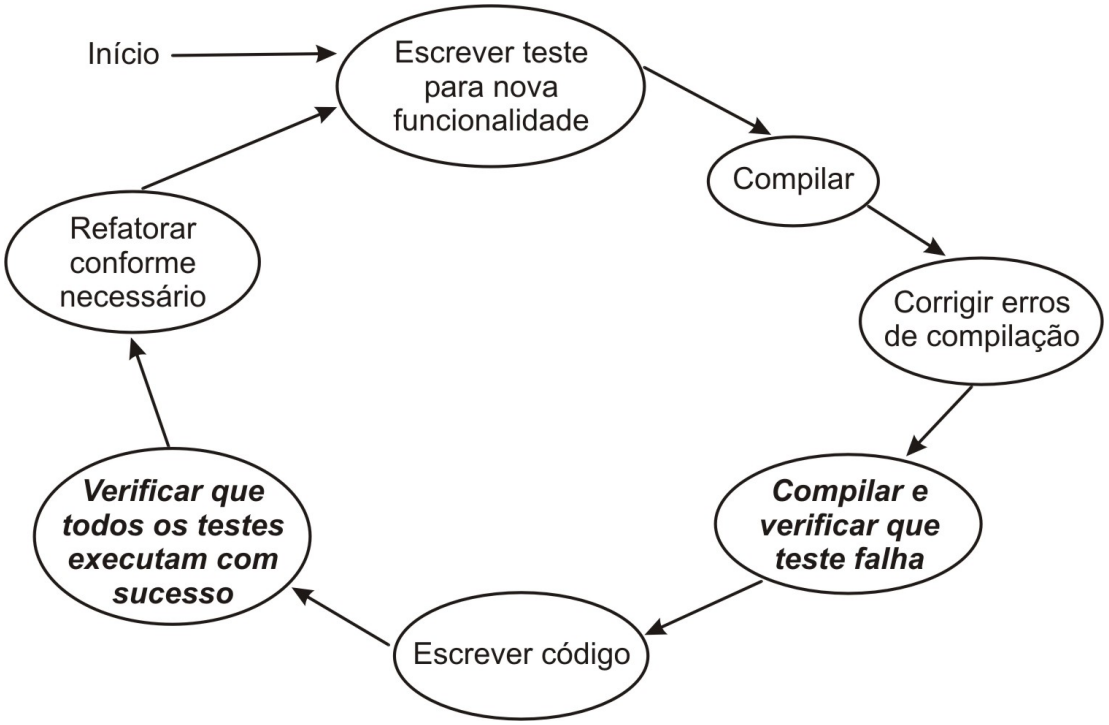
2.5 Test Driven Development (TDD)

Teste de software é um elemento crítico da garantia de qualidade de software e representa a revisão final da especificação, projeto e geração de código (Pressman, 2001).

Test Driven Development (TDD) (Beck, 2003) é uma prática de desenvolvimento de software que tem sido usada há décadas (Gelperin and Hetzel, 1987). Era usada na NASA por volta de 1960 no projeto Mercury (Larman and Basili, 2003). A prática de TDD ganhou uma maior visibilidade recentemente, após ter sido incorporada pelo *eXtreme Programming* (XP) (Beck, 2000). TDD é também conhecida por nomes como *Test First Design* (TFD), *Test First Programming* (TFP) e *Test Driven Design* (TDD) (George and Williams, 2003). A prática desenvolve-se iniciando o projeto de um sistema com casos de *teste de unidade* para um determinado objeto. Uma regra importante em TDD é: "Se você não pode escrever casos de teste sobre o que você quer codificar, então você sequer deveria pensar na codificação"(Chaplin, 2001).

O *teste de unidade* tem por objetivo explorar a menor unidade do projeto, procurando identificar erros de lógica e de implementação em cada módulo, separadamente (Rocha et al., 2001). Segundo o padrão IEEE 610.12-1990 (IEEE, 1990), uma unidade é um componente que não pode ser subdividido.

No TDD, antes de o desenvolvedor escrever a implementação do código, ele escreve casos de teste de unidade automáticos para cada nova funcionalidade que irá implementar. Após escrever o caso de teste, o desenvolvedor compila e verifica que a execução do teste falha. Em seguida o desenvolvedor implementa o código da nova funcionalidade com o objetivo de passar no caso de teste. A nova funcionalidade só é considerada implementada após ser validada pelos casos de teste. O desenvolvedor escreve um pouco de casos de testes, implementa o código, escreve mais um pouco de casos de teste, implementa mais código, e assim sucessivamente, conforme mostra a Figura 2.9. O trabalho é mantido dentro dos limites intelectuais do desenvolvedor, por que ele continuamente faz um pouco de projeto e um pouco de implementação, aumentando gradativamente as funcionalidades do sistema sem aumentar a complexidade a um nível não gerenciável.



Fonte: Elaborado pelo Autor

Figura 2.9: Ciclo de execução do TDD

Intelectualmente, podemos considerar TDD de vários pontos de vista (George and Williams, 2003; Erdogmus, 2005):

- **Feedback**: em qualquer processo, existe uma lacuna entre decisão (projeto desenvolvido) e *feedback* (desempenho obtido pela implementação do projeto). O sucesso do TDD pode ser atribuído a redução, se não eliminação, dessa lacuna, porque o ciclo granular teste-codifica fornece um constante *feedback* ao desenvolvedor. Em consequência, os defeitos e as causas dos defeitos podem ser facilmente identificados;

- **Orientação de tarefas**: atividades de codificação dirigidas por testes encorajam os programadores a decompor o problema em pedaços gerenciáveis, formalizando suas tarefas de programação, ajudando-o a manter o foco e fornecendo um progresso mensurável constante;

- **Garantia de qualidade**: TDD seduz os desenvolvedores a escrever código que é automaticamente testável, com funções/métodos retornando valores que podem ser checados com os resultados previstos. Os benefícios de automatização de testes incluem: 1) produção de um sistema confiável, 2) melhoria de empenho do teste e 3) redução de esforço do teste e minimização do cronograma. Manter armazenados os testes atualizados e executá-los frequentemente, garantem a qualidade do software através do teste de regressão, onde são facilmente identificados se as alterações introduziram algum tipo de defeito no sistema;

- **Projeto de baixo nível**: testes fornecem o contexto no qual são tomadas as decisões de níveis mais baixos nos projetos, como por exemplo a escolha de classes e métodos a serem criados, como serão nomeados, quais interfaces possuirão e como serão usadas.

Vários estudos têm sido conduzidos abordando TDD, incluindo diversos autores e segmentos diferentes, como academia e indústria, dos quais podemos citar vários trabalhos: (Muller and Hagner, 2002; George and Williams, 2003; Mugridge, 2003; Williams et al., 2003; Pancur et al., 2003; Maximilien and Williams, 2003; Edwards, 2003; Jones, 2004; Tilley, 2004; Erdogmus and Wang, 2004; Erdogmus, 2005).

A prática TDD é flexível e pode ser adaptada a outros métodos, sejam Métodos Dirigidos por Planejamento ou Métodos Ágeis (Boehm and Turner, 2004). Um bom

exemplo é o processo XP, que incorpora suas técnicas de projeto e testes de unidades (Williams et al., 2003).

2.6 Métodos Híbridos

A natureza complexa do desenvolvimento de software e a grande variedade de métodos torna a comparação entre os Métodos Dirigidos por Planejamento e os Métodos Ágeis difícil e imprecisa (Boehm and Turner, 2004). Há características importantes relacionadas ao projeto de software que permitem delinear com mais clareza as categorias de problemas a que cada classe de métodos pode ser aplicada com mais sucesso.

Os Métodos Dirigidos por Planejamento podem ser caracterizados pelos seguintes conceitos: melhoria e aquisição de competência em processos, maturidade organizacional, gerência de risco, gerência da qualidade, verificação e validação de sistemas, e utilização de arquiteturas de software que favoreçam a reutilização pelo uso de componentes (Boehm and Turner, 2004). O PSP é um exemplo dessa classe de métodos.

Métodos Ágeis, por sua vez, podem ser caracterizados pelos seguintes conceitos: permitir alterações e mudanças com facilidade (*embrace change*), ciclos curtos e freqüentes entregas de versões, projeto simplificado, adoção de refatoração (*refactoring*) como técnica de melhoria e padronização no texto dos programas, programação em pares (*pair programming*), compartilhamento do conhecimento adquirido nos projetos e desenvolvimento dirigido por testes (TDD) (Boehm and Turner, 2004). XP, Crystal e Scrum são exemplos dessa classe de métodos (Ambler, 2002).

O grande desafio é estabelecer uma linha de separação clara entre os tipos de problemas a que as classes de métodos citadas se aplicam. Problemas cujo contexto muda constantemente, exigindo adaptação constante do software, são mais propensos aos Métodos Ágeis. Problemas que têm requisitos mais estáveis, exigindo processo definido para desenvolvimento, documentação detalhada e que possam envolver riscos de vida são mais propensos a Métodos Dirigidos por Planejamento (Boehm and Turner, 2004).

O amadurecimento das duas tendências permite visualizar melhor uma tênue linha de separação entre elas. Segundo a proposta contida em (Boehm and Turner, 2004), as variáveis que devem ser levadas em consideração são: *tamanho*, relacionado ao esforço necessário ao desenvolvimento; *criticalidade* relacionada ao risco a vidas humanas pelo uso do software, com impacto direto no nível de qualidade e possibilidade de falhas; *dinamismo* relacionado à possibilidade de mudanças no ambiente que possam se refletir em mudanças no software; *competência do pessoal* envolvido no desenvolvimento, relacionado ao tipo de profissional necessário ao desenvolvimento, variando de pouco experientes até especialistas e finalmente *cultura* relacionada à zona de conforto para cada classe de métodos. Métodos Ágeis se baseiam mais na cultura em que os desenvolvedores se sentem mais confortáveis tendo muitos graus de liberdade para produzir o software, e os Métodos Dirigidos por Planejamento se baseiam mais na cultura em que os desenvolvedores se sentem mais confortáveis trabalhando com regras, procedimentos e políticas bem definidas e estabelecidas.

O que fica bem visível na literatura sobre o assunto é que os Métodos Dirigidos por Planejamento e os Métodos Ágeis têm muito a lucrar um com o outro. Características dos Métodos Ágeis podem ser introduzidas em Métodos Dirigidos por Planejamento aumentando o leque de problemas a que eles são mais naturalmente aplicáveis, e vice-versa. Surgem assim o que pode ser denominado de Métodos Híbridos, com características de ambas as tendências.

2.7 Engenharia de Software Experimental

Experimentação é o centro do processo científico. Somente experimentos verificam as teorias. Somente experimentos podem explorar os fatores críticos e dar luz ao fenômeno novo para que as teorias possam ser formuladas e corrigidas. Experimentação oferece o modo sistemático, disciplinado, computável e controlado para avaliação da atividade humana. Novos métodos, técnicas, linguagens e ferramentas não deveriam ser apenas sugeridos, publicados ou apresentados para venda sem experimentação e validação. Portanto, é preciso avaliar novas invenções e sugestões em comparação com

as existentes (Travassos et al., 2002).

Os objetivos relacionados à execução de experimentos em Engenharia de Software são a caracterização, avaliação, previsão, controle e melhoria a respeito de produtos, processos, recursos, modelos e teorias entre outros. A importância e o esforço de um experimento com o objetivo *caracterização* aumentam em relação a um experimento com o objetivo *melhoria*. Isso significa que é bastante simples conduzir um experimento com a finalidade de caracterização respondendo questões do tipo *"o que está acontecendo?"*. É mais difícil medir algo, por exemplo, um processo ou produto e defini-lo *"quão bom é isto?"*. Os experimentos com a finalidade de previsão além da medição precisam de meios de estimativa para mostrar a possibilidade de responder perguntas como: *"posso estimar algo no futuro?"*. Para atender à finalidade de controle deve existir a possibilidade de gerenciar os atributos de um processo ou produto e dar a resposta à pergunta *"posso manipular o evento?"*. Finalmente, a finalidade da melhoria supõe que possamos caracterizar, avaliar, predizer e controlar, e há os objetivos da melhoria de um processo ou produto que possam ser atingidos respondendo a última questão *"posso melhorar o evento?"* (Travassos et al., 2002).

Segundo Amaral (2003), um experimento deve ser tratado como um processo de formulação e verificação de uma teoria. A fim de que ofereça os resultados válidos, ele deve ser propriamente organizado e controlado ou, pelo menos, acompanhado.

Um processo bem definido pode ser observado e medido e desta forma melhorado. Processos podem ser usados para capturar as melhores práticas para se tratar um determinado problema. A utilização de um determinado processo permite que práticas de trabalho sejam disseminadas mais rapidamente do que a construção de experiência pessoal.

A literatura oferece algumas outras definições dos objetivos da experimentação em Engenharia de Software. Em Conradi et al. (2001) são listados:

- Para compreender a natureza dos processos da informação os pesquisadores devem observar o fenômeno, encontrar explicação, formular a teoria, e verificá-la;

- A experimentação pode ajudar a construir uma base de conhecimento confiável e reduzir assim incerteza sobre quais teorias, ferramentas, e metodologias são adequadas;

- A observação e experimentação podem levar a novos e úteis meios da introspecção, e abrir novas áreas de investigação. A experimentação pode encontrar novas áreas onde a engenharia age lentamente;

- A experimentação pode acelerar o processo eliminando abordagens inúteis e suposições errôneas. A experimentação ajuda também a orientar a engenharia e a teoria nas direções promissoras de pesquisa;

- Os experimentos podem ser custosos, mas um experimento significativo geralmente pode se encaixar no orçamento de um pequeno laboratório. Por outro lado, um experimento caro pode valer a pena muito mais do que seu custo e, por exemplo, oferecer à companhia liderança de três, quatro ou cinco anos sobre a competição;

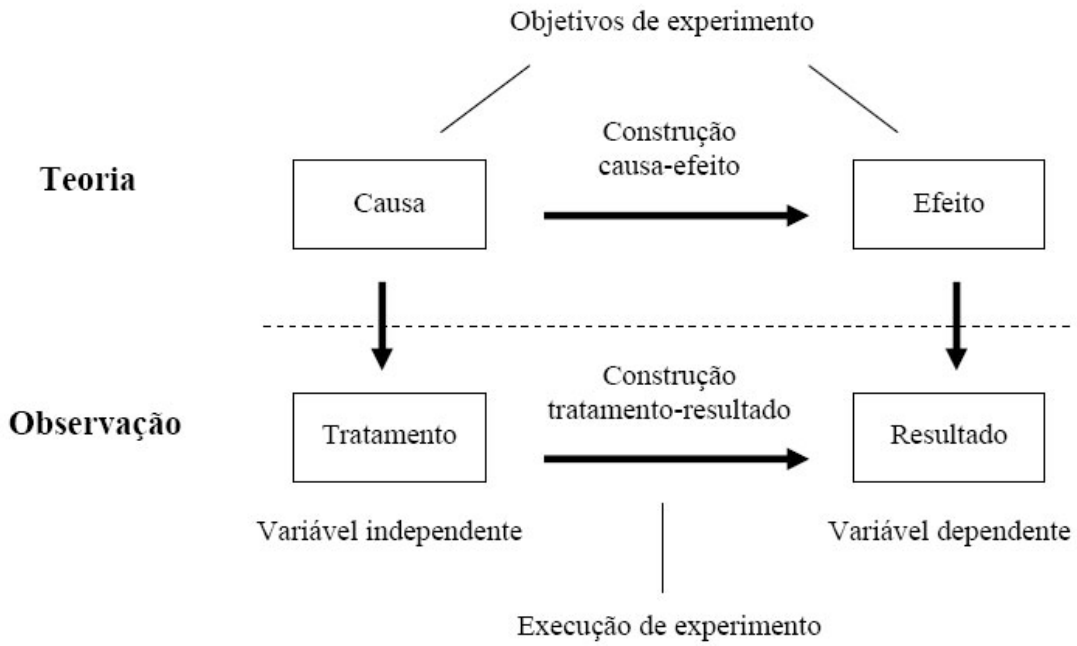
- O crescimento do número de trabalhos científicos com uma validação empírica significativa possui a boa chance de acelerar o processo de formação da Engenharia de Software como ciência. As idéias duvidosas serão rejeitadas mais rapidamente e os pesquisadores poderão concentrar-se nas abordagens promissoras;

- A tecnologia vem se modificando rapidamente. As mudanças sempre trazem ou eliminam as suposições. Os pesquisadores devem então antecipar as mudanças nas suposições e aplicar os experimentos para explorar as conseqüências dessas mudanças.

Os elementos principais do experimento são as variáveis, os objetos, os participantes, o contexto do experimento, hipóteses, e o tipo de projeto do experimento (Travassos et al., 2002). Há dois tipos de variáveis do experimento: dependentes e independentes. As variáveis independentes referem-se à entrada do processo de experimentação. Essas variáveis também se chamam *fatores* e apresentam a causa que afeta

o resultado do processo de experimentação. O próprio valor de um fator se chama *tratamento*. As variáveis dependentes referem-se à saída do processo de experimentação. Essas variáveis apresentam o efeito que é causado pelos fatores do experimento. O próprio valor de uma variável dependente se chama *resultado*.

A Figura 2.10 apresenta os relacionamentos entre os conceitos descritos acima.



Fonte: Travassos et al. (2002)

Figura 2.10: Os conceitos de um experimento (Wohlin, 2000)

Os participantes são os indivíduos que foram especialmente selecionados da população sob interesse para conduzir o experimento. Isso significa que para generalizar os resultados de um experimento a uma população desejada, o conjunto de participantes deve ser representativo para aquela população. Para atingir a meta mencionada os parâmetros como o modo de seleção dos participantes e o tamanho do conjunto selecionado que influem o resultado do experimento, devem ser considerados. A princípio, quanto maior é a variedade da população tanto maior deve ser o tamanho do conjunto de participantes.

O contexto do experimento é composto das condições em que o experimento está sendo executado. O contexto pode ser caracterizado de acordo às três dimensões:

- *In-vitro vs. In-vivo*: O primeiro refere-se à experimentação no laboratório sob as condições controladas. O segundo considera o estudo de um projeto real.
- Alunos vs. Profissionais: Define a equipe que vai executar o experimento.
- Problema de sala de aula vs. Problema real: Mostra o tamanho do problema que está sendo estudado.
- Específico vs. Geral: Mostra se os resultados do experimento são válidos para um contexto particular ou para o domínio da Engenharia de Software.

Um experimento geralmente é formulado através de hipóteses. A hipótese principal se chama hipótese nula e declara que não há nenhum relacionamento estatisticamente significante entre a causa e o efeito. O objetivo principal do experimento é, então, rejeitar a hipótese nula a favor de uma ou algumas hipóteses alternativas. A decisão sobre rejeição da hipótese nula pode ser tomada baseado nos resultados da sua verificação utilizando um teste estatístico.

O processo da execução de um experimento presume a realização de diferentes atividades. O número e a complexidade dessas atividades podem variar de acordo com as características do estudo. A literatura apresenta cinco fases gerais que sempre estão presentes num processo da experimentação (Travassos et al., 2002).

A definição é a primeira fase onde o experimento é expresso em termos dos problemas e objetivos. A fase de planejamento vem em seguida onde o projeto do experimento é determinado, a instrumentação é considerada e os aspectos da validade do experimento são avaliados. A execução do experimento segue o planejamento. Nesse momento os dados experimentais são coletados para serem analisados e avaliados na fase de análise e interpretação. Finalmente, os resultados são apresentados e empacotados durante a fase da apresentação e empacotamento (Amaral, 2003).

Capítulo 3

PSPm - PSP modificado

"Para mudar seu destino, você precisa primeiro mudar sua atitude".

3.1 Introdução

Um caminho promissor para a obtenção de qualidade na produção de software é investir na melhoria do nível de qualidade pessoal de cada programador. O PSP tem o objetivo de auxiliar nesse processo de auto-conhecimento e posterior melhoria de competência em produção de software, fornecendo mecanismos para que o programador alcance a disciplina e a organização necessárias.

Segundo alguns autores (Balbino and Braga, 2004; Pfleeger, 2001), programadores inserem uma parte significativa dos defeitos nas fases de Análise e Projeto do software, os quais podem ser atribuídos às falhas de comunicação ou ao não entendimento dos requisitos de usuário. Esses defeitos, caso não sejam descobertos no processo de desenvolvimento do software, podem causar inúmeros transtornos aos usuários, o que é considerado negativo do ponto de vista de qualidade. Se defeitos forem injetados, o melhor é que sejam identificados e corrigidos o mais cedo possível no processo. Isso tem impacto positivo na redução do re-trabalho e do risco de introdução de novos defeitos.

Uma das possíveis causas para a introdução de defeitos nas fases iniciais do processo é o pouco entendimento dos requisitos do problema que está sendo resolvido.

A análise do problema tende a ser deficiente e conduzida em curto espaço de tempo, e os programadores têm a tendência de querer partir logo para a fase de produção de código, deixando os defeitos para serem descobertos nos testes. O ciclo "codificação $\Longleftrightarrow$ testes" tende a ser o padrão para a solução da maioria dos problemas. A conseqüência é um aumento significativo do tempo gasto na produção do software, com uma diminuição no padrão de qualidade do produto final, que ficará mais propenso a apresentar defeitos e a não satisfazer às necessidades do usuário final.

3.2 PSPm

O uso do PSP tende a disciplinar os programadores, eliminando ou atenuando o ciclo "codificação $\Longleftrightarrow$ testes", ajudando-os a melhorar a capacidade de planejamento, acompanhamento e qualidade dos resultados. Mesmo utilizando o PSP, muitos programadores têm dificuldade de entender os requisitos de usuário na hora de escrever o projeto do software, e isso pode implicar na inserção de defeitos no software final, não satisfazendo assim, as necessidades dos usuários. Em conseqüência disto, a remoção destes defeitos torna-se cara, aumentando o volume de re-trabalho e as chances de inserção de novos defeitos.

O TDD oferece o benefício de ajudar os programadores a entenderem de forma mais clara os requisitos de usuário, utilizando para isto, a elaboração de casos de teste de unidade antes da codificação - *test-first*. Segundo Erdogmus (2005) TDD ajuda a quebrar o problema em partes menores e a entender melhor os requisitos através da elaboração de casos de testes antes da codificação.

O principal objetivo da Engenharia de Software é produzir software de qualidade e, tanto o PSP quanto o TDD procuram levar os programadores a alcançarem esse objetivo. Visando isto, esse trabalho propõe a inserção da técnica *test-first* oriundo do TDD na fase Projeto do PSP, o que permitirá ao programador elaborar incrementalmente os testes de unidade para um determinado requisito antes de elaborar o projeto para o mesmo, ajudando aos programadores a trabalharem com um processo pessoal bem definido e inserindo uma menor quantidade de defeitos. Dessa

forma, pode-se obter um processo híbrido agregando os benefícios já proporcionados pelo PSP com o incremento das melhorias que o TDD oferece.

O PSP modificado - PSPm é uma proposta de evolução do PSP, com a introdução da técnica *test-first* do TDD na fase Projeto do PSP. A Tabela 3.1 apresenta o *script* de execução do PSPm, sendo que as partes em **negrito** representam as principais alterações propostas pelo PSPm.

3.2.1 Projetando o Software

Um bom projeto de software transforma um requisito mal definido e ambíguo em uma especificação de produto implementável. A qualidade do projeto é muito importante, porque é quase impossível produzir uma implementação de alta qualidade de um projeto de má qualidade. A qualidade do projeto tem duas partes: a qualidade do conteúdo do projeto e a qualidade de representação do projeto.

O PSP direciona a prevenção de defeitos no projeto do ponto de vista da evolução, onde existe uma grande probabilidade de o programador se tornar mais cuidadoso com o aumento de sua consciência ao construir o código. Defeitos de projeto, entretanto, são mais difíceis de reduzir (Humphrey, 1995). O PSP não especifica um método para o projeto, mas diz que existe a necessidade de melhorar a qualidade dos projetos de software. Ele se concentra na qualidade da representação do projeto (Humphrey, 1995).

Partindo deste ponto, o PSPm leva vantagem em relação ao PSP pois, além de o PSPm possuir todas as vantagens que o PSP oferece, o PSPm se concentra também na qualidade do conteúdo do projeto, que é feita através da elaboração de testes de unidade antes de iniciar o projeto do software, o que força o programador a quebrar o problema em partes menores e eliminar as ambiguidades, definindo assim quais serão as entradas e quais serão as saídas para cada caso de teste. Isso torna o projeto menos suscetível a erros.

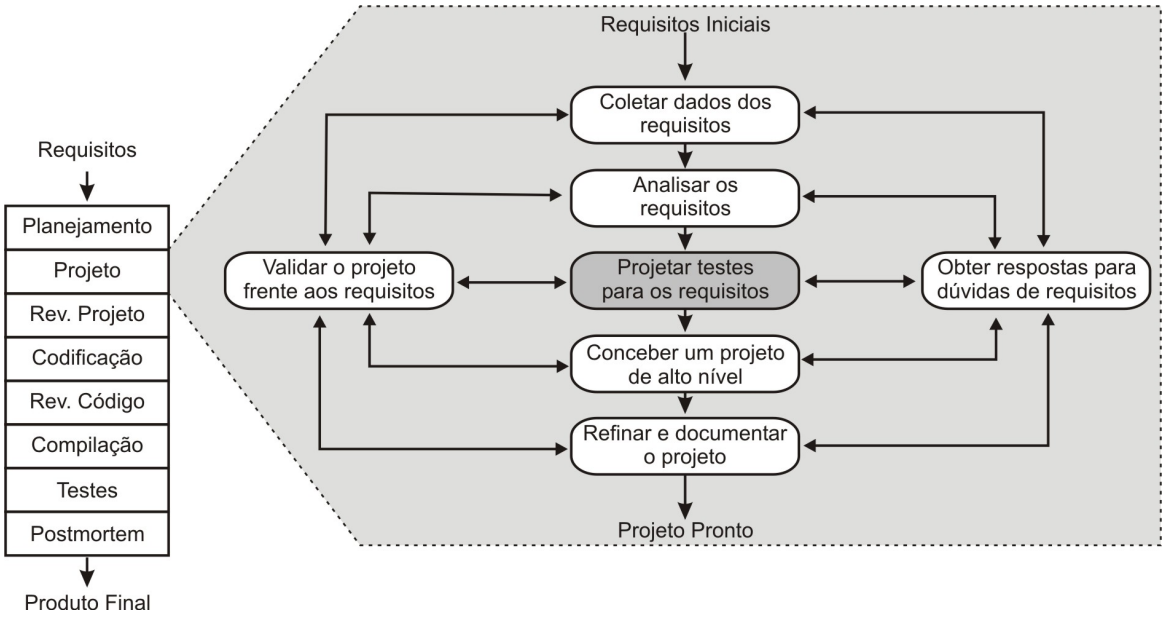
A Figura 3.1, adaptada de Humphrey (1995), mostra um *framework* definido para o PSPm, com uma expansão na fase Projeto, onde a ténica *test-first* é inserida. A parte em destaque no centro do fluxo mostra a alteração principal no processo, onde

Tabela 3.1: Script de execução do PSPm

Objetivo	Ser um guia no desenvolvimento de pequenos programas
Critérios de Entrada	Descrição do Problema Formulários de planejamento do PSP - *Project Plan Summary* *Time Recording Log* e *Defect Recording Log* Padrão dos tipos de defeitos - *Defect Type Standard*
1-Planejamento	Obter a descrição das funções do programa - requisitos Avaliar se os requisitos estão claros e sem ambiguidade Estimar o tempo de desenvolvimento do programa Entrar com os dados do planejamento no formulário *Project Plan Summary* Gravar o tempo de planejamento no *Time Recording Log*
2-Projeto	Revisar os requisitos **Escrever os testes para o programa** Projetar o programa Gravar o projeto em um formato específico Gravar o tempo gasto no projeto no *Time Recording Log*
3-Codificação	Implementar o projeto do programa Utilizar um padrão de codificação Gravar os defeitos encontrados no *Defect Recording Log* Gravar o tempo gasto na codificação no *Time Recording Log*
4-Compilação	Compilar o programa Corrigir os defeitos Gravar os defeitos encontrados no *Defect Recording Log* Gravar o tempo gasto na compilação no *Time Recording Log*
5-Teste	**Testar o programa com os testes projetados** Testar até que todos os testes sejam executados sem erros Corrigir todos os erros encontrados Gravar os defeitos encontrados no *Defect Recording Log* Gravar o tempo gasto no teste no *Time Recording Log*
6-Postmortem	Completar o formulário *Project Plan Summary* com o tempo real, defeitos e tamanho dos dados Gravar o tempo gasto no Postmortem no *Time Recording Log*
Critérios de Saída	Programa perfeitamente testado Projeto devidamente documentado *Project Plan Summary* completo com os dados estimados e reais *Time* e *Defect Recording logs* completos

Fonte: Elaborado pelo Autor a partir de dados de Humphrey (1995)

é feita a elaboração de testes de unidade.



Fonte: Extraído de Silva et al. (2006)

Figura 3.1: *Framework* da fase Projeto do PSPm

O projeto de software é um processo criativo que não pode ser reduzido a um procedimento rotineiro. Entretanto, o processo de projeto necessita ser estruturado. Geralmente se inicia o projeto pela definição do objetivo do produto, coletando os dados relevantes, produzindo um *overview* do projeto e preenchendo com os detalhes.

Uma especificação de projeto de software exige informações que incluem a definição de classes, objetos e seus relacionamentos, identificando interações entre eles, definindo os dados necessários e o estado de transformações, especificando as entradas e saídas do sistema.

3.3 Possíveis Melhorias no Processo

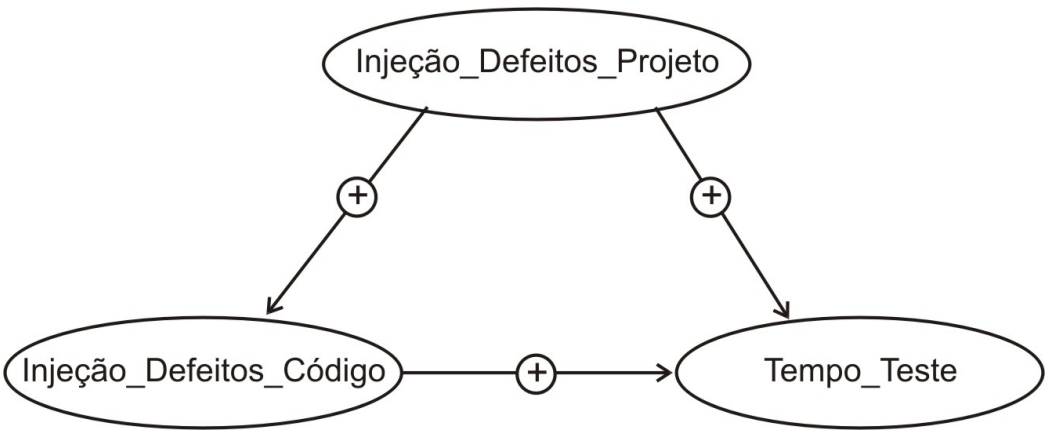
Um *defeito* ocorre quando uma pessoa comete um engano, chamado de *erro*, na realização de alguma atividade relacionada a um software. Por exemplo, um projetista pode não compreender corretamente um requisito e criar um projeto que não condiz com a verdadeira intenção do analista de requisitos e do usuário. Esse defeito

do projeto é uma codificação do erro e pode levar a outros defeitos, tais como código incorreto e descrição incorreta do manual do usuário. Portanto, um simples erro pode gerar muitos defeitos, e um defeito pode estar presente em qualquer produto do desenvolvimento ou da manutenção do software (Pfleeger, 2001).

Utilizando conceitos de Abordagem Sistêmica (Senge, 1994), foram identificadas as principais variáveis envolvidas neste contexto:

- *Injeção_Defeitos_Projeto* que representa o número de defeitos inseridos na fase de projeto do software;
- *Injeção_Defeitos_Código* que representa o número de defeitos inseridos na fase de codificação do software;
- *Tempo_Teste* que representa o tempo gasto na fase de testes do software.

Na Figura 3.2 apresentam-se as relações principais existentes entre essas variáveis. A notação utilizada é a de Diagramas de Influência (Senge, 1994) da Análise Sistêmica, onde uma relação anotada com "+" indica que as duas variáveis variam no mesmo sentido, e a anotada com "−" indica que elas variam no sentido inverso. Por exemplo, no diagrama deve-se ler: "se aumenta o valor da variável *Injeção_Defeitos_Projeto*, o valor da variável *Injeção_Defeitos_Código* também aumenta".



Fonte: Elaborado pelo Autor

Figura 3.2: Diagrama de Influência das principais variáveis do PSPm

Ainda de acordo com a Figura 3.2, pode-se observar o que foi dito por Pfleeger (2001): "*Esse defeito do projeto é uma codificação do erro e pode levar a outros defeitos...*". Quando se injeta um defeito no projeto do software, esse defeito é codificado, podendo ou não ser descoberto na fase de testes. Defeitos de projeto tendem a ser difíceis de serem descobertos, o que pode causar um aumento de tempo gasto no momento de testar o software. Neste caso, se o defeito for descoberto com os testes, ótimo!, pois, pode-se refazer o projeto e a codificação. Mas, o pior problema é quando esses defeitos são descobertos pelos usuários após a distribuição do software, onde a correção destes defeitos são extremamente caros e podem ocasionar a injeção de novos defeitos.

De acordo com este contexto, o PSPm atua diretamente na variável *Injeção_Defeitos_Projeto*, com o objetivo principal de reduzir a injeção de defeitos.

Além das melhorias proporcionadas pelo uso do PSP, espera-se que o PSPm leve os programadores a obterem as seguintes possíveis melhorias:

- *Melhorar o entendimento dos requisitos.* Segundo Erdogmus (2005), experimentos realizados com alunos mostram que o TDD ajuda a obter um melhor entendimento dos requisitos, uma vez que o programador, ao escrever os testes, expressa sem ambiguidade as funcionalidades dos requisitos de usuário;

- *Gerar código mais limpo e em menor tempo.* Uma vez que o escopo do problema a ser resolvido esteja bem definido, o programador irá escrever apenas o código necessário para resolver o problema especificado;

- *Reduzir o tempo na fase de teste.* Com uma codificação limpa e clara, fica mais fácil encontrar e corrigir os defeitos;

- *Produzir um software com menor número de defeitos e conseqüentemente com maior qualidade.* Partindo do princípio que defeitos influenciam diretamente a qualidade do produto final, quanto menor a quantidade de defeitos inseridos, maior a qualidade final do produto e maior a chance de o software satisfazer as necessidades do usuário final.

O PSPm tende a eliminar ou amenizar a injeção de defeitos no projeto do software, trazendo benefícios diretos à qualidade do processo de desenvolvimento de software e ao próprio software.

Capítulo 4

Metodologia, Resultados e Discussão

4.1 Introdução

Este capítulo apresenta a metodologia e a discussão dos resultados do estudo empírico realizado com os processos PSP e PSPm. O estudo foi projetado para verificar se o PSPm reduz a injeção de erros na fase de projeto do software e portanto, contribui para que programadores possam construir softwares de melhor qualidade, unindo boas práticas do PSP e do TDD. O experimento foi planejado de acordo com os arcabouços propostos em Pfleeger (1994), Seaman (1999) e Travassos et al. (2002).

4.2 Protocolo Experimental

O experimento consiste na implementação de 3 programas pequenos. Os mesmos programas foram implementados por 80 alunos de graduação do curso de Ciência da Computação da Universidade Federal de Viçosa. Todos os alunos estavam, pelo menos, no terceiro ano de graduação, por isso, alunos seniores. O experimento foi conduzido durante algumas aulas da disciplina Engenharia de Software I e sua aplicação durou aproximadamente 30 dias em cada uma das turmas.

Os programas utilizados para o experimento foram extraídos do livro *A Discipline for Software Engineering* de Humphrey (1995). Os exercícios propostos em Humphrey (1995) para aplicação do PSP são de duas séries, A ou B, sendo que a

série A é composta por 10 exercícios e a série B é composta por 9 exercícios, ambas aumentam a dificuldade de forma progressiva do início para o fim de cada série. Por questões de limitação de tempo e escopo da disciplina, não foi possível aplicar todos os exercícios, tendo sido utilizados os três primeiros programas da série A (1A, 2A e 3A) e também os três primeiros relatórios (R1, R2 e R3) - todos estão disponíveis no Apêndice A, e fazem parte da *baseline* do PSP (descrito na Seção 2.4.1). Pela experiência anterior de aplicação do PSP em sala de aula (o Prof. José Luis Braga já aplica o PSP em sala de aula desde o ano 2002), esses exercícios se mostraram suficientes para incutir nos alunos as principais idéias e práticas do PSP (Balbino, 2004).

A coleta de dados do experimento durou dois anos, sendo que no ano de 2004 foi aplicado o PSP para 40 alunos e no ano de 2005 foi aplicado o PSPm para outros 40 alunos, distintos do primeiro grupo. Vale ressaltar que não havia aluno repetindo a disciplina no ano de 2005.

Todos os alunos foram instruídos pelo mesmo professor (José Luis Braga) e foi utilizado o mesmo material didático disponível em Humphrey (1995). Vale destacar que os alunos de 2005 receberam o mesmo treinamento em PSP que os alunos de 2004, diferindo apenas na fase Projeto, onde foi adicionada a técnica de *Test-First* ao PSP.

O objetivo principal do experimento foi realizar uma análise comparativa entre o PSP e o PSPm com o intuito de verificar

- a injeção de defeitos na fase de projeto e
- o impacto da inserção da técnica de *Test First Design* no PSP

com o foco direcionado para a qualidade final dos produtos de software, mensurada através da diminuição de defeitos injetados na fase de projeto do software.

4.3 Metodologia

Objetos de estudo: PSP e PSPm.
Propósitos: Avaliar se houve impacto na melhoria da qualidade dos programas 1A, 2A e 3A pela redução do número de defeitos injetados na fase de projeto.
Foco da qualidade: Injeção de defeitos na fase de projeto.
Perspectiva: Programadores.

O experimento foi conduzido de forma independente com duas turmas de 40 alunos. As duas turmas eram formadas por alunos seniores do curso de Ciência da Computação da Universidade Federal de Viçosa. A primeira turma, do ano de 2004, utilizou o PSP e a segunda turma, do ano de 2005, utilizou o PSPm, ambas as turmas desenvolveram os mesmos programas. Todos os alunos das duas turmas passaram por um treinamento no PSP e já haviam cursado, anteriormente, disciplinas de programação orientada a objetos com C++.

O experimento foi fundamentado nas seguintes premissas:

- o número médio de defeitos injetados na fase de projeto do PSPm é menor que do PSP;
- o tempo médio gasto na fase de teste do PSPm é menor que o tempo gasto na fase de teste do PSP.

As variáveis chave *independentes* analisadas durante o experimento visando à avaliação das hipóteses foram:

- *Processo usado:* Como previamente descrito, uma turma de 40 alunos (2004) utilizou o PSP e outra turma distinta de 40 alunos (2005) utilizou o PSPm para desenvolvimento dos programas 1A, 2A e 3A de forma individual. O programa 1A foi desenvolvido utilizando nível 0 do PSP e os programas 2A e 3A foram desenvolvidos utilizando o nível 0.1 do PSP. Todos os alunos obtiveram um prazo total de 30 dias para desenvolvimento dos programas, escalonado por tarefa.

- *Experiência dos alunos com a linguagem de programação:* Todos os alunos já haviam desenvolvido programas com a linguagem de programação orientada a objetos C++ em disciplinas cursadas anteriormente.

- *Métodos e técnicas usadas:* Todos os alunos utilizaram o mesmo processo, seguindo as mesmas fases e preenchendo todos os formulários indicados em Humphrey (1995). A única diferença foi que a turma do ano de 2005 utilizou o PSPm, que teve sua fase Projeto modificada com a inserção da técnica *Test-First Design*.

As variáveis chave *dependentes* analisadas durante o experimento visando a avaliação das premissas foram:

- *Defeitos Injetados na fase Projeto:* Durante o desenvolvimento dos programas os programadores registram no formulário *Defect Recording Log* todos os erros injetados e removidos durante todo o processo de desenvolvimento do software, indicando em qual fase o erro foi injetado e em qual fase ele foi removido e se durante a correção de algum erro foi injetado um novo erro. Assim é possível rastrear onde o programador erra mais e qual tipo de erro ele comete.

- *Tempo gasto na fase Teste:* Quando o programador começa a execução do processo, ele começa a contabilizar todo o tempo gasto em cada fase, lançando esses dados no formulário *Time Recording Log*. Através destes dados é possível saber quanto tempo está sendo gasto em cada fase, mantendo um histórico que ajuda o programador a ter uma maior previsibilidade para execução de novos projetos.

Como em todo experimento, existem fontes de variação não controláveis. No presente estudo podemos citar como exemplo a inacurácia dos alunos em registrar tempo e defeitos, a possibilidade de que alguns alunos tenham copiado trabalhos de outros, a interação aluno x programa, isto é, uma tarefa considerada difícil para um aluno pode ser considerada fácil para outro.

4.3.1 Treinamento dos Alunos

No início da disciplina Engenharia de Software I foram abordados temas relevantes para a compreensão do que representa o termo Engenharia de Software, tais como qualidade de software, processos de desenvolvimento de software, modelos para avaliação da maturidade organizacional no processo de engenharia de software, dentre outros.

Após a exposição destes conceitos relacionados a Engenharia de Software, foi introduzido o PSP. Os alunos assistiram um total de oito aulas de 1 hora e 50 minutos cada, sendo que eram dadas duas aulas por semana, o que dá um total de duas semanas de exposição do conteúdo do PSP.

No ano de 2005, os alunos foram treinados no mesmo rítmo - abordagem em sala de aula e tempo para entrega dos trabalhos, coletando os dados em planilhas iguais e utilizando o mesmo material didático, diferenciando apenas a fase Projeto, onde foram introduzidos os conceitos apresentados no PSPm.

Após a conceitualização do que é o PSP, os alunos foram treinados para começarem a execução da série A de exercícios propostos por Humphrey (1995).

4.3.2 Coleta dos Dados

Para execução do experimento, foi adotado a estrutura de treinamento proposta no livro "*A Discipline for Software Engineering*"(Humphrey, 1995). Como o propósito do treinamento não era deixar os alunos "*experts*" em PSP, foi utlizado apenas o material necessário para os níveis PSP0 e PSP0.1. A estrutura utilizada é mostrada na Tabela 4.1.

Conforme mostrado na Tabela 4.1, o primeiro exercício exigido foi o programa 1A. Este exercício é o primeiro da série e serve para os programadores terem um referencial de como programam. Mesmo contendo os roteiros de execução do processo, a maioria dos alunos encontram dificuldades na hora de registrar o tempo gasto em cada fase, o que fazer em cada fase, quais os limites de cada fase, entre outros. Isso se deve ao fato dos alunos não terem consciência do que é um processo de desenvolvimento de

Tabela 4.1: Estrutura dos trabalhos

Tópico	Exercício	Descrição do Exercício	Nível
Estratégia do PSP	1A	Calcular a média e o desvio padrão de N números reais	PSP0
Planejando o Processo	2A	Contar linhas de código de um programa fonte	PSP0.1
	R1	Produzir um padrão para contagem de LOC[1]	
	R2	Produzir um padrão de codificação	
Medindo o tamanho do Software	3A	Aumentar o programa 2A para contar LOC de objetos	PSP0.1
	R3	Relatório de Análise de Defeitos	

Fonte: Adaptado de Humphrey (1995)

[1] - do inglês - *Line of Code* - Linhas de Código

software e não terem o hábito de projetar o software e muito menos registrar o tempo gasto no desenvolvimento. Para realização do exercício 1A, os alunos utilizaram o PSP0.0.

Após a entrega do exercício 1A, foram esclarecidas todas as dúvidas dos alunos relacionadas a execução do processo e feito uma revisão.

O segundo e o terceiro exercícios foram os relatórios R1 e R2, os quais consistem na especificação do padrão de codificação e de contagem de linhas de código.

Os próximos exercícios realizados, 2A e 3A, utilizaram o PSP0.1. A execução desses programas não gerou dúvidas referentes ao processo, por parte dos alunos.

Para encerrar, os alunos escreveram o relatório R3, que consiste em analisar os defeitos encontrados no desenvolvimento dos programas.

O experimento foi executado em 30 dias, a contar da primeira aula até o dia de entregar todos os exercícios resolvidos. Durante o desenvolvimento dos programas

Data	Start	Stop	Interruption Time	Delta Time	Phase	Comments
28/3/04	9:21	9:26		5,00	planejamento	
28/3/04	9:36	10:20		44,00	projeto	
28/3/04	10:21	11:20	5	54,00	codificacao	ida ao banheiro
28/3/04	11:21	11:36		15,00	compilacao	
28/3/04	11:39	13:15	70	26,00	testes	parada para almoco
28/3/04	13:23	13:38		15,00	post-mortem	

Fonte: Adaptado de Humphrey (1995)

Figura 4.1: Exemplo da planilha *Time Recording Log*

1A, 2A e 3A os alunos coletaram seus próprios dados e os armazenaram em planilhas eletrônicas padronizadas, que foram desenvolvidas seguindo os modelos propostos por Humphrey (1995). Essas planilhas continham os seguintes formulários:

- *Time Recording Log:* Responsável pela coleta do tempo gasto em cada fase do projeto, sendo registrados também todas as interrupções sofridas durante o processo. O tempo delta é calculado com base no tempo total subtraído do tempo de interrupções, sendo portanto o tempo real gasto em cada fase, conforme mostra a Figura 4.1;

- *Defect Recording Log:* Reponsável pela coleta de todos os defeitos injetados e removidos durante o processo, conforme mostra a Figura 4.2;

- *Project Plan Summary:* É preenchido na última fase do processo - *PostMortem*, onde é feito um resumo de todos os dados das duas planilhas anteriores, mantendo um histórico dos dados de todos os programas desenvolvidos até a presente data, conforme mostra a Figura 4.3.

Como primeiro passo da análise, todos os dados coletados pelos alunos foram lançados em uma planilha eletrônica única, com o intuito de facilitar a análise. Nesta fase, que precede a análise por métodos estatísticos, foi realizada uma análise criteriosa dos dados e optou-se por não utilizar os dados do exercício 1A, pois julgou-se que eles não condiziam com a realidade de competência dos alunos. O principal motivo é que no exercício 1A os alunos sentem muita dificuldade na execução e na coleta dos dados referentes ao processo, por ser o primeiro contato com o PSP. Dessa forma, optou-se por utilizar na análise estatística apenas os dados referentes aos exercícios 2A e 3A.

Date	Number	Type	Inject	Remove	Fix Time	Fix Defect
03/28/04	1	20	codificacao	compilacao	1	
Description:	Esqueci de colocar os parenteses na frente da declaracao do cabecalho do construtor da classe					
Date	Number	Type	Inject	Remove	Fix Time	Fix Defect
28/3/2004	2	20	codificacao	compilacao	1	
Description:	Erro na digitacao do identificador do objeto					
Date	Number	Type	Inject	Remove	Fix Time	Fix Defect
28/3/2004	3	80	codificacao	compilacao	5	
Description:	Esqueci de colocar alguns comando dentro do bloco try e capturar as excessoes.					
Date	Number	Type	Inject	Remove	Fix Time	Fix Defect
28/3/2004	4	80	codificacao	testes	2	
Description:	Erro no laco while causou um looping infinito					
Date	Number	Type	Inject	Remove	Fix Time	Fix Defect
28/3/2004	5	80	codificacao	testes	5	
Description:	Erro no laco for causou um looping infinito					

Fonte: Adaptado de Humphrey (1995)

Figura 4.2: Exemplo da planilha *Defect Recording Log*

4.4 Análise e Interpretação

4.4.1 Notação e Definições

Seja Y_{ijk} o valor da variável resposta obtido na k-ésima repetição da j-ésima fase com o i-ésimo método, para $k = 1, 2, \ldots, 40$; $j = 1, 2, 3$ e $i = 1$ para PSP e $i = 2$ para PSPm. Seja Δ_j a seguinte variável aleatória,

$$\Delta_j = \left| \overline{Y}_{1j.} - \overline{Y}_{2j.} \right|$$

em que $\overline{Y}_{1j.}$ e $\overline{Y}_{2j.}$ denotam as médias aritméticas das amostras para os métodos 1 (PSP) e 2 (PSPm) na j-ésima fase respectivamente e $|x|$ indica o valor absoluto ou módulo de x. No presente trabalho tomou-se Δ_j como sendo a estatística de interesse para se concluir quanto à rejeição ou não rejeição das hipóteses apresentadas a seguir.

Summary	Plan	Actual	To Date		
Minutes/LOC			0		
LOC/Hour	68,2	35,5	0,377659574		
Defects/KLOC	100	85,1	0,905319149		
Yield			0		
A/FR			0		
Program Size (LOC):					
Total New & Changed	100	94	1,282978723		
Maximum Size					
Minimum Size					
Time in Phase(min.)	Plan	Actual	To Date	To Date%	
Planning	3	5	6	3,28	
Design	25	44	47	25,50	
Code	25	54	68	37,00	
Code Review	0	0	0	0,00	
Compile	10	15	17	9,20	
Test	20	26	29	15,82	
Postmortem	5	15	17	9,20	
Total	88	159	184	100,00	
Maximum Time					
Minimum Time					
Defects Injected	Plan	Actual	To Date	To Date%	**Def/.Hour**
Planning				0,00	
Design	5	1	2	20,00	0,65
Code	5	7	8	80,00	2,6
Code Review				0,00	
Compile				0,00	
Test				0,00	
Total Development	10	8	10	100,00	
Defects Removed	Plan	Actual	To Date	To Date%	**Def/.Hour**
Planning				0,00	
Design				0,00	
Code			1	10,00	
Code Review				0,00	0
Compile	5	5	6	60,00	2
Test	5	3	3	30,00	1
Total Development	10	8	10	100,00	
After Development					

Fonte: Adaptado de Humphrey (1995)

Figura 4.3: Exemplo da planilha *Project Plan Summary*

4.4.2 Hipóteses Estatísticas

O objetivo da análise é testar a hipótese de igualdade entre as médias, $H_0 : \mu_{PSP,j} = \mu_{PSPm,j}$. Esta hipótese deve ser interpretada como "os métodos PSP e PSPm não diferem entre si quanto aos valores médios obtidos em qualquer uma das j-ésimas fases de avaliação". Ou seja, as modificações propostas no PSP não alteram significativamente os valores médios. No contexto de uma análise estatística de hipótese, μ denota a verdadeira média de uma população hipotética de usuários e o teste de H_0 é conduzido com base na diferença entre as médias amostrais $\Delta_j = \overline{Y}_{PSP,j} - \overline{Y}_{PSPm,j}$. Por simplificação denotaremos a hipótese como H_0 :PSP=PSPm.

Em um teste de hipótese usual há uma estatística do teste que possui uma distribuição amostral conhecida sob a pressuposição de que a hipótese H_0 é verdadeira. Portanto pode-se inferir com o valor da estatística observado com base em um nível de significância (designado α) previamente estabelecido. Define-se p-valor (ou nível descritivo) como sendo a probabilidade de se obter um valor igual ou mais extremo do que o observado, em favor da hipótese alternativa. A hipótese alternativa, geralmente designada H_1, é a hipótese que contraria a hipótese H_0. No presente trabalho foram consideradas hipóteses H_1 unilaterais conforme explicado adiante no texto. A decisão do teste é com base no p-valor, sendo um p-valor *baixo* uma evidência contrária à hipótese H_0 ou favorável a H_1. Define-se como p-valor *baixo* quando ele é menor ou igual ao nível de significância (α) adotado. Os níveis de significância mais utilizados pela comunidade científica são: $\alpha = 0,01$, $\alpha = 0,05$ e $\alpha = 0,10$. Portanto a decisão estatística em um teste de hipótese é da seguinte forma:

$$\text{p-valor} \leq \alpha \Longrightarrow \text{rejeita-se } H_0 \text{ com um nível de significância } \alpha$$

Neste caso, as hipóteses alternativas, que foram extraídas com base no diagrama de influência apresentado na Figura 3.2, têm o objetivo de validar as vantagens do PSPm em relação ao PSP, citadas no Capítulo 3. Um resumo das hipóteses alternativas (H_1) consideradas nos testes da hipótese H_0 : PSP=PSPm é apresentado na Tabela 4.2.

Tabela 4.2: Variáveis resposta e respectivas hipóteses alternativas (H_1).

Variável resposta	Fase avaliada	Hipótese alternativa
Defeitos Injetados	Projeto	H_1:PSP > PSPm
Tempo	Teste	H_1:PSP > PSPm

Fonte: Elaborado pelo Autor

Ainda de acordo com a Tabela 4.2, a hipótese de o PSPm injetar menor número de erros na fase Projeto, indica um melhor entendimento dos requisitos e conseqüentemente uma maior qualidade introduzida no software.

4.4.3 *Bootstrap* - Teste por Reamostragem

No presente trabalho, para o número de defeitos injetados, podemos afirmar que a distribuição amostral de Δ_j é desconhecida. Esta afirmação é devida ao fato de as amostras de valores Y_{ijk} obtidos para a variável Defeitos Injetados que contêm muitos valores iguais a zero ou um e no máximo o valor 5. Portanto, assumir que estes valores são oriundos de uma distribuição normal não é uma decisão coerente. Como consequência, a utilização de técnicas estatísticas usuais como o teste t para comparação de médias também não parece ser uma boa alternativa. Outra alternativa é a utilização de testes não-paramétricos, como o teste Mann-Whitney (Conover, 1980), entretanto, é amplamente conhecido que os procedimentos não paramétricos possuem menor poder do que os equivalentes paramétricos. O poder de um teste é a probabilidade de ele rejeitar uma hipótese H_0 falsa.

O Teste por Reamostragem aqui empregado é uma alternativa para se estimar o p-valor associado ao Δ_j observado na pesquisa. O objetivo é gerar uma amostra aleatória de valores Δ_j obtidos por reamostragem com reposição da amostra original, metodologia designada no meio estatístico como *bootstrap*. Duas referências clássicas

que abordam as inúmeras alternativas de aplicações são Efron and Tibshirani (1993) e Davison and Hinkley (1997).

O procedimento exige recursos computacionais e pode ser descrito resumidamente nas seguintes etapas:

1. Os valores amostrais obtidos pelos dois métodos são agrupados de modo que uma única amostra de valores é definida. Denote esta amostra como sendo o vetor $\mathbf{Y}$.

2. Os números do vetor $\mathbf{Y}$ são aleatoriamente reembaralhados ou realocados em outras posições e em seguida duas amostras aleatórias com reposição são obtidas: a primeira constitui uma nova amostra de valores para o método PSP e a média desta amostra é designada $\overline{Y^*}_{1,b}$; analogamente, a segunda é para o método PSPm e designada $\overline{Y^*}_{2,b}$.

3. O valor $\Delta_{b,j} = \overline{Y^*}_{1,b} - \overline{Y^*}_{2,b}$ é calculado utilizando-se os valores obtidos por reamostragem conforme descrito em (2) e designado por $\Delta^*_{b,j}$ para denotar que foi obtido por reamostragem dos valores originais.

4. Repetir os procedimentos descritos em (2) e (3) B vezes de modo que $b = 1, \ldots, B$. Foi adotado no presente trabalho $B = 10.000$.

5. Os 10.000 valores $\Delta^*_{b,j}$ gerados por reamostragem são utilizados para se estimar os p-valores e intervalos de confiança $(1-\alpha)100\%$ associados aos Δ_j's observados na pesquisa.

4.4.4 P-valores e Intervalos de Confiança

Os p-valores estimado por reamostragem ou *bootstrap* são obtidos por,

1. Para $H_1 : \mu_{1j.} > \mu_{2j.}$ ou seja PSP > PSPm,

$$\widehat{p} = \frac{\sum_{b=1}^{B} I\left(\Delta^*_{b,j} \geq \Delta_j\right)}{B},$$

2. para $H_1 : \mu_{1j.} < \mu_{2j.}$ ou seja PSP $<$ PSPm,

$$\widehat{p} = \frac{\sum_{b=1}^{B} I\left(\Delta_{b,j}^{*} \leq -\Delta_j\right)}{B},$$

em que $I(\cdot)$ é uma função indicadora, isto é, $I(x \in A) = 1$ se $x \in A$ ou $I(x \in A) = 0$ se $x \notin A$ e Δ_j é o valor observado na pesquisa ou obtido com a amostra original. O p-valor estimado desta forma é também denominado nível de significância observado ou α-observado, pois é exatamente a proporção de valores *bootstrap* observados que são mais extremos do que o valor obtido na amostra.

Vale salientar que se a hipótese $H_1 : \mu_{1j.} > \mu_{2j.}$ é lançada, então espera-se que $\overline{Y}_{1j.} - \overline{Y}_{2j.} \geq 0$ e portanto um valor $\overline{Y}_{1j.} - \overline{Y}_{2j.} < 0$ descartaria a necessidade de se testar H_0 pois neste caso não haveria nenhuma evidência contrária a H_0.

Valores **inf** e **sup** que formam intervalos (**inf**; **sup**) com $(1-\alpha)100\%$ de confiança para Δ_j foram obtidos de modo que para $b = 1, 2, \ldots, B$ observou-se $(\alpha/2)\,100\%$ dos valores $\Delta_{b,j}^{*}$ satisfazendo,

$$\Delta_{b,j}^{*} \leq \textbf{inf} \quad \text{e} \quad \Delta_{b,j}^{*} \geq \textbf{sup}$$

Adotou-se $\alpha = 0,10$ no presente trabalho e portanto os intervalos *bootstrap* foram obtidos de modo que 90% dos valores $\Delta_{b,j}^{*}$ observados estavam contidos no intervalo (**inf**; **sup**). Este método é conhecido como o método dos percentis pois **inf** e **sup** são simplesmente percentis da distribuição amostral gerada por reamostragem, então (**inf**; **sup**)=(5^0; 95^0) percentis. Segundo Walters and Capbell (2004), embora este seja um método válido, pode resultar em intervalos viesados.

4.4.5 Resultados e Discussão

Os testes estatísticos foram realizados utilizando-se o programa *Resampling Stats* em sua versão *trial* (Stats, 2005). No Apêndice C são apresentados alguns scripts utilizados para verificação das hipóteses.

Um resumo dos resultados das análises estatísticas é apresentado a seguir. São apresentados os valores médios observados para as variáveis indicadas na Tabela 4.2, onde são indicadas as fases analisadas com os programas 2A e 3A, com os respectivos p-valores e intervalos de confiança (I.C.), obtidos por reamostragem, referentes ao teste da hipótese de igualdade entre médias.

Tabela 4.3: Resumo dos resultados das análises para a variável defeitos injetados com PSP e PSPm.

	Média				
Exercício	PSP	PSPm	Δ_j	p-valor[/1]	I.C.[/2]
PSP2A	0,48	0,10	0,38	0,045	(-0,34 ; 0,34)
PSP3A	0,50	0,14	0,36	0,058	(-0,35 ; 0,35)

Fonte: Dados da pesquisa

[/1] - A hipótese H_0 : PSP = PSPm é rejeitada com nível de significância α quando p-valor $\leq \alpha$.

[/2] - Para a diferença entre médias (Δ_j).

Os resultados apresentados na Tabela 4.3, permitem concluir que as diferenças entre os valores médios de defeitos injetados, entre PSP e PSPm, podem ser declaradas como significativas a 5% para o exercício 2A e a 10% para o exercício 3A, sendo que, nos dois exercícios, o PSPm resultou em menor média de defeitos injetados. Portanto, esses resultados são a favor da hipótese H_1:PSP>PSPm, o que é um indício de que houve uma melhoria do processo, através da diminuição da quantidade de defeitos injetados na fase Projeto do PSPm.

Os resultados apresentados na Tabela 4.4 sugerem que as diferenças entre os valores médios de tempo, entre PSP e PSPm, podem ser declaradas como significativas a 5% apenas para o exercício 2A, entretanto, o PSPm resultou em menor tempo médio para ambos exercícios. Esses resultados também são favoráveis à hipótese H_1:PSP>PSPm referente à variável de resposta Tempo para a fase Teste mostradas

Tabela 4.4: Resumo dos resultados das análises para a variável tempo com PSP e PSPm.

	Média				
Exercício	PSP	PSPm	Δ_j	p-valor[/1]	I.C.[/2]
PSP2A	23,21	13,86	9,35	0,0379	(-8,62 ; 8,59)
PSP3A	31,89	26,14	5,75	0,28	(-17,23 ; 17,33)

Fonte: Dados da pesquisa

[/1] - A hipótese H_0 : PSP = PSPm é rejeitada com nível de significância α quando p-valor $\leq \alpha$.
[/2] - Para a diferença entre médias (Δ_j).

na Tabela 4.2.

Para realizar um estudo mais aprofundado que permitisse concluir se o tempo gasto na fase Teste do PSPm é realmente menor que o tempo gasto no PSP, seria necessário que a análise fosse feita de forma pareada, isto é, diferenças de tempo calculadas para o mesmo indivíduo. Essa alteração permitiria comparar valores de tempo entre a execução de dois processos diferentes com os mesmos indivíduos.

A segunda parte da análise consistiu em segmentar os dados coletados em duas partes:

i) alunos com CA $\geq$ 80: esse segmento é composto por alunos considerados bons e alunos ótimos, pelo padrão da UFV definido no Catálogo de Graduação
80 – 90: Alunos bons
90 – 100: Alunos ótimos

ii) alunos com CA $<$ 80: já este segmento é composto por alunos regulares, medianos e bons pelo padrão UFV
60 – 70: Alunos regulares
70 – 79: Alunos medianos e bons

Entende-se por CA o coeficiente de rendimento acumulado, que é o indicador adotado pela Universidade Federal de Viçosa como forma de avaliar o desempenho dos alunos. O CA é obtido pelo cálculo da média ponderada das notas obtidas pelos alunos nas disciplinas cursadas. Após segmentados os dados, foram realizados os mesmos procedimentos estatísticos utilizados na primeira parte da análise, e que são mostrados nas seções subsequentes.

Segundo a literatura na área pedagógica, estudantes aprendem em velocidade diferente, em horários diferentes, e de forma diferente. Em uma sala de aula, não há uniformidade nesses aspectos, nem que a turma seja muito homogênea. A preparação de uma disciplina requer atenção para essa verdade inquestionável, para que os objetivos de aprendizado sejam atingidos. Segundo Wankat (2002), o que funciona de fato para aumentar o nível de aprendizado é **envolver os estudantes nos temas das aulas**, o que leva aos seguintes aspectos a serem considerados:

- certificar que os estudantes estão processando de maneira ativa o material das aulas;
- dar aos estudantes algum controle sobre o que e como estão aprendendo;
- utilizar a prática para fixação de conceitos;
- encorajar e auxiliar na reflexão sobre o que está sendo aprendido, e como esse aprendizado está influenciando sua forma de trabalhar o assunto;
- propor desafios aos estudantes, dando a eles oportunidade de terem sucesso para enfrentá-los;
- encorajar troca de experiências e permitir que um estudante ensine a outro;
- encorajar o uso do tempo disponível com foco exclusivo nas tarefas a serem executadas.

A decisão de considerar os dados coletados nos experimentos para dois grupos de alunos, os com CA ≥ 80 e os com $60 \leq$ CA < 80 se baseia nessas constatações experimentais da área pedagógica. Embora esses não devam ser os únicos critérios

para segmentar as turmas, para os objetivos deste trabalho eles são suficientes. O tipo de aprendizado incentivado pelo PSP é fortemente influenciado pelas técnicas de PBL (*Problem Based Learning*), atendendo a vários dos aspectos enumerados (Wankat, 2002).

4.4.6 Alunos com CA $\geq$ 80

Um resumo dos resultados das análises estatísticas utilizando os dados coletados pelos alunos com CA $\geq$ 80 é apresentado a seguir. São apresentados os valores médios observados para as variáveis indicadas nas hipóteses mostradas na Tabela 4.2 com o PSP e PSPm e suas respectivas fases analisadas com os programas 2A e 3A, com seus respectivos p-valores e intervalos de confiança (I.C.), obtidos por reamostragem e referentes ao teste da hipótese de igualdade entre médias.

Tabela 4.5: Resumo dos resultados das análises para a variável defeitos injetados com PSP e PSPm - CA $\geq$ 80.

	Média				
Exercício	PSP	PSPm	Δ_j	p-valor[/1]	I.C.[/2]
PSP2A	0,43	0,14	0,29	0,24	(-0,57 ; 0,57)
PSP3A	0,29	0,00	0,29	0,27	(-0,57 ; 0,57)

Fonte: Dados da pesquisa

[/1] - A hipótese H_0 : PSP = PSPm é rejeitada com nível de significância α quando p-valor $\leq \alpha$.

[/2] - Para a diferença entre médias (Δ_j).

Os resultados apresentados na Tabela 4.5 não permitem concluir que as diferenças entre os valores médios de defeitos injetados, entre PSP e PSPm, podem ser declaradas como significativas a 10%, mesmo com o PSPm resultando em menor média de defeitos injetados na fase Projeto. Portanto, esses resultados não são a favor

da hipótese H_1:PSP>PSPm referente à variável de resposta Defeitos Injetados.

Tabela 4.6: Resumo dos resultados das análises para a variável tempo com PSP e PSPm - CA $\geq$ 80.

	Média				
Exercícios	PSP	PSPm	Δ_j	p-valor[/1]	I.C.[/2]
PSP2A	16,286	15,42	0,857	0,427	(-7,00 ; 7,14)
PSP3A	48,28	40,42	7,85	0,40	(-54,00 ; 53,71)

Fonte: Dados da pesquisa

[/1] - A hipótese H_0 : PSP = PSPm é rejeitada com nível de significância α quando p-valor $\leq \alpha$.
[/2] - Para a diferença entre médias (Δ_j).

Os resultados apresentados na Tabela 4.6 sugerem que as diferenças entre os valores médios do tempo, entre PSP e PSPm, não podem ser declaradas como significativas a 10% para os exercícios 2A e 3A, mesmo com o PSPm resultando em menor tempo médio. Esses resultados não são favoráveis à hipótese H_1:PSP>PSPm referente à variável resposta Tempo para a fase Teste mostradas na Tabela 4.2.

4.4.7 Alunos com CA<80

Um resumo dos resultados das análises estatísticas utilizando os dados coletados pelos alunos com CA $<$ 80 é mostrado a seguir, onde são apresentados os valores médios observados para as variáveis indicadas nas hipóteses mostradas na Tabela 4.2 com o PSP e PSPm e suas respectivas fases analisadas com os programas 2A e 3A, com seus respectivos p-valores e intervalos de confiança (I.C.), obtidos por reamostragem e referentes ao teste da hipótese de igualdade entre médias.

Os resultados apresentados na Tabela 4.7 permitem concluir que as diferenças entre os valores médios de defeitos injetados, entre PSP e PSPm, podem ser declaradas

Tabela 4.7: Resumo dos resultados das análises para a variável defeitos injetados com PSP e PSPm - CA<80.

	Média				
Exercício	PSP	PSPm	Δ_j	p-valor[/1]	I.C.[/2]
PSP2A	0,45	0,00	0,45	0,032	(-0,36 ; 0,36)
PSP3A	0,63	0,00	0,63	0,060	(-0,63 ; 0,63)

Fonte: Dados da pesquisa

[/1] - A hipótese H_0 : PSP = PSPm é rejeitada com nível de significância α quando p-valor $\leq \alpha$.

[/2] - Para a diferença entre médias (Δ_j).

como significativas a 5% para o exercício 2A e a 10% para o exercício 3A, sendo que o PSPm resultou em menor média e, neste caso, nenhum defeito injetado. Portanto, esses resultados são a favor da hipótese H_1:PSP>PSPm referente à variável resposta Defeitos Injetados para a fase Projeto mostrada na Tabela 4.2.

Tabela 4.8: Resumo dos resultados das análises para a variável tempo com PSP e PSPm - CA<80.

	Média				
Exercício	PSP	PSPm	Δ_j	p-valor[/1]	I.C.[/2]
PSP2A	30,00	11,72	18,27	0,05	(-18,818 ; 18,364)
PSP3A	27,27	19,63	7,63	0,22	(-17,545 ; 17,545)

Fonte: Dados da pesquisa

[/1] - A hipótese H_0 : PSP = PSPm é rejeitada com nível de significância α quando p-valor $\leq \alpha$.

[/2] - Para a diferença entre médias (Δ_j).

Os resultados apresentados na Tabela 4.8 sugerem que as diferenças entre os valores médios do tempo, entre PSP e PSPm, podem ser declaradas como significativas a 5% para o programa 2A, mas não para o programa 3A. Mesmo assim, o PSPm resultou em menor tempo médio para ambos os programas. Os resultados para o programa 2A são favoráveis à hipótese H_1:PSP>PSPm referente à variável resposta Tempo para a fase Teste mostrada na Tabela 4.2.

4.5 Considerações Finais

No geral os valores médios observados e os resultados dos testes estatísticos realizados indicam que houve redução dos defeitos injetados pelos alunos que utilizaram o PSPm em relação aos alunos que utilizaram o PSP.

Entre os alunos com CA ≥ 80, apesar de os testes estatísticos não indicarem a validação da hipótese alternativa H_1, o valor médio de defeitos injetados foi menor. Isso pode ser explicado pelo fato dos alunos serem bons o suficiente para desenvolver um bom trabalho indiferente do processo utilizado.

Já entre os alunos de pior desempenho acadêmico (CA < 80) o PSPm se mostrou promissor, dando resultados satisfatórios à hipótese alternativa H_1, indicando que houve diminuição dos defeitos injetados.

Em relação ao tempo gasto na fase de testes, apesar dos testes estatísticos não serem em sua maioria favoráveis à hipótese alternativa H_1, o PSPm apresentou um menor valor médio para todos os casos observados.

Todo o procedimento necessário para repetição do experimento está apresentado no Apêndice D.

4.5.1 PSPm e MPS.BR

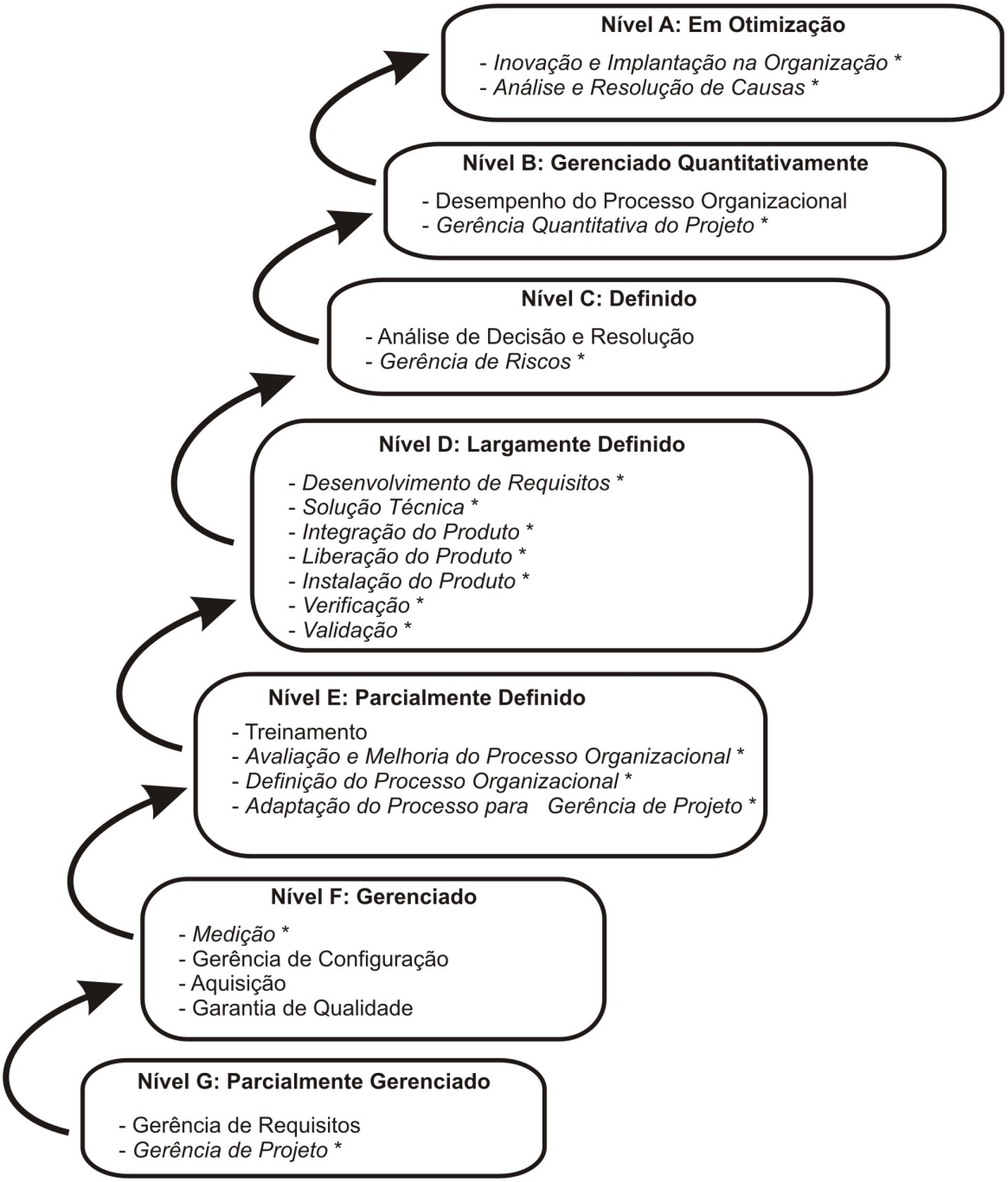
Como dito na Seção 2.4.2, o PSP foi desenvolvido pelo mesmo autor do CMM, com o intuito de simplificar a aplicação do CMM e proporcionar às pequenas e médias organizações a oportunidade de melhorarem a qualidade de seus produtos através da disciplina e previsibilidade. O PSPm é uma versão adaptada do PSP que proporciona

aos programadores uma menor média de injeção de defeitos na fase de projeto do software, o que contribui para a melhoria da qualidade do software.

O MPS.BR foi criado como iniciativa do governo brasileiro para ajudar as pequenas e médias organizações produtoras de software a melhorar a qualidade de seus produtos, com o intuito de desenvolver softwares com qualidade de exportação e fortalecer o mercado interno brasileiro. Conforme mostrado na Seção 2.3.4, o MPS.BR combina as melhores práticas e idéias dos modelos de avaliação internacionais, tais como o CMM/CMMI e as normas ISO. Isso o torna uma arma poderosa a favor da qualidade das empresas nacionais.

Enquanto o PSPm objetiva a melhoria do desenvolvimento de software a nível pessoal, o MPS.BR se concentra na melhoria da capacidade organizacional. Com o intuito de auxiliar essas organizações a serem avaliadas pelo MPS.BR, o PSPm pode ser uma alternativa para iniciar esse processo de avaliação, pois combina tudo que o PSP possui com uma das melhores práticas apresentadas pela própria indústria de software, que é o TDD.

Fazendo um mapeamento do PSPm para o MPS.BR, semelhante ao mapeamento existente entre o CMM e o PSP mostrado na Seção 2.4.2, o PSPm permite atender, pelo menos de forma parcial, a dezesseis Áreas de Processo (AP) do MPS.BR, conforme mostra a Figura 4.4, onde as APs em *itálico* e com um * representam as APs cobertas. Isso deixa clara a estreita relação existente entre eles.



Fonte: Elaborado pelo Autor a partir de dados de Paulk et al. (1993); Humphrey (1995); SEI (2002); SOFTEX (2005); Silva et al. (2006)

Figura 4.4: Áreas de Processo do MPS.BR cobertas pelo PSPm

Capítulo 5

Conclusões

Alguns dos problemas com a qualidade final de produtos de software derivam do fato de se confiar ou delegar aos testes finais a responsabilidade da qualidade do software. Conforme comprovado na literatura, detectar e remover defeitos a partir dos testes é mais caro, não efetivo e imprevisível (Hayes and Over, 1997; Humphrey, 1995). A forma mais efetiva de gerenciar a qualidade do software é focar na remoção de defeitos o quanto antes, se possível no mesmo passo de desenvolvimento em que foram cometidos.

Os estudos, experiências e o amadurecimento vivenciadas no decorrer deste trabalho evidenciaram essa questão, tomando a melhoria da competência e padrões pessoais de qualidade como pontos de partida fundamentais para atingir a melhoria da qualidade no desenvolvimento de software.

Com o intuito de alcançar todos os objetivos propostos na Seção 1.2, foram realizadas várias etapas neste trabalho, conforme descrito nos parágrafos abaixo.

O primeiro objetivo proposto foi promover uma combinação de dois processos de desenvolvimento de software de origens diferentes, ambos tendo como foco principal o desenvolvimento de software no nível de competência individual. Os dois processos objeto de estudos, o PSP e o TDD, são aplicáveis diretamente pelos programadores em seu dia a dia. A interseção do PSP com o TDD gerou um processo modificado tendo como base o PSP, chamado de PSPm, conforme apresentado no Capítulo 3, que pode ser considerado um processo híbrido integrando partes de Métodos Ágeis com

Métodos Baseados em Planejamento.

O segundo objetivo proposto para este trabalho foi a aplicação do PSPm em sala de aula. O PSPm foi aplicado por quarenta alunos do curso de Ciência da Computação da Universidade Federal de Viçosa que cursavam a disciplina Engenharia de Software I, conforme pode ser observado todo o processo no Capítulo 4.

Buscando alcançar o próximo objetivo, que é a comparação dos dados dos alunos que utilizaram o PSP com dados dos alunos que utilizaram o PSPm, todos os dados coletados foram lançados em planilhas apropriadas, conforme descrito na Seção 4.3.2, e foi realizado o cálculo das médias para todos os valores.

Comparando os valores médios para defeitos injetados no projeto do PSP e do PSPm, verificou-se que o PSPm resultou em um menor valor médio de defeitos injetados que o PSP. Também verificou-se que o tempo médio gasto na fase de teste dos programas foi reduzido pelo uso do PSPm. Dessa forma, mais dois objetivos propostos na Seção 1.2 foram alcançados.

Como o próximo objetivo foi validar os resultados com métodos estatísticos, foram criadas hipóteses alternativas (Seção 4.4.2), e foram realizados Testes por Reamostragem *bootstrap*, conforme apresentado na Seção 4.4.3, com o intuito de validar essas hipóteses. Os resultados dos testes estatísticos apresentados na Seção 4.4.5 permitem concluir que houve uma melhora significativa na redução de defeitos injetados na hora de se projetar os softwares.

Ainda na Seção 4.4.5 são apresentados os resultados referentes a análise dos dados de todos os alunos e uma análise segmentada pelos Coeficientes de Rendimento Acumulado (CA), onde a principal melhoria é notada nos alunos que apresentam um pior rendimento acadêmico.

A análise estatística dos dados de todos os alunos referente a defeitos injetados no projeto dos softwares elaborados, indica que o PSPm leva vantagem no combate à injeção de defeitos. Da mesma forma, a análise estatística dos dados dos alunos com pior rendimento acadêmico indica que o PSPm é mais eficaz[1] na elaboração de

[1]Que produz o efeito desejado; que dá bom resultado. Neste caso, o efeito desejado é a não inserção de defeitos no projeto do software.

projeto de softwares que o PSP.

Já entre os alunos que possuem um melhor rendimento acadêmico, o PSPm não apresentou melhorias significativas, de acordo com o teste estatístico usado, apesar de apresentar em todos os casos, um menor valor médio de defeitos injetados e um menor tempo médio gasto na fase de teste. Isso pode ser atribuído ao fato de esses alunos serem bons o suficiente para realizarem um bom trabalho indiferente do processo que está sendo utilizado.

Neste contexto, os resultados permitem concluir que o PSPm apresenta vantagens sobre o PSP, do ponto de vista de garantia de qualidade. O próprio Humphrey afirma que defeitos de projeto são difíceis de serem reduzidos e que o PSP direciona a prevenção de defeitos no projeto do ponto de vista da evolução, onde existe uma grande probabilidade de o programador se tornar mais cuidadoso com o aumento de sua consciência ao construir o código, mas que o PSP não especifica um método para o projeto, dizendo apenas que existe a necessidade de melhorar a qualidade dos projetos de software (Humphrey, 1995).

O PSPm, além de especificar uma técnica para o projeto do software, no estudo de caso aqui descrito, ele se mostrou mais eficaz na prevenção de defeitos, porque faz com que os programadores enxerguem melhor o escopo do problema a ser resolvido.

Um ponto digno de ser notado é que o PSPm atua nas fases de processo onde, reconhecidamente, a maior parte dos erros são introduzidos, que são as fases iniciais de entendimento de requisitos do problema e sua descrição. Vale ressaltar que aqui não foi definida nenhuma "bala de prata" para a solução do problema da engenharia de software, mas sim, uma forma de amenizar o problema de injeção de erros no software por parte dos programadores.

Segundo Boehm and Turner (2004) Métodos Dirigidos por Planejamento e Métodos Ágeis não oferecem individualmente uma solução definitiva para atacar qualquer tipo de problema em qualquer contexto. Métodos Híbridos tendem a ser mais adequados como alternativa para resolver uma maior gama de problemas.

Finalizando, o PSPm reduz a injeção de defeitos de teste e ajuda o desenvolvedor a encontrar e remover defeitos onde eles são injetados, antes da compilação e

dos testes. Com poucos defeitos para remover nos testes os custos são reduzidos, os cronogramas são mais previsíveis, gastos com manutenção são reduzidos e a satisfação do usuário será maior (Hayes and Over, 1997).

5.1 Trabalhos Futuros

Este trabalho foi realizado no ambiente acadêmico, utilizando alunos da Universidade Federal de Viçosa do curso de Ciência da Computação que estavam cursando a disciplina Engenharia de Software I. Devido a restrições de tempo, o experimento consistiu apenas em parte do treinamento completo proposto por Humphrey (1995), utilizando os exercícios 1A, 2A e 3A como base da experimentação. Uma proposta de trabalho futuro é que seja realizado o mesmo experimento utilizando todos os exercícios de uma das séries A ou B propostas em Humphrey (1995).

Uma outra proposta é sair do meio acadêmico e realizar esse experimento com empresas produtoras de software, verificando o impacto de adoção do PSPm na indústria.

Apêndice A

Exercícios do PSP - Série A

Aqui são apresentados os exercícios que foram aplicados em sala de aula para os alunos da disciplina de Engenharia de Software I do curso de Ciência da Computação da Universidade Federal de Viçosa, cuja disciplina é lecionada pelo Prof. José Luis Braga. Esses exercícios foram extraídos de Humphrey (1995).

A.1 PROGRAMA 1A

Pré-requisitos: usar o PSP0.0.

Escrever um programa para calcular o ponto central e o desvio padrão de uma série de números reais n. O ponto central é a média dos números. A fórmula para o desvio padrão é:

$$S(X_1, ..., X_n) = \sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - x_{avg})^2}{n-1}}$$

Onde $S(X_1, ..., X_n)$ ou σ é o desvio padrão do valor X, e X_{avg} é a média destes n valores. Usar uma lista encadeada para guardar os números n para os cálculos.

Testes: Testar o programa completamente. Pelo menos três dos testes devem usar

os dados de cada uma das três colunas da tabela mostrada na Figura A.1. Os desvios padrão para as colunas nesta tabela são *LOC* Objeto: 572,03; *LOC* Novas & Alteradas: 625,63 e Horas de Desenvolvimento: 62,26.

Tabela de apoio Programa 1 – Tamanho do Objeto Pascal e Horas de Desenvolvimento.

Programa Número	LOC Objeto	LOC Novas & Alterada	Horas de Desenvolvimento
1	160	186	15.0
2	591	699	69.9
3	114	132	6.5
4	229	272	22.4
5	230	291	28.4
6	270	331	65.9
7	128	199	19.4
8	1657	1890	198.7
9	624	788	38.8
10	1503	1601	138.2

A.2 R1 - PADRÃO DE CONTAGEM DE LINHAS DE CÓDIGO

Objetivos da Tarefa:

- Definir os padrões de contagem de *LOCs* que sejam apropriados para a linguagem de programação usada;
- Prover uma base para desenvolver um padrão de codificação;
- Preparar-se para desenvolver um programa para contar *LOCs*.

Exercício R1: Desenvolver e submeter um padrão para contagem de LOCs lógicas para a linguagem de programação usada para escrever os programas dos exercícios do PSP.

Resultado esperado: Produzir, documentar e submeter o padrão completo.

Cronograma: Fazer esta tarefa antes ou em conjunto com a escrita do programa 2A.

A.3 R2 - PADRÃO DE CODIFICAÇÃO

Objetivos da Tarefa:

- Estabelecer um conjunto consistente de práticas de codificação;
- Prover critérios para julgamento da qualidade do código produzido;
- Facilitar a contagem de *LOCs*, assegurando que os programas são escritos com uma linha física separada para cada linha lógica de código.

Exercício R2: Produzir um padrão de codificação que exija qualidade de práticas de codificação. Também assegurar-se que uma linha física separada é usada para cada linha lógica de código, como definido no exercício R1.

Resultado esperado: Produzir, documentar e submeter o padrão de codificação completo.

Cronograma: Fazer esta tarefa depois de produzir o relatório R1, antes ou em conjunto com a escrita do programa 2A.

A.4 PROGRAMA 2A

Pré-requisitos: usar o PSP0.1.

Escrever um programa para contar as linhas lógicas em um programa, omitindo comentários e linhas em branco. Usar o contador padrão produzido através do exercício de relatório R1 para colocar uma linha lógica em cada linha física e contar linhas físicas. Produzir uma única contagem para o arquivo fonte do programa inteiro.

Testes: Testar completamente o programa. Contar as *LOCs* nos programas 1A e 2A.

A.5 PROGRAMA 3A

Pré-requisitos: programa 2A, usar o PSP0.1.

Escrever um programa para contar as *LOC* totais de programas, *LOCs* totais de cada objeto que o programa contém e o número de métodos em cada objeto. Produzir um contador de *LOCs* para um arquivo fonte inteiro e um contador separado de métodos e *LOCs* para cada objeto. Imprimir cada nome de objeto junto com suas *LOCs* e o método de contagem. Também imprimir a contagem total de *LOC* do programa. Se uma linguagem orientada a objetos não é usada, contar as *LOCs* de procedimentos e funções e imprimir os nomes dos procedimentos e funções e a contagem de *LOC*. Usar o padrão de contagem produzido para o exercício de relatório R1. É aceitável melhorar o programa 2A ou reusar alguns de seus métodos, procedimentos, ou funções no desenvolvimento do programa 3A.

Testes: testar completamente o programa. No mínimo, testar o programa contando as *LOCs* totais do programa e objetos nos programas 1A, 2A e 3A. Incluir no relatório de teste uma tabela que dá as contas obtidas com o programa 2A e 3A para todos os programas escritos até a data.

A.6 R3 - O RELATÓRIO DE ANÁLISE DE DEFEITO

Objetivos da Tarefa:

- Entender a densidade e os tipos de defeitos introduzidos e achados ao se desenvolver os programas iniciais deste trabalho;

- Demonstrar a importância de juntar cuidadosamente, registrar e relatar dados de processo.

Tarefa: analisar os defeitos encontrados no desenvolvimento dos programas iniciais e produzir um relatório que inclua o seguinte:

- o número total de defeitos achados;
- as *LOCs* Novas & Alteradas, e;
- os defeitos por $KLOC$;
- o número de defeitos achados na compilação;
- o número de defeitos achados nos testes;
- o número de defeitos por $KLOC$ achados na compilação, e;
- o número de defeitos por $KLOC$ achados nos testes;
- produzir uma tabela que mostre a média de tempo de correção para os defeitos achados na compilação, defeitos achados nos testes, defeitos injetados no projeto, e defeitos injetados na codificação.

Resultados esperados: fornecer estes dados para cada programa e para o total dos programas escritos até a Data.

Cronograma: fazer esta tarefa junto com ou imediatamente após a conclusão do programa 3A.

Apêndice B

Dados Brutos

Neste Apêndice são apresentados os dados coletados pelos alunos durante a elaboração dos programas 1A, 2A e 3A, que foram apresentados no Apêndice A. Esses dados foram utilizados para realização da análise estatística e conclusão deste trabalho. Vale lembrar que estes dados não possuem tratamento algum.

B.1 Dados Coletados em 2004 - PSP

B.1.1 Tempo - Programa 1A - PSP

Planning	Design	Code	Code Review	Compile	Test	Postmortem
3	8	15	5	4	24	5
1	5	30	2	5	4	1
1	3	37	0	25	6	5
2	0	12	0	2	10	0
5	4	40	0	22	3	6
4	29	39	24	2	8	5
3	6	65	3	17	6	7
1	3	14	0	2	3	2
13	15	42	0	8	5	24
11	2	23	0	26	11	30
4	5	127	0	6	7	10
1	2	5	0	5	2	5
2	5	31	0	22	17	8
7	10	27	0	2	10	6
20	9	31	0	1	3	13
2	13	52	0	83	12	0
0	5	36	0	1	6	0
2	7	38	0	3	12	10
2	5	53	0	6	4	11
1	1	29	1	1	11	1
3	5	90	7	1	7	6
10	10	30	10	5	10	5
10	10	25	0	5	4	0
0	0	50	0	3	10	0
2	2	14	3	3	20	12
0	5	10	0	7	0	0
0	30	31	0	13	27	0
1	7	36	0	15	12	4
1	2	24	0	1	9	3
3	14	19	4	47	5	11
1	1	43	0	1	1	1
15	13	50	0	23	16	0
20	5	31	3	2	30	10

Figura B.1: Planilha de coleta de dados - Tempo 1A

B.1.2 Tempo - Programa 2A - PSP

Planning	Design	Code	Code Review	Compile	Test	Postmortem
2	2	15	4	5	114	2
3	5	80	6	19	9	2
9	3	71	0	10	21	17
4	5	27	0	22	29	4
3	27	43	0	6	34	17
4	29	16	0	26	13	6
8	16	33	2	13	4	5
5	44	54	0	15	26	15
8	31	56	0	0	10	8
5	9	60	5	5	12	5
5	5	35	0	40	5	5
3	12	37	1	20	29	9
10	15	109	18	5	15	10
6	22	31	0	6	11	7
5	5	77	22	1	18	7
3	3	108	0	12	25	0
4	5	65	0	7	9	5
10	7	214	0	5	10	7
5	4	70	4	28	12	8
15	15	70	60	10	25	20
8	20	76	10	40	59	0
5	3	25	4	5	6	5
1	3	74	0	7	20	10
10	4	30	5	2	14	12
11	17	85	19	45	9	19
70	14	77	10	3	6	0
35	16	36	0	46	42	0
9	3	30	5	3	84	10
3	7	29	0	3	2	7

Figura B.2: Planilha de coleta de dados - Tempo 2A

B.1.3 Tempo - Programa 3A - PSP

Planning	Design	Code	Code Review	Compile	Test	Postmortem
2	3	74	5	6	114	2
3	4	105	7	8	13	3
1	37	52	0	10	8	20
4	24	43	2	12	21	5
3	7	40	0	1	56	7
1	6	34	0	17	7	4
9	12	39	3	15	7	4
8	34	34	17	9	87	13
9	73	101	0	13	11	5
4	7	63	0	4	8	13
5	9	35	3	3	4	5
5	5	20	0	7	8	5
4	10	21	0	1	11	2
18	58	79	17	120	50	20
34	61	233	0	49	70	9
10	20	45	0	86	10	5
6	7	63	22	4	7	5
1	3	41	0	1	195	16
5	18	80	6	5	11	12
20	10	88	0	10	10	7
5	2	20	25	4	20	14
4	6	113	6	4	17	4
5	5	100	5	30	10	10
0	0	17	0	30	15	2
2	5	58	0	23	70	20
8	6	50	5	2	11	12
3	14	19	4	47	5	11
16	30	162	0	60	15	35
29	19	20	10	5	30	25

Figura B.3: Planilha de coleta de dados - Tempo 3A

B.1.4 Defeitos Injetados - Programa 1A - PSP

Planning	Design	Code	Code Review	Compile	Test
0	1	4	0	0	0
0	0	5	0	0	0
0	0	0	3	0	0
0	0	4	0	0	0
0	0	3	0	0	0
0	0	3	0	0	0
0	0	3	0	0	0
0	1	1	0	0	0
0	0	2	0	0	0
0	0	5	0	0	1
0	0	8	0	0	0
0	0	2	0	0	0
0	1	5	0	0	0
0	0	4	0	0	0
0	0	4	0	0	0
0	0	3	0	0	0
0	0	4	0	0	0
0	0	6	0	0	0
0	1	4	0	0	0
0	0	2	0	0	0
0	0	6	0	0	0
0	0	1	0	0	0
0	0	2	0	0	0
0	0	4	0	0	0
0	6	1	0	1	0
0	0	4	0	0	0
0	0	1	0	0	0
0	0	11	0	0	0
0	0	1	0	0	0
0	1	3	0	0	0
0	0	1	0	0	1

Figura B.4: Planilha de coleta de dados - Defeitos Injetados 1A

B.1.5 Defeitos Injetados - Programa 2A - PSP

Planning	Design	Code	Code Review	Compile	Test
0	0	5	0	0	0
0	2	6	0	0	0
0	0	6	0	0	0
0	0	6	0	0	0
0	3	1	0	0	0
0	0	5	0	0	0
0	0	3	0	0	0
0	1	7	0	0	0
0	1	0	0	0	0
0	0	7	0	0	0
0	0	1	0	0	0
0	1	15	0	0	0
0	0	4	0	0	0
0	0	8	0	0	0
0	0	7	0	0	0
0	0	8	0	0	0
0	0	4	0	0	0
0	0	10	0	0	0
0	0	10	0	0	0
0	0	0	0	1	1
0	0	5	0	0	0
3	1	13	0	2	2
0	0	3	0	1	0
0	0	2	0	0	0
0	1	10	0	0	0
0	0	0	0	1	1
1	0	6	0	1	0
0	0	1	0	0	0
0	0	2	0	0	0

Figura B.5: Planilha de coleta de dados - Defeitos Injetados 2A

B.1.6 Defeitos Injetados - Programa 3A - PSP

Planning	Design	Code	Code Review	Compile	Test
0	0	7	0	0	0
0	4	5	0	0	0
0	0	4	0	1	0
0	0	4	0	0	0
0	2	3	0	0	0
0	0	7	0	0	0
0	0	4	0	0	0
0	2	3	0	0	0
0	2	1	0	0	0
0	0	4	0	0	0
0	0	0	0	0	0
0	0	1	0	0	0
0	0	7	0	0	0
0	1	22	0	0	0
0	0	4	0	0	0
0	0	4	0	0	0
0	0	5	0	0	0
0	0	4	0	0	0
0	0	11	0	0	0
1	0	1	0	0	0
0	0	6	0	0	0
0	0	7	0	5	0
1	3	6	0	0	0
0	0	5	0	0	0
0	0	3	0	0	0
0	0	11	0	0	0
0	0	0	0	0	0
0	0	7	0	0	0

Figura B.6: Planilha de coleta de dados - Defeitos Injetados 3A

B.1.7 Defeitos Removidos - Programa 1A - PSP

Planning	Design	Code	Code Review	Compile	Test
0	0	1	0	2	2
0	0	0	0	5	0
0	0	0	0	2	1
0	0	0	0	2	2
0	0	0	0	3	0
0	0	0	0	3	0
0	0	0	0	2	1
0	0	1	0	1	0
0	0	0	0	2	0
0	0	1	0	5	0
0	0	0	0	5	3
0	0	0	0	2	0
0	0	2	0	2	0
0	0	0	0	4	0
0	0	0	0	2	0
0	0	0	0	1	3
0	0	0	0	2	4
0	1	2	0	1	1
0	0	0	0	2	0
0	0	0	0	6	0
0	0	0	0	1	0
0	0	0	0	1	1
0	0	0	0	2	2
0	1	2	0	5	0
0	0	2	0	0	2
0	0	0	0	0	1
0	0	1	0	9	1
0	0	0	0	1	0
0	0	0	0	1	3
0	0	0	0	0	2

Figura B.7: Planilha de coleta de dados - Defeitos Removidos 1A

B.1.8 Defeitos Removidos - Programa 2A - PSP

Planning	Design	Code	Code Review	Compile	Test
0	0	0	0	3	2
0	0	0	2	2	4
0	0	0	0	4	2
0	0	0	0	1	3
0	0	0	0	5	0
0	0	0	0	2	1
0	0	0	0	5	3
0	0	0	0	0	1
0	0	0	0	5	2
0	0	0	0	1	0
0	0	0	0	15	1
0	0	0	3	0	1
0	0	0	0	6	2
0	0	0	0	3	4
0	0	0	0	4	4
0	0	0	0	3	1
0	0	0	0	4	6
0	0	0	0	10	0
0	0	0	0	1	1
0	0	0	2	2	1
0	4	0	0	13	4
0	0	0	0	3	1
0	0	0	0	1	1
0	0	1	3	7	0
0	0	0	0	1	1
0	0	0	0	7	1
0	0	0	0	0	1
0	0	0	0	1	1

Figura B.8: Planilha de coleta de dados - Defeitos Removidos 2A

B.1.9 Defeitos Removidos - Programa 3A - PSP

Planning	Design	Code	Code Review	Compile	Test
0	0	0	0	4	3
0	0	0	1	2	6
0	0	0	0	3	2
0	0	0	0	3	1
0	0	0	0	0	5
0	0	0	0	7	0
0	0	0	0	2	2
0	0	0	0	3	2
0	0	0	0	3	0
0	0	0	0	3	1
0	0	0	0	0	0
0	0	0	0	1	0
0	0	0	0	6	1
0	0	0	0	19	4
0	0	0	0	4	0
0	0	0	0	2	2
0	0	0	0	2	3
0	0	0	0	3	1
0	0	4	0	7	0
0	0	0	0	6	0
0	0	0	2	8	2
1	1	0	0	8	0
0	0	1	0	1	3
0	0	0	0	1	2
0	0	1	0	9	1
0	0	0	0	0	0
0	0	1	2	2	2

Figura B.9: Planilha de coleta de dados - Defeitos Removidos 3A

B.2 Dados Coletados em 2005 - PSPm

B.2.1 Tempo - Programa 1A - PSPm

Planning	Design	Code	Code Review	Compile	Test	Postmortem
5	10	30	0	2,5	2,5	1
4	8	147	34	17	12	10
5	10	20	0	15	10	1
4	1	22	0	5	8	6
8	11	30	5	3	18	10
5	5	15	0	1	14	5
2	10	123	0	47	20	1
7	47	90	0	39	18	1
7	36	149	13	63	162	17
2	10	67	8	5	7	4
13	12	41	6	15	12	6
10	9	94	0	1	17	8
20	13	18	0	5	10	10
17	15	95	0	12	12	17
3	8	60	0	23	8	3
9	30	51	0	6	34	5
23	12	33	26	21	11	5
1	5	68	0	16	49	8
2	18	135	0	95	65	5
10	14	106	0	6	36	18
3	12	15	0	7	2	1
30	25	51	5	5	18	2
20	13	18	0	5	10	10
12	37	80	35	2	15	25
20	37	173	0	209	37	1
16	9	43	8	11	10	9
4	10	50	0	69	56	6
20	13	18	0	5	10	10
27	16	84	0	1	25	2
20	10	40	0	15	8	10
5	24	202	0	2	152	60
45	70	97	0	66	45	2
5	24	88	16	4	5	5
5	10	12	0	3	5	15
15	15	130	40	5	10	5
1	2	130	0	41	26	2
5	10	40	0	1	9	10
21	12	16	0	7	10	10
25	15	80	30	35	40	5

Figura B.10: Planilha de coleta de dados - Tempo 1A

B.2.2 Tempo - Programa 2A - PSPm

Planning	Design	Code	Code Review	Compile	Test	Postmortem
10	10	53	0	19	3	1,5
3	2	65	23	8	19	1,8
5	10	20	0	15	10	1,5
16	4	40	5	17	3	7
5	9	29	2	16	14	6
3	4	15	0	3	3	5
15	15	41	0	34	17	1
9	21	94	0	5	31	16
3	10	60	10	5	10	5
6	5	20	7	6	9	3
1	5	19	0	3	4	15
2	21	75	0	1	32	2
2	6	55	0	14	7	4
13	38	37	0	18	19	10
10	4	32	0	11	4	35
8	26	33	0	4	31	13
2	6	25	0	25	5	2
13	9	51	0	6	15	21
4	3	13	0	3	2	2
12	5	21	1	7	5	10
5	8	80	0	6	15	7
2	4	73	17	1	9	5
2	42	79	0	15	4	35
5	5	15	7	5	8	3
3	8	102	0	9	43	1
1	1	12	2	4	1	1
8	9	61	0	1	47	12
15	8	26	10	11	9	9
2	12	188	0	54	10	4
3	4	28	0	10	14	1
4	10	28	5	13	4	5
2	3	12	0	2	8	9
5	5	85	15	4	15	5
4	7	82	0	4	12	1
6	14	47	0	2	18	14
2	2	13	0	6	10	10
14	12	72	0	30	50	19

Figura B.11: Planilha de coleta de dados - Tempo 2A

B.2.3 Tempo - Programa 3A - PSPm

Planning	Design	Code	Code Review	Compile	Test	Postmortem
2	25	95	0	22	22	2
5	16	240	0	8	3	7
25	25	30	0	15	15	15
15	20	36	0	30	25	8
7	22	78	0	10	19	5
3	20	75	15	5	13	3
3	7	42	0	36	18	5
35	5	115	0	10	10	8
7	32	152	0	158	14	9
19	13	90	0	65	18	7
34	25	68	0	12	192	14
10	118	204	0	223	56	12
9	10	77	0	27	17	20
6	11	78	0	2	6	1
15	20	130	0	36	25	8
5	12	200	0	50	10	1
15	11	55	13	15	12	3
3	3	90	0	30	50	1
30	30	90	0	40	23	2
14	11	29	0	1	10	6
15	8	26	10	11	9	9
6	12	62	0	30	20	10
60	30	65	70	55	80	10
5	3	22	0	11	12	8
5	5	100	20	4	10	3
6	17	65	0	40	28	2
15	30	42	0	2	12	9
2	3	15	0	10	3	1

Figura B.12: Planilha de coleta de dados - Tempo 3A

B.2.4 Defeitos Injetados - Programa 1A - PSPm

Planning	Design	Code	Code Review	Compile	Test
0	0	0	0	0	0
0	0	2	0	0	0
0	2	6	0	0	0
0	0	1	0	0	0
0	0	6	0	0	0
0	0	0	0	0	0
0	0	7	0	0	0
0	0	2	0	0	0
1	0	15	0	0	3
0	0	3	0	0	0
0	0	6	0	0	0
0	1	3	0	0	0
0	0	7	0	0	0
0	0	4	0	0	1
0	1	7	0	1	0
0	0	5	0	0	0
0	1	8	0	0	0
0	0	5	0	0	0
0	0	10	0	0	0
0	0	6	0	0	0
0	0	1	0	0	0
2	0	5	0	0	0
0	0	5	0	0	0
0	0	2	0	0	0
0	0	5	0	2	1
0	0	6	0	0	0
0	0	5	0	0	2
0	0	7	0	0	0
0	0	1	0	0	0
0	0	6	0	0	0
0	0	4	0	0	0
0	1	7	0	0	0
0	0	3	0	0	0
0	0	5	0	0	0
0	1	5	0	0	0
0	1	5	0	0	0
0	0	0	0	0	0
0	0	5	0	0	0
0	1	7	0	0	0

Figura B.13: Planilha de coleta de dados - Defeitos Injetados 1A

B.2.5 Defeitos Injetados - Programa 2A - PSPm

Planning	Design	Code	Code Review	Compile	Test
0	0	4	0	0	0
0	0	3	0	0	0
0	1	7	0	0	0
0	0	1	0	0	0
0	0	1	0	0	0
0	0	3	0	0	0
0	0	3	0	0	0
0	0	4	0	0	0
0	0	2	0	0	0
0	0	3	0	0	0
0	0	1	0	0	0
0	1	3	0	0	0
0	0	9	0	0	0
0	1	5	0	0	0
0	0	5	0	0	0
0	0	16	0	0	0
0	0	3	0	0	0
0	0	3	0	0	1
0	0	0	0	0	0
0	0	4	0	0	0
0	0	5	0	1	0
0	0	4	0	0	0
0	0	5	0	2	4
0	0	1	0	0	0
0	0	2	0	0	0
0	0	4	0	0	0
0	0	2	0	0	0
0	0	6	0	0	0
0	0	4	0	0	0
0	0	7	0	0	0
0	0	3	0	0	0
0	0	3	0	0	0
0	0	4	0	0	0
0	0	4	0	0	0
0	0	1	0	0	0
0	0	2	0	0	0
0	0	10	0	0	1

Figura B.14: Planilha de coleta de dados - Defeitos Injetados 2A

B.2.6 Defeitos Injetados - Programa 3A - PSPm

Planning	Design	Code	Code Review	Compile	Test
0	0	8	0	0	0
0	0	4	0	0	0
1	0	6	0	1	0
0	0	5	0	1	0
0	0	5	0	0	0
0	0	2	0	0	0
0	0	3	0	0	0
1	1	3	0	0	0
0	1	5	0	0	0
0	0	9	0	0	1
0	0	6	0	0	0
0	0	7	0	0	0
0	0	5	0	0	0
0	0	1	0	0	0
0	0	7	0	2	0
0	0	7	0	0	0
0	0	6	0	1	0
0	0	4	0	0	0
0	0	7	0	0	0
0	0	4	0	0	0
0	0	6	0	0	0
0	0	3	0	0	0
5	0	10	0	0	0
0	0	6	0	0	0
0	1	4	0	0	0
0	1	2	0	0	0
0	0	1	0	0	0
0	0	4	0	0	0

Figura B.15: Planilha de coleta de dados - Defeitos Injetados 3A

B.2.7 Defeitos Removidos - Programa 1A - PSPm

Planning	Design	Code	Code Review	Compile	Test
0	0	0	0	0	0
0	0	0	0	2	0
0	0	0	0	5	3
0	0	0	0	1	0
0	0	0	0	5	1
0	0	0	0	0	0
0	0	0	0	7	0
0	0	0	0	2	0
0	0	0	0	11	8
0	0	0	0	3	0
0	0	0	0	4	2
0	0	1	0	1	2
0	0	0	0	7	0
0	0	0	0	1	4
0	0	3	0	4	2
0	0	0	0	1	4
0	0	0	0	6	3
0	0	0	0	4	1
0	0	0	0	9	1
0	0	0	0	5	1
0	0	0	0	1	0
0	0	0	0	7	0
0	0	0	0	5	0
0	0	0	0	2	0
0	0	4	0	3	1
0	0	0	0	4	2
0	0	0	0	3	4
0	0	0	0	7	0
0	0	0	0	1	0
0	0	0	0	5	1
0	0	0	0	1	3
0	0	0	0	7	1
0	0	0	0	3	0
0	0	0	0	5	0
0	0	0	0	5	1
0	0	0	0	5	1
0	0	0	0	0	0
0	0	0	0	5	0
0	0	0	0	7	1

Figura B.16: Planilha de coleta de dados - Defeitos Removidos 1A

B.2.8 Defeitos Removidos - Programa 2A - PSPm

Planning	Design	Code	Code Review	Compile	Test
0	0	0	0	3	1
0	0	0	0	2	1
0	0	0	0	5	3
0	0	0	0	1	0
0	0	0	0	1	0
0	0	0	0	3	0
0	0	0	0	3	0
0	0	0	0	2	2
0	0	0	0	2	0
0	0	0	0	2	1
0	0	0	0	0	1
0	0	1	0	0	3
0	0	4	0	5	0
0	0	0	0	5	1
0	0	0	0	5	0
0	0	0	0	2	14
0	0	0	0	3	0
0	0	0	0	3	1
0	0	0	0	0	0
0	0	0	0	4	0
0	0	0	0	4	2
0	0	0	0	3	1
0	0	0	0	7	4
0	0	0	0	1	0
0	0	0	0	2	0
0	0	0	0	4	0
0	0	0	0	0	2
0	0	0	0	4	2
0	0	1	0	3	0
0	0	0	0	7	0
0	0	0	0	3	0
0	0	0	0	3	0
0	0	0	0	4	0
0	0	0	0	4	0
0	0	0	0	1	0
0	0	0	0	2	0
0	0	0	0	6	5

Figura B.17: Planilha de coleta de dados - Defeitos Removidos 2A

B.2.9 Defeitos Removidos - Programa 3A - PSPm

Planning	Design	Code	Code Review	Compile	Test
0	0	0	0	8	0
0	0	0	0	4	0
0	0	0	0	7	1
0	0	0	0	5	1
0	0	0	0	3	2
0	0	0	0	2	0
0	0	0	0	2	1
0	0	1	0	2	2
0	0	0	0	5	1
0	0	2	0	7	1
0	0	0	0	3	3
0	0	0	0	7	0
0	0	0	0	4	1
0	0	0	0	1	0
0	0	0	0	8	1
0	0	2	0	5	0
0	0	0	0	5	2
0	0	0	0	4	0
0	0	0	0	7	0
0	0	0	0	0	4
0	0	0	0	4	2
0	0	0	0	3	0
0	0	0	0	10	5
0	0	0	0	6	0
0	0	0	0	3	2
0	0	0	0	2	1
0	0	0	0	1	0
0	0	0	0	4	0

Figura B.18: Planilha de coleta de dados - Defeitos Removidos 3A

Apêndice C

Scripts de Execução

Neste Apêndice são apresentados os scripts utilizados para execução dos cálculos por reamostragem com a utilização do software RESAMPLING STATS (Versão demonstrativa obtida em julho de 2005 no URL:
http://www.resample.com/content/software/download.shtml).

C.1 Análise de Tempo - Fase Teste

C.1.1 Programa 2A

```
'-------------------------------------------------------------------------
'RENATO AFONSO COTA SILVA
'DEPARTAMENTO DE INFORMÁTICA
'UNIVERSIDADE FEDERAL DE VIÇOSA
'ANÁLISE ESTATÍSTICA PARA A DISSERTAÇÃO DE MESTRADO - MELHORIA DE QUALIDADE
'PESSOAL DE DESENVOLVIMENTO DE SOFTWARE
'PROGRAMA ANALISADO: PSP2A
'MÉTODO USADO: PSP E PSPm
'ANÁLISE DE TEMPO GASTO NA FASE TEST
'-------------------------------------------------------------------------
```

```
numbers (114 9 21 29 34 13 4 26 10 12 5 29 15 11 18 25 9 10 12 25 59 6 20 14
9 6 42 84 2) psp
numbers (3 19 10 3 14 3 31 10 9 4 32 7 19 4 31 5 15 2 15 8 43 1 9 4 8 15 18
10 50) pspm

mean psp mpsp
mean pspm mpspm
print mpsp
print mpspm
subtract mpsp mpspm delta
abs delta mdelta
print delta
print mdelta

concat psp pspm desid
maxsize cl1 10000
repeat 10000
  shuffle desid sim
  sample 29 sim psp
  sample 29 sim pspm
  mean psp mpsp
  mean pspm mpspm
  subtract mpsp mpspm diff
  score diff cl1
end
histogram cl1
count cl1 >= mdelta k
multiply mdelta -1 mdelta
print mdelta
```

```
count cl1 <= mdelta q
divide k 10000 mm
divide q 10000 nn
percentile cl1 (5 95) p
print mm
print nn
print p
```

C.1.2 Programa 3A

```
'-------------------------------------------------------------------------
'RENATO AFONSO COTA SILVA
'DEPARTAMENTO DE INFORMÁTICA
'UNIVERSIDADE FEDERAL DE VIÇOSA
'ANÁLISE ESTATÍSTICA PARA A DISSERTAÇÃO DE MESTRADO - MELHORIA DE QUALIDADE
PESSOAL DE DESENVOLVIMENTO DE SOFTWARE
'PROGRAMA ANALISADO: PSP3A
'MÉTODO USADO: PSP E PSPm
'ANÁLISE DE TEMPO GASTO NA FASE TEST
'-------------------------------------------------------------------------

numbers (114 13 8 21 56 7 7 87 11 4 8 11 50 70 10 7 195 11 10 20 17 10 15 70
 11 5 15 30) psp
numbers (22 3 15 25 19 13 18 10 14 18 192 56 17 6 25 10 12 50 23 10 9 20 80
 12 10 28 12 3) pspm

mean psp mpsp
mean pspm mpspm
print mpsp
print mpspm
subtract mpsp mpspm delta
```

```
abs delta mdelta
print delta
print mdelta

concat psp pspm desid
maxsize cl1 10000
repeat 10000
  shuffle desid sim
  sample 28 sim psp
  sample 28 sim pspm
  mean psp mpsp
  mean pspm mpspm
  subtract mpsp mpspm diff
  score diff cl1
end
histogram cl1
count cl1 >= mdelta k
multiply mdelta -1 mdelta
print mdelta
count cl1 <= mdelta q
divide k 10000 mm
divide q 10000 nn
percentile cl1 (5 95) p
print mm
print nn
print p
```

C.2 Análise de Defeitos Injetados - Fase Projeto

C.2.1 Programa 2A

```
'------------------------------------------------------------------------
'RENATO AFONSO COTA SILVA
'DEPARTAMENTO DE INFORMÁTICA
'UNIVERSIDADE FEDERAL DE VIÇOSA
'ANÁLISE ESTATÍSTICA PARA A DISSERTAÇÃO DE MESTRADO - MELHORIA DE QUALIDADE
'PESSOAL DE DESENVOLVIMENTO DE SOFTWARE
'PROGRAMA ANALISADO: PSP2A
'MÉTODO USADO: PSP E PSPm
'ANÁLISE DE DEFEITOS INJETADOS NA FASE DESIGN
'------------------------------------------------------------------------

numbers (0 2 0 0 3 0 0 1 1 0 0 1 0 0 5 0 0 0 0 0 0 0 0 0 1 0 0 0 0) psp
numbers (0 0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0) pspm

mean psp mpsp
mean pspm mpspm
print mpsp
print mpspm
subtract mpsp mpspm delta
abs delta mdelta
print delta
print mdelta

concat psp pspm desid
maxsize cl1 10000
repeat 10000
  shuffle desid sim
```

```
  sample 29 sim psp
  sample 29 sim pspm
  mean psp mpsp
  mean pspm mpspm
  subtract mpsp mpspm diff
  score diff cl1
end
histogram cl1
count cl1 >= mdelta k
multiply mdelta -1 mdelta
print mdelta
count cl1 <= mdelta q
divide k 10000 mm
divide q 10000 nn
percentile cl1 (5 95) p
print mm
print nn
print p
```

C.2.2 Programa 3A

```
'-------------------------------------------------------------------------
'RENATO AFONSO COTA SILVA
'DEPARTAMENTO DE INFORMÁTICA
'UNIVERSIDADE FEDERAL DE VIÇOSA
'ANÁLISE ESTATÍSTICA PARA A DISSERTAÇÃO DE MESTRADO - MELHORIA DE QUALIDADE
PESSOAL DE DESENVOLVIMENTO DE SOFTWARE
'PROGRAMA ANALISADO: PSP3A
'MÉTODO USADO: PSP E PSPm
'ANÁLISE DE DEFEITOS INJETADOS NA FASE DESIGN
'-------------------------------------------------------------------------
```

```
numbers (0 4 0 0 2 0 0 2 2 0 0 0 0 1 0 0 0 0 0 0 0 0 3 0 0 0 0 0) psp
numbers (0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0) pspm

mean psp mpsp
mean pspm mpspm
print mpsp
print mpspm
subtract mpsp mpspm delta
abs delta mdelta
print delta
print mdelta

concat psp pspm desid
maxsize cl1 10000
repeat 10000
  shuffle desid sim
  sample 28 sim psp
  sample 28 sim pspm
  mean psp mpsp
  mean pspm mpspm
  subtract mpsp mpspm diff
  score diff cl1
end
histogram cl1
count cl1 >= mdelta k
multiply mdelta -1 mdelta
print mdelta
count cl1 <= mdelta q
divide k 10000 mm
```

```
divide q 10000 nn

percentile cl1 (5 95) p

print mm

print nn

print p
```

Apêndice D

Empacotamento do Experimento

Uma das características mais importantes de um experimento é a necessidade de sua repetição. Com a repetição os pesquisadores adquirem o conhecimento adicional a respeito dos conceitos estudados, e percebem os resultados que são iguais ou diferentes dos resultados do experimento original. De qualquer maneira, o aumento das repetições traz o aumento do aprendizado dos conceitos investigados e, também, a calibração das características do experimento (Travassos et al., 2002).

O pré-requisito necessário para a repetição do experimento de alta qualidade é o seu empacotamento. Neste Apêndice são apresentadas as rotinas necessárias para repetição do experimento.

D.1 Comunidade de Engenharia de Software

Neste trabalho a comunidade de Engenharia de Software utilizada foram alunos seniores do curso de Ciência da Computação da Universidade Federal de Viçosa, que estavam matriculados na disciplina Engenharia de Software I.

É interessante que os executores do experimento tenham conhecimento semelhantes da linguagem de programação escolhida para realização do experimento e os dois grupos a serem estudados sejam equivalentes em conhecimento e intelecto.

Os executores do experimento podem ser: pesquisador, estudante ou desenvolvedor da indústria.

É importante que os estudantes selecionados para o experimento sejam seniores, cursando o terceiro ano de graduação, possuindo experiência de programação na linguagem escolhida.

Não necessariamente o experimento deve ser repetido em sala de aula, pelo contrário, é interessante que ele seja realizado na indústria de software, com programadores experientes. É importante ressaltar que para re-experimentação em sala de aula, não é necessário esperar dois anos, pois pode-se realizar o experimento em apenas uma turma, dividindo a turma ao meio e selecionando os grupos de forma a ficarem balanceados.

É importante lembrar que o experimento deve ser realizado a nível individual, pois, os processos aqui abordados se referem à competência pessoal.

O pesquisador que conduzirá a repetição do experimento deve ter profundo conhecimento em engenharia de software, PSP e PSPm, para que possa instruir os executores de maneira que eles executem o experimento adequadamente.

D.2 Organização do Experimento

Para realização do experimento é necessário que a comunidade de Engenharia de Software seja treinada com o PSP.

O livro "*A Discipline for Software Engineering*" de Humphrey (1995), apresenta todos os conceitos e exercícios necessários para treinar pessoas para execução do PSP. No livro são propostas duas séries de exercícios, num total de 19 programas (1A a 10A e 1B a 9B) e cinco relatórios (R1 a R5) a serem elaborados.

Humphrey (1995) sugere que o treinamento seja iniciado com a série A de exercícios e posteriormente adicionar exercícios da série B, em momento oportuno. Caso não seja possível aplicar todos os exercícios, pode-se selecionar os exercícios mais propícios ao objetivo do treinamento, de acordo com o tempo disponível para este. Cada um dos exercícios deve ser realizado utilizando os níveis de maturidade corretos do PSP. Todos os exercícios encontram-se disponíveis no Apêndice D de Humphrey (1995).

Humphrey (1995) descreve as lições e os capítulos que devem ser abordados durante o treinamento, antes da realização de cada exercício, conforme mostra a Tabela D.1.

É importante que os exercícios sejam aplicados e corrigidos ao longo do treinamento, pois assim fica evidente quais são as maiores dificuldades que os alunos encontram. É de fundamental importância que sejam mostrados os resultados de tempo, defeitos injetados e defeitos removidos de toda a turma, em forma de gráfico, após a correção de cada exercício. Dessa forma é traçado um perfil para a turma que está sendo treinada.

Após o término de cada exercício, os dados de tempo e defeitos devem ser lançados em uma planilha única contendo os dados de todos os participantes do treinamento para cada um dos exercícios realizados. Sendo uma planilha para o PSP e outra para o PSPm.

Concluído o treinamento, deve ser feita uma análise prévia de todos os dados, eliminando incorreções e ajustando discrepâncias.

O próximo passo é considerar quais serão as hipóteses alternativas H_1 que serão verificadas. Como o intuito principal deste experimento é verificar a injeção de defeitos no projeto do software, a principal hipótese alternativa é: $H_1 : PSP > PSPm$. Outras hipóteses alternativas podem ser consideradas, de acordo com o projetista do experimento, desde que tenham fundamento.

Para validar as hipóteses alternativas, deve ser usado, conforme justificado na seção 4.4.3, o Teste por Reamostragem *bootstrap*. Para que seja realizado o procedimento estatístico, faz-se necessário o uso de software. É recomendado que seja utilizado o programa Resampling Stats (Stats, 2005) e, para utilização desse software é necessário a criação de *scripts* de execução, os quais encontram-se disponíveis no Apêndice C.

D.3 Instrumentos

Formulários utilizados pelos executores para coleta de dados:

- *Time Recording Log:* Responsável pela coleta do tempo gasto em cada fase do projeto, sendo registrados também todas as interrupções sofridas durante o processo. O tempo delta é calculado com base no tempo total subtraído do tempo de interrupções, sendo portanto o tempo real gasto em cada fase, conforme mostra a Figura 4.1;

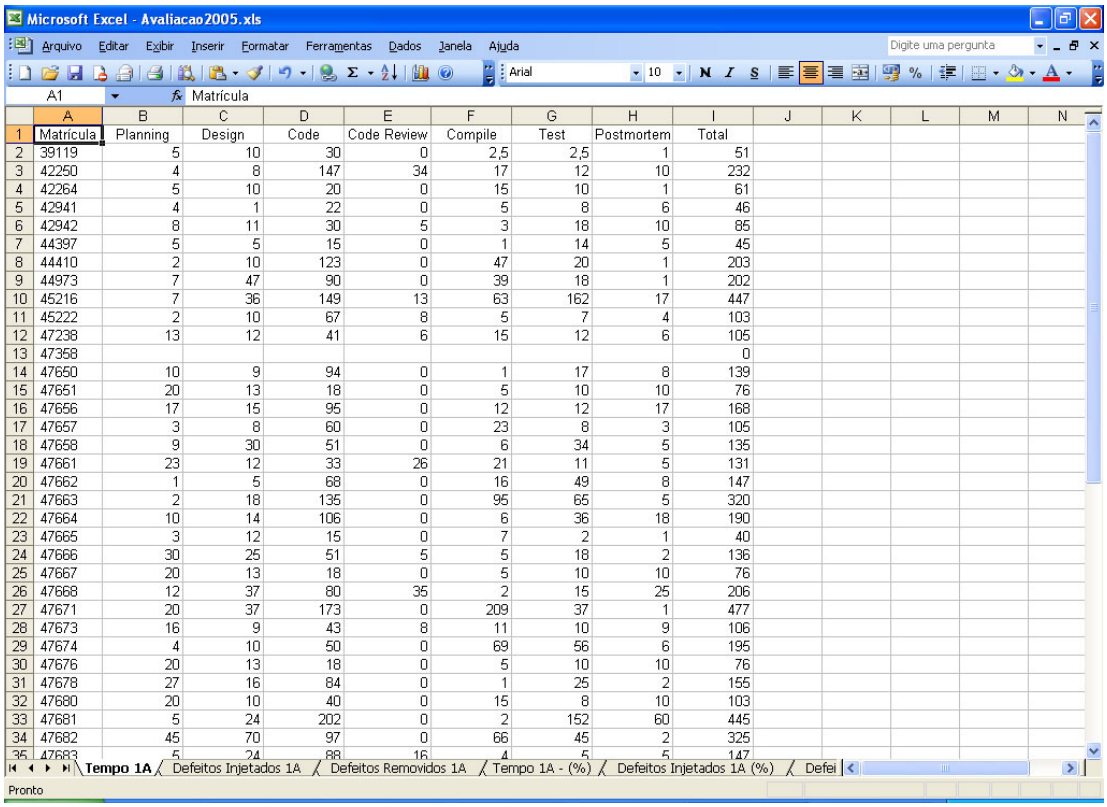
- *Defect Recording Log:* Reponsável pela coleta de todos os defeitos injetados e removidos durante o processo, conforme mostra a Figura 4.2;

- *Project Plan Summary:* É preenchido na última fase do processo - *PostMortem*, onde é feito um resumo de todos os dados das duas planilhas anteriores, mantendo um histórico dos dados de todos os programas desenvolvidos até a presente data, conforme mostra a Figura 4.3.

Um exemplo de planilha a ser utilizada para lançamento dos dados é apresentado na Figura D.1.

D.4 Roteiro de Re-experimentação

A Tabela D.2 apresenta um roteiro resumindo as atividades que devem ser realizadas na condução de um novo experimento.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Matrícula	Planning	Design	Code	Code Review	Compile	Test	Postmortem	Total					
2	39119	5	10	30	0	2,5	2,5	1	51					
3	42250	4	8	147	34	17	12	10	232					
4	42264	5	10	20	0	15	10	1	61					
5	42941	4	1	22	0	5	8	6	46					
6	42942	8	11	30	5	3	18	10	85					
7	44397	5	5	15	0	1	14	5	45					
8	44410	2	10	123	0	47	20	1	203					
9	44973	7	47	90	0	39	18	1	202					
10	45216	7	36	149	13	63	162	17	447					
11	45222	2	10	67	8	5	7	4	103					
12	47238	13	12	41	6	15	12	6	105					
13	47358								0					
14	47650	10	9	94	0	1	17	8	139					
15	47651	20	13	18	0	5	10	10	76					
16	47656	17	15	95	0	12	12	17	168					
17	47657	3	8	60	0	23	8	3	105					
18	47658	9	30	51	0	6	34	5	135					
19	47661	23	12	33	26	21	11	5	131					
20	47662	1	5	68	0	16	49	8	147					
21	47663	2	18	135	0	95	65	5	320					
22	47664	10	14	106	0	6	36	18	190					
23	47665	3	12	15	0	7	2	1	40					
24	47666	30	25	51	5	5	18	2	136					
25	47667	20	13	18	0	5	10	10	76					
26	47668	12	37	80	35	2	15	25	206					
27	47671	20	37	173	0	209	37	1	477					
28	47673	16	9	43	8	11	10	9	106					
29	47674	4	10	50	0	69	56	6	195					
30	47676	20	13	18	0	5	10	10	76					
31	47678	27	16	84	0	1	25	2	155					
32	47680	20	10	40	0	15	8	10	103					
33	47681	5	24	202	0	2	152	60	445					
34	47682	45	70	97	0	66	45	2	325					
35	47683	5	24	88	16	4	5	5	147					

Fonte: Elaborado por Balbino (2004)

Figura D.1: Exemplo de planilha para coleta dos dados

Tabela D.1: Plano de treinamento do PSP

Aula[1]	Capítulo	Exercício	Nível	Outros
1	1			
2	2	1A	PSP0	
3	3			
4	4	2A	PSP0.1	R1 R2
5	5			
6	5	3A	PSP0.1	R3
7	5	4A	PSP1	
8	6			
9	6	5A	PSP1.1	
10	7			
11	7	6A	PSP1.1	
12	8			
13	8			R4
14	9			
15	9	7A	PSP2	
16	10			
17	10	8A	PSP2.1	Ler Apêndice B de Humphrey (1995)
18	10			
19	10	9A	PSP2.1	
20	11			
21	11	10A	PSP3	
22	12			
23	12			R5
24	13			
25	14			

Fonte: Adaptado de Humphrey (1995)

[1] Considerando uma aula de 50 minutos

Tabela D.2: Roteiro de Re-experimentação

Passo	Atividade	Artefatos
1	Definir população	Alunos, Programadores ou outros
2	Definir Ambiente de Experimentação	Sala de Aula, Laboratórios, etc.
3	Definir Linguagem de Programação	C++, JAVA, etc...
4	Definir exercícios a serem aplicados	
5	Dividir a população em dois grupos	PSP e PSPm
6	Treinar a população de cada grupo com seu respectivo processo	PSP ou PSPm
7	Aplicar os respectivos exercícios conforme apresentado na Tabela D.1	
8	Coletar os dados da população	*Time Record Log* *Defect Record Log* *Project Plan Summary*
9	Resultados Crus	Lançar dados em planilha única, conforme apresentado no Apêndice B
10	Análise Prévia dos Dados	Eliminar dados discrepantes e incorretos
11	Hipóteses H_1: PSP $>$ PSPm	Defeitos Injetados no Projeto Tempo Gasto no Teste
12	Utilização de Teste po Reamostragem *bootstrap* para verificação das hipóteses. Os scripts de execução se encontram no Apêndice C	$\alpha = 0,10$

Fonte: Elaborado pelo Autor

Referências Bibliográficas

Albuquerque, J., Jr., C. C. N., de P. Paula Filho, W., and Fernandes, A. O. (1999). Using psp on undergraduate computer science programs. In *SoST'99: Proceedings of the Symposium on Software Technology*, pages 1–6, Buenos Aires - Argentina. 28th International Conference of the Argentine Informatics and Operations Research Society (SADIO).

Amaral, E. A. G. G. (2003). Empacotamento de experimentos em engenharia de software. Master's thesis, Universidade Federal do Rio de Janeiro, COPPE - Rio de Janeiro - RJ - Brasil. Dissertação de Mestrado.

Ambler, S. W. (2002). *Agile modeling effective practices for eXtreme programming and the unified process.* Wiley, New York.

Balbino, M. S. (2004). Um modelo para diagnóstico e melhoria de competência pessoal em qualidade de software. Master's thesis, Universidade Federal de Minas Gerais, ICEX. Dissertação de Mestrado.

Balbino, M. S. and Braga, J. L. (2004). Competência pessoal em qualidade de software com PSP na sala de aula. In *XII WEI - Workshop de Educacão em Computacão*, Salvador, BA. Sociedade Brasileira de Computacão, Anais do XXIV Congresso da SBC. Porto Alegre, RS.

Beck, K. (2000). *Extreme Programming Explained: Embrace Change.* Reading, MA: Addison Wesley Longman, Inc.

Beck, K. (2003). *Test-Driven Development : By Example.* Addison-Wesley, Boston.

Boehm, B. and Turner, R. (2004). *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley, Boston.

Chaplin, D. (2001). Test first programming. TechZone.

Conover, W. J. (1980). *Practical Nonparametric Statistics*. New York: John Wiley and Sons, Inc., 2nd edition.

Conradi, R., Basili, V. R., Carver, J., Shull, F., and Travassos, G. H. (2001). A pragmatic documents standard for an experience library: Roles,documen, contents and structure. Technical report, University of Maryland. CS-TR-4235.

Davison, A. C. and Hinkley, D. (1997). *Bootstrap Methods and their Applications*. Cambridge:Cambridge University Press.

Deming, W. E. (1986). *Out of the crisis*. Cambridge, Mass. : Massachusetts Institute of Technology, Center for Advanced Engineering Study.

Dion, R. (1993). Process improvement and the corporate balance sheet. *IEEE Softw.*, 10(4):28–35.

Edwards, S. H. (2003). Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance. In *Proceedings of the International Conference on Education and Information Systems: Technologies and Applications (EISTA '03)*.

Efron, B. and Tibshirani, R. J. (1993). *An Introduction to the Bootstrap*. London: Chapman and Hall.

Erdogmus, H. (2005). On the effectiveness of the test-first approach to programming. *IEEE Trans. Softw. Eng.*, 31(3):226–237. Member-Maurizio Morisio and Member-Marco Torchiano.

Erdogmus, H. and Wang, Y. (2004). The role of process measurement in test-driven development. In *Proceedings of XP Agile Universe*, Calgary, Alberta, Canada.

Ferguson, P., Humphrey, W., Khajenoori, S., Macke, S., and Matvya, A. (1997). Introducing the personal software process: Three industry case studies. *IEEE Computer*, 30(5):24–31.

Gelperin, D. and Hetzel, W. (1987). Software quality engineering. In *Fourth International Conference on Software Testing*, Washington, D.C.

George, B. and Williams, L. (2003). An initial investigation of test driven development in industry. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 1135–1139, New York, NY, USA. ACM Press.

Group, S. (2004). The chaos report. http://www.standishgroup.com/sample_research/PDFpages/spotlight.pdf. Acessado em 01/09/2005.

Hayes, W. and Over, J. (1997). The personal software process (psp): An empirical study of the impact of psp on individual engineers. Technical report, CMU/SEI. CMU/SEI-97-TR-001.

Hilburn, T. B., Hirmanpour, I., and Kornecki, A. J. (1995). The integration of software engineering into a computer science curriculum. In *Proceedings of the 8th SEI CSEE Conference on Software Engineering Education*, pages 87–97, London, UK. Springer-Verlag.

Humphrey, W. S. (1995). *A Discipline for Software Engineering*. Reading, Mass. : Addison-Wesley.

Humphrey, W. S. (1996). Using a defined and measured personal software process. *IEEE Softw.*, 13(3):77–88.

Humphrey, W. S. (1997). *Introduction to the Personal Software Process*. Reading, Mass. : Addison-Wesley.

Humphrey, W. S. (2000). The Personal Software Process (PSP). Technical report, CMU/SEI-2000-TR-022. ESC-TR-2000-022.

Humphrey, W. S. (2005). *PSP: A Self-Improvement Process for Engineers*. Reading, Mass. : Addison-Wesley.

IEEE (1990). IEEE Standard Glossary of Software Engineering Terminology, Standard 610.12. IEEE Press.

ISO/IEC12207, N. (2004). Tecnologia da informação - processos de ciclo de vida. ABNT 1998, Emenda 1: 2002, Emenda 2: 2004.

ISO/IEC15504 (2003/2004). Information technology - process assessment.

ISO/IEC9126 (1991). Software product evaluation - quality characteristics and guidelines for their use.

Jacobson, I., Booch, G., and Rumbaugh, J. (1999). *The Unified Software Development Process*. Addison-Wesley, Reading, Mass.

Jones, C. G. (2004). Test-driven development goes to school. *J. Comput. Small Coll.*, 20(1):220–231.

Juran, J. M. and Gryna, F. M. (1988). *Juran's quality control handbook*. New York : McGraw-Hill.

Kelly, D. P. and Culleton, B. (1999). Process improvement for small organizations. *IEEE Computer*, 32(10):41–47.

Kruchten, P. (2000). *The Rational Unified Process: An Introduction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Larman, C. and Basili, V. R. (2003). Iterative and incremental developments: A brief historys. *IEEE Computer*, 36:47–56.

Maletic, J. I., Marcus, A., and Howald, A. (2001). Incorporating psp into a traditional software engineering course: An experience report. In *Proceedings of the 14th IEEE Conference on Software Engineering Education and Training (CSEE&T)*, pages 89–97, Charlotte, NC, USA.

Maximilien, E. M. and Williams, L. (2003). Assessing test-driven development at ibm. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 564–569, Washington, DC, USA. IEEE Computer Society.

Mugridge, R. (2003). Test driven development and the scientific method. In *ADC '03: Proceedings of the Conference on Agile Development*, page 47, Washington, DC, USA. IEEE Computer Society.

Muller, M. M. and Hagner, O. (2002). Experiment about test-first programming. In *Software, IEE Proceedings*, pages 131 – 136, Karlsruhe, Germany. IEE Proc.-Softw.

Pancur, M., Ciglaric, M., Trampus, M., and Vidmar, T. (2003). Towards empirical evaluation of test-driven development in a university environment. In *EUROCON 2003*, Ljubljana, Slovenia. The IEEE Region 8, Computer as a Tool.

Paula Filho, W. P. (2003). *Engenharia de Software: Fundamentos, Métodos e Padrões.* LTC, Rio de Janeiro, segunda edition.

Paulk, M. C., WEBER, C. V., and GARCIA, S. M. (1993). Key practices of the capability maturity model. Technical report, CMU/SEI-93-TR-025. ESC-TR-93-178.

Peterson, I. (1996). *Fatal Defect: Chasing Killer Computer Bugs.* Vintage Books, New York.

Pfleeger, S. L. (1994). Design and analysis in software engineering: the language of case studies and formal experiments. *SIGSOFT Softw. Eng. Notes*, 19(4):16–20.

Pfleeger, S. L. (1995a). Experimental design and analysis in software engineering: Part 2: how to set up and experiment. *SIGSOFT Softw. Eng. Notes*, 20(1):22–26.

Pfleeger, S. L. (1995b). Experimental design and analysis in software engineering, part 4: choosing an experimental design. *SIGSOFT Softw. Eng. Notes*, 20(3):13–15.

Pfleeger, S. L. (1995c). Experimental design and analysis in software engineering, part 5: analyzing the data. *SIGSOFT Softw. Eng. Notes*, 20(5):14–17.

Pfleeger, S. L. (1995d). Experimental design and analysis in software engineering: Types of experimental design. *SIGSOFT Softw. Eng. Notes*, 20(2):14–16.

Pfleeger, S. L. (2001). *Software engineering: theory and practice*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2th edition.

Prechelt, L. and Unger, B. (2001). An experiment measuring the effects of personal software process (psp) training. *IEEE Trans. Softw. Eng.*, 27(5):465–472.

Pressman, R. S. (2001). *Software Engineering: A Practitioner's Approach*, volume 5. McGraw-Hill.

Rocha, A. R. C., Maldonado, J. C., and Weber, K. C. (2001). *Qualidade de Software: Teoria e Prática*. Prentice Hall, São Paulo, SP, BRASIL.

Seaman, C. B. (1999). Qualitative methods in empirical studies of software engineering. *IEEE Trans. Softw. Eng.*, 25(4):557–572.

SEI (2002). Capability maturity model integration (cmmi). Technical report, CMU/SEI. Version 1.1.

Senge, P. M. (1994). *The fifth discipline : the art and practice of the learning organization*. New York : DoubledayCurrency.

Silva, R. A. C. (2005). Inteligência artificial aplicada a ambientes de engenharia de software: Uma visão geral. *INFOCOMP Journal of Computer Science*, 4(4):27–37.

Silva, R. A. C., Braga, J. L., Silva, C. H. O., and Soares, L. S. (2006). Pspm na melhoria da competência pessoal no desenvolvimento de software. In *Proceedings of JIISIC'06: 5ta. Jornada Iberoamericana de Ingeniería del Software e Ingeniería del Conocimiento*, Puebla, Puebla, Mexico.

SOFTEX (2005). Mps.br - melhoria de processo do software brasileiro. Technical report, SOFTEX. Version 1.0.

Solingen, R. V. and Berghout, E. (1999). *The goal/question/metric method : a practical guide for quality improvement of software development*. London; Chicago : McGraw-Hill.

Sommerville, I. (2004). *Software Engineering*. Pearson Addison Wesley, 7th edition.

Stats, R. (2005). Resampling stats software. http://www.resample.com/. Acessado em 30/08/2005.

Tilley, S. (2004). Test-driven development and software maintenance. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 488 – 491. IEEE Computer Society.

Towhidnejad, M. and Hilburn, T. (1997). Integrating the personal software process (psp) across the undergraduate curriculum. In *Teaching and Learning in an Era of Change*, pages 162–168, Pittsburgh, PA, USA. Frontiers in Education Conference.

Travassos, G. H., Gurov, D., and Amaral, E. A. G. (2002). Introdução à engenharia de software experimental. Technical report, Programa de Engenharia de Sistemas e Computação - COPPE/UFRJ. RT-ES-590/02.

Tsukumo, A. N., Rêgo, C. M., Salviano, C. F., Azevedo, G. F., Meneghetti, L. K., Costa, M. C. C., Carvalho, M. B., and Colombo, R. M. T. (1997). Qualidade de software : Visões de produto e processo de software. In *II Escola Regional de Informática da Sociedade Brasileira de Computação Regional de São Paulo*, pages 173–189, Piracicaba, SP. II ERI da SBC.

Walters, S. J. and Capbell, M. J. (2004). The use of bootstrap methods for analysing health-related quality of life outcomes (particularly the sf-36). *Health and Quality of Life Outcomes*, page 2:70.

Wankat, P. C. (2002). *The effective, efficient professor: teaching, scholarship and service*. Allyn and Bacon, Boston, MA, USA.

Williams, L., Maximilien, E. M., and Vouk, M. (2003). Test-driven development as a defect-reduction practice. In *ISSRE '03: Proceedings of the 14th International*

Symposium on Software Reliability Engineering, page 34, Washington, DC, USA. IEEE Computer Society.

Wohlin, C. (2000). *Experimentation in Software Engineering : an introduction*. Kluwer Academic.