GILBERTO MOREIRA MARTINS

DOCUMENTANDO OS ARTEFATOS DE PROJETO NO PRÓPRIO CÓDIGO: ESTUDO DE FERRAMENTAS NO AMBIENTE JAVA

Monografia apresentada à Escola Politécnica da Universidade de São Paulo para conclusão do curso de MBA em Engenharia de Software

Área de Concentração:
Engenharia de Software

Orientador:
Professora Maria Alice Ferreira

SÃO PAULO
2003

RESUMO

A figura da documentação, no escopo do desenvolvimento de sistemas informatizados, é apresentada como um meio para garantir o registro de todo o conhecimento do produto e a comunicação efetiva entre todos os envolvidos nas diversas fases do desenvolvimento. Sua relação com o processo de desenvolvimento permite definir sua forma e conteúdo. As qualidades exigidas e as dificuldades na sua produção também são enfocadas neste estudo. Técnicas e ferramentas *open source* existentes no ambiente de programação Java para a documentação de sistemas informatizados são descritas. Tais técnicas, de uma forma ou de outra auxiliam a incluir a documentação dentro do próprio código fonte, como uma maneira simples, flexível e de baixo custo, para permitir o sincronismo entre as alterações do código e a documentação.

SUMÁRIO

Lista de Figuras:

Lista de Tabelas:

1 INTRODUÇÃO

Projeto e programação são atividades humanas;
esqueça isso e tudo estará perdido.
Bjarne Stroustrup

... documentação que não tem sido usada antes de ser publicada, documentação que não é importante para o seu autor, sempre será uma documentação ruim.
Paul C. Clements

Este trabalho visa apresentar as dificuldades do desenvolvimento de software oriundas da falta ou desatualização da documentação, principalmente na fase de manutenção. Visa também dar uma visão introdutória das metodologias e ferramentas existentes no ambiente Java, muitas do tipo *open source* ou baseadas em estudos acadêmicos, enfatizando as vantagens de manter a documentação no próprio código fonte como forma de complementar e ajudar a manter atualizada a documentação, pela sua proximidade com a implementação.

1.1 Motivação

A grande parte do tempo despendido na minha atividade como desenvolvedor, e na de outros desenvolvedores dentro da empresa em que trabalho, pode ser resumida em duas atividades: acrescentar novas funcionalidades a um sistema de software e realizar correção de "bugs". Minha empresa possui um sistema computacional principal e praticamente 95% do trabalho da área de sistemas está relacionado a esse produto: instalação, suporte, treinamento, correções, acréscimo de funcionalidades e suporte a novo hardware.

As duas atividades citadas enfatizam uma única fase do ciclo de vida do software: a manutenção. O problema é que o número de correções e de novos recursos

tem crescido e o tempo gasto - e o correspondente custo - com cada um deles também tem crescido, tornando-se um grande gargalo para nossa área. Resta-nos duas alternativas: contratar mais desenvolvedores ou melhorar a produtividade dos atuais.

No fundo, o problema não é tanto da qualidade dos profissionais, mas de alguns aspectos da qualidade do produto: manutebilidade é um deles. Em van Deursen (2001), é apresentado que 40 a 60% do tempo gasto com manutenção está relacionado com a compreensão do software. Assim, melhorando a qualidade interna do produto - aqueles aspectos que não são vistos diretamente pelo cliente -, melhorando sua compreensão e diminuindo a possibilidade de defeitos, estar-se-á melhorando a produtividade na fase de manutenção. O caminho escolhido é, portanto, buscar o aumento da produtividade na manutenção com uma diminuição conjunta na ocorrência de defeitos a fim de reduzir o gargalo do desenvolvimento, liberando recursos para focar o acréscimo de novas funcionalidades para a conquista de mercado.

1.2 Objetivos

O objetivo deste trabalho é o enfoque no problema da compreensão do produto, e a conseqüente elaboração da documentação, como forma de garantir a manutenção de todo o conhecimento sobre um sistema de software e uma maneira efetiva de comunicação entre os diversos elementos envolvidos nas diversas fases do desenvolvimento.

Esse trabalho serve de ponto de partida sobre o que pode ser feito em projetos de software para desenvolver código que custe menos para ser mantido.

Deve-se apresentar a documentação como algo simples de ser produzido, reaproveitável, e, de alguma forma, manter-se consistente com o produto desenvolvido. Deve-se ter em mente que a documentação também tem de evoluir em conjunto com o produto.

Este trabalho apresenta ferramentas existentes no ambiente Java, geralmente do tipo *free* e *open source*, e sua aplicação na manutenção da documentação no próprio código fonte, como forma de complementar e ajudar em manter atualizada a documentação pela sua proximidade com a implementação.

1.3 Metodologia

Para a elaboração deste trabalho, utilizou-se de pesquisa na Internet sobre o tema básico "gerenciamento de documentação", além da pesquisa sobre diversas ferramentas para a linguagem Java relacionadas a documentação, especificação formal e extensões da linguagem.

1.4 Estrutura do trabalho

Este trabalho constitui-se cinco capítulos, conforme se descreve a seguir.

O Capítulo 1 é formado por esta introdução, na qual se descrevem as motivações, objetivos, metodologia e estruturação da monografia.

Apresenta-se este trabalho introduzindo, no Capítulo 2, a importância da documentação e sua relação com o processo de desenvolvimento e com a qualidade do produto, os diversos artefatos que compõem o processo de produção e quais são mais adequados para determinadas audiências.

Em seguida, no capítulo 3, apresentam-se as mais diversas técnicas e ferramentas *open source* e baseadas em Java, usadas para a criação e manutenção da documentação dentro do próprio código, ou no sentido oposto, gerando o código a partir dos documentos de projeto, operando como uma linguagem de mais-alto-nível.

Finalmente, no Capítulo 4, mostram-se alguns exemplos de documentação, extraídos de um código fonte devidamente comentado.

No Capítulo 5 são apresentadas as considerações finais.

2 IMPORTÂNCIA DA DOCUMENTAÇÃO NO DESENVOLVIMENTO DE SISTEMAS

Programadores vêm e vão: o grupo inicial, que uma vez compreendeu o problema e as questões envolvidas, escreveu o código e se foi; novos programadores vieram, deixaram suas pequenas contribuições no código e também se foram. Eventualmente, nenhum indivíduo ou grupo conhece toda a abrangência do problema por trás do programa, as soluções que foram escolhidas, as que foram rejeitadas e porque isso ocorreu.

Essa constatação é claramente apresentada em Kaplan (2003). Mesmo que se tenha o código fonte na frente, existem limites que um leitor humano pode absorver de milhares de linhas concebidas fundamentalmente para exprimir uma funcionalidade, não para conter significado. Quando o conhecimento passa para o código, ele muda de estado, como quando a água passa para gelo, tornando-se uma coisa nova, com novas propriedades.

É necessário, assim, complementar o código com algo mais:

- "Porque foi escolhido se fazer desta forma? "

- "Quem tomou essa decisão e quando?"

- "Quais requisitos essa função implementa?"

- "Que outros elementos mais podem ser afetados por uma mudança nesse procedimento?"

São perguntas que também devem estar claras para quem tenta entender um programa, além daquela mais comum: "Para que serve?".

Em Arkley et al. (2002) é apresentada uma proposta de pesquisa para se conhecer melhor os problemas de rastreabilidade, tentando entender porque é tão difícil responder a questão sobre o impacto das alterações no resto do sistema.

2.1 Documentação de sistema: ferramenta ou estorvo?

Em Kaplan (2003), mostra-se que enquanto é verdade que qualquer programador experiente, fluente na linguagem utilizada para a construção do software,

pode entender o que o software faz, ele pode também levar um grande tempo para compreender porque ele foi estruturado daquela maneira. Comentários não estão lá para descrever as funções óbvias do código, mas para explicar a arquitetura do software e as interações dos componentes. Isso é que não é visto quando se lê um programa.

Os programas mais difíceis de entender são aqueles com alguns comentários e pouca documentação externa. É evidente que a documentação é a chave para se produzir software com uma manutenção mais fácil. Se o código tiver comentários claros e alguma documentação que descreva a arquitetura de alto nível, será possível entender mais rapidamente o que faz este código e corrigir os defeitos detectados em menos tempo. A solução não é tão simples, pois antes duas perguntas têm de ser esclarecidas:

- Qual a forma que a documentação deve ter?

- Como os analistas/programadores podem ser encorajados a escrever a documentação, o que eles relutam tanto em fazer?

Stout (2001) apresenta uma noção bem conhecida de que os desenvolvedores não são apaixonados por documentação. Eles acreditam que manter uma documentação rastreável e consistente produz como efeito colateral, uma diminuição na sua produtividade diária, além de impor uma burocracia adicional.

Outro ponto negativo é uma tendência de não saber quando parar. Algumas vezes podemos ser tão detalhistas na geração da documentação que o esforço resultante não é produtivo e impõe custo elevado para o projeto.

Parnas e Madey (1995) mostram que, em muitas organizações que desenvolvem software, a documentação não é vista como uma parte da atividade de projeto, mas como uma tarefa adicional, às vezes entediante, mas que deve ser completada por razões burocráticas. Com freqüência, os programas são escritos antes da documentação, e também, a documentação pode ser escrita por um grupo separado, que não inclui os projetistas. Usualmente, os programadores consideram a documentação que chega às suas mãos vaga e praticamente inútil. Conseqüentemente, os documentos de projeto de um sistema computacional são usualmente inapropriados, quando disponibilizados, e raramente são atualizados.

2.2 Relação da documentação com o processo de desenvolvimento utilizado

Em Booch et al. (1998), é apresentada uma noção verdadeira, mas no mínimo curiosa, sobre a necessidade de se impor um processo no desenvolvimento de software: o medo! Sim, medo basicamente de que:

- o projeto irá produzir o produto errado;
- o projeto irá produzir um produto de qualidade inferior;
- o projeto atrase;
- todos terão de trabalhar 80 horas por semana;
- todos terão que quebrar compromissos;
- não se terá prazer no que se faz.

Sob um outro ponto de vista também, Booch et al. (1998), apresentam que o objetivo de um processo de software é garantir o cumprimento do que é chamado "Relação de direitos dos clientes e desenvolvedores", cujos principais pontos são os Direitos do Desenvolvedor e os Direitos do Cliente.

Direitos do Desenvolvedor

- tem o direito de conhecer o que é solicitado, através de requisitos claros e com uma priorização definida;
- tem o direito de dizer quanto tempo cada requisito irá levar para ser implementado e de revisar essas estimativas através de sua experiência;
- tem o direito de aceitar as suas responsabilidades, ao invés de que elas lhe sejam atribuídas;
- tem o direito de produzir com qualidade durante todo o tempo;
- tem o direito de trabalhar em ambiente calmo, alegre, produtivo e agradável;

Direitos do Cliente

- tem o direito a um planejamento geral, conhecer o que pode ser resolvido, quando e a que custo;
- tem o direito de ver o progresso em um sistema executando, provando que funciona ao passar por testes repetitíveis que a equipe de desenvolvimento especificou;

- tem o direito de mudar a sua opinião, substituindo funcionalidade e alterando prioridades;
- tem o direito de ser informado sobre mudanças de cronograma, em tempo de escolher como reduzir o escopo para restaurar a data original. Pode cancelar o projeto a qualquer momento, e ficar com um sistema funcionando, que reflita o investimento até aquela data.

Para atender a esses direitos inerentes das duas partes principais envolvidas na elaboração de um produto de software, fica clara a necessidade de se registrar necessidades, decisões, cronogramas, custos e prioridades, a fim de que ambas as partes estejam cientes e de acordo com o que for estabelecido. O registro e comunicação desses itens consiste no papel principal da documentação em um processo de desenvolvimento de software.

Produzir essa documentação, no entanto, esbarra em uma série de dificuldades. Em DeMarco (1995), duas variantes são apresentadas, que mostram os extremos do problema da documentação:

a) A equipe produz toda a documentação interna que sabe que é necessária e "paga" um terrível preço por ela.

b) A equipe não produz toda a documentação que precisa e "paga" um terrível preço por isso.

Qual é a quantidade adequada de documentação depende, principalmente, da relação entre o custo de produzi-la e o custo de não produzi-la. Quais itens devem ou não ser produzidos está relacionado com o processo de desenvolvimento de software que se está adotando, com a análise de riscos do projeto e com os requisitos de qualidade do produto.

Muitos autores relacionam a falta de atualização da documentação com a ausência de um processo de Gerência de Configuração, uma vez que a documentação é considerado um dos produtos do processo de Engenharia de Software. Em seu trabalho sobre o controle de documentação em pequenas companhias, Pacheco e Sanches (2000) explicam que, durante o ciclo de vida do software, os documentos evoluem e são alterados, sendo criadas novas versões do item em questão. Para manter um melhor controle desta situação, é necessário que sejam estabelecidas normas para a criação e alteração dos documentos. Essas normas constituem a prática

da Gerência de Configuração de Software, que, inclusive, constitui uma das principais *key process areas* – KPA - do CMM.

Vários processos de desenvolvimento de software foram sugeridos ao longo do tempo, mas duas vertentes têm se sobressaído atualmente: o Processo Unificado, mais burocrático, acadêmico e abrangente em oposição aos Métodos Ágeis, mais dinâmico, enxuto e guiado por "boas práticas". Nos itens a seguir será detalhado um pouco desses dois métodos, enfatizando sua relação com a documentação produzida.

2.2.1 Processo Unificado

O Processo Unificado, também conhecido como RUP (*Rational Unified Process*), é o resultado da integração do trabalho de três autores, Ivar Jacobson, Grady Booch e James Rumbaugh na *Rational Software Corporation*. Segundo [Booch] , podemos sintetizar o RUP como um *framework* de projeto que descreve uma classe de processos que são iterativos e incrementais. Ele produz funcionalidades em pequenos incrementos, cada um construído baseado no anterior, sendo guiados por "casos de usos"ao invés da construção de subsistemas. No RUP as atividades de estimativa e de planejamento de tarefas são feitas com base nas iterações anteriores. As primeiras iterações do projeto são altamente focadas sobre uma arquitetura de software: uma implementação rápida dos recursos do produto são postergadas até que uma arquitetura robusta e confiável seja identificada e testada.

Em sua forma mais simples, RUP consiste de alguns fluxos fundamentais:

- Engenharia de negocio: entendendo as necessidades do negócio;

- Requisitos: traduzindo as necessidades do negócio no comportamento do sistema automatizado;

- Análise e projeto: traduzindo os requisitos em uma arquitetura de software;

- Construção: criação de software que se ajusta a arquitetura proposta e possui os comportamentos desejados;

- Teste: garantir que os comportamentos desejados são corretos que todos estão presentes;

- Configuração e gerenciamento de mudanças: mantendo uma referência de de todas as diferentes versões de todos os produtos.;

- Gerenciamento do projeto: atribuindo e mantendo o ambiente de desenvolvimento;

- Entrega: tudo que é necessário para a implantação do projeto;

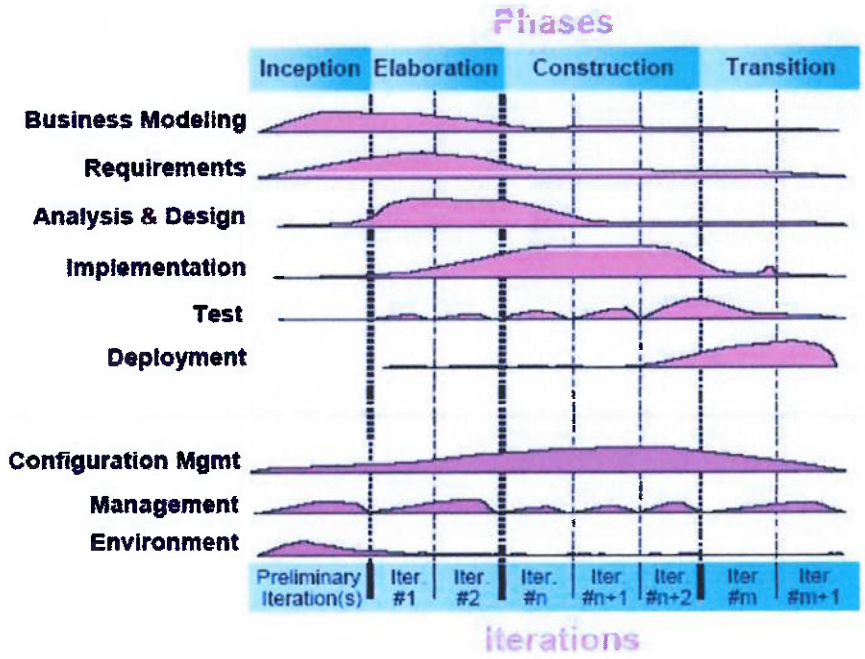

Figura 2.1 – Ciclo de Vida do RUP (fonte: JACOBSON et al., 1999)

Essas atividades não são separadas no tempo. Ao contrário, elas são executadas concorrentemente durante o ciclo de vida do projeto. Como se vê na Fig. 2.1, pouquíssimo código é escrito no início, mas sua quantidade não é zero. Posteriormente, muitos dos requisitos são conhecidos, mas alguns novos requisitos ainda serão identificados.

Assim, conforme o projeto evolui, a ênfase em certas atividades aumenta ou diminui, mas os diferentes tipos de atividades ainda podem ser executados a qualquer momento, durante o ciclo de vida do projeto.

O projeto evolui em incrementos, chamados iterações. O objetivo de cada iteração é desenvolver algum software funcional que possa ser demonstrado aos solicitantes, e que eles possam achá-lo funcional.

Existem quatro fases em um projeto RUP: iniciação, elaboração, construção e transição. Essas fases representam uma certa ênfase em algumas atividades dentro de uma iteração. A transição de uma fase à outra é reconhecida ao se completar determinados artefatos. Os artefatos produzidos em cada uma das fases estão listados a seguir.

Iniciação:

- Relação simples dos principais requisitos, possivelmente na forma de casos de uso;
- Quadro geral da arquitetura do software;
- Descrição dos objetivos do software;
- Planejamento do projeto preliminar;
- Plano de negócio para o projeto.

Elaboração:

- Corpo geral do software, na forma de um protótipo arquitetural;
- Casos de uso que descrevem a maior parte do comportamento do software;
- Plano de projeto detalhado, descrevendo as iterações seguintes;
- Principais testes para verificar a operação do software.

Construção:

- O sistema de software: composto por suas várias partes como código fonte, executável, componentes e suas interfaces, arquivos, bibliotecas e outros;
- Os testes do sistema incluindo planos de testes, procedimentos e casos de testes e resultados dos testes;
- O manual do usuário;
- Planos de integração das partes.

Transição:

- Basicamente os mesmos da construção, apenas são ajustados de acordo com as necessidades da instalação. Pode envolver o planejamento de novas atividades

como a realização de testes Beta e a migração de dados para o novo sistema.

2.2.2 Processos ágeis

Os processos ágeis indicam procedimentos e recomendações para ganhar tempo na produção de software. Isto muitas vezes vai de encontro à produção de documentação, por ser considerado que o tempo gasto na produção da documentação não foi usado para a produção do software.

Com um visão mais focada na modelagem do que na produção do código, Ambler (2001), relacionou alguns pontos críticos que devem nortear a produção da documentação, segundo essa perspectiva de agilidade:

- O problema fundamental é a comunicação, não a documentação;
- A documentação deve ser enxuta e objetiva;
- A documentação deve ser apenas boa o suficiente;
- Os modelos não são, necessariamente, documentos e vice-versa;
- A documentação faz tanto parte do sistema quanto o código fonte;
- O objetivo principal da equipe de desenvolvimento é desenvolver software; o segundo é capacitar o seu próximo esforço;
- O benefício de se ter a documentação deve ser maior que o custo de criá-la e mantê-la;
- Nunca confiar na documentação, sem ter garantias de que ela está atualizada;
- Cada sistema tem a sua própria necessidade de documentação, não existe "tamanho único";
- Questionar a necessidade da documentação, e não considerá-la como uma vontade;
- O investimento na documentação de um sistema é uma decisão de negócio, e não técnica;
- Criar a documentação apenas quando ela se fizer necessária;
- Atualizar a documentação somente quando quando houver uma necessidade explícita para isso.

A seguir apresentamos o processo ágil mais comentado - *Extreme Programming* - e o papel da documentação nesse processo.

2.2.2.1 Extreme Programing (XP)

O método XP é comumente atribuído a Kent Beck, Ron Jeffries, e Ward Cunningham. Ele é direcionado a equipes de tamanho pequeno a médio, construindo software com pouco conhecimento dos requisitos, ou que sejam alterados rapidamente. As equipes do XP estão normalmente situadas num mesmo local físico e se compõem de menos de 10 membros (PAULK, 2001).

O ciclo de vida na XP tem quatro atividades básicas: codificação, teste, captura e projeto. O dinamismo é demonstrado através dos seguintes valores:

- comunicação contínua com o cliente e o time;
- simplicidade, através do foco constante na solução minimalista;
- rápido *feedback* através dos testes de unidades e funcionais;
- coragem para lidar com os problemas pró-ativamente.

Muitos dos princípios da XP, como minimalismo, simplicidade, ciclo de vida evolutivo e envolvimento do usuário, são práticas de senso comum que fazem parte de qualquer processo disciplinado. A tabela 2.1 resume o "extremo" na XP, que vem de levar essas comentadas boas práticas para níveis extremos.

Sob o ponto de vista da documentação, a XP segue o princípio de minimalismo: praticamente toda a documentação é descartada depois de utilizada, pode ser feita a mão livre ou num quadro-branco e deve existir enquanto sirva ao seu propósito.

Os requisitos são capturados na forma de um conjunto de estórias[1], que podem ser acrescidas de novas estórias a qualquer momento. O segredo está em programar a entrega de partes do produto a cada duas semanas, facilitando o gerenciamento dessa volatilidade.

---

1 *user stories* – em Ambler, S. W. Modelagem Ágil: Práticas eficazes para a programação eXtrema e o Processo Unificado. Trad. Acauan Fernandes. Porto Alegre: Bookman, 2004, a tradução para o termo é histórias. Aqui se traduz por estórias, por se achar este termo mais correto.

Tabela 2.1 – Boas práticas levadas ao extremo na XP (transcrita de Paulk, 2001)

Senso comum	Extremo no XP	Prática de implementação
Revisão de código	Revisar o código todo o tempo	Pair programming
Teste	Testar todo o tempo, e pelo cliente	Teste de unidade e funcional
Projeto	Fazer o projeto parte deas atividades diárias de todos	Refactoring
Simplicidade	Sempre trabalhe com o projeto mais simples que proporcione a funcionalidade atual do sistema	A coisa mais simples que possa funcionar
Arquitetura	Todos trabalham para refinar a arquitetura o tempo todo	A Metáfora
Teste de Integração	Integre e teste várias vezes ao dia	Integração contínua
Iterações curtas	Torne as iterações curtas, em dias ou horas	Planning game

Os desenvolvedores do XP aceitam a idéia de ter uma visão geral do sistema, enfatizando o projeto, enquanto que ao mesmo tempo minimizam a necessidade de documentação do projeto. Por adotar o princípio de reprojeto[2] contínuo para melhoria do código, uma documentação detalhada facilmente torna-se obsoleta e o custo de produzir e manter uma documentação dessas, fora do código, torna-se proibitivo. Os responsáveis pela manutenção geralmente só confiam no próprio código fonte.

Em Jeffries (2001), é resumido de forma clara que, para a XP, a documentação deve ser produzida quando ela é importante. "Externamente do seu projeto XP, você provavelmente irá necessitar de documentação escrita: de qualquer forma, escreva-a. Internamente no entanto, existe tanta comunicação verbal que você precisará de muito pouca documentação. Verifique se você sabe a diferença."

Em van Deursen (2001), são apresentados dois aspectos que tornam XP um processo interessante sob o ponto de vista da compreensão de um sistema:

- o código fonte tem um papel dominante em todos os passos do XP: o código é documentado através dos testes, os testes são baseados em programas e não em dados, o código é melhorado continuamente para manter o projeto simples e esse *refactoring* substitui uma fase de projeto explícita;

2 *redesign*

- a comunicação do time é explicitada no XP, através da codificação por pares, onde duas pessoas, trabalhando juntas, discutem a melhor forma de implementar um recurso. O planejamento é feito em conjunto com toda a equipe de desenvolvimento, discutindo o impacto de cada um dos recursos a serem implementados.
- Como se vê, a ênfase que XP coloca no código fonte e nas pessoas, sugere que a necessidade das pessoas em compreender os trechos de código é o coração da manutenção no XP.

O principal artefato produzido, porém geralmente descartado são as estórias. Elas são o ponto de partida no desenvolvimento XP. É através da escolha delas que o cliente dá início ao processo iterativo. São definidas como uma "coisa" que o cliente quer que o sistema faça, têm uma estimativa entre uma e cinco semanas de programação ideal e devem ser testáveis.

Assumindo que a documentação do sistema produzida pelo XP é o código, uma forma de enriquecer essa documentação, mantendo a visão do XP, é agregando as estórias (requisitos do sistema) ao produto final, o código. O que não está claro é a forma de como fazer isso.

2.2.3 Relação entre os artefatos XP e RUP

Na tabela 2.2 relacionam-se os principais artefatos produzidos pelo XP e pelo RUP, ajustados para pequenos projetos, segundo apresentado em Booch et al. (1998).

Tabela 2.2 – Relação entre os artefatos do XP e do RUP (transcrita de Booch et al., 1998)

XP	**RUP para pequenos projetos**
Estórias Documentos adicionais das conversações	Visão Glossário Modelo de Casos de Uso
Ressalvas, restrições	Especificações complementares
Testes de aceitação Testes de unidades Dados para testes Resultados de testes	Modelo de testes

XP	RUP para pequenos projetos
Código fonte	Modelo de implementação
Releases	Produtos Notas dos *releases*
Metáfora	Documento de arquitetura do software
Projeto: CRC, esboços em UML Tarefas Documentos de projetos, produzidos ao final do projeto Documentação de suporte	Modelo de Projeto
Padrões de codificação	Referências de projeto Referências de programação
Área de trabalho (desenvolvimento e outros) Ferramentas para testes	Ferramentas
Plano de *releases* Estimativas das estórias Estimativas das tarefas Plano de iterações	Plano de desenvolvimento de software Plano de iterações
Plano geral – custos/orçamento	Plano de negócios Lista de riscos Plano de aceitação de produto
Relatórios de progresso Registros de horas de trabalho em tarefas Dados de métricas: recursos, escopo, qualidade, tempo Outras métricas Rastreamento de resultados Relatórios e notas de reuniões	Avaliação de estado
Defeitos e dados associados	Requisições de mudanças
Ferramentas de gerenciamento de código	Plano de gerenciamento de configuração Repositório de projeto Ambiente de trabalho
Spike (ponta)	Protótipos
	Plano de desenvolvimento *Template* para projetos específicos

2.3 Principais ítens que constituem uma boa documentação

Ao se discutir documentação de projeto, algumas questões emergem:

- "Qual documentação de um projeto realmente deve ser criada"?

- "Existe uma real utilidade para a documentação que justifique o esforço e o custo de produzi-la?"

Tenta-se aqui expor alguns motivos que respondam positivamente essas questões, desde que se leve em consideração uma série de limites.

Em Ambler (2001), são citados alguns dos motivos válidos para se criar documentação:

- Os participantes do projeto a estão requisitando;
- Ela é usada para definir um modelo de contratação;
- Ela é usada para apoiar a comunicação com grupos externos;
- Ela pode ser empregada como ajuda para raciocinar sobre alguma coisa.

Outros, não tão válidos, são:

- O solicitante quer parecer que está no controle;
- O solicitante quer justificar sua existência;
- O solicitante não conhece coisa melhor;
- O seu processo exige a criação da documentação;
- Alguém quer ter certeza de que tudo está indo bem;
- Está-se especificando um trabalho para outro grupo.

Assim como Smith (1999), acredita-se na documentação como fator preponderante para melhorar a capacidade de manutenção de um software. Ele cita que a documentação que é estruturada e contida no programa é capaz de satisfazer imediatamente a demanda de alterações de informações para o responsável pela tarefa de modificação de um software. Exemplos dessas necessidades são:

- para solucionar erros não-triviais ou problemas de modificação, o mantenedor deve possuir uma compreensão detalhada do programa;
- para localizar uma parte do código, é necessário conhecimento da estrutura do programa;
- saber como uma seqüência de instruções se relaciona com outras partes do programa é importante para a alteração e teste de software.

A informação pode ser apresentada para a equipe de manutenção do software de diversas formas. Ao colocar-se a documentação no próprio fonte, três formas são possíveis. Uma é um resumo de utilização no início de cada rotina. Outra é através de cabeçalhos de seções de procedimentos posicionados na seqüência (bloco) de instruções. A terceira é através de sentenças curtas colocadas ao lodo do código. A complexidade e o tamanho do módulo vão determinar qual tipo de informação será usada.

Como o objetivo básico deste trabalho é mostrar a importância da documentação, principalmente na fase de manutenção, cita-se a seguir quatro assuntos que necessariamente a documentação do sistema deve, como um todo, cobrir. Ter esses itens documentados é um dos principais motivos de produzir a documentação com qualidade, e determina um mínimo ideal de documentação que se deseja atingir.

1) Verificação e validação dos requisitos

Segundo Parnas e Madey (1995), a validação de um projeto é uma tarefa técnica que pode ser conduzida se a documentação do projeto for suficientemente precisa para permitir uma análise sistemática. Muitos dos documentos produzidos antes da implementação não são técnicos, mas narrativas explicando o papel do sistema, cenários descrevendo como eles podem ser usados ou descrição das qualidades que ele deveria ter. Pouca análise técnica pode ser feita com base nesse tipo de documentos. Outros documentos fornecem análises precisas de algoritmos abstratos, mas ignoram muito do código atual. Uma análise real de todo o projeto deve esperar até que a implementação esteja praticamente completa. Neste ponto, correções são muito mais difíceis - e caras - do que aquelas que poderiam ter sido feitas anteriormente.

2) Rastreabilidade dos requisitos

Tanto na fase de modelagem quanto na de implementação, deve-se ter documentados, e de alguma forma relacionados, os requisitos que motivaram a criação dos itens sendo criados. Este conhecimento permite:

- relacionar todos os itens afetados por uma modificação nos requisitos;
- os requisitos dependentes e como são afetados, quando for necessário alterar um item.

Modelos podem facilmente capturar essas informações, principalmente requisitos funcionais, mas no código, geralmente, essa referência é mais difícil. Alguma forma de anotação deve ser fornecida nesse caso.

3) Registro das decisões de projeto

Segundo Arkley et al. (2002), ao analisarmos as causas da falta de registro das decisões de projeto, aparecem três problemas principais:

- programadores sobrecarregados não são muito compreensivos, quando se pede para registrar a razão de toda decisão tomada;
- não é suficiente documentar a decisão de projeto, mas ela também deve estar acessível, quando alguém realmente precisar saber porque uma decisão foi feita;
- uma justificação desatualizada é potencialmente mais perigosa do que nenhuma justificativa. As alterações no sistema devem estar refletidas na documentação.

4) Especificação da arquitetura

É extremamente importante descrever de alguma forma como o sistema lida ou irá lidar com os problemas relacionados com os requisitos não funcionais, como desempenho e escalabilidade. A forma de modularização do sistema, o agrupamento das funcionalidades em componentes, a distribuição destes componentes entre os diversos equipamentos de um ambiente distribuído e os mecanismos de ativação e troca de informação entre os componentes é a maneira de representar a arquitetura de um sistema, juntamente com o registro das decisões de projeto que levaram a escolher tal estrutura de componentes. Esta documentação deve guiar a implementação e os testes, de forma a comprovar que a arquitetura escolhida satisfaz os requisitos não funcionais previstos.

2.4 Documentação e qualidade do software

A produção de uma documentação influencia na qualidade do software produzido, principalmente daqueles fatores de qualidade ligados a compreensão (vide tabela 2.5). Por sua vez, essa capacidade de compreensão influencia diretamente no custo de reuso, ou de manutenção, de um software.

No modelo para cálculo de estimativas COCOMO-II, apresentado em Boehm (2000), são introduzidos conceitos relacionados ao efeito de um software ser bem ou

mal estruturado e compreensível no cálculo do custo/esforço de manutenção. Estes conceitos estão representados por três fatores usados no cálculo da quantidade de linhas de código equivalentes a se fosse produzir o software do início (fig 2.2). Esses fatores são:

- Avaliação e Assimilação indica quanto tempo e esforço estará envolvido no teste, avaliação e documentação de telas e outras partes do programa para saber o que pode ser reutilizado;
- Compreensão de Software (SU) estima a dificuldade em entender o código que se está alterando, e como a estruturação do código, sua correlação com a aplicação e seus comentários podem auxiliar na compreensão.
- Desconhecimento do Programador (UNFM) indica quanto sua equipe já está familiarizada com o código em questão, se é a primeira vez que estão vendo ou ele já é muito familiar.

Reuse-Model

$$Equivalent\_KSLOC = Adapted\_KSLOC \cdot \left(1 - \frac{AT}{100}\right) \cdot AAM$$

$$AAF = (0.4 * DM) + (0.3 * CM) + (0.3 * IM)$$

$$AAM = \begin{cases} \dfrac{AA + AAF \cdot (1 + [0.02 \cdot SU \cdot UNFM])}{100}, & AAF \leq 50 \\ \dfrac{AA + AAF + (SU \cdot UNFM)}{100}, & AAF > 50 \end{cases}$$

DM = percentage design modified
CM = percentage code modified
IM = percentage of integration effort
AAF = amount of modification
AAM = adaptation adjustment modifier
AA = Assessment and Assimilation
AT = percentage of adapted KSLOC that is re-engineered by automatic translation
SU = software understandability
UNFM = programmers realative unfamiliarity

Figura 2.2 – Modelo de estimativa para reuso de software do COCOMOII (retirado de Boehm, 2000)

Para a determinação do valor para SU consulta-se a Tabela 2.3 para cada um dos três fatores que determinam o grau de compreensão – estrutura, clareza da aplicação e auto-descrição – e pondera-se cada um deles e se obtém a média.

Tabela 2.3 – Influência de alguns fatores no custo/esforço de manutenção (SU)

Peso	Estrutura	Clareza da Aplicação	Auto-descrição
Muito baixo =0,5	Muito baixa coesão, alto acoplamento, código "espaguete"	As visões do mundo da aplicação e do programa não combinam	Código obscuro, documentação falha, obscura ou desatualizada
Baixo =0,4	Baixa coesão (moderada), alto acoplamento	Pouca correlação entre o programa e a aplicação	Poucos comentários e cabeçalhos de código e pouca documentação útil
Normal =0,3	Estrutura bem razoável, algumas áreas falhas	Razoável correlação entre o programa e a aplicação	Razoável nível de comentários; cabeçalhos de código e de documentação útil
Alto =0,2	Alta coesão, baixo acoplamento	Boa correlação entre o programa e a aplicação	Bons comentários e cabeçalhos de código e grande parte da documentação útil; algumas áreas falhas.
Muito Alto =0,1	Grande modularidade, informação escondida em estrutura de dados/controle	As visões do mundo da aplicação e do programa se encaixam claramente	Código alto-explicativo, documentação atualizada, bem organizado, com as decisões de projeto.

Para a determinação do valor do UNFM, basta classificar o código pelo o grau de conhecimento e obter o valor correspondente da tabela 2.4.

Tabela 2.4 – Influência de alguns fatores no custo/esforço de manutenção (UNFM)

UNFM	Nível de desconhecimento
0.0	Completamente conhecido
0.2	Muito conhecido
0.4	Razoavelmente conhecido
0.6	Razoavelmente desconhecido
0.8	Muito desconhecido
1.0	Completamente desconhecido

A Tabela 2.5 apresenta os principais fatores de qualidade ligados à documentação, segundo nossa avaliação do grau de importância desta para a obtenção desse fator, tomando como base a relação de fatores de qualidade apresentada em Cysneiros et. Al(2001), e características e necessidade de desenvolver o documento descrito em Ambler (2001).

Essa classificação ajuda no momento de ponderar se é vantajoso ou não a produção de determinada documentação em relação ao esforço necessário para produzi-lá, levando em consideração os fatores de qualidade desejados no projeto.

Tabela 2.5- Principais fatores de qualidade do software que estão diretamente ligados à documentação

Fator Principal de Qualidade	**Fatores Secundários de Qualidade**	**Característica da documentação**	**Grau de importância da documentação**
Capacidade de manutenção	Facilidade para encontrar onde está o erro	Compreensão, rastreabilidade	Muito alto
Capacidade de manutenção	Facilidade para modificar	Rastreabilidade, descrição das interfaces	Muito alto
Capacidade de manutenção	Estabilidade, capacidade de suportar mudanças	Descrição das decisões de projeto e arquitetura	Alta
Capacidade de manutenção	Facilidade para testar	Descrição dos casos de teste, rastreabilidade, validação e verificação	Média
Clareza	Informação compreensível	compreensão	Média
Custo	Redução do custo de manutenção	Todos os fatores citados	Muito alta
Portabilidade	Adaptação, instalação em diferentes plataformas e ambientes	Descrição das decisões de projeto e arquitetura	Média
Usabilidade	Facilidade de uso e de aprender	Compreensão, atualização, sem ambiguidade	Alta
Rastreabilidade	Caminho que algo percorre ou qual o estado de algo	Rastreabilidade, atualização, sincronismo	Média

2.5 Principais artefatos que documentam um projeto de software

A seguir é apresentada uma relação dos principais documentos gerados durante o processo de desenvolvimento de software, com uma breve descrição de sua utilização, em duas visões principais.

2.5.1 As várias "visões" da documentação

Podemos considerar como "visões", um aspecto que é enfocado para se classificar a documentação. Observaremos principalmente duas visões:

- Por fase de projeto: a documentação cuja produção é enfatizada em cada fase

(tabela 2.6).

- Por tipo de usuário: a documentação que é mais adequada a cada usuário (tabela 2.7).

Tabela 2.6 - principais documentos por fase do projeto

Fase do ciclo de vida do projeto	**Documento**	**Descrição**
Análise	Especificação de requisitos	Descreve o que o sistema fará e é composto de um ou mais artefatos como casos de uso e regras de negócio.
	Visão geral executiva / Contrato	Visão geral dos objetivos do sistema e um resumo dos custos estimados, benefícios esperados, riscos, recursos necessários e prazos.
	Plano de projeto	Engloba a visão executiva e as ferramentas, tecnologias e processos empregados, além dos principais artefatos produzidos e os pontos de controle do cronograma.
Projeto	Modelo de contrato	Descreve a interface de um sistema ou porção do sistema
	Decisões de Projeto	Um resumo das decisões críticas pertencentes ao projeto e a arquitetura que a equipe fez durante o desenvolvimento. Visão de alto nível
Construção	Manual do sistema	Visão geral dos sistema, detalhes de arquitetura, de projeto e uma API dos módulos e funções implementadas.
	Manual do usuário	Constitui normalmente de 4 documentos básicos: Um guia de referência, um tutorial, um guia de suporte e material para treinamento
Instalação/ Transição	Manual de operação	Inclui uma indicação das dependências em que o sistema está envolvido, a relação com outros sistemas, banco de dados, arquivos; procedimentos de backup e limpeza de históricos; lista de contatos; requisitos de disponibilidade; estimativas de carga e guias de resolução dos principais problemas.
	Manual de Suporte	Material de treinamento, a documentação do usuário como material de referência para resolução de problemas; lista de contatos da equipe de manutenção; procedimentos para atender problemas mais complexos.

Tabela 2.7 - Principais documentos por tipo de usuário

Usuário	Documento
Usuário final	Manual do Usuário
Desenvolvedor	Manual do sistema
Instalador	Manual de Instalação
Treinador	Manual de Treinamento
Requisitante	Contrato, Especificação de Requisitos
Gerente de Projeto	Plano de Projeto
Operador	Manual de Operação
Suporte	Manual de Suporte
Comercial	Material de Divulgação

2.5.2 Taxonomia da documentação

Entendemos que as técnicas de reutilização aplicadas ao software também devem ser aplicadas aos documentos. Sempre que possível um documento deve ser reaproveitado, como copiar sua estrutura básica de um projeto para outro. Também levando a noção de herança de classes para a documentação, ou seja, quando uma classe herda métodos de uma outra, também deve herdar a sua documentação.

Outra noção de reuso é dentro do mesmo projeto, quando trechos de um documento servem em outro. Normalmente isso ocorre quando um documento maior é formado por outros documentos básicos, que podem aparecer em vários documentos. A tabela 2.8 listamos alguns ítens básicos que compõem documentos maiores. Esta relação, elaborada a partir de uma tabela de possíveis documentos encontrada em Scott Ambler (2001), pode servir de guia a fim de tornar mais clara a reutilização de trechos de documentos, evitar retrabalho e proporcionar uma forma de automatizar a geração de trechos da documentação.

Tabela 2.8 - Ítens básico de uma documentação de sistema

Ítens de documentação	Macro-documentos
Visão geral do sistema	Plano de Projeto, Manual do sistema, Visão geral executiva, Especificação de requisitos, Manual do Usuário, Manual de Instalação

Ítens de documentação	Macro-documentos
Lista de contatos do cliente	Plano de Projeto, Manual de Instalação,Manual de Operação
Lista de desenvolvedores por caso de uso/ módulo	Plano de Projeto, Manual de Instalação, Manual de Operação, Manual de Suporte
Tecnologia e ferramentas utilizadas	Plano de Projeto, Manual de Instalação
Descrição do processo	Plano de Projeto
Repositório do projeto	Plano de Projeto
Cronograma (custo/prazo)	Plano de Projeto, Visão geral executiva
Requisitos funcionais/ casos de uso	Especificação de requisitos
Requisitos não funcionais	Especificação de requisitos
Requisitos de alto-nível/ objetivos do negócio	Especificação de requisitos, Manual do sistema, Visão geral executiva
Modelos de contrato/ descrições de interfaces	Manual do sistema
Procedimentos operacionais (backup, limpeza de histórico)	Manual de Treinamento, Manual de Operação
Requisitos de disponibilidade	Especificação de requisitos, Manual de Operação
Carga projetada	Especificação de requisitos, Manual de Operação
FAQ e Troubleshooting	Manual do Usuário, Manual de Treinamento, Manual de Operação, Manual de Suporte
Guia de referência	Manual do Usuário
Tutorial	Manual do Usuário, Manual de Treinamento
API de funções	Manual do sistema
Modificações por versão	Manual do Usuário, Manual de Treinamento, Manual de Operação, Manual de Suporte, Material de Divulgação
Guia de treinamento	Manual de Treinamento, Manual de Suporte
Padrões da Interface Homem-Computador	Manual do sistema
Requisitos para instalação (HW e SW)	Manual de Instalação, Manual de Operação, Manual de Suporte, Material de Divulgação
Alternativa/ disponibilidade de módulos	Manual do sistema, Manual de Instalação, Manual de Operação, Material de Divulgação
Instruções de instalação	Manual de Instalação, Manual de Treinamento, Manual de Suporte
Registro de licenças	Manual de Instalação
Instruções de configuração	Manual de Instalação, Manual de Treinamento, Manual de Operação, Manual de Suporte
Decisões de projeto	Manual do sistema, Plano de Projeto
Modelo de arquitetura	Manual do sistema, Plano de Projeto
Glossário	Manual do sistema, Manual do Usuário, Manual de Treinamento, Manual de Suporte

2.6 Fatores de qualidade dos documentos de softwares

Ao se produzir um documento de software, deve-se sempre ter em mente alguns fatores de qualidade que devem ser seguidos e que garantem a adequação do documento aos seus propósitos. Os principais fatores listados a seguir aparecem em grande parte da literatura ligados aos requisitos (ERS), porém pode-se aplicá-los em outros documentos sempre que possível:

- não conter ambigüidade, quando só existe uma possível interpretação;
- ser compreensível, ou possuir uma linguagem clara e ter um glossário para referência dos termos do domínio do negócio;
- ser verificável;
- ser consistente, eliminando conflitos internos e entre documentos;
- ser concisa, com construções objetivas;
- ser completa, incluindo tudo que o software deve fazer, tratadas todas as possíveis entradas de dados e todas as seções do documento foram elaboradas;
- ser rastreável, permitindo referência cruzada, ou seja, cada item deve ser identificado e também a sua origem e quem o utiliza.

2.6.1 Problemas relacionados com o processo de documentação

Em seu ensaio sobre *Agile Modeling,* Scott Ambler (2001) relaciona uma série de problemas relacionados com a criação de documentação, apresentando algumas dicas do ponto de vista da *Agile Modeling*, que basicamente consistem de um balanceamento da quantidade certa de documentação, elaborada no momento certo e para a audiência correta.

Também em Parnas e Clements (sem data) são relacionados problemas organizacionais mais difíceis de resolver do que apenas uma documentação incompleta e imprecisa; se este somente fosse o problema, bastaria completá-la ou corrigi-la.

Levantam-se e relacionam-se, a seguir, alguns desses pontos apontados por ambos os autores sobre o motivo de por que uma documentação não é bem elaborada:

- o tempo gasto no desenvolvimento da documentação deixa de ser usado no desenvolvimento do software;
- a falta de habilidade do desenvolvedor para escrever e expressar seus conhecimentos;
- durante o desenvolvimento tenta-se entender o que é para ser feito, sendo diferente da fase posterior ao desenvolvimento, onde se tenta entender porque foi feito daquela forma e como opera;
- deve-se escrever a documentação em conjunto com o desenvolvimento, ou no final dele. Simultaneamente, seria o ideal, mas num processo altamente incremental é grande a chance de que o que se escreve num dia tenha de ser reescrito no dia seguinte. Ao deixar para o final, corre-se o risco de não haver mais tempo, esquecer-se de algumas das razões ou não haver mais sentido para isso;
- problema da miopia - quando a documentação é produzida no final pelo desenvolvedor que conviveu com todo o processo, este pode documentar pequenos detalhes em detrimento de considerações mais gerais, resultando numa documentação útil para quem conhece o sistema, mas incompreensível para os novatos;
- manter a documentação interna ao código ou externa está relacionado principalmente a quem se destina. Além de desenvolvedores, também os usuários, gerentes, vendedores e pessoal de operação, que não têm acesso ao código, necessitam de documentação. Nestes casos, documentos externos ao código são imprescindíveis;
- manter a documentação no nível da empresa e não só do projeto, para aqueles ítens como definições de regras de negócio, interfaces para sistemas legados e os meta-dados corporativos. Isso facilita a reutilização da documentação;
- quantidade versus qualidade, depende basicamente da confiança que se pode ter, se a informação contida nos documentos está correta. É bom lembrar que é mais fácil manter atualizada, ou até mesmo refazer uma documentação mais enxuta, de um ponto de vista mais geral, do que uma que entra mais nos detalhes, e por isso mesmo mais fácil de estar inconsistente;
- terminologia confusa e inconsistente, falhas em produzir definições de novas

terminologias precisas implica na existência de muitos termos usados para os mesmos conceitos e conceitos distintos descritos pelo mesmo termo.

- textos longos, várias palavras usadas para descrever algo que poderia ser descrito através de um diagrama ou uma fórmula, perda de objetividade e concisão são fatos que são repetidos em diversas seções.

2.7 Conclusões

Neste capítulo, apresentou-se a necessidade de produzir uma boa documentação, sua relação com os processos de desenvolvimento que delineiam a quantidade ideal a ser produzida, a relação entre qualidade, quantidade e custo, principais ítens e sub-ítens e sua reutilização.

Apresentou-se as quatro funções básicas da documentação, do ponto de vista de auxiliar a comunicação entre os membros, não só durante o desenvolvimento, mas também posteriormente durante a manutenção: servir de modelo de contrato e permitir a validação e verificação do produto; localizar cada funcionalidade desejada no elemento que a implementa; registrar as decisões tomadas pelos projetistas e desenvolvedores durante o processo; permitir uma visualização geral da organização dos componentes que implementam o produto. É uma ênfase na compreensão do produto.

Entretanto, a maior contribuição se relaciona em expor os principais problemas relacionados à documentação de sistemas. Falta de definição e de automatização do processo de documentação, sua integração no processo de desenvolvimento, ferramentas de integração para reaproveitamento da documentação para gerar código e outra documentação, direcionar a documentação para a sua audiência específica, foram alguns dos ítens abordados.

No capítulo seguinte serão apresentadas ferramentas e técnicas existentes que podem ser utilizadas para ajudar a minimizar os problemas apresentados.

3 TÉCNICAS E FERRAMENTAS PARA A CRIAÇÃO E MANUTENÇÃO DA DOCUMENTAÇÃO

O código é a fonte! - A documentação registrada no próprio código. Essa idéia é defendida por muitos pesquisadores, de formas variadas, como será visto nos ítens que se seguem. Uma vasta gama de desenvolvedores aparece neste grupo, desde aqueles que não escrevem nenhuma documentação - nem mesmo comentários - mas mantém uma codificação clara, padronizada de nomeação de variáveis e funções, o que torna o código "auto-explicável", até os que optam por não escrever o código propriamente dito, mas gerá-lo através dos modelos (código sem documentação *versus* documentação sem código).

Em seu trabalho, Kotula (2000) apresenta que a documentação no código fonte é uma prática de engenharia crítica para um desenvolvimento eficiente de software. Independente da intenção do desenvolvedor, todo código fonte eventualmente pode ser reutilizado, seja diretamente ou não (apenas para ser compreendido). Seja como for, ele atua como a especificação do comportamento para outros desenvolvedores. Sem uma documentação, eles são forçados a obter a informação de que precisam assumindo alguns fatos de forma perigosa, detalhando a implementação ou interrogando o desenvolvedor. Nenhuma dessas alternativas são aceitáveis.

Apesar de alguns desenvolvedores acreditarem que o código fonte pode ser "auto-documentado", existe uma grande parte da informação sobre o comportamento do código que não pode ser expressa na forma de código, mas que requer o poder e a flexibilidade de uma linguagem natural para isso. Assim sendo, a documentação do código fonte é uma necessidade, assim como uma disciplina importante para aumentar a eficiência e qualidade do desenvolvimento.

Apresentam-se neste capítulo várias técnicas relacionadas ao desenvolvimento de documentação retiradas da literatura e, quando possível, exemplos de ferramentas *open-source* para o ambiente Java que as implementam ou suportam.

3.1 Gerenciadores de documentação

São ferramentas específicas para tratar da escrita e manutenção da documentação. Geralmente mantém um repositório para facilitar o rastreamento e a

referência cruzada entre os elementos que compõe os vários documentos relacionados.

3.1.1 REM

Requirements Manager - REM - é apresentado em Duránet al. (2000) como parte de sua tese de PhD. A ferramenta REM destina-se a auxiliar na determinação e registro de requisitos. Basicamente este sistema mantém em seu repositório uma representação em XML de seus modelos de documentação e utiliza ferramentas XLST para produzir a transformação do XML em documentos, que podem ser personalizados através de *templates*. Como apresentado, REM trabalha com três documentos em um projeto:

- documento de requisitos orientado ao cliente, expresso em linguagem natural;

- documento de requisitos orientado ao desenvolvedor, contendo modelos e outras informações técnicas, basicamente modelado através de um sub-conjunto da notação UML

- e um registro para conflitos e suporte a negociações, usando alguns padrões de linguagem e *templates* pré-definidos.

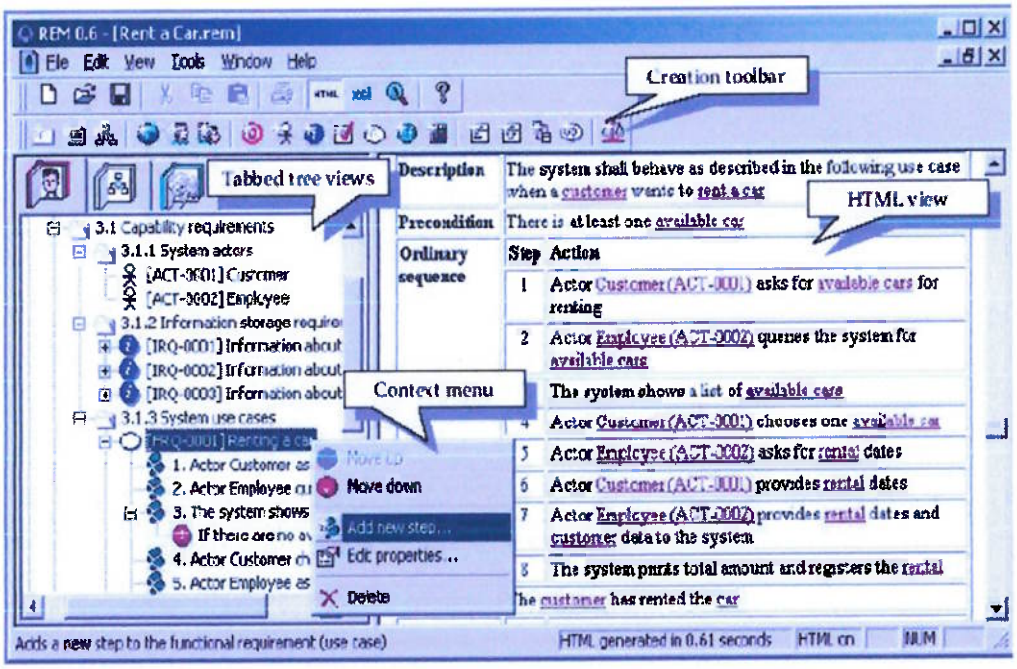

Figura 3.1 - Tela da ferramenta REM mostrando alguns recursos (fonte: Duran et al.,2000)

A figura 3.1 apresenta uma tela do sistema REM. Nela são apresentadas duas vistas do sistema. À esquerda, apresenta-se uma visão dos requisitos de um sistema exemplo, na forma de índice estrutural. À direita apresenta-se a visão da documentação produzida em HTML, segundo o *template* utilizado.

Através da utilização de transformações XSLT e sendo os requisitos armazenados eletronicamente em formato XML de acordo com a especificação DTD do REM, é possível que alguns fatores de qualidade exigidos numa boa documentação possam ser automaticamente verificados, como não conter ambiguidade, ser compreensível, ser verificável, ser consistente, conciso e permitir referência cruzada.

A Figura 3.2 mostra a arquitetura do REM, conforme descrita por Duran et al. (2000). Nesta figura, à direita apresenta-se os documentos HTML gerados a prtir da linguagem XML. À esquerda estão os repositórios de documentos REM. Na parte inferior da figura estão as planilhas de estilos de documentos e as especificações DTD (*Document Type Definitions*).

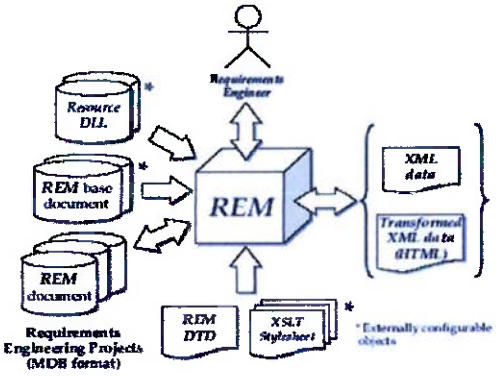

Figura 3.2 - Arquitetura REM

3.2 Geração automática da documentação.

As ferramentas desta classe tem em comum a proximidade física da documentação com o código, sendo a documentação registrada no próprio código fonte, ou através de anotações especiais (comentários em Javadoc), ou inversamente,

anotando a documentação com trechos de código (*Literate Programing*).

Segundo Ducasse e Nierstraz (2000), o principal benefício desta técnica, é a redução do esforço necessário para manter o código e a documentação sincronizados. A técnica proposta foi nomeada *Tie Code and Questions*, uma espécie de *design pattern* para reengenharia, cujo objetivo é relacionar as dúvidas e informações sobre um código (que está passando por um processo de reengenharia) no próprio código, para facilitar o rastreamento e manter sincronizado com o código a que se refere.

Os principais benefícios desta abordagem comparado a ter a documentação em um arquivo separado, que não se aplicam só a reengenharia, mas também às várias fases do desenvolvimento, são:

- Reduzir a descrição do contexto, ou seja, a necessidade de descrever o trecho de código de que se está tratando dentro do documento, ou de ter de repetir trechos desse código;
- Não existe a preocupação de manter dois documentos separados sincronizados, o que muitas vezes não é feito imediatamente e que pode causar perda de informações. Ressalta-se que a medida que passa o tempo torna-se cada vez mais difícil de, posteriormente, alocar tempo para essa sincronização;
- Usufruir do benefício da proximidade: sempre que estiver examinando um código e surgir uma dúvida, você tem a resposta a poucas linhas de distância; isto é muito mais efetivo do que ter de procurar um outro documento para esclarecimentos;
- Melhorar a comunicação da equipe de desenvolvedores, evitando problemas de acesso a documentos desatualizados ou de versões diferentes, pois estando no arquivo fonte, você sempre terá certeza de que possui a versão da documentação correspondente ao código.

A seguir examinam-se as principais pesquisas nesta categoria de ferramentas.

3.2.1 Literate Programming (LP)

Em sua forma tradicional, um programa de computador consiste de um arquivo de texto contendo o código do programa. Espalhados pelo programa estão comentários, que descrevem as várias partes do código. Quando Donald Knuth propôs essa filosofia de documentação LP, em 1992 (RAMSEY, 1994), justificando seu nome

disse:

"I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature. Hence, my title: 'Literate Programming' ".

Em LP, a ênfase é o inverso da documentação tradicional. Ao invés de escrever código contendo descrições, o programador escreve documentação contendo código. Ao invés de comentários delimitados por marcadores e restringidos ao início dos arquivos e dos módulos, eles passam a ser o foco principal. O programa é direcionado para a leitura por um ser humano, com o código sendo incorporado, através de delimitadores especiais, como se ele fosse o comentário do texto.

Dentro do paradigma da LP, o mesmo texto fonte pode ser processado por dois programas diferentes para se obter duas saídas distintas. O fonte pode ser "emaranhado" para produzir o código fonte que será traduzido por um compilador, ou pode ser "combinado" para produzir a documentação que contém uma extensa referência aos elementos do programa.

Devido a alguns problemas implícitos na técnica, cujos detalhes estão no item 3.2.3, e de estar relacionada com linguagens estruturadas e praticamente não suportar linguagens orientadas a objetos, ela não tem sido amplamente utilizada, apesar de possuir alguns adeptos e, conceitualmente, se encaixar perfeitamente na visão de um único arquivo contendo documentação e código com a finalidade de mantê-los em sincronismo.

A seguir apresenta-se a descrição de uma ferramenta para o desenvolvimento de aplicações usando LP, bem como exemplo de um programa com ela documentado, conforme apresentado por Ramsey (1994). A ferramenta NOWEB compila o código fornecido – composto por código e documentação – separando o conteúdo em dois arquivos distintos: um de código fonte na linguagem escolhida (C, por exemplo) e outro em formato HTML ou Tex (do Látex, de Knuth), que pode ser utilizado para gerar a documentação. Segundo Ramsey (1994) a ferramenta pode ser adaptada a qualquer linguagem através de uma configuração conveniente.

A Fig. 3.3 mostra um exemplo, o programa "Hello World" escrito em NOWEB. Na parte superior da figura (a) apresenta-se o texto original, onde se mescla texto e código em linguagem C. Abaixo de (a) apresenta-se o código do programa em C (b).

Na parte inferior da figura (c) apresenta-se a saída documentada.

3.1.1.1 NoWEB

A ferramenta NoWEB foi criada com o objetivo de levar a LP à sua essência, visto que a ferramenta anteriormente desenvolvida, WEB, era mais complicada de configurar, específica para uma linguagem de programação (Pascal), possuía cerca de 27 estruturas de controle e produzia uma saída num padrão de processador exclusivo (LaTex).

O autor dessa ferramenta descreve em Ramsey (1994) que a simplicidade do NoWEB está na utilização de um modelo simples de arquivos, que também são marcados com uma sintaxe simples, como pode ser visto no exemplo da figura 3.3.

Nele podemos ver que a separação dos trechos se dá pelo uso do @ para indicar um comentário, ou um nome entre << e >>= de um trecho de código , no começo da linha. Qualquer linguagem de programação pode ser usada, e a saída pode ser processada em texto simples, LaTex e HTML.

```
The first program one writes in a new language
is invariably the ''Hello World'' program.
Here it is in C.
<<greet>>=
<<includes>>
<<definitions>>
<<main program>>
@ The [[<<definitions>>]] chunk hard-codes the text.
<<definitions>>=
const char *GREETING = "Hello World";
@ The [[<<main program>>]] could be changed to read the
greetee's name from a command line argument.
<<main program>>=
int main(int argc, char **argv) {
    printf("%s", GREETING);
}
@ Only the standard IO library is needed.
<<includes>>=
#include <stdio.h>
@ That's all there is to it.
```

(a) noweb input

```
#include <stdio.h>
const char *GREETING = "Hello World";
int main(int argc, char **argv) {
    printf("%s", GREETING);
}
```

(b) output of tangle

The first program one writes in a new language is invariably the "Hello World" program. Here it is in C.

⟨*greet*⟩≡
  ⟨*includes*⟩
  ⟨*definitions*⟩
  ⟨*main program*⟩

The ⟨*definitions*⟩ chunk hard-codes the text.

⟨*definitions*⟩≡
```
const char *GREETING = "Hello World";
```

The ⟨*main program*⟩ could be changed to read the greetee's name from a command line argument.

⟨*main program*⟩≡
```
int main(int argc, char **argv) {
    printf("%s", GREETING);
}
```

Only the standard IO library is needed.

⟨*includes*⟩≡
```
#include <stdio.h>
```

That's all there is to it.

(c) output of weave

Figura 3.3 Exemplo de "Hello World" escrito em NoWEB

3.1.2 Elucidative Programming

Normark e Vestdam (2001) definem um paradigma de documentação (*Elucidative Programming*) criado como uma renovação das idéias da LP, descrita no item anteior, e que foi introduzida como uma maneira de apresentar os programas para

seres humanos, ao contrário de instruir uma máquina, através de uma descrição do programa intercalada com código fonte.

Na *Elucidative Programming*, a idéia é manter as descrições e os fragmentos de programa separados, mas internamente ligados. Para isso, é necessário que a prática seja apoiada por uma ferramenta que produza uma navegação em ambiente WEB, usando dois "*frames*", um para a documentação e outro para o código. A principal vantagem nessa abordagem em relação ao caso anterior é que o programador deixa de ser forçado a reorganizar seus trechos de código, quando acrescenta novas descrições. A possibilidade de perda de sincronismo entre a documentação e o código, por estarem separados, é minimizada pela edição através da ferramenta, que disponibiliza sempre as duas janelas permitindo a visualização simultânea dos dois documentos. Um exemplo é apresentado na Fig. 3.4.

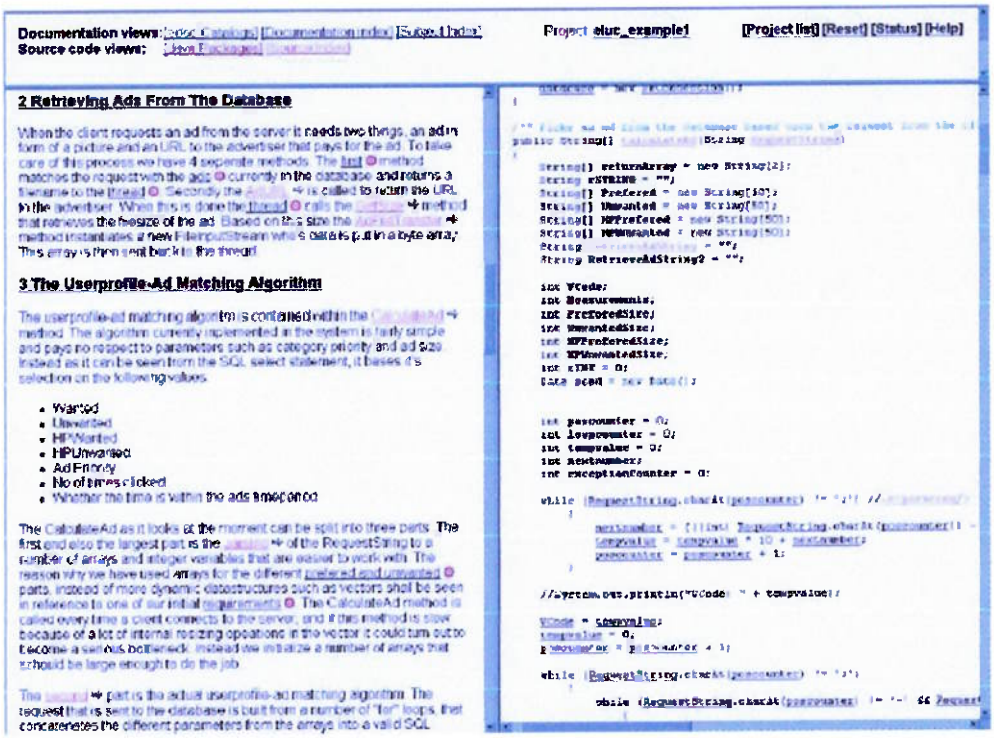

Figura 3.4 - Exemplo de *Elucidative Programming* apresentado num browser

3.2.3 Theme-based Literate Programing

A LP apresenta uma série de limitações. Kacofegitis e Churcher (2002)

apresentam em seu trabalho uma discussão sobre como ampliar a LP para solucionar algumas de suas limitações, permitindo a sua adoção por um maior número de desenvolvedores. Os autores chamam a sua ferramenta de *Theme-based Literate Programming*. Nesta proposta, os tipos de trechos em que o texto se divide não estão limitados a código ou documentação, e as ferramentas para extrair a documentação e o código estão embutidas em um modelo de processamento mais genérico. É introduzido o conceito de *temas*, que permitem uma grande liberdade aos autores para expressar seus pensamentos, sem necessariamente ter de seguir uma abordagem *top-down*.

Principais problemas da LP que a TBLP pretende resolver:

- as linguagens OO necessitam de facilidades de indexação e referência cruzada que facilitem a navegação entre classes com associações, por exemplo;
- os projetos atuais envolvem vários arquivos fontes, escritos em diferentes linguagens;
- a distinção entre código e documentação, em projetos que envolvem páginas HTML e JSP, imagens e texto, podem ser códigos em um contexto e documentação em outros;
- no mundo HTML, a documentação tem de ser altamente referenciada, visando à interatividade, e não ter a forma monolítica de uma cópia impressa;
- é difícil uma integração com sistemas de controle de versão e de configuração;
- depurar programas é difícil, uma vez que o programador trabalha com a versão "literata" do programa e o compilador retorna informações de erros por linha do código;
- o problema das "três sintaxes" (a ferramenta LP, a linguagem de programação e a diagramação) é um desafio para a construção de ferramentas;
- não está claro qual a melhor forma de projetar o software usando LP; a escolha do nível de detalhe e a ordem dos trechos é tão difícil como em outras técnicas.

A TBLP é motivada pela criação de três modelos:

- Modelo de trechos: mais tipos de trechos podem ser definidos, cada um com seus próprios atributos. A Fig. 3.5 apresenta um Modelo de Trechos, expresso em UML, onde se podem ver diferentes tipos de trechos como código,

documentos, testes etc.

- Modelo de tema: são permitidos vários caminhos de navegação sobre um conjunto de trechos, adequados para diferentes audiências e propósitos. A Fig. 3.6 apresenta um Modelo de Temas, onde vários níveis estão inter-relacionados.
- Modelo de processamento: a aparência e conteúdo dos documentos produzidos é independente do conjunto de trechos fonte e são configuráveis pelo usuário. A Fig. 3.7 mostra o Modelo de Processamento.

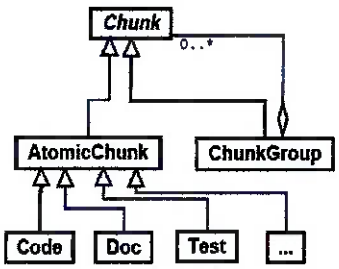

Figura 3.5 - Modelo de Trechos

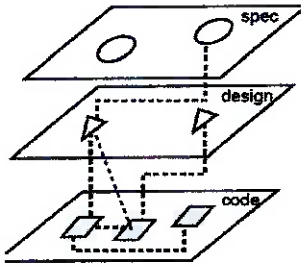

Figura 3.6 - Modelo de Temas

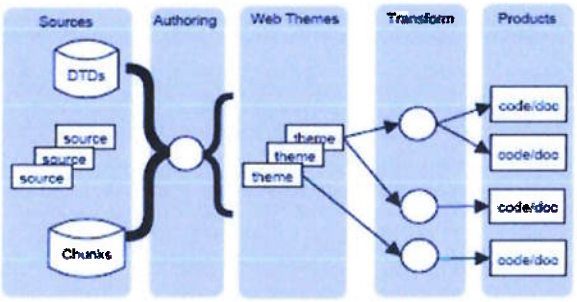

Figura 3.7 - Modelo de Processo

Uma ferramenta para suportar esse paradigma está em desenvolvimento pelos autores, mas muito de sua contribuição em esclarecer e propor soluções para muitos dos problemas apresentados pela LP pode ser utilizado também para expandir os conceitos de outras técnicas, como a Javadoc que será vista a seguir.

3.2.5 Documentação em comentários

Documentação embutida em documentários é, segundo o ponto de vista do autor, a melhor técnica para manter sincronizados código e documentação devido à alta proximidade. Entretanto, esse sincronismo não é automático e exige a disciplina do programador para mantê-lo atualizado.

Outras vantagens seriam a simplicidade de implantação, não intrusiva ao código, aplicável a qualquer linguagem que implemente um esquema de comentários interno ao código. Além disso, é altamente personalizável, permitindo ao cliente a especificação de seus próprios itens de documentação e formato/*layout* de saída.

Como será visto a seguir, Javadoc é a ferramenta padrão da linguagem Java usada para implementar documentação em comentários, apesar de encontrarmos outras com a mesma finalidade.

3.2.5.1 Javadoc e *doclets*

Javadoc é uma ferramenta desenvolvida pela Sun e distribuída junto com o JSDK (*Java Standart Development Kit*), que extrai as declarações de pacotes, classes, atributos e métodos e os comentários da documentação de um conjunto de arquivos de código fonte e produz um conjunto de páginas HTML, descrevendo e documentando essas declarações (SUN, 2003). Para estender essa documentação, o Javadoc faz uso de marcadores especiais nos comentários que guiam a ferramenta no tipo de documentação gerada.

A documentação gerada pelo Javadoc é conhecida como documentação de API e descreve a utilização de classes e métodos e a audiência a quem essa documentação se destina, que consiste, basicamente, de programadores.

Um exemplo de fonte documentado com "tags" Javadoc é apresentado na Fig. 3.8.

A documentação gerada em HTML está apresentada na Fig. 3.9

Exemplo:

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link url} The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param  url  an absolute URL giving the base location of the image
 *         name the location of the image, relative to the url arg.
 * @return     the image at the specified URL
 * @see        Image
 */
public Image getImage(URL url, String name) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null;
    }
}
```

Figura 3.8 - Programa com comentários em Javadoc

getImage

```
public Image getImage(URL url,
                      String name)
```

Returns an `Image` object that can then be painted on the screen. The `url` argument must specify an absolute URL. The `name` argument is a specifier that is relative to the `url` argument.

This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

Parameters:

`url` - an absolute URL giving the base location of the image

`name` - the location of the image, relative to the `url` argument

Returns:

the image at the specified URL

See Also:

Image

Figura 3.9 - Documento HTML correspondente ao código fonte da Fig. 3.8

Doclets são programas escritos em Java, usando uma API específica, que permite estender as marcações em Javadoc, criando saídas personalizadas, ou

adicionando novas marcações. Por exemplo, além do *doclet* padrão que produz a documentação em HTML, a Sun também disponibiliza um *DocCheck*, que faz uma checagem dos comentários e produz um documento apontando os erros, irregularidades e omissões. Outro exemplo é o *MIF Doclet*, que permite uma saída em formato PDF.

Outros *doclets* são produzidos por terceiros como por exemplo:

- UMLGraph: permite uma especificação usando marcações personalizadas e específicas para a criação de diagramas de classes em UML, utilizando a especificação de diagramas *Graphviz*, que permitem que, automaticamente, sejam gerados documentos em GIF, SVG ou JPEG. Maiores detalhes de sua implementação podem ser encontrados em Spinellis (2003). A Fig. 3.10 mostra uma saída obtida diretamente do código fonte com o uso de UMLGraph.

```
/*
 * Advanced relationships
 * UML User Guide p. 137
 */

/**
 * @opt attributes
 * @opt operations
 * @hidden
 */
class UMLOptions {}

class Controller {}
class EmbeddedAgent {}
class PowerManager {}

/**
 * @extends Controller
 * @extends EmbeddedAgent
 * @navassoc - - - PowerManager
 */
class  SetTopController
implements URLStreamHandler {
        int authorizationLevel;
        void startUp() {}
        void shutDown() {}
        void connect() {}
}

/** @depend -  - SetTopController */
class ChannelIterator {}

interface URLStreamHandler {
        void OpenConnection();
        void parseURL();
        void setURL();
        void toExternalForm();
}
```

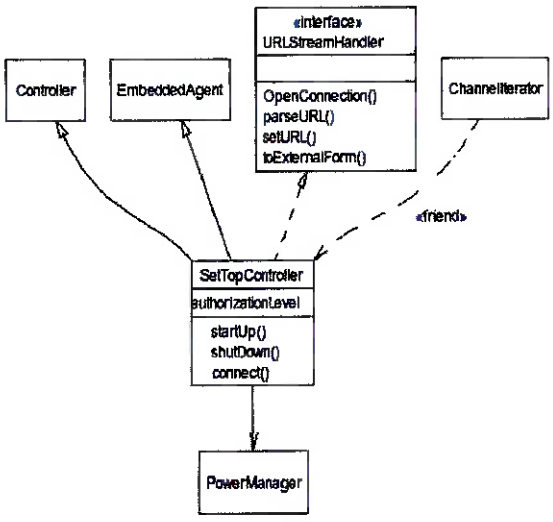

Figura 3.10 - Diagrama de classes UML obtido com UMLGraph

Outra funcionalidade possível para os *doclets* é a sua utilização para a geração de código. Um exemplo interessante é o seguinte, do *Patternity Doclet.*

O *Patternity doclet* é um *doclet* para documentação e geração de códigos, guiado por padrões de projeto (*design-patterns*). Este *doclet* utiliza marcações personalizadas de Javadoc para auxiliar na implementação de padrões de projeto, através da adição do código gerado - que nunca exclui ou altera um código já existente -, além de adicionar a documentação dos padrões ao Javadoc (MARTRAIRE, 2002).

A Fig. 3.11 apresenta a saída do *Patternity doclet.*

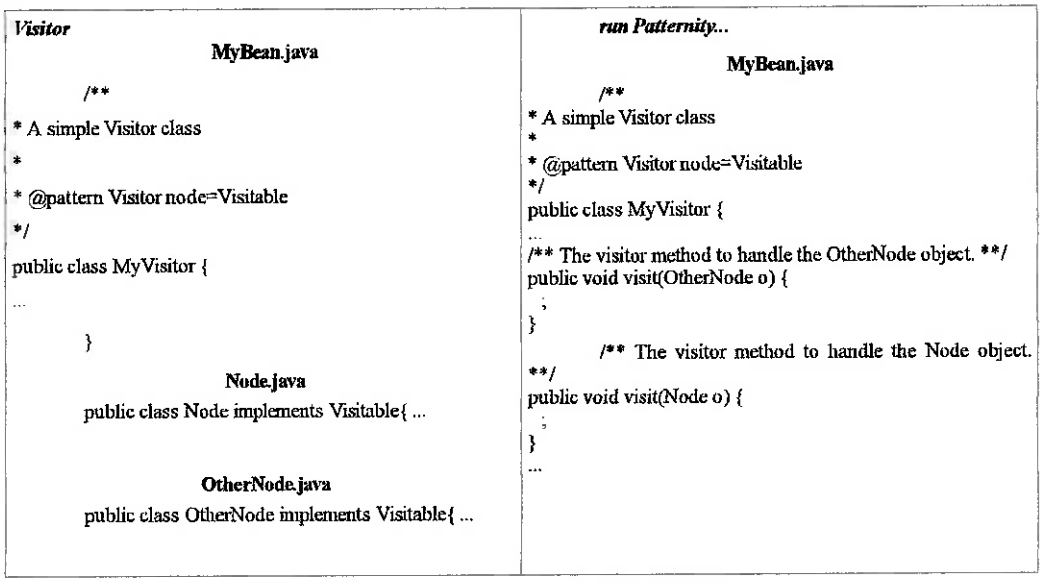

Figura 3.11 - Saída do *Patternity Doclet*

3.2 Especificação Formal e Semi-Formal

Especificação formal é um método de desenvolvimento de software através do qual se pode definir precisamente um sistema e desenvolver implementações garantidamente corretas em relação a esta definição.

Uma especificação formal é expressa através da utilização de um linguagem formal. Estas linguagens são baseadas em modelos matemáticos, oferecendo uma análise muito mais precisa das especificações, possibilitando a verificação dos modelos, animação, reutilização ou ainda refinamentos sucessivos.

A especificação de requisitos funcionais e a especificação formal podem ser

vistas como duas formas de representar o comportamento esperado do sistema, sendo que uma diferença entre elas é o grau de formalismo. Se por um lado, a especificação dos requisitos pode ser utilizada para validar as necessidades do sistema com o cliente, a especificação formal pode ser utilizada pelos engenheiros de software nas fases seguintes do ciclo de vida, através dos recursos supracitados.

A especificação pode ser dita semi-formal quando lida com alguns aspectos da formalidade. Neste estudo, serão vistos exemplos de formalismo apenas na especificação dos comportamentos desejados por classes e métodos que são compartilhados nas interfaces dos componentes, através da construção de assertivas (*assertions*) , como será visto no ítem seguinte.

3.2.1 Design by Contract

A técnica de desenvolvimento de software conhecida como *Design by Contract* (DBC) pretende permitir um software de alta qualidade ao garantir que cada componente de um sistema funcione segundo o que é esperado. Usando DBC, deve-se especificar os contratos do componentes como parte de sua interface. O contrato especifica o que é esperado pelo componente de seus clientes e o que os outros componentes esperam deste. (ENSELING, 2001; KRAMER, 1998).

Essa técnica foi desenvolvida por Bertrand Meyer como parte integrante de sua linguagem de programação, *Eiffel.* Entretanto, não deixa de ser uma técnica valiosa também para outras linguagens de programação.

O conceito central na DBC é a noção de assertivas *(assertions)*, uma expressão booleana sobre o estado do sistema. Durante a execução, essa expressão é avaliada em pontos de checagem específicos. Se por algum motivo essa expressão não for verdadeira, então podemos considerar que é sistema está inválido. São considerados três tipos básicos de assertivas: pré-condições, pós-condições e invariantes.

Pré-condições são as assertivas que devem ser satisfeitas antes de um método ser executado. Envolvem o estado do sistema e os argumentos passados para o método. Especificam obrigações que o cliente deste componente deve cumprir antes de poder chamar o método em questão. Se a expressão falhar, é porque o defeito está no software cliente.

Pós-condições são as que devem ser satisfeitas após o método ser completado.

Envolvem o estado anterior do sistema, o novo estado, os argumentos do método e o seu valor de retorno. Elas garantem o que o componente deve fazer para seus clientes. Se ela falhar, então o defeito está no componente.

Invariantes são as assertivas que devem ser satisfeitas em qualquer momento no qual o cliente possa chamar um método do objeto. São definidas como parte da definição da classe. São avaliadas antes e depois da chamada de um método e na instanciação da classe. Pode indicar um defeito tanto no cliente quanto no componente.

Todas as assertivas podem ser especificadas para uma classe e, automaticamente, se aplicam para suas sub-classes, assim como para classes que implementam uma interface.

3.2.1.1 Icontract

iContract é uma ferramenta desenvolvida para a utilização do DBC em Java, utilizando-se do mesmo princípio do Javadoc, ou seja, incluindo a especificação das assertivas através de marcadores especiais nos comentários de definição das classes e dos métodos.

Ela é apresentada por Enseling (2001), que a descreve como uma técnica de pré-processamento para incluir as expressões no código fonte Java. Ou seja, primeiro processa-se o seu fonte com as marcações em comentários, e depois se compila o código resultante pelo compilador Java. Entretanto, pelo fato de estarem contidas em comentários, essas assertivas podem simplesmente ser ignoradas ao se compilar diretamente o código fonte escrito. Assim pode-se usar a versão intermediária com a inclusão das assertivas, enquanto se testa o produto e encaminhar a versão sem as assertivas para o ambiente de produção. As assertivas podem também ser mantidas em algumas classes do ambiente de produção, que não tenham restrições críticas de desempenho. A Fig. 3.12 apresenta um exemplo.

```
package iContract.doc.tutorial.Person;

public interface Person
{
  /**
   * @post return > 0 // age always positive
   */
  public int getAge();

  /**
   * @pre age > 0 // age always positive
   */
  public void setAge(int age);
}
```

Exemplo de uma interface

```
package iContract.doc.tutorial.Person;

public class Employee implements Person
{
 private int age_;

 /**
  * @pre age > 0
  */
 public Employee( int age ) {
  age_ = age;
 }

 public int getAge()
 {
  return age_;
 }

 public void setAge( int age )
 {
  age_ = age;
 }
}
```

Exemplo de uma classe implementando a interface e acrescentando uma pre condição ao construtor

```
package iContract.doc.tutorial.Person;

public class Main {

  /**Test driver method.
   * Create an employee and print its age.
   */
  public static void main( String argv[] )
  {
    Employee employee = new Employee( 25 );
    System.out.println( "Employee's age is: " + employee.getAge() );

    employee.setAge( -1 ); // this will break!
  }
}
```

Exemplo de aplicação utilizando a classe passando parâmetros inválidos

```
$ java iContract.doc.tutorial.Person.Main
Employee's age is: 25
java.lang.RuntimeException: Employee.j:22: error: precondition violated
  (iContract.doc.tutorial.Person.Employee::setAge(int)):
  (/*iContract.doc.tutorial.Person.Person::setAge(int)*/ (age > 0))
at iContract.doc.tutorial.Person.Employee.setAge(Employee.java:124)
at iContract.doc.tutorial.Person.Main.main(Main.java:14)
```

Exemplo de saída apresentando a checagem do erro ocorrida

Figura 3.12 - Exemplo de iContract

A documentação das assertivas pode ser feita diretamente do código, através da ferramenta *iDoclet*, uma extensão do *doclet* padrão da SUN para o JDK 1.2. As assertivas são incluídas nos documentos API produzidos (SOURCEFORGE.NET, 2001).

3.3 Anotação de decisões de projeto e descrição da arquitetura

Uma importante contribuição à melhoria da compreensão do software é a possibilidade de incluir na documentação os modelos de projetos utilizados, as razões de sua escolha e a organização estrutural dos componentes e como eles se comunicam, através da descrição da arquitetura utilizada.

Apresentam-se três métodos e ferramentas que os implementam, relacionados a "codificar" decisões de projeto e de arquitetura dentro do código fonte de linguagem Java.

3.3.1 Explicit Programming

Bryant et al. (2002) introduzem o conceito do que foi chamado de *Explicit*

Programming (EP), ou seja, permitir ao desenvolvedor introduzir novo vocabulário ao código fonte. A definição de um item de vocabulário modulariza os detalhes de implementação associados com a representação de um conceito num código de propósito geral. O uso do item do vocabulário no código torna o conceito de projeto, como um *design pattern* ou um algoritmo de estruturas de dados, explícito para o leitor.

Resumidamente, é uma abordagem que suporta modularização, sem separação, de um conceito de projeto. Usando a EP, um desenvolvedor pode fazer com que decisões de projeto se tornem explícitas em seu código pela introdução incremental de novos vocabulários à linguagem, que podem ser usados onde o conceito acontece no código e cuja definição modulariza os detalhes de implementação do conceito.

3.3.1.1 Elide

Com o objetivo de demonstrar a utilização dos conceitos da EP, em Bryant et al. (2002), é descrita a ferramenta *Extension Language for Iterative Design Encoding* (ELIDE). A ferramenta é constituída de um pré-processador que transforma o código escrito na versão estendida de Java em código fonte padrão Java, de acordo com suas especificações. Ela provê uma extensão à linguagem Java, permitindo que o desenvolvedor crie novos modificadores para classes, atributos e métodos. A técnica permite que estas extensões sejam usadas da mesma forma que os modificadores padrão de Java como *public*, *private* e *synchronized* e, também, podem ser usados em conjunto com estes.

Cada modificador ELIDE inicia uma transformação no código fonte, que pode adicionar novas classes, inserir atributos, modificar métodos existentes ou gerar novos.

Um exemplo simples, apresentado na Fig. 3.13, mostra a introdução da palavra-chave *property*, que adiciona os métodos de acesso (*get* e *set*) no estilo *JavaBeans* a cada um dos atributos públicos de uma classe.

Código ELIDE	Código Java gerado
public class ElideTest {	**public class** ElideTest
property<> **public** int x;	{
property<> **public** String[] y;	**private** int x;
}	**private** String[] y;
	public int getX() {
	return x;
	}
	public void setX(int x) {
	this.x = x;
	}
	public String[] getY() {
	return y;
	}
	public void setY(String[] y) {
	this.y = y;
	}
	}

Figura 3.13 - Exemplo de transformação usando ELIDE

Outros exemplos de utilização da ELIDE, como um modificador que indica uma classe como participante de um padrão *Visitor,* e identifica com qual classe visitor ele interage, é apresentado a seguir

public *visited*<*NodeVisitor*> **class** ParseNode { ... }

Parâmetros também podem ser passados, como nesse exemplo que introduz o DBC usando ELIDE; nesse exemplo, o parâmetro é definido como positivo.

precondition<"positive", %{ x > 0 }%>

void someMethod (**int** x) { ... }

3.3.2 Aspect Oriented Programming

Em seu artigo Spurlin (2002), apresenta a noção de "Aspecto" dentro de uma linguagem de programação, que é similar a uma classe, mas em um nível de abstração acima. Eles encapsulam um conceito que pode afetar várias classes, mas que seria

difícil ou impossível de ser colocado em uma classe separada. Um exemplo ilustrativo dessa situação é quando se deseja escrever mensagens de *log* sempre que um método for chamado. Ao invés de procurar em todo o código onde o método é chamado e inserir manualmente as mensagens de *log* em cada classe, um aspecto poderia ser escrito com uma expressão indicando o ponto de inserção e uma única linha de código para a mensagem de *log*.

Conhecida como uma proposta para resolver o problema do entrelaçamento de código -por exemplo o entrelaçamento do código de negócio com o de controle-, a separação de conceitos é o principio básico da AOP (Mota, Farias,). Características que não são modularmente acopladas no paradigma orientado a objetos são classificadas como aspectos e isoladas do sistema original. Portanto, se um sistema é concorrente, podemos separá-lo em duas partes: uma não concorrente, que faz menção ao aspecto sobre concorrência e uma que contém o aspecto sobre como lidar com concorrência. Em tempo de compilação, os compiladores integram o programa original com os aspectos referenciados e geram um código na linguagem base.

A Fig. 3.14 mostra a utilização desta técnica, num ambiente gráfico.

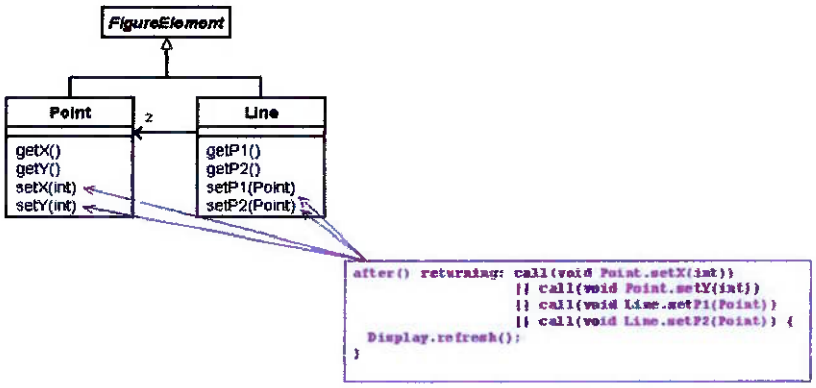

Figura 3.14 - Representação de um aspecto.

3.3.2.1 AspectJ

AspectJ é uma das ferramentas que implementam a AOP no ambiente Java. Essa ferramenta foi desenvolvida pelo mesmo grupo que definiu os conceitos da AOP e surgiu nos laboratórios da Xerox-PARC; hoje é mantida pela comunidade de código

livre.

Alguns conceitos utilizados em AOP são representados em *AspectJ* conforme descrito a seguir por Spurlin (2002)

- *Join Point*: é um ponto de execução em um programa Java. É o ponto onde será inserida alguma funcionalidade do aspecto. Exemplos de *join point* são:

  2. *Call* – o ponto onde um método é chamado,
  3. *Execution* o ponto em um objeto chamado, quando sua execução começa;
  4. Set – quando um atributo não privado recebe um valor.

  Exemplo:

  execution(*.getGreeting(..))*

  significa o início de execução de qualquer método chamado getGreeting com qualquer parâmetro.

- *Pointcut*: é essencialmente um predicado, definido sobre um conjunto de *join points*. Um *pointcut* particular pode ser verdadeiro ou falso em cada *join point* no código fonte. Ele consiste do identificador *pointcut*, de um identificador e de um predicado, a condição que deve ser verdadeira para um *join point* pertencer a este *pointcut*.

  Exemplo:

  pointcut log()

  execution(* *.getGreeting(..)) || *execution*(* *.setGreeting(..));

- *advice declaration*: é a parte executável de um aspecto. Consiste de um *advice type*, que determina exatamente quando a execução ocorre, seguido de um identificador de *pointcut* e seguido por uma parte executável chamada de *advice body*.

  Exemplo:

  before() : *log*() { System.out.println(" Logger: " + thisJoinPoint.getSignature()); }

- *inter-type member declaration*: é um mecanismo para adicionar membros a uma classe de fora da definição dessa classe, sem modificar o código fonte da classe.

Dessa forma é possível adicionar atributos e métodos que não estão relacionados com o propósito principal da classe.

- *aspect*: é um construção de mais alto nível, que pode conter *pointcuts*, *advice declarations*, *inter-type members declarations* e outras declarações que serão válidas, como numa definição de membros de uma classe Java.

Um programa exemplo, contendo um aspecto para exibir uma mensagem, é mostrado na Fig. 3.15.

```
public class Demo {
    static Demo d;
    public static void main(String[] args){
        new Demo().go();
    }
    void go(){
        d = new Demo();
        d.foo(1,d);
        System.out.println(d.bar(new Integer(3)));
    }
    void foo(int i, Object o){
        System.out.println("Demo.foo(" + i + ", " + o + ")\n");
    }
    String bar (Integer j){
        System.out.println("Demo.bar(" + j + ")\n");
        return "Demo.bar(" + j  + ")";
    }
}
```

Figura 3.15 – Trecho de programa

As Fig. 3.15 e Fig. 3.16 mostram um exemplo de uso de AOP. A Fig. 3.15 apresenta o código de negócio enquanto a Fig. 3.16 apresenta o código de controle. O código da Fig. 3.16 mostra o aspecto *getInfo*, que intercepta as chamadas aos métodos *foo* e *bar* e mostra as informações obtidas no pointcut.

```
aspect GetInfo {
  static final void println(String s){ System.out.println(s); }
  pointcut goCut(): cflow(this(Demo) && execution(void go()));
  pointcut demoExecs(): within(Demo) && execution(* *(..));
  Object around(): demoExecs() && !execution(* go()) && goCut() {
    println("Intercepted message: " +
          thisJoinPointStaticPart.getSignature().getName());
    println("in class: " +
          thisJoinPointStaticPart.getSignature().getDeclaringType().getName());
     printParameters(thisJoinPoint);
     println("Running original method: \n" );
     Object result = proceed();
    println("  result: " + result );
    return result;
  }
  static private void printParameters(JoinPoint jp) {
    println("Arguments: " );
    Object[] args = jp.getArgs();
    String[] names = ((CodeSignature)jp.getSignature()).getParameterNames();
    Class[] types = ((CodeSignature)jp.getSignature()).getParameterTypes();
    for (int i = 0; i < args.length; i++)
      println("  " + i + ". " + names[i] + " : " + types[i].getName() +" = " +args[i]);
  }
}
```

Figura 3.16 – Aspecto GetInfo (com informações no pointcut)

3.3.3 Architecture Description Languages

Uma *Architeture Description Language* (ADL) enfatiza estruturas de alto nível, se opondo a detalhes de implementação e engloba algumas propriedades que se tornam desejáveis e importantes para a descrição de arquiteturas de um produto de software. Em Lazilha (2001), é apresentado que uma ADL deve ser simples, compreensível e possibilitar uma sintaxe gráfica bem compreendida, mas não necessariamente uma semântica formalmente definida.

ADLs são linguagens formais que podem descrever e representar produtos de software, e possuem características que permitem especificar a natureza dos

componentes, suas propriedades, a semântica das conexões e o comportamento do sistema especificado. Uma ADL suporta a descrição de um sistema em termos de componentes e conectores.

A descrição de uma arquitetura, formalizada através de uma ADL, pode ajudar na especificação e análise do projeto de alto nível. Ela também facilita a implementação e evolução de grandes sistemas. Aldrich et al. (2002) mostram, porém, que a análise de arquitetura nas ADLs existentes podem revelar importantes propriedades arquiteturais, mas não garantem que estejam presentes na implementação.

Para mostrar as razões arquiteturais de uma implementação, a mesma deve estar em conformismo com sua arquitetura, segundo três critérios:

- decomposição: para cada componente na arquitetura deve haver um na implementação;
- conformidade da interface: cada componente na implementação deve estar em conformismo com a interface arquitetural;
- Integridade de comunicação: cada componente na implementação só pode comunicar-se diretamente com o componente ao qual está conectado na arquitetura.

3.3.3.1 ArchJava

Para o ambiente Java, foi desenvolvida uma ADL, chamada ArchJava (ARCHJAVA), através de uma extensão da linguagem Java, obtendo o desejável enfoque na proximidade com a implementação. Aldrich et al. (2002) mostram que a ArchJava permite um estilo de programação orientada a objetos flexivel, permitindo compartilhamento de dados e dando suporte a arquiteturas dinâmicas, onde componentes são criados e conectados em tempo de execução.

Um recurso da ArchJava que não é encontrado em outras ADLs é um sistema "tipado" que garante a integridade de comunicação entre uma arquitetura e sua implementação, mesmo na presença de objetos compartilhados e configurações de arquiteturas dinâmicas.

Como exemplo de um programa usando ArchJava, apresenta-se a arquitetura de um WEB Server. O subcomponente ROUTER aceita as requisições HTTP que

chegam, passando-as a um conjunto de componentes WORKER, que processam a resposta. Quando chega uma requisição, o ROUTER requisita uma nova conexão a um WORKER através de sua porta "request". O WEB Server então cria uma nova instância de WORKER e o conecta ao ROUTER. O ROUTER passa as requisições ao WORKER através de sua porta "workers". A Fig. 3.17 apresenta esta arquitetura. A Fig. 3.18 apresenta a descrição desta arquitetura.

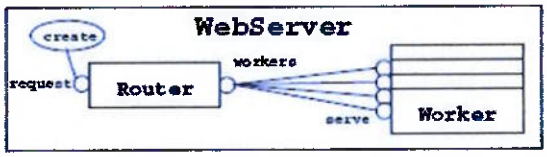

Figura 3.17 – Arquitetura do Web Server

O compilador ArchJava aceita uma lista de arquivos ArchJava (.archj), compila cada um para um código fonte Java, e chama o compilador javac sobre os arquivos . java resultantes. O compilador ArchJava traduz cada classe componente em uma classe comum Java, deixando os atributos e os corpos do métodos praticamente inalterados.

No trabalho apresentado em Sazawal et al. (2002) é indicado que ArchJava possui a possibilidade de expandir os tipos de conectores, permitindo que o programador crie os seus próprios. Um *conector* é um elemento de projeto reutilizável, que suporta um estilo particular de interação entre componentes. Cada conector é modularmente definido em sua própria classe. Componentes interagem com conectores de forma clara, usando a sintaxe já existente da chamada de procedimentos em Java. Em ArchJava, cada conector é traduzido para uma classe normal em Java que implementa a estratégia de comunicação.

Como exemplo dessas estratégias, temos oito principais tipos de conectores descritos: chamada de procedimento (AsynchronousConnector, TCPConnector), evento (EventDispatchConnector), acesso a dados (CachingConnector, PersistConnector), ligação , encadeamento (BufferedStreamConnector), árbitro (LoadBalancingConnector), adaptador (AdaptorConnector) e distribuidor (EventDispatchConnector).

```
public component class WebServer {
    private final Router r = new Router();
    connect r.request, create;
    connect pattern Router.workers, Worker.serve;

    public void run() { r.listen(); }
    private port create {
        provides r.workers requestWorker() {
            final Worker newWorker = new Worker();
            r.workers connection = connect(r.workers, newWorker.serve);
            return connection;
        }
    }
}
public component class Router {
    public port interface workers {
        requires void httpRequest(InputStream in, OutputStream out);
    }
    public port request {
        requires this.workers requestWorker();
    }
    public void listen() {
        ServerSocket server = new ServerSocket(80);
        while (true) {
            Socket sock = server.accept();
            this.workers conn = request.requestWorker();
            conn.httpRequest(sock.getInputStream(),  sock.getOutputStream
            ());
        }
    }
}
public component class Worker extends Thread {
    public port serve {
        provides void httpRequest(InputStream in, OutputStream out) {
```

Figura 3.18 – Programa ArchJava que implementa a arquitetura do Web Server.

No site ArchJava (ArchJava) são descritos alguns benefícios em se aplicar a ferramenta ArchJava, a saber:

- Através da integridade de comunicação, pode-se garantir que o código fonte de um programa permanece consistente com a arquitetura enquanto o mesmo evolui em resposta às mudanças dos requisitos.
- A arquitetura tornada explícita em um programa ArchJava ajuda os programadores a entender a interação entre diferentes partes de um código fonte, o que auxilia a tratar as tarefas de evolução do software de maneira mais eficiente.
- A checagem da integridade de comunicação do ArchJava ajuda a melhorar a estrutura do programa ao capturar as violações de encapsulamento.

3.4 Geração de códigos a partir de modelos UML

Um dos maiores argumentos usados por quem não produz documentação e modelos do projeto é que praticamente nada pode ser automaticamente aproveitado na implementação. O ideal seria a possibilidade de ter um sistema sendo processado diretamente a partir dos modelos, como se estes fossem uma espécie de linguagem de programação de altíssimo nível.

Este poderia ser o modelo ideal de documentação de sistema, que dispensaria o código fonte, eliminando qualquer problema de falta de sincronismo ou de documentação desatualizada, principalmente aquela referente a decisões de projeto e rastreamento de requisitos.

Utilizando o UML como linguagem de modelagem, por ser esta padronizada e altamente difundida, fica mais fácil produzir ferramentas que consigam tal proeza, apesar das limitações existentes. A seguir analisamos duas propostas que tentam atingir esse objetivo.

3.4.1 Ferramentas CASE (Computer-Aided Software Engineering)

As mais recentes ferramentas CASE do mercado já apresentam recursos para a

geração de código a partir de modelos UML, com possibilidades de alterar o código fonte e poder refletir essa alteração no modelo e vice-versa. Essas técnicas são conhecidas como *forward* e *reverse engeneering*.

Porém, na maioria, essas ferramentas estão limitadas à utilização dos modelos estáticos: são capazes de gerar o código estrutural das classes, ou seja, as definições de atributos e métodos. Algumas poucas ferramentas, dispõem de recursos básicos para incorporar os modelos de comportamento, geralmente o diagrama de estado.

Como neste trabalho estão sendo abordadas ferramentas *open source* e apenas como ilustração de um conceito, apresenta-se uma ferramenta dessa categoria que implementa uma funcionalidade básica de geração de código.

3.4.1.1 Poseidon/ArgoUML (ferramenta CASE em UML)

ArgoUML iniciou-se como um projeto de pesquisa de Jason Robbins e depois colocado como um projeto *open-source,* ao qual outros colaboradores se juntaram. Alguns desses colaboradores montaram uma empresa e decidiram criar uma derivação do projeto, mantendo uma versão comercial do produto, denominada *Poseidon for UML.* O foco principal no desenvolvimento dessa ferramenta é a integração de UML, JAVA, MDA e XML, mantendo sempre as características de usabilidade e alta produtividade.

Em Boger et al. (2003), estão indicados os recursos existentes nesta ferramenta para a geração de código, engenharia reversa e *round-trip engineering.* Neste estudo apenas nos interessa a geração de código, visto que é o único recurso disponível na versão *free.*

A ferramenta possui um mecanismo flexível de geração de código baseado em *templates.* A geração de código é baseada exclusivamente no diagrama de classes. A associação entre as classe é, por definição, bidirecional. A geração para o código de uma associação é feita de forma a permitir a navegação nos dois sentidos.

Existem algumas possibilidades de ajustar a geração de código, como por exemplo:

- métodos de acesso: pode-se marcar quais atributos terão os métodos de acesso gerados ou não;
- tipos de associação: dependendo da multiplicidade da associação, a

implementação pode se dar através de um atributo (1..1, 0..1), um elemento do tipo *Colletion* (..*), ou um *array* (..<n>);

- definição do comando *Import*: pode ser feito graficamente no diagrama ou através da definição do tag *JavaImportStatement*.

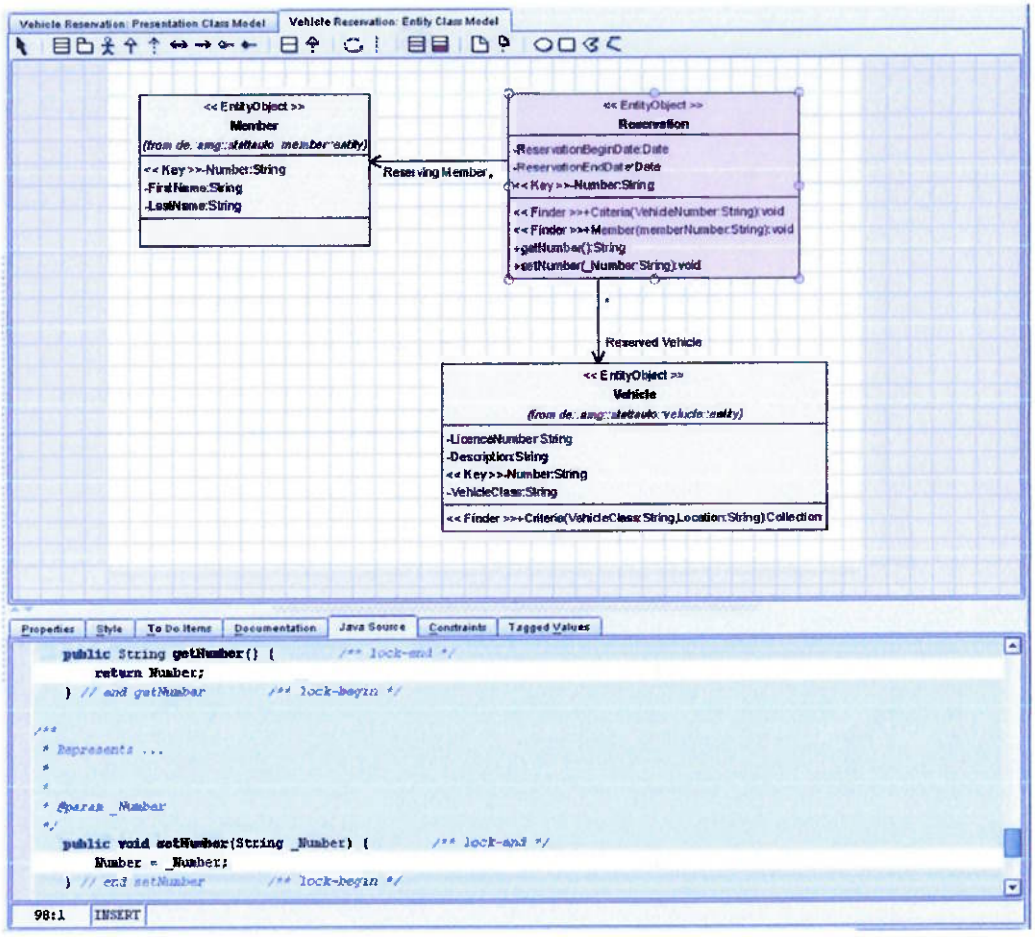

Figura 3.19 – Tela do sistema Poseidon for UML

A Fig. 3.19 apresenta uma tela do Poseidon for UML. Na parte superior da tela apresenta-se o Diagrama de Classes e na parte inferior o código gerado, correspondente.

3.4.2 xUML - Actions Semantics

A *Executable UML* (xUML) é uma metodologia automatizada que usa um subconjunto altamente especializado da notação UML. Esse subconjunto compreende:

o diagrama de classe UML, que especifica a estrutura do sistema através da descrição dos objetos que o compõem, diagrama de estado UML, indicando o ciclo de vida de cada objeto como uma máquina de estado finitos e uma *Actions Semantics*, uma proposta de extensão à UML que permite a especificação de comandos executáveis de alto nível, que especifica o comportamento dos objetos ao mudarem de estado no seu ciclo de vida. A natureza "executável" desses modelos é que permite que eles facilmente possam ser implementados em uma linguagem de programação tradicional, ou até mesmo serem executados diretamente por um interpretador que pode ser implementado diretamente na ferramenta CASE. A Fig. 3.20 apresenta um exemplo de um Diagrama de Estados com notações de ações em xUML.

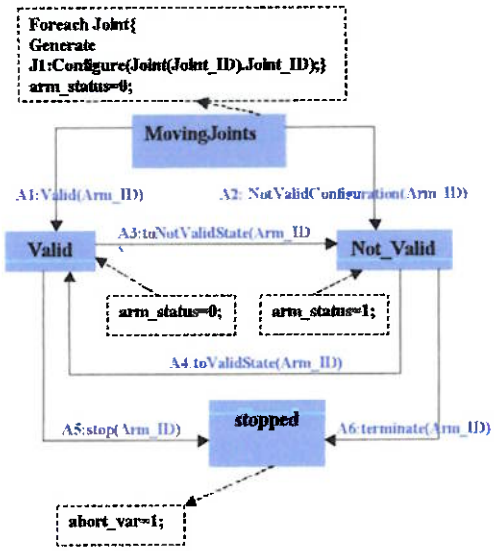

Figura 3.20 – Exemplo de Diagrama de Estados com xUML

As regras de execução do xUML especificam que os mesmos modelos podem ser traduzidos em uma variedade de arquiteturas diferentes, sem mudanças nos modelos. Múltiplos modelos xUML podem ser montados juntos, para construir sistemas mais complexos.

Exemplo desse conjunto de ações de alto nível de abstração, que resultam em uma forma de execução da especificação em UML, foi retirado da apresentação de Ignjatovic (s/data) da PAL - *Pathfinder Solutions Action Language*, e são

apresentados na figura 3.21.

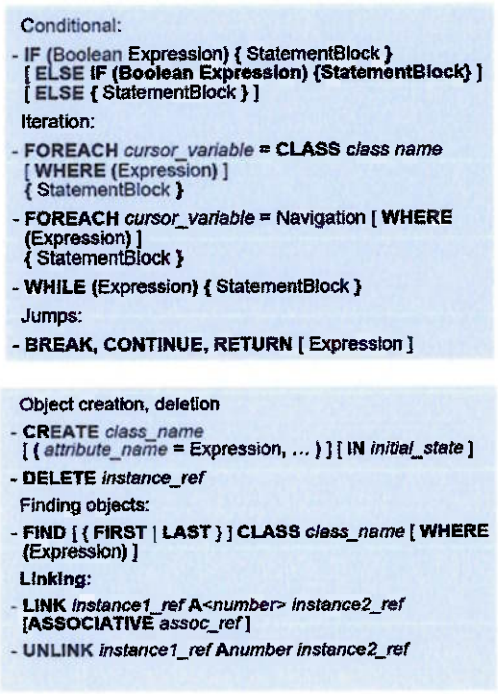

Figura 3.21 - Exemplo de um conjunto de ações de uma implementação da xUML

3.5 Conclusão

Neste capítulo procurou-se apresentar um conjunto representativo de métodos e ferramentas relacionados à documentação de software, direcionados ao ambiente Java e disponíveis como *open source*.

Podemos classificar as técnicas apresentadas neste capítulo em 4 grupos, cada um com seus pontos positivos, sendo mais indicado para registrar um ou outro aspecto da documentação:

- Documentação independente do código fonte (ex.:REM, Elucidative Programming), a grande desvantagem é a dificuldade de sincronismo com o código fonte, porém pode ser utilizados em documentos em que não tenham desenvolvedores como audiência principal;

- Documentação gerada a partir do código fonte (ex. Javadoc, Literate Programming, Theme-based Literate Programming), é a mais flexível e extensível, permitindo a criação de padrões personalizados de documentação para cada empresa, cobrindo todos os tipos de documentos, gerados automaticamente. Permite o reaproveitamento de um ítem da documentação em mais de um documento. Fácil de implementar e de manter o sincronismo com o código por estarem fisicamente juntos;
- Código fonte gerado a partir da documentação (ex. Poseidon, xUML), seria a solução ideal, onde, a partir da modelagem de alto nível, poderia-se ter o produto diretamente. No entanto não existe ainda uma implementação que cubra toda a geração de código fonte, sendo necessária uma codificação manual, podendo ser vista como um retrabalho pelos desenvolvedores. De qualquer forma, é um recurso que deve ser usado sempre que possível para automatizar parte do desenvolvimento;
- O próprio código fonte é a documentação (ex. Icontract, Elide, AspectJ, ArchJava), não pode ser considerado como uma documentação, mas um complemento desta. Podem ser usados como ferramentas de automação do processo e de auxílio a compreensão do software, ao documentarem as decisões de projeto e arquitetura;

Na análise feita por este autor, compreende-se que a documentação por comentários dentro do próprio fonte é a que melhor atende às soluções dos problemas referentes à documentação: sincronismo, automação, reutilização, diversas audiências, simplicidade e adaptação.

Usando *doclets* personalizados, e definindo uma estrutura para os ítens básicos de documentação apresentados no capítulo 2, ítem 2.5.2, podemos usar o conceito da Javadoc e criar praticamente toda a documentação através de comentários do código fonte. A extração e geração são automatizadas, combinando e reutilizando trechos para gerar os diversos documentos.

No capítulo seguinte, sugerimos uma forma para essa documentação em Javadoc num pequeno exemplo de programa comentado.

4 ESTUDO DE CASO: DOCUMENTAÇÃO NO CÓDIGO JAVA

Apresenta-se um exemplo da utilização de algumas das técnicas apresentadas no capítulo 3, sempre que possível, ilustrando o princípio da documentação no próprio código. Ou seja, todo o código fonte da documentação estará escrito em arquivos de programas fontes em linguagem Java, que podem ser compilados para gerar um programa executável, ou passar por outras ferramentas que geram a documentação em um formato impresso (como um manual) ou navegável (como um *help on-line*).

Para se atingir alguns ítens não cobertos pelas ferramentas apresentadas, serão feitas algumas sugestões de como elas podem ser implementadas em expansões de ferramentas existentes ou em ferramentas futuras cuja implementação, entretanto, está fora do escopo deste trabalho.

Neste exemplo serão usados alguns dos ítens de documentação descritos na taxonomia apresentada no ítem 2.5.2.

Um módulo do sistema de Controle de Acesso (MDCBe) , responsável pela comunicação e controle de placas de comando com comunicação Ethernet, via TCP/IP, será usado na exemplificação. A Fig. 4.1 apresenta este exemplo.

A aplicação que aqui será usada como exemplo, será parcialmente descrita para fins ilustrativos de sua documentação.

As primeiras fases de um projeto de software são as mais difíceis de se documentar no código fonte, devido a não existir uma relação "um para um" simples entre o código e os documentos, e também porque a linguagem Java tem a estrutura montada em um arquivo fonte para cada classe. Assim, documentação que se relaciona a um grupo de classes, ou até mesmo ao sistema inteiro, não possui um único local onde possa ser registrada. Situa-se neste caso a documentação da visão geral do sistema, dos requisitos não-funcionais e de algumas outras partes.

Entretanto, através da utilização de ArchJava, pode-se criar essas estrutura de mais alto nível, onde podem ficar documentadas as estruturas mais globais de um projeto de software, além de descrever diretamente no código a arquitetura proposta. ArchJava permite hierarquia de arquiteturas, podendo, assim, descrever as relações entre subsistemas e depois quebrar-se esses subsistemas em partes menores.

Pode-se descrever, em um arquivo fonte ArchJava, o componente principal, que é o programa, e nele acrescentar a documentação que trata do projeto como um todo.

Para registrar tais documentos no código fonte, pode-se usar o princípio da JavaDoc, usando *tags* específicos em comentários. Neste caso, utilizam-se *tags* novos, personalizados, que efetivamente não existem na JavaDoc, mas que podem ser, facilmente, implementados no futuro.

```
/**
 * @projeto MDCBe:Módulo MDCBe //visão geral
 * Este módulo é responsável pelo recebimento das informações em tempo-real dos
 * crachás lidos em uma placa BeNet, verificar no banco de dados as condições para
 * a liberação e comandar a liberação (destravar fechadura ou catraca)
 * e indicação luminosa (led) adequada, sendo ( verde ou vermelho)
 *
 * @contato.solicitante Reginaldo, Trielo, Depto. Comercial, fone:xxxx-xxxx
 * @contato.operador Gilberto, Trielo, Depto Tecnico, fone xxxx-xxxx
 *
 * @ferramentas Ambiente IDE: NetBeans
 *  ArchJava para descrição da arquitetura.
 *  Banco de dados: access, Oracle e SQL-Server
 *
 * @repositorio.tipo CVS
 * @repositorio.local \\servidor\CVS\projetos\MDC\MDCbe
 *
 *
*/
public component class MdcBe()
{...}
```

Figura 4.1 – Módulo de exemplo MDCBe

Os requisitos do sistema, principalmente os funcionais, podem ser descritos basicamente de duas maneiras:
através de uma lista de recursos desejados,
através da descrição do comportamento através das iterações com o usuário, conhecido como cenários ou casos de uso.

Em ambas as situações, a principal dificuldade é o fato de um requisito envolver várias classes na implementação.

Observando a orientação a objetos, percebe-se que a descrição de cenários apresenta maior facilidade de mapeamento com a implementação, principalmente se os passos do cenário forem introduzindo os objetos internos do sistema que interagem para a produção da resposta ao estímulo do usuário. Mesmo assim, haveria uma dispersão, pois cada "passo" descrito no cenário "estaria" em uma classe distinta. A solução neste caso, é manter nesta classe apenas a referência ao cenário/passo, e

manter a descrição do cenário como um "todo" no componente principal que o implementa ou é responsável pela iniciação do processo. É conveniente lembrar que todos os *tags* são opcionais. Os itens usados na descrição dos casos de uso foram baseados em Gelperin (2003).

Outro ítem de documentação importante é o glossário. Neste caso, os termos que apresentam definição no glossário podem ser definidos a qualquer momento. Os ítens do glossário são referenciados no texto dos comentários entre sinais de '!' (!glossário!), indicando uma possibilidade de *hiperlink* . A Fig. 4.2 mostra alguns exemplos de glossário. A figura mostra também exemplos relativos a outros *tags*.

Decisões de projeto podem ser documentadas onde elas aparecerem, seja sobre um componente, uma classe ou método. Normalmente uma decisão de projeto é tomada para atender um ou mais requisitos não-funcional, e o mesmo deve sempre ser anotado. A Fig. 4.2 mostra a documentação dos requisitos funcionais e não funcionais.

```
/**
 *
 * @Glossario{ //adiciona ítens ao glossario
 *   [leitora]: equipamento que interpreta o código existente em um crachá
 * do mesmo tipo de tecnologia, como código de barra, rádio-freqüência (proximidade)
 * smart-card, e do mesmo fabricante. O código é então transferido à controladora.
 *   [controladora]: placa responsável pela comunicação entre os elementos de
 * bloqueio (catracas, portas) e o sistema MDC. Ela recebe a leitura de um crachá e
 * aciona um relê e/ou led executando a liberação ou aviso de um bloqueio.
 *   [BeNet]: modelo de !controladora! com comunicação via rede ethernet.
 *   [Log da ocorrência]: registro no banco de dados do sistema contendo
 * basicamente o número do crachá, a data/hora da leitura, o código da ocorrência
 * (tipo de liberação ou motivo do bloqueio) e a controladora/leitora.
 * }
 *
 * @UC.id UC001:Recebe crachá    //requisitos funcionais - casos de uso
 * @UC.atores AT001:Controladora BeNet:Deseja verificar se o crachá apresentado
 *   possui ou não !permissão de acesso!
 * @UC.desc Processo de solicitação de liberação de crachá. Quando um usuário
 *   apresenta um crachá numa !leitora! controlada pela !BeNet!, a mesma envia
 *   uma solicitação ao sistema para a checagem e poder enviar a resposta na forma
 *   de comandos de liberação ou bloqueio para a !Controladora!.
 * @UC.trigger Mensagem UDP enviada pela placa ao sistema na porta 66
 * @UC.pre O sistema deve ter conseguido estabelecer conexão com o banco de dados
 * @UC.pos Deve ter sido registrado o !Log da ocorrência!
 * @UC.fonte Reginaldo
 * @UC.risco{Medio} @UC.prioridade{Alta} @UC.custo{22HH} @UC.prazo{DL:15/10/2003}
 * @UC.1 [Controladora BeNet] envia na porta 66 do IP do host, a mensagem UDP
 *    <Lnccccccccc>, onde n=número da leitora, c=número do crachá
 * @UC.1.det //usado para descrever no manual do usuário mais detalhes
 *    O aplicativo MDCBe é iniciado na linha de comando com a sintaxe:
 *      >java MDCbe mdc.props
 * @UC.2 [MDCBe] verifica o IP de origem da mensagem no cadastro de placas e envia
 *    a mensagem para ser processado pela thread ProcessaAcesso correspondente
 * @UC.2a [MDCBe] o IP de origem não está cadastrado, então é criada uma nova
 *    instância de ProcessaAcesso
 * @UC.3 [ProcesaAcesso] consulta para verificar a existência do crachá
 * @UC.3a.1 [ProcesaAcesso] não existe -> envia sinal de bloqueio (Led vermelho) na
 *   porta UDP 37 da placa.
 * @UC.3a.2 [ProcesaAcesso] registra !Log da ocorrência! Com ocorrência 881-usuario
 *   não cadastrado.
 * ...
*/
```

Figura 4.2 – Documentação para o módulo exemplo MDCBe

```
/**
 * @decisao 001:A arquitetura do módulo MDC constitui de dois processos (threads),
um <Escuta> que fica aguardando uma comunicação na porta 66 e depois repassa a
mensagem recebida para o processo <ProcessaAcesso> correspondente à placa que gerou
a mensagem.  @atende:[melhor modularidade]
 * @decisao 002:Foi escolhido usar um <ProcessaAcesso> para cada placa, para liberar
o componente que recebe a mensagem para poder processar rapidamente outro
recebimento. Cada <ProcessaAcesso> processa e responde à requisição feita pela sua
placa. @Atende:[minimo de 5 req/s], [resposta até 1.5s]
*/
// tradicional comentário Javadoc
/**
 * Classe/Componente principal do módulo MDCbe.
 * Este módulo implementa o Caso de Uso 001: Recebe crachá.
 *
 * @autor GMM
 * @version 1.0
 */
public component class MdcBe()
{
 private final Escuta e = new Escuta();
 private Connection con=DriverManager.getConnection(JdbcUrl, JdbcUser, JdbcPass);
 private Vector controladoras= new Vector;
 connect e.request, select;
 connect pattern Escuta.send, ProcessaAcesso.processa;

 public static void main(){
   ResultSet rs = con.createStatement().executeQuery("select ip from unidades");
   while (rs.next()) {
     q = new ProcessaAcesso();
     controladoras.addElement(q);
     q.start();
   }
   e.start();
 }
 private port select{
   provides e.send achaIP(Vector unid, String IP) {
     for(int i=0; i<unid.size(); i++) {
      if (unid.elementAt(i).ip.equalsTo(IP) ) {
        final ProcessaAcesso pa= (ProcessaAcesso)unid.elementAt(i);
        e.send connection = connect(e.send, pa.processa);
      }
     }
   }
 }
}
```

Figura 4.2 – Documentação para o módulo exemplo MDCBe (Cont.)

4.1 Conclusão

Apresentou-se neste capítulo, um exemplo de como expandir a utilização de

tags Javadoc para a inclusão de novos tipos de documentação. Este pequeno exemplo serve para demonstrar a flexibilidade da documentação inserida no código, e tornar evidente as tarefas complementares necessárias:

- Construir os *doclets* que implementam as rotinas de extração e geração da documentação;
- Elaborar os *templates* para geração dos documentos personalizados para cada organização;
- Selecionar um conjunto de *tags* representativo do grupo de documentos que se espera gerar;
- Definir um método para mapear os itens de documentação nos elementos estruturais da Java, como pacotes, classes e métodos.

5 CONCLUSÕES

Este trabalho apresentou um trabalho de pesquisa sobre as diversas ferramentas do ambiente Java relacionadas com a documentação de um sistema de software. Essa pesquisa procurou esclarecer alguns pontos sobre os problemas de manter uma documentação atualizada, consistente com o produto, a fim de agilizar a etapa de compreensão do código e localização das classes e métodos afetados por uma mudança de requisito, durante uma etapa de manutenção.

Produzindo uma documentação com qualidade desde o início do desenvolvimento, e fornecendo recursos para facilitar o sincronismo com o código e automatização do seu reuso, torna-se efetiva a comunicação entre os diversos participantes nas várias fases, mesmo aqueles que tem necessidades distintas, pois pode-se filtrar e apresentar apenas aquilo que lhe é importante.

No início do capítulo 2, apresentou-se duas questões sobre as dificuldades de se produzir a documentação que agora podem ser respondidas:

- A documentação deve a forma mais adequada ao tipo de projeto, processo utilizado e características da empresa, desde que o custo e prazo para produzi-lá não inviabilize o projeto.
- Os analistas serão encorajados a escrever a documentação quando ela é simples de elaborar, reaproveitável em grande parte, sem ter de ser reescrita em diversos documentos diferentes, está próxima para ser atualizada e sempre ao alcance quando ela é necessária.

Apresentou-se a visão que diferentes processos de desenvolvimento tem da importância da documentação, os principais artefatos que guiam o desenvolvimento, a fim de exemplificar os diferentes tipos de documentos e suas formas. Esclareceu-se também as principais finalidades da documentação – requisitos, modelo de contrato, comunicação entre os participantes e registro de decisões – que, conjuntamente com o processo, definem o conteúdo da documentação.

A simplicidade de implementação da documentação, o reaproveitamento e os filtros para a geração de documentos distintos, não só reduzem o custo de produção da própria documentação, como também o custo de manutenção do produto.

O capítulo 3 foi reservado à apresentação de várias técnicas e ferramentas relacionadas. Elas cobrem 4 grupos de documentação: independente do código fonte, é o método comumente utilizado, onde se sobressai a dificuldade de sincronismo com o código fonte; gerada a partir do código fonte, é a mais flexível e extensível, fácil de implementar e de manter o sincronismo; código fonte gerado a partir da documentação, onde poderia-se ter o produto diretamente a partir da modelagem de alto nível; o próprio código fonte é a documentação, através de extensões da linguagem Java para registrar conhecimentos das decisões de projeto.

Encerra-se o trabalho no capítulo 4 indicando que a utilização da técnica de incluir a documentação em comentários nomeados e padronizados, que podem ser extraídos e transformados nos documentos desejados, é a potencialmente mais indicada a atender as necessidade de uma boa documentação: simplicidade, proximidade do código, reaproveitamento e automatização da geração. Atende a qualidade de se permitir sincronizar com o código fonte.

Entretanto, um trabalho adicional deve ser executado para adaptar a solução ao ambiente do projeto e equipe de desenvolvimento. Esse trabalho consiste nas atividades de construir as rotinas de extração e geração da documentação, elaborar os *templates* para a personalização dos documentos, selecionar um conjunto de *tags* representativo e definir um método para mapear os ítens de documentação nos elementos estruturais da Java. Este último é o mais complexo, pois Java não apresenta estruturas de alto nível, e também porque o mapeamento de diversos ítens da documentação não são "um para um" com o código fonte.

Outras técnicas e ferramentas podem ser utilizadas em conjunto, como descrição da arquitetura, formalismo das interfaces e explicitação das decisões de projeto. Elas auxiliam a comunicação entre os membros e a compreensão pelos que farão a manutenção posteriormente.

REFERÊNCIAS:

ALDRICH, J.; CHAMBERS, C.;NOTKIN, D. Architectural Reasoning in ArchJava. In Lecture Notes in Computer Science. Springer-Verlag Heidelberg. v.2548. 2002. ECOOP 2002 Workshop, Málaga, Spain, June/2002. *Proceedings.* J. Hernández, A. Moreira (Eds.). Disponível em: http://archjava.fluid.cs.cmu.edu/papers/ecoop02.pdf. Acesso: 10/12/2003.

AMBLER, S. W. *Agile Documentation.* Disponível em: The Official Agile Modeling (AM) Site. http://www.agilemodeling.com/essays/agileDocumentation.htm Acesso: 05/12/2003.

[ArchJava]___ ArchJava: Home. Disponível em: http://www.archjava.org . Acesso em: 10/12/2003.

[Arkley] ARKLEY, P.; MASON P.; RIDDLE S. *Position Paper: Enabling Traceability.* 1st. International Workshop on Traceability in Emerging Forms of Software Engineering (in conjunction with the 17th IEEE International Conference on Automated Software Engineering), Edinburgh, UK, 28 September 2002. Proceedings. p. 61-65.

[Boehm] BOEHM, B. *Software Cost Estimation with COCOMO II,* Prentice Hall, 2000.

[Boger] BOGER, M.; STURM, T.; SCHILDHAUER, E.; GRAHAM, E. *Poseidon for UML Users Guide*, 2003, http://www.gentleware.com/products/index.php4

[Booch] BOOCH, G.; MARTIN, R. C.; NEWKIRK, J. *Object Oriented Analysis and Design with Applications.* 1998. Addison Wesley Longman, Inc.

[Bryant] BRYANT, A.; CATTON, A.; DE VOLDER, C.; MURPHY, G. C. *Explicit Programming.* AOSD' 2002. 1st International Conference on Aspect-Oriented Software Development, Enschede, the Netherlands, Apr. 22-26. *Proceedings.* p. 10-18.

[Bryant] BRYANT, A.; CATTON, A.; DE VOLDER, C.; MURPHY, G. C. *ELIDE: Explicit Programming for Java.* Disponível em:

http://www.cs.ubc.ca/labs/spl/projects/explicit.html. Acesso em: 10/12/2003.

[Bryant] BRYANT, A.; CATTON, A.; DE VOLDER, C.; MURPHY, G. C. *Explicit Programming: Improving the Design Vocabulary of Your Program.* Demonstração em OOPSLA 2001. Disponível em: http://www.cs.ubc.ca/labs/spl/papers/2001/OOPSLA2001-EP-demo.pdf Acesso: 10/12/2003.

[Cysneiros] CYSNEIROS, L. M.; LEITE, J. C. S. P.; SABAT NETO, J. M. *A Framework for Integrating Non-Functional Requirements into Conceptual Models,* Requirements Eng, 2001.

[De Marco]DE MARCO, T. *Why Does Software Cost So Much?*, Dorset House, 1995 ENSELING, O. iContract: Design by Contract in Java. *JavaWorld.* Fev., 2001. Online: http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-cooltools_p.html . Acesso:02/12/2003.

[Ducasse] DUCASSE, S.; NIERSTRASZ, O. *Tie Code And Questions: a Reengineering Pattern*, University of Bern, 2000.

[Durán] DURÁN A.; BERNÁRDEZ, B.; RUIZ A.; TORO, M. *An XML–based Approach for the Automatic Verification of Software Requirements Specifications*, 2000, (REM), University of Seville, 2000.

[Enseling) ENSELING, O. *iContract: Design by Contract in Java*, JavaWorld, fev/2001. Online: http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-cooltools_p.html. Acesso: 10/12/2003.

JACOBSON, I.; BOOCH, G.; RUMBAUGH, J. *The Unified Software Development Process.* Addison-Wesley, Boston, Ma. 1999.

[Gelperin] GELPERIN, D. *Precise Use Cases*, 2003, http://www.livespecs.com/downloads/LiveSpecs06V01_PreciseUseCases.pdf.

[Ignjatovic] IGNJATOVIC, M.; *UML with Action Semantics: Concepts, Application and Implications*, Zuehlke Engineering AG, Março/2003. Online: http://www.jugs.ch/html/events/slides/xUML.PDF, Acesso: 14/12/2003

[Jeffries] JEFFRIES. R. *Essential XP: Documentation.* 21/12/2001. Disponível em: Xprogramming.com. An Extreme Programming Resource. http://www.xprogramming.com/xpmag/expDocumentationInXp.htm Acesso em:

05/12/2003.

[Kacofegitis] KACOFEGITIS, A.; CHURCHER, N. *Theme-Based Literate Programming.* Relatório Técnico TR-COSC 03/02. 2002. University of Canterbury. Disponível em: http://citeseer.nj.nec.com/cache/papers/cs/26184/http:zSzzSzwww.cosc.canterbury.ac.nzzSzresearchzSzreportszSzTechRepszSz.zSz2002zSztr_0203.pdf/theme-based-literate-programming.pdf. Acesso em: 10/12/2003.

[Ian03] KAPLAN, I.; *Software and Documentation,* Fev/2003. http://www.bearcave.com/software/prog_docs.html. Acesso em: 14/12/2003

[Kotula] KOTULA, J. *Source Code Documentation: An Engineering Deliverable,* Technology of Object-Oriented Languages and Systems (TOOLS 34'00), **July 30 - August 03,** Santa Barbara, California. 2000. P. 505.

KRAMER, R. *iContract – The Java Design by Contract Tool.* Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings , 3-7 Aug. 1998. p. 295 – 307.

[Lazilha] LAZILHA, F. R. ; PRICE, R. T.; GIMENES, I. M. de S. *Fundamentos para uma Proposta de Arquitetura de Linha de Produção para Workflow Management Systems.* VI Workshop de Teses em Engenharia de Software. SBES 2001. Disponível em: http://www.din.uem.br/~expsee/docs/artigos/Artigo-Itana_Fabricio.pdf. Acesso em: 10/12/2003.

[Patternity] MARTRAIRE, C. *Patternity - Pattern-driven code generation and documentation doclet.* Versão: 2002-11-27. Disponível em: http://www.patternity.netliberte.com. Acesso em: 10/12/2003.

[Mota] MOTA, A.; FARIAS, A. *Identificação de Aspectos Apoiada por Computador - Projeto de Pesquisa.* Projeto de Pesquisa. Faculdade Integrada do Recife – FIR. Disponível em: www.fir.br/edital/si/projeto_2.doc. Acesso em: 10/12/2003.

[Normark] NORMARK, K.; VESTDAM, T. *Elucidative Programming in Computer Science Education.* 2001 (unpublished). Disponível em: http://dopu.cs.auc.dk/publications/eluc_in_edu.pdf. Acesso em: 10/12/2003.

[Pacheco] PACHECO, R. F.; SANCHES, R. *Keeping the Software Documentation Up*

to Date in Small Companies, CLEI Electronic Journal 3, 2000.

[Clements] PARNAS, D. L.; CLEMENTS, P. C. *A RATIONAL DESIGN PROCESS: HOW AND WHY TO FAKE IT.* (sem data) Disponível em: http://cobolreport.com/columnists/parnas&clements/09152003.asp. Acesso em 05/12/2003.

[Parnas] PARNAS, D. L.; MADEY, J. *Functional Documents for Computer Systems*, Science of Computer Programming, 1995

[Paulk] PAULK, M. C. *Extreme Programming from a CMM Perspective.* IEEE Software, Los Alamitos, v. 18, n. 6, Nov./Dec. 2001. p. 19-26.

Poseidon for UML. Versão 2.1. 15/09/2003. Disponível em: http://www.gentleware.com/. Acesso: 10/12/2003.

RAMSEY, N. *Literate Programming Simplified.* IEEE Software. Los Alamitos. v. 11 , n. 5, Sept. 1994. p. 94-105. (Reprinted). Online: http://www.eecs.harvard.edu/~nr/pubs/lpsimp-abstract.html Acesso: 02/11/2003.

[Sazawal] SAZAWAL V.; ALDRICH, J.; CHAMBERS, C.; NOTKIN, D. *Language Support for Connector Abstractions.* Technical Report UW-CSE-02-04-01. University of Washington. 2002. Disponível em: ftp://ftp.cs.washington.edu/tr/2002/04/UW-CSE-02-04-01.pdf. Acesso em: 10/12/2003.

[Smith] SMITH, D. *Designing Maintainable Software*, Springer-Verlag, 1999.

SourceForge.net. *Project: iContract Plus: Summary.* Disponível em: http://sourceforge.net/projects/icplus/ . Versão 1.0. 20/09/2001. Acesso em 10/12/2003.

[UMLGraph] SPINELLIS, D. D. *UMLGraph - Declarative Drawing of UML diagrams.* Software Público. Version 1.24 - 2003/07/30. Disponível em: http://www.spinellis.gr/sw/umlgraph/. Acesso em: 10/12/2003.

[Spurlin] SPURLIN, V. *Aspect-Oriented Programming with Sun ONE Studio.* Nov. 2002. Disponível em: http://developers.sun.com/tools/javatools/articles/aspectJ.html Acesso em: 10/12/2003.

[STOUT]STOUT, G. A. *Requirements Traceability and the Effect on the System*

Development Lifecycle. Research Paper 2. Springer Cluster, 2001.

[Javadoc] SUN. *The Javadoc 1.4.2 Tool.* Disponível em: http://java.sun.com/j2se/1.4.2/docs/tooldocs/javadoc/ . Acesso em 10/12/2003.

[Arie01] VAN DEURSEN, A. *Program Comprehension Risks and Opportunities in Extreme Programming*, SEN-R0110 May 31, 2001, CWI Report