Uma Abordagem Sistemática para a Instanciação de Frameworks Orientados a Objetos

Candidato : Toacy Cavalcante de Oliveira
Orientador : Carlos José Pereira de Lucena

Agradecimentos

É importante ressaltar a influência e participação de várias pessoas no decorrer deste trabalho. Sem elas seria muito mais árduo o caminho rumo ao doutoramento. A todos aqui mencionados gostaria de reportar meus sinceros agradecimentos.

Orientador Carlos José Pereira de Lucena pelo apoio e acolhida dados durante os momentos difíceis e decisivos para a elaboração deste trabalho bem como por proporcionar momentos intensos de interação seu grupo de pesquisa.

Professor, mentor e amigo Sérgio Eduardo Rodrigues Carvalho pelo encorajamento e apoio dados desde o período de graduação quando começamos a trabalhar juntos. Durante o período em que interagimos foi o principal responsável por todos os ensinamentos na área de Engenharia de Software bem como grande responsável por mudanças significativas em meu modo de pensar a vida.

Professor Paulo Alencar, pelo apoio dado na fase final de formulação do escopo do trabalho.

Professores Arndt von Staa, Eber Assis Schmidt e Julio Cesar Leite, pelo apoio dado na fase de revisão deste trabalho.

Aos colegas Ivan Mathias Filho, Sylvia Oliveira e Cruz, Cristina von e Flach Viviane Torres por momentos intensos de debates que em muito contribuíram para este trabalho.

A Vera Sotero Menezes por todo apoio logístico e psicológico dado.

Aos meus pais Niva e Toacy pelo patrocínio e apoio o caminho dentro da área acadêmica.

A minha esposa Angela e a meus filhos Pedro e Julia por toda dedicação e paciência durante todo este árduo processo.

Resumo

A incorporação de técnicas que aumentem a produtividade e qualidade no ciclo de desenvolvimento de software é um dos maiores obstáculos a serem ultrapassados pela disciplina de Engenharia de Software. Dentre estas técnicas, a reutilização do conhecimento adquirido em situações similares é um dos veículos mais eficazes no tocante a diminuição de fatores como custo e tempo (diretamente ligados a produtividade). Um passo importante para que esta reutilização seja adotada é a especificação do processo envolvido que em última análise serve de guia para o (re) utilizador de um artefato de software. Neste cenário frameworks orientados a objetos representam um grande avanço uma vez que capturam o conhecimento sobre um domínio particular de aplicações no qual reutilização pode ser obtida em larga escala. Em seu processo de reutilização, chamado de instanciação, é especificado um conjunto atividades que devido ao número e complexidade (oriundos da condicionalidade e seqüencialidade), devem ser executadas de forma sistemática, com o auxílio de uma ferramenta computacional de modo a impactar positivamente o ciclo de desenvolvimento. Tendo isto em mente, este trabalho tem como objetivo apresentar um conjunto de técnicas tais como; documentação do framework; linguagem de descrição do processo de reutilização; execução assistida do processo de reutilização, que estão integradas em um ambiente que auxilia o reutilizador do framework na execução deste processo.

Índice

1 Introdução 1-1

1.1 Motivação 1-1

1.2 Contexto do Problema 1-5

1.3 Visão Geral da Abordagem Proposta 1-7

1.4 Contribuições 1-13

1.5 Evolução do Trabalho 1-14

1.6 Estrutura da Tese 1-16

2 Técnicas de Reutilização 2-17

2.1 Introdução 2-17

2.2 Framework 2-20

2.2.1 Introdução 2-20

2.2.2 Definição 2-21

2.2.3 Vantagens & Desvantagens 2-24

2.2.4 Classificação 2-28

2.2.5 Desenvolvimento 2-29

2.2.6 Instanciação 2-31

2.2.7 Documentação 2-33

2.3 Componentes 2-38

2.4 Design Patterns 2-40

2.5 Aspectos e Concerns 2-42

2.6 Conclusão 2-43

3 Tecnologias Relacionadas 3-44

3.1 Ambientes de Reutilização 3-44

3.2 Especificação do Design OO – UML 3-50

3.3 Representação do Design OO – XMI....3-52

4 Motivação - Um Exemplo de Reutilização.... 4-55

5 Documentação da Reutilização.... 5-68

5.1 Identificação dos Pontos de Extensão....5-74

5.2 Integração com o Meta-Modelo de UML 5-88

5.3 Resumo....5-93

6 Especificação do Processo de Reutilização.... 6-96

6.1 A BNF Comentada....6-98

6.2 O Desenvolvimento do Cookbook....6-103

6.3 Execução de um Cookbook....6-106

6.4 Um Exemplo....6-117

6.5 Resumo....6-123

7 A Ferramenta xFIT.... 7-125

7.1 XFIT - Visão geral....7-126

7.2 Arquitetura Interna....7-128

7.3 Funcionalidade....7-131

7.4 Interface....7-134

7.5 Resumo....7-138

8 Conclusões & Trabalhos Futuros....139

8.1 Conclusões....139

8.2 Trabalhos Futuros....143

Apêndice A - Histórico de Trabalhos....148

Apêndice B - Diagramas de xFIT....154

Bibliografia....158

Índice de Figuras

Figura 1 - Composição da aplicação final. 1-3

Figura 2 Fases do processo tradicional + Fases da reutilização. 1-8

Figura 3 Processo de reutilização utilizado pela abordagem. 1-9

Figura 4 Visão geral do ambiente de reutilização. 1-12

Figura 5 Frameworks como ferramenta de reutilização de design. 2-22

Figura 6 Hot-Spots como mecanismos de extensão de frameworks. 2-23

Figura 7 Instanciação do Framework 2-25

Figura 8 Inversão de Controle existente na aplicação de um Framework. 2-26

Figura 9 Escopo X Abstração 2-31

Figura 10 Processo de Instanciação. 2-33

Figura 11 Utilização de um componente. 2-39

Figura 12 Instanciação via Inspetor de objetos. 3-45

Figura 13 Tela do Program Explorer que apresenta a interação dos objetos observáveis. .3-47

Figura 14 Especificação da instanciação de Hooks[Froelich97a] 3-48

Figura 15 – Ambiente HiFI. 3-49

Figura 16 Exemplo de uma regra em HIFI. 3-50

Figura 17 Mecanismos de extensão de UML. 3-52

Figura 18 Integração das aplicações via XMI 3-53

Figura 19 Ambiente 2GOOD. 4-56

Figura 20 Diagrama de Classes do Framework DTFrame. 4-59

Figura 21 Solução do problema de sobreposição semântica via modularização. 4-63

Figura 22 – Atividades de instanciação inter-dependentes. 4-66

Figura 23 Aumento do numero de aplicações com a introdução da opcionalidade. 5-71

Figura 24 Forma de uma instancia válida. 5-72

Figura 25 No DTFrame, a especificação dos aspectos opcionais têm que ficar destacados de alguma forma. 5-73
Figura 26 Propagação da especificação de opcionalidade. 5-78
Figura 27 Hierarquia de elementos reutilizáveis. 5-79
Figura 28 – Especificação da Classe Figure como classe para extensão. 5-80
Figura 29 Aplicando o padrão Strategy. 5-81
Figura 30 Especificação de um ponto de reutilização que utiliza Design Pattern. 5-82
Figura 31 Representação das escolhas possíveis de uma extensão por seleção. 5-83
Figura 32 Especificação de redefinição do método draw(). 5-84
Figura 33 Especificação da redefinição de método via Design Pattern. 5-85
Figura 34 Especificação de uma extensão por atribuição de valor. 5-87
Figura 35 Especificação de extensão por seleção de valor. 5-88
Figura 36 Representação de elementos reutilizáveis como um Diagrama de Classes. 5-89
Figura 37 Incorporação ao metamodelo de UML. 5-90
Figura 38 Obtenção do Cookbook. 6-103
Figura 39 Abordagem evolutivo incremental para o desenvolvimento do Cookbook. 6-106
Figura 40 Alocação da variável figClass a sua respectiva atribuição. 6-107
Figura 41 Escolha dentre as atividades. 6-108
Figura 42 Seqüencialidade na execução. 6-108
Figura 43 Execução concorrente de A e B. 6-109
Figura 44 Execução do CLASS_EXTENSION. 6-109
Figura 45 Execução do SELECT_CLASS_EXTENSION. 6-110
Figura 46 Execução do PATTERN_CLASS_EXTENSION. 6-111
Figura 47 Execução do METHOD_EXTENSION 6-111
Figura 48 Execução do PATTERN_METHOD_EXTENSION. 6-112

Figura 49 Execução do VALUE_ASSIGNMENT 6-113
Figura 50 Execução do VALUE_SELECTION. 6-113
Figura 51 Execução do REQUIRES. 6-114
Figura 52 Execução de REQUIRES BEFORE. 6-114
Figura 53 Execução de REQUIRES AFTER 6-115
Figura 54 Execução de REQUIRES SYNC. 6-116
Figura 55 Execução do REQUIRES EXLUSIVE. 6-116
Figura 56 Padrão Factory Method 6-121
Figura 57 Execução de Factory Pattern. 6-121
Figura 58 – DTFrame após execução do cookbook de instanciação. 6-123
Figura 59 Insumos da ferramenta xFIT. 7-127
Figura 60 Arquitetura interna de xFIT. 7-128
Figura 61 Execução do Compilador. 7-129
Figura 62 Execução do distribuidor de tarefas. 7-130
Figura 63 Execução da Máquina de Execuçã de XFIT. 7-130
Figura 64 Atividades em que o reutilizador está envolvido. 7-131
Figura 65 Especificação da atividade de carga de artefato. 7-132
Figura 66 Especificação do Caso Executa Instanciação. 7-134
Figura 67 Introdução da interface GUI à ferramenta. 7-134
Figura 68 Tela principal de XFIT. 7-136
Figura 69 Tela para capturar CLASS_EXTENSION 7-137
Figura 70 Tela para capturar ELEMENT_CHOICE. 7-137
Figura 71 Tela para capturar SELECT_CLASS_EXTENSION. 7-138
Figura 72 Relacionamento dos principais componentes de xFIT. 155
Figura 73 Diagrama de Sequencia para Carga do Artefato 156

Figura 74 Diagrama de Sequencia para execução do processo de instanciação. 157

Índice de Tabelas

Tabela 1 Publicações X Contribuições.....1-15

Tabela 2 Camadas do Metamodelo da OMG. 3-54

Tabela 3 Especificação das Classes do Framework DTFrame. 4-58

Tabela 4 - Representação dos meta-elementos introduzidos por UML-FI. 5-91

Tabela 5 Resumo da especificação em UML-FI. 5-94

Tabela 6 Comparação entre a atividade com e sem a declaração da variável.....6-120

1 Introdução

1.1 Motivação

O processo de desenvolvimento de software vem sofrendo uma crescente demanda por desenvolvimento de sistemas onde qualidade, custo e "time-to-market" são fatores determinantes para o sucesso. Tendo isto em mente, a utilização de técnicas consideradas fundamentais para disciplina de engenharia, se tornam imprescindíveis. Estas técnicas permitem que durante processo de desenvolvimento de software sejam utilizados elementos cuja eficácia e eficiência foram previamente comprovadas. Dentre estas técnicas, padronização e reutilização [Biggerstaff89] podem ser citadas como duas das mais eficazes, pois permitem que desenvolvedores de software abandonem a abordagem "made-from-scratch" e durante o ciclo de desenvolvimento do software, passem a incorporar soluções cuja utilização tenha sido previamente confirmada.

Os benefícios trazidos pela reutilização podem ser traduzidos como:

- Uma diminuição do tempo de desenvolvimento, uma vez que o artefato reutilizável representa uma parte já desenvolvida do software como um todo.
- Uma melhoria na qualidade do produto final, uma vez que o artefato reutilizável já passou por uma fase de teste.
- Um aumento no vocabulário da linguagem utilizada durante o processo de desenvolvimento de software, uma vez que o artefato reutilizável introduz sua própria linguagem para representar conceitos mais abrangentes (conceitos reutilizáveis).
- Uma diminuição do custo total de desenvolvimento, dado a diminuição do tempo construção e redução do tempo de manutenção do software.

A materialização desta reutilização se dá através do uso de um artefato de software reutilizável, ou simplesmente artefato reutilizável. Este artefato contém a especificação de uma determinada funcionalidade a qual propiciará um ganho de qualidade (custo, tempo, etc..) ao processo de desenvolvimento de software como um todo.

Atualmente, a disciplina de engenharia de software provê técnicas como Bibliotecas, Componentes [Szyperski98] , Design Patterns [Gamma95] , Frameworks [Pree95] [Fayad99a], Aspectos [Kiczales97] e Concerns [Tarr00], que permitem dentre outras coisas a representação de artefatos reutilizáveis e a reutilização do conhecimento[1] adquirido em desenvolvimentos anteriores. Esta reutilização se dá de forma organizada sendo usualmente apoiada em um estilo de desenvolvimento bem conhecido, como por exemplo chamada de funções (programação estruturada), herança (programação orientada a objetos), uso de interface (programação baseada em componentes) etc.. Dentre estas técnicas, frameworks vem se tornando uma alternativa bastante atraente, pois sua reutilização (instanciação) permite a geração de sistemas inteiros de forma rápida que podem ser distribuídos a diversos clientes [Jacobson97]. Além destas técnicas ainda é possível citar Sistemas Gerativos [Staa93] [Batory98] e Sistemas Transformacionais [Neighbors84] [Leite94] como incentivadores da reutilização.

Uma vez que as técnicas de reutilização usualmente não representam um sistema completo e sim especificações e/ou implementações de soluções parciais, o processo de utilização

[1] Dado que software deixa de ser encarado como um mero produto e passa a ser visto como um processo de aquisição de conhecimento em um domínio [Armour00].

passa por uma atividade de parametrização para que o reutilizador adeque esta especificação de acordo com suas necessidades. Vale ressaltar que esta adaptação deverá preservar a semântica original do artefato reutilizável. Como apresentado na figura 1, a aplicação final é obtida pelo somatório do artefato reutilizável e do artefato reutilizador.

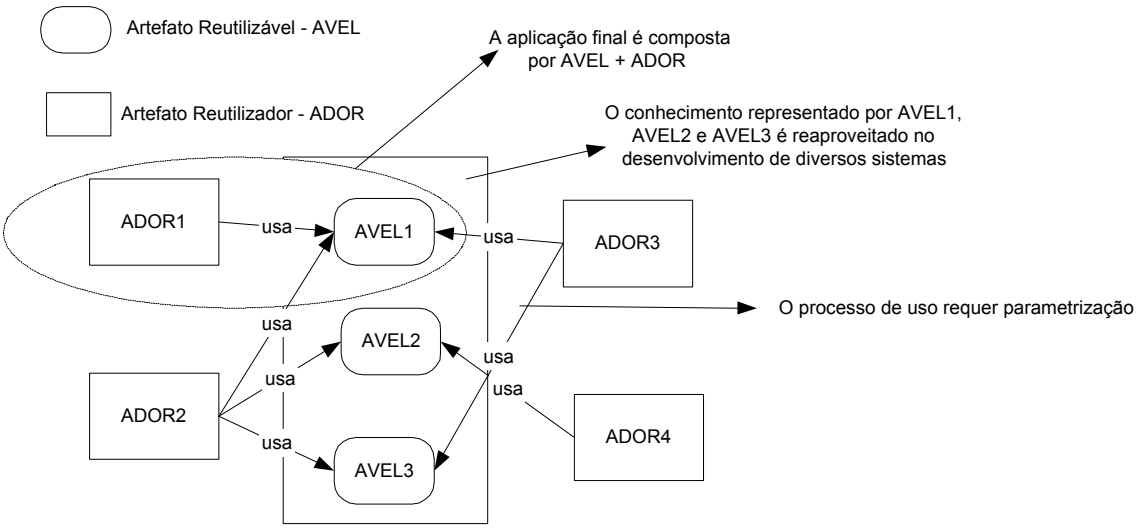


Figura 1 - Composição da aplicação final.

A parametrização mencionada acima, dá origem a um processo complexo que envolve a composição, em nível semântico e sintático, da especificação reutilizável (artefato reutilizável) com a especificação inicial do sistema em desenvolvimento (artefato reutilizador). Tal complexidade exige um suporte computacional que automatize e controle ao máximo este processo de composição/reutilização, uma vez que este envolve várias ações interligadas e é usualmente feito de forma dispersa no tempo e espaço.

Tendo isto em mente, o objetivo deste trabalho é apresentar um conjunto de técnicas que sistematizem a definição e uso de artefatos reutilizáveis, a saber:

⇒ Documentação precisa dos pontos vitais para o processo de reutilização através da extensão da Linguagem de Descrição de Frameworks [Fontoura99].

⇒ Programação do processo de reutilização utilizando-se uma linguagem denominada RDL (Reuse Description Language) especialmente desenvolvida para tal propósito.

⇒ Execução e verificação do processo de reutilização utilizando-se uma ferramenta denominada xFIT (XMI based Framework Instantiation Tool), que permite a execução assistida de programas escritos na linguagem RDL, bem como possibilita a manipulação das representações envolvidas no processo.

A utilização destas técnicas permite que especificações parciais de sistemas orientados a objetos possam ser combinadas de forma sistemática, através de um processo controlado que auxilia o desenvolvedor a obter o design final do sistema em desenvolvimento. Este processo controlado permite dentre outras coisas a identificação, atenuação e documentação de problemas inerentes à composição de especificações como sobreposição semântica e sintática de entidades [Mattsson00] e aparecimento de herança múltipla [Mattsson00]. Um outro benefício observado é relativo a estrutura final do design da aplicação. Uma vez que a execução deste processo pode ser não contínua e/ou escalonada, tarefas interdependentes podem ter seus requisitos de execução violados levando a uma estrutura final mal formada. Sendo assim, a verificação estrutural do design final se faz necessária.

Os benefícios desta abordagem são alcançados através:

⇒ Do uso de UML [Booch99] para representação dos modelos de design relevantes.

⇒ De mecanismos de extensão de UML para representação dos pontos importantes ao processo de reutilização.

⇒ Representação em XMI[XMI] dos modelos utilizados para a especificação do artefato reutilizável.

Um ponto importante da abordagem apresentada neste trabalho é referente ao termo artefato reutilizável. Podemos dizer que este é um termo "guarda-chuva" pois cobre uma diversidade de técnicas de reutilização. Neste trabalho utilizaremos este termo para significar Frameworks, pois em última análise esta técnica de reutilização cobre Bibliotecas, Componentes e de certa forma Concerns e Aspetos .

1.2 Contexto do Problema

Como dito anteriormente e apresentado por, [Biggerstaf89], [Poulin99], [Jacobson97], [Dusink95], [Pree95] e [Armour00], reutilização é uma técnica eficaz e eficiente para se produzir software. Em [Jacobson97] é reportado uma estatística que apresenta resultados de várias empresas produtoras de software comprovando a melhoria de sua produtividade, a saber:

- Redução de 2 a 5 vezes no tempo de entrega.
- Redução de 5 a 10 vezes na quantidade de defeitos.
- Redução de 5 a 10 vezes do custo de manutenção.
- Redução no custo de desenvolvimento como um todo de aproximadamente 15% até 75%, incluindo o overhead gasto para desenvolver artefatos reutilizáveis.

Quanto à reutilização de bibliotecas e componentes, temos um exemplo grande sucesso no desenvolvimento de interfaces gráficas, que corrobora com estas estatísticas. O sucesso de ambientes de programação ("GUI Builders") como Borland Integrated Development

Environment para Delphi[Delphi], Java[JBuilder] e C++[CBuilder], IBM VisualAge para Java[VAge] e C++[VAge], comprovam que a reutilização de artefatos bem definidos é possível ao menos dentro do mesmo domínio de aplicação (Interfaces Gráficas). Neste caso, o processo de reutilização se dá através da parametrização de propriedades e/ou operações para prover a customização necessária do artefato reutilizável. Esta customização é normalmente feita tomando como base uma documentação de reutilização informal, normalmente não estruturada, que obviamente pode levar a erros de interpretação. No caso do domínio das interfaces gráficas, a existência de uma linguagem comum, construída ao longo de anos de desenvolvimento, facilita o entendimento do processo de reutilização por parte do reutilizador, tornando desnecessário o desenvolvimento de uma documentação rigorosa para artefato reutilizável.

Quando falamos em reutilização de frameworks problemas mais sérios surgem uma vez que de em geral seus reutilizadores são inexperientes no domínio em questão, dificultando a obtenção/entendimento de uma linguagem comum. Uma vez que frameworks são aplicações semi-acabadas para um domínio[1] [Fayad99a] temos que, em primeiro, lugar situar a aplicação em desenvolvimento neste domínio específico. Em segundo lugar, para sua alcançarmos sua parametrização total precisamos preencher espaços vazios, chamados de hotspots [Pree95]. Para tal, devemos adicionar novas classes ou redefinir operações que podem levar a conflito de nomes devido à sobreposição semântica de especificações e a geração de herança múltipla, que pode impossibilitar o mapeamento automático para determinadas linguagens de programação orientada a objetos.

[1] É uma das definições de frameworks existentes. As outras serão abordadas posteriormente.

Abordagens como Cookbooks [Krasner88], Hooks [Froehlich97], Metapatterns [Pree95] e UML-F [Fontoura99] são tentativas de sistematizar a documentação e instanciação (reutilização) de frameworks, porém em sua maioria ou introduzem conceitos não utilizados de forma abrangente pelo mercado produtor de software ou não são automatizáveis por uma ferramenta computacional. Sendo assim se faz necessário o desenvolvimento de uma abordagem que represente os pontos de reutilização e/ou parametrização de um framework e permita uma assistência ao reutilizador durante o processo de reutilização.

1.3 Visão Geral da Abordagem Proposta

O processo tradicional de desenvolvimento de software [Sommerville00] [Pressman00] pode ser caracterizado por uma seqüência de atividades que capturam, representam e implementam um determinado conjunto de funcionalidades que resolvem um determinado problema. Estas atividades são em grande parte, executadas ciclicamente para possibilitar um refinamento dos modelos envolvidos no processo de forma gradativa. Como apresentado em [Jacobson97], uma vez que os sistemas de software desenvolvidos atualmente estão fortemente baseados no conceito de reutilização, as atividades inerentes a esta reutilização, devem sem integradas ao processo como um todo para permitir uma evolução gradativa dos modelos envolvidos.

Tendo isto em mente, um passo importante para se obter a sistematização do processo de reutilização, é a representação das atividades envolvidas na abordagem proposta (Figura 2). Inicialmente identificamos a presença de dois atores principais: o reutilizador [Biggerstaff89] e o projetista da reutilização. O primeiro desempenha o papel do *desenvolvimento baseado em reutilização* e tem como finalidade aprimorar a construção do software através da reutilização. O segundo desempenha o papel do *desenvolvimento para reutilização* e tem

como finalidade desenvolver o artefato reutilizável e toda a documentação de reutilização associada.

Analisando a figura 2 é possível perceber que o processo de reutilização presente na abordagem proposta pode ser caracterizado pelas atividades de busca[1], integração e adaptação. Estas atividades se mesclam com as atividades presentes no processo tradicional de análise de domínio, design e implementação respectivamente, para introduzirem o conceito de reutilização no processo como um todo.

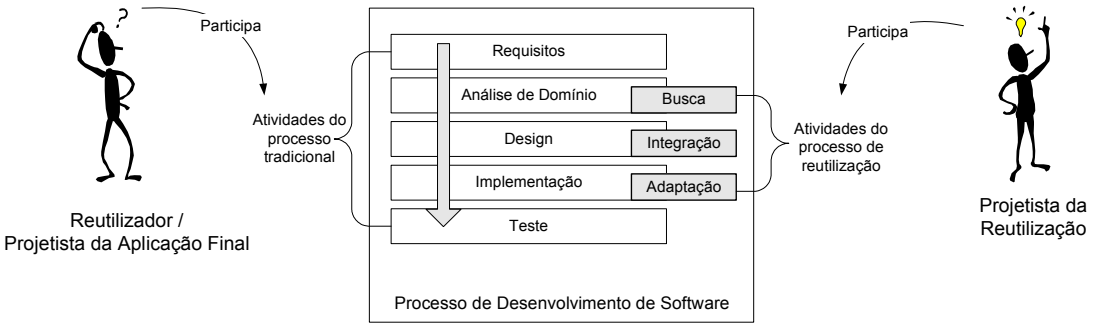


Figura 2 Fases do processo tradicional + Fases da reutilização.

Como dito anteriormente, a abordagem proposta se baseia no fato do processo de reutilização ser fortemente influenciado por um processo de composição de especificações. Sendo assim, como primeiro passo da abordagem, o reutilizador tem que esboçar uma especificação inicial da aplicação em desenvolvimento. Esta especificação tem como objetivo identificar o espaço de soluções que irá solucionar o problema em questão. Vale ressaltar que esta especificação pode ser feita utilizando qualquer técnica de analise de domínio como [Kang93] [Arango93] e não será abordado durante este trabalho.

[1] Não sistematizado pela abordagem.

Após este esboço inicial, o reutilizador busca por artefatos reutilizáveis em um repositório para verificar a existência de algum "compatível" com a aplicação em desenvolvimento. Uma vez que este artefato seja encontrado, o reutilizador inicia as atividades de integração e adaptação.

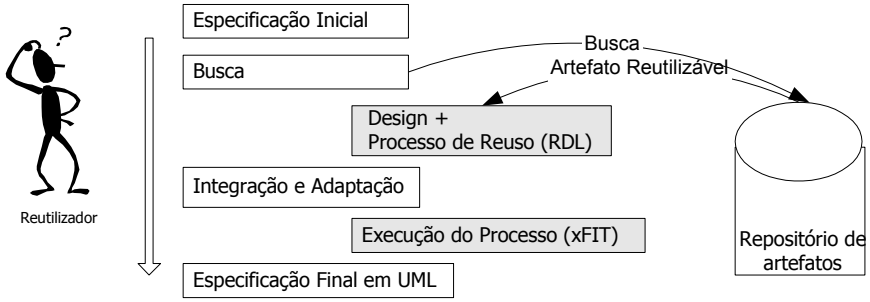


Figura 3 Processo de reutilização utilizado pela abordagem.

Um ponto importante para a abordagem proposta é relativo a estrutura interna do artefato reutilizável. Esta estrutura é composta por dois tipos de documentos; uma especificação em UML e um programa em RDL (o CookBook). A especificação em UML representa o design do artefato reutilizável através de diagramas de classes decorados com a extensão de UML-F [Fontoura99]. O programa em RDL especifica o processo de reutilização. Vale ressaltar que ambos os documentos são criados pelo projetista do artefato reutilizável.

Um ponto chave durante o processo de reutilização é a necessidade que o reutilizador tem de parametrizar o artefato reutilizável para prover a integração e adaptação necessárias. Independente do tipo de técnica utilizada pelo artefato reutilizável a execução das ações de reutilização e a gerência desta parametrização não podem ser feitas manualmente pelo reutilizador uma vez que as ações de reutilização envolvidas podem ser complexas e/ou

repetitivas. Sendo assim uma vez que o artefato reutilizável especifica o processo envolvido em sua reutilização, um programa (xFIT) executa esta especificação com o intuito de guiar o reutilizador neste processo, como um *wizard* para o processo de reutilização.

A definição deste programa de reutilização é obtida tomando como base à linguagem RDL que em uma análise simplista, permite o encadeamento de comandos usuais presentes na programação orientada a objetos. Estes comandos representam ações como redefinição de classes, uso de Design Patterns [Gamma95], etc., que são utilizadas na parametrização dos pontos de extensão [Jacobson97] de um artefato reutilizável.

Um fator determinante para o sucesso na sistematização do processo de reutilização é a sua simplicidade e interoperabilidade. Ambientes de reutilização como [Froehlich97] [Fontoura99] possuem uma representação particular de suas estruturas internas o que dificulta a integração destes, com ambientes de desenvolvimento consagrados pelo mercado. Tendo isto em mente a abordagem desenvolvida tem as seguintes características:

- *Compatibilidade* - O ambiente só utiliza (ou estende) padrões de mercado largamente utilizados como Linguagens de Programação Orientada a Objetos, XML, XMI e UML, para prover compatibilidade e/ou interoperabilidade com outros sistemas.
- *Legibilidade da Especificação Final* - Durante o desenvolvimento de software as fases de compilação e depuração são normalmente feitas em ambientes existentes no mercado, sendo assim o usuário final tem que entender a especificação final.
- *Orientado a Objetos* - O reutilizador deverá apenas saber técnicas utilizadas em desenvolvimento orientado a objetos.

- *Extensibilidade* – As ações de reutilização devem ser extensíveis para prover uma customização da reutilização como um todo.

Um ponto importante relativo às técnicas envolvidas na abordagem proposta é que estas fazem parte de um contexto maior que diz respeito ao processo de reutilização de artefatos de software [Oliveira00b] [Oliveira01] [Mathias01].

Do lado do projetista da reutilização, a abordagem facilita a tarefa de documentação do artefato através da extensão da Linguagem de Design de Frameworks (UML-F) proposta por [Fontoura99]. Esta extensão introduz conceitos de identidade, opcionalidade e condicionalidade aos tipos de pontos de extensão existentes em FDL. Um fator decisivo na escolha de FDL como mecanismo de documentação foi o seu estreito relacionamento com UML, que proporciona a utilização de conceitos consagrados no mercado de desenvolvimento de software, bem como possibilita o uso de ferramentas CASE como ArgoUML[Argo], Rational Rose[Rose] ou Together [Together]. O que falta para completar a abordagem sistemática de reutilização é obter uma forma manipulável do design do artefato. Isto foi alcança com a introdução do formato XMI (XML Metadata Interchange) que permite intercambiar especificações orientadas a objetos entre aplicações.

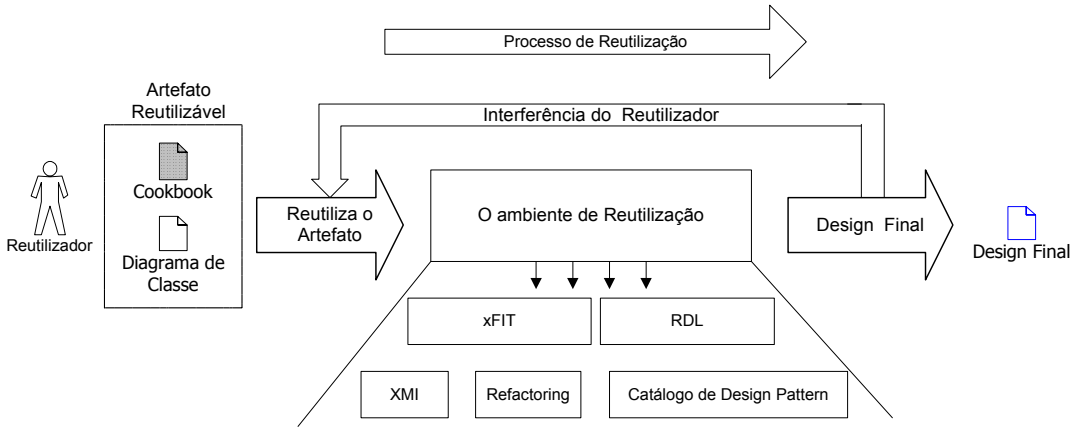


Figura 4 Visão geral do ambiente de reutilização.

Sendo assim a abordagem proposta para o processo de reutilização pode ser sintetizada (figura 4) em um ambiente que toma como entrada um artefato reutilizável composto por um cookbook escrito em RDL e um conjunto de classes representados em XMI. Este ambiente por sua vez executa este cookbook que comanda a manipulação da representação em XMI, para introduzir a parametrização necessária. Como pode ser observado pela seta "Interferência do Reutilizador" (Figura 4), o processo de reutilização sofre forte influencia do reutilizador, pois este necessita informar as características específicas de sua parametrização como nome de classes, escolha de opcionais e coisas afins. Vale ressaltar que boa parte desta parametrização esta representada na especificação inicial da aplicação em desenvolvimento uma vez que esta representação contém características como terminologia, estruturas e funcionalidades relativas ao domínio da aplicação.

Uma vez terminado o processo de reutilização, o ambiente gera um documento em XMI que pode ser re-introduzido nas ferramentas CASE citadas acima para uma eventual geração de código ou visualização do design final.

Um outro ponto abordado pelo ambiente é relativo a sobreposição semântica e sintática de entidades. É usual que a especificação inicial da aplicação use uma terminologia que de alguma forma já está representada dentro do artefato. Sendo assim, seria possível, por exemplo, obter um design final com classes com nomes iguais e semânticas diferentes, o que poderia levar a erros de interpretação (mesmo estando em módulos diferentes). Para evitar tal situação, o ambiente de reutilização utiliza o mecanismo de Refactoring [Fowler99] para renomear termos coincidentes. Esta abordagem permite que durante a geração do design final, comentários sejam introduzidos ao termo alterado para que o reutilizador possa rastrear sua origem na especificação inicial.

1.4 Contribuições

Como resultados obtidos deste trabalho podemos citar:

Resultados Principais

- A criação de UML-FI como uma extensão da linguagem de descrição de frameworks proposta por [Fontoura99]. Esta extensão introduz o suporte a regras para parametrização de atributos, a definição de hotspots[Pree95] como opcionais e mandatários e a especificação dos pontos de extensão de acordo com o tipo de extensão associada.
- Extensão do meta modelo de UML para incorporar graficamente as extensões mencionadas acima.
- Definição de uma linguagem de descrição de reuso a RDL(Reuse Description Language), que permite a especificação das atividades envolvidas durante a reutilização.
- Desenvolvimento de um ambiente o xFIT (XMI based Framework Instantiation Tool) que proporciona a execução da especificação de reutilização descrita em RDL de modo assistida.

Resultados Secundários

- Permite o rastreamento dos nomes presentes no design final que foram alterados devido a uma aplicação de Refactoring.
- Promove a aplicação sistemática de Design Patterns uma vez que inclui a descrição destes no cookbook em RDL.
- Permite a divisão do esforço de reutilização por entre grupos de desenvolvedores devido a possibilidade de definir assincronismo no cookbook.

1.5 Evolução do Trabalho

{referencias completas mais contribbui;ao de cada trabalho a tese}

O desenvolvimento do trabalho apresentado nesta tese se deu através do refinamento das técnicas de reutilização utilizadas durante o desenvolvimento de aplicações reais [Carvalho98] [Oliveira00a].

Originalmente utilizado como uma maneira para disseminar a reutilização de um framework de desenho desenvolvido no projeto 2GOOD [Carvalho98], esta abordagem foi expandida de modo a contemplar um escopo maior de aplicabilidade, incorporando características que sistematizam o processo de reutilização de frameworks tornando-o simples e apoiado por uma ferramenta.

Esta evolução pode ser verificada através dos trabalhos publicados [Oliveira00a] [Oliveira00b] [Oliveira01] [Mathias01] [Alencar01] onde foram representadas diversas contribuições para o produto final desta tese e cujos extratos estão descritos no apêndice A.

Na tabela abaixo encontra-se a relação de cada publicação e sua contribuição associada.

Tabela 1 Publicações X Contribuições

Publicação	**Contribuição para a Tese**
[Oliveira00a]	Relata a necessidade de se especificar de forma estruturada os pontos de adaptação/integração do framework de apoio a decisão DSSFrame. Esta representação, feita através de uma "BNF-like", permitia instanciar o DSSFrame de forma sistemática através do mapeamento de certos símbolos da BNF em classes da aplicação em desenvolvimento.
[Oliveira00b]	Relata a necessidade de se mapear a BNF reportada em [Oliveira00a] para um mecanismo presente em uma linguagem de design como UML. Neste momento foi adotada a representação UML-F presente em [Fontoura99] como ponto inicial desta representação.
[Oliveira01]	Relata a necessidade de representar de elementos opcionais que são encontrados em modelos oriundos da análise de domínio, bem como introduz XMI em substituição da BNF para representação dos pontos de adaptação/integração.
[Mathias01]	Relata a necessidade de representar os pontos de adaptação/integração a partir de uma especificação de alto nível.
[Alencar01]	Introduz a especificação de RDL como linguagem de representação do processo de instanciação bem como a extensão de UML-F para incorporar questões opcionais.

1.6 Estrutura da Tese

O capítulo 2 apresenta um resumo sobre as técnicas de reutilização utilizadas pela engenharia de software bem como os processos de reutilização que regulam a aplicação destas técnicas. No capítulo 3, são apresentados um conjunto de tecnologias relacionadas a abordagem proposta.

O capítulo 4 descreve em detalhes os problemas encontrados durante o processo de reutilização de frameworks orientados a objetos tomando como base o framework DTFrame e serve de motivação para os capítulos a seguir.

O capítulo 5 descreve a notação UMF-FI que captura os pontos de flexibilização/extensão presentes em um artefato de software reutilizável, bem como introduz esta notação ao metamodelo de UML.

No capítulo 6 pode ser encontrado a BNF comentada de RDL que tem como objetivo descrever a linguagem utilizada para especificar o processo de reutilização. É neste capítulo também que se encontra a especificação de execução dos comandos RDL.

No capítulo 7 é apresentada a ferramenta xFIT especialmente construída para apoiar a execução do processo de reutilização descrito por UML-FI e RDL. O capítulo 8 descreve a conclusão e os trabalhos futuros desta tese.

Por fim, o apêndice A traz um resumo dos artigos desenvolvidos para aprimorar a proposta apresentada por este tese.

2 Técnicas de Reutilização

2.1 Introdução

Desde o final dos anos 60 a reutilização é considerada como um ponto chave para a melhoria da qualidade e produtividade do desenvolvimento de software, pois permite a re-aplicação de conhecimento [Biggerstaf89] adquirido no desenvolvimento de aplicações futuras, evitando o trabalho recorrente.

Os benefícios trazidos por esta abordagem podem ser resumidos como:

- Redução do tempo total de desenvolvimento.
- Diminuição da taxa de aparecimento de erros.
- Desenvolvimento de uma linguagem comum.
- Diminuição do custo total de desenvolvimento.
- Formação de *experts* em domínios de aplicação específicos (especialistas em reutilização).
- Aumento da capacidade de produção de software.

Existem diversas definições para o termo reutilização. Originalmente proposto em 1968, durante a conferência sobre engenharia de software da OTAN, por Doug McIlroy, este termo foi inicialmente definido como a criação de uma biblioteca de componentes para serem simplesmente reutilizados. Esta proposição foi uma resposta ao problema intitulado "Crise do Software" definido por [Naur68], alegando a falta de "habilidade" demonstrada pela industria de software para desenvolver sistemas de software confiáveis, flexíveis e de baixo custo.

Historicamente, o desenvolvimento de artefatos reutilizáveis se tornou viável com a introdução do conceito de extensibilidade de funcionalidade em linguagens de programação [Salus98]. Este conceito permitiu que linhas de código fossem encapsuladas e rotuladas em unidades chamadas procedimentos ou funções para serem reutilizadas a posteriori, como em Fortran. Uma vez definidas estas unidades, tornou-se necessário seu armazenamento de forma persistente. Sendo assim, foi criado o conceito de bibliotecas, como pacotes de Ada [Ada], que permitem o armazenamento de um conjunto de operações, preferencialmente com semântica relacionada, em um arquivo que pode ser incorporado à aplicação em desenvolvimento.

Linguagens de programação orientadas a objetos aparecerem neste cenário como a ultima palavra em tecnologia incentivadora de reutilização, pois afirmam que o conceito de herança em conjunto com a reutilização de unidades encapsuladas e com semântica bem definidas denominadas classes aumentaria significativamente o reaproveitamento de conhecimento. Esta abordagem evoluiu no tempo para a reutilização de coleções de classes que podem estar relacionadas a um domínio específico denominada Frameworks [Pree95] [Fayad99a] ou a uma solução especifica denominada Design Patterns [Gamma95].

Componentes por sua vez têm um papel de destaque na história de reutilização, pois são exemplos de sucesso em outras disciplinas de engenharia (como a engenharia eletrônica), uma vez que permite a adoção do conceito de "caixa-preta", que possibilita ao desenvolvedor um maior nível de abstração e independência.

Tecnologias como as presentes em Programação Orientada a Concerns [Tarr00] e Concerns Multidimensionais[Tarr00], também são consideras incentivadoras de reutilização, pois

permitem que uma especificação base, normalmente orientada a objetos, seja modificada com (AspectJ [Kiczales96]) ou sem (HyperJ[Tarr00]) a necessidade do código fonte. Esta modificação se dá através da alteração desta especificação base a nível estrutural e/ou funcional de modo a se obter a funcionalidade desejada.

Além das tecnologias de reutilização ligadas a mecanismos de linguagens de programação, existem também as técnicas baseadas em Sistemas Gerativos. Estas técnicas têm como objetivo utilizar programas que geram programas, a partir de uma especificação inicial. Dentre as técnicas de sistemas gerativos temos [Biggerstaff89]: Sistemas Baseados em Linguagens, Geradores de Aplicações e Sistemas baseados em transformação.

- **Sistemas Baseados em Linguagens** – São sistemas onde uma linguagem de especificação é definida para representar um domínio específico de aplicações, elevando o nível de representação de um programa escrito nesta linguagem. Isto pode ser obtido através do ocultamento dos detalhes de implementação que viabilizam e execução deste programa em uma máquina comum. Ferramentas geradoras de compiladores como LEX/YACC[Lex] ou FLEX/BISON[Flex] são exemplos deste tipo de sistema.
- **Geradores de Aplicações** – São sistemas onde a aplicação final é gerada a partir de um padrão, definido internamente pelo gerador. Ambientes de programação como, [Staa] e *Gui Builders*, com o conceito de modelos e/ou *wizards*, são exemplos destes sistemas.
- **Sistemas baseados em transformações** – São sistemas onde a aplicação final é gerada tomando-se como entrada uma especificação e um conjunto de regras de transformação da especificação alvo na especificação destino (a aplicação final).

Ambientes como TXL [Txl] e Draco [Neighbors80] [Leite94] são exemplos destes sistemas.

Embora pareça uma idéia simples, o processo envolvido durante a aplicação da tecnologia de reutilização encontra diversos obstáculos [Jacobson97] que exigem uma abordagem sistemática para trazer de forma eficaz os benefícios a ela associados. Nas sub-seções a seguir serão detalhadas as tecnologias de reutilização que mais se relacionam com a abordagem apresentada, indicando vantagens e desvantagens de sua aplicação. Inicialmente será apresentado Frameworks, uma vez que este é o foco principal deste trabalho.

2.2 Framework

2.2.1 Introdução

Como foi apresentada anteriormente, a reutilização de software teve seu início com a introdução de bibliotecas de funções e/ou procedimentos e em seguida, com o advento da programação orientada a objetos, biblioteca de classes. Um ponto chave no uso de bibliotecas como técnica de reutilização é o seu foco na reutilização de código, uma vez que estas bibliotecas precisam ser ligadas em tempo de compilação ou execução à aplicação em desenvolvimento. A reutilização de código é um tanto restritiva, pois no processo de implementação de uma determinada abstração em uma linguagem de programação específica, as idéias originais (a abstração) são normalmente intercaladas e escondidas pelo idioma da linguagem utilizada. Isto não permite que todo ou parte do conhecimento adquirido durante o processo de desenvolvimento do artefato reutilizável seja reaproveitado em situações diversas. A menos que o domínio da aplicação seja bem conhecido, como o caso das interfaces gráficas ou aplicações matemáticas, o alto custo de desenvolvimento de

tais bibliotecas, bem como a baixa probabilidade desta biblioteca ser útil, desencorajam seu desenvolvimento[Jacobson97].

Ainda que a reutilização de código tenha se mostrado restritiva, as idéias que regulam este código podem ser úteis uma vez que estas independem de implementação[1]. Sendo assim surgiu uma nova técnica de reutilização que tem como objetivo capturar e representar as idéias gerais (a abstração) que estão por traz de um artefato de software. Está técnica, denominada reutilização de design, mostrou-se bastante promissora por pelo menos três fatores como apresentado em [Garlan96].

⇒ Melhoria no entendimento do sistema em geral, através do uso de uma representação de alto nível (próximo à abstração), não vinculada a um idioma (próximo à implementação) e sim a um estilo de representação de conhecimento (OO, por exemplo)

⇒ Melhoria na captura de erros de projeto, uma vez que a reutilização é decidida em fases iniciais do desenvolvimento.

⇒ Induz à reutilização de código, uma vez que este design necessita de uma representação "executável" que comprove sua utilidade.

2.2.2 Definição

Frameworks são em primeira análise uma abordagem para se obter reutilização de design e código (figura 5). Existem diversas definições para o termo framework. Embora não utilizando o termo Framework, Parnas [Parnas76], enfatiza de forma pioneira as vantagens

[1] Muito similar à idéia de pseudo-códido.

da reutilização de design, observando a existência de famílias de programas que permitem construção de novas aplicações a partir de um design pré-existente.

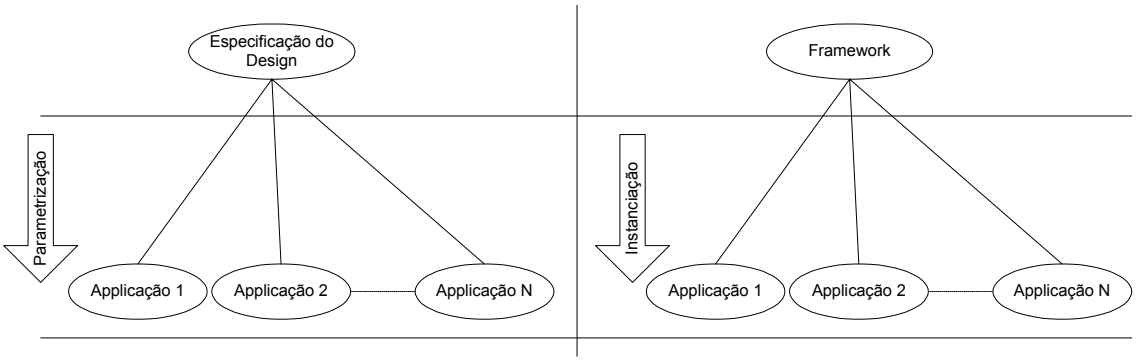


Figura 5 Frameworks como ferramenta de reutilização de design.

Em dois de seus vários artigos sobre reutilização, Johnson [Johnson88] [Johnson97] afirmou que "um framework é um esqueleto de uma aplicação que deve ser parametrizado pelo desenvolvedor da aplicação" e "um framework é um conjunto de classes que representa um design abstrato para soluções em uma família de aplicações". Podemos notar que a primeira afirmação está realçando utilidade de um framework, isto é, ser parametrizado através do processo de reutilização. A segunda afirmação trata mais o aspecto estrutural (classes) do framework bem como o direcionamento para um domínio específico.

Em seu livro sobre desenvolvimento de software, Pree [Pree95], ressalta que frameworks são "constituídos por pedaços de software semi-acabados e prontos para usar…, sendo que reutilizar um framework significa adaptar estes pedaços para uma necessidade específica, através da redefinição de métodos e algumas classes". A grande contribuição trazida por este trabalho é sem dúvida a definição do termo hot-spot, que permite identificar de forma clara os pontos de flexibilização de um framework (figura 6), que servirão de base para o processo de reutilização.

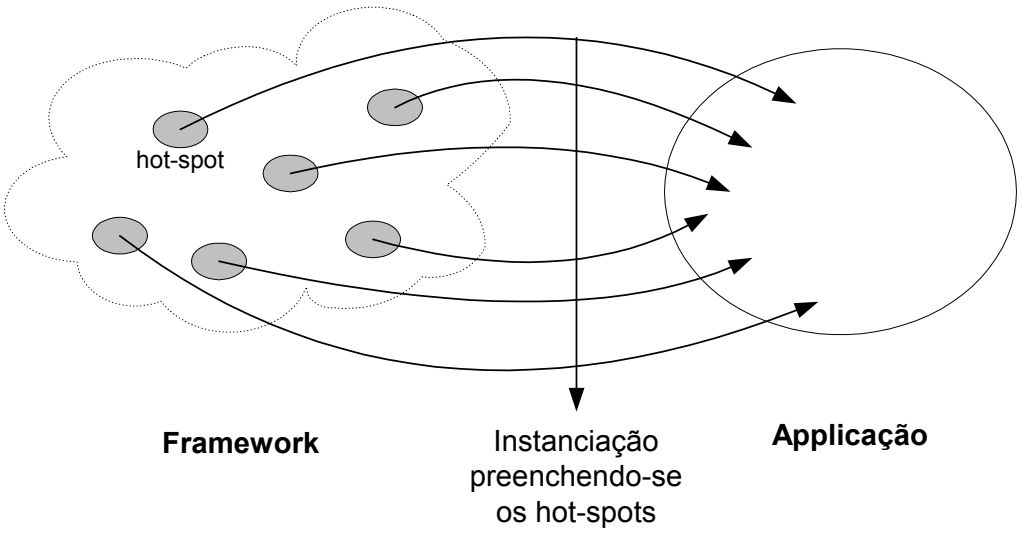


Figura 6 Hot-Spots como mecanismos de extensão de frameworks.

Em [Fontoura99] foi afirmado que "um framework é definido como um sistema composto de um subsistema núcleo, que é comum a todas as aplicações que são instanciadas a partir deste framework, e um subsistema de hot-spots que o implementa o comportamento especifico das aplicações instanciadas. Sendo assim, um desenvolvedor de uma aplicação gera uma instância do framework adequando o subsistema de hot-spots durante o processo de instanciação". Claramente esta definição está baseada nos conceitos introduzidos em [Pree95] e enfatiza a utilidade de frameworks proposta em [Johnson88].

Um dos trabalhos mais completos na área de frameworks foi elaborado por Fayad [Fayad99a] [Fayad99b] [Fayad99c]. Este trabalho é composto de uma coletânea de artigos organizados em três livros que relatam a obtenção, desenvolvimento, documentação, evolução e experiências na área. Fayad explora o termo frameworks de aplicações onde "… frameworks de aplicação orientados a objetos são uma tecnologia promissora para

materializar/reificar projetos e implementações de softwares comprovados, levando a redução de custo e ao aumento a qualidade do software". O interessante nesta abordagem é que ela abrange de forma sucinta os conceitos mais importantes por traz da tecnologia de frameworks: reutilização, design e orientação a objetos. Durante o curso deste trabalho adotaremos esta definição de frameworks, uma vez que a sistematização proposta não está restrita a um domínio especifico e sim ao estilo de representação do conhecimento reutilizável (OO).

2.2.3 Vantagens & Desvantagens

Independente da definição adotada é possível classificar os benefícios essenciais de frameworks como [Fayad99a]: modularidade, reusabilidade, extensibilidade e inversão de controle, sendo esta última uma das principais.

- **Modularidade** - Frameworks melhoram a modularidade de um design através do encapsulamento de detalhes de implementação por trás de interfaces estáveis. Esta modularidade torna possível incrementar a qualidade do software, uma vez que os impactos causados por alterações de design e implementação são localizados, reduzindo o esforço necessário para o entendimento e manutenção do software existente.
- **Reusabilidade** – As interfaces estáveis de um framework incentivam reusabilidade uma vez que definem um componente genérico que pode ser re-aplicado para criar novas aplicações. Esta reusabilidade carrega o conhecimento em um domínio e o esforço anterior de desenvolvedores experientes, para evitar re-criação e re-validação de soluções comuns presentes em requisitos de aplicações recorrentes.
- **Extensibilidade** – Frameworks aprimoram extensibilidade através da definição de pontos de extensão que permitem a uma aplicação estender suas interfaces estáveis.

Estes pontos de extensão permitem o desacoplamento sistemático da parte fixa do framework, presente no domínio da aplicação, da parte variável introduzida pelo processo de instanciação.

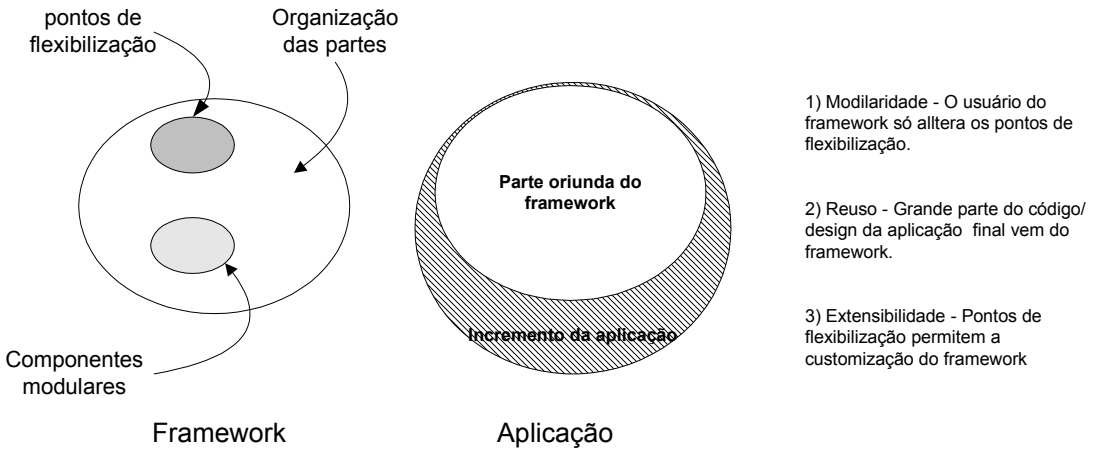


Figura 7 Instanciação do Framework

- **Inversão de Controle** - Uma novidade trazida pela reutilização de frameworks é a possibilidade da inversão do fluxo de controle (Figura 8), isto é, quem comanda o fluxo de execução principal do programa é o artefato reutilizável e não o artefato reutilizador. Este conceito permite que uma aplicação especifique seu funcionamento como um todo, principalmente no que se refere à coordenação de seus componentes principais. Em abordagens convencionais como as encontradas na reutilização de bibliotecas e componentes o artefato reutilizável é passivo[1], o que torna seu desenvolvimento muito mais complexo uma vez que o contexto em que este artefato será inserido é totalmente desconhecido pelo projetista.

[1] Embora exista a possibilidade de funções callback que como em Delphi podem se tipadas, beneficiando-se dos mecanismos de compilação.

A inversão de controle está intimamente ligada aos mecanismos de extensão presentes em linguagens orientadas a objetos nos quais frameworks se baseiam. Estes mecanismos, como polimorfismo e late-binding, permitem que objetos "executem" um código a ser definido futuramente pelo desenvolvedor da aplicação, durante o processo de instanciação. Esta execução se dá através de um protocolo bem definido, normalmente especificado pelo mecanismo de herança. É esta inversão de controle que, em última análise, possibilita a criação dos esqueletos de aplicação mencionados por Johnson [Johnson88].

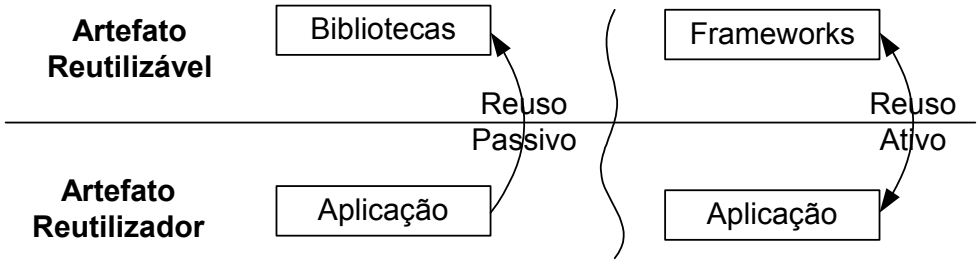


Figura 8 Inversão de Controle existente na aplicação de um Framework.

Embora a reutilização de frameworks pareça bastante vantajosa, a aplicação desta tecnologia de forma prática traz alguns problemas como: esforço de desenvolvimento, curva de aprendizado, integrabilidade, manutenibilidade e eficiência.

- **Esforço de Desenvolvimento** – Uma vez que o desenvolvimento de sistemas complexos é difícil, o desenvolvimento destes sistemas de forma abstrata tendo em mente reutilização é mais difícil ainda. É necessário além de tempo, o auxílio de expertise no domínio para o qual frameworks está sendo desenvolvido bem como uma boa dose de criatividade.

- **Curva de Aprendizado** – Um problema comum no processo de reutilização é o tempo necessário para se tornar capacitado para obter vantagens desta reutilização. Quando tratamos de frameworks isto não é diferente. Foi constatado em [Fayad99a] que são necessários de 6 a 12 meses para um desenvolvedor se tornar produtivo na utilização de frameworks para interfaces gráficas como MFC [Mfc]. Sendo assim a menos que o custo de aprendizagem seja amortizado por vários projetos ou que o ganho de produtividade e qualidade sejam expressivos, este investimento não se tornará atraente.
- **Integrabilidade** – A maioria dos frameworks são desenvolvidos exclusivamente para o propósito de extensão e não integração com outros artefatos de software. Sendo assim problemas difíceis de serem solucionados, como, por exemplo, quem comanda do fluxo de controle da aplicação final, podem surgir e dificultar este processo de integração. Este tipo de problema é muito comum quando se tenta integrar frameworks[Mattsson00][Garlan96].
- **Manutenibilidade** – Como todo artefato de software, seus requisitos iniciais evoluem no tempo, obrigando a criação de novas versões do framework. Sendo assim, as aplicações instanciadas a partir de um dado framework também devem evoluir com a finalidade de se manterem de acordo com a especificação do framework. Este problema está relacionado com a modificação de aplicações que já estão em produção o que pode ser problemático quando, por exemplo, o serviço prestado por estas aplicações não puder parar.
- **Eficiência** – Frameworks promovem extensibilidade empregando níveis de indireção adicionais através de mecanismos como polimorfismo e late-binding, presentes em linguagens orientadas a objetos. Esta indireção usualmente provoca queda na eficiência do código final, uma vez que chamadas adicionais a tabelas virtuais de métodos serão necessárias para executar uma determinada tarefa. Sistemas com restrições de tempo,

como telecomunicações, precisam de ferramentas adicionais para gerar/transformar o código instanciado a partir do framework, de forma eficiente na plataforma alvo [Carvalho98].

2.2.4 Classificação

Frameworks podem ser classificados de acordo com seu escopo ou forma de extensão. Quanto ao escopo, é observado a camada na qual o framework está atuando e pode ser de infraestrutura, integração ou relacionados a um domínio específico de atividades [Fayad99a].

- **Infraestrutura** – Frameworks de infraestrutura simplificam o desenvolvimento de sistemas portáteis e eficientes como sistemas operacionais, comunicações e interface com o usuário. São normalmente utilizados internamente em uma organização e não são vendidos a clientes.
- **Integração** – São comumente utilizados para integrar sistemas distribuídos permitindo a troca de dados entre sistemas heterogêneos. Sistemas compatíveis com o modelo ORB (Object Request Broker), são exemplos destes frameworks.
- **Domínio Específico** – Estes frameworks são direcionados para amplos domínios de aplicação como telecomunicações, manufatura e finanças. Em geral são sistemas complexos que envolvem infraestrutura e integração tornando-os dispendiosos para serem desenvolvidos.

Quanto à forma de extensão, é observada a técnica utilizada para estender um dado framework podendo ser: whitebox frameworks, graybox frameworks e blackbox frameworks.

- **Whitebox** – Whitebox frameworks são fortemente baseados nas características de linguagens de programação orientadas a objetos como herança e late-binding, para

expressarem/implementarem pontos de flexibilização. Uma vez que redefinição é a palavra chave para o processo de instanciação, o reutilizador precisa ter profundo conhecimento de sua estrutura, bem como das colaborações dos objetos envolvidos. São comumente utilizados em conjunto com bibliotecas de componentes para facilitar o processo de instanciação.

- **Blackbox** - Blackbox frameworks baseiam seu mecanismo de extensão através da composição de objetos. Isto se dá através da definição de interfaces que serão utilizadas para permitir o acoplamento de componentes externos. São mais fáceis de usar que Whitebox frameworks uma vez que o reutilizador não precisa conhecer as entranhas do artefato reutilizável. Frameworks com estas características são comumente chamados de Componentes [Mattson2000].
- **Graybox** – São projetados para evitar as desvantagens presentes em whitebox frameworks e blackbox frameworks, permitindo um certo grau de extensibilidade sem a necessidade de se expor informações internas. Frameworks visuais como o Borland VCL são exemplos, pois permitem instanciação a partir da ligação de componentes e ainda provém parametrização via herança.

2.2.5 Desenvolvimento

O desenvolvimento de frameworks pode ser alcançado de duas formas. A primeira é muito similar a qualquer desenvolvimento de software reutilizável. Começa com uma extensa análise de domínio [Arango93] [Prieto94] [Kang93], onde entre outras coisas, exemplos para instancias são coletados e analisados para se obter partes comuns e partes variáveis. Esta forma de desenvolvimento requer uma grande capacidade de gerenciamento e classificação das informações coletadas, uma vez que as fontes utilizadas para se obter os exemplos citados anteriormente, são em sua maioria, não estruturas, como livros e experts no

domínio. Um outro problema observado nesta abordagem é o custo de desenvolvimento de tal artefato. Este pode ser proibitivo, pois a primeira aplicação instanciada é meramente feita para validar o framework e não tem propósitos comerciais.

A segunda forma de se obter framework está baseada no conhecimento adquirido pelos desenvolvedores durante o processo de desenvolvimento de três ou mais aplicações em mesmo domínio. Uma vez existente, este conhecimento pode ser organizado de modo a obter um design genérico para as N aplicações desenvolvidas, permitindo classificação e diferenciação entre partes comuns e específicas. A vantagem desta abordagem é a pulverização dos recursos necessários para se desenvolver um framework uma vez que uma pequena parte de análise de domínio é feita em cada aplicação. Em contra partida, dada a inexistência de uma análise de domínio sistemática como a presente na forma anterior, caso as aplicações em questão contenham vícios de projeto, estes serão repassados para todas as instancias do framework.

Independente da forma de desenvolvimento um dos fatores mais importantes em um framework é sua usabilidade. Este fator está diretamente ligado a fase de desenvolvimento, que define o escopo e grau de abstração presentes no framework. Quanto mais abstrato e abrangente for o framework mais difícil se torna seu uso e desenvolvimento.

Pode ser observado na figura 9 que frameworks com poucas classes e poucos pontos de extensão tendem ser facilmente utilizados. O seu desenvolvimento, na maioria das vezes, não requer uma grande quantidade de recursos e normalmente são desenvolvidos sem uma extensa fase de análise de domínio. Componentes visuais são bons exemplos disto.

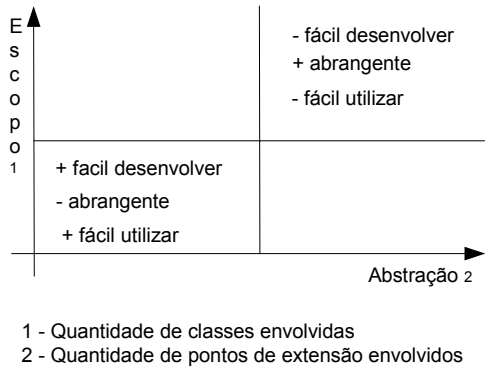


Figura 9 Escopo X Abstração

Quando aumentamos o número de classes e particularmente os pontos de extensão, os frameworks tendem a aumentar o escopo de aplicabilidade dentro de um domínio de aplicação. Em contrapartida, sua utilização requer uma longa curva de aprendizado e seu desenvolvimento está normalmente baseado em uma extensa fase de análise de domínio. Frameworks desenvolvidos para o domínio de telecomunicações são exemplos deste fato [Mattson00].

2.2.6 Instanciação

O processo de reutilização de frameworks é comumente chamado de processo de instanciação. É durante este processo que os pontos de extensão existentes no framework são preenchidos para se obter a aplicação final. Como dito anteriormente, este processo está fortemente baseado na classificação do framework e pode requerer um maior ou menor conhecimento do artefato reutilizável.

O que pode ser observado durante a instanciação, é que o processo tradicional de desenvolvimento de uma aplicação [Somerville00] [Pressman00] [Rup] deve ser alterado de modo a incorporar nesta aplicação as características impostas pelo framework. Este processo começa de forma similar ao desenvolvimento de aplicações comuns com uma fase de requisitos [Sommerville00], no qual os requisitos funcionais e não funcionais são coletados e expressos em uma notação específica. Em seguida o domínio da aplicação é investigado para se produzir um modelo conceitual do problema em questão. Normalmente, é durante esta fase que frameworks são apresentados como soluções para o problema. Uma vez escolhido o framework, inicia-se o processo de instanciação que tem como objetivo, no caso de frameworks Whitebox, integrar/adaptar (figura 10) o modelo conceitual da aplicação com o modelo de classes presente no framework. Em seguida temos os passos tradicionais de codificação e teste.

A "integração[1]" de designs, presente na instanciação está fortemente baseada na documentação do framework e é normalmente executada de forma intuitiva. O reutilizador tem que conhecer os ponto de extensão e a forma usada para entende-los. Abordagens como UML-F [Fontoura99], CookBook [Krasner88] e Hooks [Froelich96] auxiliam este processo permitindo que o projetista do framework especifique como o framework deve ser instanciado. Estas abordagens serão apresentadas na seção subseqüente que trata documentação de frameworks.

[1] Esta composição se dá no nível das especificações, não se tratando de uma fusão de designs propriamente dita.

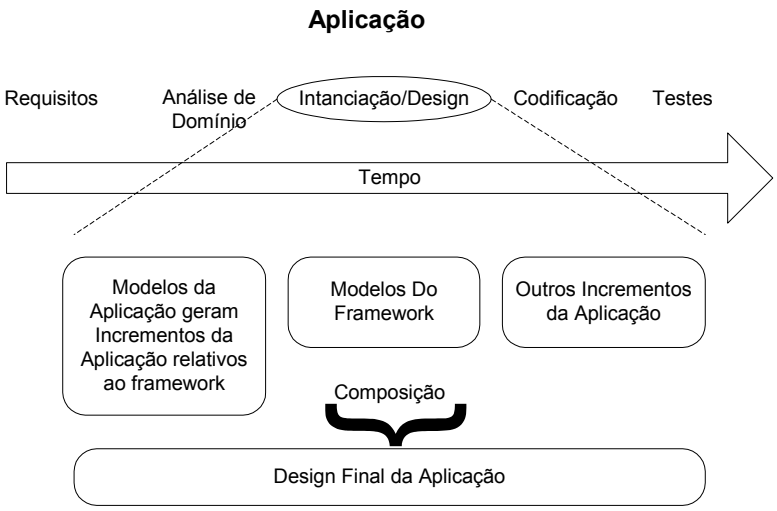


Figura 10 Processo de Instanciação.

2.2.7 Documentação

Uma das tarefas mais áridas e difíceis de ser realizada durante o processo de desenvolvimento de software é sem sombra de dúvidas a documentação. Produzir uma documentação que possibilite, dentre outras coisas, a substituição do desenvolvedor original para uma futura manutenção, requer a capacidade de catalogar e estruturar todos os dados necessários para a compreensão do design/código em questão. No caso de uma aplicação acabada, ferramentas como Browsers e Depuradores auxiliam no processo entendimento, pois permitem a navegação pelas estruturas da aplicação.

Quando falamos de artefatos reutilizáveis, o problema da documentação é ainda mais grave, uma vez que além do usuário da documentação intitulado mantenedor/desenvolvedor, existe também o usuário reutilizador que normalmente não é o desenvolvedor original e conseqüentemente não é um *expert* no domínio da aplicação. Sendo assim, a documentação

do framework precisa apresentar de forma clara, não só suas estruturas internas mas como sua (re) utilização deverá proceder. Em [Johnson92] é afirmado que a documentação de um framework deve abranger:

- **Propósito do framework** – Descrição informal da utilidade do framework em questão. Pode ser expressa em varias mídias, mas normalmente são utilizados textos não estruturados. No caso de blackbox frameworks descreve sua interface. Serve para todos os usuários do framework. Normalmente consultada na fase de análise de domínio para a verificação da reutilizabilidade no contexto em análise.
- **Design do framework** – A descrição das classes, objetos, componentes e módulos presentes no artefato, bem como suas interações. No caso de whitebox frameworks serve para todos os usuários. Normalmente consultada na fase de reutilização/integração.
- **Propósito das aplicações exemplo** – Descrição do propósito e funcionamento das aplicações exemplo. Serve para o reutilizador conhecer o *look and feel* do framework e às vezes é o ponto de partida para o desenvolvimento da nova aplicação. Pode ser utilizada nas fases de domínio e implementação. Na primeira para uma verificação do potencial de reutilização. Na segunda como um guia para a reutilização.
- **Como usar o framework em questão** – Descreve como o reutilizador deve proceder durante o processo de instanciação. Diversas tentativas para se estruturar esta descrição foram propostas e serão descritas a seguir.

Sendo assim, a documentação de um framework não pode conter apenas diagramas de classes descrevendo de forma detalhada o design do framework, mesmo que certos usuários necessitem desta informação. Além disto, uma documentação que descreva sua utilidade de

forma mais abrangente e superficial integrada a uma especificação do fluxo de atividades envolvidas no processo de instanciação são de fundamental importância irão ajudar ao reutilizador.

Cookbook foi originalmente proposto em [Krasner88] como um tutorial para a utilização do framework *Model-View-Controller* presente na biblioteca de classes de Smalltalk [Stroustrup97]. Primeiramente esta forma de documentação, descreve o framework de forma geral, através de linguagem natural, e em seguida descreve as partes relevantes à instanciação. Por último apresenta uma série de exemplos que utilizam o framework.

Motifs [Lajoie96] e Patterns [Johnson92] propõem que a experiência e o conhecimento dos desenvolvedores de como o framework deve ser utilizado pode ser capturado com uma série de padrões sendo que cada padrão deve ter:

- **Nome**: Nome que identifica o padrão.
- **Problema**: Descreve o problema a ser solucionado.
- **Solução**: Descreve a solução proposta, bem como faz comentários sobre ela.
- **Exemplo**: Descreve um exemplo de instanciação do hot-spot.
- **Resumo da Solução**: Descreve a solução de redefinição de forma sucinta.
- **Padrões Relacionados**: Lista alguns padrões que se relacionam com este.

É possível caracterizar pelo menos dois problemas com as abordagens anteriores. Primeiro é a falta de estruturação das descrições presentes, pois estas são extensamente baseadas em linguagem natural, levando a erros de interpretação. Em segundo lugar, é a falta de ligação

entre o design presente no framework e estas descrições devido ao fato de serem feitas como uma documentação auxiliar.

Hooks [Froelich97] são considerados um aprimoramento de cookbooks uma vez que descrevem pontos de flexibilização de forma mais estruturada, detalhando aspectos relevantes como participantes no hook e alterações necessárias para usar o hook. Em sua estrutura, hooks descrevem:

- **Nome** – Nome único no contexto do framework, dado para cada hook.
- **Requisito** – O problema que o hook soluciona.
- **Tipo** – Um par ordenado que especifica o método de adaptação e apoio fornecido pelo framework.
- **Área** – As partes do frameworks que são afetadas pelo hook.
- **Usa** – Outros hooks necessários para se usar este hook.
- **Participantes** – Os componentes que participam deste hook, existentes ou criados pelo reutilizador.
- **Mudanças** – Seção principal do hook e descreve as alterações que devem ser feitas nos componentes que participam do hooks. É aqui que se colocam as alterações como herança e redefinições de métodos.
- **Restrições** – Indicam as restrições impostas ao uso do hook.
- **Comentários** – Descrições adicionais necessárias.

Embora seja uma forma muito mais precisa de se documentar um framework, Hooks carecem de informações necessárias para se automatizar o processo de instanciação. Para tal seria necessário obter uma descrição manipulável de ambos design e hooks com a finalidade de ajudar o reutilizador a especificar os refinamentos/parametrizações

necessários. Com a formatação presente na seção de alterações seria imprescindível a criação de uma ferramenta capaz de analisar esta seção e de alguma forma combina-la com as classes presentes no design.

MetaPatterns [Pree95] por sua vez, usam meta abstrações para descrever formas flexíveis de conexões entre classes. Cada metapattern identifica um relacionamento entre métodos template que são definidos no framework e métodos hook que são deixados em aberto para serem preenchidos pelo desenvolvedor da aplicação. A classe que contem o método template é chamada de classe template e a classe que contem o método hook é chamada de classe hook. Aqui a documentação atingida é mais ao nível de design permitindo a identificação do propósito por traz do ponto de flexibilização.

Em [Fontoura99] foi proposto uma linguagem como extensão de UML para capturar os pontos de flexibilização no nível de design. Esta linguagem permite expressar pontos de flexibilização como métodos de variação, classes de extensão e interfaces. Um fator importante desta abordagem é o mapeamento dos pontos de flexibilização existentes em possíveis formas de instanciação, que permitem uma rápida execução pelo desenvolvedor da aplicação. De forma adicional este trabalho também apresenta uma série de ferramentas baseadas em Prolog[Prolog], que auxiliam o usuário do framework no processo de instanciação.

Embora existam outras técnicas de documentação de Frameworks como Exemplars [Gangopadhyay95] e Contracts [Helm90], nenhuma delas conseguiu associar a usabilidade e eficácia uma vez que são normalmente baseadas em ferramentas que não acoplam as partes envolvidas no processo de entendimento/instanciação. Para tal seria necessário que, por

exemplo, Hooks fossem feitos durante a construção do diagrama de classes em uma ferramenta case específica, ou que a abordagem presente em [Fontoura99] fosse integrada a uma forma padrão de representação.

2.3 Componentes

Mesmo sendo considerados como Frameworks BlackBox, componentes tem um papel de destaque na industria de reutilização e devem ser tratados a parte. Embora proposto desde 1968 por Doug McIlroy, a primeira abordagem de sucesso para a tecnologia de componentes veio quase 25 anos depois, com a definição e introdução em 1992 do Visual Basic eXtension, ou VBX, para o ambiente de programação em Visual Basic da Microsoft. Com isto foi possível criar, de forma rápida, um mercado de desenvolvedores e reutilizadores que naquele momento necessitavam de extensas e complicadas bibliotecas para desenvolverem aplicações para o ambiente Windows 3.X. A partir deste momento vários ambientes de desenvolvimento para Windows foram criados tomando como base ou incorporando este padrão como, por exemplo, o VisualTool [Tool] e o Borland Delphi [Delphi].

Na mesma direção apareceram os padrões CORBA [Orfali96] da OMG, SOM [Orfali96] da IBM e COM [Orfali96] da Microsoft que seriam padrões de objetos distribuídos sendo que o primeiro seria uma especificação independente de fabricante.

A despeito de padronização, componentes tem uma definição aceita pela comunidade de desenvolvedores e reutilizadores que pode ser descrita como[Szyperski98] (Figura 11):

Um componente é uma unidade encapsulada que pode ser composta com outras partes de um sistema maior através de uma interface bem definida. Esta composição se dará através

da parametrização desta interface para customizar o componente de acordo com as necessidades da aplicação.

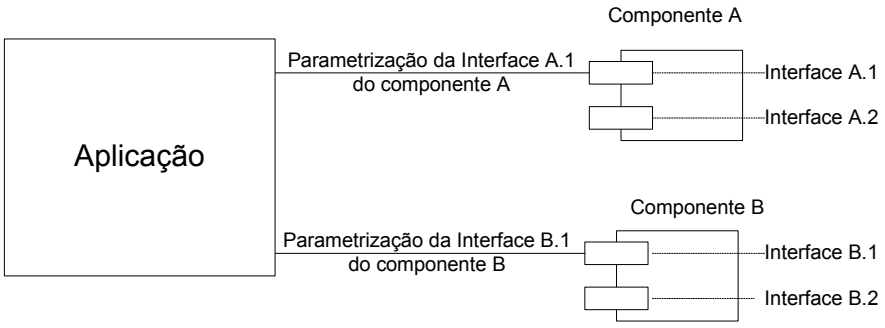


Figura 11 Utilização de um componente.

Desta definição podemos ressaltar que o processo de reutilização de um componente pode ser resumido pela palavra parametrização. Como em frameworks whitebox, para o reutilizador alcançar esta parametrização, é necessário um total conhecimento das regras e/ou restrições presentes na interface. Os padrões de definição de componentes apresentados não publicam estas restrições de maneira sistemática deixando para o projetista/desenvolvedor do componente o trabalho de checar se as customizações do componente são coerentes. Um exemplo deste problema pode ser ilustrado através do uso de um componente visual que tem na interface a possibilidade de usar a mesma fonte definida pelo controle mais externo, o que é mais usual, ou definir a própria fonte. Obviamente estas situações são contraditórias e podem levar a interpretações inconsistentes da customização, uma vez que a escolha da fonte do componente depende de qual interface foi usada por último.

Situações como a descrita acima, poderiam ser evitadas através do uso de regras de reutilização. Estas regras poderiam ser analisadas pelo ambiente de reutilização, o qual guiaria e/ou informaria ao reutilizador o aparecimento de inconsistências. O ambiente desenvolvido neste trabalho tem como um de seus objetivos permitir a definição de scripts de reutilização com estas características.

2.4 Design Patterns

Embora não sendo uma técnica de reutilização de código, Design Patterns despontou como uma das maiores contribuições na área de reutilização, pois permitem a reutilização do conhecimento de forma organizada e independente do domínio da aplicação. Originalmente proposto por Erich Gamma em sua tese de doutorado, este desenvolveu seu trabalho esta baseado nas afirmações do arquiteto Chirstopher Alexander que diz que "cada padrão descreve um problema no nosso ambiente e o núcleo da sua solução, de tal forma que você possa usar esta solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira" [Alexander77]. Muito embora Alexander estivesse falando de padrões em arquitetura, Gamma observou que estes conceitos se aplicavam de forma precisa em relação padrões de projeto orientados a objetos. Sendo assim em sua tese, Gamma definiu que "Design Patterns são descrições de objetos e classes comunicantes que são customizados para resolver um problema geral de projeto num contexto particular".

Junto com esta definição, Gamma também desenvolveu um gabarito para que Design Patterns pudessem ser documentados de maneira uniforme. Neste gabarito um padrão é composto por:

- **Nome -** Nome do padrão que expressa sua idéia de forma sucinta.

- **Intenção -** O que faz o padrão?
- **Também conhecido como -** outros nomes.
- **Motivação -** Um cenário que ilustra um problema de projeto e como o padrão soluciona este problema.
- **Aplicabilidade -** Exemplos de problemas que o padrão pode tratar?
- **Estrutura -** Representação gráfica das classes do padrão usando OMT [Rumbaugh91].
- **Participantes -** Lista de classe e objetos que participam do padrão.
- **Colaborações -** Como os participantes colaboram para executar suas responsabilidades.
- **Conseqüências -** Quais são os custos e benefícios de sua utilização?
- **Implementação -** Dicas de implementação.
- **Exemplo de Código -** Fragmentos de código que ilustram a implementação.
- **Usos Conhecidos -** Usos em sistemas reais.
- **Padrões Relacionados -** Padrões que se relacionam com padrão o em questão.

Como pode ser observada, a documentação de um padrão é bastante extensa cobrindo diversas facetas de usabilidade. Do lado do projetista existem seções que indicam a motivação e aplicabilidade do padrão. Do lado do implementador existem seções com exemplos e dicas, bem com diagramas de classes que facilitam o desenvolvimento do código a ser escrito.

Mesmo com toda esta documentação a aplicação de padrões ainda tem que ultrapassar pelo menos duas barreiras. Como em frameworks, a aplicabilidade de um padrão para solucionar um determinado problema não pode ser feita de forma sistemática uma vez que ambos não estão descritos segundo uma estrutura comum. Em segundo lugar, também igual a frameworks, diz respeito ao processo de reutilização em si, ou seja, qual a seqüência de

operações necessárias para implementar um padrão. É tarefa única e exclusivamente do reutilizador verificar se todas os participantes do padrão foram devidamente desenvolvidos.

2.5 Aspectos e Concerns

Programação orientada a aspectos [Kiczales97] (POA) e Programação orientada a Concerns (POC) [Tarr00] são técnicas de representação da abstração promissoras, pois permitem a especificação de áreas de interesse relevantes ao desenvolvimento de software independentes do domínio da aplicação. Esta representação se concentra na descrição de uma dimensão específica do desenvolvimento como, por exemplo, Persistência, Concorrência ou Distribuição, informando todas as classes, objetos e coisas afins que são relevantes na sua realização. Sendo assim, POA e POC podem ser consideradas como técnicas de representação de um design e/ou implementação que descreve a estrutura e a funcionalidade da solução de um determinado problema (a dimensão). Esta definição é muito parecida com a de Frameworks e Design Patterns, diferindo apenas na forma de aplicação (o "cross-cutting") que em sua maioria violam o mecanismo de encapsulamento presente em linguagens de programação orientadas a objetos, onde Frameworks e Design Patterns estão fortemente baseados.

Sendo assim, embora não se tratando de uma forma explicita de reutilização e sim uma técnica de design, a reutilização dos conceitos presentes na dimensão também sofrem dos problemas como integridade e documentação, podendo ser parcialmente atenuados com a utilização da abordagem proposta.

2.6 Conclusão

É possível afirmar que as técnicas de reutilização evoluíram significativamente desde a proposição inicial de McIlroy, permitindo o desenvolvimento de aplicações em tempo e custo menores com uma maior qualidade. A tabela abaixo faz um resumo das tecnologias de reutilização mencionadas, relacionando a técnicas de reutilização, o nível de reutilização obtido, o tipo a forma de reutilização.

⇒ **Técnica** – Nome da técnica adotada.

⇒ **Nível** – Qual é o maior benefício da abordagem. A nível de código e/ou design.

⇒ **Tipo** - Tipo de integração com a aplicação em desenvolvimento: composicional por simples chamadas; adaptativo através de incrementos; gerativo com o auxílio de um programa.

⇒ **Utilização** – Descrição da utilização.

Técnica	**Nível**	**Tipo**	**Utilização**
Biblioteca	Código	Composicional	Parametrização de Funções
Componentes	Código	Composicional	Composição com a aplicação
Frameworks	Design e Código	Adaptativo	Redefinição de Pontos de Flexibilização.
Aspectos E Concerns	Design e Código	Adaptativo	Cross-Cutting
Sistemas Gerativos	Design e Código	Gerativo	Através da definição de especificações dos programas alvo e destino.

3 Tecnologias Relacionadas

O principal objetivo da abordagem proposta é sistematizar o processo de reutilização de especificações orientadas a objetos através da descrição e execução guiada deste processo. Um ponto importante em tal ambiente é relativo à sua usabilidade. Como mencionado no capítulo 1, este ambiente tem como um de seus requisitos, a capacidade de se integrável ao processo de desenvolvimento de software bem como às ferramentas por ele utilizadas. Para viabilizar tal nível de usabilidade, um conjunto de tecnologias foi investigado. Estas tecnologias são relativas à funcionalidade de um ambiente de reutilização e representação das estruturas internas, e serão apresentadas neste capítulo.

3.1 Ambientes de Reutilização

O processo de reutilização é caracterizado pela execução de um conjunto de atividades que vão desde a busca por um artefato útil, passando por uma verificação de compatibilidade, até chegar à sua efetiva utilização. Em sua maioria, ambientes voltados à reutilização não abordam as duas atividades iniciais, devido incapacidade de se obter uma especificação precisa das semânticas envolvidas que possa ser benéfica ao processo como um todo. Por outro lado, estes ambientes focam seu auxílio durante a utilização destes artefatos seja por meio de visualizadores como os Inspetores dos Gui-Builders, seja por meio de execução de um script como em HookTool [Froelich97].

Especificamente, ambientes como os *Gui-Builders* da Borland [Dephi] (figura 12) ou Visual Studio da Microsoft[Mfc] entre outros, são exemplos de sucesso no âmbito da reutilização. Estes ambientes possuem uma estratégia "drag n´ drop" para reutilização de componentes (figura 12). Uma vez escolhido a partir de um repositório, o componente é arrastado para a

aplicação em questão e configurado de acordo com a necessidade do reutilizador. Esta configuração é elaborada e verificada por um inspetor de objetos (componentes) através da apresentação de um conjunto de propriedades (atributos do componente) e verificação dos efeitos colaterais de tal configuração. Vale ressaltar que esta verificação é feita a partir de um código presente dentro do próprio componente e não especificado de forma genérica.

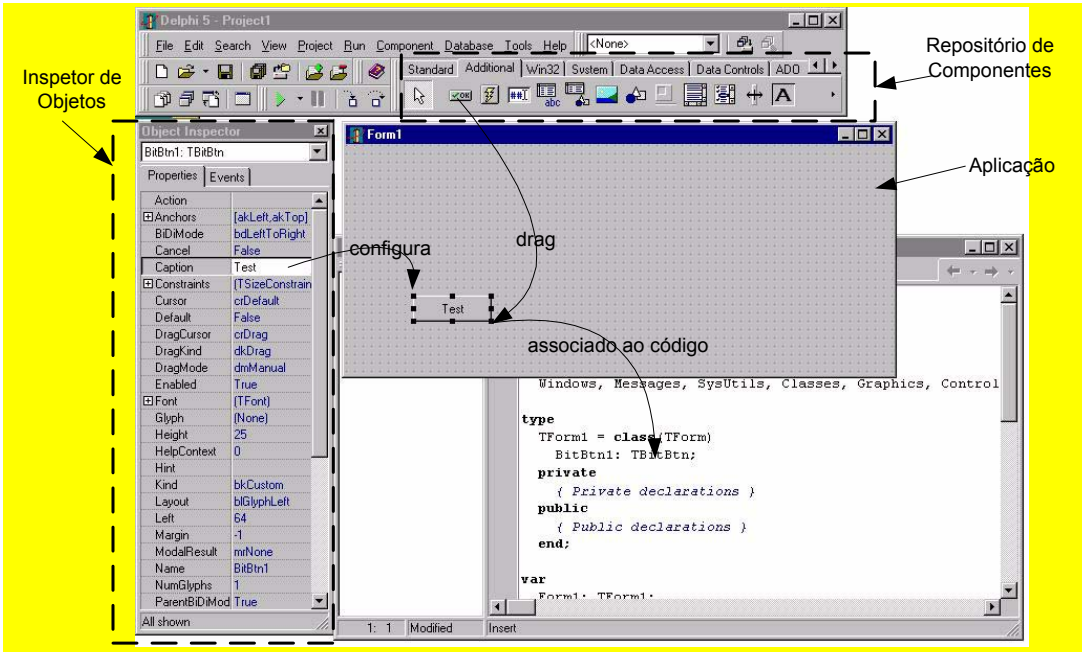


Figura 12 Instanciação via Inspetor de objetos.

Analisando estes ambientes foi possível verificar dois tipos de problemas. O primeiro diz respeito à tecnologia de reutilização adotada, ou seja, a tecnologia de componentes. É possível afirmar que esta tecnologia é extremamente útil para a introdução de funcionalidades na aplicação em desenvolvimento como interfaces gráficas, acesso à banco

de dados e etc. Entretanto, quando e necessário o desenvolvimento de uma aplicação como um todo, este tecnologia não se aplica devido à complexidade de configuração[1] necessária.

O segundo problema é relativo à amarração com um único tipo de framework que é distribuído com o próprio ambiente. Esta amarração limita o auxílio ao processo de reutilização, uma vez que a ferramenta de auxílio, no caso os inspetores, estão intimamente ligados à forma de codificação do framework (figura 12).

Uma outra tentativa de se auxiliar o processo de entendimento (e, por conseguinte a utilização) de um framework é descrito em [Lange95] Este trabalho descreve a utilização de design patterns [Gamma95] para promover o entendimento da estrutura de design existente em um framework. O auxílio no entendimento é obtido através do uso se um programa denominado Program Explorer, que ao invés de apresentar o fluxo de execução completo de um programa, permite a filtragem de determinadas características como, por exemplo, todos os objetos "observáveis" (do padrão Observer) de uma aplicação (figura 13). Vale ressaltar que tal visualização é feita com o framework, ou seja, uma aplicação dele instanciada, executando. Embora a utilização de design patterns auxilie o entendimento de frameworks, nem todos os pontos de extensão de um framework são expressos com esta tecnologia. É possível afirmar que mecanismos como herança e parametrização normalmente são escolhidos mais freqüentemente devido à sua facilidade de no uso. Este fato é comentado em [Lange95], pois lá se enfatiza que tal método é eficaz quando o desenvolvedor e o reutilizador são fluentes em patterns ("pattern-literate").

[1] Sistemas como o SAP [SAP] precisam de aproximadamente 4000 parâmetros para ser configurado.

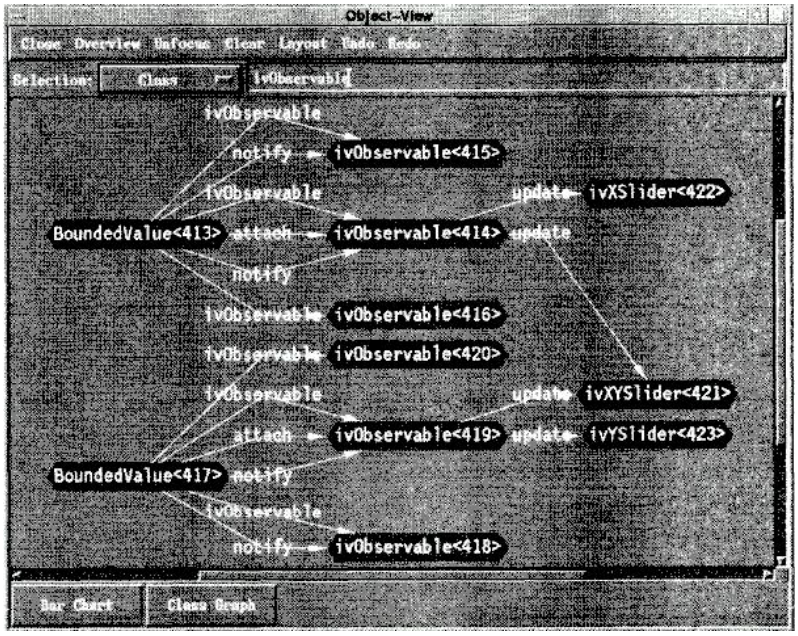


Figura 13 Tela do Program Explorer que apresenta a interação dos objetos observáveis.

Uma iniciativa interessante para o auxílio ao processo reutilização e que mais se assemelha à abordagem proposta nesta tese, é a ferramenta para Hooks proposta em [Froelich97a]. Nesta abordagem uma ferramenta permite ao reutilizador navegar, executar e desfazer as ações de reutilização, bem como visualizar o design OO do framework distinguindo as classes do framework das classes da aplicação (figura 14). O interessante desta proposta é a forma independente com que auxílio à reutilização é representado. Isto é alcançado através do uso de uma documentação externa ao framework denominada Hooks (apresentado no capítulo 2), que especifica os passos necessários à reutilização. O problema com esta abordagem está na quantidade de elementos presentes na ferramenta (CASE, Linguagem, Interpretador desta Linguagem , etc.), que aparentemente inviabilizou seu desenvolvimento, uma vez que não existem publicações que apresentam sua aparência e funcionalidade mas apresentam apenas seus requisitos.

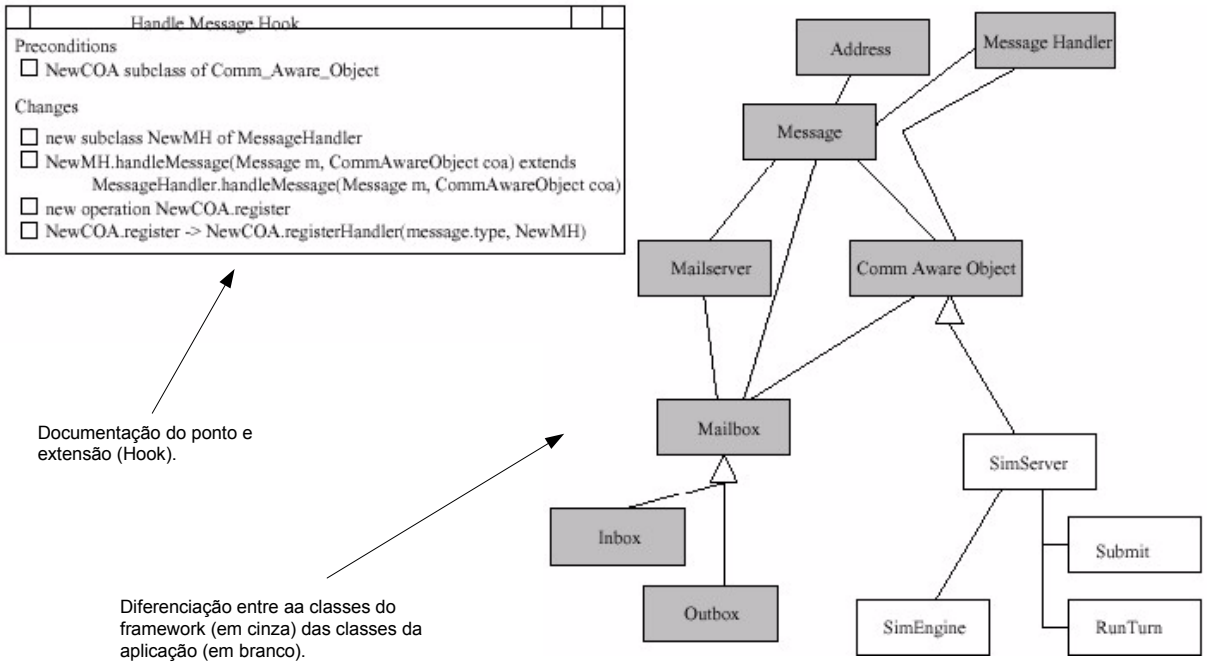
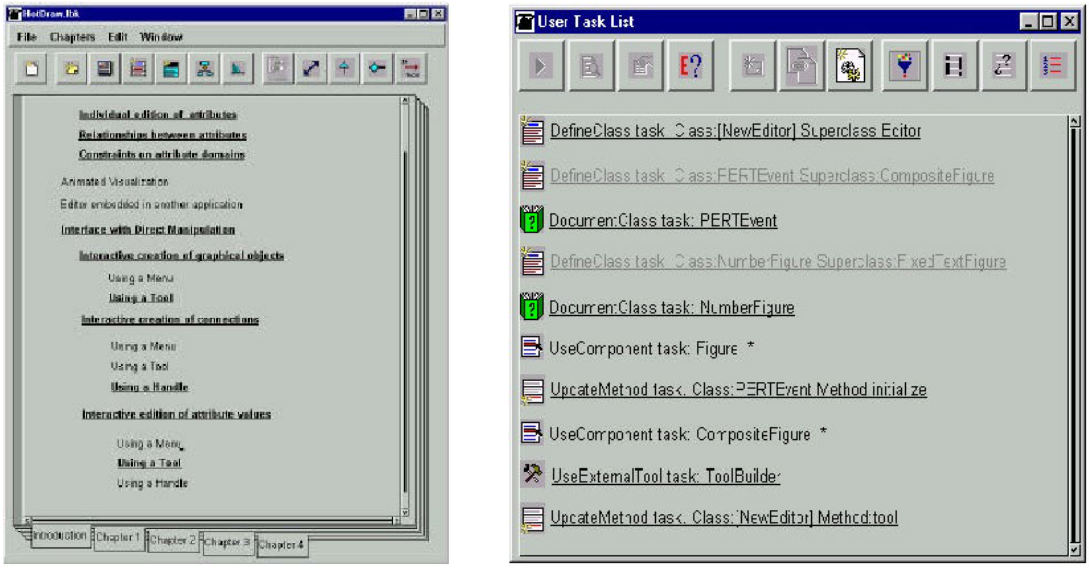


Figura 14 Especificação da instanciação de Hooks[Froelich97a]

Um dos últimos trabalhos apresentados para auxiliar o processo de instanciação foi o ambiente HiFI (Helping in Framework Instantiation) proposto por [Ortigosa00]. Este trabalho introduz o uso da tecnologia de agentes [Omg00] para assistir o reutilizador no processo de adaptação dos pontos de extensão. A abordagem inicia com uma descrição funcional (Figura 15) do framework que servirá de base para o reutilizador escolher a funcionalidade desejada. Após esta escolha, o agente monta um plano de instanciação, o SmartBook [Pree97], que contém as atividades básicas necessárias à reutilização e servirá base para o processo de assistência. Esta assistência é obtida através da utilização de regras de instanciação que descrevem um par, pré-condição -> efeito (figura 16), ou seja, quando o reutilizador desejar a ocorrência de um efeito específico durante a instanciação, o que ele tem que fazer é tornar a pré-condição verdadeira.

(a) (b)



Tela de escolha de funcionalidade. Tela de tarefas a serem executadas.

Figura 15 Ambiente HiFI.

Embora esta abordagem seja interessante devido à união das tecnologias de agentes e instanciação de frameworks, ainda existe muito a ser feito. Primeiramente, a utilização do termo agente é um tanto quanto prematura, pois segundo diversos autores [Omg00] , um agente é caracterizado por um conjunto de propriedades que vão desta adaptabilidade , passando por comunicabilidade e aprendizagem. Sendo assim, o agente utilizado pelo ambiente HiFI deveria ser denominado wizard (como o presente nesta tese), pois este não apresenta tais características. A construção do plano de instanciação, aparentemente, não representa nenhuma tarefa que necessite alguma característica de agentes, pois o mapeamento entre a funcionalidade e o conjunto de regras que a instanciam é feito manualmente pelo desenvolvedor do framework.

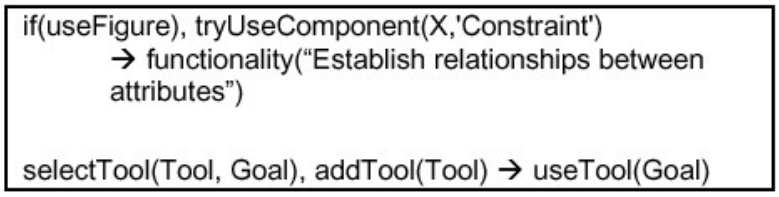


Figura 16 Exemplo de uma regra em HIFI.

Quando o reutilizador necessitar usar uma ferramenta (useTool), ele necessita selecionar uma ferramenta (selectTool) e adicionar esta ferramenta (addTool).

Um outro problema diz respeito à introdução de um conceito totalmente diferente dos utilizados na programação orientada a objetos (base para frameworks), que é a regra de instanciação. Representar um processo de instanciação com pares pré-condição => efeito, exige uma capacidade de abstração normalmente presente em linguagens como VDM , onde o sistema é especificado através de pré-condições, pós-condições e invariantes. Desenvolvedores que utilizam a programação orientada a objetos estão acostumados a especificar seqüências de atividades diretamente no código ou quando muito, utilizando os diagramas presentes em linguagens de modelagem. Sendo assim, estes desenvolvedores têm que passar por um processo de mudança de paradigma para utilizar tal forma de especificação.

3.2 Especificação do Design OO – UML

A necessidade de uma forma de expressão comum para representação de especificações de sistemas, levou a comunidade de desenvolvimento de software a adotar a linguagem UML [Booch99] como tal representação. Inicialmente desenvolvida como uma união das metodologias Booch [Booch95], OMT [Rumbaugh96] e OOSE [Jacobson94], UML reúne em seus conceitos as práticas mais utilizadas na aplicação da tecnologia de objetos. Aceita em

1997 como um padrão da OMG (Object Management Group), esta linguagem tem como objetivo principal promover a visualização, especificação, modelagem e documentação da arquitetura de um sistema.

UML define nove tipos de diagramas são agrupados de acordo com o tipo de especificação:

Diagramas Estruturais – Especificam a estrutura do sistema.

- ⇒ Diagrama de classes
- ⇒ Diagrama de objetos

Diagramas Comportamentais – Especifica o comportamento dinâmico do sistema.

- ⇒ Diagrama de Caso de Uso
- ⇒ Diagrama de Seqüência
- ⇒ Diagrama de Colaboração
- ⇒ Diagrama de Estados
- ⇒ Diagrama de Atividades

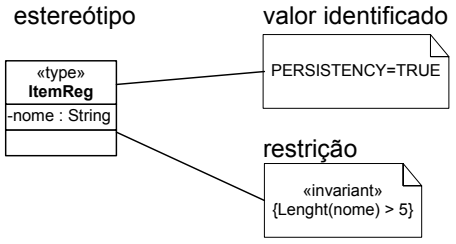
Diagramas de Implementação – Especificam a estrutura da implementação.

- ⇒ Diagrama de Componentes
- ⇒ Diagrama de Configuração

Analisando a programação orientada a objetos (base para frameworks) e a utilização de ferramentas CASE que implementam UML [Rose] [Argo] [Together], é possível verificar que os diagramas que contribuem para o design final do sistema na verdade manipulam classes. Uma vez que estas classes são completamente acessíveis através do diagrama de classes, a abordagem proposta por este trabalho tratará apenas este tipo de diagrama.

Um ponto importante presente na definição de UML é relativo aos seus mecanismos de extensão. Estes mecanismos permitem a introdução de elementos ao meta-modelo de UML para que novas características possam ser representadas na modelagem de um sistema.

Estes elementos podem ser definidos através de estereótipos, valores identificados e restrições (figura 17). Estereótipos aumentam o vocabulário de UML permitindo a definição de novos elementos que podem ser utilizados em conjunto com os existentes. Valores identificados permitem a extensão de uma propriedade de um elemento. Restrições, por sua vez, podem ser utilizadas para modificar a semântica de um elemento.



Os mecanismos de extensão utilizados permitem especificar que o elemento presente no design :

- é um tipo utilizando o estereótipo "type"
- é persistente utilizando o valor identificado " persistency=true"
- seu nome tem pelo menos 5 caracteres utilizando a restrição "invariant=Lenght(nome)>5"

Figura 17 Mecanismos de extensão de UML.

Como será visto nos próximos capítulos, a abordagem proposta neste trabalho utiliza estereótipos e valores identificados para definir os tipos de pontos de extensão que um elemento presente no design pode assumir.

3.3 Representação do Design OO – XMI

Um ponto importante da abordagem proposta é a usabilidade da ferramenta de auxílio ao processo de instanciação desenvolvida. Esta ferramenta tem como um de seus requisitos manipular designs (diagramas de classes) desenvolvidos em ferramentas CASE externas ao ambiente. A manipulação automática de diagramas de classes em seu formato gráfico é

bastante difícil e normalmente não é utilizado por tais ambientes. Para resolver este problema, foi investigado um formato de representação padrão, que fosse utilizado por diversos ambientes de desenvolvimento de designs orientado a objetos. Esta investigação conduziu ao formato XMI (XML Metatada Interchange) para a representação dos designs envolvidos.

O formato XMI foi desenvolvido com a intenção propiciar ,de forma fácil, o intercambio de metadados entre ferramentas CASEs, Bancos de Dados, Ferramentas de desenvolvimento, Repositórios de Geradores de Relatório (figura 18).

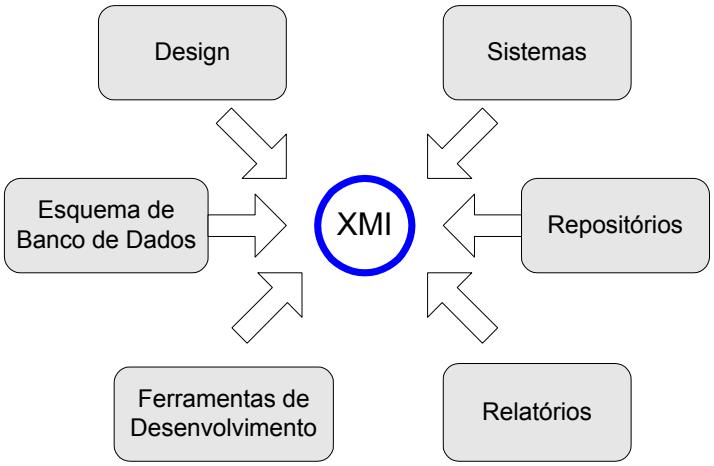


Figura 18 Integração das aplicações via XMI

Este intercambio é alcançado através da integração de três padrões presentes no âmbito do desenvolvimento de software:

⇒ XML - eXtensible Markup Language, padrão W3C.

Linguagem de marcação que permite a descrição do design a ser intercambiado em um formato estruturado independente da tecnologia de representação.

⇒ UML - Unified Modeling Language, padrão de modelagem OMG.

Permite a descrição e visualização de um design orientado a objetos.

⇒ MOF - Meta Object Facility, padrão OMG para descrição de meta-metamodelos.

Permite a descrição do metamodelos de UML.

A união destes padrões pode ser melhor verificada analisando-se a descrição das camadas de metamodelos proposta pela OMG (tabela X). Vale ressaltar que a descrição em cada camada inferior é feita em conformidade com a camada superior. Sendo assim, XMI pode ser definida como a representação do metamodelo que define UML em XML, ou seja, quais são os componentes que especificam um design (modelo) em UML.

Camada	Descrição	Exemplo
meta-metamodel	Define a linguagem de especificação de metamodelos. (MOF)	*MetaClass, MetaAttribute, MetaOperation*
metamodel	Define a linguagem de especificação de um modelo. (Define UML)	*Classe, Atributo, Operação, Componente.*
model	Define a linguagem que descreve a informação de um domínio. (Define um modelo em UML)	*Figura, Janela, Mover.*
user objects (user data)	Define a informação de um domínio. (Define os Dados)	*<Circulo,10,Azul>*

Tabela 2 Camadas do Metamodelo da OMG.

4 Motivação - Um Exemplo de Reutilização

O processo de reutilização de um design orientado a objetos é composto pela execução de diversas atividades básicas de reutilização tais como especialização e redefinição de operações. Embora estas atividades sejam extremamente simples, seu encadeamento pode levar a situações nada triviais uma vez que estas podem ser feitas de forma não contínua e/ou por várias pessoas. Tendo isto em mente, o objetivo deste capítulo é apresentar um exemplo do processo de instanciação como um todo, ressaltando as situações não triviais originadas. Este exemplo servirá de base para as explicações da proposta apresentada nos capítulos posteriores.

Este exemplo de reutilização se baseará no framework DTFrame[1] (Drawing Tool Framework). O DTFrame é um framework *whitebox* , semelhante ao HotDraw [Johnson92], originalmente desenvolvido para prover capacidade de desenho para a ferramenta CASE 2GOOD[Carvalho98] (figura 19) desenvolvida no âmbito do projeto ARTS [Carvalho98]. A escolha do DTFrame como exemplo foi devido à maturidade existente no domínio de ferramentas de desenho que possibilita um fácil entendimento dos problemas relativos à instanciação sem desviar o foco para problemas de design.

[1] O Design original do framework DTFrame foi levemente alterado para ressaltar os problemas inerentes ao processo de reutilização.

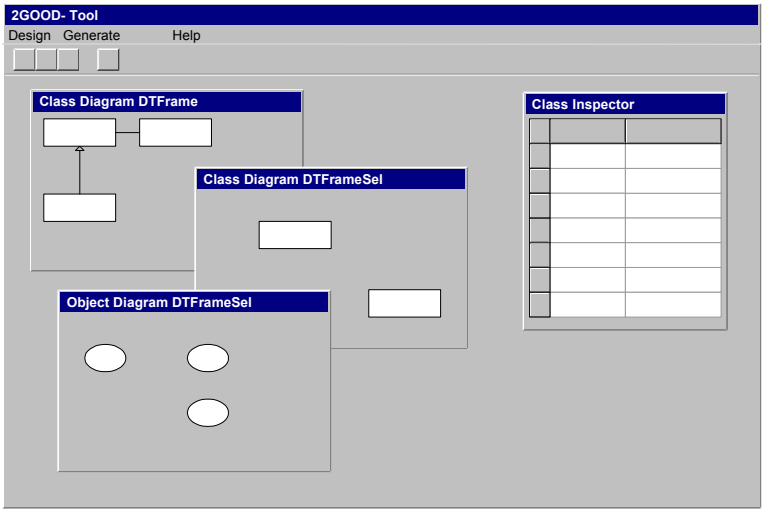


Figura 19 Ambiente 2GOOD.

O processo de reutilização tem início com a consulta a um documento (Descrição Informal) que descreve informalmente o artefato reutilizável. Este documento é consultado pelo reutilizador para verificar se o artefato em questão atende a suas necessidades de desenvolvimento. Para o framework DTFrame este documento segue abaixo.

Descrição Informal – (Parte I)

"O Framework DTFrame tem como objetivo facilitar o desenvolvimento aplicações que necessitam características de desenho. Estas características são : criação de uma figura, movimentação de uma figura, remoção de uma figura e redimensionamento de uma figura. Junto com as figuras existem conectores que provêm a ligação destas figuras entre si. Estes conectores são representados como retas que podem conter mais de um segmento. Outras características presentes em todos os elementos do desenho são : adicionar um texto alteração da cor, alteração da fonte. O desenho como um todo tem características de impressão, persistência e exportação. A característica de persistência provê salvamento em

disco do desenho e é opcional. A característica de exportação provê a criação do desenho em um formato externo e também é opcional."

Um outro ponto importante para um documento como este é descrever superficialmente como os elementos do artefato podem ser utilizados.

Descrição Informal – (Parte II)

"*A criação de novas figuras se dá por especialização da classe Figure. Esta classe contem a funcionalidade básica de todas as figuras que o DTFrame pode manipular.*

Sendo uma característica opcional, o mecanismo de persistência pode não aparecer no design final. Caso o reutilizador não necessite de tal funcionalidade, ele deverá ignorar o atributo DrawingTool.thePM. Caso contrário, o reutilizador deverá especializar a classe PersistencyTool.

A característica de exportação segue o apresentado para persistência, sendo que o atributo opional é DrawingTool.theEM e a classe para especialização é ExportTool."

Uma vez consultado estes documentos, o reutilizador já tem uma boa noção do artefato e de como reutilizá-lo. Como se trata de um framework whitebox, o próximo passo é apresentar um diagrama de classes seguido de uma descrição mais detalhada dos elementos importantes ao processo de reutilização.

Descrição das Classes

Classe BMPFormat - Classe que faz a exportação do desenho em formato Bitmap.

Classe Canvas - Tela onde as figuras são efetivamente desenhadas.

Classe DrawingTool - Classe principal da ferramenta de desenho. Contem uma lista de janelas, um gerente de persistencia e um gerente de exportação.

Classe DrawingWindow - Janela de Desenho

Classe ExportManager - Classe que gerencia todos os aspectos de exportação.

Classe ExportTool - Classe que efetivamente faz a exportação.

Classe Figure -Classe abstrata que descreve o comportamento mínimo de uma figura

Classe FigureAction - Classe que descreve as ações que podem ser feitas em uma figura como: Apagar, MudarNome.

Classe FigureData - Classe que descreve os dados da figura como : ID, nome

Classe GIFFormat - Classe que faz a exportação do desenho em formato GIF.

Classe MSSQLDB - Wrapper para banco de dados SQL Server.

Classe OODB - Wrapper para banco de dados OO.

Classe OracleDB -Wrapper para banco de dados Oracle.

Classe PersistencyManager - Classe que gerencia todos os aspectos de persistência.

Classe PersistencyTool - Classe que efetivamente faz a persistência.

Interface Persistent - Interface que declara o protocolo de comunicação para persistência.

Class RecDB - wrapper para banco de dados orientado a registro.

Classe RelDB - Wrapper para banco de dados relacional.

Classe ToolBar - Barra de ferramentas associada a uma janela de Desenho.

Classe XMIDB - Wrapper para banco de dados em XMI.

Tabela 3 Especificação das Classes do Framework DTFrame.

Diagrama de Classes

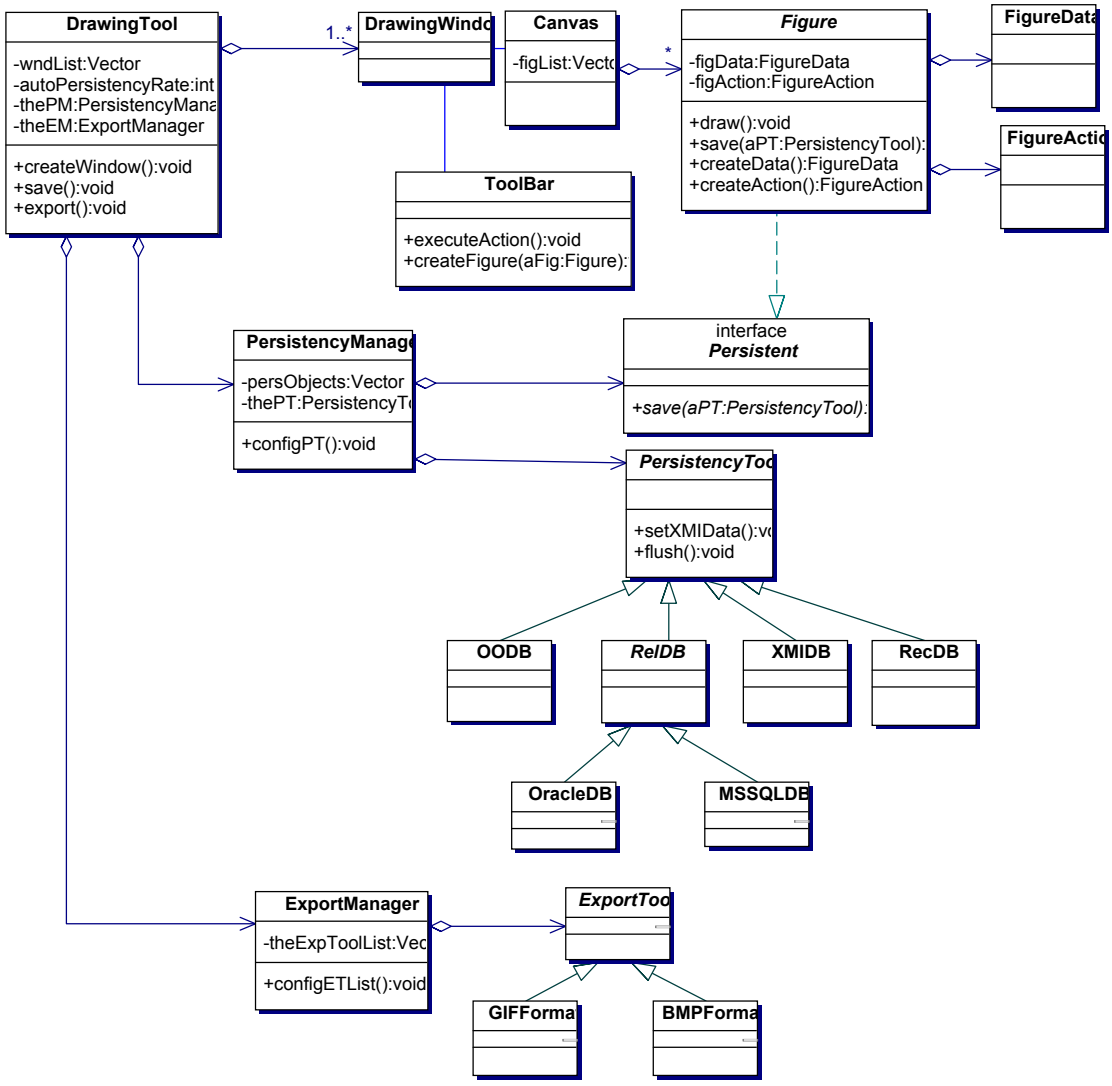


Figura 20 Diagrama de Classes do Framework DTFrame.

Embora o diagrama de classes seja útil ao processo de entendimento do design, ele por si só não diz muito ao processo de instanciação, uma vez que as atividades de instanciação não estão especificadas neste diagrama. Sendo assim, se faz necessário uma documentação auxiliar com tal propósito. No caso do DTFrame, esta documentação é descrita como:

Descrição da Reutilização (Parte I)

"*Para uma efetiva reutilização do DTFrame o reutilizdor deverá :*

⇒ *Para cada tipo figura presente na aplicação final, criar uma nova classe herdando de Figure que especifique a nova figura. Criar também classes que herdam de FigureData e FigureAction para especificar os dados e as ações associados a esta figura.*

⇒ *Caso a aplicação do reutilizador necessite de persistência, redefinir a operação save() para cada tipo figura criada. Esta operação deverá especificar tuplas no formato <TIPO, NOME, VALOR> de cada atributo a ser salvo.*

⇒ *Para especificar o aspecto de persistência, o reutilizador deverá escolher uma subclasse concreta de PersistencyTool, e definir seu uso em PersistencyManager.configPT(). Caso o tipo de banco de dados utilizado pela aplicação final não esteja representado, criar uma subclasse de PersistencyTool como encapsuladora do SGBD. Vale ressaltar que este aspecto é opcional.*

⇒ *Para especificar o aspecto de exportação, o reutilizador deverá escolher um conjunto de subclasses de ExportTool, e definir seu uso em ExportManager.configET().Vale ressaltar que este aspecto é opcional.*

Uma questão importante durante a reutilização do DTFrame é relativa a seus aspectos opcionais que implicam em atividades de reutilização condicionais com :

Descrição da Reutilização (Parte II)

⇒ *A utilização do mecanismo de persistência obriga ao reutilizador a usar/definir as operações Figure.save() e DrawingTool.save() e configurar o atributo DrawingTool.autoPersistencyRate.*

⇒ *A utilização do mecanismo de exportação obriga ao reutilizador a usar/definir a operação DrawingTool.export()".*

De posse de toda esta documentação, Descrição Informal, Diagrama de Classes, Descrição das Classes e Descrição da Reutilização, o reutilizador pode dar início ao processo de reutilização. O primeiro passo seria construir um esboço de funcionamento da aplicação em desenvolvimento. Este esboço tem como objetivo, descrever de forma superficial os elementos do sistema.

Esboço Inicial

"*A aplicação em desenvolvimento deverá ser capaz de prover capacidade de desenho para":*

⇒ *Desenhar Classes usando a notação UML*

⇒ Ligar estas classes de acordo com os relacionamentos de classes existentes em UML e Etc.

Elementos:

⇒ *Figura para Classe*

⇒ *Figura para Ligações de Classes*"

Uma vez feito este esboço, dá-se inicio ao processo de reutilização (instanciação para frameworks) onde é possível observar o surgimento de problemas cujas soluções podem ser complicadas.

Onde estão os pontos de flexibilização? – Um fator de extrema importância para o processo de reutilização e que define a utilidade semântica do artefato é a definição do que pode ou não ser alterado (parametrizado) pelo reutilizador. No caso do DTFrame, por

exemplo, o ponto de flexibilização representado pela classe *Figure* desempenha um papel mandatório na aplicação gerada. Sendo assim, a especificação da classe *Figure* deve ser incrementada de forma a destacar esta classe das outras presentes no design.

Como preencher estes pontos? – Uma vez identificado o ponto de flexibilização é necessário saber como proceder à reutilização (parametrização). Existem casos onde a esta reutilização é simples como uma redefinição ou complexa como a aplicação de um Design Pattern [Gamma95].

No framework exemplo, é possível notar a presença do método *Figure.draw()* . Este método especifica o formato do desenho da figura. Claramente esta parametrização é via redefinição do método em questão. Entretanto é possível imaginar uma situação onde a forma de desenho da classe base (*Figure*) não se adequa à aplicação em desenvolvimento, pois esta necessita de um desenho com cor (o que não é possível diretamente via DTFrame). Neste caso, esta redefinição poderia ser feita utilizando-se o padrão *Strategy*[Gamma95] que permite aumentar a funcionalidade da classe base, sem alterar sua interface. Neste caso, a reutilização de Figure.draw() poderá ser ou via uma redefinição normal ou via aplicação do Design Pattern *Strategy*.

Estas decisões de quais possíveis reutilizações podem ser aplicadas a um determinado ponto de flexibilização devem ser feitas pelo projetista do framework e não pelo reutilizador.

Sobreposição semântica – O espaço de nomes presente no esboço inicial do design da aplicação já contém algumas definições que podem conflitar semanticamente com o existente no framework. Esta situação pode ser verificada durante a reutilização de

DTFrame, caso o reutilizador necessite a criação de uma classe *Canvas*. Como esta classe já existe no framework, sua introdução no design final implicaria na utilização de um artifício presente em de linguagens de programação como modularização (pacotes). Embora esta solução seja simples, sua utilização leva a um design de baixa qualidade que tende a separar os módulos em função de nomes e não da afinidade semântica (figura 21).

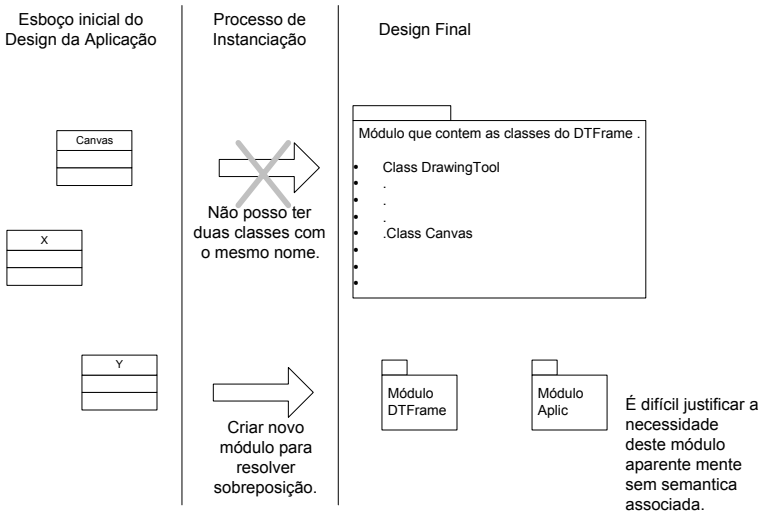


Figura 21 Solução do problema de sobreposição semântica via modularização.

Herança Múltipla – Da mesma forma que a sobreposição semântica, o aparecimento de herança múltipla tem sua origem no esboço do design inicial da aplicação. Este esboço pode conter a especificação de uma subclasse a qual, durante o processo de reutilização, poderá tornar-se subclasse de uma classe originada no framework. Isto acarreta o surgimento de herança múltipla, cujo mapeamento não é direto para certas linguagens de programação[Delphi] [Java] [Ada], levando a problemas de rastreabilidade entre design e implementação. Embora desenvolvedores tentam evitar esta prática de design, em sistemas grandes isto é quase impossível [Eden98].

Parada e Retomada das Atividades – Embora existam frameworks cuja reutilização pode ser feita de forma rápida e trivial [JUnit] , as atividades envolvidas neste processo normalmente não são executadas de forma contínua. Esta execução é comumente interrompida e retomada tempos depois. Sendo assim, a marcação do ponto de parada é de extrema importância para que o reutilizador possa restaurar o contexto exato do ponto de retomada de execução processo.

O problema com esta marcação de parada é onde e como representa-la, de forma que a retomada do processo possa ser imediata (ou pelo menos correta). Uma estratégia muito utilizada por programadores é a introdução no código de algum comentário no ponto exato da parada. Esta solução obviamente não traz muitos resultados ao processo de reutilização pois este normalmente é orientado a design e não a código. Por outro lado, ferramentas CASE em sua maioria não suportam o processo de reutilização de design e como conseqüência não abordam este problema.

Uma outra solução adotada é utilizar uma ferramenta externa, como um processador de texto, para descrever esta marcação com a utilização de linguagem natural. Sem sombra de dúvidas esta abordagem pode trazer problemas de interpretação, que se tornam mais evidentes quando a pessoa que descreveu a marcação for diferente da que irá consultar. Alem de tudo, para se obter uma representação útil, seria necessária a descrição das atividades que antecederam a parada para representar o contexto como um todo, o que pode ser extremamente exaustivo.

Verificação de Restrições – Uma questão importante para o processo de reutilização é relativo às atividades dependentes. A verificação das restrições de dependência pode se tornar uma tarefa "*error-prone*" uma vez que estas podem ocorrer em cascata e de forma pulverizada no design. No framework DTFrame é possível verificar a presença de um aspecto de persistência como opcional, que implica na existência de uma atividade de instanciação que escolhe este aspecto (Figura 22). O resultado desta atividade, aspecto escolhido ou não, é necessário para a atividade de instanciação que redefine o método Figure.save(), ou seja, só faz sentido incluir características de persistência nas figuras da aplicação, caso a aplicação como um todo necessite persistência.

Caso o método Figure.save() seja redefinido para todas as subclasses de Figure, e o aspecto de persistência não seja escolhido, o design final da aplicação não irá refletir a funcionalidade obtida.

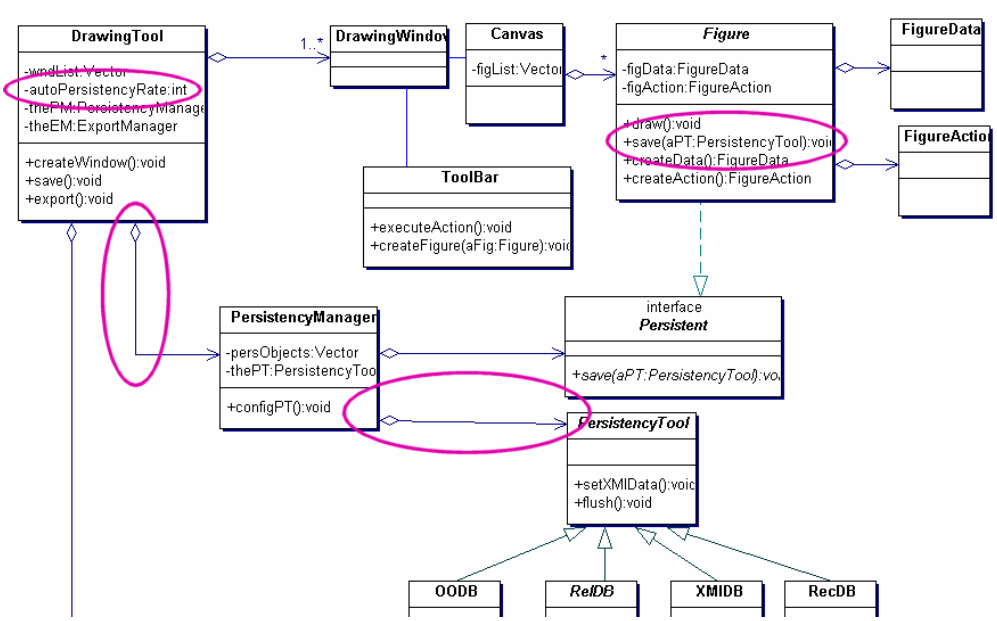


Figura 22 – Atividades de instanciação inter-dependentes.

Distribuição das Atividades – A complexidade envolvida nos sistemas atuais leva a distribuição das atividades de desenvolvimento por pessoas ou grupo de pessoas [Herbsleb01] [Bosch01]. No caso de desenvolvimento baseado em frameworks isto também é verificado [Mattsson00]. Uma vez distribuído o processo de instanciação, a acurácia do design final da aplicação pode ser comprometida devido ao problema de sincronização de atividades condicionais e a gerência do problema da parada (anterior).

No caso do framework exemplo, é possível imaginar uma situação hipotética onde a atividade de instanciação do aspecto de persistência fosse delegada a um especialista em banco de dados. Esta atividade implica no mapeamento da estrutura OO (originado nas subclasses de Figure) para as estrutura de um SGBD Relacional (tabelas), o qual é totalmente dependente do tipo de banco de SGBD. Sendo assim, a atividade de instanciação

que escolhe o banco é mandatória para a atividade de redefinição do método save() . Neste ponto é possível destacar dois problemas:

a) A atividade de escolha do banco tem que ser feita;

b) e antes da atividade de redefinição do método *save().*

5 Documentação da Reutilização

Independente do tipo da tecnologia de reutilização adotada, o processo envolvido na reutilização de um artefato está fortemente baseado em duas etapas: a busca pelo artefato e a reutilização através de parametrização e/ou extensão, deste artefato. Estas etapas têm como ponto de partida o entendimento de uma documentação, normalmente não estruturada e informal, que apresenta os conceitos envolvidos no artefato e em ambos os casos requer uma grande esforço de aprendizado por parte do reutilizador.

Sobre a primeira etapa é possível afirmar que seria facilmente resolvida caso existisse uma forma padronizada de representação do conhecimento, para que as especificações envolvidas pudessem ser comparadas e/ou acopladas de forma sistemática. Uma vez que esta forma padronizada ainda não foi desenvolvida (ou pelo menos não tenha sido adotada em larga escala) e devido à capacidade de raciocínio dos atores envolvidos no processo, a documentação envolvida nesta etapa é fortemente marcada pela informalidade de representação.

A segunda etapa, a de parametrização e/ou extensão do artefato, é caracterizada pela identificação, preenchimento e seleção dos pontos de flexibilização, através da execução de tarefas repetitivas como adaptação de interfaces, redefinição de operações, atribuição de propriedades e verificação de restrições. A sistematização desta etapa esta fortemente ligada a uma padronização para a forma de representação dos conceitos envolvidos no processo de uso do artefato e serão abordadas a seguir.

Em se tratando de tarefas repetitivas, como as mencionadas acima, o uso de uma abordagem sistemática/automática ajudaria a diminuir os erros inerentes a processos com estas características (citados no capitulo anterior), acarretando em uma maior confiabilidade do design final. Neste caso específico, ferramentas de Design (ferramentas CASE) e ferramentas de programação (compiladores e geradores de código) em geral não têm utilidade, uma vez que não estão voltadas para o processo de instanciação. No caso de ferramentas CASE, a incompletude das informações inerente ao processo cíclico de desenvolvimento de software, torna inviável a checagem das restrições presentes durante este desenvolvimento. Do lado das ferramentas de programação, esta checagem é ainda mais difícil uma vez que linguagens de programação usualmente não capturam (e nem deveriam) tais restrições de design.

Como dito anteriormente, para se obter uma forma sistemática de reutilização, primeiramente se faz necessário desenvolver uma forma adequada de representação dos pontos de flexibilização/extensão presentes no design. Esta representação deverá se basear em conceitos que caracterizem:

1. O que é um artefato reutilizável?
2. Como se caracteriza uma instância válida?

e ser capaz de responder as seguintes questões:

3. O que é reutilizar?
4. O que a documentação deverá representar?

O que é um artefato reutilizável?

Para responder a primeira questão, recorreremos a definição de frameworks[1] de aplicação presente em [Fayad99a], o qual afirma que "... framework de aplicação orientados a objetos é uma tecnologia promissora para materializar/reificar projetos e implementações de softwares comprovados, levando a redução de custo e ao aumento a qualidade do software". Para os artefatos reutilizáveis que serão abordados neste trabalho, esta definição também se aplica, uma vez que estes estão baseados em projetos e/ou implementações orientados a objetos comprovados e estão completamente direcionados para reutilização. A restrição de ser uma aplicação completa ou não pode ser relaxada no caso de componentes por exemplo, sem prejuízo aos conceitos presentes na definição adotada.

Um outro ponto importante na definição adotada é relativo à obrigatoriedade dos elementos presentes no design. Embora não apresentado por [Fayad99a], para se atingir um alto grau de reutilização, alguns elementos do design poderiam ser classificados como opcionais aumentando o número de possíveis aplicações candidatas.

Analisando a figura 23, é possível perceber que no caso de um framework monolítico, sem opcionais, todas as aplicações oriundas deste framework têm que "carregar" todo o design. Isto restringe o número de aplicações que podem ser geradas devido ao custo de se "evitar" esta funcionalidade extra. Em contrapartida, um framework com opcionais aumenta a usabilidade do design pois parte deste design pode ser simplesmente descartada.

[1] Vale lembrar que o termo artefato reutilizável esta sendo utilizado neste trabalho como sinônimo de framework.

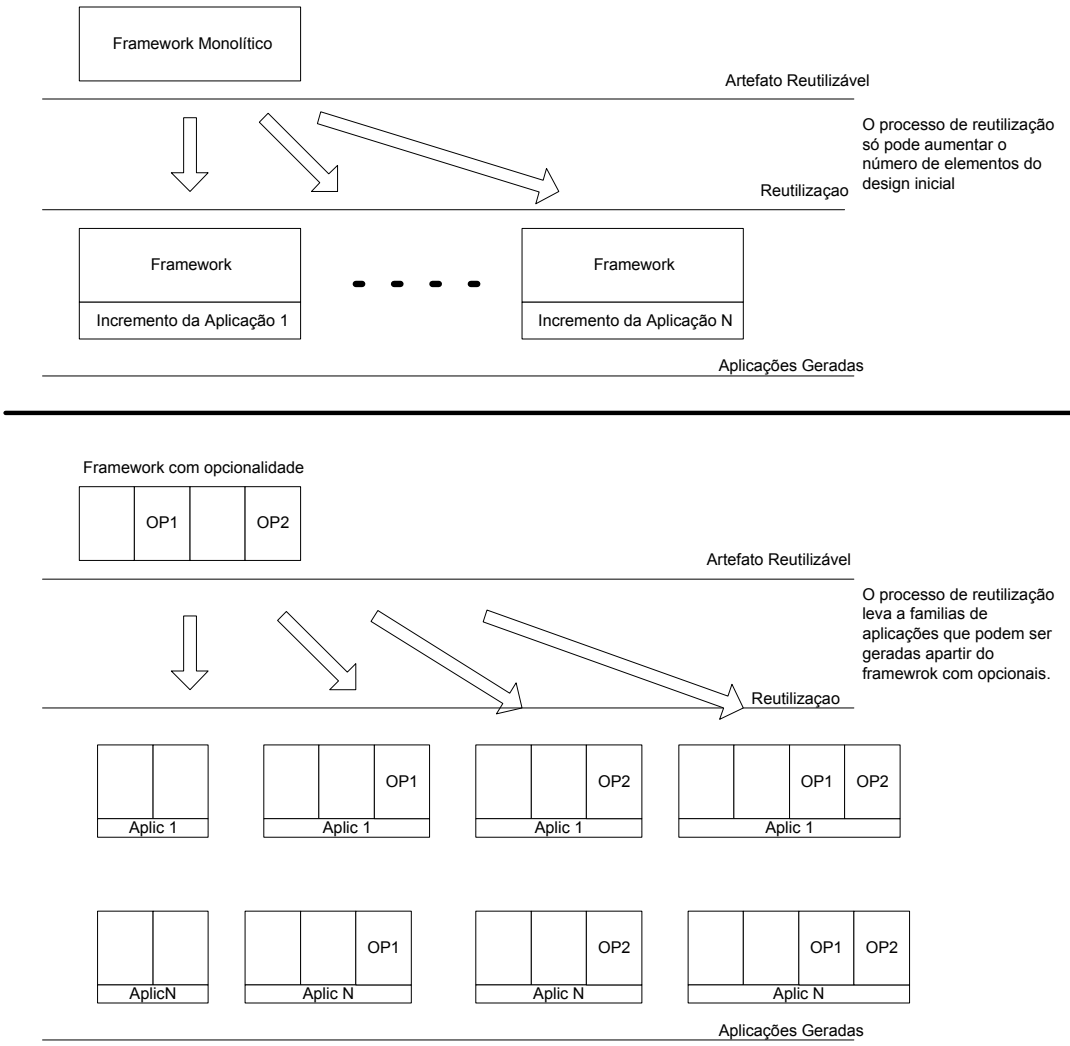


Figura 23 Aumento do numero de aplicações com a introdução da opcionalidade.

Como se caracteriza uma instância válida?

A caracterização de uma instância válida de um artefato reutilizável está diretamente relacionada com a sua definição. Dado que a definição adotada enfatiza reutilização e introduz o conceito de elementos opcionais no design do framework, é possível afirmar que uma instância válida é aquela que inclui em seu projeto/implementação, as partes obrigatórias especificadas pelo artefato (figura 24).

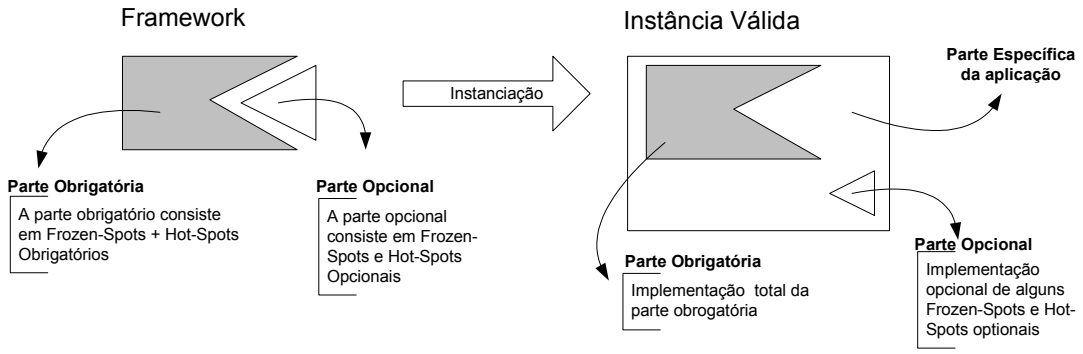


Figura 24 Forma de uma instancia válida.

Analisando a figura 24 é possível perceber que a representação de um framework é composta por duas partes, uma obrigatória (em cinza) e uma opcional (triangulo). A primeira parte representa o núcleo do framework e é responsável por todas as funcionalidades indispensáveis ao seu funcionamento como em [Fontoura99]. Nela estão representados seus hot-spots[1] obrigatórios, que são as partes flexíveis/extensíveis do artefato, bem como os frozen-spots obrigatórios [Pree95] que são as partes não flexíveis/extensíveis do artefato que comandam a execução dos hot-spots (como os template methods definidos em [Pree95]). Na segunda parte é representada a contrapartida opcional dos frozen-spots e hot-spots do artefato e podem não ser representados na aplicação final.

A necessidade de uma diferenciação (figura 25) clara para a representação da opcional e obrigatória advém do fato de que algumas funcionalidades do artefato podem não ser relevantes para a aplicação em desenvolvimento, mas não invalidam o processo de reutilização. Vale ressaltar que a definição de artefato reutilizável adotada está focada na

[1] Sinônimo de ponto de flexibilização.

reutilização do conhecimento para uma melhoria da qualidade/tempo/preço do produto final e não está restrita a um domínio específico.

Um outro ponto importante para a validade da instância, é a verificação das restrições envolvidas no processo. Estas restrições dizem respeito a seqüencialidade, necessidade de um estado e exclusão mútua e devem ser integralmente atendidas respeitando-se sempre as regras de opcionalidade.

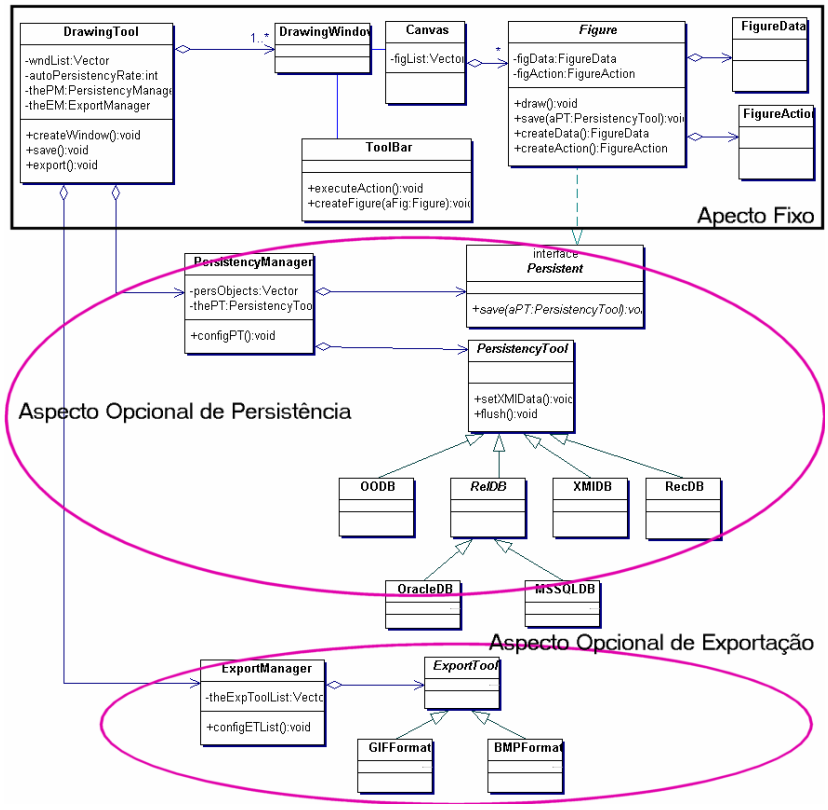


Figura 25 No DTFrame, a especificação dos aspectos opcionais têm que ficar destacados de alguma forma.

O que é reutilizar?

Tomando como base às definições anteriores, é possível especificar o ato de reutilizar como sendo um processo que tem como objetivo o preenchimento dos pontos de flexibilização/extensão presentes no artefato de forma a obter uma instância válida.

O que a documentação deverá representar?

Uma vez definido o que é reutilizar, a documentação de um artefato deverá representar de forma clara o processo de preenchimento, propiciando ao reutilizador a inclusão dos incrementos específicos da aplicação (IEA) em desenvolvimento, [Mattsson00] no design do artefato reutilizável, de forma a obter a uma instancia válida. Para tal, é possível identificar algumas necessidades como:

1. Em que lugar estes incrementos devem der colocados ("colados")?
2. Como efetuar esta "cola"?
3. Quais são os problemas e possíveis soluções inerentes a esta operação de "cola"?
4. Quais são as restrições de dependência envolvidas na extensão?

As necessidades 1 e 2 são demonstradas na próxima seção. As necessidades 3 e 4 estão mais relacionadas ao processo como um todo e serão demonstradas nos capítulos seguintes.

5.1 Identificação dos Pontos de Extensão

Uma forma de descrevermos onde e como estes incrementos devem ser colocados é através de clara representação do design do framework. O capítulo 3 introduz a linguagem de modelagem UML como sendo a linguagem utilizada para representar o design envolvido na

abordagem. Sendo assim, uma forma natural de identificar os pontos de extensão presentes neste design é estender UML de modo a capturar as características relevantes ao processo de reutilização. Em uma primeira abordagem, estas características relevantes podem ser identificadas como as classes, métodos e atributos, envolvidos em um ponto de extensão.

Esta extensão tem como base um subconjunto do trabalho apresentado em [Fontoura99], o qual foi desenvolvido com o objetivo representar o design do framework através da definição de uma linguagem para descrição de frameworks (UML-F) que estende UML. Esta extensão é obtida utilizando-se os mecanismos de extensão presentes em UML como os citados no capítulo 3. UML-F foi adotada devido a sua capacidade de representar pontos de extensão relativos a classes e métodos de forma clara e totalmente integrada a linguagem UML. Um outro fator importante para tal escolha é a grande aceitação desta linguagem no meio acadêmico.

Resumidamente, UML-F descreve pontos de extensão como sendo:

Classes de Extensão – Estereotipo *Extensible*

Especifica uma classe que pode ter sua funcionalidade aumentada através a introdução de novos métodos.

Métodos de Variação – Valor Identificado *Variable = TRUE*

Especifica um método cujo comportamento depende do processo de instanciação.

Interfaces – Restrição *Incomplete*

Especifica uma classe abstrata ou interface que deverá ser especializada pelo processo de instanciação.

Para prover uma documentação que atenda de forma mais completa às necessidades do reutilizador, é necessário ampliar este vocabulário. UML-F, por exemplo, não faz menção a aspectos relativos a atributos e também não deixa claro questões como uso de Design Patterns que são de extrema utilidade para o processo de reutilização. Um outro ponto não abordado pela UML-F é a opcionalidade dos elementos presentes no design. Tendo isto em mente, este trabalho introduz a UML-FI (UML – Framework Instantiation) como sendo uma extensão de UML-F que tem como objetivo especificar de forma mais precisa os elementos relevantes para o processo de instanciação.

Como abordado anteriormente, o processo de reutilização de um artefato é totalmente dependente da linguagem de especificação em que ele é representado. Sendo assim, a categorização dos tipos de elementos que estão presentes na especificação do artefato implica na definição do tipo da atividade de reutilização que será executada. No caso de frameworks, como o definido para este trabalho, sua especificação é representada em termos de classes, métodos e atributos, que são elementos básicos para a programação orientada a objetos. Estes elementos estão sujeitos a execução de atividades como especialização, redefinição de métodos e valoração de atributos. A abordagem UML-FI segue este raciocínio para especificar as atividades do processo de reutilização.

Em primeiro lugar UML-FI introduz o conceito de elemento reutilizável. Um elemento reutilizável pode ser uma classe, um método ou um atributo[1] sujeito à atividade de reutilização. A atividade de reutilização aplicada a este elemento é a definição de sua

[1] Uma associação é vista como um atributo.

presença, ou não, no design final. Esta especificação caracteriza a opcionalidade do elemento.

Usando a notação UML, esta especificação pode ser representada utilizando-se o valor identificado denominado REQUIRED com valor associado igual a OPTIONAL. Ou seja, para expressar que uma classe, método ou atributo pode ser opcional é necessário anexar ao elemento a construção UML, REQUIRED = OPTIONAL. Caso o elemento reutilizável não possua a especificação de opcionalidade é assumido um valor *default* REQUIRED = MADATORY, o que caracteriza o elemento como mandatário para o design final.

Vale ressaltar que a especificação de opcionalidade é transitiva, sendo propagada para os membros do elemento especificado. Sendo assim, é possível afirmar que:

⇒ Se uma classe é opcional, seus métodos e atributos também são opcionais;
⇒ Se um atributo é opcional, a classe que modela este atributo também é opcional;
⇒ Se um método é opcional, as classes que modelam seus parâmetros também são opcionais.

No exemplo DTFrame, para representar o aspecto de persistência como opcional é necessário decorar o atributo DrawingTool.thePM com o valor identificado REQUIRED=OPTIONAL. De acordo com a característica de transitividade presente na especificação, é possível inferir que a classe que modela este atributo, PersistencyManager, também é opcional e propaga esta opcionalidade para todos os seus membros (Figura 26). Um ponto importante desta propagação é a existência de outra referência a classe

PersistencyManager. Neste caso, uma vez que alguma outra referência seja não opcional, esta não opcionalidade é dominante.

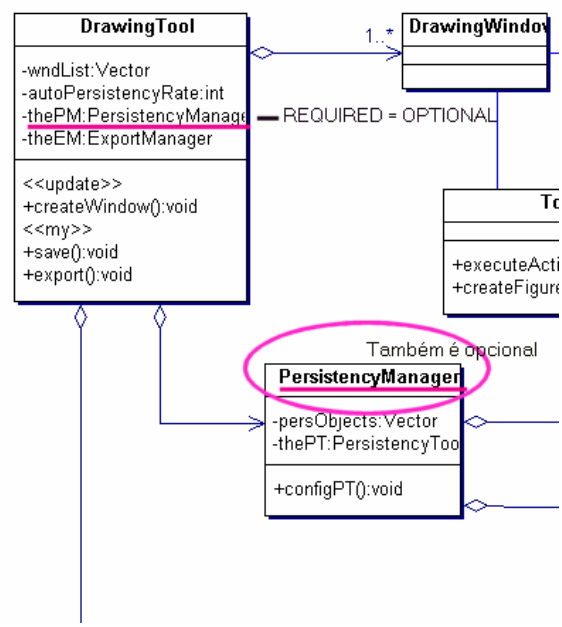


Figura 26 Propagação da especificação de opcionalidade.

É possível fazer uma analogia entre a definição de elemento reutilizável e uma especificação OO. Isto se deve ao fato que este elemento pode ser visto como a raiz de uma hierarquia de tipos de elementos sujeitos a atividades de reutilização (Figura 27). Estes tipos podem ser classes, métodos e atributos, e como em OO, definem atividades específicas para o seu contexto. Vale ressaltar que esta "classe" Elemento Reutilizável é definida como abstrata indicando que os elementos do design que são sujeitos a atividades de reutilização são suas "subclasses".

É possível destacar duas vantagens com a adoção desta analogia OO. Em primeiro lugar, é possível destacar a flexibilidade que ela traz para a solução proposta, uma vez que torna

possível a introdução de outros tipos de elementos reutilizáveis por especialização. Em segundo lugar permite utilização do termo Elemento Reutilizável como um termo único para os elementos participantes no processo de reutilização.

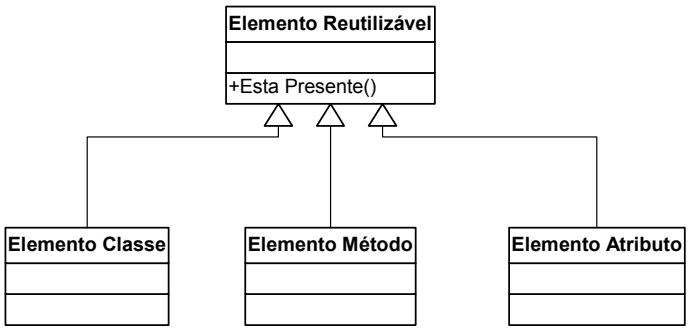


Figura 27 Hierarquia de elementos reutilizáveis.

Dando seqüência a definição de elemento reutilizável, existe o elemento do tipo Classe que representa uma classe em um design OO. A operação básica que o processo de reutilização pode executar sobre uma classe é redefinição. Esta atividade de redefinição tem como objetivo criar uma nova classe, de modo a representar as características específicas da aplicação [Mattsson00]. No exemplo DTFrame, a classe *Figure* representa uma classe que deve ser especializada durante o processo de reutilização para que o aspecto de desenho seja configurado de acordo com as necessidades do reutilizador (Figura 28). Utilizando a notação UML, esta classe deve ser decorada com o estereótipo CLASS_EXTENSION (Figure 28).

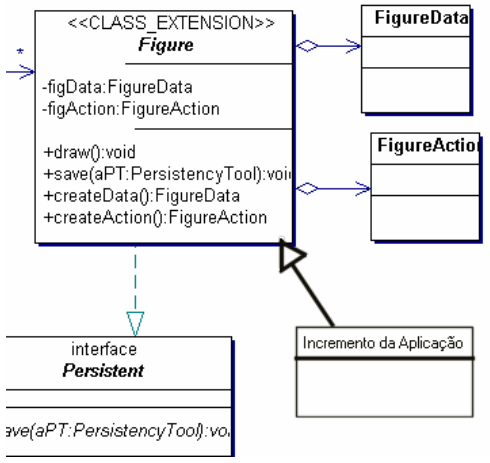


Figura 28 – Especificação da Classe Figure como classe para extensão.

Sob o ponto de vista do processo de reutilização, a atividade de especialização é encarada como trivial, pois em última análise, trata-se da atribuição de um nome a uma nova classe. Analisando de forma mais minuciosa, é possível verificar a existência de atividades mais complexas por trás de uma simples especialização.

É possível imaginar uma situação no framework DTFrame, onde esta classe *Figure* tenha sido previamente especializada pelo projetista do framework para prover alguma funcionalidade adicional (por exemplo, para prover figuras básicas como círculo, quadrado, ligação entre figuras etc..) (Figura 28). O reutilizador de um framework com tais características poderia desenvolver sua aplicação tomando como base estas especializações de figura para diminuir o esforço de desenvolvimento. Esta especialização prévia leva a um impasse. Uma vez que o reutilizador tenha decidido utilizar as classes que herdam de figura, ele assume que o comportamento definido pela classe Figure é imutável, pois qualquer extensão de *Figure* feita durante o processo de reutilização, não afetaria as classes previamente especializadas.

No caso do DTFrame hipotético, isto pode ser notado caso o reutilizador queira adicionar um atributo de cor na classe *Figure*. Para resolver tal situação, o projetista do framework pode definir que a estratégia de reutilização adotada será a aplicação de um Design Pattern (no caso o Strategy). A introdução deste Design Pattern permite a delegação da definição das características especiais necessárias para a aplicação em desenvolvimento para um componente externo a *Figure.* Pela figura 29 é possível notar que componente Figure.theSrategy é acionado quando a figura está para ser desenhada, permitindo assim que o atributo FigureStrategy.color seja utilizado.

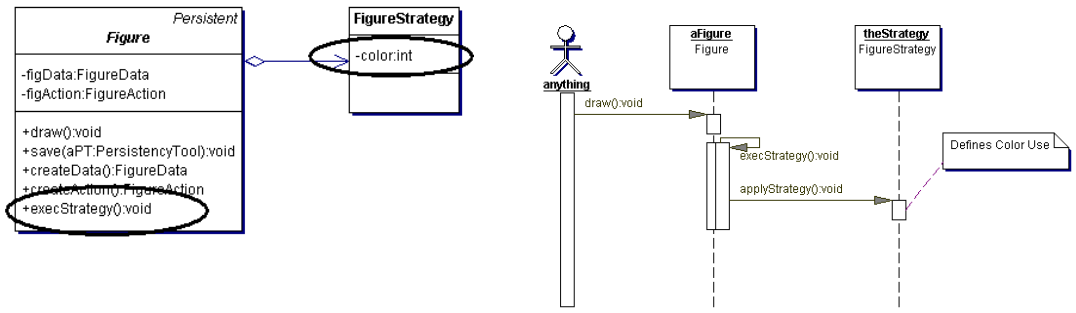


Figura 29 Aplicando o padrão Strategy.

Embora a adoção de um Design Pattern seja apenas uma estratégia para especialização, a indicação que esta estratégia pode ser adotada durante o processo de reutilização deve ser sinalizada no design do framework, pois esta reutilização exige um conhecimento mais profundo de técnicas de desenvolvimento OO.

A especificação UML para representar a extensão através da aplicação de um Design Pattern é feira através do estereótipo PATTERN_CLASS_EXTENSION (Figura 30).

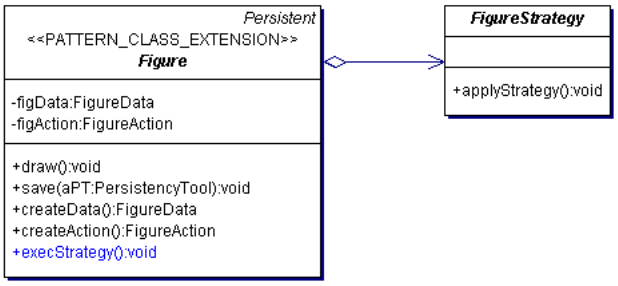


Figura 30 Especificação de um ponto de reutilização que utiliza Design Pattern.

Para completar os tipos de especialização de classes é introduzido o conceito extensão por seleção. Este conceito tem como objetivo realçar a definição prévia de classes especialmente desenvolvidas pelo projetista do framework para auxiliar o reutilizador a obter uma instância do artefato.

Mais uma vez recorrendo a especificação do aspecto de persistência presente no DTFrame é possível destacar um conjunto de classes que ilustra claramente esta situação (Figura 31). Este conjunto de classes especifica que para obter persistencia, o reutilizador tem que desenvolver um mapeamento entre o formato OO dos dados da aplicação e o formato do banco de dados. Este mapeamento esta integralmente baseado no protocolo de armazenamento (OO, Relacional, Textual, etc..) utilizado pelo SGBD.

Em uma situação como esta, seria extremamente confortável para o reutilizador ter como opção válida para o processo de reutilização, a escolha de um mecanismo de persistência que esteja completamente desenvolvido no design do framework. Vale lembrar que o objetivo de um framework é proporcionar reutilização para melhorar custos, tempo e qualidade do processo de desenvolvimento de software. Sendo assim, a representação de tal

facilidade para o reutilizador é de extrema importância, pois indica que o projetista do framework proporcionou atalhos para algumas situações de reutilização.

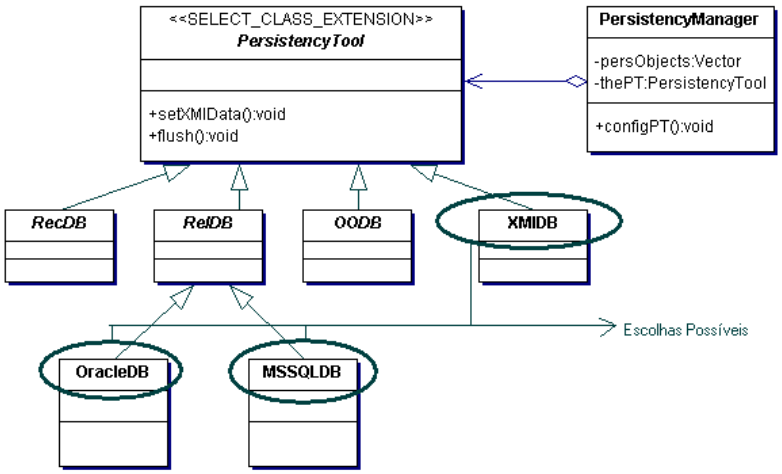


Figura 31 Representação das escolhas possíveis de uma extensão por seleção.

Em UML, a especificação de tal facilidade seria através do estereótipo SELECTION_CLASS_EXTENSION (Figura 31). Note que somente as classes concretas podem ser selecionadas uma vez que a escolha de uma classe abstrata teria que seguir a especificação de extensão simples.

Um outro elemento importante para o processo de reutilização de um design OO é o método. A reutilização de um método se dá através de sua redefinição. Esta redefinição proporciona a introdução de características funcionais necessárias à aplicação em desenvolvimento. Redefinição de métodos são comumente utilizadas para completar a especialização de uma classe.

Como na reutilização de classes, as atividades de reutilização de métodos podem ser redefinição simples ou por aplicação de um Design Pattern. A redefinição simples é

caracterizada pela introdução de uma operação na subclass, com a mesma assinatura que a existente na superclasse (Figura 32).

No DTFrame, é possível observar que a configuração do aspecto de desenho é feita através da especialização da classe Figure. Uma vez especializada esta nova classe tem que especificar o layout da figura a ser desenhada na aplicação. Este layout é obtido através da redefnição do médodo subClasse.draw()(Figura 32). Para obter uma representação no design desta característica, o método associado devera ser decorado com o estereótipo METHOD_EXTENSION.

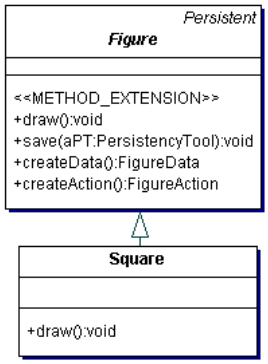


Figura 32 Especificação de redefinição do método draw().

De forma análoga à reutilização de classes, a reutilização de métodos também pode ser feita através da aplicação de um Design Pattern. Tomando como base o exemplo DTFrame com as especializações da classe Figure (classes Circle e Square) previamente desenvolvidas, e a necessidade do reutilizador introduzir o atributo cor nas figuras, surge uma pergunta .

Como utilizar este atributo cor para desenhar as figuras já especificadas?

Para responder esta pergunta é necessário recorrer novamente ao Design Pattern Strategy, que permite introduzir um comportamento novo a um código existente. Este Design Pattern, denominado Strategy, permite ao projetista do framework delegar a estratégia de desenho (que no caso usaria o atributo cor) para o reutilizador.

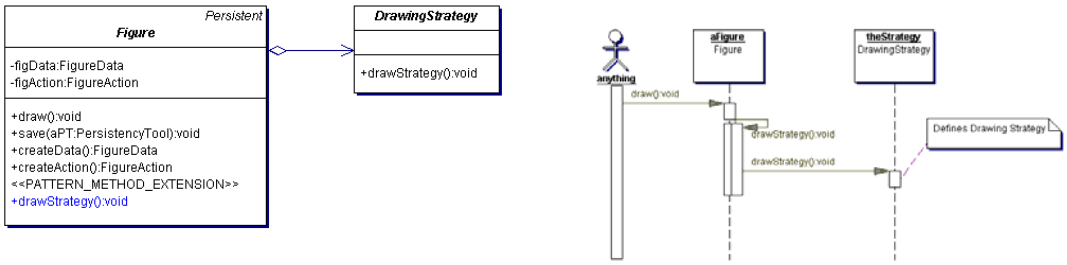


Figura 33 Especificação da redefinição de método via Design Pattern.

A especificação de tal característica em um design UML é obtida com o uso do estereótipo PATTERN_METHOD_EXTENSION.

O último tipo de elemento reutilizável é o atributo. Diferente dos outros tipos, o atributo não esta sujeito a nenhuma atividade de reutilização originada na programação OO e talvez por este motivo não seja objeto de muita atenção por parte das linguagens de documentação de frameworks [Fontoura99] [Froelich97] [Ortigosa00]. Entretanto, quando o artefato reutilizável é um componente[1] (framework caixa-preta muito utilizado por ambientes de desenvolvimento do tipo *drag n´ drop*), a parametrização de um atributo tem um papel de destaque, tornando a especificação de como utiliza-lo de extrema importância.

[1] Em certos ambientes estes atributos são chamados de propriedades.

UML-FI permite a reutilização de um atributo através da especificação de duas atividades:

- atribuição de um valor;
- e seleção e atribuição de um valor.

A especificação da atividade de atribuição tem como objetivo evitar o surgimento de estados inconsistentes durante a execução da aplicação. Estes estados inconsistentes são originados quando a semântica da aplicação gerada não está de acordo com a semântica original do framework. Isto ocorre quando um, ou um conjunto de atributos não contiverem o valor esperado (um valor não nulo, por exemplo), e são utilizados durante a execução da aplicação. Para evitar tal situação, o reutilizador deve definir um valor válido e tipo-compatível com o atributo.

Retornando ao aspecto persistência presente no framework DTFrame é possível observar a definição de um atributo denominado DrawingTool.autoPersistencyRate, que tem como objetivo definir a taxa de salvamento automático da aplicação. A atividade de reutilização deste framework deve preencher este atributo com um valor inteiro. Caso o reutilizador não especifique este valor, será assumido um valor default, que dependendo da linguagem de programação em que a aplicação for implementada pode ser qualquer inteiro válido. Obviamente esta situação pode levar a um estado inconsistente caso este valor seja 0, por exemplo.

Para especificar esta necessidade em UML-FI é utilizado o estereótipo VALUE_ASSIGNMENT como representado na figura 34 atributo DrawingTool.autoPersistencyRate.

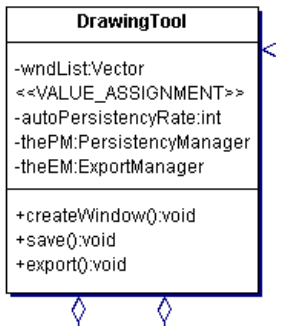


Figura 34 Especificação de uma extensão por atribuição de valor.

Em certos casos a especificação da obrigatoriedade da atribuição não é suficiente para resolver o problema da atribuição. Isto pode ser verificado quando o tipo do atributo tem uma gama muito ampla de valores válidos, tornando difícil a sua determinação por parte do reutilizador. No caso exemplo, o atributo DrawingTool.autoPersistencyRate é do tipo inteiro. Atributos deste tipo podem assumir valores que além de depender da plataforma de implementação, podem ser negativos. No caso o atributo DrawingTool.autoPersistencyRate, a atribuição de um valor negativo fere a semântica dada ao atributo (tempo de …) e certamente levará a aplicação a um estado inconsistente.

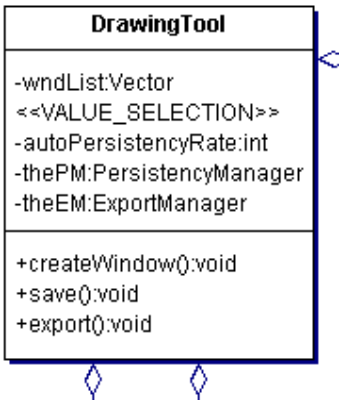


Figura 35 Especificação de extensão por seleção de valor.

Para minimizar este problema, UML-FI introduz a especificação de uma atividade de atribuição a partir da seleção de um conjunto de valores pré-definidos pelo projetista do framework. Esta representação facilita o processo de reutilização, pois como na extensão de classes por seleção, indica com bastante clareza a semântica idealizada pelo projetista do artefato. A forma de representação deste elemento de reutilização é através do estereótipo VALUE_SELECTION como representado na figura 35.

5.2 Integração com o Meta-Modelo de UML

Como apresentado anteriormente, UML-FI estende UML de modo a possibilitar a representação de elementos de destaque para o processo de reutilização. Esta extensão é representada através dos mecanismos básicos de extensão já disponibilizados em UML, como descrito no capítulo 3. Entretanto, para uma perfeita integração desta extensão, é necessária sua especificação junto ao meta-modelo que define UML, de forma a delimitar seu escopo de aplicação, uma vez que esta extensão introduz uma semântica especial quando utilizada.

O meta-modelo de UML é descrito de maneira semiformal utilizando-se três tipo de visão:

- ⇒ Sintaxe Abstrata – Descreve a sintaxe dos elementos de UML utilizando um diagrama de classes e linguagem natural.
- ⇒ Regras de boa formação – Descreve restrições de formação utilizando OCL (Object Constraint Language) e linguagem natural.
- ⇒ Semântica – Descreve o significado do elemento utilizando linguagem natural.

Tomando como base esta forma de descrição, o primeiro passo a ser feito é introduzir os elementos reutilizáveis de UML-FI na sintaxe abstrata de UML. Para tal, é necessário recorrer representação OO destes elementos como apresentado na figura 36.

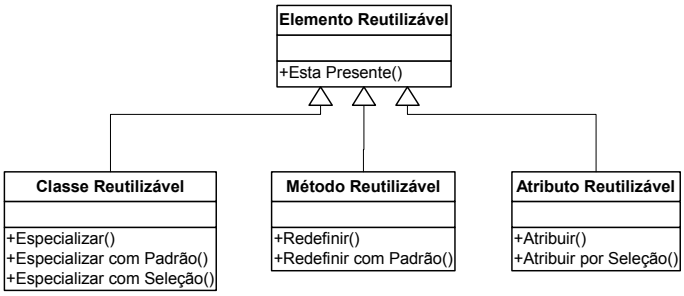


Figura 36 Representação de elementos reutilizáveis como um Diagrama de Classes.

Claramente os elementos reutilizáveis do tipo Classe, Método e Atributo têm que se relacionar com os meta-elementos Class, Method e Attribute respectivamente. Este relacionamento deverá prover a semântica necessária para que estes meta-elementos possam ser classificados como reutilizáveis. Uma vez que os diagramas presentes no meta-modelo de UML são meramente lógicos [Booch99], isto é, não são feitos para serem implementados, é possível agregar funcionalidade as meta-classes já existentes (classes do meta-elemento) através de herança múltipla. Esta técnica permite que hierarquia de classes

que representam elementos do modelo seja preservada, mantendo-se assim a semântica inicial destes meta-elementos (Ex: O meta-elemento Class continua com suas mesmas atribuições no nível de Design, porém com uma conotação específica para o processo de reutilização). Vale ressaltar que esta técnica é utilizada na própria definição do meta-modelo de UML. Como resultado desta integração, a visão da sintaxe abstrata ficaria como apresentado na figura 37.

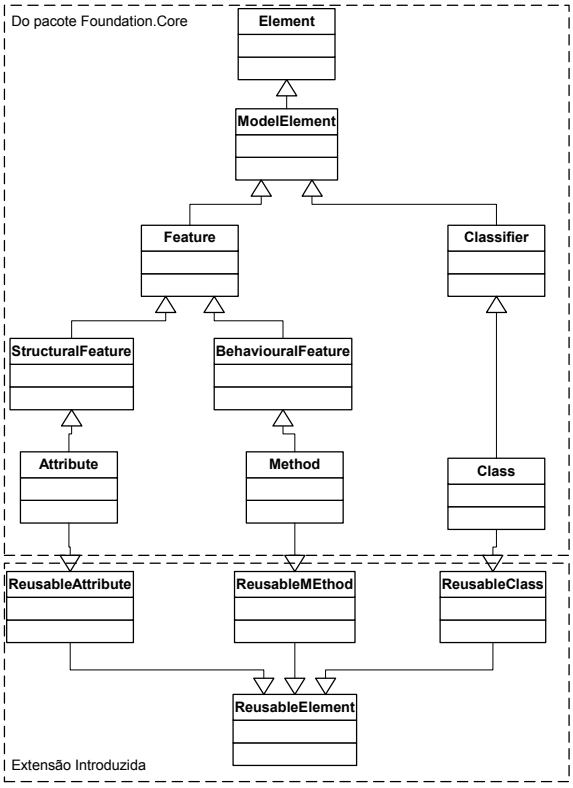


Figura 37 Incorporação ao metamodelo de UML.

Uma vez introduzido no diagrama é necessária uma representação em linguagem natural de cada elemento (Tabela 3). Para manter um maior compatibilidade os nomes dos elementos foram introduzidos em inglês.

Tabela 4 - Representação dos meta-elementos introduzidos por UML-FI.

ReusableElement -	Raiz da hierarquia de elementos reutilizáveis. Identifica um elemento reutilizável.	
Atributo		
Presence	Lógico Especifica se o elemento foi escolhido ou não. Valor default Verdadeiro.	
Valor Identificado		
Required	Indica se o elemento é obrigatório ou não no design final. Valores possíveis (OPTIONAL e MANDATORY). Sendo que MANDATORY é o valor default e preserva a semântica original da meta-elementos.	
ReusableClass -	Identifica uma classe reutilizável.	
Estereótipo		
↔class_extension≈	Utilizado quando a classe em questão necessita redefinição.	
↔pattern_class_extension≈	Utilizado quando a classe em questão é redefinida via Design Pattern.	
↔select_class_extension≈	Utilizado quando a classe em questão é redefinida via seleção.	
ReusableMethod -	Identifica um método reutilizável.	
Estereótipo		
↔method_extension≈	Utilizado quando o método em questão necessita ser redefinido.	
↔pattern_method_extension≈	Utilizado quando o método em questão necessita redefinição via Design Pattern.	

ReusableAttribute - Identifica um atributo reutilizável.	
Estereótipo	
↔value_assignement≈	Utilizado quando o atributo necessita ser preenchido com algum valor.
↔value_selection≈	Utilizado quando o atributo tem sua atribuição especificada por uma faixa de valores.

O segundo passo para a integração ao meta-modelo é a definição das regras de formação (Tabela 4).

ReusableElement – Não adiciona regra

ReusableClass –

Se uma classe é opcional, seus métodos e atributos também são opcionais.

Expressão OCL

⇒ not self.presence **implies** self.allAttributes -> forAll (att | not att.presence)

⇒ not self.presence **implies** self.allMethods -> forAll (met | not met.presence)

ReusableMethod –

Se um método é opcional, as classes que modelam seus parâmetros também são opcionais caso não exista nenhuma outra referencia de obrigatoriedade a esta classe.

Expressão OCL

⇒ not self.presence **implies self.allParameters -> forAll (par | not par.type.allInstances.presence implies not type.presence)**

Se um método não é opcional, as classes que modelam seus parâmetros também não são opcionais.

Expressão OCL

⇒ self.presence **implies self.allParameters -> forAll (par | type.presence)**

ReusableAttribute

Se um atributo é opcional, a classe que modela este atributo também é opcional, caso não exista nenhuma outra referencia de obrigatoriedade a esta classe.

Expressão OCL

⇒ not self.presence **implies (type | not self.type.allInstances.presence implies not type.presence)**

Se um atributo não é opcional, a classe que modela este atributo também não é opcional.

Expressão OCL

⇒ self.presence **implies (type | type.presence)**

A última visão necessária para integração dos modelos é a descrição semântica. Uma vez que esta descrição é representada em linguagem natural e é a mesma da apresentada na seção 5.1 () não se faz necessário sua repetição.

5.3 Resumo

O processo de reutilização tem como objetivo facilitar o desenvolvimento de aplicações de um modo geral. Para que este objetivo seja atingido, o artefato reutilizável deve ser bem documentado para propiciar um rápido entendimento das tarefas envolvidas neste processo.

Com este objetivo, foi elaborado uma extensão de UML, a UML-FI (UML- Framework Instantiation). Esta extensão tem como objetivo representar de forma clara os elementos que estão diretamente envolvidos no ponto de reutilização. Esta extensão levou a introdução de oito novos construtores em UML representados na tabela abaixo.

Tabela 5 Resumo da especificação em UML-FI.

Aplica-se a	Extensão a UML	Significado
Elemento	Valor Identificado REQUIRED = OPTIONAL	Permite a escolha de um elemento em um Design.
Classe	Estereótipo CLASS_EXTENSION	Determina a especialização de uma classe.
Classe	Estereótipo PATTERN_CLASS_EXTENSION	Determina a especialização de uma classe com o uso de um Design Pattern.
Classe	Estereótipo SELECT_CLASS_EXTENSION	Determina a especialização de uma classe através da seleção de uma das suas subclasses concretas.
Método	Estereótipo METHOD_EXTENSION	Determina a redefinição do método nas subclasses da classe onde a marcação se encontra.
Método	Estereótipo PATTERN_METHOD_EXTENSION	Determina a redefinição do método através da aplicação de um Design Pattern.
Atributo	Estereótipo	Determina a atribuição de um valor a

	VALUE_ASSIGNEMENT	um atributo.
Atributo	Estereótipo VALUE_SELECTION	Determina a atribuição de um valor a um atributo através da seleção em um conjunto de valores possíveis.

A notação UML-FI representa o processo de reutilização sob o ponto de vista estático de um diagrama de classes. Esta representação permite uma rápida visualização, identificação e entendimento dos pontos de flexibilização presentes no artefato. Entretanto, esta representação não esta completa uma vez que alguns itens foram omitidos para não prejudicar a legibilidade do design. Esta omissão diz respeito, por exemplo, aos valores envolvidos em uma escolha ou a identificação do Design Pattern utilizado. Um outro ponto omitido em UML-FI é a seqüência em que as atividades de reutilização devem ocorrer. Especificar tal seqüência em um diagrama seria como programar um sistema inteiro de forma visual em algum tipo de notação como um diagrama de seqüência ou colaboração. Esta idéia fere o propósito de tais diagramas uma vez que estes não são computacionalmente completos.

6 Especificação do Processo de Reutilização

O passo seguinte para o desenvolvimento de uma aplicação baseada em frameworks é a representação desta aplicação de acordo com as premissas estabelecidas pelo design (em UML-FI) do artefato. Para tal, o reutilizador deve preencher os pontos de extensão / flexibilização existentes com as características especificas desta aplicação. Este preenchimento deve seguir uma ordem pré-estabelecida pelo projetista do framework, uma vez que certos pontos de extensão podem depender de outros.

Tendo isto em mente, o processo de instanciação (ou processo de reutilização) deve ser representado pelo projetista do framework, através da especificação de uma seqüência de atividades, parâmetros pertinentes e restrições envolvidas que auxiliam o reutilizador a obter o design da aplicação final. Vale ressaltar que esta representação tem como ponto central os elementos reutilizáveis definidos no capítulo anterior.

Por definição , a especificação de uma seqüência de atividades acompanhadas de informações pertinentes é denominada programa, sendo assim, a tarefa desempenhada pelo projetista do artefato após a representação em UML-FI é a programação do processo de instanciação.

Uma possibilidade para a representação deste programa seria através da utilização da notação UML e seus diagramas comportamentais como proposto por [Fontoura99]. Esta abordagem utiliza uma extensão dos diagramas de colaboração para especificar um ”programa de instanciação”. Um problema que surge com esta abordagem é relativo a

especificação de um diagrama que possa representar completamente as atividades de programação como iterações, condicionais e etc.

Para não incorrer em problemas de visualização, este trabalho define uma linguagem de domínio (Domain Specific Language - DSL) [Hudak96] que representa as atividades pertinentes ao domínio de instanciação de frameworks de forma textual. Esta representação é um programa de instanciação, podendo ser visto como um CookBook [Krasner88], e sua execução pode ser feita de forma manual ou assistida (como será mostrado no capítulo 7). A esta DSL foi dado o nome RDL (Reuse Description Language).

Uma visão superficial de RDL permite observar que esta linguagem possibilita a especificação de cookbooks (programas de instanciação) com as seguintes características.

⇒ **Declaração de iterações**. Permite a especificação de atividades repetitivas.

⇒ **Declaração de variáveis**. Permite guardar valores originados em uma atividade de instanciação.

⇒ **Dependência de ordem de uma ação.** Permite a especificação de uma dependência do tipo INSTANCIAÇÃO A vem antes de INSTANCIAÇÃO B.

⇒ **Dependência de um estado futuro.** Permite a especificação da obrigatoriedade ou não, de um elemento no design final para que a instanciação em questão possa ocorrer.

⇒ **Declaração de atividades de instanciação paralelas**. Permite a especificação de execução de atividades de instanciação disjuntas.

⇒ **Declaração de comentários.** Permite a introdução de texto em linguagem natural.

⇒ **Especificação dos pontos de extensão presentes em UML-FI**. Permite a especificação completa dos pontos de extensão definidos em UMF-FI.

⇒ **Declaração de receitas rotuladas**. Permite a definição de um conjunto de atividades afins em um único bloco identificando-o com um nome. Este bloco pode ser chamado como se fosse uma rotina em linguagens de programação imperativas.

6.1 A BNF Comentada

A especificação de RDL é introduzida através da descrição de sua BNF para expressar a sintaxe, e comentários em linguagem natural para especificar a semântica. Vale ressaltar que esta BNF tem como objetivo esboçar a sintaxe da linguagem e deve ser aprimorada para a construção de um compilador/interpretador.

COOKBOOK ::=	**cookbook** NAME IP_RECIPE* IP_MAIN **end_cookbook;**

Um programa de instanciação expresso em RDL começa com a definição de um nome (NAME) que deve ser igual ao nome do arquivo onde este programa é armazenado. Após este nome, segue um conjunto de receitas de instanciação (IP_RECIPE) que definem as atividades de instanciação propriamente ditas.

IP_RECIPE ::=	[COMMENT_EXP] **recipe** IP_NAME **;** IP_RECIPE_BODY **end_recipe**;

Cada receita de instanciação define uma parte significativa do programa de instanciação e identifica um conjunto de atividades afins através de um nome (IP_NAME).

IP_RECIPE_BODY ::=	IP_CMD*

O corpo de uma receita é composto por conjunto de atividades de instanciação (IP_CMD). Não é permitido aninhamento de receitas.

IP_MAIN ::=	**recipe main** IP_RECIPE_BODY **end_recipe;**

Declaração da recita principal por onde o programa de instanciação inicia.

IP_NAME ::=	NAME

String válida que identifica o bloco de atividades.

IP_CMD ::=	IP_ASSIGN_CMD ;| IP_EXP_CMD; | IP_LOOP_CMD;

Um comando pode ser uma atribuição (IP_ASSIGN_CMD), uma expressão (IP_EXP_CMD) ou um comando de repetição (IP_LOOP_CMD).

IP_ASSIGN_CMD::=	NAME **=** IP_BASIC

Declaração de uma variável. RDL é uma linguagem fracamente tipada e suas variáveis assumem automaticamente o tipo do elemento do lado direito da atribuição. EM RDL estes tipos podem ser Classe, Método ou Atributo e normalmente são utilizados para referenciar um elemento introduzido no design durante o processo de instanciação.

IP_LOOP_CMD ::=	**Loop** IP_RECIPE_BODY **end_loop**

Permite e especificação de atividades repetitivas. Nesta versão de RDL a parada da repetição é identificada pelo reutilizador, ou seja, faça enquanto necessário.

IP_EXP_CMD ::=	IP_TASK | IP_TASK # IP_TASK | IP_TASK o IP_TASK | IP_TASK || IP_TASK

Especifica uma expressão podendo ser:

⇒ IP_TASK – Especifica a execução de uma atividade comum.

⇒ IP_TASK # IP_TASK – Especifica escolha de atividades, ou seja, apenas uma das atividades será executada. A escolha será especificada pelo reutilizador.

⇒ IP_TASK o IP_TASK – Especifica seqüência de atividades, ou seja, ambas atividades serão executadas em seqüência.

⇒ IP_TASK || IP_TASK – Especifica paralelismo entre as atividades, ou seja, as atividades podem ser executadas concorrentemente. Utilizado para representar reutilização distribuída.

IP_TASK ::=	[COMMENT_EXP] IP | [COMMENT_EXP] IP_NAME

Uma atividade de instanciação pode ser simples ou uma chamada a uma receita.

IP ::=	IP_BASIC REQUIRES_EXP*

Uma atividade comum é uma atividade básica (IP_BASIC) que pode ser seguida de uma expressão de necessidade.

IP_BASIC ::=	IP_CLASS | IP_METHOD | IP_ATTRIBUTE | IP_ELEMENT

Especifica a atividade de instanciação propriamente dita podendo ser:

⇒ IP_CLASS – A atividade de instanciação é referente a uma classe.

⇒ IP_METHOD – A atividade de instanciação é referente a um método.

⇒ IP_ATTRIBUTE – A atividade de instanciação é referente a um atributo.

⇒ IP_ELEMENT – A atividade de instanciação é referente a um elemento genérico.

Esta atividade de instanciação esta diretamente relacionada com os marcadores descritos em UML_FI.

IP_CLASS ::=	**class_extension** (CLASS_EXP) | **selection_class_extension** (CLASS_EXP) | **pattern_class_extension** (CLASS_EXP , NAME,LIST)

Especifica o tipo de atividade instanciação a ser executada sobre uma classe.

⇒ **class_extension** (CLASS_EXP) – Será criada uma nova classe que especializa CLASS_EXP. O nome desta classe nove é especificado pelo reutilizador.

⇒ **selection_class_extension** (CLASS_EXP) - Será escolhida uma subclasse concreta de da classe CLASS_EXP. Esta seleção é feita pelo reutilizador.

⇒ **pattern_class_extension** (CLASS_EXP , NAME, LIST) – Será criada um conjunto de elementos que promovem a especialização da classe CLASS_EXP. Este conjunto de elementos é decorrente do Design Pattern (NAME) e lista de parâmetros (LIST) a ser utilizado na extensão.

IP_METHOD ::=	**method_extension** (CLASS_EXP, CLASS_EXP, METHOD_EXP) |

	pattern_method_extension (CLASS_EXP, CLASS_EXP, METHOD_EXP , NAME, LIST)

Especifica o tipo de atividade de redefinição a ser executada sobre um método.

⇒ **method_extension** (CLASS_EXP, CLASS_EXP, METHOD_EXP) – Redefine o método METHOD_EXP presente na primeira classe (CLASS_EXP), introduzindo este método na segunda classe.

⇒ **pattern_method_extension** (CLASS_EXP, CLASS_EXP, METHOD_EXP , NAME, LIST) – Redefine o método METHOD_EXP aplicando o Design Pattern NAME e lista LIST.

IP_ATTRIBUTE ::=	**value_selection** (CLASS_EXP, ATTRIB_EXP, LIST) | **value_assignment** (CLASS_EXP, ATTRIB_EXP)

Especifica o tipo de atividade de atribuição a ser executada sobre um atributo.

⇒ **value_selection** (CLASS_EXP, ATTRIB_EXP, LIST) - Permite o preenchimento do atributo ATTRIB_EXP presente na classe CLAS_EXP através da escolha de um valor em LIST.

⇒ **value_assignment** (CLASS_EXP, ATTRIB_EXP) - Permite o preenchimento do atributo ATTRIB_EXP presente na classe CLASS_EXP com um valor livre.

IP_ELEMENT::=	**element_choice** (ELEMENT)

Especifica a atividade de escolha de um elemento.

ELEMENT::=	CLASS_EXP | METHOD_EXP | ATTRIB_EXP

Os elementos em RDL podem ser Classes (CLASS_EXP), Métodos (METHOD_EXP) ou atributos (ATTRIB_EXP).

CLASS_EXP ::=	NAME

Especifica um nome válido para uma classe.

METHOD_EXP ::=	NAME

Especifica um nome válido para um método.

ATTRIB_EXP ::=	NAME

Especifica um nome válido para um atributo.

LIST::=	(LIST_EXP)

Especifica uma lista.

LIST_EXP::=	STRING_EXP , LIST_EXP | STRING_EXP

Uma lista pode conter um ou mais nomes válidos.

NAME::=	STRING_EXP

Um nome válido é uma string.

REQUIRES_EXP::=	**requires** ORDER_EXP | **requires** ELEMENT

Especifica a cláusula de dependência de uma atividade (ORDER_EXP) ou elemento (ELEMENT)

ORDER_EXP::=	**before** IP_TASK | **after** IP_TASK | **sync** IP_TASK | **exclusive** IP_TASK

Especifica a ordem da dependência:

Before – Especifica que a atividade corrente tem que acontecer antes de IP_NAME.

after – Especifica que a atividade corrente tem que acontecer após IP_NAME.

Sync – Especifica que a atividade corrente tem que acontecer sincronizada com IP_NAME.

Exclusive – Especifica que a atividade corrente é mutuamente exclusiva a IP_NAME.

COMMENT_EXP ::=	// STRING_EXP

Especifica um comentário.

STRING_EXP ::=	**String**

Especifica uma string comum.

6.2 O Desenvolvimento do Cookbook

A programação de sistemas é vista como uma tarefa criativa uma vez que executa uma transformação vertical [Fontoura99] entre uma especificação de alto nível (Diagramas UML, Diagramas de Features[Kang93], etc.) e uma linguagem de programação qualquer (Figura 38). Esta criatividade é responsável, por exemplo, pela definição das seqüências de ações necessárias para cobrir um requisito especificado em um use-case [Jacobson94]. O desenvolvimento de um cookbook em RDL também exige uma dose de criatividade, porém em menor escala. Esta atenuação é atribuída à integração existente entre as especificações UML-FI e RDL que pode ser verificada pela representação em RDL de todos os marcadores introduzidos por UML-FI, facilitando assim a representação das atividades de reutilização a serem executadas.

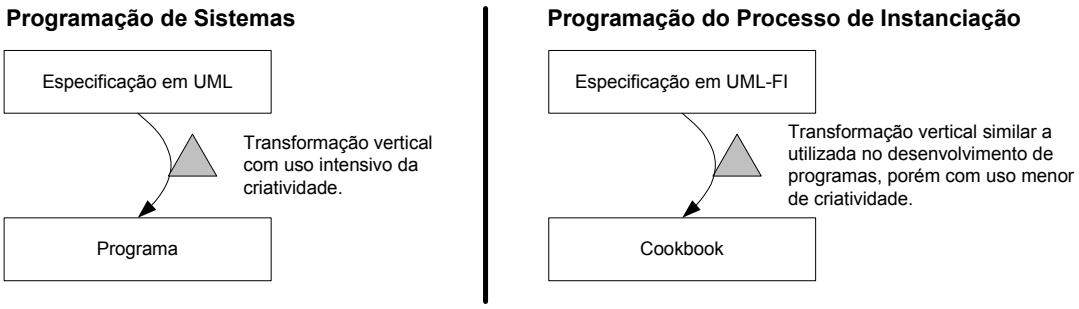


Figura 38 Obtenção do Cookbook.

Tendo isto em mente é possível afirmar que a principal tarefa do desenvolvedor de um cookbook é organizar estas atividades de modo a facilitar ao máximo o processo de reutilização. Esta organização visa agrupar tarefas afins em receitas, especificar restrições de instanciação, especificar execução paralela e definir um programa principal que governa o processo de instanciação como um todo. Vale ressaltar que o cookbook desenvolvido deve

representar todos os pontos de flexibilização/extensão expressos nos diagramas de classes do artefato em questão.

Para facilitar ainda mais o desenvolvimento de um cookbook as atividades de reutilização definidas em RDL são classificadas tomando como base a complexidade de representação e/ou percepção por parte do desenvolvedor. Estas atividades são em ordem crescente de complexidade: básicas, estendidas, restritivas ou organizacionais.

As atividades básicas têm como objetivo representar técnicas básicas de programação orientada a objetos. Estas atividades têm sua especificação extraída diretamente do diagrama UML-FI e representam o núcleo do processo de instanciação, podendo ser: escolha de elemento, especialização de classe, redefinição de método e atribuição de um valor. Em RDL estas atividades são representadas pelas palavras reservadas: **element_choice, class_extension, method_extension, value_assignment.**

O objetivo das atividades estendidas é aumentar o vocabulário definido pela POO. Esta extensão visa proporcionar o uso de atividades especiais para aprimorar a qualidade do design da aplicação final. Estas atividades também podem ser extraídas diretamente de um diagrama UML-FI, porém em alguns casos necessitam de informação complementar. Atividades como aplicação de Design Pattern, Especialização por Seleção e Atribuição por Seleção são exemplos de tais atividades. Em RDL estas atividades são representadas pelas palavras reservadas: **pattern_class_extension, selection_class_extension, pattern_method_extension, value_selection**.

As atividades restritivas especificam alguma dependência na execução das atividades. Esta dependência pode ser de ordem ou de estado futuro. A dependência de estado futuro especifica a necessidade de presença de um determinado elemento no design final para que a ação em execução seja pertinente. Já a dependência de ordem especifica como uma atividade de instanciação se comporta em relação à outra, ou seja, se é executada antes, depois, em exclusão mútua ou em sincronia com outra atividade. Em RDL estas atividades são representadas pelas palavras reservadas: **requires**, **before, after, exclusive, sync**. Vale ressaltar que o script do cookbook já define uma seqüência de execução natural.

As atividades organizacionais visam melhorar a legibilidade e, por conseguinte a exeqüibilidade de um cookbook. Esta melhoria é obtida com o uso de receitas, especificação de atividades paralelas e declaração de variáveis para evitar aninhamento de atividades. Em RDL estas atividades são representadas pelas palavras reservadas: **recipe, ||, =.**

Uma vez definido este agrupamento de tipos de atividades, o desenvolvedor de um cookbook pode se concentrar em um determinado tipo de atividade e utilizar a abordagem de desenvolvimento evolutivo-incremental [Pressman00] como técnica de aprimoramento. Para tal, este desenvolvedor deve primeiramente especificar completamente as atividades básicas e estendidas que representam a base do processo de instanciação. Depois, seria necessário capturar e representar as restrições na execução das atividades especificadas. Neste ponto o cookbook já possui toda a funcionalidade necessária a reutilização do artefato, entretanto, como em linguagens de programação imperativas, se faz necessário um passo de organização destas atividades de modo a racionalizar sua execução.

É importante mencionar que a utilização da abordagem evolutivo-incremental aprimora o produto final através de melhorias em todos os níveis de representação podendo em alguns casos afetar o design original em UML-FI (Figura 39).

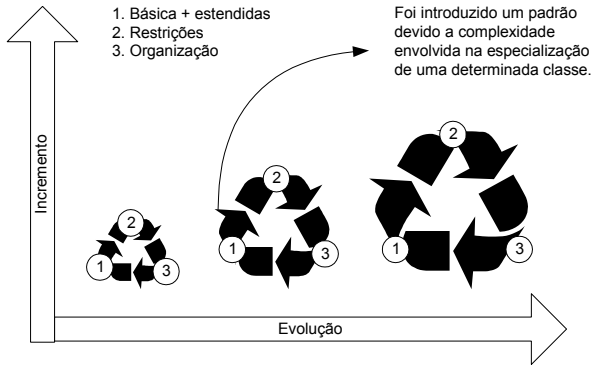


Figura 39 Abordagem evolutivo incremental para o desenvolvimento do Cookbook.

6.3 Execução de um Cookbook

O principal objetivo da execução de um cookbook é modificar o design de um artefato reutilizável, expresso em UML-FI, de modo a obter uma representação em UML padrão da aplicação a ser desenvolvida. Nesta seção serão apresentados em mais detalhes os resultados da execução dos principais operadores e comandos presentes em RDL.

Operador Atribuição/Alocação -> A = B

Execução – A execução deste tem as seguintes finalidades; 1) Alocar a variável representada por A em um repositório de variáveis, caso não exista. Caso exista, seu conteúdo é apagado e seu espaço reutilizado. 2) Armazenar o resultado da execução da atividade B em A. Este resultado é o nome válido de um elemento.

Exemplo – novoAnimal = CLASS_EXT(Animal). Pela figura (40) abaixo é possível verificar que é criada uma nova classe, cujo nome é armazenado na variável novoAnimal.

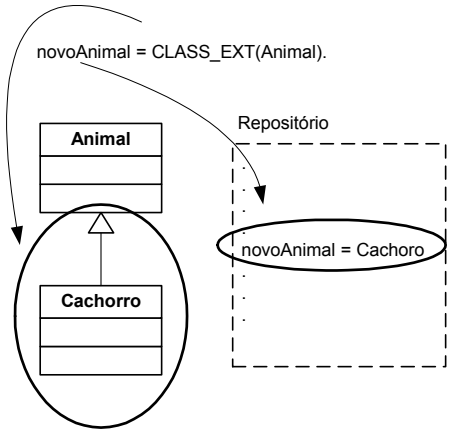


Figura 40 Alocação da variável figClass a sua respectiva atribuição.

Operador OU – A # B

Execução – O reutilizador escolhe uma entre duas atividades representadas.

Exemplo – SELECT_CLASS_EXTENSION (Animal) # CLASS_EXTENSION (Animal). Especifica a especialização da classe Animal por seleção ou por introdução de uma classe nova.

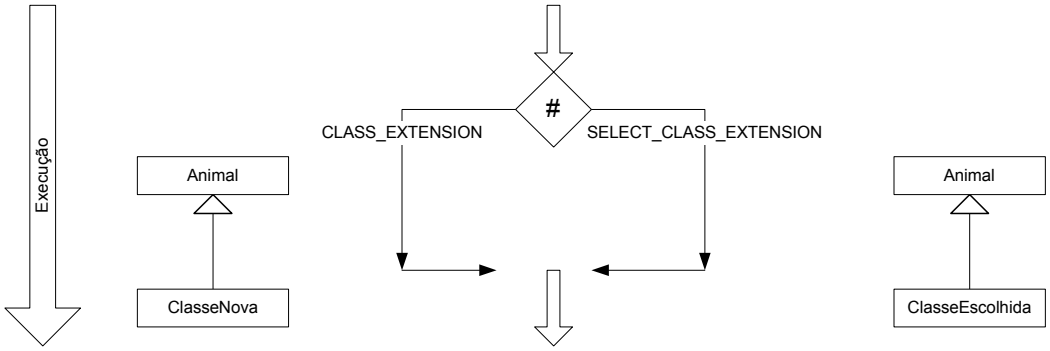


Figura 41 Escolha dentre as atividades.

Operador E – A o B

Execução – O reutilizador executa as duas atividades em seqüência.

Exemplo – ((class_extension (Animal) o class_extension(novoAnimal)) # …..).

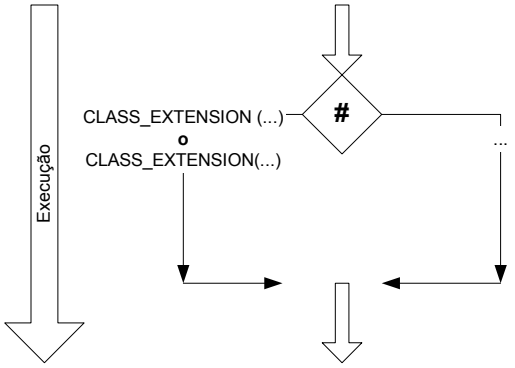


Figura 42 Seqüencialidade na execução.

Operador Paralelo – A || B

Execução – Especifica a possibilidade de execução concorrente. Neste ponto, as atividades A e B podem ser executadas por pessoas ou grupo de pessoas diferentes.

Exemplo - class_extension (Animal) || class_extension (Persistencia).

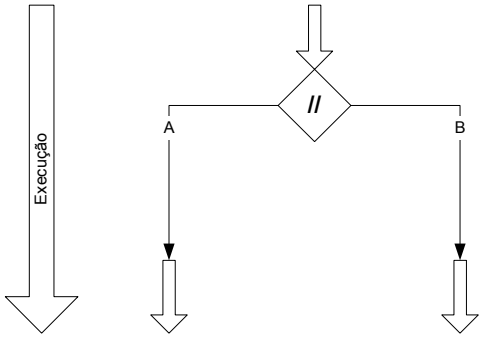


Figura 43 Execução concorrente de A e B.

Comando para especialização de classe - class_extension(A).

Execução – O reutilizador deve criar uma classe nova em seu design dando-lhe um nome válido e inexistente. Esta classe deve ter um relacionamento de herança com a classe A. Retorna o nome da classe criada.

Exemplo – class_extension(Animal).

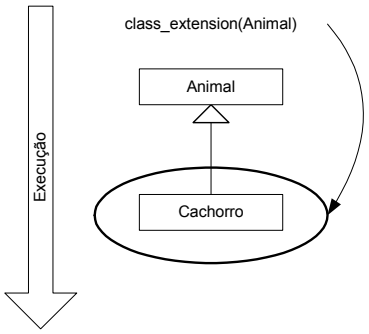


Figura 44 Execução do CLASS_EXTENSION.

Comando para especialização de classe - select_class_extension(A)

Execução – O reutilizador deve escolher uma das subclasses da classe A como a classe que irá permanecer no design. Retorna o nome da classe selecionada.

Exemplo – select_class_extension(Animal)).

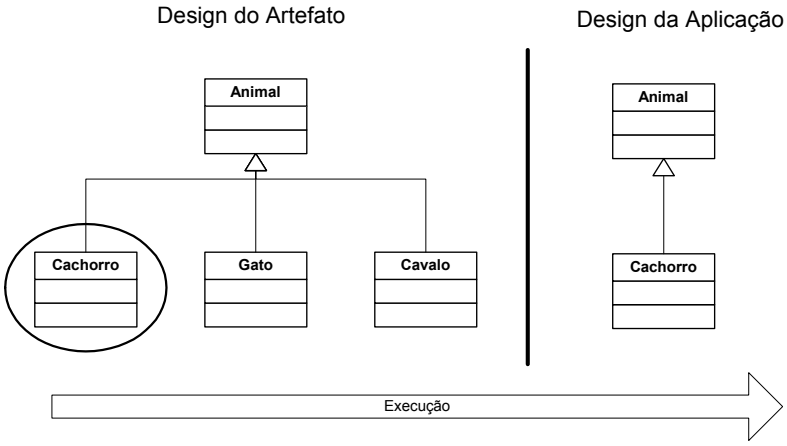


Figura 45 Execução do SELECT_CLASS_EXTENSION.

Comando para extensão de classe com padrão – pattern_class_extension (A,P,(pars)).

Execução – Neste ponto o reutilizador deve aplicar o padrão especificado pelo parâmetro P. A execução deste comando depende de um catálogo de padrões associado a RDL. Cada padrão representado neste catálogo deve expressar as atividades envolvidas na aplicação do padrão utilizando RDL. Este comando retorna o nome da classe estendida.

Exemplo – pattern_class_extension (Homem,Decorator,(ClasseDecoradora,listaBens)). Neste caso o reutilizador aplicará o padrão Decorator cuja execução é especificada como: Estender a classe Homem através da especialização dos tipos de componentes (ClasseDecoradora) que podem ser a agregado a um objeto desta classe e satisfazem ma determinada interface (listaBens).

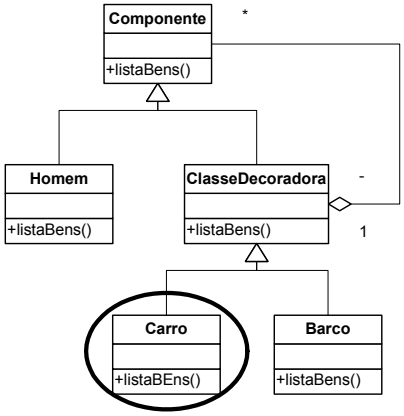


Figura 46 Execução do PATTERN_CLASS_EXTENSION.

Comando para redefinição de método – method_extension(C1,C2, M)

Execução – O reutilizador deverá introduzir o método M na classe C2 de modo que este método redefina o método de mesma assinatura presente em C1.

Exemplo - method_extension (Animal, Cachorro, anda).

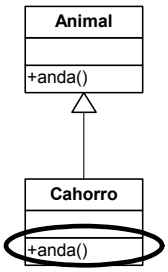


Figura 47 Execução do METHOD_EXTENSION

Comando para redefinição de método via padrão – pattern_method_extension(C1,C2, M,P,(L)).

Execução – O reutilizador deverá utilizar o padrão P para redefinir o método M. A utilização do padrão está condicionada a especificação em um catálogo externo, como na extensão de classes.

Exemplo – pattern_method_extension (Animal,Cachorro,anda,Strategy()). Redefine o método anda(), através da aplicação do padrão Strategy.

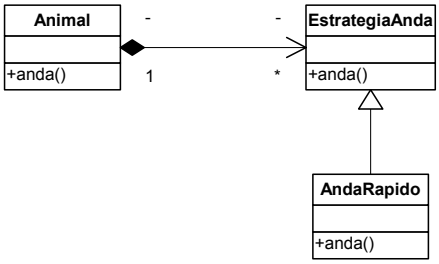


Figura 48 Execução do PATTERN_METHOD_EXTENSION.

Comando para atribuição de valor – value_assignment(C,A)

Execução – Por definição, a atribuição de um valor é uma tarefa que acontece em tempo de execução e não pode ser representada na fase de design de forma adequada. Sendo assim, na execução deste comando, o reutilizador deve simplesmente anexar algum tipo de lembrete junto ao par classe/atributo que sinaliza a obrigatoriedade desta atribuição. Em UML este lembrete poderia ser uma nota.

Exemplo – value_assignment (Animal,peso).

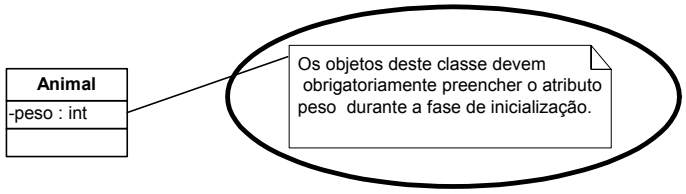


Figura 49 Execução do VALE_ASSIGNMENT

Comando para atribuição de valor por seleção - value_selection(C,A,(LIST))

Execução – O reutilizador deve escolher um valor dentre os listados e proceder como na atribuição simples, anexando um lembrete a classe C.

Exemplo - value_selection(Animal,sexo,(masculino,feminino, hermafrodita))

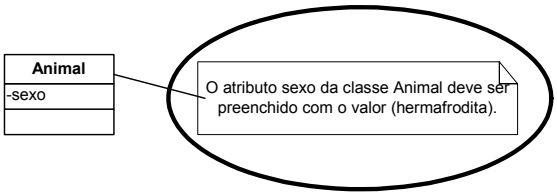


Figura 50 Execução do VALUE_SELECTION.

Operador para especificar condição – A requires (B)

Execução – O reutilizador deve garantir que atividade B seja satisfeita até a obtenção do design final da aplicação de modo a validar a atividade A.

Exemplo – method_extension(Animal,save) requires (System.gerentePersistencia). Salvar os dados de um animal só faz caso o sistema use o módulo de persistência.

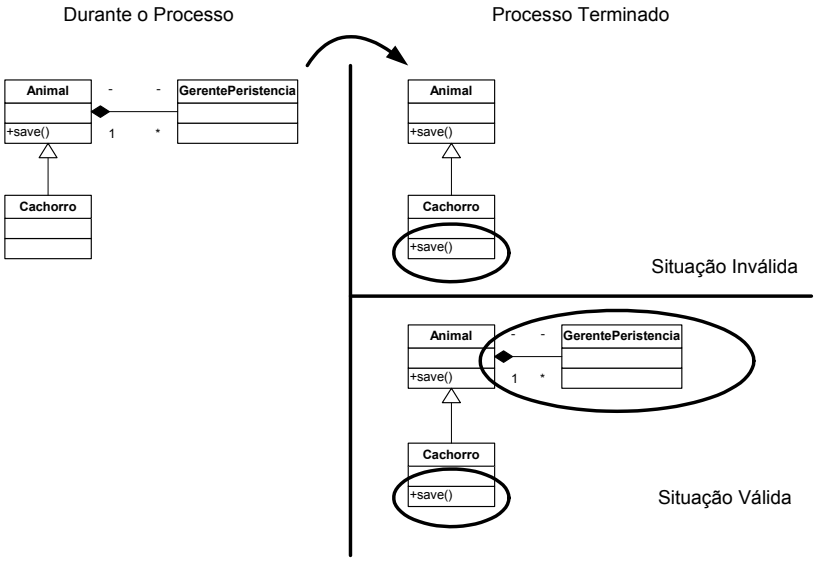


Figura 51 Execução do REQUIRES.

Operador para execução anterior – A requires before B

Execução – Para executar a atividade A o reutilizador tem que antes executar a atividade B.

Exemplo - method_extension(Animal,save) requires before (EscolhaPersistencia) – Para redefinir o método de salvamento é necessário antes saber o tipo SGBD. Este tipo é determinado pela execução da receita EscolhaPersistencia.

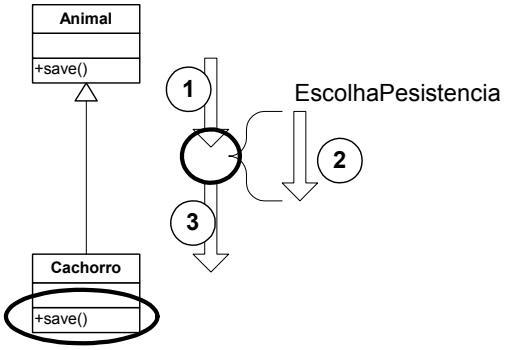


Figura 52 Execução de REQUIRES BEFORE.

Operador para execução posterior – A requires after B

Execução – O reutilizador **tem** que executar a atividade B após A.

Exemplo - EscolhaPersistencia requires after (RedefineSalvamento) – Uma vez definido o tipo de SGBD, é necessário especificar o salvamento (receita RedefineSalvamento).

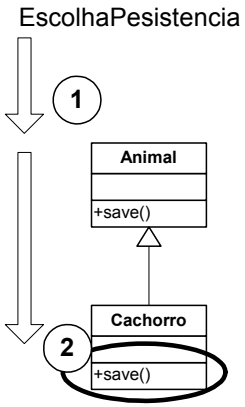


Figura 53 Execução de REQUIRES AFTER

Operador para execução sincronizada – A requires sync B

Execução – O reutilizador tem que garantir que as atividades A e B terminam simultaneamente, ou seja, o processo de instanciação só pode continuar após o témido as duas atividades.

Exemplo – class_extension(Animal) requires sync (EscolhaPersistencia). Neste caso, a instanciação só pode prosseguir após a especialização da classe Animal e a escolha do SGBD.

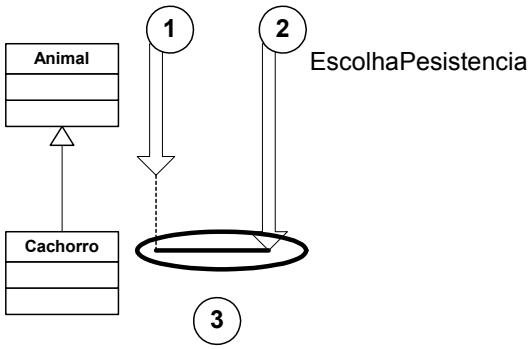


Figura 54 Execução de REQUIRES SYNC.

Operador para execução exclusiva – A requires exclusive B.

Execução – Neste ponto o reutilizador tem que garantir que a atividade B não tenha sido executada para que A possa ser. Caso B tenha sido executada, não executar A.

Exemplo – class_extension(Oracle) requires exclusive class_extension(SQLServer). Neste ponto o reutilizador verifica que só pode especializar uma classe para representar o banco de dados.

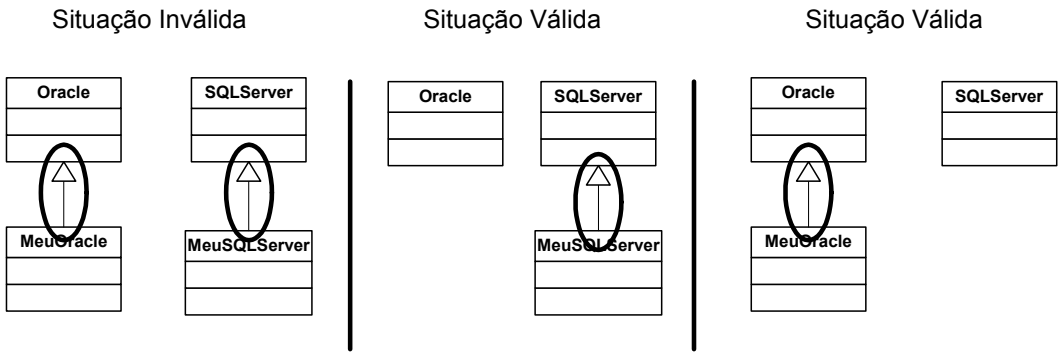


Figura 55 Execução do REQUIRES EXLUSIVE.

6.4 Um Exemplo

Seguindo a ordem de complexidade das atividades de instanciação, um cookbook para o framework DTFrame deve :

1) Expressar as seguintes características:

- ⇒ Instanciação do aspecto desenho,
- ⇒ Instanciação do aspecto exportação e
- ⇒ Instanciação do aspecto persistência,

2) Especificar as restrições:

- ⇒ A especificação da persistência de uma figura depende da presença da capacidade de persistência no design final,
- ⇒ A especificação do tipo de persistência depende da presença da capacidade de persistência no design final e
- ⇒ A especificação do tipo de exportação depende da presença da capacidade de exportação no design final,

3) Organizar as atividades de modo a:

- ⇒ As características mencionadas acima devem ser representadas por receitas,
- ⇒ Os aspectos de persistência e exportação podem ser feitos de forma paralela e
- ⇒ O aspecto de persistência deve ser feito antes instanciação das figuras.
- ⇒ Colocar variáveis para reduzir aninhamento de atividades relacionadas.

A representação destas características de instanciação citada acima poderia ser descrita pelo cookbook a seguir:

```
1  COOKBOOK DTFrameCok
2  //================================================
3    RECIPE FigureAspect;
4      LOOP
5        figClass =  CLASS_EXTENSION  (Figure);
6        figDataClass = CLASS_EXTENSION  (FigureData);
7        figAction = CLASS_EXTENSION  (FigureAction);
8        PATTERN_METHOD_EXTENSION (Figure, figClass, createData , Factory,
   (figClass, createData, figDataClass);
9        PATTERN_METHOD_EXTENSION (Figure, figClass, createAction ,
   Factory, (figClass, createAction, figActionClass);
10       METHOD_EXTENSION (Figure, figClass,save) REQUIRES
   (DrawingWindow.thePersistencyMan);
11     END_LOOP;
12   END_RECIPE;
13 //================================================
14   RECIPE PersistencyAspect;
15     ELEMENT_CHOICE (DrawingWindow.thePersistencyMan);
16     SELECT_CLASS_EXTENSION (PersistencyTool) REQUIRES
   (DrawingWindow.thePersistencyMan) #
17     CLASS_EXTENSION (PersistencyTool)REQUIRES
   (DrawingWindow.thePersistencyMan);
18     VALUE_SELECTION (DrawingTool , autoPersistencyRate,
   (0,5,10,30)) REQUIRES (DrawingWindow.thePersistencyMan);
19   END_RECIPE;
20 //================================================
21   RECIPE ExportAspect;
```

```
22     ELEMENT_CHOICE (DrawingWindow.theExportMan);
23     LOOP
24       SELECT_CLASS_EXTENSION (ExportTool)
   REQUIRES(DrawingWindow.theExportMan) #
25       CLASS_EXTENSION (ExportTool)
   REQUIRES(DrawingWindow.theExportMan);
26     END_LOOP;
27     END_RECIPE;
28 //=================================================
29   RECIPE MAIN;
30     PersistencyAspect ||  ExportAspect;
31     FigureAspect;
32   END_RECIPE;
33 END_COOKBOOK;
```

Analisando o cookbook DTFrameCok é possível verificar que este é organizado em quatro receitas denominadas FigureAspect (Linhas 3 à 12), PersistencyAspect (Linha 14 à 19), ExportAspect (Linhas 21 à 27) e Main (Linha 29 à 32).

A receita FigureAspect especifica a instanciação do aspecto figura. Esta instanciação tem como característica a repetição (Linhas 4 à 11) da especialização das classes pertinentes. Na linha 5 são definidas duas atividades. Seguindo a ordem de execução, a primeira atividade cria uma nova classe que especializa Figure. Em seguida, é criada uma variável denominada figClass que conterá uma referência para nova. Esta variável é necessária para evitar a inserção desta atividade de especialização (CLASS_EXTENSION) na atividade da linha 8. O quadro abaixo mostra o ganho de legibilidade decorrido da introdução da variável figClass.

Tabela 6 Comparação entre a atividade com e sem a declaração da variável.

PATTERN_METHOD_EXTENSION (Figure, CLASS_EXTENSION (Figure), createData , Factory, (figClass, createData, figDataClass);
PATTERN_METHOD_EXTENSION (Figure, figClass, createData , Factory, (figClass, createData, figDataClass);

A linha 8 especifica a redefinição do método createData() através da aplicação do padrão Factory. Para entender a especificação desta atividade é necessário relembrar o padrão Factory presente em [Gamma95]. Este padrão tem como objetivo definir uma interface para criar um determinado objeto, delegando para uma subclasse a definição de seu tipo. É composto por quatro classes:

⇒ Product – Define o tipo abstrato do objeto a ser criado.

⇒ ConcreteProduct – Concretiza Product.

⇒ Creator – Declara o método Factory que retorna um objeto do tipo Product.

⇒ ConcreteCreator – Concretiza Creator redefinindo o método Factory para especificar o tipo produto a ser retornado.

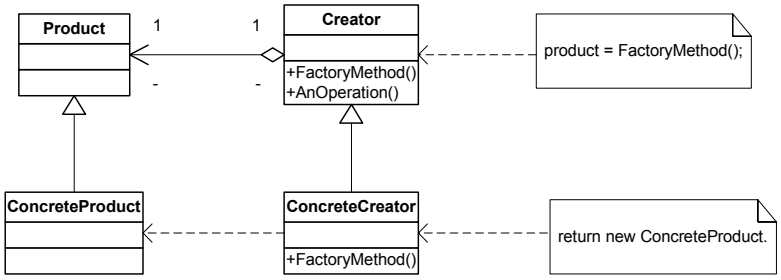


Figura 56 Padrão Factory Method

Voltando a linha 8, os parâmetros Figure, figClass e createData representam respectivamente a superclasse, subclasse e o método a ser redefinido na subclasse. O quarto parâmetro especifica o padrão a ser utilizado, no caso Factory. Em seguida é colocada entre parêntesis, a lista de parâmetros necessários para a aplicação do padrão que para o padrão em questão representam ConcreteCreator, FactoryMethod e ConcreteProduct.

Neste ponto é importante mencionar que a definição do que fazer quando a aplicação de um padrão é especificada em um catálogo externo a RDL. No caso do padrão Factory é introduzido um método FactoryMethod na classe ConcreteCreator e anexado um comentário que especifica o corpo do método introduzido. Após a execução desta linha teríamos (figura 57).

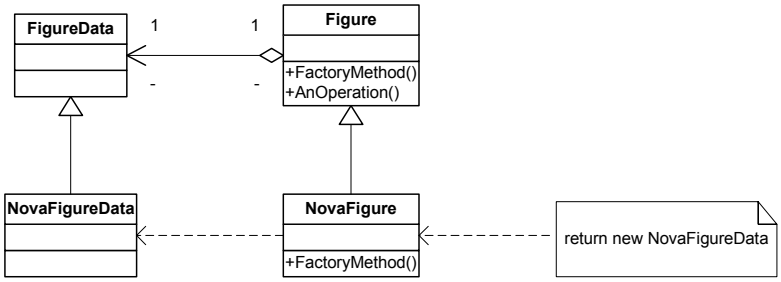


Figura 57 Execução de Factory Pattern.

Ainda na receita FigureAspect é possível observar a especificação de uma restrição na linha 10. Esta restrição indica a necessidade a presença do elemento DrawingWindow. thePersistencyManager no design final.

A receita PersistencyAspect (linha 14) inicia especificando a escolha do elemento DrawingWindow. ThePersistencyManager (Linha 15). Em seguida, há a especificação de uma seleção por extensão (Linha 16). A lista de classes que atendem a seleção é extraída diretamente das subclasses da classe PersistencyTool. Ainda na linha 16 é possível verificar a presença de um sinal ou (#). Este sinal especifica a possibilidade de criação de uma nova classe, caso as subclasses de PersistencyTool não atenda a reutilização. Por fim na linha 18 é especificada a escolha de um valor a ser atribuído no atributo DrawingTool. autoPersistencyRate. Os valores possíveis são 0,5,10,30 e estão especificados no último parâmetro da atividade.

A última receita, Main, especifica a ordem de execução das outras receitas. Na linha 30 Main especifica a execução paralela das atividades definidas pelas receitas PersistencyAspect e ExportAspect. Por fim na linha 31 a receita FigureAspect é invocada. Neste ponto é ressaltada a opcionalidade do comando **after** para especificar que a receita FigureAspect deve ser executada após a receita PersistencyAspect. Isto é devido à seqüência natural imposta pelo cookbook.

Uma vez executado este cookbook é obtido o diagrama abaixo (Figura 58).

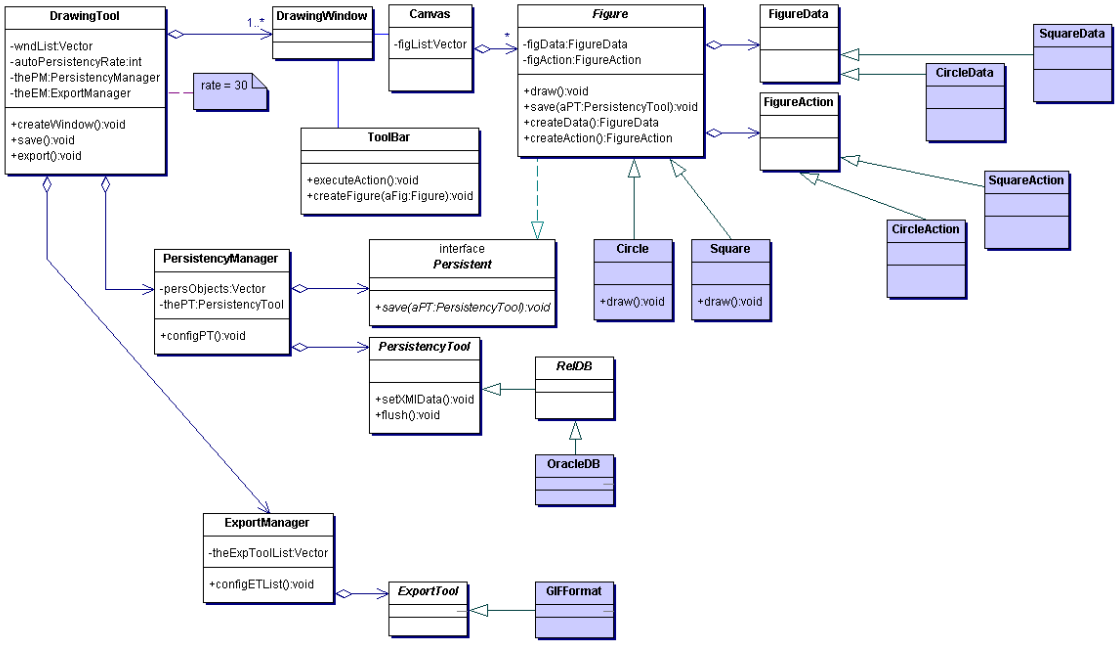


Figura 58 – DTFrame após execução do cookbook de instanciação.

6.5 Resumo

Os diagramas de classes especificados em UML-FI não são suficientes para capturar todas as informações necessárias para a realização da instanciação. Para tal, informações como seqüência, organização e restrição das atividades envolvidas no processo necessitam ser especificadas. Com este objetivo, foi desenvolvida uma DSL denominada Reuse Description Language, ou RDL, que representa de forma textual todos os dados necessários à realização da instanciação.

Esta representação textual dá origem a um programa, denominado cookbook, que serve de guia para a execução da instanciação. E linhas gerais um cookbook é organizado em receitas que especifica determinadas características relevantes para o domínio do artefato reutilizável

a ser instanciado. Estas receitas por sua vez, especificam a seqüência em que atividades básicas como especialização, redefinição e escolha devem ocorrer.

7 A Ferramenta xFIT

O processo de instanciação de um artefato reutilizável pode se tornar uma tarefa árdua. A falta de conhecimento no domínio do artefato e principalmente a complexidade e dependência das tarefas envolvidas no processo, dificultam o desenvolvimento do design final. Neste contexto, "*GUI Builders*" como Borland, VisualBasic, VisualAge facilitam o processo permitindo o desenvolvimento rápido da interface de sistemas, através da configuração de componentes pré-definidos (Corba, COM [Orfali96] e JavaBeans [Java]). Entretanto, artefatos reutilizáveis existem em outras áreas [Jacobson97]. Nestas áreas, sistemas típicos possuem dezenas até centenas de pontos de flexibilização [Jacobson97] [Mattsson00] que necessitam de algum tipo de parametrização de modo a adaptar seu funcionamento à aplicação em desenvolvimento. Além disto, uma outra característica destes sistemas é que, devido ao seu tamanho e complexidade, são normalmente instanciados por um grupo de pessoas [Herbsleb01]. Sendo assim, a utilização de uma ferramenta que automatize o processo de forma parcial ou completa e que facilite o controle e execução do processo como um todo se torna mandatório.

Como apresentado em [Froehlich98], uma ferramenta de auxílio ao processo de desenvolvimento de aplicações baseado em reutilização deve propiciar aos reutilizadores suporte para configuração dos pontos de flexibilização de modo a facilitar seu uso e entendimento. Adicionalmente a isto, é possível destacar que esta ferramenta deve ter as seguintes características:

⇒ Permitir a integração dos dados obtidos nas fases anteriores à de design.

⇒ Mostrar possíveis pontos propícios a erros na representação.

⇒ Apresentar formas de corrigir estes erros.

⇒ Gerar uma documentação adicional onde atividades de instanciação possam ser rastreadas.

⇒ Verificar se as restrições de instanciação foram atendidas.

⇒ Controlar/Sincronizar atividades executadas de forma assíncrona para que o design final seja integro.

Tendo isto em mente, foi elaborada uma ferramenta que auxilia o reutilizador durante o processo de reutilização, através da execução semi-automática de um cookbook em RDL. De um modo geral, a ferramenta xFIT, XMI based Framework Instantiation Tool. Esta ferramenta permite:

⇒ Manipulação do design do artefato de modo a obter a aplicação final.

⇒ A introdução e utilização do espaço de nomes descoberto nas fases anteriores a reutilização.

⇒ Correção de problemas devido à sobreposição semântica [Mattsson00] através da aplicação de Refactoring [Fowler99].

⇒ Aplicação semi-automática de Design Patterns [Gamma95].

⇒ Verificação do conjunto de propriedades representadas no cookbook.

⇒ Rastreamento das atividades de reutilização que deram origem ao elemento final.

7.1 XFIT - Visão geral

Sob o ponto de vista do reutilizador, a ferramenta XFIT contem um agente de assistência [Bradshaw97] [Omg] ao usuário (reutilizador), que conduz as ações deste reutilizador junto ao processo de instanciação. Este assistente interage com o(s) reutilizador(es) de modo a

capturar do dados necessários aos incrementos específicos da aplicação que completarão os pontos de flexibilização/extensão do artefato. Esta assistência tem como base à interpretação da especificação do processo de instanciação representado em RDL, o cookbook.

Para desempenhar tal funcionalidade, XFIT tem como parâmetros de entrada:

⇒ **O programa de instanciação (cookbook)** – Parâmetro obrigatório que faz referencia ao arquivo que descreve o processo de instanciação em RDL.

⇒ **O design do artefato reutilizável** – Parâmetro obrigatório que faz referencia ao arquivo que especifica o artefato reutilizável em UML-FI. Este arquivo deve ser no formato XMI[Xmi].

⇒ **Espaço de nomes** – Certos processos de desenvolvimento capturam informações relevantes à aplicação em fases anteriores a de projeto. Para que esta informação não seja de todo perdida, XFIT permite a incorporação deste espaço durante sua interação com o reutilizador. Este parâmetro é opcional e deve ser representado de acordo com a dupla UML <-> XMI.

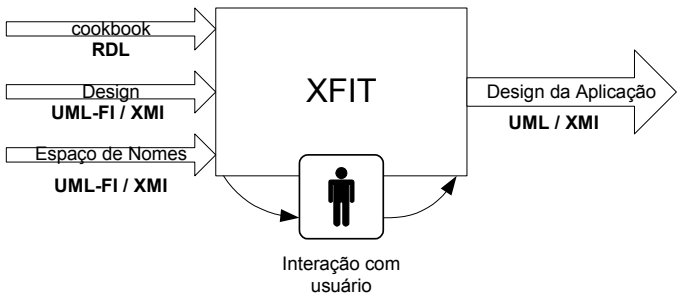


Figura 59 Insumos da ferramenta xFIT.

Uma vez executado o processo, a ferramenta preenche integralmente os pontos de flexibilização descritos no cookbook obtendo como produto final uma especificação em UML que contempla as funcionalidades da aplicação em desenvolvimento (figura 59). Vale ressaltar que caso o cookbook não represente as atividades necessárias para uma instanciação completa do artefato, este processo gerará um outro artefato reutilizável, porém mais especializado.

7.2 Arquitetura Interna

Para atender a funcionalidade especificada por RDL, bem como as características de uma ferramenta de auxílio à instanciação citadas anteriormente, a ferramenta XFIT foi dividida em 6 módulos onde cada fase da execução da ferramenta é contemplada.

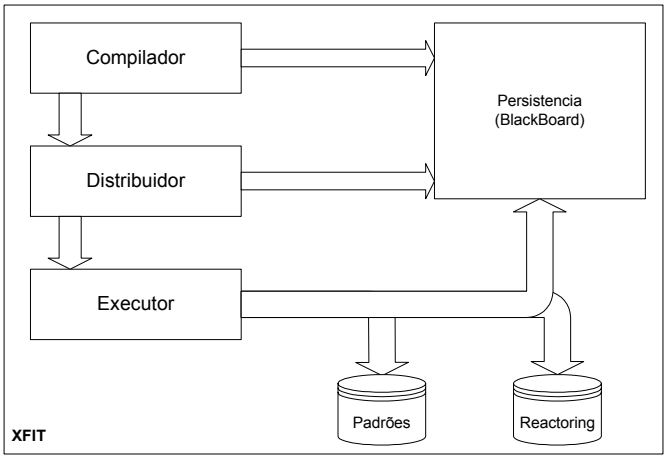


Figura 60 Arquitetura interna de xFIT.

Como apresentado na figura 60, estes seis módulos que interagem de forma a propiciar; a validação do cookbook RDL, geração do plano de execução, execução deste plano e persistência do processo. Estes módulos são:

⇒ **Compilador** – O compilador RDL tem como finalidade à validação sintática de um cookbook RDL, bem como a geração do plano de execução do processo de instanciação. O importante deste plano de execução é atender aos requisitos de assincronismo presentes na especificação em RDL. Sendo assim, como ilustrado na figura 61, o compilador agrupa atividades em unidades que podem ser executadas assincronamente (especificados pelo operador ||) de modo a permitir a distribuição deste processo.

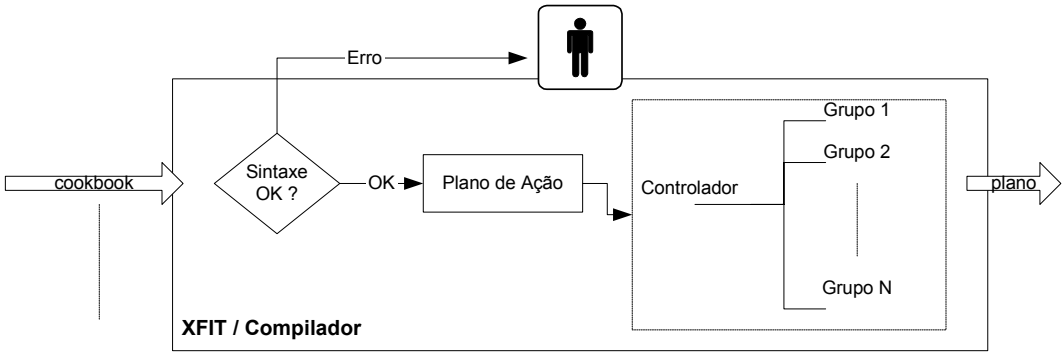


Figura 61 Execução do Compilador.

⇒ **Distribuidor Tarefas** – Dado que o processo instanciação é inerentemente multi-usuário [Jacobson97][Bosch01] (e por isso implica em assincronismo), a finalidade do distribuidor de tarefas é alocar um grupo de atividades a cada um destes usuários (Figura 62). Esta atribuição é feita com auxílio de uma interface gráfica.

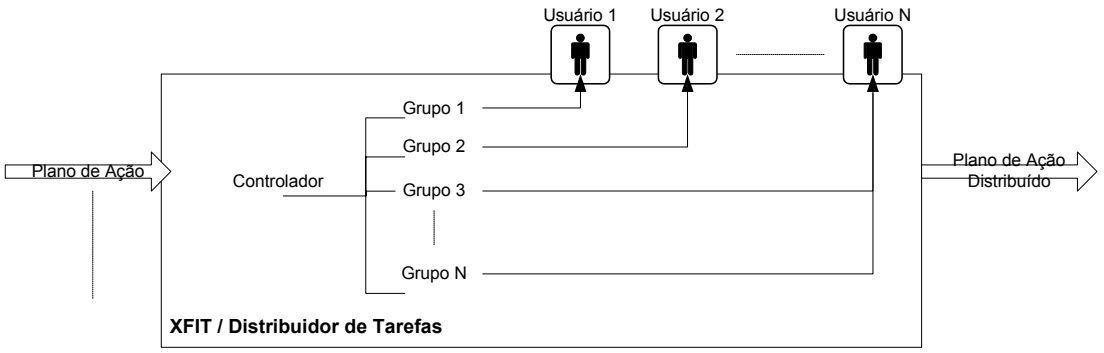


Figura 62 Execução do distribuidor de tarefas.

⇒ **Máquina de Execução (Executor)** – De posse do plano de ação devidamente alocado aos usuários e em conjunto com o ambiente DOM (Document Object Model) [Xmi] criado a partir do arquivo XMI que representa o artefato em uso, a máquina de execução aciona o assistente de modo a realizar a instanciação.

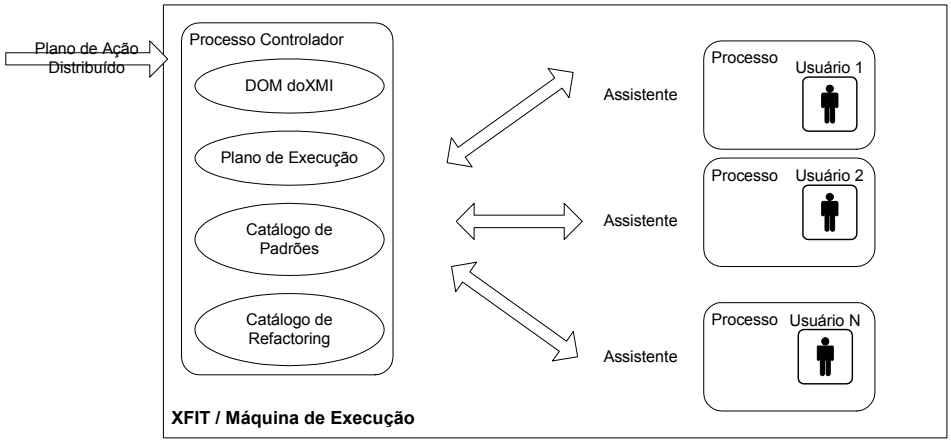


Figura 63 Execução da Máquina de Execuçã de XFIT.

⇒ **Persistência (Blackboard)** – O módulo de persistência é responsável pela gerência de todo o mecanismo de armazenamento em disco bem como toda conversão XMI <-> XFIT.

⇒ **Catálogo de Design Pattern** – O catálogo de padrões contem uma relação dos padrões utilizados durante o processo de instanciação. O formato deste catálogo é:

- Nome do Padrão.
- Lista de Parâmetros.
- Lista de atividades a serem executadas tomando como base RDL e a lista de parâmetros.

⇒ **Catálogo de Refactoring** – Este módulo descreve textualmente cada Refactoring utilizado na ferramenta e segue os mesmos moldes do catálogo de padrões.

7.3 Funcionalidade

Tomando como ponto partida à visão do reutilizador, a ferramenta XFIT pode ser dividida em duas grandes funcionalidades: a carga do artefato e a execução da instanciação (ilustrado pelo UseCase da Figura 64).

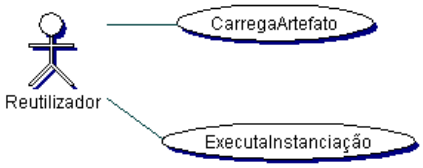
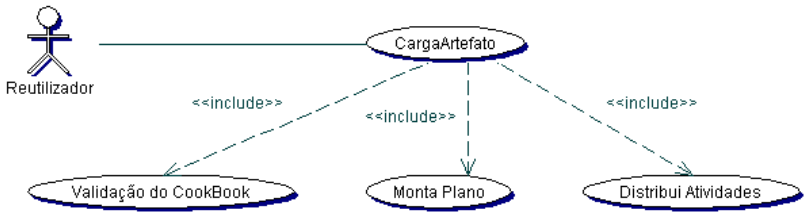
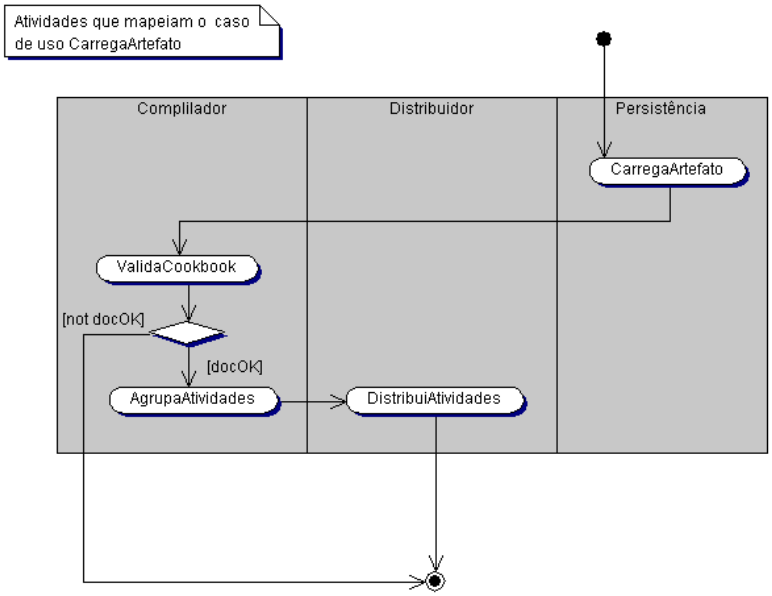


Figura 64 Atividades em que o reutilizador está envolvido.

Na fase de carga, o reutilizador define o artefato (cookbook + design + espaço de nomes), que será objeto da reutilização. Este artefato é então validado permitindo a criação do ambiente interno de execução do processo (UseCase e Diagrama de Atividades da Figura 65).



(a)



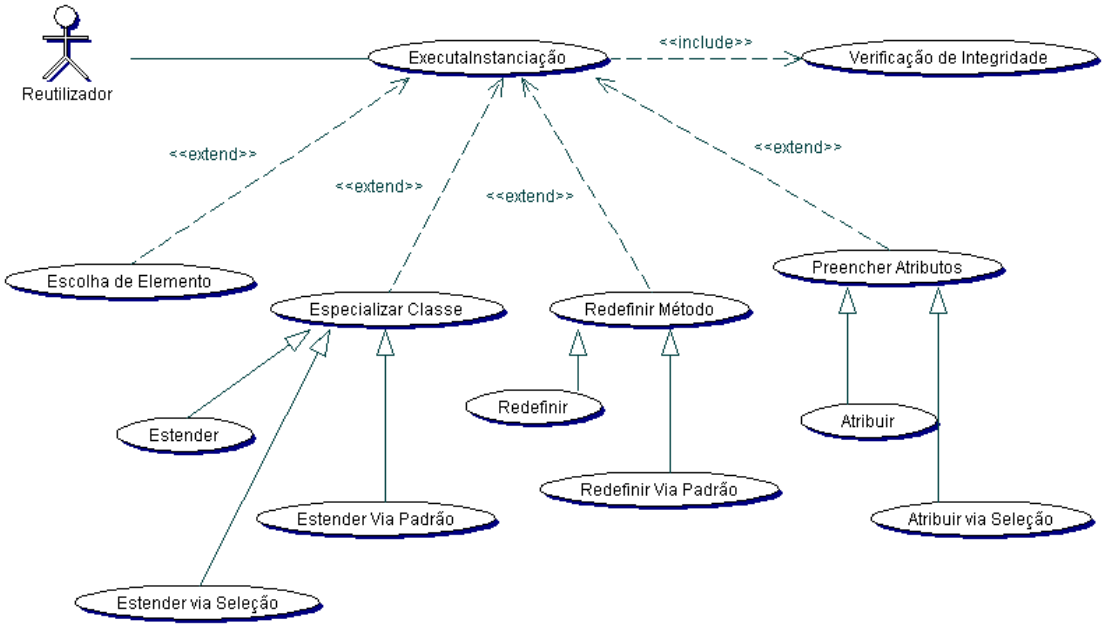
(b)

Figura 65 Especificação da atividade de carga de artefato.

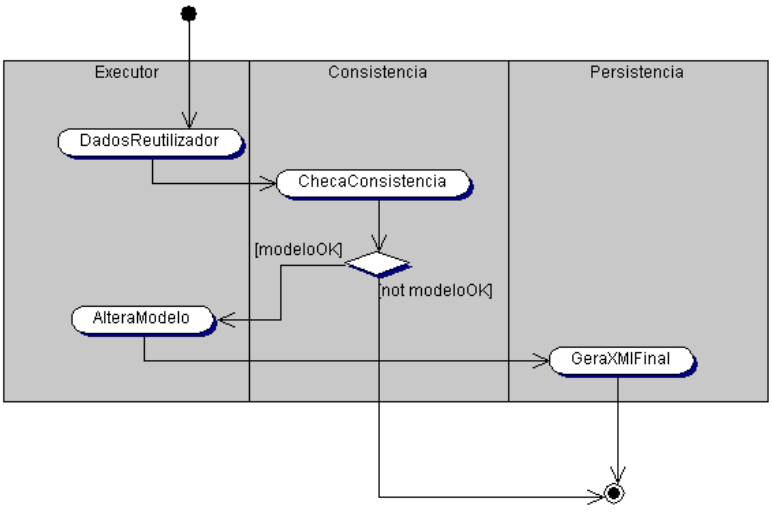
(a) Use Case (b) Diagrama de Atividades.

Uma vez obtido o plano de execução o assistente conduz uma seqüência de interações reutilizador ⇔ ferramenta de modo a completar os pontos de flexibilização segundo a notação RDL (UseCase da Figura 66). Um ponto importante desta execução é representado pelo caso Verificação de Integridade. Este case tem como atribuição principal o

monitoramento das restrições de integridade representadas no cookbook, bem como a ativação do módulo de Refactoring no caso houver sobreposição semântica.



(a)



(b)

Figura 66 Especificação do Caso Executa Instanciação.

7.4 Interface

Uma característica importante do processo de instanciação é a intervenção do reutilizador. Esta intervenção tem como objetivo capturar informações sobre o espaço de nomes que expressa as adaptações necessárias para o desenvolvimento da aplicação. Uma das formas para aumentar a legibilidade desta intervenção (interação homem ⇔ máquina) é através da utilização de interface GUI (Graphical User Interface) amigável. Com tal intuito, foi construído um protótipo em JAVA[Java] de um ambiente baseado em janelas que se integra e completa a arquitetura interna apresentada anteriormente (Figura 67).

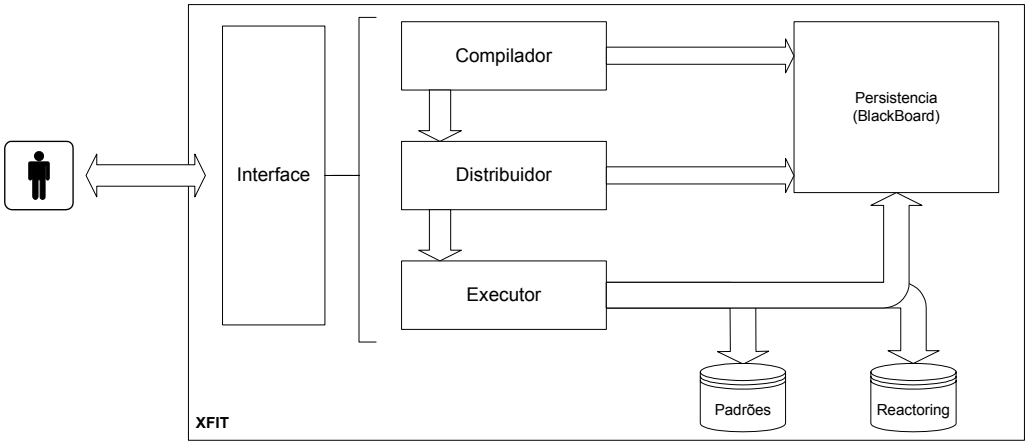


Figura 67 Introdução da interface GUI à ferramenta.

Esta "camada" de interface é responsável por toda a comunicação processo de instanciação ⇔ reutilizador. É através dela que nomes de classes novas , valores de elementos origatórios

e escolhas de elementos opcionais bem como todas as ações expressas por RDL que necessitam alguma informação adicional acontece.

Quando iniciada sua execução, a ferramenta xFIT apresenta uma janela principal a partir da qual o reutilizador tem uma visão geral dos elementos envolvidos no processo. Como ilustrado na figura 68, esta visão geral disponibiliza :

a) O espaço de nomes que contêm a lista de elementos do artefato reutilizável, que permite ao reutilizador visualizar em forma hierárquica as classes, métodos e atributos do artefato;
b) O espaço de nomes que contêm a lista de elementos da aplicação, que permite ao reutilizador visualizar em forma de uma lista, os nomes importantes descobertos durante alguma análise prévia;
c) O cookbook que especifica o processo de execução, que permite ao reutilizador tem uma visão geral das atividades de instanciação que este necessita executar;
d) A atividade corrente, que permite visualizar em destaque a atividade em execução e ;
e) Uma breve explicação de como proceder nesta atividade, que descreve em linhas gerais a atividade em execução.

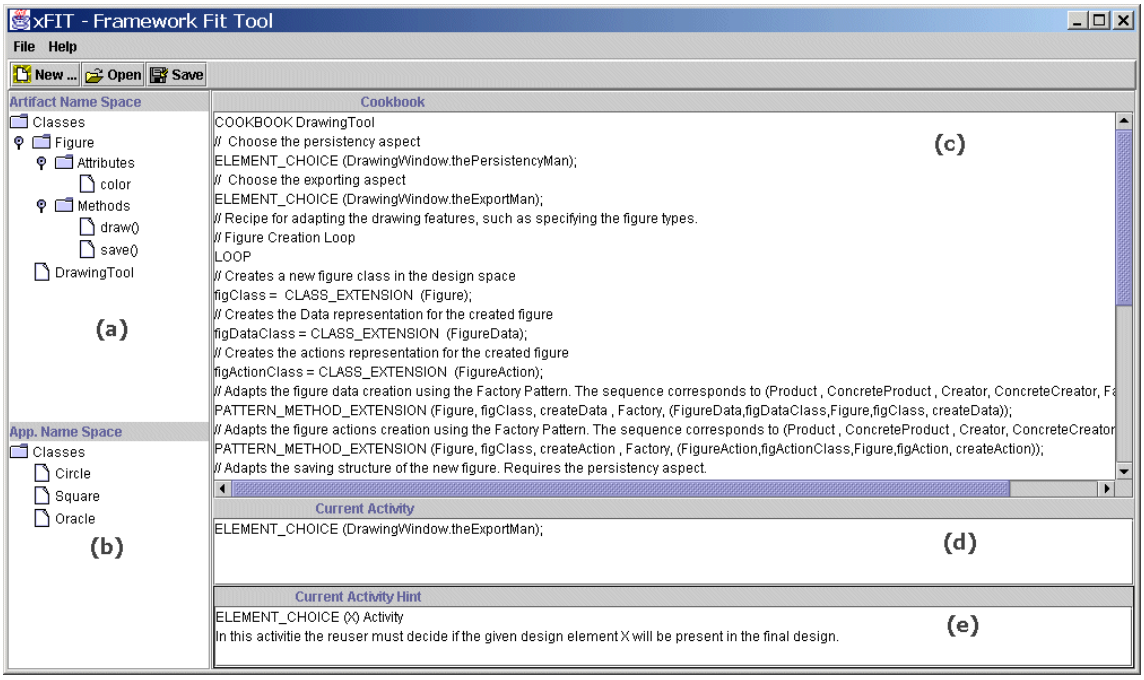


Figura 68 Tela principal de XFIT.

Uma vez apresentada esta janela principal, o processo de instanciação se inicia. Doravante, a camada de interface de xFIT apresenta algumas janelas cujos conteúdos estão diretamente ligados a especificação de RDL, pois capturam as informações requeridas pelas ações desta linguagem. Estas janelas podem ser de quatro tipos : Entrada de Nome Válido, Escolha Sim/Não, Escolha na Lista de Nomes e Escolha entre Duas Ações.

Entrada de Nome Válido – Esta janela (figura 69) é apresentada quando a ação associada requer um nome válido de acordo com as regras de criação de nomes para uma linguagem de programação convencional. As ações em RDL que utilizam esta janela são as especializações **class_extension e pattern_class_extension e** atribuição de valor **value_assignment**.

Figura 69 Tela para capturar CLASS_EXTENSION

Escolha Sim/Não - Esta janela (figura 70) é apresentada quando a semântica da ação associada requer uma escolha do tipo Sim/Não. Em RDL esta ação caracteriza a definição da presença de um elemento no design final (**element_choice**).

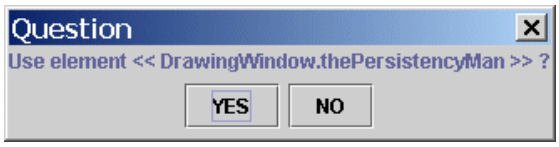


Figura 70 Tela para capturar ELEMENT_CHOICE.

Escolha na Lista de Nomes – Esta janela (figura 71) é apresentada quando a ação em questão requer a escolha de um elemento (simbolizado pelo seu nome) dentro uma lista. Em RDL as ações que necessitam de tal informação são a especialização por escolha **select_class_extension** e a atribuição por seleção **select_value**.

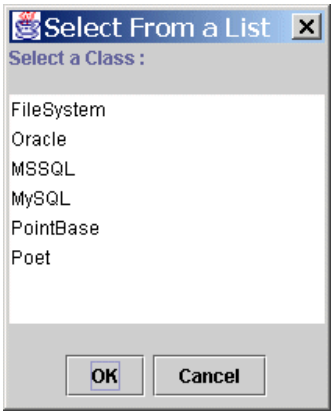


Figura 71 Tela para capturar SELECT_CLASS_EXTENSION.

Escolha entre Duas Ações – Esta janela é apresentada quando a ação em questão requer a escolha entre duas opções. Em RDL a ação que escolhe qual a opção será executada (**#**), necessita de tal informação.

7.5 Resumo

O processo de instanciação de um artefato reutilizável é considerado em muitos casos uma tarefa árdua. Esta dificuldade pode ser resumida por diversos fatores dentre eles a falta de expertise em técnicas de programação orientada a objetos como Design Patterns e a gerência requerida por um processo não seqüencial e em grupo [Herbsleb01]. Com o objetivo de minimizar esta complexidade, foi desenvolvida uma ferramenta de auxílio ao processo de instanciação do artefato nos moldes de um assistente de interface[Omg].

Embora não completamente funcional, a ferramenta xFIT demonstra claramente as vantagens de se utilizar um ambiente com tais características, pois esta guia o reutilizador por entre as atividades necessárias para se obter o design final através do uso de uma interface gráfica baseada em janelas.

8 Conclusões & Trabalhos Futuros

8.1 Conclusões

Frameworks Orientado a Objetos, também denominados por este trabalho de artefatos de software reutilizáveis, são considerados uma tecnologia promissora para a reutilização de projetos e implementações de software. Esta reutilização é obtida através de um processo de instanciação ou processo de reutilização cujo objetivo é introduzir os incrementos específicos que a aplicação em desenvolvimento traz a este artefato, de modo a completar e adaptar sua funcionalidade original.

Os benefícios obtidos por tal abordagem impactam fortemente o processo de desenvolvimento de software e podem ser expressos em termos do custo e tempo total de construção de um sistema. Estes benefícios são tão decisivos para as indústrias de desenvolvimento de software que instituições como HP, Ericsson e IBM passaram por intensos programas de reutilização na ultima década [Jacobson97] [Morisio99].

Muito embora os benefícios trazidos por tal abordagem sejam expressivos [Mattsson00], desenvolvedores de software (reutilizadores) enfrentam certos problemas ao utiliza-la. A longa curva de aprendizado que tem como objetivo capturar o conhecimento necessário para usar estes artefatos, atenua alguns destes benefícios. A aquisição deste conhecimento é necessária pois o processo de reutilização está fortemente baseado na composição de representações pouco estruturadas, pois em última analise, o reutilizador necessita introduzir um espaço de nomes que representa a aplicação em desenvolvimento (representa também os incrementos trazidos por ela) a um espaço de nomes estruturado como um diagrama de classes.

Aliado a esta descoberta de conhecimento, o reutilizador necessita obter informações de como (re)utilizar tal artefato. Neste ponto, o processo de aquisição destas informações é ainda mais difícil uma vez que sua representação normalmente baseada em linguagem natural.

Com o intuito de atenuar tais problemas este trabalho desenvolveu um conjunto de técnicas que somadas facilitam o trabalho de entendimento e uso de um design reutilizável quando estes são construídos com tecnologia orientada a objetos. Estas técnicas são descritas como:

⇒ Representação explicita dos pontos de flexibilização do artefato que indica ao reutilizador onde exatamente este deve focar sua busca de informação. Esta representação é descrita em UML-FI que é uma extensão de UML e UML-F.

⇒ Descrição explicita do processo de instanciação que indica quais passos o reutilizador deve executar para adaptar um determinado ponto de flexibilização. Esta descrição é feita na linguagem RDL especialmente criada para tal propósito.

⇒ Implementação da ferramenta xFIT, que auxilia e acompanha a execução do processo de instanciação. Esta ferramenta usa tecnologias como XMI para obter uma representação manipulável do design do artefato.

Em relação às abordagens existentes como Patterns [Johnson92], Cookbooks [Krasner88], Hooks [Froehlich97] e UML-F [Fontoura99], a abordagem proposta traz benefícios como :

⇒ Fácil aprendizagem devido à utilização de UML.

⇒ Integração com ferramentas CASE devido ao uso de XMI.

⇒ Descrição do processo de instanciação de forma precisa através de uma linguagem de específica para o domínio de reutilização de designs OO.

⇒ Reutilização guiada a pela interface de xFIT.

⇒ Atenuação do problema de sobreposição semântica e dispersão entre do design final e as especificações através do uso de Refactoring e Trace.

A comprovação da eficácia desta abordagem foi observada dentro da academia pela receptividade dada aos quatro artigos aceitos que ela originou e de forma prática durante seu uso em dois projetos um na academia e outro na industria. A utilização na academia foi durante o desenvolvimento da ferramenta CASE 2GOOD[Carvalho98]. Nesta ocasião foi necessário desenvolver um framework de desenho, o DTFrame, para atender aos requisitos gráficos da ferramenta. Durante o desenvolvimento, o framework foi idealizado e implementado por um desenvolvedor que a seguir utilizou uma notação ainda informal para especificar como e onde reutilizar. De posse desta descrição foram desenvolvidos quatro instâncias do framework.

A experiência prática na industria ocorreu durante o desenvolvimento e instanciação de um framework para sistemas de apoio à decisão , o DSSFrame[Oliveira00a], feito para o Centro de Análise de Sistemas Navais (CASNAV) que é um órgão de excelência quando se fala em informática dentro da Marinha do Brasil . Analogamente ao desenvolvimento acadêmico, foi idealizado e implementado um framework para atender os requisitos deste sistema que eram: executar, apresentar, organizar e manipular o ferramental matemático que está por traz de um sistema com esta natureza de modo a facilitar a sua utilização por usuários menos preparados.

Este projeto levou aproximadamente três anos e envolveu uma equipe de sete pessoas com diversos perfis de desenvolvimento. Em termos numéricos, o WFFrame tem aproximadamente 60 classes representando 30% do total de classes da instância final. Na ocasião foi possível observar :

- ⇒ O tempo de desenvolvimento da última instância foi de aproximadamente 40% do tempo do projeto original.
- ⇒ As instâncias desenvolvidas eram rapidamente absorvidas pelos usuários devido a homogeneidade de interface que estas apresentavam.
- ⇒ Pessoas com pouco preparo em programação orientada a objetos conseguiam rapidamente desenvolver sistemas com tal tecnologia pois iam direto no ponto de flexibilização. Estas pessoas também aprenderam a importância desta tecnologia.
- ⇒ Pessoas com pouca interação devido à incompatibilidade de horários conseguiam desenvolver as instâncias devido a documentação apresentada.

Vale ressaltar que na época deste desenvolvimento à documentação ainda era produzida em caráter informal através de anotações em diagramas de classes. Isto levou a uma perda de produtividade pois os desenvolvedores não eram guiados por um script. Esta falta de "guia" gerava também um outro problema. Alguns pontos de flexibilização tinham sua adaptação "esquecida" e só eram descobertos através de um erro de compilação ou até de execução da instancia. Na verdade foram estes problemas que deram origem ao desenvolvimento e formalização das técnicas presentes neste trabalho.

8.2 Resumo das Técnicas de Relacionadas

Como abordado anteriormente, este trabalho consiste em desenvolver e aprimorar técnicas para a sistematização do processo de reutilização. Para tal um conjunto de abordagens similares foram analisadas. A tabela a seguir apresenta um resumo destas abordagens segundo as premissas adotadas neste trabalho.

Premissas

Exposição dos Pontos de Extensão (Coluna 1) – Expressa como o PE é representado.

Como Customizar ?

	Expor PE	Como Customizar	Especifica Restrições	Especifica Distribuição	Ferramenta
• Cookbooks [Krasner88]	LN	LN	LN	Não	Não
• Patterns [Johnson92]	LN + Est	LN + Est	LN	Não	Não
• Metapatterns [Pree95]	LN + Est	LN + Est	Não	Não	Não
• Hooks[Froelich97]	LN + Est	LN + Est	LN	Não	Não
• UML-F [Fontoura99]	UML	Diagrama Atividades	Prolog	Não	Prolog
• HiFi [Ortigosa00]	TOON	Regras	Regras	Nos Agentes ?	HiFI

LN = Linguagem Natural
Est = Estrutura em Tópicos

8.3 Trabalhos Futuros

Ainda que a abordagem proposta tenha se mostrado extremamente útil ainda existem melhorias a serem feitas e resultados a serem extraídos.

No campo dos resultados[1], poderia ser feito algum tipo de heurística para determinar o esforço de reutilização de um determinado artefato. Esta heurística poderia atribuir pesos a alguns parâmetros facilmente identificados no script de instanciação. Estes parâmetros poderiam ser, por exemplo:

⇒ Número de pontos de reutilização – Quanto maior o número de elementos manipulados durante o processo de instanciação, mais demorado seria este processo.

⇒ Tipos de pontos de reutilização – A atividades de instanciação que estão relacionadas a uma escolha normalmente são mais fáceis de serem executadas do que as que levem em conta Design Patterns por exemplo.

⇒ Número de cláusulas condicionais – As cláusulas condicionais caracterizam um alto acoplamento das atividades de instanciação. Quanto maior este acoplamento mais difícil é executar este processo.

De posse deste resultado, o reutilizador poderia escolher um dentre muitos artefatos que satisfazem os requisitos de sua aplicação.

Um outro ponto importante é no que diz respeito a difícil identificação da funcionalidade em ponto grande do artefato. A documentação desenvolvida por esta abordagem identifica de forma clara a descrição e utilização de um ponto de flexibilização específico (ponto pequeno) porém não mapeia este ponto de flexibilização a nenhum requisito do sistema. Seria necessária alguma técnica que fosse capaz de encapsular um ou um conjunto de pontos de flexibilização em algo mais alto nível como um *Concern* [Tarr00] uma *Feature* [Kang93] ou

[1] Isto poderia ser feito sem nenhuma modificação da descrição apresentada neste trabalho.

um Use-Case do sistema. Com isto o reutilizador poderia escolher o artefato reutilizável através de descrições com maior poder de expressão [Ortigosa01]. O trabalho apresentado em [Mathias01] apresenta uma abordagem nesta direção.

No que diz respeito à ferramenta de auxílio xFIT propriamente dita algumas melhorias seriam extremamente interessantes sendo as mais importantes delas são em relação à: consistência do processo de instanciação ; consistência do design final ; aumento da funcionalidade da máquina de execução e nível de assistência.

A descrição apresentada por RDL caracteriza um processo distribuído e em grupo. Sendo assim atividades com restrições temporais podem introduzir situações como "dead-lock" no processo levando a um aborto prematuro de sua execução. Para minimizar tais problemas, propriedades como safety, liveness e reachability de uma atividade poderiam ser checadas através de um mapeamento a um formalismo adicional como apresentado em [Alencar01] de forma a garantir a consistência do processo como um todo.

No que tange a consistência do design final, seria extremamente importante garantir que a execução do processo de instanciação gere uma instancia válida. Como dito no capítulo sobre Documentação da Reutilização (Cap. 5), uma instancia válida pode ser caracterizada pelo preenchimento de todos os pontos de flexibilização obrigatórios. Isto é verdade quanto a parte estrutural do artefato, uma vez que as operações de preenchimento são baseadas em mecanismos de especialização, os quais preservam a semântica original do design. Entretanto quando for considerado o aspecto comportamental este argumento não é mais válido.

Os mecanismos de amarração tardia (late-binding) e polimorfismo introduzem um fluxo de mensagens não especificado no artefato e pode levar a uma instabilidade na execução do aplicativo. Sendo assim seria interessante introduzir na especificação do artefato um conjunto de propriedades [Alencar01] que deveriam ser atendidas para garantir a validade da instância.

Um outro ponto importante no campo das melhorias seria em relação a máquina de execução do processo. Esta máquina se encontra no núcleo da ferramenta xFIT e não atende a todos os requisitos descritos por RDL limitando seu escopo de aplicação. Para aumentar a funcionalidade da ferramenta, ao invés de desenvolver uma máquina de execução nova, a linguagem RDL poderia ser mapeada para alguma arquitetura de descrição de workflow como a proposta por Endeavors [Endeavors].

Por fim, o nível de assistência ao reutilizador poderia ser aumentado através da utilização de uma combinação de Agentes de Software [Bradshaw97] e Concerns[Tarr00]. A descrição dos Concerns de um sistema tende a agrupar elementos de design de modo a representar características funcionais e/ou não funcionais deste sistema. Uma vez que esta modelagem baseada em Concerns tenha sido feita tanto para o framework quanto para o artefato, uma equivalência entre estes Concerns poderia ser feita durante o processo de reutilização. De posse desta equivalência, um agente de software poderia "intuir" a próxima ação de reutilização a ser executada dentro de Concerns equivalentes.

Apêndice A - Histórico de Trabalhos

A abordagem apresentada nesta tese teve como um de seus frutos o desenvolvimento de diversos artigos dentre os quais alguns publicados em ou submetidos para, congressos e revistas internacionais. Estes artigos demonstram a evolução e maturação da abordagem até o presente momento e serão apresentados a seguir de forma resumida .

Artigo 1 – Dezembro / 1999 [Oliveira00a]

DSSFrame – A Decision Support System Framework with Agents

Este artigo descreve a elaboração do framework DSSFrame (Decision Support Systems Framework) desenvolvido para a Marinha do Brasil no âmbito do Projeto CASNAV/FPLF. Em sua essência este artigo apresenta duas propostas: o framework propriamente dito e uma forma de utilizar este framework através de descrições hierárquicas.

O framework DSSFrame tem como objetivo facilitar o desenvolvimento de sistemas de apoio à decisão que tem como característica a execução de um processo cíclico (workflow) finalizado por uma fase de escolha [Oliveira00a]. Uma vez que esta escolha leva a uma solução ótima de um determinado problema de acordo com determinados requisitos, a acurácia dos dados e ações deste processo são de extrema relevância. Tendo isto em mente, o DSSFrame integrou em sua arquitetura elementos como interface gráfica, persistência, suporte matemático intercambiável e um agente de software, que interagem de modo a criar um ambiente simples e guiado para auxiliar o usuário da aplicação.

Uma vez desenvolvido o framework com as características mencionadas acima, tornou-se necessário à definição de uma descrição para a utilização de tal design. Este descrição tem

início na representação dos pontos de flexibilização no framework que no caso do DSSFrame são basicamente a descrição das atividades envolvidas no processo.

O outro ponto desta descrição é a representação de como capturar as informações para tal preenchimento. Uma vez que as representações das tarefas/dados envolvidas no processo de decisão são eminentemente hierárquicas foi adotada uma descrição "BNF-like" destes elementos. Esta de BNF por sua vez se encaixaria no processo de instanciação (e por conseguinte nos pontos de flexibilização) uma vez que define o espaço de nomes da aplicação em desenvolvimento de forma estruturada.

O processo de "encaixe" desta BNF seria executado mapeando de cada nó desta representação seria (feito manualmente pelo reutilizador), para uma classe que representa um ponto de flexibilização do DSSFrame.

Esta abordagem demonstrou-se eficaz uma vez que define uma sistematização para o processo de preenchimento dos pontos de flexibilização com as características do DSSFrame. Um ponto importante a mencionar é que esta abordagem deu origem a todo o trabalho desenvolvido nesta tese.

Artigo 2 – Maio/2000 [Oliveira00b]

A Framework Based Approach for Workflow Software Development

Como evolução da solução proposta no artigo descrito acima, foram feitas duas modificações: a incorporação de um modelo que representasse a fase de análise de domínio e o relaxamento do DSSFrame para atender sistemas de informação de um modo em geral.

A fase de análise de domínio vem se colocando como uma etapa de destaque dentro do ciclo de desenvolvimento de um sistema. Esta fase tem como objetivo estudar abordagens com características similares a que está sendo desenvolvida de modo a capturar elementos imprescindíveis em um momento anterior ao inicio efetivo da construção da aplicação. Esta análise de domínio se encaixa na abordagem proposta uma vez que define o espaço de nomes da aplicação em desenvolvimento de maneira mais elaborada. Este espaço de nomes por sua vez é representado através de um conjunto de BNFs o qual pode ser mapeado de forma manual para os pontos de flexibilização do framework alvo.

No tocante ao relaxamento do DSFrame, foi a retirada de todo o suporte a customização do ferramental matemático de modo a obter um framework para capturar unicamente o encadeamento das atividades de um processo qualquer. Este relaxamento deu origem ao WFFrame (Workflow Framework).

Artigo 3 – Outubro / 2000 [Oliveira01a]

Using XML and Frameworks to Develop Information Systems

Este artigo deu origem a forma com que a abordagem proposta nesta tese foi elaborada, uma vez que apresentou a integração dos elementos básicos para a execução de um processo de desenvolvimento de software baseado na reutilização de frameworks orientados a objetos. Estes elementos básicos são: o modelo de Features [Kang93], a representação UML-F e a representação XML/XMI.

A integração destes elementos permite definir um processo sistemático para a instanciação de um framework. Este processo tem como ponto central a definição dos pontos de flexibilização do framework utilizando uma extensão da notação UML. Com esta definição é

possível perceber de forma clara: onde deve acontecer a integração entre o espaço de nomes da aplicação em desenvolvimento e o framework; e como esta integração deve acontecer de acordo com o tipo de ponto de flexibilização.

O modelo de Features entra em cena como uma excelente notação para representar os elementos descobertos na fase de análise de domínio. Este modelo se integra a idéia de frameworks uma vez que permite a representação de características alternativas, opcionais e obrigatórias dando flexibilidade ao design final (o que é imprescindível em um framework).

Por fim, uma outra contribuição deste artigo é a substituição de BNF por XML/XMI para representar a estrutura dos espaços de nomes envolvidos no processo de reutilização. Esta substituição tem como vantagem permitir que um programa possa manipular estes modelos de forma fácil de modo a garantir a corretude do processo.

Artigo 4 – Dezembro / 2000 [Mathias01]

Domain Oriented Framework Construction

Este artigo descreve a utilização da abordagem que integra Features + UML + XMI para a construção do framework propriamente dito. Isto é possível uma vez que frameworks por definição representam um conjunto de aplicações com características afins e devem representar estas afinidades através de pontos de flexibilização.

O produto obtido após a construção de um framework orientado a objetos é um design que representa as classes e seus relacionamentos. Estas classes possuem uma granularidade alta tornando difícil sua concepção a partir de uma metodologia tradicional de desenvolvimento[Pressman00]. Sendo assim a árdua tarefa de descobrir de pontos de

flexibilização (representados por classes) pode ser atenuada através da análise do modelo de Feutures, pois este contempla em sua representação a opcionalidade/alternatividade das características que compõem o domínio da aplicação.

Artigo 5 – Maio / 2001 [Alencar01]

Process-Based Representation and Analysis of Framework Instantiation

Este artigo estabelece a organização final da abordagem desta tese. Nele foram identificados: a necessidade de abordar o processo de instanciação através a elaboração de um programa que descreve as atividades deste processo e; a necessidade de estender UML-F de modo a capturar elementos alternativos, opcionais e obrigatórios de acordo com o modelo de Features.

Embora frameworks orientados a objetos sejam uma excelente técnica para obter reutilização em larga escala, a materialização desta reutilização é uma tarefa árdua principalmente a falta de uma documentação de como se obtém esta materialização (instanciação) . Para minimizar tal falta de documentação, foi apresentado neste trabalho um conjunto de técnicas que além de conduzir o reutilizador durante o processo de instanciação, garantem que este processo será bem sucedido.

Estas técnicas descrevem o processo de instanciação através da integração da notação UML estendida com uma linguagem especifica para o domínio de reutilização, desenvolvida especialmente para tal propósito (RDL). Esta integração se dá através da execução de uma ferramenta (xFIT) que de posse de um script de instanciação, manipula a representação em XMI do design do framework e captura informações do reutilizador necessárias para preencher os pontos de flexibilização.

No tocante a verificação do sucesso da execução deste processo, uns conjuntos de propriedades podem ser verificados por um Model Checker durante e depois desta execução, de modo a garantir que estas propriedades sejam satisfeitas após a obtenção a aplicação final.

Apêndice B - Diagramas de xFIT

Com o intuito de completar a especificação interna da ferramenta xFIT, este apêndice descreve os principais componentes utilizados, bem como suas interações.

Os principais componentes observados no design de xFIT são :

XFIT – Classe responsável pela inicialização do sistema.

RDLCompiler – Classe resposável pela compilação do script RDL. Tem como saída o conjunto de instruções que será executado pela máquina virtual.

RDLVirtualMachine – Classe responsável pela execução do script RDL.

UserGUI – Pacote onde se encontram as janelas responsáveis por capturar as escolhas efetuadas pelo reutilizador durante a execução do script RDL.

XMIProxy – Pacote onde se encontram as classes que efetuam a comunicação com o sistema de arquivo através do padrão XMI.

PatternManager – Classe responsável pela especificação, execução e controle do mecanismo de aplicação de padrões de projeto existente na ferramenta.

InstantiationCode – Classe responsável pelo armazenamento do código a ser executado pela máquina virtual.

XMIClassProxy – Classe responsável pelo armazenamento do espaço de nomes do framework a ser adaptado.

Estes componentes se relacionam de acordo com o diagrama de classes abaixo:

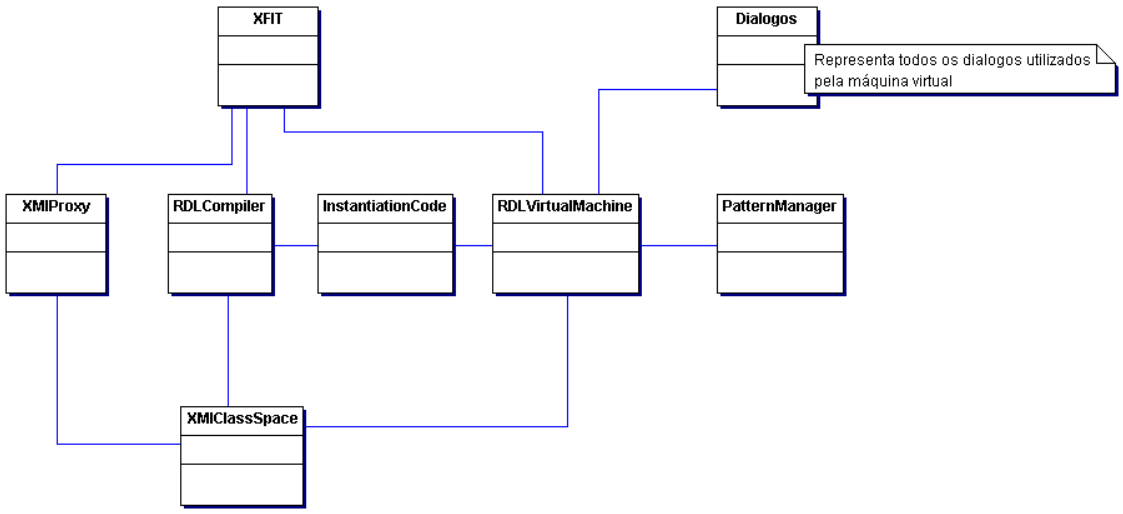


Figura 72 Relacionamento dos principais componentes de xFIT.

Como apresentado pelos casos de uso presentes nas figuras 64, 65a e 66a, as principais ações executadas pelo reutilizador junto a xFIT são: Carregamento de um Artefato Reutilizável e Execução de Atividades de Instanciação.

Como apresentado no diagrama de seqüência abaixo, a ação de carga de um artefato é caracterizada por dois momentos: 1) Construção do Espaço de nomes do framework a partir do XMI correspondente (operação *loadXMI*) e; 2) Geração de código para a máquina de execução do cookbook RDL (operação *parseRDLCode*).

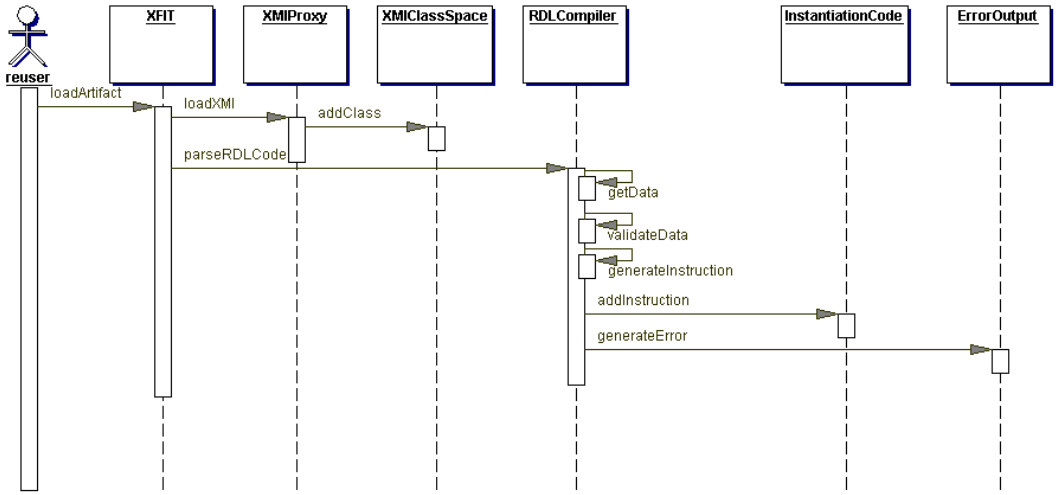


Figura 73 Diagrama de Sequencia para Carga do Artefato

As atividades de instanciação são responsáveis pela integração/adaptação do espaço de nomes do framework (gerado anteriormente). Esta integração/adaptação leva em conta dados fornecidos pelo reutilizador como pode ser observado no diagrama de seqüência abaixo. Durante a execução destas atividades, um conjunto de diálogos (janelas) é apresentado de modo a capturar esta informação de forma amigável.

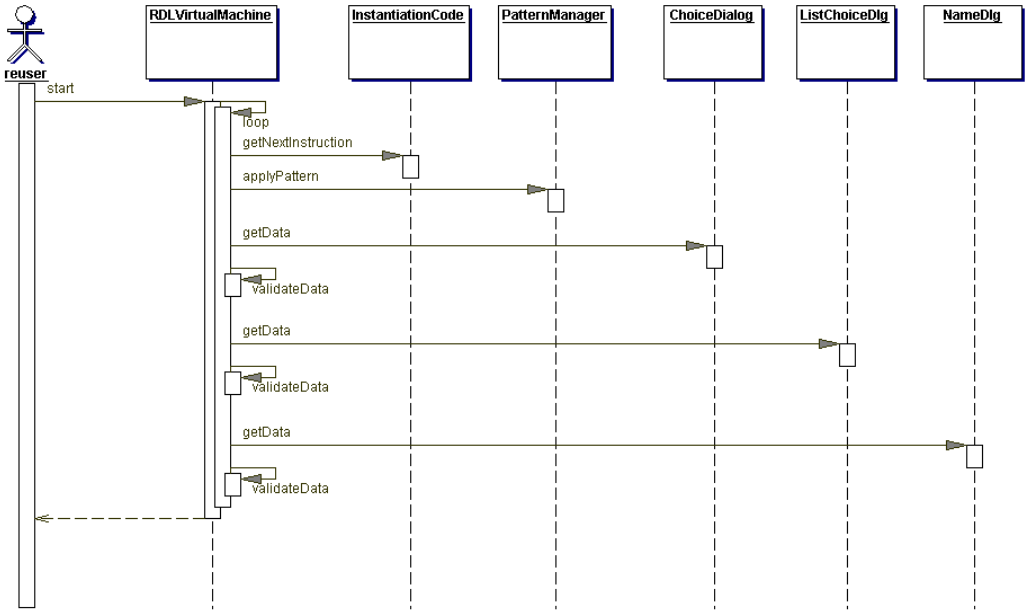


Figura 74 Diagrama de Sequencia para execução do processo de instanciação.

Bibliografia

⇒ [Ada] Especificações em www.adahome.com

⇒ [Alencar01] Alencar P.S.C. , Cowan, D.D. Oliveira, T.C. , Lucena C.J. P. Process-Based Representation and Analysis of Framework Instantiation – Submetido para Journal of System and Software , Elsevier.

⇒ [Alexander77] Alexander, C., S. Ishikawa, & M. Silverstein, *A Pattern Language*, Oxford University Press, 1977.

⇒ [Arango93] Arango, G.. *Domain Analysis Methods*. In Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability, p.17-49, March 1993, Lucca, Italy. Edited by Ruben Prieto-Diaz and William B. Frakes, IEEE Computer Society Press, 1993.

⇒ [Argo] Argo/UML Descrição encontrada em http://www.argouml.org.

⇒ [Armour00] Armour P.G. , The five orders of ignorance. p17-20 Communication of the ACM, Outubro de 2000.

⇒ [Batory98] Batory D. ,Smaragdakis, Y. Application Generators , ftp://ftp.cs.utexas.edu/pub/predator/generators.pdf

⇒ [Biggerstaff89] Biggerstaff, T. , Software Reuse , ACM Press 1989.

⇒ [Bosch01] Bosch J. Software Product Lines :Organizational Alternatives , p91-102, International Conference on Software Engineering, Toronto, Canada May 2001.

⇒ [Booch95] Booch, G. Object-Oriented Analysis and Design With Applications Addison Wesley 1994.

⇒ [Booch99] Booch, G., Rumbaugh, J., Jacobson, I., The Unified Modeling Language User Guide, Addison-Wesley, 1999.

⇒ [Bradshaw97] Bradshaw, J. M. , Software Agents MIT Press 1997

⇒ [Carvalho98] Second Generation Object-Oriented Development - CARVALHO, S.E.R.; CRUZ, S.O.; OLIVEIRA, T.C. 2nd CNPq/NSF Workshop on Formal Aspects of Computation - New Orleans, USA. 1998.

⇒ [CBuilder] Descrição encontrada em www.borland.com/cbuilder

⇒ [Delphi] Descrição encontrada em www.borland.com/delphi

⇒ [Dusink95] Dusink, L. , Katwijk, J.V. Reuse Dimensions Symposium of Software Reuse 1995.

⇒ [Endeavours] Descrição encontrada em http://www.ics.uci.edu/pub/endeavors/end/.

⇒ [Fayad99a] Fayad, M.E., Schmidt, D.C., Johnson, R.E., Domain-Specific Application Frameworks, Wiley Computer Publishing, 1999.

⇒ [Fayad99b] Fayad, M.E., Implementing Application Frameworks: Object-Oriented Frameworks at Work, Wiley Computer Publishing, 1999.

⇒ [Fayad99c] Fayad, M.E., Schimidt D.C., Building Application Frameworks : Object-Oriented Foundations of Framework Design , Wiley Computer Publishing, 1999.

⇒ [Flex] Descrição encontrada em www.gnu.org

⇒ [Fontoura99] Marcus Felipe Montenegro Carvalho Da Fontoura. A Systematic Approach to Framework Development. Ph.D. Thesis, Computer Science Department, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), 1999.

⇒ [Fontoura00] Fontoura, M.; Crespo, S.; Lucena, C.J.P.; Alencar, P.S.C.; Cowan, D.D. *Using Viewpoints to Derive Object-oriented Frameworks: a Case Study in the Web-based Education Domain*. The Journal of Systems and Software, 54 (2000) 239-257

⇒ [Fowler99] Fowler, M. *Refactoring : Improving the Design of Existing Code*. Addison-Wesley, , Reading, Massachusetts, June 1999.

⇒ [Frakes91] Frakes, W.B., Biggerstaff, T.J., Prieto Dias, R., Matsumura, K., Schafer, W., Software Reuse: is it delivering? ICSE, pp. 52-59, 1991.

⇒ [Froehlich97] Froehlich, G., Hoover, H.J., Liu L. and Sorenson, P.G. Hooking into Object-Oriented Application Frameworks, Proc. 19th Int'l Conf. on Software Engineering, Boston, May 1997, 491-501.

⇒ [Froehlich98] Froehlich, G., Hoover, H.J., Liu L. and Sorenson, P.G. Requirements for a Hoot Tool.

⇒ [Gamma95] Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company, 1995.

⇒ [Gangopadhyay95] D. Gangopadhyay, S. Mitra, "*Understanding Frameworks by Exploration of Exemplars*", in Proc. of 7th International Workshop on CASE, July 1995, pp. 90-99.

⇒ [Garlan96] Garlan D. Shawn M. *Software Architecture: Perspectives on an Emerging Discipline,* , Prentice Hall, April 1996.

⇒ [Helm90] Helm, R. Holland I. , Gangopadhyay D. , Contracts: Specifying Behavioral Composition in Object Oriented Systems. OOPSLA 1990.

⇒ [Herbsleb01] Herbsleb J.D. , Mokckus, A. , Finholt T.A., Grinter R.E. , An Empirical Study of Global Software Development: Distance and Speed, p81-90, International Conference on Software Engineering, Toronto, Canada May 2001.

⇒ [Jacobson94] Jacobson, I. Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley.

⇒ [Jacobson97] Jacobson, I.; Griss, M.; Jonsson, P. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, Reading, Massachusetts, June 1997.

⇒ [JBuilder] Descrição encontrada em www.borland.com/jbuilder

⇒ [Johnson88] Johnson R. , Brian F. , Designing Reusable Classes Journal of Object Oriented Programming 22-35 1988.

⇒ [Johnson92] Johnson, R., Documenting Frameworks Using Patterns, Proceedings of OOPSLA'92, ACM/SIGPLAN, New York, 1992.

⇒ [Johnson97]Johnson R. Components, Frameworks Patterns .Symposium of Software Reusability 1997 USA.

⇒ [Java] Descrição em www.javasun.com.

⇒ [Kang93] Kang, K.C.; Cohen, S.G.; Hess, J.A.; Novak, W.E. and Peterson, A.S.. *Feature-Oriented Domain Analysis (FODA) Feasibility Study (CMU/SEI-90-TR-21)*. Pittsburgh, PA, Software Engineering Institute, Carnegie Mellon University, Nov 1993.

⇒ [Krasner88] Krasner, G.E., Pope, S.T., A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80, Journal of Object-Oriented Programming 1(3), 1988.

⇒ [Krueger92] C. W. Software Reuse, ACM Computing Surveys Volume 4 Issue 2 p131-183.

⇒ [Kiczales97] Kiczales, G. , Lamping, J. , Mendhekar, A. , Maeda A. , Lopes C.V., Loingtier J.M. , Irwin, J., " Aspect-Oriented Programming". In *proceedings of the European Conference on Object-Oriented Programming (ECOOP),* Finland. Springer-Verlag LNCS 1241. June 1997.

⇒ [Lajoie96] R. Lajoie and R. K. Keller. *Design and reuse in object-oriented frameworks: Patterns*, contracts, and motifs in concert. In V. Alagar and R. Missaoui, editors, Object-Oriented Technology for Database and Software Systems, pages 295-- 312, Singapore, 1995. World Scientific

⇒ [Leite94] Leite J. C. S, Sant'Anna, M.,Freitas F. G. Draco-PUC: a Technology Assembly for Domain Oriented Software Development. In W. B. Frakes, editor, *3rd International Conference on Software Reusability*, pages 102-109, Rio de Janeiro, Brazil, November 1994. IEEE Press.

⇒ [Lex] Descrição encontrada em http://www.combo.org/lex_yacc_page/

⇒ [Mathias01] OLIVEIRA, T. C., MATHIAS , I. F., LUCENA, C. J. P. Domain Oriented Framework Construction In: ICEIS, 2001, Setubal.

⇒ [Mfc] Descrição encontrada em www.microsoft.com/mfc

⇒ [Mattsson00] Mattsson, M. *Evolution and Composition of Object-Oriented Frameworks*, PhD Thesis, Department of Software Engineering and Computer Science, University of Karlskrona/Ronneby, 2000.

⇒ [Morisio99] Morizio, M. , Ezran, M. Tully C. Introducint Reuse in Companies : A Survey of European Experiences. Symposium of Software Reusability Los Angeles USA -1999

⇒ [Naur86] Naur, P. and Randell, B., Eds., Software Engineering: Report on a Conference by the NATO Science Committee, *NATOScientific Affairs Division*, Brussels. 1968.

⇒ [Neighbors84] Neighbors , J. M. The DracoApproach to Constructing Software from Reusable Components. *IEEE Transactions on Software Engineering*, SE-10(5):564-574, September 1984.

⇒ [Oliveira00a] Oliveira, T.C. ; Carvalho, S.E.R. ;Lucena C.J. P. *DSSFrame - A Decision Suppot System with Agents*. Techinical Report Pontifícia Universidade Católica do Rio de Janeiro – Brazil 2000.

⇒ [Oliveira00b] Oliveira T. C., Mathias I., Lucena, C. J. P. *A Framework Approach for Workflow Software Development* Proceedings of IASTED International Conference on Software Engineering and Application, p330-335, Las Vegas USA, November 2000.

⇒ [Oliveira01] OLIVEIRA, T. C., LUCENA, C. J. P., MATHIAS , I. F. Using XML and Frameworks to develop Information Systems In: ICEIS, 2001, Setubal

⇒ [Ortigosa99] Ortigosa, A., Campo, M., Smartbooks: A Step Beyond Active-Cookbooks to Aid in Framework Instantiation, Technology of Object-Oriented Languages and Systems 25, IEEE Press, June 1999.

⇒ [Ortigosa00] Ortigosa A., Campo M., Salomon R., *Towards Agent-Oriented Assistance for Framework Instantiation*. In Proc. OOPSLA '00, Minneapolis, Minnesota USA, ACM SIGPLAN Notices, 35, 10, 2000, 253-263.

⇒ [Parnas76] D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(1):1-9, March 1976.

⇒ [Poulin99] –Poulin J. Reuse: Been There, Done That COMMUNICATIONS OF THE ACM May 1999/Vol. 42, No. 5 pg 99-100.

⇒ [Pree95] Pree, W., Design Patterns for Object-Oriented Software Development, Addison-Wesley Publishing Company, 1995.

⇒ [Pressman00] Pressman, R.S. *Software Engineering : A Practitioner's Approach*. McGraw Hill, New York, NY, June 2000.

⇒ [Prieto94] R. Prieto-Diaz, W. Schafer, M. Matsumoto, editors. *Software Reusability*. Ellis Horwood, New York, 1994

⇒ [Prolog] P. H. Salus, editor. *Handbook of Programming Languages: Functional and Logic Programming Languages*, volume 4, chapter Prolog: Programming in Logic. Macmillan Technical Publishing, 1998.

⇒ [Rose] Descrição encontrada em www.rational.com

⇒ [Rumbaugh96] Rumbaugh, J., Booch, G. , , e Jacobson I. *The unified modeling language for object-oriented development*. Technical report, Rational Software Corporation, 1996. http://www.rational.com/ot/uml.html

⇒ [Rup] Descrição encontrada em www.rational.com/rup

⇒ [Rumbaugh91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *ObjectOriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991

⇒ [Sommerville00] Sommerville, I. *Software Engineering*. Addison-Wesley, Reading, Massachusetts, August 2000.

⇒ [Salus98] Salus, P. H.: "*Handbook of Programming Languages: Object-Oriented Programming Languages*". Vol. 1. Macimillian. Indianapolis. 1998.

⇒ [Staa93] A. **Staa** .*Talisman: Ambiente de Engenharia de Software Assistido por Computador*. 1993

⇒ [Szyperski98] Clemens Szyperski Component Software - Beyond Object-Oriented Programming Addison-Wesley / ACM Press, 1998.

⇒ [Tarr00] Tarr, P. , Ossher, H. Multi-Dimensional Separation of Concerns and the Hyperspace Approach." Proceedings Architectures and Component Technology: The State-of-the-Art in Software Development, January 2000

⇒ [Tiberghien81] Tiberghien, J. *The Pascal Handbook*. Sybex, 1981.

⇒ [Together] Descrição encontrada em www.togethersoft.com

⇒ [Txl] Descrição encontrada em //www.combo.org/lex_yacc_page/

⇒ [VAge] Descrição encontrada em www-4.ibm.com/software/ad/smalltalk/

⇒ [XMI] Descrição encontrada em http://www.omg.org/technology/xml/index.htm

⇒ [Cress70] Cress, P.; Dirksen, P.; Graham, J.W. *FORTRAN IV with WATFOR and WATFIV*. Prentice-Hall, Englewood Cliffs, N. J., 1970.

⇒ [Pinson88] Pinson, L.J.; Wiener, R. S. *An Introduction to Object-oriented Programming and Smalltalk*. Addison-Wesley, Reading, Massachusetts, 1988.

⇒ [Stroustrup97] Stroustrup, B.; *The C++ Programming Language*. 3rd Edition, Addison-Wesley, Reading, Massachusetts, July 1997.

⇒ [Orfali96]Orfali, R , Harkey, D e Edwards - The Essential Distributed Objects. Wiley 1996

⇒ [Omg00]Object Management Group – Agent Platform Special Interest Group. "Agent Technology – Green Paper". Version 1.0, September 2000

⇒ [Lange95] Lange D.B. Nakamura Y, Interactive visualization of design patterns can help in framework understanding p342-357 Proceedings of the 10th annual conference on

Object-oriented programming systems, languages, and applications October, 1995, Austin, TX USA.

⇒ [Pree97] W. Pree, G. Pomberger, A. Schappert, and P. Sommerlad. *Active Guidance of Framework Development*. Software-Concepts and Tools, 16(3):136--45, 1995.

⇒ [Eden98] Discussão sobre herança múltipla encontrada em http://www.csd.uu.se/kurs/oop/ht98/Lectures/D7/html/Introduction_to_Multiple_Inheritance/tsld006.htm

⇒ [Junit] Descrição encontrada em www.junit.org

⇒ [Hudak96] "Building Domain-Specific Embedded Languages", *ACM ComputingSurveys*, 28(4es), 196-es, 1996.