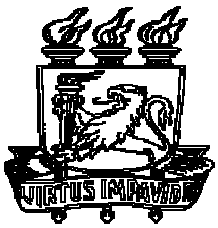

Universidade Federal de Pernambuco
Centro de Ciências Exatas e da Natureza
Departamento de Informática
Pós-graduação em Ciência da Computação

Dissertação de Mestrado

PSP-JOA

Processo de Software Pessoal - Uma Abordagem Orientada a Java

por

Jones Oliveira de Albuquerque

Orientador: Silvio Romero de Lemos Meira

Recife
20 de fevereiro de 1997

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE CIÊNCIAS EXATAS E DA NATUREZA
DEPARTAMENTO DE INFORMÁTICA

Jones Oliveira de Albuquerque

PSP-JOA

Processo de Software Pessoal: Uma Abordagem Orientada a Java

Este trabalho será apresentado à Pós-Graduação em Ciência da Computação do Centro de Ciências Exatas e da Natureza da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: Prof. Silvio Romero de Lemos Meira

Recife, 20 de fevereiro de 1997

A meus pais Oscar e Vera Lúcia
e a Líliam.

Agradecimentos

A todos aqueles que fazem o Departamento de Informática da UFPE. Especialmente ao pessoal de suporte técnico, Carlos Henrique, Carlos Lago, Júlio Glasner e Nadja Lins por manterem, competentemente, uma confortável infra-estrutura de trabalho.

À minha família, Oscar, Vera Lúcia, Shirley e Wilkinson que, sem medir esforços nem sacrifícios, contribuiram de forma exemplar para minha formação.

Ao pessoal da arroZ por ter lido minhas pouco construtivas mails sobre as obras dos rinocerontes.

A Silvio Meira pelas lições de vida e por acreditar e investir na minha formação.

Obrigado a Julianne e a Denise pelas correções e críticas ao texto desta Dissertação.

Agradecimento especial ao povo brasileiro que consciente ou inconscientemente contribui, pagando seus impostos, para a pesquisa no Brasil.

Resumo

Este trabalho apresenta contribuições para a Engenharia de Software, especificamente para a área de gerenciamento e controle do processo de software desenvolvido utilizando a linguagem de programação orientada a objetos Java. O processo de desenvolvimento de software compreende todas as atividades realizadas durante as fases do ciclo de vida do software. É através do controle de cada uma destas atividades que se obtém um completo domínio, uma gerência eficaz e um planejamento adequado das ações tomadas, e a serem tomadas, durante todo o processo de desenvolvimento.

Modelos de gerência de processo de software, presentes na literatura, propõem-se a fornecer uma metodologia para garantir condições para que o desenvolvedor de software consiga avaliar e melhorar o controle do processo. Os mais consolidados baseiam-se numa estrutura composta de níveis de maturidade, nos quais requisitos têm que ser alcançados para se evoluir no modelo. Estes modelos fornecem uma estrutura com tabelas e pontos-chave que permitem ao desenvolvedor de software identificar os pontos críticos do processo e, assim, melhorar, disciplinar e otimizar as suas ações.

PSP, *Personal Software Process*, é um modelo de gerência de processo de software voltado ao indivíduo, e desenvolvido por Watts Humphrey e pelo SEI, *Software Engineering Institute*. PSP visa disciplinar o engenheiro de software como forma de garantir a disciplina do processo de software. O modelo PSP está estruturado na forma de um curso, no qual o indivíduo ou aluno segue os exercícios propostos, preenche tabelas e identifica seus pontos críticos através de cálculos estatísticos sobre os dados coletados durante o desenvolvimento dos programas-exercício.

PSP, pela sua generalidade, depende da linguagem, do paradigma de programação e dos recursos oferecidos pelo ambiente de programação que se está utilizando para a implementação dos programas. Este trabalho apresenta dados que tornam evidente esta dependência.

Para isto, foi utilizada a linguagem de programação orientada a objetos Java para a implementação dos programas-exercícios. Também, foi construído um editor (mini-ambiente de programação) para suportar a implementação dos exercícios e ilustrar como um ambiente, por mais simples que seja, influencia nos dados coletados através das tabelas PSP. Como consequência dos dados obtidos, foram redefinidas as tabelas utilizadas e criada a versão Java para PSP: PSP-JOA *Personal Software Process - A Java Oriented Approach*, principal resultado deste trabalho.

Abstract

This work presents contributions to the field of Software Engineering in the area of software process management, especially for softwares developed using the object-oriented programming language Java. The software development process includes all the activities carried out during the software life cycle phases. By managing each of these activities, it is possible to obtain a complete control, efficient management and a suitable planning of the decisions taken and the ones still to be performed during the development process.

A number of software process management models is described in the literature, proposing methodologies to guarantee good conditions for the software developer to evaluate and improve the software development process. The most well known of such models are based on a structure composed by maturity levels, in which requirements must be fullfilled to evolve in the model.Such models provide an structure with tables and key process areas allowing the software developer to identify the process critical points, and improve, discipline and optimize his actions.

PSP, *Personal Software Process*, is a software process management model developed for the software engineer by Watts Humphrey and SEI, *The Software Engineering Institute*. The goal of PSP is to discipline the software engineer to guarantee the discipline of the software process. PSP is structured as a course where the software engineer must follow the proposed sequence of exercises, fill in tables and identify their critical points through statistical calculations on data collected during the development of the programming exercises.

But PSP, which is a general process, depends on the language, programming paradigm and resources available in the programming environment being used to implement the programs. This work presents data that show evidence of this. Using the object-oriented programming language Java for the implementation of the programming exercises. We also created an editor (small programming environment) to support the implementation of the

exercises and show that an environment, even though a very simple one, influences the data collected to the PSP tables. As a consequence of the data acquired, the tables used were redefined and PSP-JOA: *Personal Software Process - A Java Oriented Approach*, a Java version for PSP, was created as the major result of this work.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

1.1 Noções Gerais de Engenharia de Software

Com a evolução do software, os problemas associados à construção e ao desenvolvimento de software se intensificaram e se tornaram mais complexos. Alguns fatores contribuiram para este aumento de complexidade: falta de metodologia na construção do software; capacitação inadequada dos desenvolvedores; manutenção dos programas ameaçada por projetos ruins e recursos inadequados; construção de hardware mais poderoso exigindo software que extraisse todo o seu potencial; estimativas de prazos e custos imprecisas; baixa produtividade dos profissionais desenvolvedores de softwares; qualidade de software inadequada. A engenharia de software fornece métodos e práticas em resposta a estes problemas.

A engenharia de software é constituída por uma tríplice formada de métodos, ferramentas e procedimentos [Pre95]. Os métodos são os modelos para a construção de software, as ferramentas fornecem apoio automatizado ou semi-automatizado aos métodos e os procedimentos definem a sequência de métodos a serem aplicados para a construção do software. Estes três componentes fornecem ao desenvolvedor uma estrutura básica para controlar e gerenciar o processo de desenvolvimento do software e para construir software de qualidade de forma eficiente.

1.1.1 Métodos

Os métodos utilizados nas fases de desenvolvimento de software são específicos e alguns dependem fortemente da tecnologia utilizada, pois são desenvolvidos voltados a satisfazer um determinado objetivo. Alguns métodos desenvolvidos para um projeto perdem totalmente o seu significado se aplicados a outro projeto.

O principal objetivo da engenharia de software é produzir software de alta qualidade a baixo custo [Jal92]. O custo pode ser calculado ao final do desenvolvimento ou estimado mediante métodos estatísticos consolidados, e.g., COCOMO [Boe81] e SLIM [KT85]. Estes métodos são bastante genéricos e fornecem recursos para avaliação quantitativa do processo de desenvolvimento de software.

1.1.2 Ferramentas

A utilização de ferramentas no processo de software mostra que ou utilizamos planejamento, processos adequados e ferramentas especializadas, ou sem o uso de ferramentas temos perda de produtividade, baixa qualidade e custo elevado nas fases de teste e manutenção do software [FH93].

As ferramentas possuem a função de automatizar e otimizar os recursos e a realização de tarefas. A automação possui dois componentes básicos:

- A mecanização das tarefas existentes; e
- A capacidade de realizar tarefas diferenciadamente.

O princípio básico da automação é o rigor e a disciplina de processo, como ilustrado na Figura 1.1 [FH93].

Dentre os aspectos de CASE, *Computer Aided Software Engineering*, as ferramentas são os mais discutidos, pois são mais visíveis e de maior custo. São ainda mais visíveis quando não respondem às necessidades do processo de software no qual estão inseridas. A falta de

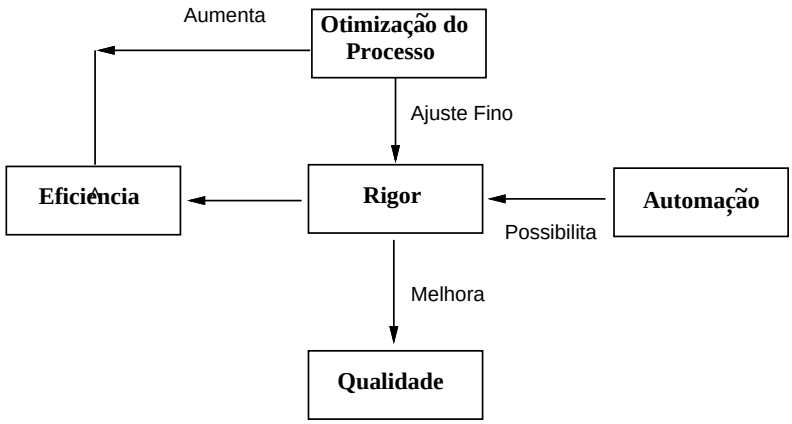

Figura 1.1: Princípios de Engenharia de Software

critérios na seleção de uma ferramenta é o principal fator para a utilização inadequada e ineficiente [Zar90].

Ferramentas podem suportar as diversas fases do ciclo de vida do software, e são classificadas de acordo com a sua funcionalidade. Em [Som92], as ferramentas estão classificadas, ortogonalmente, em dois critérios: orientadas à atividade e orientadas à funcionalidade. Esta classificação pode ser melhor observada na Figura 1.2.

Os benefícios alcançados pelo uso de ferramentas são inúmeros, entre eles, destacamos, aumento da capacidade de produção, automação de processos não-criativos, controle das atividades do processo e controle da capacidade de repetição de atividades iguais.

Algumas desvantagens no uso de ferramentas são provenientes de fatores como uso inadequado, seleção inadequada, forte dependência do fabricante, resistência às mudanças de processo provocadas pelo uso de ferramentas, dificuldade de integração das ferramentas e uso de ferramentas mal projetadas e mal desenvolvidas. Estas desvantagens provocam descrédito nos processos automáticos ou semi-automáticos e desperdício de tempo com atividades que seriam facilmente realizadas sem o auxílio de ferramentas.

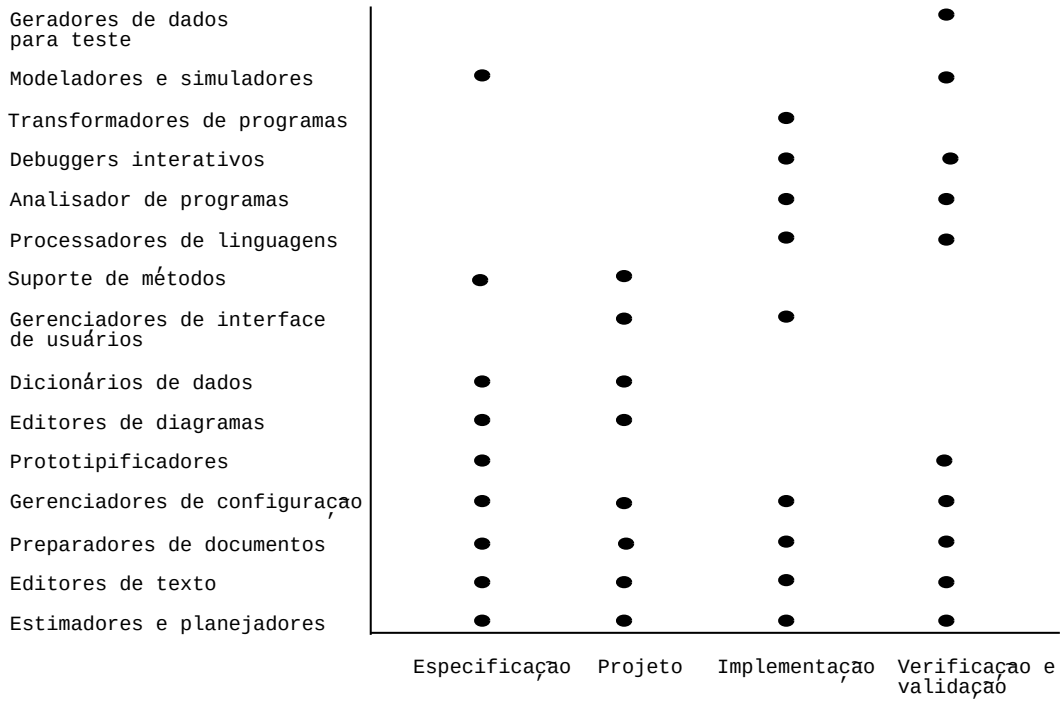

Figura 1.2: Classificação de Ferramentas Vs. Atividades de Processo Suportadas

1.1.3 Procedimentos

Os procedimentos garantem sincronismo e integração entre a utilização de ferramentas e os métodos adotados no processo de software. Em qualquer projeto de desenvolvimento é necessário selecionar ou definir o processo de software, tanto do ponto-de-vista técnico quanto gerencial. Do ponto-de-vista técnico, as mais importantes decisões e definições são a escolha das fases que irão compor o ciclo de vida do software e a escolha dos métodos que serão executados em cada fase.

As fases do ciclo de vida do software já estão praticamente consolidadas e sofrem poucas variações entre os projetos de desenvolvimento (análise de requisitos, especificação do sistema, projeto da arquitetura, projeto detalhado, implementação e integração, manutenção e evolução). Contudo, algumas aplicações específicas exigem fases diferenciadas e definidas visando as características particulares destas aplicações.

Devido ao aspecto técnico, os métodos a serem executados por fase exigem um esforço maior para serem definidos. Para a seleção dos métodos, são considerados critérios que incluem maturidade, nível de suporte de ferramenta, tamanho da aplicação, formação do time de desenvolvimento, tipo de aplicação, entre outros.

Depois de formado o contexto com os critérios citados, é escolhida a sequência de métodos que irá suportar as fases do ciclo de vida do software, constituindo o que chamamos de "modelo de processo de desenvolvimento de software".

A qualidade não é uma grandeza de fácil mensurabilidade, contudo, os modelos de processo de desenvolvimento fornecem uma representação abstrata para assegurar um mínimo de controle e garantia de qualidade do software a ser produzido. Alguns destes modelos estão descritos, sumariamente, a seguir [Jal92, Som92]:

Cascata. Neste modelo, as fases do ciclo de vida do software são organizadas de maneira linear. Uma fase só é iniciada quando a anterior for concluída. A sequência de atividades segue a seguinte ordem: análise de requisitos, planejamento de projeto, projeto do sistema, projeto detalhado, codificação e teste de módulos, integração do sistema e teste.

Prototipação. Um protótipo é desenvolvido para validar a análise de requisitos realizada. Com o protótipo satisfazendo todos os requisitos desejados, segue-se, então, a sequência de projeto, codificação e teste. Geralmente, o protótipo não é aproveitado nas demais fases do modelo.

Iterativo. Os modelos prototipação e cascata são combinados de modo que o software é produzido de maneira incremental. Um núcleo-base do sistema é construído e, então, são adicionadas as funcionalidades desejadas através da incorporação de módulos que são desenvolvidos seguindo o modelo cascata. Segue-se assim, até que o sistema esteja completamente implementado.

Espiral. Como o nome sugere, a sequência de atividades está organizada, ciclicamente, como numa espiral. O passo da espiral representa o custo por ciclo e a distância radial,

o custo acumulado. A distância angular representa a evolução das atividades a cada ciclo da espiral e os quadrantes representam as atividades realizadas, respectivamente:

1. Determinação de objetivos, alternativas de solução e restrições;
2. Avaliação das alternativas, identificação e resolução dos riscos;
3. Desenvolvimento: projeto, codificação, integração e teste. Esta etapa pode ser realizada seguindo quaisquer outros modelos, inclusive, permite um modelo diferente a cada ciclo;
4. Planejamento das próximas fases.

Programação Exploratória. É semelhante ao modelo Iterativo, entretanto, o sistema é completamente definido, rapidamente implementado, e exaustivamente modificado, até que atenda aos requisitos definidos.

Transformação Formal. Parte-se de uma especificação formal do sistema e realiza-se finitas transformações no sentido de refiná-la a um código concreto, implementável. Isto é realizado de forma que a corretude é preservada a cada transformação.

Programação através do uso de componentes reutilizáveis. Este modelo assume que o sistema está, na sua maioria, implementado em componentes anteriormente desenvolvidos. O desenvolvimento do sistema consiste, então, em: agrupar estes componentes já implementados, desenvolver os não existentes, integrar e testar todo o sistema.

Vários modelos de gerenciamento se consolidaram entre as empresas desenvolvedoras de software. Modelos que além de garantir os aspectos técnicos e gerenciais presentes nos modelos de processo, fornecem controle e garantia de qualidade do software. Alguns destes modelos de gerenciamento de software são descritos no Capítulo 2.

1.2 Noções de Qualidade

A crise da qualidade do software, caracterizada pela falta de planejamento e de recursos para controle e garantia da qualidade dos softwares, obrigou empresas desenvolvedoras de softwares a priorizarem investimentos em mecanismos de qualidade.

Com o surgimento de grupos de pesquisa, institutos e organizações voltadas exclusivamente para o estudo dos processos de qualidade, surgiram vários modelos e técnicas fornecendo um suporte à engenharia de software nesta direção [Sys95]. Alguns destes modelos são descritos no Capítulo 2.

A qualidade na engenharia de software é composta de três aspectos: qualidade do produto, qualidade do processo de desenvolvimento e qualidade do time de desenvolvimento.

1.2.1 Qualidade de Software

Definir a qualidade de um software é uma tarefa difícil. Contudo, qualidade pode ser definida como composta por fatores internos e externos ao software; minimamente, como o conjunto de quais requisitos são exigidos no software.

Os requisitos são classificados em funcionais e não-funcionais [McD94]. Os requisitos funcionais são os aplicados a trechos do software, a módulos, ou ao sistema completo (e.g., todo dado deverá ser armazenado em disco antes de qualquer transação ser efetuada) e representam "o que fazer". Os não-funcionais são aplicados a quaisquer produtos do processo de desenvolvimento: especificações, código, manuais, etc. (e.g., o sistema deverá ser capaz de operar numa máquina com um mega de memória) e significam "como fazer".

Entretanto, a definição dos requisitos deve ser consistente e concreta. Requisitos abstratos como robustez, extensibilidade, etc., não fornecem parâmetros mensuráveis para suas avaliações e verificações no software desenvolvido ou em desenvolvimento. Os requisitos para a verificação da qualidade do software devem ser precisos, de preferência utilizando valores numéricos, e.g., o tempo de resposta de qualquer consulta deve ser inferior a três milisegun-

dos. Desta forma, os requisitos podem ser facilmente avaliados, garantindo a qualidade de corretude desejada. Contudo, a definição de tais requisitos não é simples, e muitas vezes deixa a desejar por tentar discretizar grandezas abstratas e contínuas.

Com os requisitos definidos, o desenvolvedor de software conhece quantitativamente que nível de qualidade deseja alcançar no seu software e precisa estruturar métodos e técnicas para garantir e controlar esta qualidade desejada.

1.2.2 Qualidade do Processo

O desenvolvimento de software vem se tornando cada vez mais complexo. A quantidade de recursos disponíveis, a complexidade dos sistemas desenvolvidos e a exigência por altos padrões de qualidade são alguns dos fatores que contribuem para este aumento de complexidade. A tarefa de desenvolver tais softwares exige um processo muito bem definido, pois, caso contrário, corremos o risco de provocarmos um completo caos.

Os processos, na engenharia de software, não são fixos nem rotineiros; variam de projeto para projeto e, por isso, a complexidade em se defini-los a contento. Contudo, modelos bastante abstratos estão disponíveis na literatura e fornecem uma estrutura básica para, a partir deles, definirmos nosso próprio processo de software.

Uma vez definido o processo, há a necessidade de garantir que este seja seguido e executado. Para isso, é necessário disciplina do time de desenvolvimento, supervisão e acompanhamento do processo, além, claro, de padrões de qualidade a serem seguidos a cada fase. Só assim, a qualidade de todo processo é garantida.

1.2.3 Qualidade Individual

A qualidade individual é um fator bastante complexo para avaliação e depende de quase infinitos aspectos: formação do profissional, satisfação profissional, estado de saúde do indivíduo, estado emocional do indivíduo, etc. Podemos listar inúmeros fatores que influenciam

na qualidade do indivíduo e na qualidade do serviço oferecido por este indivíduo. Contudo, fatores de ordem psicológica, financeira, emocional, etc. não estão no escopo deste trabalho.

Tornando os fatores a serem avaliados mais mensuráveis, podemos avaliar o código desenvolvido pelo indivíduo e assim avaliar sua qualidade. A avaliação do programador não fornece uma avaliação ideal, mas contribui de forma significativa para a avaliação do engenheiro.

Durante o desenvolvimento de sistemas de grande porte, observa-se que ao término do projeto normalmente não existe um histórico relatando erros, tempo efetivamente consumido de programação, taxa de homens/hora e outros fatores. Existe clara evidência [Jur88] que todos esses fatores são úteis quando se deseja avaliar a relação custo versus benefício do processo de desenvolvimento de software.

Uma análise desses fatores demonstra que a grande maioria dos problemas são causados por erros individuais dos programadores envolvidos. [KM88] mostra que, quase invariavelmente, os programadores desconhecem métricas e simples indicadores sobre sua própria performance, como as linhas de código que são capazes de produzir em determinado tempo ou quantos erros, em média, cometem por linha de código.

Se cada programador possuísse dados sobre sua performance em diversos projetos, de diversos graus de complexidade e de conhecimento do domínio, os resultados seriam melhorados. Além de se conhecer melhor, o programador poderia planejar e controlar o tempo necessário para desenvolvimento de uma determinada tarefa, melhorando, assim, sua maneira de trabalhar.

Os benefícios desta melhoria individual se refletem de modo positivo nos resultados de desempenho do grupo no qual se trabalha. Existe pouca literatura e investigação no que se refere a determinar quais os ganhos reais para o grupo, a partir desta melhoria individual de performance.

1.3 Conteúdo dos Capítulos

Este trabalho possui seis capítulos e três apêndices. Os temas estão dispostos de forma a compor incrementalmente um contexto mínimo para o perfeito entendimento e avaliação de PSP-JOA *Personal Software Process - A Java Oriented Approach*, apresentado no Capítulo 4.

O Capítulo 2 apresenta uma descrição geral dos modelos de gerenciamento de processo de software mais comumente utilizados. Os modelos descritos são CMM-*Capability Maturity Model*, SPICE-*Software Process Improvement and Capability dEtermination*, Trillium e ISO9000-*International Organization for Standardization*. Este capítulo contém sugestões sobre como fazer uma seleção entre os modelos disponíveis e considerações sobre como os modelos apresentados estão voltados à organização e não ao indivíduo. Estas considerações são a motivação para o desenvolvimento dos modelos apresentados no Capítulo 3.

O Capítulo 3 possui as descrições detalhadas de modelos para gerenciamento pessoal de software. Os modelos analisados foram P-CMM, *People - Capability Maturity Model*, e PSP, *Personal Software Process*. Neste Capítulo, também são apontados alguns pontos críticos encontrados nestes modelos. Um estudo sobre PSP foi realizado e algumas restrições são apresentadas, as quais são as motivações para a elaboração de PSP-JOA.

O Capítulo 4 apresenta a definição de PSP-JOA, uma versão definida única e exclusivamente voltada ao processo de softwares desenvolvidos utilizando a linguagem orientada a objetos Java. Neste capítulo, são apresentadas as tabelas modificadas de PSP e uma descrição geral da linguagem de programação Java.

Algumas sugestões para trabalhos futuros e as conclusões são encontradas no Capítulo 5.

Os Apêndices A,B e C contém, respectivamente:

- Os enunciados e soluções dos exercícios PSP realizados;
- JavaEdit: Um micro-ambiente de programação para desenvolvimento de código Java.

JavaEdit é um editor de propósito geral com formatação de texto Java e um contador de linhas de código;

- As tabelas PSP-JOA geradas a partir de modificações nas tabelas PSP são apresentadas neste apêndice.

Capítulo 2

Gerenciamento do Processo de Software

2.1 Introdução

Existem vários modelos de referência definidos para garantir um processo de desenvolvimento de software satisfatório. Todos estes modelos surgiram da necessidade de melhorar e controlar o processo de software nas organizações. O aumento da complexidade dos sistemas desenvolvidos tornou praticamente impossível o controle do processo de software sem o conjunto de métodos, técnicas e guias fornecido por tais modelos.

Os modelos de referência não são as soluções para todos os problemas apontados pela engenharia de software, mas fornecem um conjunto de definições e procedimentos que resolvem uma grande parte dos problemas comumente encontrados no desenvolvimento de sistemas. Estes problemas são provocados principalmente pela indefinição do processo de software, provocando um desenvolvimento completamente caótico do ponto de vista gerencial.

As organizações que funcionam sem um modelo para o processo de desenvolvimento de software sobrevivem graças ao esforço individual de alguns "heróis" presentes no seu time

de desenvolvimento. Por exemplo, não tendo um processo de software bastante definido, para repetir uma tarefa anteriormente realizada é necessário o mesmo time de desenvolvedores trabalhando nesta nova tarefa. Isto torna o risco envolvido em repetir uma tarefa previamente realizada, que seria próximo de zero com um processo bem definido, muito alto.

A adoção de modelos de gerenciamento de processo de software tornou-se item obrigatório se a organização deseja alcançar padrões elevados de qualidade, tanto do processo quanto do produto como consequência da qualidade do processo.

2.2 Processo de Software

O desenvolvimento de software alcançou um nível tal de complexidade que os métodos artesanais de construção e implementação não satisfazem mais os requisitos organizacionais e operacionais exigidos. Para se ter um perfeito controle e uma gerência eficaz do desenvolvimento de software é preciso se ter um processo de software muito bem definido.

O processo de software são todas as atividades realizadas durante as fases do ciclo de vida do software. É através do controle de cada uma destas atividades que se obtém um completo domínio, uma gerência eficaz e um planejamento adequado das ações tomadas, e a serem tomadas, durante todo o processo de desenvolvimento. A definição do processo de software é a descrição dessas atividades.

Contudo, definir um processo de software não é uma tarefa fácil. As pessoas oferecem uma resistência natural a mudanças, e implantar um novo processo de software numa organização demanda custos e tempo. Há alguns conceitos errados que favorecem a resistência à adoção de um novo processo [Hum90]:

"Só começamos com os requisitos totalmente definidos". Existe uma falsa cultura de que análise de requisitos é tarefa do cliente e que o desenvolvimento só deve começar com todos os requisitos definidos.

"Se passa nos testes, então está OK". Esta é uma visão minimalista dos requisitos de

qualidade a serem satisfeitos.

"Qualidade de software não pode ser medida". Até que se prove o contrário, esta afirmação é errada. O que existe são parâmetros difíceis de serem medidos, mas alguns outros já estão bastante consolidados [Gra92].

"Os problemas são de ordem técnica". Existem muitos problemas técnicos devido a ferramentas e linguagens de programação, mas a questão é: o processo de software não tem a sua parcela de culpa nestes problemas? Por exemplo, atualização de versões das ferramentas não comunicada ao corpo técnico.

"Precisamos de funcionários melhores". Esta é uma frase bastante comum nos meios empresariais, contudo tem-se mostrado que o problema é do processo e não das pessoas [Hum96].

Os modelos de referência para processo de software são guias para avaliação, qualificação e melhor definição do processo de software.

2.3 Descrição de Modelos

Os mais conhecidos modelos de referência para gerenciamento de processo de software são descritos nesta seção. Alguns dos modelos são específicos para software, outros como ISO9000 e CMM, *Capability Maturity Model*, são aplicados aos mais variados tipos de produtos, pois visam o processo da organização.

As seções apresentam as descrições gerais dos modelos CMM, SPICE, ISO9000 e Trillium visando o processo de software. Também são apresentados outros modelos de menor repercursão no meio empresarial.

2.3.1 CMM para Software

"Um processo de evolução baseado em níveis de maturidade com metas a serem alcançadas a cada nível", é a melhor definição para o CMM [PCCW93].

Um nível de maturidade, no CMM para software, é um platô bem definido a ser alcançado durante o processo de maturidade de software. Cada nível de maturidade alcançado compõe uma camada a ser acrescentada ao processo de evolução do software. Um nível de maturidade possui um conjunto de objetivos bem definidos que, quando satisfeitos, estabilizam um dos componentes do processo de desenvolvimento do software. Alcançando cada nível de maturidade estabelecido no processo de desenvolvimento do software, está-se melhorando o processo de capacitação da organização como um todo.

CMM é organizado em cinco níveis, como mostrado na Figura 2.1.

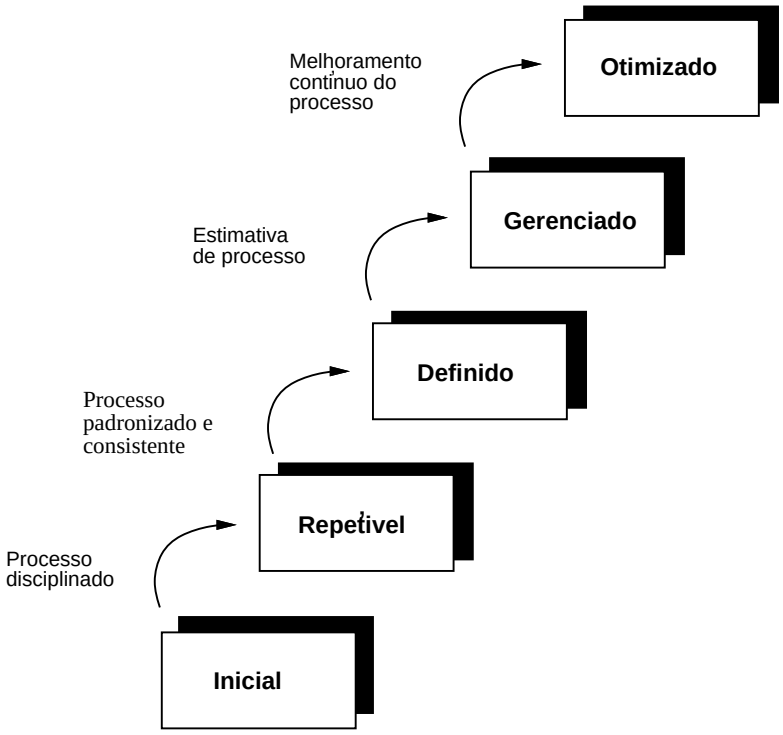

Figura 2.1: Níveis de Maturidade do Processo de Software

As características presentes na organização que se classifica em um destes níveis são:

Inicial. O processo do software é desorganizado, caótico, e depende do esforço individual dos membros da equipe de desenvolvimento. Neste nível, a organização quando apresenta um processo, ele é instável e imprevisível, pois muda constantemente com a urgência nos prazos de entrega do software. Em tempos de crise, as fases de desenvolvimento e planejamento são abandonadas e o ciclo de vida do software se resume a implementação e teste.

Repetível. O básico do processo de gerenciamento do desenvolvimento do software está estabelecido, ou seja, a instituição é capaz de repetir com sucesso uma aplicação similar a uma já desenvolvida anteriormente. Para alcançar este nível a organização deve possuir um histórico mínimo das atividades realizadas, no qual as tarefas são registradas de modo a fornecer dados para um planejamento estimativo sobre atividades semelhantes que poderão ser realizadas *a posteriori*. Desta forma, o processo está disciplinado.

Definido. O processo de gerenciamento e das atividades de engenharia está documentado, padronizado e integrado ao processo de desenvolvimento do software da instituição. Apesar de seguir o CMM, a organização é capaz de identificar o seu próprio processo de software, o qual possui características próprias do projeto que se está desenvolvendo.

Gerenciado. Medições detalhadas do processo de desenvolvimento do software e de qualidade do produto estão sendo realizadas de forma a controlar e entender quantitativamente o processo de desenvolvimento da instituição. Assim, a organização pode definir que objetivos pretende alcançar a nível de qualidade, tanto do processo quanto do produto. O processo, neste nível, pode ser considerado previsível, pois a organização consegue medir os seus limites e, com isto, melhor estimar e planejar suas ações.

Otimizado. Melhorias contínuas no processo de desenvolvimento do software estão sendo realizadas. De posse dos dados obtidos durante os seus projetos, a organização pode melhorar e ampliar os seus limites, otimizando seus objetivos em relação às suas potencialidades.

Exceto pelo nível um, cada nível é composto por KPA's *Key Process Areas* [PWG+93], que indicam as áreas na organização que devem ter seus processos melhorados. Cada KPA é descrita através de práticas-chave *Key Practices*, que descrevem especificamente quais as atividades que devem ser melhoradas.

As KPA's estão distribuídas entre os níveis de maturidade da seguinte forma:

- **Repetível:**
  Gerenciamento de Requisitos
  Planejamento de Projeto de Software
  Acompanhamento e Supervisão de Projeto de Software
  Gerenciamento de Sub-contrato de Software
  Garantia de Qualidade de Software
  Gerenciamento de Configuração de Software

- **Definido:**
  Focalização do Processo da organização
  Definição do Processo da organização
  Programa de Treinamento
  Gerenciamento Integrado de Software
  Engenharia de Produto de Software
  Coordenação entre os Grupos
  Revisões Constantes

- **Gerenciado:**
  Gerenciamento Quantitativo de Processo
  Gerenciamento Qualitativo de Software

- **Otimizado:**
  Prevenção de Defeitos
  Gerenciamento de Mudança de Tecnologia
  Gerenciamento de Mudança de Processo

Algumas das práticas destas áreas variam muito pouco de processo para processo, contudo precisam ser identificadas e definidas a cada novo processo.

Entretanto, CMM não resolve todos os problemas de engenharia de software [dTdS96]. Por exemplo, CMM é muito deficiente no que diz respeito a que técnicas utilizar em cada fase ou em cada nível de maturidade. Ele define que metas alcançar, mas nada diz com respeito a como alcançar, ou seja, fornece apenas uma estrutura conceitual.

2.3.2 SPICE

SPICE, *Software Process Improvement and Capability dEtermination*, é um projeto da ISO/IEC *International Organization for Standardisation/International Electrotechnical Commission* JTC1/SC7/WG10 [fS96, Com96, JTC95] *Joint Technical Commitee/software engineering/process assessment* e desenvolve um padrão para avaliação de processo de software, visando melhoria contínua e determinação de capacitação. SPICE [Lig96] faz planejamento, gerenciamento, execução, controle e melhoria de aquisição, fornecimento, desenvolvimento, operação, manutenção e suporte de software.

O projeto SPICE, depois de dois anos de estudos, foi definido em três etapas:

- Produzir guias de trabalho em direção à padronização para a avaliação de processo de software;
- Encarregar-se de, através de resultados coletados com testes de usuários com o uso dos guias, definir o padrão completo para avaliação de processo de software;
- Formar uma consciência no mercado sobre a padronização da avaliação de processo de software e consolidar a aceitação deste padrão.

O projeto ainda não está consolidado [Dor96, Dor97], mas já possui o conjunto de guias definidos que estão sendo utilizados provisoriamente de forma a obter informações e experiência de seu uso prático. O conjunto de guias é composto de nove partes, descritas a seguir e ilustradas na Figura 2.2:

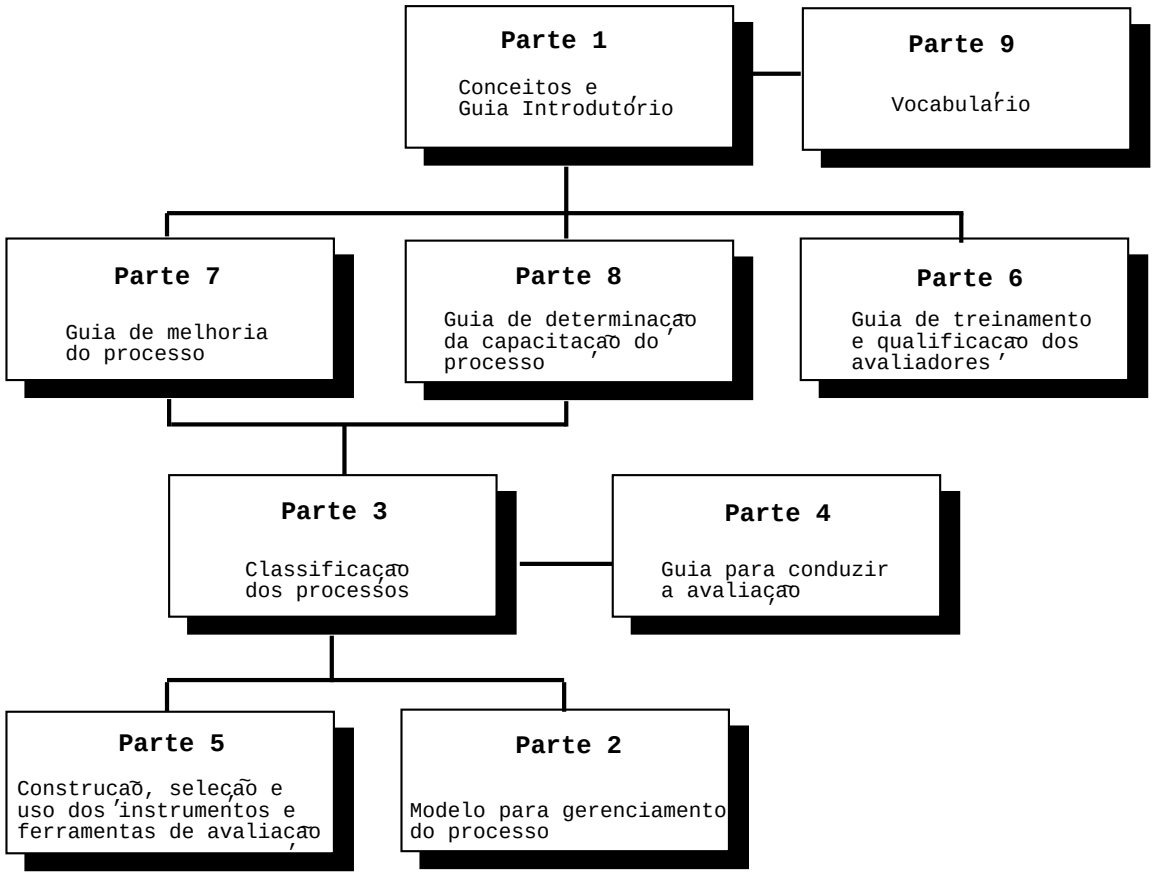

Figura 2.2: Componentes do Padrão SPICE

Parte 1. Nesta parte, é descrito como as demais se relacionam e fornece sugestões para seleção e uso delas, fornecendo uma explicação dos requisitos contidos no padrão e sua aplicabilidade na avaliação.

Parte 2. A Parte 2 deste padrão define, em linha gerais, as atividades que são essenciais para a engenharia de software. É estruturada seguindo o modelo de níveis de maturidade do processo.

Parte3. Na parte 3, é definida uma estrutura para a realização de uma avaliação mais concreta, fornecendo uma base para taxas, valores e medições de maturidade de processo. A Parte 3 pode ser considerada como a concretização do guia fornecido pela Parte 4,

ou ainda, como definido seguindo as linhas traçadas pela Parte 4.

Parte 4. Esta parte fornece um guia para realização de uma avaliação. O guia é bastante genérico e pode ser utilizado em diversas organizações, não só de software. O guia utiliza as mais variadas técnicas e métodos.

Parte 5. A parte 5 deste padrão define elementos numa estrutura para construção de instrumentos e ferramentas para auxiliar o avaliador durante a avaliação de processo. Além de fornecer, também, um guia para seleção e utilização de aspectos desejados nos vários tipos de ferramentas e instrumentos.

Parte 6. A parte 6 descreve o treinamento e a experiência exigida aos avaliadores para conduzirem uma avaliação competente. Fornece mecanismos que podem ser utilizados para demonstrar a competência, a qualificação e o conhecimento do avaliador.

Parte 7. A parte 7 descreve como definir a entrada e o uso dos resultados de uma avaliação para a melhoria e otimização do processo.

Parte 8. Nesta parte, é descrito como definir a entrada e o uso dos resultados de uma avaliação para a determinação de maturidade do processo. Este guia se propõe tanto a fornecer uma estrutura para determinar a maturidade da organização que está sendo avaliada quanto a de seus fornecedores.

Parte 9. A parte 9 possui um vocabulário com todos os termos especificamente definidos para este padrão.

Estes guias estão em constante evolução, à medida que mais resultados são obtidos e novos parâmetros detectados, novas versões são criadas. Alguns resultados [EG96, mS97] têm sido bastante otimistas e encorajadores para os pesquisadores. Contudo, aguarda-se pela completa conclusão do padrão para uma completa e eficaz avaliação de SPICE.

2.3.3 ISO 9000

ISO, *International Organization for Standardization*, [ISO96, Mat96] é uma entidade não-governamental destinada a promover o desenvolvimento da padronização e das atividades relacionadas com a criação de padrões.

Padrões são acordos documentados contendo especificações técnicas e outros critérios precisos [fS96]. Estes padrões são usados consistentemente como regras ou definições de características para garantir que materiais, produtos, processos e serviços satisfaçam os propósitos desejados. Os padrões internacionais são definidos com o propósito de facilitar nossas vidas e aumentar a confiabilidade e a eficácia dos bons produtos e serviços.

A compatibilidade proporcionada por padrões internacionais é de suprema importância para o desenvolvimento mundial, tornando as fronteiras físicas e burocrátricas simples demarcações territoriais. A interoperabilidade entre produtos e serviços de diversos países que seguem determinado padrão contribui significantemente para o progresso de variados setores destes países como economia, ciência, comunicação, liberação comercial e qualidade de vida. Alguns exemplos de importantes padronizações: SI - Sistema Internacional de pesos e medidas, tamanhos de papel (e.g., A4, Carta, Ofício), símbolos de trânsito, tamanho dos cartões de crédito (e.g, espessura ideal de 0,76mm). Ao todo são 9.300 padrões.

ISO não trabalha sozinha na atividade de padronização internacional. IEC, *International Eletrotechnical Commission*, é responsável pelas áreas que dizem respeito às engenharias elétrica e eletrônica, as demais áreas são de responsabilidade da ISO. É claro que em determinadas áreas há uma cooperação mútua.

ISO 9000 é um conjunto de cinco padrões para garantia de qualidade de sistemas. O Brasil é um dos 90 países que adotam ISO 9000 como padrão nacional.

ISO 9001 é o padrão relacionado a desenvolvimento e manutenção de sistemas em geral. ISO 9000-3 são as orientações para aplicação do ISO 9001 ao software, ou seja, fornecendo um padrão para desenvolvimento, fornecimento e manutenção de software.

A Tabela a seguir contém a descrição sumária dos padrões que constituem a família

ISO 9000.

Padrão	**Ano**	**Descrição**
ISO 9000-1	1994	Padrão para garantia e gerenciamento da qualidade - Parte 1: uso e seleção.
ISO 9000-2	1993	Padrão para garantia e gerenciamento da qualidade - Parte 2: linhas gerais para aplicação de ISO 9001, ISO 9002 e ISO 9003.
ISO 9000-3	1991	Padrão para garantia e gerenciamento da qualidade - Parte 3: aplicação de ISO 9001 a desenvolvimento, fornecimento, e manutenção de software.
ISO 9000-4	1993	Padrão para garantia e gerenciamento da qualidade - Parte 4: confiabilidade do gerenciamento.
ISO 9001	1994	Modelo para garantia de qualidade em projeto, desenvolvimento, produção, instalação e prestação de serviço.
ISO 9002	1994	Modelo para garantia de qualidade em produção, instalação e prestação de serviço.
ISO 9003	1993	Modelo para garantia de qualidade em inspeção final e teste.
ISO 10011-1	1990	Guia de orientação para auditoria em sistemas de qualidade. Parte 1: auditoria.
ISO 10011-2	1991	Guia de orientação para auditoria em sistemas de qualidade. Parte 2: critérios de qualificação para auditores de sistemas de qualidade.
ISO 10011-3	1991	Guia de orientação para auditoria em sistemas de qualidade. Parte 3: gerenciamento de programas de auditoria.

Família ISO 9000 - Continua ...

Padrão	Ano	Descrição
ISO 10012-1	1992	Requisitos de garantia de qualidade para equipamentos de medição. Parte 1: sistema de confirmação de medidas para equipamentos de medição.
ISO 10013	-	Guia de orientação para desenvolvimento de manuais de qualidade.
ISO/TR 13425	-	Guia de orientação para seleção de métodos estatísticos na padronização e especificação.
ISO 8402	1994	Vocabulário para gerenciamento e garantia de qualidade.

Família ISO 9000

ISO 9000, mais especificamente ISO 9001, determina o conjunto mínimo de requisitos para se ter um sistema de qualidade, enquanto que modelos como SPICE e CMM visam a melhoria contínua de qualidade de processo. Esta é a principal diferença entre estes modelos.

Uma organização que tem o certificado ISO 9001 pode ser classificada no modelo CMM como estando no nível 3 ou 4. Existem casos de organizações com certificado ISO 9001 e que não passam do nível 1 (inicial) na avaliação CMM. Um dos principais motivos para esta discordância é o nível de abstração com que é definido o padrão ISO 9001, dando margens a diferentes interpretações por parte dos auditores e, consequentemente, diferentes critérios de avaliação.

A conversão CMM para ISO 9000 não é bi-direcional. CMM, apesar de conceitual, utiliza dados mais concretos que as definições abstratas do ISO 9001. Isto possibilita que uma organização no nível 3 CMM obtenha com certeza o certificado ISO 9001, mas o contrário nem sempre é verdade, chegando a casos que nem o nível dois seria alcançado. [Pau95] possui mais detalhes sobre esta comparação.

2.3.4 Trillium

Trillium é um modelo de domínio público, baseado em níveis de maturidade e utilizado pela Bell Canada para avaliar principalmente o desenvolvimento de produtos de telecomunicações e produtos baseados em tecnologia de informação.

O modelo se torna mais efetivo se utilizado em todos os níveis da organização, envolvendo todos os departamentos, pessoal e suporte. Trillium é utilizado com vários objetivos, de quatro diferentes modos:

Avaliação de Maturidade. O objetivo é avaliar o desenvolvimento do produto e a maturidade dos fornecedores, avaliando os riscos associados e monitorando o programa de melhoria de maturidade e qualidade.

Avaliação Conjunta de Maturidade. A avaliação do desenvolvimento de produtos na organização é realizada em consenso com os fornecedores, trocando experiências e informações úteis às duas partes.

Auto-avaliação de Maturidade. Para auto-avaliação é requerido da organização um compromisso em formar um comitê interno para avaliação: sério, capacitado e respeitado por todos os membros da organização.

Melhoria Contínua. Este modo tem como objetivo tornar a filosofia de melhoria contínua uma cultura na organização, garantindo processos e mecanismos apropriados para encorajar esta cultura. Definir e otimizar os processos para alcançar os requisitos desejados, também fazem parte do conjunto de metas deste modo de utilização do modelo.

A estrutura e as práticas são baseadas principalmente no SEI/CMM V1.1. Também são utilizados os padrões ISO 9001 e ISO 9000-3.

As principais diferenças entre Trillium e CMM dizem respeito à arquitetura baseada em guias ao invés de áreas-chave, e à perspectiva voltada ao produto mais presente do que a voltada ao processo. Os níveis de evolução Trillium, ilustrados na Figura 2.3, são caracterizados por:

Não-estruturado. O processo de desenvolvimento é caótico e baseado no esforço individual ao invés de ser fundamentado na infra-estrutura da organização. (Alto-risco)

Repetível e orientado a projeto. Os projetos individuais são baseados num rigoroso controle e planejamento de projeto, com ênfase em gerenciamento de requisitos, técnicas de estimativas e gerenciamento de configuração. (Médio-risco)

Definido e orientado a processo. Os processos são definidos e utilizados a nível da organização. Neste nível é ainda permitida uma adaptação individual ao desenvolvimento de projeto, mas mantendo o processo da organização como meta principal. Processos são controlados e melhorados. (Baixo-risco)

Gerenciado e integrado. Análise e utilização de ferramentas CASE, *Computer Aided Software Engineering*, no processo são os mecanismos-chave para a melhoria do processo. (Risco muito reduzido)

Completamente integrado. Utilização extensiva de metodologias formais. Histórico de desenvolvimentos e de dados de processo são utilizados. (O menor risco possível)

Para alcançar um dos níveis de Trillium, a organização deve satisfazer, no mínimo, a 90% dos critérios das 8 áreas de maturidade em cada nível. Trillium é formado por áreas de maturidade, guias e práticas. Cada área de maturidade é composta por um ou mais guias. Um guia é um conjunto de práticas relacionadas à área de maturidade a que estão associadas.

O conjunto de práticas de Trillium é definido utilizando-se as práticas de outros modelos e acrescentando algumas específicas. Todas as práticas do SEI/CMM V1.1 são incorporadas. As práticas de ISO 9001, ISO 9000-3 e de outros modelos sofrem pequenas adaptações e, também, são incorporadas.

As áreas de maturidade com os seus respectivos guias estão listados a seguir. Para mais detalhes, consultar [Can94, Coa94, Coa95].

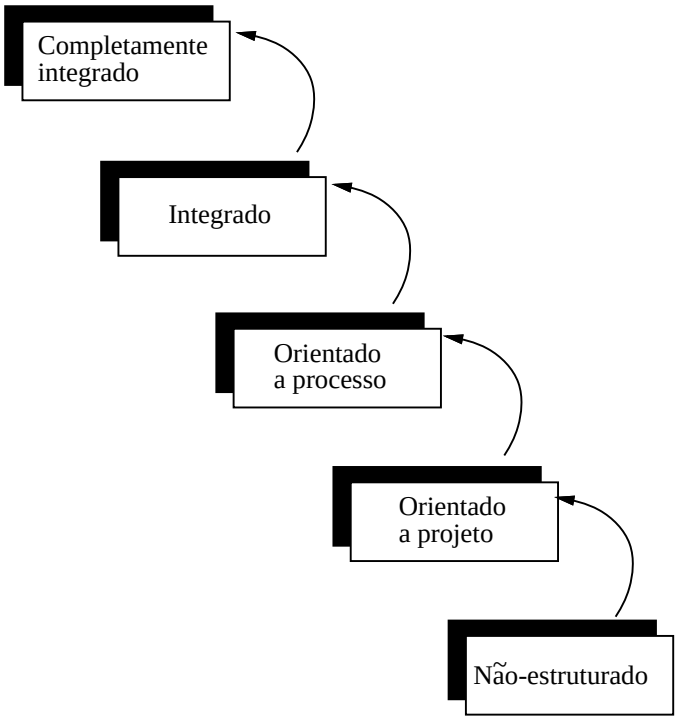

Figura 2.3: Estrutura de Trillium

- **Qualidade de Processo da Organização:**
  Gerenciamento da Qualidade
  Engenharia do Processo Comercial
- **Gerenciamento e Desenvolvimento de Recursos Humanos:**
  Gerenciamento e Desenvolvimento de Recursos Humanos
- **Processo:**
  Definição de Processo
  Gerenciamento de Tecnologia
  Engenharia e Melhoria de Processo
  Medidas
- **Gerenciamento:**
  Gerenciamento de Projeto

Gerenciamento de Sub-contrato
Relacionamento Cliente-Fornecedor
Gerenciamento de Requisitos
Estimativa

- **Sistema de Qualidade:**
  Sistema de Qualidade

- **Práticas de Desenvolvimento:**
  Processo de Desenvolvimento
  Técnicas de Desenvolvimento
  Documentação Interna
  Validação e Verificação
  Gerenciamento de Configuração
  Reusabilidade
  Gerenciamento de Confiabilidade

- **Ambiente de Desenvolvimento:**
  Ambiente de Desenvolvimento

- **Suporte ao Cliente:**
  Sistema de Resposta a Problemas
  Engenharia de Usabilidade
  Modelagem de Custo de Ciclo de Vida
  Documentação de Usuário
  Engenharia de Cliente
  Treinamento de Usuário

2.3.5 Outros Modelos

Nesta seção, apresentamos alguns modelos que não são tão difundidos no meio empresarial como os citados anteriormente, mas que possuem suas características e definições voltadas ao gerenciamento de processo de software. Entre os modelos menos conhecidos e utilizados comercialmente, podemos citar:

Bootstrap [Ins96]. *Bootstrap Institute* tem como objetivos principais:

- Explorar e definir novos e mais poderosos paradigmas para trabalho cooperativo e melhoria organizacional;
- Promover discussões sobre tópicos organizacionais e de aspecto técnico;
- Compartilhar estratégias, requisitos, práticas e dados obtidos durante a execução de modelos de gerência de processo.

Kaizen. É uma estratégia japonesa para melhoria contínua [Cus91]. Seus princípios são:

- Recursos humanos são os bens mais importantes da organização;
- Processos devem sofrer melhorias contínuas e progressivas, e não mudanças bruscas e radicais;
- Melhoria de processo deve ser baseada na avaliação estatística e quantitativa da performance do processo.

QIP, *Quality Improvement Paradigm.* Este modelo é composto de três fases [EMM93]:

Entendimento. Esta fase tem como objetivo melhorar o processo de software e seus produtos, caracterizando o ambiente de produção . Isto implica em caracterizar tipos de desenvolvimento de software, definição de problemas, características de processo e de produto.

Avaliação. Na fase de avaliação é medido o impacto das tecnologias disponíveis e das mudanças de processo na geração do produto. Também, determina quais

tecnologias são benéficas e apropriadas para um determinado ambiente e, ainda, como tecnologias ou processos devem ser refinados para melhor se adaptarem.

Empacotamento. Identificadas as melhorias de processo, organizar o pacote formado pela tecnologia que será utilizada na organização. Isto inclui desenvolvimento e melhoria de padrões e política de treinamento e desenvolvimento.

2.4 Seleção de Modelos

As soluções apresentadas neste capítulo para gerenciamento de processo de software são modelos de referência, ou seja, descrevem em linhas gerais as atividades a serem executadas. Modelos de referência precisam de adaptações e definições para serem implantados numa organização qualquer. A implantação de um modelo de gerenciamento de processo de software possui algumas etapas geralmente utilizadas, mas que não se configuram como um padrão de atividades a serem seguidas para seleção. As etapas estão listadas e descritas a seguir:

Necessidade. Nesta primeira etapa, o corpo administrativo e gerencial da organização avalia resultados e dados estatísticos, e percebe a carência de um processo de controle e melhoria de qualidade de seus produtos ou serviços. Há, também, pesquisa com potenciais clientes, entrevistas com usuários e consulta ao corpo técnico responsável pelo desenvolvimento e implementação de seus produtos. Os principais fatores que levam a organização a adotar um modelo de gerência de processo de software são a necessidade de garantia de qualidade e o desconhecimento do processo de software utilizado no desenvolvimento de seus produtos.

Objetivos. Consolidada a necessidade de definição de um processo através da utilização de um modelo, os principais objetivos a serem alcançados a curto, médio e longo prazo precisam ser definidos para que o modelo seja escolhido de forma a satisfazê-los. Dentre os objetivos, figuram índices e padrões almejados, e.g, ISO 9001, CMM - nível 2.

Pré-seleção. A pré-seleção consiste em avaliar, com as necessidades e os objetivos a serem alcançados definidos, quais modelos melhor se enquadram com o perfil sugerido pelas definições. Nesta pré-seleção, ainda não são considerados os custos envolvidos nem o tempo necessário para capacitação de pessoal.

Análise de Riscos. Um levantamento de riscos é realizado. São levados em consideração custos, tempo de implantação, grau de confiabilidade do modelo, resultados já alcançados por outras organizações, reação do corpo técnico a mudanças, capacidade de implantação do corpo gerencial, entre outros. Os riscos são ponderados de acordo com o contexto e com as necessidades da organização, e são analisados para cada modelo pré-selecionado.

Refinamento da Pré-seleção. Sucessivos refinamentos são realizados até se chegar a uma seleção satisfatória. Estes refinamentos seguem o seguinte ciclo até que um único modelo seja selecionado: avaliação dos riscos analisados, realização de nova seleção e nova análise de riscos.

Com estas etapas seguidas, a implantação de um modelo de gerenciamento de processo de software torna-se menos traumática e pode ser realizada de forma planejada, dado que necessidades, objetivos e riscos associados estão bem definidos. Isto, do ponto de vista organizacional, pois a adaptação humana ao novo modelo é completamente imprevisível. Um acompanhamento psicológico e de conscientização através de palestras e cursos explicativos devem estar presentes durante a implantação de um modelo de gerenciamento de processo de software [Hum96]. Este acompanhamento é para evitar e prevenir uma possível aversão ao novo modelo por parte dos indivíduos da organização.

2.4.1 Considerações

Alguns trabalhos presentes na literatura fornecem estudos comparativos entre os modelos apresentados [Pau95, PKG95, Mes97]. Estes trabalhos fornecem dados que auxiliam a seleção e tornam mais evidentes as diferenças e semelhanças entre objetivos, particularidades e

características de cada modelo. Ainda assim, um estudo detalhado das necessidades e dos objetivos da organização são indispensáveis durante a seleção e implantação de um dos modelos na organização.

Os modelos de referência para gerenciamento de processo de software fornecem recursos variados para a melhoria do processo e controle da qualidade. Muitas organizações têm alcançado melhorias significativas nos seus processos e modos de trabalho [HCR+94]. Com a utilização de um modelo de processo de software, a organização possui dados e parâmetros para auto-melhoria e para comparação com outras organizações, de forma a estender seus limites e almejar novos objetivos.

Contudo, estes modelos estão totalmente voltados para avaliação e melhoria de tecnologia e processos organizacionais, pouco dizem em relação ao indivíduo. Com a utilização eficiente de modelos, tornando o processo definido e gerenciado, alcança-se índices excelentes de melhoria de processo. Entretanto, para alcançar a otimização do processo, o investimento na capacitação individual é fundamental e indispensável. O impacto de talentos individuais na realização de projetos algumas vezes consegue superar a falta de um processo de software. [CKI88] mostra alguns casos em que talentos individuais alcançam índices e fatores de produção e de qualidade superiores a equipes utilizando processos de software bem definidos, mas que isso não é uma regra geral.

As organizações detectaram a necessidade de capacitação e melhoria do modo de trabalho dos indivíduos de seus corpos técnicos. Sem um modelo de capacitação de pessoal bem definido, um estado caótico de gerenciamento de pessoal é provocado e assim pouco pode-se progredir no sentido de otimizar o modo de trabalho. Os modelos de processo de software presentes na literatura, dos quais alguns foram apresentados neste capítulo, não fornecem métodos nem técnicas para a "otimização de pessoal".

Tentativas para atrair, desenvolver, manter e motivar talentos individuais nas organizações são constantes preocupações do corpo gerencial. Instalam-se academias, contratam-se psicólogos, estimulam-se terapias de grupo e trabalhos criativos, pagam-se cursos, reservam-se horários de distração e meditação durante o expediente, mas todos estes esforços aconte-

cem de maneira desorganizada e pouco controlada. Deste modo, o objetivo de fornecer um processo de melhoria contínua das práticas utilizadas pelos indivíduos nos seus processos de trabalhos não é alcançado.

Na área de engenharia de software, estudos têm sido realizados para a definição de processos individuais de construção de software que atendam às necessidades de melhoria de processo de trabalho do próprio indivíduo e da organização à qual ele pertença. Estes estudos visam capacitar o indivíduo para que ele próprio identifique seus principais pontos críticos e, desta forma, apresente soluções para superar seus pontos fracos e melhorar suas qualidades.

No Capítulo 3 são apresentados dois modelos utilizados com o objetivo de identificar e melhorar o processo individual de trabalho.

Capítulo 3

Gerenciamento Pessoal de Software

Seguindo um modelo de gerenciamento de processo de software, organizações têm alcançado melhorias significativas nos seus processos e modos de trabalho [HCR+94]. Contudo, muitas organizações perceberam que para obterem índices melhores dependiam, ainda, do talento individual de seus funcionários [CKI88]. Com isso, estudos e pesquisas foram direcionados para a capacitação e melhoria do processo pessoal. Processo pessoal são as técnicas, os métodos e as práticas utilizadas pelos recursos humanos de uma organização para executar tarefas.

O SEI, *The Software Engineering Institute*, com o objetivo de preencher a lacuna deixada pelos modelos de gerência de processo de software com relação ao processo pessoal, desenvolveu alguns modelos direcionados à melhoria de processo tanto de pessoal quanto individual. P-CMM, *People-Capability Maturity Model*, e PSP, *Personal Software Process*, são os modelos apresentados pelo SEI como recursos para melhoria e otimização do processo pessoal e individual de trabalho, respectivamente.

Alguns modelos desenvolvidos sugerem práticas e métodos para que o próprio indivíduo consiga identificar e corrigir seus pontos fracos. Entretanto, o mérito do talento individual jamais será superado por tais técnicas. A criatividade e a capacidade de iniciativa necessárias e presentes nos bons funcionários não são castradas por tais modelos. Os modelos são sugestões para organizar e disciplinar os processos individuais e não diminuem nem restringem

a capacidade criativa dos indivíduos. As seções a seguir apresentam os modelos mencionados, suas descrições, características e principais objetivos.

O modelo PSP está estruturado na forma de um curso em [Hum95a], cujo conteúdo programático e estrutura estão descritos, também, aqui neste capítulo. O PSP foi utilizado por nós e alguns resultados são apresentados, junto com considerações e pontos críticos observados durante a utilização.

3.1 P-CMM

People - Capability Maturity Model [CHMK95, CHM95a] é um modelo em desenvolvimento no SEI que tem como objetivo principal suprir a deficiência em relação ao gerenciamento e melhoria de processo pessoal em organizações que implantam programas de melhoria e otimização de processo de software seguindo o modelo CMM para software.

Com a implantação de modelos de gerenciamento de processo, as organizações precisam avaliar e capacitar também seu pessoal. O P-CMM, *People CMM*, está sendo desenvolvido para melhorar a capacidade das organizações desenvolvedoras de software de atrair, desenvolver, motivar, organizar e reter o talento de seus funcionários. O talento individual é fundamentalmente importante para a melhoria contínua do processo de software da organização. O *People* CMM fornece orientação às organizações desenvolvedoras de software para:

- Caracterizar o nível de maturidade das práticas utilizadas pelo seu pessoal;
- Orientar um programa de melhoria contínua de processo pessoal;
- Determinar prioridades para atividades urgentes;
- Integrar o desenvolvimento de pessoal com a melhoria de processo de software;
- Estabelecer a cultura de excelência em engenharia de software.

O P-CMM é organizado na forma de uma estrutura de níveis de maturidade, padronizada no formato do CMM. Cada nível possui as práticas-chave para o gerenciamento e desenvolvimento de pessoal de uma organização. Os níveis de maturidade do P-CMM estão descritos na subseção a seguir.

3.1.1 Estrutura em Níveis de Maturidade

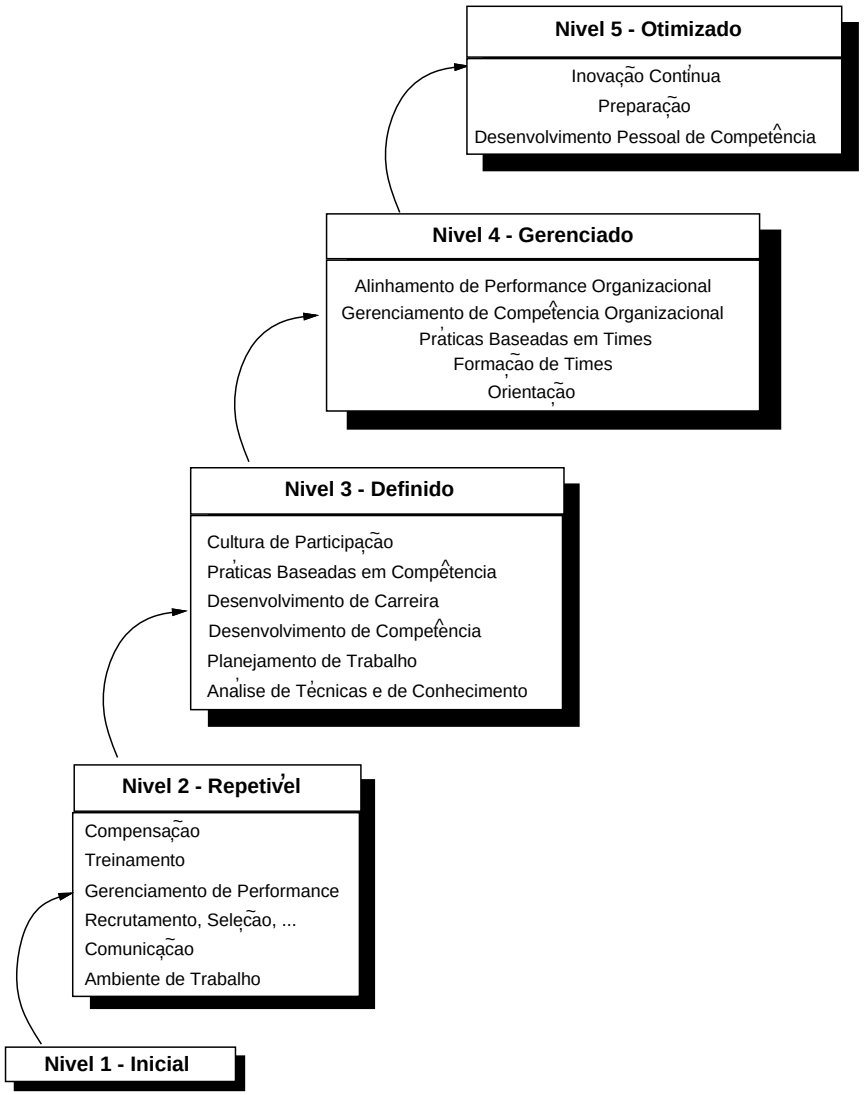

Figura 3.1: Evolução do People-CMM

Como no CMM, os níveis de maturidade descrevem um caminho de evolução que vão de um estado caótico com práticas inconsistentes a um estado maduro com práticas organizadas, técnicas bem definidas e motivações para o desenvolvimento de pessoal

Maturidade no P-CMM [CHM95b] representa a capacidade da organização em melhorar as técnicas e o conhecimento dos recursos humanos e associar isto com os objetivos da organização. A seguir são descritos os níveis de maturidade e suas respectivas práticas, ressaltando as características da organização que se encontra no nível de maturidade descrito. Os níveis de evolução estão ilustrados na Figura 3.1.

Inicial. Neste nível, a organização não possui um conjunto de técnicas nem de métodos que forneça recursos para um gerenciamento disciplinado de pessoal. Os gerentes de recursos humanos trabalham com suas próprias técnicas. A organização não possui uma padronização de tarefas. Atividades como recrutamento e seleção, que necessitam de critérios bem definidos e em completo acordo com os objetivos da organização, produzem resultados ora satisfatórios ora indesejáveis, transformando o gerenciamento de pessoal num processo totalmente caótico.

Repetível. Quando a organização encontra-se no nível repetível, políticas são adotadas e implantadas para que as práticas relacionadas com pessoal estejam completamente estabelecidas. A consciência de que gerenciamento e processo pessoal são atividades cruciais para a melhoria do processo de software está presente nos objetivos da organização. Com isso, gerentes podem adotar práticas e repeti-las com sucesso durante atividades relacionadas com pessoal.

O principal objetivo no nível repetível é institucionalizar na organização as práticas relacionadas ao gerenciamento de processo pessoal. Tornando as práticas atividades comumente realizadas e cotidianas na organização, os gerentes não terão dificuldade em adotar novas e sofisticadas técnicas relacionadas a gerenciamento de pessoal. As áreas-chave das práticas realizadas por organizações no nível repetível são:

Ambiente de Trabalho. Instalar e manter um ambiente de trabalho com condições físicas adequadas, permitindo que as tarefas sejam realizadas eficientemente e que

distrações inapropriadas sejam as mínimas possíveis.

Comunicação. Instituir um ambiente social que favoreça à interação social. Garantir que os funcionários tenham técnicas para compartilhar informações e coordenar suas tarefas.

Recrutamento, Seleção e Promoção de Pessoal. Determinar um processo formal pelo qual talentos sejam recrutados, selecionados e promovidos.

Gerenciamento de Performance. Estabelecer critérios objetivos de performance individual e de grupo para serem medidos. Os resultados destas medições são avaliados e fornecem dados para um melhoramento contínuo de performance de pessoal.

Treinamento. Garantir que todos os membros da organização recebam treinamento nas técnicas necessárias para realização de suas atividades.

Compensação. Fornecer a cada funcionário remuneração e benefícios proporcionais a sua dedicação à organização.

Definido. No nível definido a organização começa a alcançar maturidade para adaptar as práticas definidas para os funcionários no nível repetível para a sua realidade, direcionando o processo pessoal especificamente para sua atividade fim. Com a disciplina de processos pessoais obtida com a determinação das práticas do nível repetível, a organização consegue investir numa estratégia para desenvolvimento de talentos de seu pessoal.

A organização possui capacidade de fazer previsões e estimativas de performance com os dados obtidos pelas práticas do nível repetível. Assim, é capaz de avaliar e direcionar as técnicas e o conhecimento necessário para a realização de atividades e otimização da performance de pessoal. As áreas-chave de uma organização no nível definido incluem:

Análise de Técnicas e de Conhecimento. Identificar o conhecimento e as técnicas necessárias ao corpo de funcionários para a realização dos processos de software específicos de sua atividade.

Planejamento de Trabalho. Coordenar as atividades a serem realizadas com as atuais e futuras necessidades da organização.

Desenvolvimento de Competência. Ressaltar constantemente a capacidade do corpo de funcionários em realizar tarefas e cumprir com responsabilidades.

Desenvolvimento de Carreira. Garantir a motivação de todos os membros da organização, fornecendo oportunidades para o desenvolvimento de novas técnicas e práticas que enfatizem sua capacidade de alcançar seus objetivos de carreira profissional.

Práticas Baseadas em Competência. Garantir que todas as práticas sejam baseadas na capacidade e no conhecimento do corpo gerencial, cujos treinamento e aprimoramento foram realizados visando especificamente a otimização do processo de software da organização.

Cultura de Participação. Melhorar o fluxo de informações dentro da organização e incorporar o conhecimento dos membros da organização nas decisões de mudança e atualização do processo de software da organização.

Gerenciado. A organização que se encontra no nível gerenciado possui formação de times de trabalho, forte orientação resultante da experiência adquirida através dos outros níveis e práticas relacionadas a pessoal direcionadas ao desenvolvimento destes times. Objetivos são quantitativamente definidos. Dados são coletados e analisados para avaliação de quanto efetivamente as práticas de pessoal estão melhorando a competência dos funcionários e otimizando a performance em todos os níveis da organização.
Tendências futuras de capacidade e performance do corpo de funcionários são previstas. Estas previsões estimadas são possíveis devido à avaliação quantitativa da eficiência das técnicas e práticas utilizadas na organização. As áreas-chave do nível gerenciado são:

Orientação. Uso da experiência de alguns funcionários para fornecer suporte e ajuda aos menos experientes. Esta orientação pode envolver desenvolvimento de técnicas e de conhecimento, melhoria de performance, soluções para situações difíceis, e

decisões com respeito ao progresso da carreira profissional.

Formação de Times. Favorecer a formação de times de trabalho, pois possibilitam a integração de informações e técnicas.

Práticas Baseadas em Times. Adaptar as práticas desenvolvidas na organização para suportar a criação e o desenvolvimento de times efetivos de trabalho.

Gerenciamento de Competência Organizacional. Aumentar a capacidade da organização em atividades relativas à melhoria de competência. Determinar a eficácia das práticas de desenvolvimento de competência para alcançar os objetivos com respeito à otimização de competência.

Alinhamento de Performance Organizacional. Buscar o alinhamento entre os resultados de performance individual, em grupo, em times de trabalho e a nível da organização. Favorecer quantitativamente este alinhamento.

Otimizado. No nível otimizado, todos os níveis da organização são alvos para a melhoria contínua de performance. Com os dados obtidos durante a realização das práticas executadas e já consolidadas na organização com este nível de maturidade, experiências com novas técnicas podem ser realizadas. Estas novas experiências visam a constante melhoria e a otimização de performance e de técnicas de pessoal aplicadas na organização.

Com a cultura de melhoria contínua implantada na organização, cada funcionário é um fornecedor e gerente em potencial de práticas e técnicas de melhoria de performance. Assim, o controle está descentralizado e todo o processo de software da organização é beneficiado. As áreas-chave relativas a este nível incluem:

Desenvolvimento Pessoal de Competência. Fornecer condições para que os próprios membros da organização avaliem e desenvolvam suas performances ao realizar atividades.

Preparação. Fornecer assistência especializada para a melhoria de performance individual e de times de trabalho. Consultores e indivíduos da organização trocam

experiências e discutem novas técnicas para melhoria e otimização de performance.

Inovação Contínua. Identificar e avaliar técnicas de trabalho e novas tecnologias, executando as mais promissoras na organização.

A orientação fornecida pelos cinco níveis de maturidade oferece à organização uma estrutura linear, ou seja, as práticas de um nível só são exigidas depois que todas as práticas do nível anterior estão consolidadas na organização. Esta estrutura linear é importante para a melhoria gradativa do modo de trabalho de engenheiros de software e de outros profissionais.

No projeto P-CMM está prevista a inclusão de um método de avaliação de pessoal que será integrado à avaliação de processo de software já existente e realizada no CMM. A integração poderá ser realizada de maneira suave e através de SEPGs, *Software Engineering Process Groups*, [CHMK95, FR90]. Assim, as organizações poderão incorporar gerenciamento de pessoal aos programas de melhoria e otimização de gerenciamento de processo.

3.1.2 Considerações

O P-CMM fornece à organização uma estrutura de práticas distribuídas em áreas-chave que identificam, definem e melhoram a performance de trabalho de seus funcionários.

Por outro lado, a implantação do modelo P-CMM requer a iniciativa do corpo gerencial, demandando um esforço no sentido gerente - funcionário. As barreiras a serem vencidas para implantação do P-CMM envolvem custos e tempo para implantação do modelo, inércia dos membros da organização em incorporar o modelo no seu modo de trabalho, motivação do corpo gerencial para implantação do modelo e adaptação do modelo às necessidades da organização.

A participação do indivíduo nos estágios iniciais de implantação do modelo e até nos níveis iniciais de maturidade é puramente passiva. Pois, só há participação ativa dos funcionários no nível otimizado do P-CMM. Se houvesse mais participação do indíviduo em

todas as etapas de evolução do modelo, custos com adaptação, tempo de implantação e inércia na aceitação do modelo seriam reduzidos.

3.2 PSP - Visão Geral

O PSP, *Personal Software Process*, [Hum95b, Hum95c] foi desenvolvido para tornar o trabalho mais produtivo, adequado e satisfatório ao desenvolvimento de sistemas em escala individual, fazendo com que o próprio programador encontre seus limites.

Baseado no CMM para software, o PSP também possui níveis e objetivos a serem alcançados a cada nível. A evolução do processo consiste em alcançar os objetivos para mudar de nível dentro do modelo. Estas evoluções são ilustradas na Figura 3.2. Como podemos observar, a evolução é composta por fases que são seguidas até se alcançar o pleno controle sobre as atividades de desenvolvimento. As atividades de desenvolvimento são: planejamento baseado em dados obtidos com os relatórios, padrão de codificação, revisões de código e geração de relatórios de acompanhamento. Notamos, também, os principais objetivos a serem alcançados em cada etapa do desenvolvimento.

As estratégias utilizadas na elaboração do PSP são listadas a seguir:

- Identificação de técnicas e métodos utilizados em sistemas de grande escala que possam ser úteis para os sistemas individuais;
- Definição de um subconjunto destes métodos e técnicas para serem aplicados no desenvolvimento de pequenos programas;
- Estruturação destes métodos para que sejam gradualmente introduzidos ao modelo PSP;
- Fornecimento de um conjunto de exercícios a serem realizados, possibilitando o aprendizado do PSP.

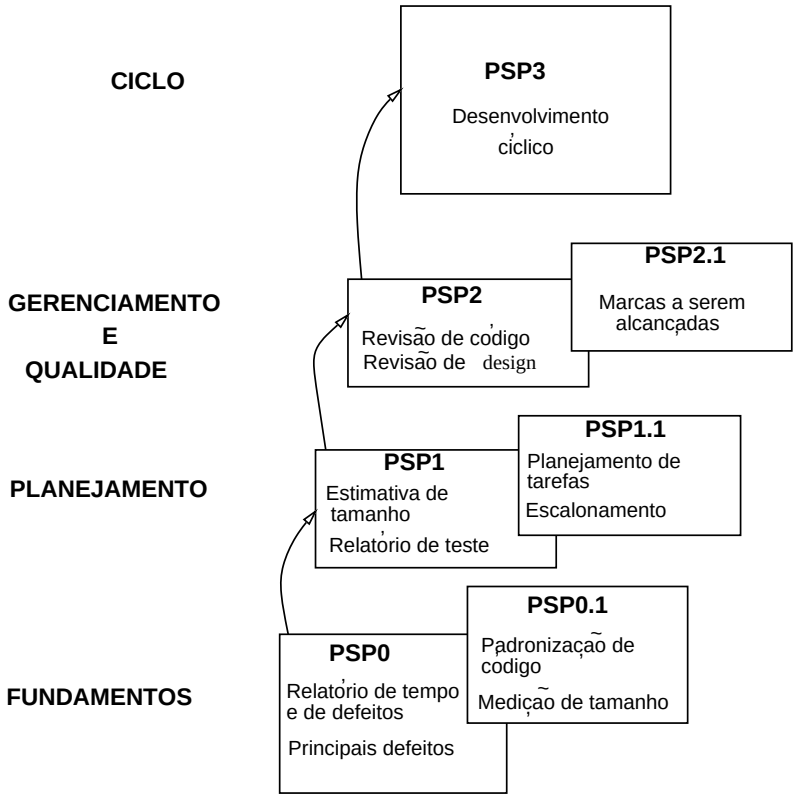

Figura 3.2: Evolução do PSP

O PSP está organizado na forma de um curso em [Hum95a] a ser descrito neste capítulo. Nas seções seguintes, descreveremos em poucas linhas as evoluções do PSP.

Todas as etapas são apenas sugestões do modelo, podendo ser modificadas a critério do programador. A princípio, segue-se as etapas sugeridas e então, com a experiência e maturidade adquirida, pode-se modificá-las. Cada uma das etapas possui relatórios que também estão sujeitos a modificações a critério do programador.

3.2.1 PSP0

Nesta etapa, mede-se o tempo gasto em cada fase do desenvolvimento, os defeitos inseridos e os encontrados em cada fase. O modelo apresenta tabelas cujos campos fornecem dados

para compor os relatórios.

Dentro do PSP0 existe PSP0.1, cujo objetivo é formar um consenso entre os programadores e adotar um padrão de escrita para os programas, de modo a tornar uniforme os programas produzidos.

O objetivo do PSP0 é construir uma base de dados, padronizada, para ser utilizada pelas outras etapas, assim como, mostrar ao programador alguns parâmetros muitas vezes esquecidos, e.g., erros que foram provocados ao se tentar consertar um problema.

3.2.2 PSP1

Planejamento é a palavra-chave desta etapa. Com a base de dados obtida com PSP0, PSP1 fornece relatórios de teste, tamanho de código e estimativa para os recursos necessários. As principais razões para a geração destes relatórios são:

- Perceber o relacionamento entre o tamanho dos programas desenvolvidos e o tempo gasto para desenvolvê-los;
- Ajudar o programador a assumir compromissos que possa cumprir;
- Fornecer um planejamento ordenado das tarefas a serem cumpridas;
- Fornecer dados para avaliação do trabalho realizado.

Em PSP1.1 utiliza-se alguns métodos bastente conhecidos para se estimar tamanho de código (**Delphi, Fuzzy Logic, Proxy-Based**, etc.) e também para estimar e escalonar os recursos (**COCOMO, Regressão Linear**, etc.).

3.2.3 PSP2

Durante o PSP2, técnicas de revisão de código são aplicadas com o objetivo de encontrar os possíveis defeitos, antes que seja tarde demais para consertá-los. Isto é feito mediante

uma análise dos defeitos encontrados nas compilações e testes de programas anteriormente desenvolvidos. Com isto, é possível estabelecer listas de ítens (*checklists*) para revisão e, assim, alcançar o nível de qualidade desejado.

Os principais métodos de revisão são inspeções, *walk-through* e revisões pessoais. As inspeções são formas estruturadas para revisão de programas. *Walk-through* consiste em seguir os passos que serão executados pelo programa para realizar uma determinada tarefa, isto com um grupo de pessoas questionando cada passo e cada decisão tomada pelo programa. Na revisão pessoal, o programador analisa os relatórios dos programas produzidos por ele. O objetivo dos métodos de revisão é encontrar e consertar quantos erros forem possíveis antes da fase final de teste do programa. Alguns princípios para revisão, tanto para os códigos quanto para os projetos, podem ser úteis:

- Produzir projetos que possam ser revistos;
- Estabelecer objetivos a serem alcançados;
- Seguir explicitamente uma estratégia de revisão;
- Dividir a revisão em estágios;
- Verificar se a lógica implementa corretamente a especificação.

Ao término do processo de revisão, relatórios são gerados. Os principais dados para a geração dos relatórios são:

- Tamanho do programa que está sendo revisto;
- Tempo de revisão;
- Número total de defeitos encontrados;
- Números de erros encontrados após a revisão;
- Porcentagem de defeitos encontrados durante a revisão;

- Número médio de defeitos encontrados a cada mil linhas de código (KLOC *K line of code*) de desenvolvimento ou código revisado;
- Número médio de defeitos encontrados por hora de revisão;
- Número médio de linhas de código revistas por hora.

3.2.4 PSP3

PSP0, PSP1 e PSP2 são baseados num processo linear para desenvolvimento de pequenos programas, o que os torna trabalhosos para grandes programas. PSP3 sugere utilizar a estratégia de dividir um programa grande em partes gerenciáveis por PSP2. A idéia é desenvolver programas incrementalmente, ou seja, primeiro constrói-se um módulo básico e depois ciclicamente incrementa-se este módulo até completar o sistema desejado. A cada iteração, o processo de PSP2 é completado, incluindo desenvolvimento, codificação, revisão, compilação e teste, semelhante ao modelo iterativo de processo de desenvolvimento de software [Jal92].

PSP3 permite concentrar a verificação de qualidade apenas no último módulo acrescentado, evitando assim, preocupações com os módulos anteriores. Os módulos anteriores são considerados quando se deseja executar testes de regressão, os quais envolvem todas as fases de desenvolvimento anteriormente implementadas.

Para facilitar a divisão de um grande programa, os softwares podem ser classificados, de acordo com o tamanho, em:

- **Tamanho 0 -** Pequenos módulos de programas, escritos por programadores individualmente.
- **Tamanho 1 -** Pequenos programas, ou módulos, que foram desenvolvidos, implementados e testados por programadores individualmente.
- **Tamanho 2 -** Grandes programas, ou componentes, que envolveram grupos de programadores, os quais desenvolveram e integraram vários módulos de tamanho 0 em módulos de tamanho 2.

- **Tamanho 3 -** Projetos grandes que envolvem vários times de programadores que são controlados e gerenciados por um gerente central.
- **Tamanho 4 -** São os super projetos (projetos que envolvem vários sistemas).

Utilizando um critério de divisão adequado, podemos utilizar as estratégias de PSP em softwares de quaisquer tamanhos.

3.3 Curso PSP

O modelo PSP possibilita que o próprio engenheiro de software melhore e otimize sua maneira de trabalhar, fornecendo uma base histórica para que possa planejar e estimar melhor sua capacidade de cumprir prazos e compromissos. Independentemente da instituição onde se encontra, o engenheiro de software pode seguir o modelo e tornar seu trabalho mais produtivo, organizado e de melhor qualidade.

Para tornar o modelo acessível a todos, PSP está estruturado na forma de um curso com quinze palestras em [Hum95a]. Desta forma, um auto-treinamento é possível. As práticas e as técnicas necessárias para alcançar os níveis de evolução do modelo PSP são apresentadas incrementalmente no curso. A cada nível alcançado, novas tabelas vão sendo incorporadas e as antigas, atualizadas. O livro possui quatorze capítulos cujos conteúdos estão distribuídos e relacionados com os níveis PSP da seguinte forma[1]:

Capítulo 1 - Introdução e Descrição Geral do Modelo PSP. Neste capítulo o modelo PSP é descrito em linhas gerais. Os objetivos pretendidos, a lógica e a estratégia de atuação do modelo, e algumas definições de termos utilizados durante o curso são apresentadas.

Capítulo 2 - PSP0. Este capítulo apresenta os guias básicos para o modelo PSP, que são divididos em duas categorias: roteiros (*scripts*) e tabelas. Os roteiros fornecem a

[1] As tabelas foram mantidas com seus nomes originais, em inglês, para seguir o padrão apresentado no Apêndice C em [Hum95a].

orientação para o preenchimento dos campos das tabelas, estas últimas possuem dados que irão compor os relatórios a serem gerados. Os guias a serem seguidos no nível PSP0 são:

- **PSP0 Process Script**
- **PSP0 Planning Script**
- **PSP0 Development Script**
- **PSP0 Postmortem Script**
- **PSP0 Project Plan Summary and Instructions**
- **Time Recording Log and Instructions**
- **Defect Recording Log and Instructions**
- **Defect Type Standard**

Basicamente, neste nível, são gerados relatórios de tempo e principais defeitos encontrados. No curso, os dados são obtidos implementando o exemplo 1A (descrito no Apêndice D do livro).

Capítulo 3 - PSP0.1. Neste capítulo são apresentados os aspectos positivos em fazer e utilizar um planejamento eficaz de tarefas e estimativas de tempo de execução de atividades durante o desenvolvimento de software. Nenhuma tabela é gerada neste capítulo nem nenhum exercício é exigido.

Capítulo 4 - PSP0.1. Este capítulo introduz medição de tamanho e padronização de código. Também apresenta uma tabela para anotação das dificuldade encontradas durante o processo de desenvolvimento. O registro de tais dificuldades é utilizado para posterior avaliação e melhoria do processo utilizado. Os guias de PSP0.1 são[2]:

- **PSP0.1 Process Script**
- **PSP0.1 Planning Script**

[2]Os guias escritos em negrito representam as tabelas modificadas ou acrescentadas e em itálico, as tabelas herdadas dos níveis anteriores.

- **PSP0.1 Development Script**
- **PSP0.1 Postmortem Script**
- **PSP0.1 Project Plan Summary and Instructions**
- **Process Improvement Proposal (PIP) and Instructions**
- **Coding Standard**
- *Time Recording Log and Instructions*
- *Defect Recording Log and Instructions*
- *Defect Type Standard*

Neste nível, além dos relatórios PSP0, são gerados um padrão para codificação e um relatório de dificuldades encontradas no processo adotado para a implementação dos exercícios (1A e 2A).

Capítulo 5 - PSP1. Métodos estatísticos para estimativa de resultados são apresentados neste capítulo. Diversos métodos são apresentados e ilustrados com exemplos, dentre eles o método PROBE utilizado no curso. Também, é ressaltada a importância do uso de métodos estimativos no planejamento de atividades e de tempo, utilizando dados históricos. PSP1 é formado de:

- **PSP1 Process Script**
- **PSP1 Planning Script**
- **PSP1 Development Script**
- **PSP1 Postmortem Script**
- **PSP1 Project Plan Summary and Instructions**
- **PROBE Estimating Script**
- **Test Report Template and Instructions**
- **Size Estimating Template and Instructions**
- *Process Improvement Proposal (PIP) and Instructions*

- *Coding Standard*
- *Time Recording Log and Instructions*
- *Defect Recording Log and Instructions*
- *Defect Type Standard*

O uso dos métodos são exercitados sobre os exercícios já realizados e a realizar (1A, 2A e 3A).

Capítulo 6 - PSP1.1. Neste capítulo são incluídos planejamento e escalonamento de tarefas no PSP, utilizando os métodos estatísticos apresentados em PSP1. PSP1.1 descreve como fazer planejamento para pequenos programas e como agrupar pequenos planos para o planejamento de um sistema formado por pequenos programas. Os guias que compõem PSP1.1 são:

- **PSP1.1 Process Script**
- **PSP1.1 Planning Script**
- **PSP1.1 Development Script**
- **PSP1.1 Postmortem Script**
- **PSP1.1 Project Plan Summary and Instructions**
- **Task Planning Template and Instructions**
- **Schedule Planning Template and Instructions**
- *PROBE Estimating Script*
- *Test Report Template and Instructions*
- *Size Estimating Template and Instructions*
- *Process Improvement Proposal (PIP) and Instructions*
- *Coding Standard*
- *Time Recording Log and Instructions*
- *Defect Recording Log and Instructions*

- *Defect Type Standard*

O exercício exigido para este nível é o 5A, utilizando os dados de todos os outros exercícios realizados anteriormente.

Capítulo 7 - PSP1.1. Este capítulo ressalta a importância da coleta e uso dos dados no modelo PSP. Critérios devem ser estabelecidos para que as avaliações sejam realistas e fundamentadas nas considerações presentes nos critérios. Neste capítulo, o exercício 6A é exigido, devendo ser implementado seguindo o processo PSP1.1.

Capítulo 8 - Definição do Relatório Parcial. Um relatório parcial, com os dados coletados até este ponto no curso (1A, 2A, 3A, 4A, 5A e 6A), é produzido como exercício do capítulo. Este relatório apresenta os resultados de performance e o histórico dos exercícios implementados.

Capítulo 9 - PSP2. Com os exercícios dos outros níveis realizados e com seus respectivos dados coletados, este nível fornece parâmetros para avaliação de qualidade do produto com consequência da qualidade do processo. Revisões de código e de projeto são acrescentadas ao PSP neste capítulo. São geradas listas para revisão de código e de projeto. Os guias para este capítulo são:

- **PSP2 Process Script**
- **PSP2 Planning Script**
- **PSP2 Development Script**
- **PSP2 Postmortem Script**
- **PSP2 Project Plan Summary and Instructions**
- **PSP2 Design Review Checklist**
- **Code Review Checklist**
- *Task Planning Template and Instructions*
- *Schedule Planning Template and Instructions*

- *PROBE Estimating Script*
- *Test Report Template and Instructions*
- *Size Estimating Template and Instructions*
- *Process Improvement Proposal (PIP) and Instructions*
- *Coding Standard*
- *Time Recording Log and Instructions*
- *Defect Recording Log and Instructions*
- *Defect Type Standard*

O exercício exigido neste capítulo é o de número 7A.

Capítulo 10 - PSP2 e PSP2.1. Neste capítulo, PSP é incorporado com tópicos de processo de projeto e critérios para projetos de qualidade. PSP2.1 possui especificações funcionais, baseadas em estados e em sentenças lógicas. Também são incorporados roteiros com várias sequências de ações a serem executadas pelo usuário. Por fim, o comportamento das reações do sistema a estas ações é analisado e corrigido, se necessário. Os guias para este capítulo são:

- **PSP2.1 Process Script**
- **PSP2.1 Planning Script**
- **PSP2.1 Development Script**
- **PSP2.1 Postmortem Script**
- **PSP2.1 Project Plan Summary and Instructions**
- **Operational Scenario Template and Instructions**
- **Functional Specification Template and Instructions**
- **State Specification Template and Instructions**
- **Logic Specification Template and Instructions**
- **PSP2.1 Design Review Checklist**

- *Code Review Checklist*
- *Task Planning Template and Instructions*
- *Schedule Planning Template and Instructions*
- *PROBE Estimating Script*
- *Test Report Template and Instructions*
- *Size Estimating Template and Instructions*
- *Process Improvement Proposal (PIP) and Instructions*
- *Coding Standard*
- *Time Recording Log and Instructions*
- *Defect Recording Log and Instructions*
- *Defect Type Standard*

O exercício exigido neste capítulo é o 7A.

Capítulo 11 - PSP3. Neste capítulo, algumas estratégias para tratar grandes problemas são apresentadas. PSP3 apresenta uma análise superficial de projeto, utilizada em grandes sistemas para se obter uma visão geral destes. Apresenta, também, orientação para tratar ciclicamente os módulos e programas que irão compor o sistema analisado. Os guias para este capítulo são:

- **PSP3 Process Script**
- **PSP3 Planning Script**
- **PSP3 High-level Design Script**
- **PSP3 High-level Design Review Script**
- **PSP3 Development Script**
- **PSP3 Postmortem Script**
- **PSP3 Project Plan Summary and Instructions**

- **Cycle Summary and Instructions**
- **Issue Tracking Log and Instructions**
- *Operational Scenario Template and Instructions*
- *Functional Specification Template and Instructions*
- *State Specification Template and Instructions*
- *Logic Specification Template and Instructions*
- **PSP3 Design Review Checklist**
- *Code Review Checklist*
- *Task Planning Template and Instructions*
- *Schedule Planning Template and Instructions*
- *PROBE Estimating Script*
- *Test Report Template and Instructions*
- *Size Estimating Template and Instructions*
- *Process Improvement Proposal (PIP) and Instructions*
- *Coding Standard*
- *Time Recording Log and Instructions*
- *Defect Recording Log and Instructions*
- *Defect Type Standard*

Neste capítulo é iniciado o exercício 10A.

Capítulo 12 - Definição do Relatório Final. Neste capítulo é indicada a utilização de métodos formais para verificação de corretude dos programas. Apesar de não adotar nenhuma linguagem de especificação formal, este método para garantir corretude é fortemente indicado no modelo PSP. O exercício 10A é concluído neste capítulo, assim como, um relatório final com todos os dados coletados durante o curso.

Capítulo 13 - Definição do Processo de Software. Este capítulo apresenta conceitos e orientação para definição do processo pessoal de software.

Capítulo 14 - Utilização do PSP. Neste capítulo são apresentados vários conselhos para uma melhor utilização do modelo.

Palestra	Capítulo	Exercício	Nível PSP
1	1		
1	2	1A	PSP0
2	3		
2	4	2A	PSP0.1
3	5	3A	PSP0.1
4	5	4A	PSP1
5	6	5A	PSP1.1
6	7	6A	PSP1.1
7	8		
8	9	7A	PSP2
9	10	8A	PSP2
10	10	9A	PSP2.1
11	11	10A	PSP3
12	12		
13	12		
14	13		
15	14		

Tabela 3.1: Curso PSP - Estrutura Básica.

O módulo básico do curso é formado por uma série de dez exercícios que cobrem todos os níveis de evolução do modelo PSP. Os exercícios são incorporados ao modelo para fixar e aplicar o conhecimento adquirido com os textos dos capítulos. A estrutura básica do curso está apresentada na Tabela 3.1 .

3.4 Exercícios Realizados

Nesta seção apresentamos os exercícios realizados e as tabelas preenchidas com os dados obtidos com a implementação dos programas. O enunciado dos exercícios e as tabelas foram mantidas com seus nomes e textos originais, em inglês.

Neste trabalho, realizamos os exercícios do curso até o capítulo 5 (PSP0.1) com duas linguagens:
Smalltalk e Java. O curso foi realizado de maneira autodidata, seguindo o livro [Hum95a] e o cronograma sugerido. Foram utilizadas máquinas Sun (SPARCstation SLC) e os exercícios foram realizados sobre as seguintes plataformas:

Smalltalk. Os exemplos foram implementados em Smalltalk VisualWorks(TM) (versão 1.0). A versão VisualWorks da *ParcPlace Systems* [Par92] foi adotada por fornecer, além de todo o ambiente Smalltalk padrão, um conjunto de ferramentas para manipulação de janelas e construção de interfaces.

Java. Java(TM)[3] (versão 1.02). Esta versão de Java não possui nenhum recurso adicional, além do compilador.

Os exercícios correspondentes aos capítulos 1 a 5 (1A, 2A e 3A) estão enunciados no Apêndice A deste texto. Assim como as tabelas preenchidas e os códigos dos programas implementados. Para os nossos propósitos, apresentamos o exercício 2A (em Smalltalk e Java) para ilustrar a avaliação do modelo e dos métodos utilizados por ele.

3.4.1 Exercício 2A

Program 2A Requirements: Write a program to count the logical lines in a program, omitting comments and blank lines. Use the counting standard produced by report exercise R1 to place

[3]No Capítulo 4 há uma breve descrição da linguagem Java.

one logical line on each physical line and count physical lines. Produce a single count for the entire program source file.

Program 2A Testing: Thoroughly test the program. As one test, count the LOC in programs 1A, 1B (if written), and 2A. Submit these data with your homework results, using the format in Table D6.

Smalltalk

A realização deste exercício depende da definição de um padrão para codificação. Contudo, utilizando o ambiente Smalltalk, temos uma opção "Format", que padroniza o código implementado. Desta forma, nossa tarefa neste exercício foi resumida a implementar o contador de linhas de código.

A grandeza que determinamos para medir os tamanhos dos códigos implementados em Smalltalk foi o número de mensagens utilizadas num método, já que o número de linhas de código é bastante reduzido devido ao alto grau de reusabilidade característico da programação utilizando Smalltalk.

Uma simples consulta de 10 minutos ao *Browser de Classes* do ambiente Smalltalk nos possibilitou resolver este exercício. Como a classe *CompiledCode* já possuia um método (apresentado na hierarquia abaixo) que retorna um conjunto com as mensagens chamadas em um código Smalltalk, o que este exercício pede já estava pronto no próprio ambiente. Devido a estas razões, as tabelas não foram preenchidas, pois os resultados de dados como tempo de codificação, tempo de teste, número de defeitos gerados, etc., seriam todos zero.

```
Kernel-Methods
   CompiledCode
      accessing-literals
         messages
```

```
messages
   "Answer a Set of all the message selectors sent by this method."

   | scanner selectorSet selector |
   selectorSet := Set new.
   self withAllBlockMethodsDo:
   [:meth |
   scanner := InstructionStream on: meth.
   scanner
      scanFor:
         [:byte |
         selector := scanner peekForSelector.
         selector == nil ifFalse: [selectorSet add: selector].
         false "keep scanning"]].
   ^selectorSet
```

Java

A implementação deste exercício em Java foi baseada na classe *StringTokenizer*. O arquivo fonte, considerado um objeto da classe *String*, é a entrada para o contador de linhas de código exigido por este exercício. A contagem das linhas de código seguiu o padrão de tabulação definido na tabela *Table C29 PSP0.1-JOA Java Coding Standard* (vide Apêndice C), desconsiderando os espaços em branco, os comentários e as linhas que só possuiam "}". Adotamos a métrica de linhas de código por ser uma métrica de fácil mensurabilidade, pela sintaxe de Java ser bastante parecida com C++ e por ser adotada por Humphrey no curso PSP. O código fonte e as tabelas PSP0.1[4] correspondentes a este exercício são apresentados a seguir:

[4]A Tabela C27 PIP *Process Improvement Proposal* não está apresentada neste exercício. O seu conteúdo foi extenso o suficiente para motivar este trabalho.

```
/* Classe que recebe uma String no padrao PSP para
   Java (vide tabela C29) e retorna o numero de linhas de codigo,
   eliminando espacos em branco e comentarios. */

import java.util.StringTokenizer;

class Loc {

   protected String javaCode;
   boolean inCommentState = false;

   public Loc(String string) {
      javaCode = string;
   }

   /* metodo que retorna o numero de linhas fisicas de codigo */
   public int lineCount() {

      StringTokenizer st = new StringTokenizer(javaCode,"\n\r");
      String lineCode = "";
      int lineNumber = 0;

      while (st.hasMoreTokens()) {

         lineCode = st.nextToken();

         if (isFuncLine(lineCode) && !isCommentLine(lineCode)) {
            lineNumber++;
         }
      }
```

```
      return lineNumber;
   }

  // ----------------------------------------
   /* metodo que retorna se eh uma linha logica de codigo */
   private boolean isFuncLine(String line){

      if (line.endsWith(";") || line.endsWith("{")){
         return true;
      } else return false;
   }
  // ----------------------------------------
   /* metodo que retorna se a linha faz parte de um comentario */
   private boolean isCommentLine(String line){

      if (line.startsWith("//")) {
         return true;
      }

      if (string1PossuiString2(line, "/*")) {
         inCommentState = true;
      }

      if (string1PossuiString2(line, "*/")) {
         inCommentState = false;
         return true;
      }
      return inCommentState;
   }
  // ----------------------------------------
```

```
   /* metodo auxiliar para verificar a presenca do string2 no string1 */
   private boolean string1PossuiString2(String string1, String string2){

      if (string1.indexOf(string2, 0) >= 0) { // sempre a partir do comeco de string1
         return true;
      } else return false;
   }
}
```

Como sugerido no enunciado, os testes foram realizados contando as linhas de código dos exercícios 1A e 2A.

Program Number	**LOC (Line Of Code)**
1A	37
2A	38

Table D6 TEST RESULTS FORMAT - PROGRAM 2A

Summary	**Plan**	**Actual**	**To Date**	**To Date %**
Program Size (LOC)	**Plan**	**Actual**	**To Date**	
Base(B)		15		
Deleted(D)		6		
Modified(M)		8		
Added(A)		24		
Reused(R)		5	5	
New and Changed (N)	20	32	69	
Total LOC (T)		38	75	
Total New Reused		0	0	

Table C14 PSP0.1 *Project Plan Summary* (Continued)

Time in Phase(min.)	**Plan**	**Actual**	**To Date**	**To Date %**
Planning	25	20	40	7.8
Design	30	65	85	16.5
Code	90	150	225	43.7
Compile	15	15	15	2.9
Test	50	30	70	13.6
Postmortem	50	40	80	15.5
Total	260	320	515	100.0
Defects Injected		**Actual**	**To Date**	**To Date %**
Planning		0	0	0
Design		1	3	9.1
Code		12	27	68.2
Compile		2	2	2.9
Test		2	7	22.7
Total Development		17	39	100.0

Table C14 PSP0.1 *Project Plan Summary* (Continued)

Defects Removed	**Plan**	**Actual**	**To Date**	**To Date %**
Planning		0	0	0
Design		1	2	5.1
Code		10	25	64.1
Compile		2	2	5.1
Test		4	10	25.7
Total Development		17	39	100.0

Table C14 PSP0.1 *Project Plan Summary*

Date	**Start**	**Stop**	**Interruption Time**	**Delta Time**	**Phase**	**Comments**
09/01	16:50	17:10	0	20 min	Planning	
	17:10	18:25	10 min	65 min	Design	Telefone
	20:30	22:45	30 min	105 min	Coding	Lanche
	23:15	00:15	0	06 min	Coding	
13/11	07:10	07:40	0	30 min	Teste	
	08:15	09:00	5 min	40 min	Postmortem	Gerais

Table C16 *Time Recording Log*

Table C18 PSP0 *Defect Recording Log*

Student Jones Albuquerque Date 10/01/96

Program Contador de linhas de código Program# 2A

Date	Number	Type	Inject	Remove	Fix Time	Fix Defect
09/01/97	1	100	design	design	1 min	

Description: consulta à classe errada

Date	Number	Type	Inject **Phase**	Remove **Phase**	Fix Time	Fix Defect
09/01/97	10	20	code	code	8 min	

Description: vários erros de digitação

Date	Number	Type	Inject **Phase**	Remove **Phase**	Fix Time	Fix Defect
09/11/97	2	20	compile	compile	1 min	

Description: utilização ineficiente de método

Date	Number	Type	Inject **Phase**	Remove **Phase**	Fix Time	Fix Defect
10/01/97	2	50	code	teste	2 min	

Description: nomes errados de métodos

Date	Number	Type	Inject **Phase**	Remove **Phase**	Fix Time	Fix Defect
10/01/97	2	40	teste	teste	2 min	

Description: nomes errados de variáveis

3.5 Avaliação do Curso e Modelo PSP

A realização dos exercícios, a implementação dos programas e a leitura dos capítulos do curso PSP de [Hum95a] nos forneceu dados para chegarmos a algumas conclusões. Estas conclusões foram obtidas após e durante o acompanhamento do curso PSP.

Ao iniciarmos o curso PSP apresentávamos o seguinte perfil:

- Não possuíamos dados sobre nossa performance no desenvolvimento de softwares;
- Não tínhamos relatórios de atividades anteriormente realizadas;
- Não conseguíamos estimar nossas tarefas e, consequentemente, deixávamos de cumprir prazos;
- O planejamento de atividades era ineficiente pois não possuíamos dados para estimar e melhor escalonar tarefas.

O nosso perfil se enquadrava, exatamente, no que Humphrey chama de "estado caótico", ou seja, não tínhamos um processo de desenvolvimento pessoal de software definido.

As principais características observadas no curso e modelo PSP podem ser divididas em dois grupos: resultados positivos apresentados pelo modelo e pontos críticos encontrados no curso. Os resultados positivos encontrados no modelo são uma comprovação do que o próprio autor do curso PSP menciona, os principais são:

- Realmente, há uma **melhoria da qualidade do código desenvolvido**. Utilizando as tabelas e os métodos sugeridos pelo curso conseguimos identificar nossos principais defeitos e, assim, melhorá-los. Características como tabulação, comentários, padrão de codificação, erros de compilação e execução, entre outras, são verificadas constantemente pelo modelo PSP.
- Com os dados fornecidos no decorrer do curso, a nossa **capacidade de estimar tempo e esforço é consideravelmente melhorada**. A experiência adquirida com os

exercícios possibilita sermos mais criteriosos e realistas no planejamento de nossas atividades. Evitamos prazos que não conseguiríamos cumprir e números mágicos para previsão de esforço.

- Um **melhor gerenciamento de dados** é alcançado através dos relatórios gerados e das tabelas preenchidas e armazenadas. Com uma base documentada e padronizada no formato das tabelas é possível recuperar informações facilmente, ainda que utilizando um processo manual e artesanal de armazenamento de dados em tabelas impressas em papel.

- **PSP fornece um histórico de performance do programador**. Como as tabelas sempre se baseiam nos dados coletados anteriormente em outros programas, o engenheiro de software sempre possuirá um histórico atualizado e incremental de suas atividades.

- **PSP disciplina o processo pessoal de desenvolvimento de software**. Com os métodos seguidos no curso, o programador disciplina-se a realizar tarefas de gerenciamento de seu próprio trabalho em paralelo à execução das atividades. Isto, sem privar a criatividade presente e necessária à atividade de desenvolver softwares.

A melhoria alcançada com o modelo PSP é indiscutível. Todas as características apresentadas no nosso perfil ao iniciar o curso foram reduzidas, algumas chegaram a ser completamente eliminadas, como falta de relatórios sobre nossa performance e falta de histórico de atividades realizadas.

Contudo, estes resultados ainda não são completos e a eficácia do modelo pode ser questionada, se utilizado em situações reais. Apesar da generalidade do modelo PSP, a aplicação do curso a indivíduos que utilizem quaisquer linguagens de programação deverá ser cautelosa e sofrer revisões quanto à forma de avaliação e medição dos dados. Os pontos críticos apresentados pelo curso PSP são descritos a seguir:

- As tabelas do curso PSP apresentam uma **dependência com relação ao paradigma de programação** que se está utilizando para implementar os exercícios. Apesar

de considerar aspectos como linhas de código reutilizáveis, número de objetos desenvolvidos, entre outros; no curso PSP não é reservado tempo em seus roteiros para busca e pesquisa de classes, identificação de métodos e de variáveis já disponíveis, e atividades relacionadas. Estas atividades estão presentes no desenvolvimento de software orientado a objetos, também presentes nas linguagens imperativas só que não tão relevantes.

- A forte **dependência dos exercícios do curso quanto à linguagem e ao ambiente de programação** utilizados é ilustrada com a implementação do Exercício 2A em Smalltalk e Java (vide seção anterior). Observamos que dependendo dos recursos que a linguagem e o ambiente utilizado oferecem, alguns exercícios podem se tornar triviais e completamente fora do objetivo do curso. Alguns destes exercícios sequer fornecem dados para as tabelas (vide Exercício 2A em Smalltalk na seção anterior), ou por exemplo, se utilizássemos o JavaEdit[5] para realizar o Exercício 2A: bastava herdar os método de contagem de linha de código.

- **PSP é fortemente concentrado em codificação e teste** e pouco se refere sobre análise de requisitos, especificação, projeto ou, garantia de corretude e de qualidade. Mesmo possuindo orientação para uso de métodos de especificação em PSP2.1, este não é um de seus principais objetivos.

- **O preenchimento de tabelas é ineficiente no desenvolvimento de um sistema real**. O preenchimento manual envolve um tempo excedente de *postmortem*, geralmente não disponível durante o desenvovimento dos sistemas reais.

- A **cronometragem manual de tempo não é realista**. Erros por distrações do engenheiro de software são comuns, isto aconteceu conosco durante a realização dos programas. Além disso, medir tempo de compilação manualmente em linguagens interpretadas, como Smalltalk, é praticamente impossível. Pois a cada método, o interpretador é acionado e seu tempo de processamento não excede 1 segundo!

[5]JavaEdit está descrito no Apêndice B.

- Para implantação do modelo PSP, **empresas têm que reservar regularmente um intervalo de tempo necessário para o treinamento** seguindo o curso PSP. O tempo em empresas de desenvolvimento de software é bastante disputado e, por esta razão, a implantação do modelo é dificultada.

Como resultado desta avaliação e com o objetivo de reduzir estes pontos críticos, sugerimos alguns procedimentos que devem ser considerados para o perfeito aproveitamento dos benefícios proporcionados pelo modelo de gerenciamento pessoal de software PSP:

1. Adaptação do curso para contornar as diferenças quanto a linguagem, ambiente e paradigma de programação. Esta adaptação deve incluir avaliação e modificação dos exercícios, tabelas e roteiros;

2. Para aplicação real do modelo, um sistema de gerenciamento de dados automático e tomada de tempo independente do usuário devem ser implementados e utilizados em conjunto com a implantação do modelo no modo de trabalho do indivíduo.

3. Criar nos níveis mais baixos de PSP (antes de PSP2.1), roteiros e tabelas para análise de requisitos e especificação do sistema, mesmo que não seja formal, mas que introduza na disciplina de desenvolver software estas fases. O que acontece no modelo atual é que o programador é disciplinado a desenvolver software sem realizar nenhuma especificação e repentinamente, no nível PSP2.1, é acrescentada a linguagem de especificação Z. Isto provoca uma mudança bastante acentuada no modo de desenvolvimento praticado pelo programador até então.

4. A inclusão do modelo PSP durante a formação do profissional engenheiro de software possibilita que as organizações já contratem indivíduos com o perfil PSP. E o tempo com treinamento e implantação do modelo seria extremamente reduzido.

No Capítulo 4, apresentamos PSP-JOA: *Personal Software Process - a Java Oriented Approach.* PSP-JOA é a adaptação de PSP para Java, ilustrando o primeiro procedimento sugerido acima para melhor aproveitamento do modelo PSP.

No Capítulo 5, são apresentados um modelo para um ambiente de programação Java com coleta de dados automática e um esboço de uma disciplina para ser lecionada nos centros de formação de engenheiros de software e de profissionais de computação em geral. Os trabalhos futuros apresentam sugestões para os procedimentos 2 e 4 acima.

Capítulo 4

PSP-JOA

4.1 Introdução

O modelo PSP sugerido por Humphrey foi desenvolvido genérico o suficiente para ser adaptado a qualquer paradigma de programação. Apesar da generalidade do PSP, Os exercícios e tabelas curso são dependentes da linguagem de programação utilizada. Quando determinadas linguagens de programação são utilizadas, os exercícios e as tabelas tornam-se inapropriados e não alcançam os objetivos desejados, como foi mostrado no Capítulo 3. Para tentar reduzir tais imprecisões, no caso específico de Java, elaboramos o que chamamos de PSP-JOA: *Processo Pessoal de Software: Uma Abordagem Orientada a Java.* PSP-JOA é um trabalho baseado no curso sugerido em [Hum95] e utiliza a mesma estrutura apresentada no livro, só alterando e reescrevendo o estritamente necessário.

Este capítulo apresenta as tabelas e os roteiros modificados para Java. Uma descrição de Java é apresentada, além de um subconjunto das tabelas contidas no Apêndice C. Explicações acompanham as tabelas PSP-JOA apresentadas neste capítulo.

4.2 Descrição Geral

PSP-JOA, *Processo Pessoal de Software: Uma Abordagem Orientada a Java* (*Personal Software Process - A Java Oriented Approach*), foi definido para melhor adaptar o curso PSP às características específicas de Java. Esta necessidade tornou-se clara quando seguimos os níveis de evolução de PSP, implementando os exercícios com linguagens orientadas a objetos diferentes das utilizadas nos exemplos ilustrados no livro (C++, Object-Pascal), e.g., Smalltalk e Java (vide Capítulo 3). Percebemos que os roteiros definidos em [Hum95a] precisavam de ajustes e que algumas tabelas precisavam ser reescritas para tornar as avaliações dos resultados mais precisas. Por exemplo, contar linhas de código em Smalltalk não é tão representativo na produtividade do indivíduo quanto em C.

Com a experiência adquirida usando Java [Tea96] e com os resultados obtidos com a implementação dos exercícios, criamos PSP-JOA: uma versão do conjunto de tabelas PSP totalmente voltada ao desenvolvimento pessoal de software, tendo Java como linguagem de implementação. Assim, melhoraremos e tornaremos mais realistas os resultados obtidos através das avaliações dos dados contidos nas tabelas PSP-JOA.

4.3 Java - Visão Geral

Java é uma linguagem de programação simples, orientada a objetos e de propósito geral. Atualmente, está sendo amplamente utilizada nas mais variadas aplicações na Internet.

Java possui dois tipos de programas que podem ser implementados: aplicações (*applications*) e *applets*. Aplicação é uma classe que possui um método `main()`, que é o primeiro a ser executado pelo interpretador Java. Aplicações executam semelhantemente a programas C++. *Applet* é um tipo especial de aplicação que é executada a partir de um *Browser*. No lugar de um método `main()` como nas aplicações, *applets* implementam vários métodos que correspondem às suas ações e representam seu comportamento: iniciação, quando desenhar sua interface, a que ações correspondem o acionamento (*click*) dos botões do mouse, etc.

Java reune várias características que a consolidaram como a linguagem de programação mais utilizada na Internet, entre elas podemos citar:

Orientada a objetos. A organização e a representação dos dados em Java utiliza classes e objetos, apenas os tipos primitivos como inteiro, caracter e número ponto flutuante não obedecem à hierarquia de classes. A modelagem utilizando objetos possibilita as mais variadas abstrações do mundo real.

Distribuída. Java possui classes e mecanismos de comunicação que permitem a implementação de aplicações e *applets* que troquem informações entre si. Estes mecanismos de comunicação favorecem o modelo de computação distribuída.

Dados Arbitrários. Java permite criar dados abstratos e fazer com que uma máquina cliente possa executar um código para "visualização" destes novos dados criados; ou seja, em Java podem ser enviados o conteúdo e o programa necessário para a perfeita visualização deste conteúdo, tudo isto em tempo real. Desta forma, o horizonte de possibilidades de aplicações na Internet é vastamente expandido.

Independente de arquitetura e portabilidade. A independência vem do fato de Java gerar um código intermediário independente da arquitetura. Além disso, Java possui portabilidade garantida por estar disponível para Sparc Solaris, Windows NT - Intel, Windows 95, MacOS 7.5, OS/2, Linux, HP-UX, DEC Alpha OSF/1, Nextstep, SunOS 4.1, Solaris X86, Amiga e algumas outras plataformas que no momento estão em desenvolvimento.

Semelhante a C++. A semelhança de sintaxe entre Java e C++ permitiu que vários programadores de C++ migrassem para Java sem maiores problemas.

As classes de Java estão organizadas hierarquicamente num conjunto chamado API *Application Program Interface*. API pode ser disponibilizado para acesso remoto através da WWW.

4.3.1 Por que Java?

Utilizamos Java para reescrever as tabelas PSP porque é uma linguagem amplamente utilizada pelos profissionais de computação e pela carência de fontes literárias contendo metodologias e processos para desenvolvimento de software utilizando-a.

A nossa experiência em programação utilizando Java permitiu que chegássemos aos problemas e soluções apresentados de forma mais rápida e consciente. A semelhança de Java com C++ tornou a adaptação do modelo PSP para Java mais natural e suave. A apresentação dos exercícios do PSP em C++ e a semelhança de sintaxe entre Java e C++ possibilitaram que os exercícios do PSP fossem diretamente reutilizados no PSP-JOA, sem perda de objetividade.

4.4 Tabelas PSP-JOA

As principais mudanças realizadas nas tabelas PSP relativas à adaptação do modelo para Java são decorrentes de:

- As várias tabelas de *Project Plan Summary Instructions* não ressaltarem a necessidade de computar o tempo necessário para classificar uma nova classe a ser desenvolvida. Entende-se por classificar, escolher de qual classe a nova classe será herdeira, ou ainda, sob que hierarquia de classes iremos implementar um determinado programa.
- A não distinção entre o código reusado da API de Java e o de outras fontes; pois o número de defeitos provocados e o perfeito entendimento do código variam fortemente com a origem do código a ser reusado.
- Um padrão de codificação para Java ainda não está completamente definido.

A seção seguinte possui exemplos das tabelas modificadas, classificadas por nível PSP-JOA. As modificações em relação às originais estão com fonte em **negrito**, mantendo a

padronização de Humphrey em fonte SMALL CAPS para as evoluções em cada fase. As tabelas relativas a *Design Review Checklist* e *Code Review Checklist* mesmo voltadas a C++, podem ser aproveitadas para Java; assim como, todos os exercícios realizados, desde que não utilizemos nenhum recurso adicional fornecido por ambientes de programação. Esta reutilização assegura uma razoável complexidade nos exercícios de forma a satisfazer os objetivos pretendidos pelo curso.

4.4.1 Exemplos

Como exemplo, apresentamos os roteiros e as tabelas modificadas relativas ao nível PSP0.1-JOA[1].

As modificações apresentadas nestas duas tabelas estão explicadas abaixo:

Tabela C26 PSP0.1-JOA. Durante o cálculo do número de linhas de código desenvolvidas, as tabelas originais não faziam distinção entre código reutilizado de fonte segura (API-Java) e de fonte qualquer. Isto acarretava resultados imprecisos nas evoluções desta tabela no modelo; pois, nas demais evoluções desta tabela é computado o número médio de erros por linha de código, e as linhas de código reutilizadas são desconsideradas nestes cálculos, assumindo que estão corretas. Contudo, linhas de código de outras fontes apresentam erros, pois muitas vezes não são testados em sua completa funcionalidade; estes erros são frequentes em Java, já que *applets* podem ser estruturas bastante complexas, com trechos de código que executam independente do resto do programa (*Threads*).

Tabela C29 PSP0.1-JOA. Um padrão para codificação em Java é apresentado nesta tabela, já que o apresentado para C++ na versão original não retratava aspectos como comentários para JavaDoc (*The Java API Documentation Generation*) e tabulação concisa e eficiente ("}" em uma linha separada para cada bloco de código).

[1] As demais tabelas encontram-se no Apêndice C. Para melhor entendimento do contexto destas tabelas sugerimos consultar [Hum95a].

Purpose	This form holds the estimated and actual project data in a convenient and readily retrievable form.
Header	Enter the following: - your name and today's date. - the program name and number. - the instructor's name. - the language you used to write the program.
Program Size (LOC)	PRIOR TO DEVELOPMENT: - IF YOU ARE MODIFYING OR ENHANCING AN EXISTING PROGRAM, COUNT THAT PROGRAM'S LOC, **showing the difference between programs from Java API (BJA) and any other font (BOF)**, AND ENTER **this sum (BJA + BOF)** AS BASE-ACTUAL. - USING YOUR BEST JUDGMENT, ESTIMATE THE NEW AND CHANGED LOC YOU EXPECT TO DEVELOP. AFTER DEVELOPMENT: - IF THE BASE LOC (B) HAS CHANGED, ENTER THE NEW VALUE. - MEASURE THE TOTAL PROGRAM SIZE AND ENTER IT AS THE TOTAL LOC - ACTUAL (T). - REVIEW YOUR SOURCE CODE AND, WITH THE HELP OF PROGRAM 3A, DETERMINE THE ACTUAL LOC THAT WERE DELETED (D), MODIFIED (M), OR REUSED (R). - CALCULATE THE LOC OF ADDED CODE AS A = T - B + D - R. - CALCULATE THE TOTAL NEW AND CHANGED LOC AS N = A + M.

Table C26 PSP0.1-JOA *Project Plan Summary Instructions* **(Continued)**

Time in Phase	- Under Plan, enter your original estimate of the total development time AND THE TIME REQUIRED BY PHASE. - Under Actual, enter the actual time in minutes spent in each development phase. - Under To Date, enter the sum of the actual time and the To Date time from your most recently developed program. - Under To Date %, enter the % of To Date time in each phase.
Defects Injected	- Under Actual, enter the number of defects injected in each phase. - Under To Date enter the sum of the actual numbers of defects injected in each phase and the To Date values from the most recently developed program. - Under To Date %, enter the % of To Date defects injected by phase.
Defects Removed	- Under Actual, enter the number of defects removed in each phase. - Under To Date enter the sum of the actual number of defects removed in each phase and the To Date value from the most recently developed program. - Under To Date %, enter the % of To Date defects removed by phase. - After development, record any defects later found during program use, reuse, or modification.

Table C26 PSP0.1-JOA *Project Plan Summary Instructions*

Purpose	To guide the development of Java programs
Program Headers	Begin each file with a history table listing dates, authors, and summaries of changes.
File Hearder Format	/* File: the file name Date Author Changes sep 1 95 Doug Lea Created jul 2 96 Joa Added new concepts */

Class Header
 	
Class Documentation Format
	Place each class in a separate file. Preface each class with a /** ... **/ comment using javadoc conventions.
	/**
	* A class representing a window on the screen.
	* For example:
	* <pre>
	* Window win = new Window(parent);
	* win.show();
	* </pre>
	*
	* @see awt.BaseWindow
	* @see awt.Button
	* @version 1.2 12 Dec 1994
	* @author Sami Shaio
	**/
	
	Class Window extends BaseWindow {
	...
	}

Table C29 PSP0.1-JOA *Java Coding Standard* **(Continued)**

Instance Var Documentation	Use javadoc conventions to describe nature, purpose, constraints and usage.
Instance Var Doc Format	/** * The X-coordiate of the window * @see window#1 **/ int x= 1263732

Method Documentation
 	
Method Documentation Format
	Use Javadoc conventions to describe nature, purpose, preconditions, effects, algorithmic notes, usage instructions, reminders, reuse instructions, etc ...
	/**
	* Return the caracter of the specified index. An index ranges
	* from <tt>0</tt> to <tt>length() - 1</tt>.
	* @param index The index of the desired character
	* @return The desired character
	* @exception StringIndexOutOfRangeException When the
	* index is not in the range <tt>0</tt> to
	* <tt>length() - 1</tt>
	*
	**/
	
	public char charAt(int index) {
	...
	}

Table C29 PSP0.1-JOA *Java Coding Standard* **(Continued)**

Identifiers	Use descriptive names for all variables, function names, constants, and other identifiers. Avoid abbreviations or single letter variables. (vide Capitalization)
Example	int number_of_students; /* This is GOOD */ float x4, j, ftave; /* These are BAD */
Internal Block Comments	Use /* ... */ comments to describe algorithmic details, notes, and related documentation that spans more than a few code statements.
Example	/* * strategy: * 1. Find the node * 2. Clone it * 3. Ask inserter to add clone * 4. If successful, delete node */
Internal Running Comments	Use // comments to clarify non-obvious code.
Good Comment	if (record_count) /* have all records been processed? */
Bad Comment	if (record_count) /* check if record_count is grater than limit */

Table C29 PSP0.1-JOA *Java Coding Standard* **(Continued)**

Identing	Indent every level of bracket from the previous one. If you use the JavaEdit Editor, this indenting is made automaticly by option Java-PSPFormat menu bar (vide Appendix B).

Example	public class Scanner {
	/**
	* Basic elements recognized by the scanner.
	* White spaces are ignored.
	*/
	static int NUMBER = 0; // an unsigned number
	static int IDENT = 1; // an identifier
	static int ERROR = 5; // error (unrecognized element)
	
	/**
	* A dictionary mapping the delimiters and operators
	* into their token types. This allows easy extension
	* of this class.
	*/
	public static Hashtable TokenMap = new Hashtable();
	
	static int i=0;
	static {
	for (i=0;i < delimiters.length;i++) {
	TokenMap.put(delimiters[i],new Integer(DELIM));
	}
	for (i=0;i < operators.length;i++) {
	TokenMap.put(operators[i],new Integer(OPER));
	}
	}
	}

Table C29 PSP0.1-JOA *Java Coding Standard* **(Continued)**

Capitalization	Try to stick to the following naming conventions: • packages: lowercase • classes: CapitalizedWithInternalWordsAlsoCapitalized • methods: firstWordLowerCaseButInternalWordsCapitalized • constants (finals): UPPER_CASE_WITH_UNDERSCORES • private or protected members: trailingUnderscore_ • method local variables: lower_case_with_underscores • factory methods: newX • converter methods: toX • method that reports an attribute x of type X: X x() • method that changes an attribute x of type X: void x(X value)

Table C29 PSP0.1-JOA *Java Coding Standard*

4.5 Considerações

Com a nossa experiência, adquirida com a realização do curso, percebemos que todas as fases e métodos devem ser seguidos estritamente como indicado. Só com a evolução do curso é que percebemos a importância de cada uma das fases no processo como um todo.

Por outro lado, como o próprio Humphrey menciona [Hum95a], as tabelas devem ser adequadas ao desenvolvedor. Mas só se consegue maturidade para modificá-las após a realização do curso. O que fizemos neste trabalho foi fornecer, devido à nossa experiência com Smalltalk e Java, as tabelas já modificadas para que as incompatibilidades entre as tabelas PSP e Java sejam amenizadas ou, em certos casos, eliminadas.

Para muitos, as tabelas ainda não representarão a melhor alternativa para modelagem do curso PSP para Java, apesar das mudanças propostas por este trabalho. Claro, o modelo PSP é para ser adaptado pelo próprio engenheiro de software ao seu modo de trabalho. Contudo, as dificuldades provocadas pelo uso de Java como linguagem de implementação dos exercícios PSP foram consideravelmente reduzidas com as tabelas sugeridas pela versão

PSP-JOA. As pequenas mudanças realizadas nas tabelas fornecerão melhores resultados, pois representam com mais detalhe as características de um código desenvolvido em Java.

Os exercícios propostos em [Hum95a], como parte do curso PSP, continuam os mesmos para PSP-JOA. Contudo, os exercícios atuais não exploram recursos de Java como *Threads*, *Interfaces*, Concorrência, entre outros.

Capítulo 5

Conclusões

O processo PSP funciona muito bem, apesar de sua ênfase quase que exclusiva nas fases de codificação e teste do software. Comprovamos que, de maneira gradativa, fatores de qualidade (geração de relatórios, padronização de código, estimativas de recursos e tempo, planejamento adequado, maior produtividade, etc.) vão sendo inseridos no processo de desenvolvimento do software realizado pelo indivíduo, disciplinando-o.

O curso PSP é oferecido no SEI como um dos cursos de capacitação de desenvolvedores de software. Entretanto, para seguir o curso proposto em [Hum95a], de forma autodidata, exige-se do programador uma excelente auto-disciplina no sentido de cumprir todos os passos e prazos exigidos.

Observamos [Hum96, dTdS96] que engenheiros de software recebem o treinamento PSP no SEI e voltam para suas empresas para difundir e implantar tal modelo. Contudo, como mostrado neste trabalho, estes engenheiros podem não obter resultados satisfatórios durante a difusão do modelo entre os demais engenheiros de software de sua empresa. Se as tabelas e os exercícios PSP não forem adaptados à linguagem e ao ambiente de programação utilizados na realização dos exercícios do curso, a avaliação do mesmo fica totalmente prejudicada e não alcança os objetivos pretendidos.

Defendemos a implantação de modelos como CMM, SPICE, Trillium, P-CMM, PSP,

etc., como disciplinas na formação de profissionais de computação. O engenheiro de software que recebe, durante a sua formação, treinamento em modelos de gerência de processo de software, possibilita à empresa que o contratará uma redução nos custos quanto à formação e treinamento de pessoal. Com uma formação em PSP, por exemplo, o engenheiro de software possui um histórico com dados sobre sua performance. Isto o ajudará a tomar e planejar melhor decisões profissionais.

As mudanças proporcionadas por PSP-JOA fornecem maior realismo nos dados coletados durante a realização do curso, o que torna o curso PSP viável quando utilizado com a linguagem Java. Caso contrário, os dados seriam distorcidos e a avaliação do curso prejudicada. PSP-JOA apresenta um padrão de codificação definido e que pode ser acrescentado a um ambiente de desenvolvimento de forma a garantir automaticamente este padrão.

Acreditamos que a coleta automática dos dados possibilita melhorias no aprendizado do modelo PSP, apesar desta não ser a mesma opnião de Watts Humphrey [Hum96]. Watts Humphrey assegura que a coleta manual dos dados possibilita ao programador uma pressão psicológica inconsciente no sentido de evitar erros para diminuir a quantidade de tabelas a serem preenchidas. O que acreditamos é que numa situação real de desenvolvimento de software comercial, se um ambiente de desenvolvimento com coleta automática de dados não suportar PSP, o programador não seguirá tal disciplina por razoes de urgeência nos prazos, estress do dia a dia de trabalho, etc. Sugerimos, como trabalhos futuros, um modelo para um ambiente que tornaria esta automação possível.

A próxima seção apresenta trabalhos futuros que poderão ser realizados. Estes tópicos são decorrentes do nosso estudo sobre o modelo PSP e demais modelos de gerenciamento de processos, e da nossa experiência adquirida com a utilização do PSP.

5.1 Trabalhos Futuros

Dentre os possíveis trabalhos futuros, apresentamos dois importantes tópicos: automação da coleta de dados do modelo PSP e difusão deste modelo nos centros de formação de

profissionais de computação.

A disciplina mencionada a seguir, já é um projeto real e conta com o apoio e a iniciativa do Genesis [Gen95].

5.1.1 WOOPS: *A Web-based Object Oriented Personal Software Process Environment*

Numa situação real de desenvolvimento de software, é bastante inviável coletar e cadastrar todos os dados PSP manualmente, pois, em certos casos, esta coleta poderia tomar mais tempo do programador do que o próprio desenvolvimento da aplicação. O que sugerimos é que isso fosse realizado automaticamente. Medição do tempo de utilização de recursos (*Login*), armazenamento do número e dos tipos de erros gerados, preenchimento das tabelas PSP, etc., poderiam ser realizados através de um poderoso gerenciador de eventos, os quais seriam gerados a cada informação a ser tratada e armazenada nas tabelas. Ainda, tudo isto poderia ser armazenado numa base de dados e assim teríamos geração de relatórios semi-automática e controle de versões. Desta forma, conseguiríamos dados mais confiáveis e, ainda, eliminaríamos o tempo de *Postmortem*. Tempo de *Postmortem* é o tempo dedicado ao preenchimento das tabelas e geração dos relatórios. Tudo isso poderia ser implementado em Java como parte de um ambiente de desenvolvimento de software em Java.

Idealizamos um ambiente protótipo de desenvolvimento de software acoplado a um módulo automatizador para coleta de dados PSP-JOA. O módulo WOOPS seria responsável pela coleta dos dados e geração das tabelas PSP-JOA neste ambiente de desenvolvimento de software idealizado, no qual WOOPS estaria incorporado. A implementação do módulo WOOPS e do ambiente protótipo de desenvolvimento poderá ser baseada no WebSteer [FdLM96].

WebSteer é um ambiente implementado em Java pelo *Recife Java Team* do Departamento de Informática - UFPE [Tea96]. O propósito do WebSteer é fornecer um ambiente educacional para ensino à distância na Internet. A maneira como foi desenvolvido e implementado possibilita que seja aplicado a diversas áreas e propósitos. Abaixo, segue uma

breve descrição funcional dos módulos do WebSteer, e como eles poderão ser utilizados e modificados para compor um ambiente protótipo de desenvolvimento de software suportando PSP-JOA.

WebSteer - Linhas Gerais

O ambiente é composto de três grandes módulos *Slide Projector*, *WorkBench* e *WebSteer Core Controller*:

Slide Projector. Um *browser HTML* modificado para, além de mostrar arquivos HTML, gerar eventos a serem tratados pelo *WebSteer Core Controller.*

WorkBench. Módulo que suporta um conjunto de instrumentos implementados em Java. Estes instrumentos reagem aos eventos gerados pelo *Slide Projector* de acordo com um roteiro escrito em JetScript[1] e que é controlado pelo WebSteer Core Controller.

WebSteer Core Controller. Módulo responsável por todo o controle de comunicação, geração e tratamento de eventos.

Toda a comunicação é realizada através de geração e tratamento de eventos. Todo o processo de comunicação e uma descrição completa de WebSteer podem ser visualizados e entendidos através da opção *About WebSteer* no próprio WebSteer [FdLM96, Tea96].

WOOPS

WOOPS = WebSteer + Java + PSP. A nossa idéia é associar os eventos do *Slide Projector*, convertido num editor Java, a chamadas a um *applet* PSP. O preenchimento das tabelas PSP-JOA é realizado automaticamente de acordo com os eventos gerados pelo *Slide Projector.*

Os passos para a implementação deste processo são:

[1] Linguagem desenvolvida para escrever a sequência de comandos a ser executada pelo WebSteer

- Modificar o *Slide Projector* para permitir edição de código Java;
- Criar o *applet* PSP para preencher as tabelas, armazenar e retornar os seus resultados;
- Escrever o código JetScript para tal módulo.

Observamos assim, que teremos um ambiente de avaliação em escala individual de desenvolvimento de software em Java na Web. Ou seja, sempre que o programador, onde quer que se localize, desejar utilizar seus parâmetros e tabelas de avaliação, estas estarão disponíveis, da mesma forma que as aulas de WebSteer estão disponíveis hoje.

5.1.2 Disciplina

Com a iniciativa da SOFTEX, *Sociedade Brasileira para Promoção da Exportação de Software*, em divulgar o modelo CMM nas universidades e escolas de computação brasileiras, o Genesis está visando a capacitação de pessoal em Gerência de Projetos com a disseminação do modelo CMM juntamente com o modelo PSP nos seus genes [Gen95].

Nosso objetivo futuro é definir e estruturar uma disciplina de Gerência de Projetos para ser lecionada nos genes. Isto, através da definição e elaboração de um programa para esta disciplina contendo módulos de CMM, PSP, Gerência de Projetos e aplicações práticas.

Esta disciplina visa capacitar os alunos em Gerência de Projetos de Software e, em conjunto com a disciplina Empreendedores em Informática, formar um completo empreendedor, incorporando o fator qualidade aos projetos desenvolvidos e empreendidos pelos alunos. Tudo isto através do estudo dos modelos de maturidade CMM e PSP.

Com o decorrer do curso, espera-se que o aluno obtenha domínio de utilização de tais modelos e que, desta forma, consiga aplicá-los aos seus futuros projetos, quer como exercícios finais de disciplinas quer como empreendimentos nos seus negócios.

Desta forma, o aluno estará utilizando um modelo de gerência e processo de desenvolvimento de software para organizar, planejar e otimizar seu modo de trabalho e de elaboração e execução de projetos de sua carreira profissional.

Nossos próximos passos serão: analisar o material do curso fornecido pelo CITS, *Centro Internacional de Tecnologia de Software*, para verificarmos a possibilidade de incorporá-lo ao conteúdo da disciplina a ser definida; estruturar um programa que reflita as necessidades e expectativas do Genesis; e preparar um kit para aula, contendo transparências, apostilas com um guia para o professor e um para o aluno, e exemplos com aplicações práticas para exercícios. Atualmente estamos na fase de estruturação do programa para a disciplina.

Referências Bibliográficas

[1] Bibliografia

[Boe81] Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[CHM95a] B. Curtis, W. E. Hefley, and S. Miller. Overview of the people capability maturity model. Technical Report CMU/SEI-95-MM-01, Software Engineering Institute, 1995.

[CHM95b] B. Curtis, W. E. Hefley, and S. Miller. People capability maturity model. Technical Report CMU/SEI-95-MM-02, Software Engineering Institute, 1995.

[CHMK95] Bill Curtis, William E. Hefley, Sally Miller, and Michael Konrad. The people-cmm. *SPN - Software Process Newsletter*, (4), Fall 1995.

[CKI88] B. Curtis, H. Krasner, and N. Iscoe. A field of the software design process for large systems. *Communications of The ACM*, 31(11):1268–1287, 1988.

[Coa94] F. Coallier. The trillium model. Bell Canada, 1994.

[Coa95] F. Coallier. Trillium: A model for the assessment of telecom product development and support capability. *IEEE TCSE Software Process Newsletter*, (2), 1995.

[Cus91] M. A. Cusumano. *Japan's Software Factories*. Oxford University Press, 1991.

[Dor96] Alec Dorling. Spice spotlight. *SPN - Software Process Newsletter*, Fall(7), 1996.

[Dor97] Alec Dorling. Spice spotlight. *SPN - Software Process Newsletter*, Winter(7), 1997.

[dTdS96] CITS Centro Internacional de Tecnologia de Software. Gestão da qualidade no desenvolvimento de software: O modelo sei/cmm. Curso oferecido pelo CITS, October 1996. Material didático: CMM-Kit de Treinamento.

[EG96] Khaled El Emam and Dennis R. Goldenson. Some initial results from the international spice trials. *SPN - Software Process Newsletter*, Spring(6), 1996.

[EMM93] K. El Eman, N. Moukheiber, and N. H. Madhavsi. Empirical evaluation of the g/q/m method. *Proceedings of The CAS Conference (CASCON'93)*, 1993.

[FH93] Tom Flecher and Jim Hunt. *Software Engineering and Case, Bridging the Culture Gap.* McGraw-Hill, Inc., 1993.

[FR90] P. Fowler and S. Rifkin. Software engineering process group guide. Technical Report CMU/SEI-90-TR-024, Software Engineering Institute, 1990.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley, 1996.

[Gra92] Robert B. Grady. *Practical Software Metrics for Project Management and Process Improvement.* Prentice Hall, 1992.

[HCR+94] J. Herbsleb, A. Carleton, J. Rozum, J. Siegel, and D. Zubrow. Benefits of cmm-based software process improvement: Initial results. Technical Report CMU/SEI-94-TR-13, Software Engineering Institute, 1994.

[Hum90] Watts S. Humphrey. *Managing The Software Process.* Addison Wesley, 1990.

[Hum95a] Watts S. Humphrey. *A Discipline for Software Engineering.* Addison Wesley, 1995.

[Hum96] Watts Humphrey. Software: A challenge for management. Segunda Semana de Engenharia de Software, July 1996.

[Jal92] Pankaj Jalote. *An Integrated Approach to Software Engineering.* Narosa Publishing House, 1992.

[Jur88] J. M. Juran. *Juran on Planning for Quality.* The Free Press, 1988.

[KM88] B. G. Kolkhorst and A. J. Macina. Developing error-free software. *Proceedings of Computer Assurance COMPASS'88*, July 1988.

[KT85] B. Kitchenham and N. R. Taylor. Software project development cost estimation. *Journal of Systems and Software*, 5:267–278, 1985.

[McD94] John McDermid. *Software Engineer's Reference Book.* Butterworth Heinemann, 1994.

[Mes97] Richard Messnarz. A comparison of bootstrap and spice. *SPN - Software Process Newsletter*, (8), 1997.

[mS97] Jean martin Simon. Experiences in the phase 1 spice trial. *SPN - Software Process Newsletter*, Winter(7), 1997.

[Par92] Parc Place Systems. *Tutorial and User's Guide*, 1992.

[Pau95] Mark C. Paulk. How iso 9001 compares with the cmm. *IEEE Software*, pages 74 – 83, January 1995.

[PCCW93] Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, and Charles V. Weber. Capability maturity model for software version 1.1. Technical Report CMU/SEI-93-TR24, Software Engineering Institute, 1993.

[PKG95] Mark C. Paulk, Michael D. Konrad, and Suzanne M. Garcia. Cmm versus spice architectures. *SPN - Software Process Newsletter*, (3), 1995.

[Pre95] Roger S. Pressman. Engenharia de software. 1995.

[PWG+93] Mark C. Paulk, Charles V. Weber, Suzanne M. Garcia, Mary Beth Chrissis, and Marilyn W. Bush. Key practices of the capability maturity model version1.1. Technical Report CMU/SEI-93-TR25, Software Engineering Institute, 1993.

[Som92] Ian Sommerville. *Software Engineering.* Addison Wesley, 1992.

[Zar90] Paul F. Zarrella. Case tool integration and standardization. Technical Report CMU/SEI-90-TR-14, Software Engineering Institute, 1990.

[2] Web Bibliografia

[Can94] Bell Canada. The trillium model. http://www2.umassd.edu:80/swpi/bellcanada/trillium-html/trillium.html, 1994.

[Com96] IEC International Electrotechnical Commission. International electrotechnical commission - home page. http://www.iec.ch/, 1996.

[FdLM96] Jorge Henrique Cabral Fernandes and Silvio Romero de Lemos Meira. A first and simple approach for interative documentation and training over the web: The websteer project1996. http://www.di.ufpe.br/~java/websteer/TheWebSteerProject/TheWebSteerProject.html, 1996.

[fS96] ISO International Organization for Standardization. International organization for standardization. http://www.iso.ch/, 1996.

[Gen95] Genesis. Genesis home page. http://www.di.ufpe.br/~genesis, 1995.

[Hum95b] Watts S. Humphrey. The personal software process. http://www.sei.cmu.edu/products/publications/95.reports/95.ar.psp.html, 1995.

[Hum95c] Watts S. Humphrey. The personal software process overview, practice, and results. http://www.sei.cmu.edu/products/publications/95.reports/95.ar.psp.over.html, 1995.

[Ins96] Bootstrap Institute. Bootstrap institute. http://www.bootstrap.org/, 1996.

[ISO96] Welcome to iso easy! http://www.exit109.com:80/~leebee, 1996.

[JTC95] JTC1. Joint technical commitee 1. http://www.jtc1tag-org/ http://www.iso.ch/meme/JTC1.html, 1995.

[Lea96] Doug Lea. Draft java coding standard. http://www-aig.jpl.nasa.gov/home/decoste/HTMLS/JAVA/LOCAL-JAVA/ HTML/doug-lea/javaCodingStd.html, 1996.

[Lig96] Tracy Lightfoot. Spice - software process improvement and capability determination. http://www-sqi.cit.gu.edu.au/spice/, 1996.

[Mat96] Design Matters. Iso9000 network - collaborative training and registration. http://www.best.com/~iso9000, 1996.

[Sys95] Creative Data Systems. The software quality page. http://www.tiac.net/users/pustaver, 1995.

[Tea96] The Recife Java Team. The recife java team. http://www.di.ufpe.br/~java, 1996.

Apêndice A

Exercícios PSP

Todas as tabelas e apêndices mencionados nestes exercícios são encontrados em [Hum95a]. Os enunciados dos exercícios, assim como, o texto das tabelas foram mantidos como forma original, em inglês.

A.1 Enunciados dos Exercícios

A.1.1 Exercício 1A

Program 1A Requirements: Write a program to calculate the mean and standard deviation of a series of n real numbers. The mean is the average of the numbers and the formula for standard deviation is:

$$\sigma(x_1, ..., x_n) = \sqrt{\frac{\sum_{i=1}^{n}(x_i - x_{avg})^2}{n-1}}$$

Here, $\sigma(x_1, ..., x_n)$ is the standard deviation of the x values and x_{avg} is the average of

these n values. The standard deviation calculation is described in Appendix A, section A2.

Use a linked list to hold the n numbers.

Program 1A Testing: Thoroughly test the program. At least three of the tests should use the data in each of the three columns on the right of the Table D5. The standard deviations for the columnns in this table are Object LOC: 572.03; Total LOC: 625.63; and Development Hours: 62.26.

Program No.	*Object LOC*	*Total LOC*	*Dev. Hours*
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*

Table D5 Pascal Object Size and Development Hours

A.1.2 Exercício 2A

Program 2A Requirements: Write a program to count the logical lines in a program, omitting comments and blank lines. Use the counting standard produced by report exercise R1 to place one logical line on each physical line and count physical lines. Produce a single count for the entire program source file.

Program 2A Testing: Thoroughly test the program. As one test, count the LOC in programs 1A, 1B (if written), and 2A. Submit these data with your homework results, using the format in Table D6.

Program Number	LOC (Line Of Code)
1A	
1B (if written)	
2A	

Table D6 *TEST RESULTS FORMAT - PROGRAM 2A*

A.1.3 Exercício 3A

Program 3A Requirements: Write a program to count the total program LOC, the total LOC in each object the program contains, and the number of methods in each object. Produce a single LOC count for an entire source program file and separate LOC and method counts for each object. Print out each object name together with its LOC and method count. Also print out the total program LOC count. If an object-oriented language is not used, count the procedure and function LOC and print out the procedures and functions names and LOC counts. Use the counting standard produced by report exercise R1. It is acceptable to enhance program 2A or to reuse some of its methods, procedures, or functions in developing program 3A.

Program 3A Testing: Thoroughly test the program. At a minimum, test the program by counting the total program and object LOC in programs 1A, 1B (if written), 2A, 2B(if written), and 3A. Include in your test report a table giving the counts obtained with program 2A and 3A for all the programs written to date. Use the format in the example in Table D7.

Program Num.	**Object Name**	**Num. of Methods**	**Object LOC**	*Total LOC*
1A	*ABC*	*3*	*86*	
	DEF	*2*	*8*	
	GHI	*4*	*92*	
				212
2A	*...*			

Table D7 *TEST RESULTS FORMAT - PROGRAM 3A*

(Format for object-oriented program designs)

Program Num.	**Proc/Func Name**	**Num. of Methods**	**Proc/Func LOC**	*Total LOC*
1A	*ABC*	*3*	*86*	
	DEF	*2*	*8*	
	GHI	*4*	*92*	
				212
2A	*...*			

Table D7 *TEST RESULTS FORMAT - PROGRAM 3A*

(Format for non object-oriented designs)

A.2 Soluções Smalltalk

Alguns esclarecimentos sobre os métodos estatísticos requeridos nos exercícios são apresentados no Apêndice A do livro e, também, um roteiro para a implementação do exercício.

A.2.1 Exercício 1A

Com o enunciado descrito e com o roteiro apresentado na tabela A8 de [Hum95a], foi implementado o código abaixo. Observamos que, como Smalltalk já possuia uma classe que

se comportava como uma lista ordenada *OrderedCollection*, a implementação consistiu em redefinir uma subclasse de *OrderedCollection* e implementar os métodos de desvio padrão e média.

```
'From VisualWorks(TM), Release 1.0 of 8 October 1992
 on 19 September 1995 at 3:41:47 pm'!

OrderedCollection variableSubclass: #LinkedList
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'Collections-Sequenceable'!

!LinkedList methodsFor: 'statisticalFunctions'!

average
"Calcula a media entre os elementos da lista"
| sum |
sum := 0.0.
self do: [:elem | sum := sum + elem].
^sum / self size!

meanDeviation
"Calcula o somatorio dos quadrados do desvio medio de cada
elemento e a media"
| avg sum |
sum := 0.0.
avg := self average.
self do: [:elem | sum := sum +( (elem - avg) raisedTo: 2)].
```

```
^sum!

standardDeviation
"Calcula o desvio padrao dos elementos do receptor"
| aux |
aux := self.
^(aux meanDeviation / (aux size - 1)) sqrt! !
```

Para este exemplo foram obtidos os valores ilustrados nas tabelas C14, C16 e C18. O teste do exemplo foi executado com o código abaixo, utilizando os dados da tabela D5.

```
" Teste para o PSP 1A"
|objectLOC totalLOC developHours |

objectLOC := LinkedList new.
totalLOC := LinkedList new.
developHours := LinkedList new.

objectLOC      add: 160; add: 591;  add: 114; add: 229; add: 230;
       add: 270; add: 128; add: 1657; add: 624; add: 1503.
totalLOC       add: 186; add: 699;  add: 132; add: 272; add: 291;
       add: 331; add: 199; add: 1890; add: 788; add: 1601.
developHours   add: 15.0; add: 69.9; add: 6.5; add: 22.4; add: 28.4;
       add: 65.9; add: 19.4; add: 198.7; add: 38.8; add: 138.2.

objectLOC standardDeviation." 572.027"
totalLOC standardDeviation. "625.634"
developHours standardDeviation. "62.2558"
```

Time in Phase(min.)	**plan**	**Actual**	**To Date**	**To Date %**
Planning		10	10	11.1
Design		20	20	22.2
Code		40	40	44.4
Test		5	5	5.6
Postmortem		15	15	16.7
Total	150	90	90	100.0
Defects Injected		**Actual**	**To Date**	**To Date %**
Planning		0	0	0
Design		0	0	0
Code		1	1	100.0
Test		0	0	0
Total Development		1	1	100.0
Defects Removed		**Actual**	**To Date**	**To Date %**
Planning		0	0	0
Design		0	0	0
Code		0	0	0
Test		1	1	100.0
Total Development		1	1	100.0

Table C14 PSP0 *Project Plan Summary*

Date	**Start**	**Stop**	**Interruption Time**	**Delta Time**	**Phase**	**Comments**
10/09	12:05	12:15	0	10 min	Planning	Ordered Collection
19/09	14:10	14:30	0	20 min	Design	
	18:50	19:35	5 min	40 min	Coding	Queda do sistema
	19:35	19:40	0	5 min	Test	
20/09	20:30	20:45	0	15 min	Postmortem	Preenchendo tabelas

Table C16 *Time Recording Log*

Table C18 PSP0 *Defect Recording Log*

Student Jones Albuquerque Date 19/09/95

Program Desvio padrão Program# 1A

Date	Number	Type	Inject	Remove	Fix Time	Fix Defect
19/09/95	1	20	code	code	1 min	

Description: parentetização errada

A.2.2 Exercício 2A

A realização deste exercício depende da definição de um padrão para codificação. Contudo, utilizando o ambiente Smalltalk, temos uma opção "Format", que padroniza o código implementado. Desta forma, nossa tarefa neste exercício foi resumida a implementar o contador de linhas de código.

A grandeza que determinamos para medir os tamanhos dos códigos implementados em Smalltalk foi o número de mensagens utilizadas num método, já que o número de linhas de código é bastante reduzido devido ao alto grau de reusabilidade característico da programação utilizando Smalltalk.

Um simples consulta de 10 minutos ao *Browser de Classes* do ambiente Smalltalk nos possibilitou resolver este exercício. Como a classe *CompiledCode* já possuia um método (apresentado na hierarquia abaixo) que retorna um conjunto com as mensagens chamadas em um código Smalltalk, o que este exercício pede já estava pronta no próprio ambiente. Devido a estas rezões, as tabelas não foram preenchidas, pois os resultados de dados como tempo de codificação, tempo de teste, número de defeitos gerados, etc., seriam todos zero.

```
Kernel-Methods
   CompiledCode
      accessing-literals
         messages


messages
   "Answer a Set of all the message selectors sent by this method."

   | scanner selectorSet selector |
   selectorSet := Set new.
   self withAllBlockMethodsDo:
   [:meth |
   scanner := InstructionStream on: meth.
   scanner
      scanFor:
         [:byte |
         selector := scanner peekForSelector.
         selector == nil ifFalse: [selectorSet add: selector].
         false "keep scanning"]].
   ^selectorSet
```

A.2.3 Exercício 3A

Como o Exercício 2A, este também já estava pronto no próprio ambiente Smalltalk. O método *withAllBlockMethodsDo:* retorna uma coleção com os métodos utilizados num bloco de código Smalltalk. Da mesma forma que no Exercício 2A, as tabelas não foram preenchidas. A hierarquia e o código do método são apresentados abaixo:

```
Kernel-Methods
   CompiledCode
      enumerating
         withAllBlockMethodsDo:
```

```
withAllBlockMethodsDo: aBlock
   "Recursively enumerate the collection of methods
   (including block methods) that are included in
   this method."

   aBlock value: self.
   self literalsDo:
      [:lit | | cl |
      cl := lit class.
      cl == CompiledBlock
         ifTrue: [lit withAllBlockMethodsDo: aBlock]
         ifFalse:
            [cl == BlockClosure
            ifTrue: [lit method withAllBlockMethodsDo: aBlock]]]
```

A.3 Soluções Java

A.3.1 Exercício 1A

A implementação deste exercício em Java foi baseada na classe *Vector*. Como é numa lista encadeada que os elementos são armazenados, tivemos que definir a classe *List* como subclasse de *Vector* (vide código abaixo). Esta foi uma opção nossa.

```
import java.util.Vector;
import java.lang.Integer;
import java.lang.Math;

class List extends Vector {

   protected Vector list;
   static int index = 0; // para armazenar o indice corrente do elemento

   // costrutor da lista
   public List() {
      list = new Vector();
   }

   // acrescenta um elemento ao final da lista
   public void add(Double elem) {
      list.addElement(elem);
   }

   // para retornar os elementos da lista
   public Vector returnList() {
```

```
      return list;
   }

   // verifica se a lista ainda possui elementos
   public boolean hasMoreElem() {
      int tam = list.size();
      if ((index < tam) & (index >= 0))
         return true;
      else {
         index = 0;
         return false;
      }
   }

   // retorna o proximo elemento da lista
   public Double nextElem() {
      return (Double) list.elementAt(index++);
   }

   // calcula a media entre os elementos da lista
   public double average() {
      double sum = 0;
      while(hasMoreElem()) {
         sum = sum + nextElem().intValue();
      }
      return (sum/list.size());
   }

   // calcula o somatorio dos quadrados do desvio medio de cada
   // elemento da lista
```

```
   public double meanDeviation() {
      double sum, avg;
      double elem;

      sum = 0;
      avg = average();

      while(hasMoreElem()) {
         elem = nextElem().doubleValue();
         sum = sum +  Math.pow(elem - avg, 2);
      }
      return sum;

   }
   // calcula o desvio padrao dos elemntos da lista
   public double standarDeviation() {
      Vector aux = list;
      return Math.sqrt(meanDeviation()/(list.size()-1));
   }
}
```

Para este exemplo foram obtidos os valores ilustrados nas tabelas C14, C16 e C18. O teste do exemplo foi executado com o código abaixo, utilizando os dados da tabela D5.

```
class Exercicio {

   // Teste para o PSP 1A
```

```
public static void main(String args[]) {

   Double y1 = new Double(186); Double y2 = new Double(699);
   Double y3 = new Double(132); Double y4 = new Double(272);
   Double y5 = new Double(291); Double y6 = new Double(331);
   Double y7 = new Double(199); Double y8 = new Double(1890);
   Double y9 = new Double(788); Double y10 = new Double(1601);

   Double z1 = new Double(15.0); Double z2 = new Double(69.9);
   Double z3 = new Double(6.5); Double z4 = new Double(22.4);
   Double z5 = new Double(28.4); Double z6 = new Double(65.9);
   Double z7 = new Double(19.4); Double z8 = new Double(198.7);
   Double z9 = new Double(68.8); Double z10 = new Double(138.2);

   Double x1 = new Double(160); Double x2 = new Double(591);
   Double x3 = new Double(114); Double x4 = new Double(229);
   Double x5 = new Double(230); Double x6 = new Double(270);
   Double x7 = new Double(128); Double x8 = new Double(1657);
   Double x9 = new Double(624); Double x10 = new Double(1503);

   List objectLOC = new List();
   List totalLOC = new List();
   List developHours = new List();

   objectLOC.add(x1); objectLOC.add(x2);
   objectLOC.add(x3); objectLOC.add(x4);
   objectLOC.add(x5); objectLOC.add(x6);
   objectLOC.add(x7); objectLOC.add(x8);
   objectLOC.add(x9); objectLOC.add(x10);
```

```
        System.out.println("objectLOC.standarDeviation() = " +
        objectLOC.standarDeviation());

        totalLOC.add(y1); totalLOC.add(y2);
        totalLOC.add(y3); totalLOC.add(y4);
        totalLOC.add(y5); totalLOC.add(y6);
        totalLOC.add(y7); totalLOC.add(y8);
        totalLOC.add(y9); totalLOC.add(y10);


        System.out.println("totalLOC.standarDeviation() = " +
        totalLOC.standarDeviation());

        developHours.add(z1); developHours.add(z2);
        developHours.add(z3); developHours.add(z4);
        developHours.add(z5); developHours.add(z6);
        developHours.add(z7); developHours.add(z8);
        developHours.add(z9); developHours.add(z10);


        System.out.println("developHours.standarDeviation() = " +
        developHours.standarDeviation());

        }
}
```

Time in Phase(min.)	**plan**	**Actual**	**To Date**	**To Date %**
Planning		20	20	12.1
Design		20	20	12.1
Code		75	75	45.5
Test		40	40	24.2
Postmortem		40	40	6.1
Total	220	195	195	100.0
Defects Injected		**Actual**	**To Date**	**To Date %**
Planning		0	0	0
Design		2	2	9.1
Code		15	15	68.2
Test		5	5	22.7
Total Development		22	22	100.0
Defects Removed		**Actual**	**To Date**	**To Date %**
Planning		0	0	0
Design		1	1	4.5
Code		15	15	68.2
Test		6	6	27.3
Total Development		22	22	100.0

Table C14 PSP0 *Project Plan Summary*

Date	**Start**	**Stop**	**Interruption Time**	**Delta Time**	**Phase**	**Comments**
12/11	08:05	08:25	0	20 min	Planning	Estudo (Vector)
	10:20	11:00	0	20 min	Design	Teste (Vector)
	16:20	17:30	50 min	20 min	Coding	Reunião...
	19:05	19:30	0	25 min	Coding	Testes parciais
	20:00	20:45	15 min	30 min	Coding	Lanche ...
13/11	09:10	09:50	0	40 min	Test	
	10:30	11:10	0	40 min	Postmortem	Preenchendo tabelas

Table C16 *Time Recording Log*

Table C18 PSP0 *Defect Recording Log*

Student Jones Albuquerque Date 13/11/96

Program Desvio padrão Program# 1A

Date	Number	Type	Inject	Remove	Fix Time	Fix Defect
12/11/96	1	40	design	design	2 min	

Description: declaração de método em classe errada

Date	Number	Type	Inject **Phase**	Remove **Phase**	Fix Time	Fix Defect
12/11/95	15	20	code	code	6 min	

Description: erros de digitação

Date	Number	Type	Inject **Phase**	Remove **Phase**	Fix Time	Fix Defect
12/11/96	1	40	design	teste	4 min	

Description: utilização ineficiente de método

Date	Number	Type	Inject **Phase**	Remove **Phase**	Fix Time	Fix Defect
13/11/96	5	50	teste	teste	1 min	

Description: nomes de métodos e estrutura errados

A.3.2 Exercício 2A

A implementação deste exercício em Java foi baseada na classe *StringTokenizer*. O arquivo fonte, considerado um objeto da classe *String*, é a entrada para o contador de linhas de código exigido por este exercício. A contagem das linhas de código seguiu o padrão de tabulação definido na tabela *Table C29 PSP0.1-JOA Java Coding Standard* (vide Apêndice C), desconsiderando os espaços em branco, os comentários e as linhas que só possuiam "}". O código fonte e as tabelas PSP0.1[1] correspondentes a este exercício são apresentados a seguir:

```
/* Classe que recebe uma String no padrao PSP para
   Java (vide tabela C29) e retorna o numero de linhas de codigo,
   eliminando espacos em branco e comentarios. */

import java.util.StringTokenizer;

class Loc {

   protected String javaCode;
   boolean inCommentState = false;

   public Loc(String string) {
      javaCode = string;
   }

   /* metodo que retorna o numero de linhas fisicas de codigo */
   public int lineCount() {
```

[1] A Tabela C27 PIP *Process Improvement Proposal* não está apresentada neste exercício. O seu conteúdo foi extenso o suficiente para motivar este trabalho.

```
      StringTokenizer st = new StringTokenizer(javaCode,"\n\r");
      String lineCode = "";
      int lineNumber = 0;

      while (st.hasMoreTokens()) {

         lineCode = st.nextToken();

         if (isFuncLine(lineCode) && !isCommentLine(lineCode)) {
            lineNumber++;
         }
      }
      return lineNumber;
   }

  // ----------------------------------------
   /* metodo que retorna se eh uma linha logica de codigo */
   private boolean isFuncLine(String line){

      if (line.endsWith(";") || line.endsWith("{")){
         return true;
      } else return false;
   }
  // ----------------------------------------
   /* metodo que retorna se a linha faz parte de um comentario */
   private boolean isCommentLine(String line){

      if (line.startsWith("//")) {
         return true;
      }
```

```
      if (string1PossuiString2(line, "/*")) {
         inCommentState = true;
      }

      if (string1PossuiString2(line, "*/")) {
         inCommentState = false;
         return true;
      }
      return inCommentState;
   }
  // ----------------------------------------
   /* metodo auxiliar para verificar a presenca do string2 no string1 */
   private boolean string1PossuiString2(String string1, String string2){

      if (string1.indexOf(string2, 0) >= 0) { // sempre a partir do comeco de string1
         return true;
      } else return false;
   }
}
```

Como sugerido no enunciado, os testes foram realizados contando as linhas de código dos exercícios 1A e 2A.

Program Number	**LOC (Line Of Code)**
1A	37
2A	38

Table D6 TEST RESULTS FORMAT - PROGRAM 2A

Summary	**Plan**	**Actual**	**To Date**	
Program Size (LOC)	**Plan**	**Actual**	**To Date**	
Base(B)		15		
Deleted(D)		6		
Modified(M)		8		
Added(A)		24		
Reused(R)		5	5	
New and Changed (N)	20	32	69	
Total LOC (T)		38	75	
Total New Reused		0	0	
Time in Phase(min.)	**plan**	**Actual**	**To Date**	**To Date %**
Planning	25	20	40	7.8
Design	30	65	85	16.5
Code	90	150	225	43.7
Compile	15	15	15	2.9
Test	50	30	70	13.6
Postmortem	50	40	80	15.5
Total	260	320	515	100.0

Table C14 PSP0.1 *Project Plan Summary* (Continued)

Defects Injected	**Actual**	**To Date**	**To Date %**
Planning	0	0	0
Design	1	3	9.1
Code	12	27	68.2
Compile	2	2	2.9
Test	2	7	22.7
Total Development	17	39	100.0
Defects Removed	**Actual**	**To Date**	**To Date %**
Planning	0	0	0
Design	1	2	5.1
Code	10	25	64.1
Compile	2	2	5.1
Test	4	10	25.7
Total Development	17	39	100.0

Table C14 PSP0.1 *Project Plan Summary*

Date	**Start**	**Stop**	**Interruption Time**	**Delta Time**	**Phase**	**Comments**
09/01	16:50	17:10	0	20 min	Planning	
	17:10	18:25	10 min	65 min	Design	Telefone
	20:30	22:45	30 min	105 min	Coding	Lanche
	23:15	00:15	0	06 min	Coding	
13/11	07:10	07:40	0	30 min	Teste	
	08:15	09:00	5 min	40 min	Postmortem	Gerais

Table C16 *Time Recording Log*

Table C18 PSP0 *Defect Recording Log*

Student Jones Albuquerque Date 10/01/96

Program Contador de linhas de código Program# 2A

Date	Number	Type	Inject	Remove	Fix Time	Fix Defect
09/01/97	1	100	design	design	1 min	

Description: consulta à classe errada

Date	Number	Type	Inject **Phase**	Remove **Phase**	Fix Time	Fix Defect
09/01/97	10	20	code	code	8 min	

Description: vários erros de digitação

Date	Number	Type	Inject **Phase**	Remove **Phase**	Fix Time	Fix Defect
09/11/97	2	20	compile	compile	1 min	

Description: utilização ineficiente de método

Date	Number	Type	Inject **Phase**	Remove **Phase**	Fix Time	Fix Defect
10/01/97	2	50	code	teste	2 min	

Description: nomes errados de métodos

Date	Number	Type	Inject **Phase**	Remove **Phase**	Fix Time	Fix Defect
10/01/97	2	40	teste	teste	2 min	

Description: nomes errados de variáveis

Apêndice B

JavaEdit

Na versão disponível atualmente (Beta), o JavaEdit possui todas as funções comuns de um editor de propósito geral mais uma opção de FORMAT que elimina a tarefa de tabulação (*indenting*) e padronização de código do programador. A opção de PSPFORMAT fornece uma padronização de código de acordo com o modelo PSP (vide tabela C29 PSP0.1-JOA *Java Coding Standard*).

B.1 Descrição Geral

A implementação de JavaEdit é composta de cinco classes [GJS96], responsáveis por frações de funcionalidade do editor. A descrição, por classe, do código é realizada nas seções que se seguem.

JavaEdit (Versão Beta) possui as seguintes funções disponíveis na barra de menus:

- File
  - New, Open, Save, Save as, Exit.
- Edit

  - Cut, Copy, Paste, Select All.

- Java

  - PSPFormat

B.1.1 Classes que compõem a implementação

As cinco classes que compõem JavaEdit são:

ContinueDialog.java Os objetos desta classe são janelas do tipo *pop-up* chamadas pelo editor para informar ao usuário pequenos erros cometidos por ele. Por exemplo, tentar salvar um texto que não foi alterado ou tentar selecionar a opção "Copy" do menu "Edit" sem antes ter selecionado uma área do texto.

JavaEdit.java Esta é a principal classe do editor. Ela é responsável pelo funcionamento do editor como um todo: cria as janelas, botões, menus; controla a ordem de execução; trata os eventos e entradas geradas pelo usuário; controla a gravação dos arquivos (a fim de que o usuário inavertidamente sobreponha arquivos sem gravar o anterior), etc. O funcionamento do JavaEdit pode ser descrito do seguinte modo: é criada a janela do editor, com área de texto e menu; a partir daí, o editor espera entradas do usuário (textuais ou escolha de opções do menu) e trata estas entradas, realizando as operações de acordo com a opção escolhida. Este processo segue até que o usuário escolha a opção "Exit" do menu "File".

JavaFormat.java Nesta classe, o código Java visível no ambiente de edição recebe identação e formato padrão de código definido, como mencionado, na tabela C29.
A implementação destas tarefas (tabulação e formatação) passa pelas seguintes etapas, mediante seleção da opção PSPFormat na barra de menus:

  - Eliminação da tabulação atual;

- Verificação da formação das linhas de código, respeitando o padrão de codificação para Java definido no PSP-JOA;
- Inclusão de tabulação padrão definida em PSP-JOA, ou seja, três espacos de tabulação para cada novo bloco de chaves aberto;
- Contagem das linhas de código eliminando linhas em branco e comentários.

Desta forma, temos um código padronizado e tabulado conforme as definições contidas no PSP-JOA.

OverwriteDialog.java Os objetos desta classe são janelas do tipo *pop-up* chamadas pelo editor para informar ao usuário que já existe um arquivo com o nome que ele deseja dar ao novo arquivo. Fornece ao usuário duas opções: sobrepor o arquivo existente com o novo arquivo ou cancelar a operação.

QuestionDialog.java Os objetos desta classe são janelas do tipo *pop-up* chamadas pelo editor para perguntar ao usuário se ele deseja salvar o arquivo atual antes de abrir um arquivo já existente ou um novo arquivo, caso o arquivo atual tenha sido editado após a última gravação. Oferece ao usuário três opções: gravar o arquivo atual, não gravar ou cancelar a operação.

B.2 Relevância

Este trabalho possui o importante objetivo de fornecer um ambiente com padronização automática de código. As Figuras B.1 e B.2 ilustram a interface e um exemplo de código não formatado e formatado pelo JavaEdit, respectivamente.

Isto, no contexto de PSP definido por Humphrey, significa passar do nível 0 para o nível 0.1 automaticamente, reduzindo todos os custos e esforços necessários para esta evolução.

B.3 Considerações

O Mini Java Editor, software de domínio público, disponível na URL

http://panda.iss.nus.sg:8000/alp/java/mje.html

é escrito em Java e, apesar de não funcionar, foi útil para o nosso trabalho de forma orientadora e esclarecedora; sem o qual, com certeza, teríamos muito mais dificuldades na nossa implementação.

O JavaEdit contou com a participação de Julianne Pepeu (http://www.di.ufpe.br/~jfsp) na implementação do código.

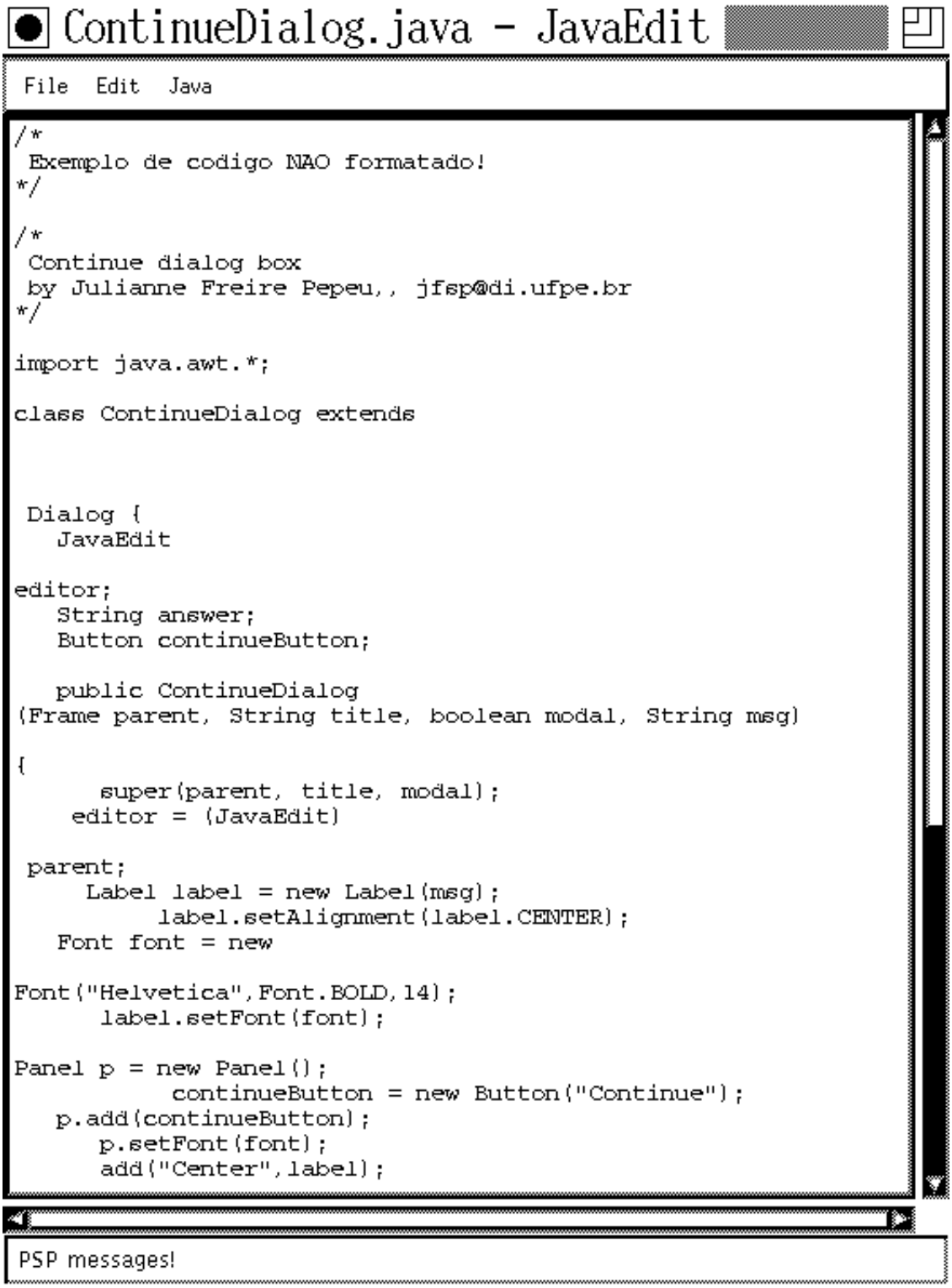

Figura B.1: JavaEdit: código não formatado.

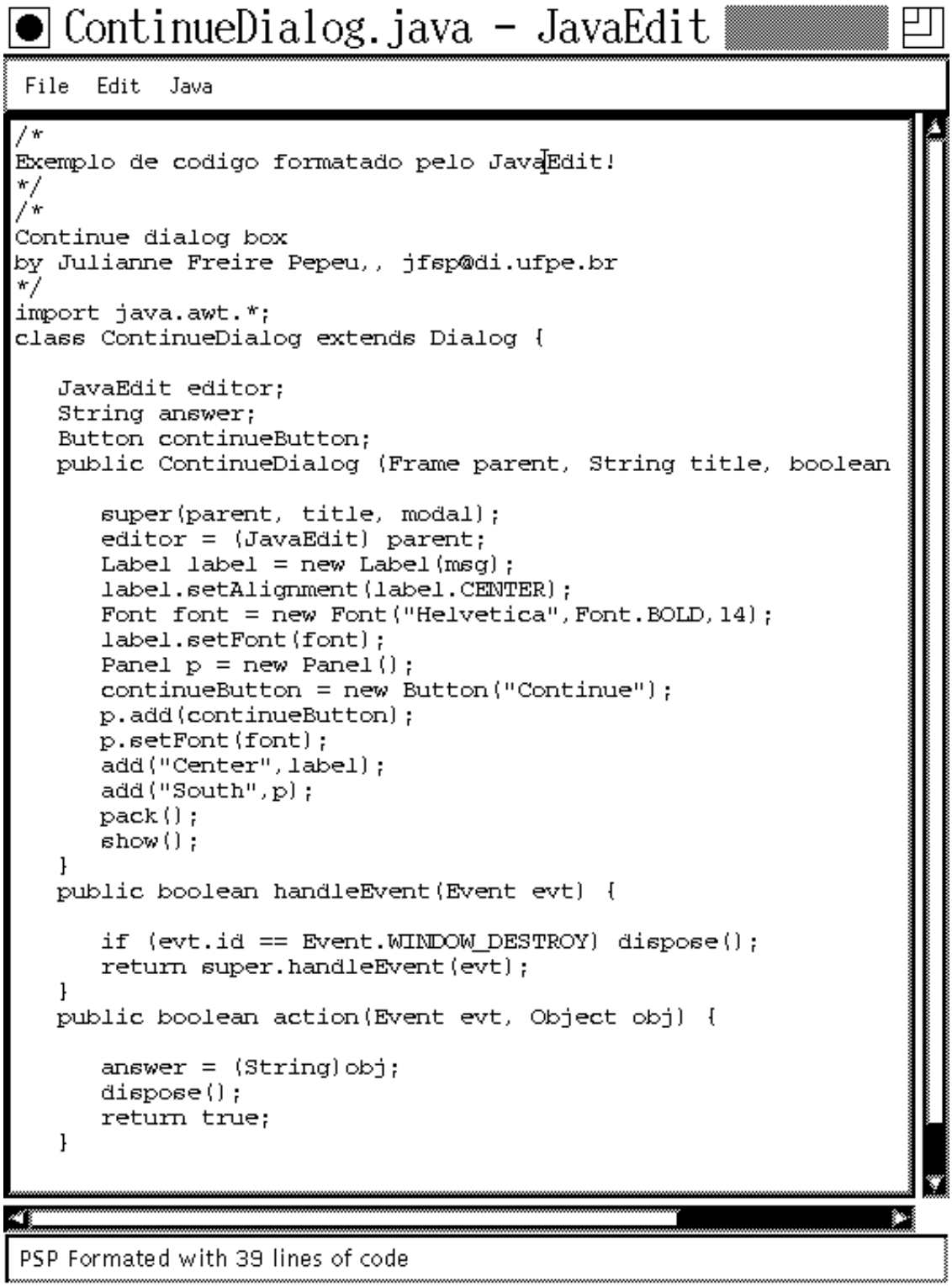

Figura B.2: JavaEdit: código formatado.

Apêndice C

Tabelas PSP-JOA

Apenas as tabelas que necessitavam de alterações estão descritas aqui. Optamos por escrever em inglês para manter a uniformidade entre as tabelas PSP-JOA e as tabelas PSP.

As seções a seguir possuem as tabelas modificadas, classificadas por nível PSP. As modificações em relação às originais estão escritas em **negrito**, mantendo a padronização de Humphrey em fonte SMALL CAPS para as evoluções em cada fase. As tabelas relativas a *Design Review Checklist* e *Code Review Checklist* mesmo voltadas a C++ podem ser aproveitadas para Java.

C.1 Tabelas Modificadas

Entre todas as tabelas utilizadas no modelo PSP, ressaltamos quais foram modificadas para PSP-JOA:

- PSP0-JOA: C11+, C12+ e C18+;
- PSP0.1-JOA: C22, C23, C26+ e C29+. Um padrão para codificação em Java é apresentado pela tabela C29. O padrão é resultado de uma miscelânea entre [Hum95a, Lea96] e Javadoc (*The Java API Documentation Generator*);

- PSP1-JOA: C31, C32 e C35.
- PSP1.1-JOA: C42, C43, C46 e C48+;
- PSP2-JOA: C52, C53 e C56;
- PSP2.1-JOA: C60, C61 e C64;
- PSP3-JOA: C75+, C78+ e C81.

As tabelas assinaladas com "+" são apresentadas a seguir, as demais são apenas evoluções de uma dessas assinaladas. As modificações acrescentadas por PSP-JOA são propagadas das tabelas assinaladas para as demais.

C.1.1 PSP0-JOA

Phase No.	Purpose:	To guide the PSP planning process.
	Entry Criteria	- Problem decription. - Project Plan Summary form. - Time recording log.
1	Program Requirements	- Produce or obtain a requirements statement for the program. - Ensure that the requirements statement is clear and unambiguos, **making a informal specification** - Resolve any questions.
2	Estimate Resources	- Make your best estimate of the time required to develop this program. - **Compute the time required to localize the convenient classes, interfaces as well, and packages in Java API or in available libraries.** - Distribute the development time over the planned project phases.
	Exit Criteria	- A documented requirements statement. - A project plan summary with estimated development time data. - Completed time log.

Table C11 PSP0-JOA *Planning Script*

	Purpose	To guide the development of small programs.
	Entry Criteria	- Requirements statement. - Project plan summary with planned development time. - Time and defect recording logs. - Defect type standard.
1	Design	- Review the requirements and produce a design to meet them. - **Compute the time used to draft the main classes.** - Record time in time log.
2	Code	- Implement the design. - Record any requirements or design defects found in the defect recording log. - Record time in time log.
3	Compile	- Compile the program until error free. - Fix all defects found. - Record defects in defect log. - Record time in time log.
4	Test	- Test until all tests run without error. - Fix all defects found. - Record defects in defect log. - Record time in time log.
	Exit Criteria	- A thoroughly tested program. - Completed defect log. - Completed time log.

Table C12 PSP0-JOA *Development Script*

Table C18 PSP0-JOA *Defect Recording Log*

Student _____ Date _____

Program _____ Program# _____

Date	Number	Type	Inject **Phase**	Remove **Phase**	Fix Time	Fix Defect

Description: _____

Date	Number	Type	Inject **Phase**	Remove **Phase**	Fix Time	Fix Defect

Description: _____

Date	Number	Type	Inject **Phase**	Remove **Phase**	Fix Time	Fix Defect

Description: _____

Date	Number	Type	Inject **Phase**	Remove **Phase**	Fix Time	Fix Defect

Description: _____

C.1.2 PSP0.1-JOA

Purpose	This form holds the estimated and actual project data in a convenient and readily retrievable form.
Header	Enter the following: - your name and today's date. - the program name and number. - the instructor's name. - the language you used to write the program.
Program Size (LOC)	PRIOR TO DEVELOPMENT: - IF YOU ARE MODIFYING OR ENHANCING AN EXISTING PROGRAM, COUNT THAT PROGRAM'S LOC, **showing the difference between programs from Java API (BJA) and any other font (BOF)**, AND ENTER **this sum (BJA + BOF)** AS BASE-ACTUAL. - USING YOUR BEST JUDGMENT, ESTIMATE THE NEW AND CHANGED LOC YOU EXPECT TO DEVELOP. AFTER DEVELOPMENT: - IF THE BASE LOC (B) HAS CHANGED, ENTER THE NEW VALUE. - MEASURE THE TOTAL PROGRAM SIZE AND ENTER IT AS THE TOTAL LOC - ACTUAL (T). - REVIEW YOUR SOURCE CODE AND, WITH THE HELP OF PROGRAM 3A, DETERMINE THE ACTUAL LOC THAT WERE DELETED (D), MODIFIED (M), OR REUSED (R). - CALCULATE THE LOC OF ADDED CODE AS A = T - B + D - R. - CALCULATE THE TOTAL NEW AND CHANGED LOC AS N = A + M.

Table C26 PSP0.1-JOA *Project Plan Summary Instructions* **(Continued)**

Time in Phase	- Under Plan, enter your original estimate of the total development time AND THE TIME REQUIRED BY PHASE. - Under Actual, enter the actual time in minutes spent in each development phase. - Under To Date, enter the sum of the actual time and the To Date time from your most recently developed program. - Under To Date %, enter the % of To Date time in each phase.
Defects Injected	- Under Actual, enter the number of defects injected in each phase. - Under To Date enter the sum of the actual numbers of defects injected in each phase and the To Date values from the most recently developed program. - Under To Date %, enter the % of To Date defects injected by phase.
Defects Removed	- Under Actual, enter the number of defects removed in each phase. - Under To Date enter the sum of the actual number of defects removed in each phase and the To Date value from the most recently developed program. - Under To Date %, enter the % of To Date defects removed by phase. - After development, record any defects later found during program use, reuse, or modification.

Table C26 PSP0.1-JOA *Project Plan Summary Instructions*

Purpose	To guide the development of Java programs
Program Headers	Begin each file with a history table listing dates, authors, and summaries of changes.
File Hearder Format	/* File: the file name Date Author Changes sep 1 95 Doug Lea Created jul 2 96 Joa Added new concepts */

Class Header
 	
Class Documentation Format
	Place each class in a separate file. Preface each class with a /** ... **/ comment using javadoc conventions.
	/**
	* A class representing a window on the screen.
	* For example:
	* <pre>
	* Window win = new Window(parent);
	* win.show();
	* </pre>
	*
	* @see awt.BaseWindow
	* @see awt.Button
	* @version 1.2 12 Dec 1994
	* @author Sami Shaio
	**/
	
	Class Window extends BaseWindow {
	...
	}

Table C29 PSP0.1-JOA *Java Coding Standard* **(Continued)**

Instance Var Documentation	Use javadoc conventions to describe nature, purpose, constraints and usage.
Instance Var Doc Format	/** * The X-coordiate of the window * @see window#1 **/ int x= 1263732

Method Documentation
 	
Method Documentation Format
	Use Javadoc conventions to describe nature, purpose, preconditions, effects, algorithmic notes, usage instructions, reminders, reuse instructions, etc ...
	/**
	* Return the caracter of the specified index. An index ranges
	* from <tt>0</tt> to <tt>length() - 1</tt>.
	* @param index The index of the desired character
	* @return The desired character
	* @exception StringIndexOutOfRangeException When the
	* index is not in the range <tt>0</tt> to
	* <tt>length() - 1</tt>
	*
	**/
	
	public char charAt(int index) {
	...
	}

Table C29 PSP0.1-JOA *Java Coding Standard* **(Continued)**

Identifiers	Use descriptive names for all variables, function names, constants, and other identifiers. Avoid abbreviations or single letter variables. (vide Capitalization)
Example	int number_of_students; /* This is GOOD */ float x4, j, ftave; /* These are BAD */
Internal Block Comments	Use /* ... */ comments to describe algorithmic details, notes, and related documentation that spans more than a few code statements.
Example	/* * strategy: * 1. Find the node * 2. Clone it * 3. Ask inserter to add clone * 4. If successful, delete node */
Internal Running Comments	Use // comments to clarify non-obvious code.
Good Comment	if (record_count) /* have all records been processed? */
Bad Comment	if (record_count) /* check if record_count is grater than limit */

Table C29 PSP0.1-JOA *Java Coding Standard* **(Continued)**

Identing	Indent every level of bracket from the previous one. If you use the JavaEdit Editor, this indenting is made automaticly by option Java-PSPFormat menu bar (vide Appendix B).

Example	public class Scanner {
	/**
	* Basic elements recognized by the scanner.
	* White spaces are ignored.
	*/
	static int NUMBER = 0; // an unsigned number
	static int IDENT = 1; // an identifier
	static int ERROR = 5; // error (unrecognized element)
	
	/**
	* A dictionary mapping the delimiters and operators
	* into their token types. This allows easy extension
	* of this class.
	*/
	public static Hashtable TokenMap = new Hashtable();
	
	static int i=0;
	static {
	for (i=0;i < delimiters.length;i++) {
	TokenMap.put(delimiters[i],new Integer(DELIM));
	}
	for (i=0;i < operators.length;i++) {
	TokenMap.put(operators[i],new Integer(OPER));
	}
	}
	}

Table C29 PSP0.1-JOA *Java Coding Standard* **(Continued)**

Capitalization	Try to stick to the following naming conventions: • packages: lowercase • classes: CapitalizedWithInternalWordsAlsoCapitalized • methods: firstWordLowerCaseButInternalWordsCapitalized • constants (finals): UPPER_CASE_WITH_UNDERSCORES • private or protected members: trailingUnderscore_ • method local variables: lower_case_with_underscores • factory methods: newX • converter methods: toX • method that reports an attribute x of type X: X x() • method that changes an attribute x of type X: void x(X value)

Table C29 PSP0.1-JOA *Java Coding Standard*

C.1.3 PSP1.1-JOA

Purpose:	- To estimate the development time for each project task. - To compute the planned value for each project task. - To estimate the planned completion date for each task. - To provide a basis for tracking schedule progress even when the tasks are not completed in the planned order.
General:	- Expand this template or use multiple pages as needed. - Include every significant task. - Use task names and numbers that support the activity and are consistent with the project work breakdown structure.
Header	Enter the following: - your name. - today's date. - the instructor's name. - the program number.
Task	Enter a task number and name. - List the tasks in the order in which you expect to complete them. Select tasks that have explicit completion criteria. - i.e., **localize the convenient classes, interfaces as well, and packages in the Java API or in available libraries to reuse, classify the classes to be developed,** planning completed, program compiled and all defects corrected, testing completed and all defects corrected, **packages created and organized,** etc.
Plan - Hours	- Enter the planned hours for each task.

Table C48 PSP1.1-JOA *Task Planning Template Instructions* **(Continued)**

Plan - Planned Value	- Total the planned hours for all the tasks. - For each task, calculate the percent its planned hours are of total hours. - Enter this percent as the planned value for that task. - The total planned value should equal 100.
Plan - Cumulative Hours	- Enter the cumulative sum of the plan hours down through each task.
Plan - Cumulative Value	- Enter the cumulative sum the planned values down through each task. - Complete the schedule planning template down through Plan-Cumulative Hours before proceeding. - Then complete the schedule and task planning templates together.
Plan Date - Monday	- For each cumulative hours entry, find the plan cumulative hours entry on the schedule planning template that equals or just exceeds it. - Enter the date from that row (of the schedule planning template) as the plan date on the task planning template. - If several weeks on the schedule template have the same cumulative value, enter the earliest date. - Unless you made daily plans, pick the plan date as the Monday of the week during which completion for that task is planned.
Actual Date	- As each task is completed, enter the completion date.
Earned Value Cumulative Earned Value	- For each completed task, enter the planned value. - As each task is completed, total all the earned value entries and enter the total beside the latest that was completed.

Table C48 PSP1.1-JOA *Task Planning Template Instructions*

C.1.4 PSP3-JOA

Phase No.	Purpose:	To guide the PSP planning process.
	Inputs Required:	Problem decription. PSP3 Project Plan AND CYCLE Summary form. Size estimating, task planning, and schedule planning templates. Historical estimated and actual size and time data. Time recording log.
1	Program Requirements	- Produce or obtain a requirements statement for the program. - Ensure that the requirements statement is clear and unambiguos. - Resolve any questions.
2	Size Estimate	- Produce a program conceptual design. - Use the PROBE method to estimate of the new and changed LOC required to develop this program. - Estimate the base, added, deleted, modified, and reused LOC. - Complete the size estimating and project plan summary forms. - Calculate the 70% size prediction interval (you may use a spreadsheet).
3	CYCLIC DEVELOPMENT STRATEGY	- SUBDIVIDE THE PROGRAM DEVELOPMENT INTO MODULES ABOUT 100 AND NO MORE THAN 250 NEW AND CHANGED LOC.

Table C75 PSP3-JOA *Planning Script* **(Continued)**

4	Estimate Resources	- Use PROBE method to estimate the time required to develop THE NEW program. - Calculate the 70% time prediction interval (you may use a spreadsheet). **- Compute the time required to locate the convenient classes in Java API for each module.** - SUBDIVIDE THIS TOTAL DEVELOPMENT TIME AMONG THE DEVELOPMENT CYCLES. - Using the To Date % from the most recently developed program as a guide, distribute the development time over the planned project phases OF EACH DEVELOPMENT CYCLE.
5	Task and Schedule Planning	- For projects requiring several days or more of work, complete the task and schedule planning forms.
6	Estimate Defects	- Based on your To Date data on defects per new and changed LOC, estimate the total defects to be found in this program. - Based on your To Date % data, estimate the numbers of defects to be injected and removed by phase.
	Exit Criteria	- A documented requirements statement. - The program conceptual design. - A completed size estimating form. - For projects of several days duration, completed task and schedule planning forms. - Project AND CYCLE plan summary with estimated program size and development time, and 70% prediction intervals. - Completed time log.

Table C75 PSP3-JOA *Planning Script*

	Purpose:	To guide the development of small programs.
	Entry Criteria	- Requirements statement. - Project plan summary with planned development time. - For projects of several days duration, completed task and schedule plans. - Time and defect recording logs. - Defect type and coding standard.
1	MODULE Design	- Review the MODULE requirements and produce an external specification to meet them. **- Compute, for each module, the time used to draft the main classes.** - Complete function specification and operational scenario templates to record this specification. - Produce a design to meet these specifications. - Record the design in functional specification, operational specification, state specification, and logic specification templates as required. - COMPLETE THE DESIGN FOR THE MODULE TEST MATERIALS AND FACILITIES. - Record time in time log.
2	Design Review	- Following the checklist, review the MODULE AND TEST MATERIALS designs. - Fix all defects found. - Record defects in defect log. - Record time in time log.

Table C78 PSP3-JOA *Development Script* **(Continued)**

3	Code	- Implement the design, following the coding standard. - Record any requirements or design defects found in the defect recording log. - Record time in time log.
4	Code Review	- Following the checklist, review the MODULE AND TEST code. - Fix all defects found. - Record defects in defect log. - Record time in time log.
5	Compile	- Compile the program AND TEST MATERIALS until error free. - Fix all defects found. - Record defects in defect log. - Record time in time log.
6	Test	- Test THE MODULES until all tests run without error. - Fix all defects found. - Record defects in defect log. - Record time in time log. - Complete a test report on the tests conducted.
7	REASSESS AND RECYCLE	- RECORD DATA ON THE DEVELOPMENT CYCLE. - REASSESS THE STATUS AGAINST PLAN AND DECIDE TO CONTINUE AS PLANNED OR MAKE CHANGES.
	Exit Criteria	- A throughly tested program that conforms to the coding standard. - Complete design specification templates. - Complete design and code review checklists. - UPDATED CYCLE SUMMARY WITH ACTUAL DATA. - Completed test report. - Completed defect log. - Completed time log.

Table C78 PSP3-JOA *Development Script*