FRAMEWORKS DE APLICAÇÕES ORIENTADAS A OBJETOS – UMA ABORDAGEM ITERATIVA E INCREMENTAL

CRISTIANE MARISE PÉREZ DA SILVA CARNEIRO

Salvador

Abril - 2003

UNIVERSIDADE SALVADOR

PROGRAMA DE PÓS-GRADUAÇÃO EM REDES DE COMPUTADORES

FRAMEWORKS DE APLICAÇÕES ORIENTADAS A OBJETOS – UMA ABORDAGEM ITERATIVA E INCREMENTAL

CRISTIANE MARISE PÉREZ DA SILVA CARNEIRO

Dissertação apresentada à Universidade Salvador, como parte das exigências do Curso de Mestrado em Redes de Computadores, área de concentração em Engenharia de Software, para obtenção do título de "Mestre".

Orientador
Prof. Dr. Manoel Gomes de Mendonça Neto

Salvador
Abril - 2003

FRAMEWORKS DE APLICAÇÕES ORIENTADAS A OBJETOS – UMA ABORDAGEM ITERATIVA E INCREMENTAL

CRISTIANE MARISE PÉREZ DA SILVA CARNEIRO

Dissertação apresentada à Universidade Salvador, como parte das exigências do Curso de Mestrado em Redes de Computadores, área de concentração em Engenharia de Software, para obtenção do título de "Mestre".

APROVADA em 10 de abril de 2003.

Banca Examinadora:

Profa. Dra. Leila Maciel de Almeida e Silva (UFS)	Membro
Prof. Dr. Celso Alberto Saibel Santos (UNIFACS)	Membro
Prof. Dr. Manoel Gomes de Mendonça Neto (UNIFACS)	Orientador

Salvador
Abril – 2003

Agradecimentos

A Deus, por permitir a conclusão deste trabalho.

Ao meu orientador, professor Manoel, por compartilhar a sua experiência e ajudar na condução do trabalho.

Ao meu marido Glauco, pelo companheirismo e pelo apoio nos momentos difíceis.

Aos meus pais Nélio e Reasilvia e a minha irmã Andréa, pelo apoio e estímulo.

A toda minha família e amigos, pela confiança.

Á UNIFACS, em especial aos professores e colaboradores do NUPERC, pela atenção dispensada.

SUMÁRIO

1. INTRODUÇÃO 1

1.1. MOTIVAÇÃO DO TRABALHO 2
1.2. ORGANIZAÇÃO DA DISSERTAÇAO 3

2. REVISÃO BIBLIOGRÁFICA 4

2.3. HISTÓRICO 4
2.4. FRAMEWORK ORIENTADO A OBJETOS - CONCEITOS 5
2.4.1. Construções Orientadas a Objetos Usadas por Frameworks 6
2.4.1.1. Herança 6
2.4.1.2. Associações Dinâmicas e Polimorfismo 6
2.4.1.3. Classes Abstratas 7
2.4.2. Hot Spots e Utilização de Frameworks 8
2.4.3. Categorização do Framework 10
2.4.4. Benefícios do Uso de Frameworks 11
2.4.5. Camadas do Framework 12
2.4.6. Evolução de Frameworks 13
2.5. FRAMEWORKS E PADRÕES DE PROJETO 14
2.5.1. Relacionamento entre Frameworks e Padrões de Projeto 16
2.6. DESENVOLVIMENTO DE *FRAMEWORKS* 18
2.6.1. Recursos Humanos Envolvidos 20
2.6.2. Metodologias de Desenvolvimento de Frameworks 21
2.6.2.1. Comparação Entre as Metodologias 26
2.7. DIFICULDADES E PROBLEMAS NO DESENVOLVIMENTO DE *FRAMEWORKS* 27
2.7.1. Detecção de Hot Spot 28
2.7.2. Implementação e Validação 28
2.7.3. Evolução 29
2.7.4. Aplicabilidade 29
2.7.5. Complexidade e Curva de Aprendizagem 29

3. DESENVOLVIMENTO ITERATIVO E INCREMENTAL 31

3.1. VISÃO GERAL 31
3.2. ETAPAS DO PROCESSO PROPOSTO PARA O DESENVOLVIMENTO ITERATIVO E INCREMENTAL 32
3.2.1. Passo 1 - Analisar Domínio 33
3.2.2. Passo 2 - Construir Framework 35
3.2.2.1. Passo 2.1 – Coletar Requisitos 35
3.2.2.2. Passo 2.2 – Análise e Detecção de *Hot Spots* 40
3.2.2.3. Passo 2.3 – Projetar *Framework* 43
3.2.2.4. Passo 2.4 – Implementação e Instanciação exemplo 47
3.2.2.5. Passo 2.5 – Elaborar Roteiro 49
3.2.3. Passo 3 - Instanciar Aplicações 50

4. ESTUDO DE CASO 53

4.1. ABORDAGEM UTILIZADA 53
4.2. DESENVOLVIMENTO 54

5. DISCUSSÃO DO TRABALHO REALIZADO 88

5.1. LIMITAÇÕES 89
5.2. ASPECTOS NÃO ABORDADOS 89
6. CONSIDERAÇÕES FINAIS 90
6.1. CONTRIBUIÇÕES 90
6.2. TRABALHOS FUTUROS 91
REFERÊNCIAS BIBLIOGRÁFICAS 92
APÊNDICE 1 - Aplicações Instanciadas do *JHotDraw* 97
APÊNDICE 2 - Padrões de Projeto 103
APÊNDICE 3 - Amostra de UML e Notações Relacionadas 109

Lista de Figuras

Figura 2.1 – Conceitos de associações dinâmicas e polimorfismo 7
Figura 2.2 – Arquitetura de um *framework* 8
Figura 2.3 – Aplicação desenvolvida totalmente sem reutilização 9
Figura 2.4 – Aplicação desenvolvida reutilizando bibliotecas de classes 9
Figura 2.5 – Aplicação desenvolvida utilizando *framework* 10
Figura 2.6 – Conjunto de abstrações arquiteturais 10
Figura 2.7 – Divisão do *framework* em camadas 13
Figura 2.8 – Padrões representando diferentes perspectivas de um assunto 15
Figura 2.9 – Integração entre padrões e *frameworks* 17
Figura 2.10 – A decomposição de uma aplicação baseada em *frameworks* e padrões .. 18
Figura 2.11 – Geração e alteração de *frameworks* e aplicações 20
Figura 2.12 – Recurso do desenvolvimento tradicional de aplicações 21
Figura 2.13 – Recursos do desenvolvimento de aplicações baseado em *frameworks* ... 21
Figura 2.14 – Processo de desenvolvimento de *framework* 22
Figura 2.15 – Projeto orientado a objetos tradicional 22
Figura 2.16 – Etapas do projeto dirigido por *hot spot* 24
Figura 2.17 – Atividades do projeto de *framework* 26
Figura 3.1 – Processo de desenvolvimento iterativo e incremental 33
Figura 3.2 – Separação dos requisitos nas aplicações e no *framework*. 36
Figura 3.3 – Identificação dos requisitos do *framework* a partir do domínio 36
Figura 3.4 – Alocando casos de uso a ciclos de desenvolvimento 37
Figura 3.5 – Casos de uso abstrato e concreto 38
Figura 3.6 – Padrão *strategy*. 45
Figura 3.7 – Padrão *factory method* 46
Figura 3.8 – Padrão *template method*. 46
Figura 3.9 – A implementação em um ciclo de desenvolvimento influencia o ciclo posterior. 48
Figura 4.1 – *Framework* gerado a cada ciclo 54
Figura 4.2 – Modelo estático 55
Figura 4.3 – Diagrama de classes do 1o. ciclo 57
Figura 4.4 – Estruturação do *framework* 58
Figura 4.5 – Diagrama de classes do projeto – 1o. ciclo 60
Figura 4.5 (Continuação) 61
Figura 4.6 – Editor de exemplo gerado com o *framework* utilizando as figuras e ferramentas padrões. 64
Figura 4.7 – Diagrama de classes do 2º. ciclo 68
Figura 4.8 – Diagrama de classes do projeto – 2o. ciclo 70
Figura 4.8 (Continuação) 71
Figura 4.9 - Editor de exemplo com menu, texto e comandos de edição. 73
Figura 4.10 – Resultado de *CommandMenu*. 76
Figura 4.11 – Resultado de *CommandChoice*. 76
Figura 4.12 – Resultado de *CommandButton*. 77
Figura 4.13 – Diagrama de classes 78
Figura 4.14 - Diagrama de classes do projeto – 3 º ciclo 80
Figura 4.14 (Continuação) 81
Figura 4.15 - Editor de exemplo executado como applet, utilizando borda, conectores e texto. 84

Figura 4.16 – Manipuladores de figuras. 86
Figura A1.1 – *Interface* de *planet application*. 97
Figura A1.2 – *Interface* de *JavaDraw*. 99
Figura A1.3 – *Interface* de *PertApplet*. 101
Figura A2.1 –Padrões de projeto *composite* e *template method* no *JhotDraw*. 103
Figura A2.2 – Padrão de projeto *decorater* no *JhotDraw*. 104
Figura A2.3 – Padrões de projeto *observer* e *strategy* no *JhotDraw*. 105
Figura A2.4 – Padrão de projeto *state* no *JhotDraw*. 106
Figura A2.5 –Padrão de projeto *factory method* no JhotDraw. 106
Figura A2.6 –Padrão de projeto *Template Method* no *JhotDraw*. 107
Figura A2.7 –Padrão de projeto *mediator* no *JhotDraw*. 107
Figura A2.8 –Padrão de projeto *command* no *JhotDraw*. 108
Figura A3.1 - Classe 109
Figura A3.2 - Generalização 109
Figura A3.3 - Agregação 109
Figura A3.4 - Multiplicidade 109
Figura A3.5 - Associações 110
Figura A3.6 - Diagrama de Caso de Uso 110

Lista de Tabelas

Tabela 3.1 – Tipos de adaptação para um *hot spot*.. 41

RESUMO

Carneiro, C**. Frameworks de Aplicações Orientadas a Objetos – Uma Abordagem Iterativa e Incremental.** 2003. 110 f. Dissertação (Mestrado em Redes de Computadores), Universidade Salvador, Salvador.

Palavras-chave: *Framework*, reutilização, orientação a objetos, desenvolvimento.

Reutilização é uma das abordagens mais usadas para melhorar a qualidade e reduzir o custo e tempo de desenvolvimento de *software*. Programação orientada a objetos (OO) é freqüentemente citada como um dos meios para se atingir a reusabilidade de *software*. *Frameworks* orientados a objetos podem ser usados para promover reutilização tanto ao nível de implementação como ao nível arquitetural. Todavia, existem várias dificuldades associadas ao desenvolvimento e uso de *frameworks* OO. As mais importantes são a complexidade de projeto e a dificuldade de reutilização de *frameworks*.

Esta dissertação apresenta uma abordagem iterativa e incremental para o desenvolvimento de *frameworks* orientados a objetos. Esta abordagem cria uma família de *frameworks* OO com crescente grau de sofisticação, com cada membro contendo mais funcionalidades que o anterior. Além disso, cada *framework* da família é por si só um *framework* completo e utilizável.

Esta abordagem reduz o risco de desenvolvimento, pois segue um processo passo a passo que produz a cada ciclo de desenvolvimento um *framework* que é um pouco mais sofisticado que os anteriores. Esta abordagem também reduz a complexidade de reutilização, pois oferece ao desenvolvedor de aplicações uma família de *frameworks* que varia de um *framework* mais simples até um mais sofisticado. Isto permite ao desenvolver escolher o *framework* que melhor se adapte às suas necessidades.

Por último, esta dissertação apresenta um estudo de caso onde a abordagem proposta é usada para criar uma família com três *frameworks* e a partir deles desenvolve três aplicações - uma para cada *framework* da família.

ABSTRACT

Carneiro, C. **Object-Oriented Application Frameworks – An Iteractive and Incremental Approach.** 2003. 110 f. Dissertation (Computer Network Master Program), Universidade Salvador, Salvador.

Keywords: Framework, reuse, object-oriented, development.

Software reuse is one of the most important approaches used to improve software quality and to reduce software costs and development time. Object-oriented (OO) programming is often cited as a means to achieve software reuse. Object-oriented frameworks can be used to promote software reuse both at implementation and architectural levels. However, there are several difficulties associated with OO framework development and use. The most notable ones are the framework project complexity and difficulty to reuse.

This dissertation presents an iteractive and incremental approach for object-oriented framework development. The approach creates a family of OO frameworks with increasing degree of sophistication, meaning that each framework of the family has more functionality than the previous one. Besides, each framework of the family is by itself a complete and usable framework.

This approach reduces the development risk by following a stepwise process to produce, at each development cycle, frameworks that are incrementally more sophisticated than the previous ones. This approach also reduces reuse complexity by offering to the application developer a family of frameworks, which ranges from simple to more complex frameworks. This allows the developer to choose the framework that is best suited to his needs.

Lastly, this dissertation presents a case study that enacts our approach by creating a family of three frameworks, and from them developing three applications - one for each of the frameworks.

1. INTRODUÇÃO

Na área de engenharia de *software* várias abordagens buscam melhorar a qualidade dos produtos gerados, bem como diminuir o tempo e o esforço necessários para produzi-los. A reutilização representa um dos principais fatores que podem levar ao aumento da qualidade e produtividade da atividade de desenvolvimento de *software.*

Com o uso da tecnologia orientada a objetos destaca-se a importância da flexibilidade para garantir a reutilização de *software*. Com este objetivo, as técnicas de orientação a objetos utilizam mecanismos tais como herança, polimorfismo e associações dinâmicas.

Entretanto apenas os mecanismos de orientação a objetos por si próprios estão longe de resolver o problema, pois a reutilização não é característica inerente da orientação a objetos, mas é obtida a partir do uso de técnicas que produzam *software*s reutilizáveis.

Para ajudar no processo de reutilização a abordagem de *frameworks* de aplicações orientadas a objetos foi introduzida. Um *framework* consiste de código de *software* que é parcialmente abstrato e cujas partes abstratas podem ser especializadas para produzir aplicações.

Um *framework* deve ser simples o bastante para ser aprendido pelo desenvolvedor de aplicações, deve prover características suficientes que possam ser usadas e métodos flexíveis para as características que podem ser mudadas.

Projetar *frameworks* verdadeiramente flexíveis e reutilizáveis é difícil e trabalhoso, pois há vários problemas durante o seu desenvolvimento e utilização. Em especial citamos:

- Detecção de *hot spots* (pontos de flexibilidade): é difícil determinar quais de suas propriedades devem ser flexíveis.
- Implementação e validação: é difícil escolher a maneira adequada de implementar os *hot spots* e verificar o comportamento de um *framework*;
- Evolução: Pode tornar sua estrutura (interfaces, hierarquia de classes e outros) complexa de gerenciar e entender.
- Aplicabilidade: é difícil determinar quando o *framework* é apropriado como um todo.
- Complexidade e curva de aprendizagem: *frameworks* freqüentemente se tornam complexos e difíceis de entender.

O objetivo deste trabalho é propor um processo para o desenvolvimento de *frameworks* de aplicações orientadas a objetos que utilizará o mecanismo de desenvolvimento iterativo e incremental para as etapas de coleta de requisitos, análise e detecção de *hot spots* , projeto, implementação e elaboração de roteiro.

A utilização dessa abordagem apresenta como vantagem o fato de manter a complexidade da construção do *framework* sob controle através da geração de vários produtos intermediários, que evolui a cada ciclo de desenvolvimento. Isso permite criar uma família de *frameworks* que pode ser utilizada pelo desenvolvedor para escolher aquele que melhor se adapte às necessidades de sua aplicação.

1.1. MOTIVAÇÃO DO TRABALHO

A presente dissertação é motivada pelo reconhecimento das vantagens do desenvolvimento de *software* com reutilização e pelo o fato de que entre as abordagens de reutilização existentes a de produção de *frameworks* possuir poucos processos de desenvolvimento que guiem o desenvolvedor e o ajude a contornar as dificuldades encontradas.

Entre as metodologias existentes encontram-se as de Johnson (1993), Pree (1999), Taligent (1994), Bosh (1999) e Schmid (1999) que propuseram processos que descrevem como produzir *frameworks* mas não detalharam os modelos que guiam o desenvolvimento de *frameworks*.

Durante a produção de *frameworks* as principais dificuldades encontradas são:

- Definição de quais propriedades podem ser flexíveis (configuráveis, extensíveis ou reimplementáveis).
- Implementação dos *hot spots* e validação de suas instâncias.
- Controle da evolução do *framework.*
- Entendimento do domínio do *framework* e sua aplicabilidade.
- Controle da complexidade e dificuldade de aprendizado para sua utilização.

Diante do que foi exposto, propomos um processo de desenvolvimento de *frameworks* iterativo e incremental, que abranje as etapas de análise de domínio, coleta de requisitos, análise e detecção de *hot spots*, projeto, implementação e instanciação exemplo, elaboração de roteiro e instanciação de aplicações.

A principal vantagem desta abordagem é que a complexidade da construção do *framework* é mantida sob controle através da geração de vários produtos intermediários utilizáveis e evoluindo-os a cada ciclo do processo.

A produção de pequenos *frameworks* a cada ciclo torna-os mais flexíveis, pois estes podem ser reutilizados mais freqüentemente e assim contribuem para que o desenvolvimento de aplicações seja menos complexo.

1.2. ORGANIZAÇÃO DA DISSERTAÇAO

Esta dissertação está organizada da seguinte forma: no capítulo 2 é realizada uma revisão bibliográfica, onde é apresentado o histórico do *framework,* mostrados os conceitos de *framework* orientados a objetos, os principais conceitos de orientação a objetos relacionados com *frameworks* tais como herança, associações dinâmicas e polimorfismo. É mostrado o relacionamento entre *frameworks* e padrões de projeto, são discutidos os processos de desenvolvimento existentes, são discutidas as principais dificuldades e problemas no seu desenvolvimento, tais como a análise de domínio e detecção de *hot spots*, implementação e evolução. No capítulo 3 é apresentado o nosso processo de desenvolvimento iterativo e incremental para a construção de *framework* de aplicações orientadas a objetos. No capítulo 4 é validado o processo proposto através de um estudo de caso. No capítulo 5 é realizada a discussão do trabalho realizado e no capítulo 6 são feitas as considerações finais apresentando as nossas contribuições e propostas para trabalhos futuros. No apêndice 1 são mostradas aplicações instanciadas dos *frameworks* gerados. No apêndice 2 são mostrados alguns dos padrões de projeto utilizados e no apêndice 3 são exibidas amostras de UML e notações relacionadas.

2. REVISÃO BIBLIOGRÁFICA

Este capítulo tem por objetivo apresentar o histórico do *framework*, mostrar os conceitos de *framework* orientados a objetos, os principais conceitos de orientação a objetos relacionados com *frameworks* tais como herança, associações dinâmicas e polimorfismo. Mostrar o relacionamento entre *frameworks* e padrões de projeto, discutir os processos de desenvolvimento existentes, discutir as principais dificuldades e problemas no seu desenvolvimento, tais como a análise de domínio e detecção de *hot spots*, implementação e evolução.

2.3. HISTÓRICO

A origem dos *frameworks* de aplicações orientadas a objetos nos remete a origem da programação orientada a objetos. Em 1967, uma linguagem de programação chamada SIMULA (Dahl e outros, 1970) e (Nygaard e Dahl, 1981) foi criada com o objetivo de criar *software* para simulação. Embora SIMULA não tenha se tornado largamente disseminada, a linguagem foi altamente influente na metodologia de programação moderna. Entre outras coisas SIMULA introduziu conceitos importantes de programação orientada a objeto como classes, objetos, encapsulamento de dados, herança, métodos e associações dinâmicas (Holmevik, 1995).

SIMULA permaneceu obscura até os pesquisadores da companhia Xerox Parc redescobrirem as suas idéias nos anos 70, iniciando o desenvolvimento da linguagem Smalltalk. Em 1980 surgiu o Smalltalk-80 e em 1983 a linguagem e sua implementação foi publicada incluindo os fundamentos da programação orientada a objetos.

Incorporada a linguagem SIMULA, havia conjuntos de classes cooperantes que podem ser descritas como *frameworks*, como por exemplo, a coleção de classes Simulation (Dahl e outros, 1970).

Apesar dos conceitos inovadores, a orientação a objetos não foi difundida até os desenvolvedores de *software* percebê-la como possível solução para o complexo problema de reutilização de *software* – uma abordagem comumente aceita para reduzir os custos do desenvolvimento de *software* e aumentar a qualidade do *software*.

O primeiro *framework* largamente utilizado foi o *framework* de interface do usuário (UI) do Smalltalk-80 chamado de *Model-View-Controller*. O seu sucesso e o posterior surgimento de um grande número de *frameworks* de interfaces de usuários

(UI) criou a imagem de que os *frameworks* se limitavam apenas a esse tipo de domínio, entretanto, há vários tipos de *frameworks* para diferentes domínios de aplicações.

2.4. *FRAMEWORK* ORIENTADO A OBJETOS - CONCEITOS

Framework é um conjunto de objetos reutilizáveis que engloba conhecimento de determinadas áreas e se aplica a um domínio específico. Uma aplicação completa ou parte significante dela pode ser especializada dessa estrutura fazendo-se as adaptações necessárias ou adicionando-se novas características ao *framework* (Viljamaa, 2001).

Um *framework* determina a arquitetura da aplicação e predefine parâmetros de projeto, permitindo ao projetista/implementador concentrar-se nos detalhes específicos da aplicação.

Ele é definido também como um projeto de reutilização que descreve como o sistema está decomposto em um conjunto de objetos que interagem entre si - O sistema pode ser uma aplicação inteira ou apenas um subsistema. O *framework* descreve a interface de cada objeto e o fluxo de controle entre eles (Jonhson e Foote, 1998).

Em (Gamma e outros 1995) *framework* é descrito como um conjunto de classes que torna um projeto reutilizável para uma classe específica de *software*. Um desenvolvedor o adapta para uma aplicação particular através de subclasses e composição de instâncias das classes do *framework*.

Taligent (1994) classificou os *frameworks* de acordo com os domínios de problemas que eles abrangem:

- *Frameworks* de suporte: provê serviços no nível de sistemas, tais como acesso a arquivos, suporte a computação distribuída ou a dispositivos de drivers. Desenvolvedores de aplicações tipicamente usam *frameworks* de suporte diretamente ou usam modificações produzidas pelos provedores de sistema. Contudo, *frameworks* de suporte podem ser especializados - por exemplo, quando desenvolvendo um novo sistema de arquivos ou dispositivo controlador.
- *Frameworks* de aplicações: encapsulam conhecimento aplicável a uma variedade de programas. Por exemplo: interface de usuário gráfica (GUI) para aplicações comerciais.
- *Frameworks* de domínio: encapsulam o conhecimento de um domínio de problema particular, por exemplo: *framework* de multimídia.

Um *framework* tem uma representação física em termos de classes e métodos, onde não apenas a implementação é reutilizável, mas também a sua estrutura é reutilizável.

2.4.1. Construções Orientadas a Objetos Usadas por *Frameworks*

Frameworks são atualmente implementados utilizando linguagens orientadas a objetos tipo Smalltalk, Java, ou C++, obtendo vantagem das técnicas de orientação a objeto que essas linguagens possuem (Viljamaa, 2001).

Apresentamos a seguir uma breve descrição das principais características de orientação a objetos que tornam o desenvolvimento de *framework* viável.

2.4.1.1. Herança

Uma classe pode normalmente herdar os atributos e características de outra classe (embora herança possa ser bloqueada usando diferentes métodos em diferentes linguagens).

A classe original é chamada de classe base ou superclasse e a nova classe é chamada de classe derivada ou subclasse. Herança é freqüentemente referida como uma extensão da superclasse.

Herança é hierárquica, ou seja, uma classe pode ser subclasse de uma classe e superclasse de outra.

A subclasse herda a representação dos dados e o comportamento da superclasse exceto quando a subclasse modifica o comportamento sobrescrevendo os seus métodos. A subclasse pode também adicionar novas representações de dados e comportamentos que são específicos aos seus propósitos (Baldwin, 1999).

2.4.1.2. Associações Dinâmicas e Polimorfismo

Associação dinâmica consiste em uma associação tardia de chamada de função, ou seja, a chamada de função é adicionada ao objeto durante o tempo de execução e não durante a compilação. Verifica-se que nos objetos construções como:

```
If (S é do tipo Circle) {drawCircle(); }
Elseif (S é do tipo Square) {drawSquare(); }
```

São substituidas por uma chamada a função comum `S.draw()` (Figura 2.1). Em tal chamada o sistema em tempo de execução toma a decisão de chamar a função

`draw()` implementada na classe `Circle` ou a função `draw()` implementada na classe `Square`. Como resultado da associação dinâmica o polimorfismo é alcançado (Figura 2.1).

Polimorfismo é a capacidade de se referir a diferentes derivações de classes da mesma maneira, mas obtendo o comportamento apropriado para a classe derivada que está sendo referenciada (Shalloway e Trott, 2002).

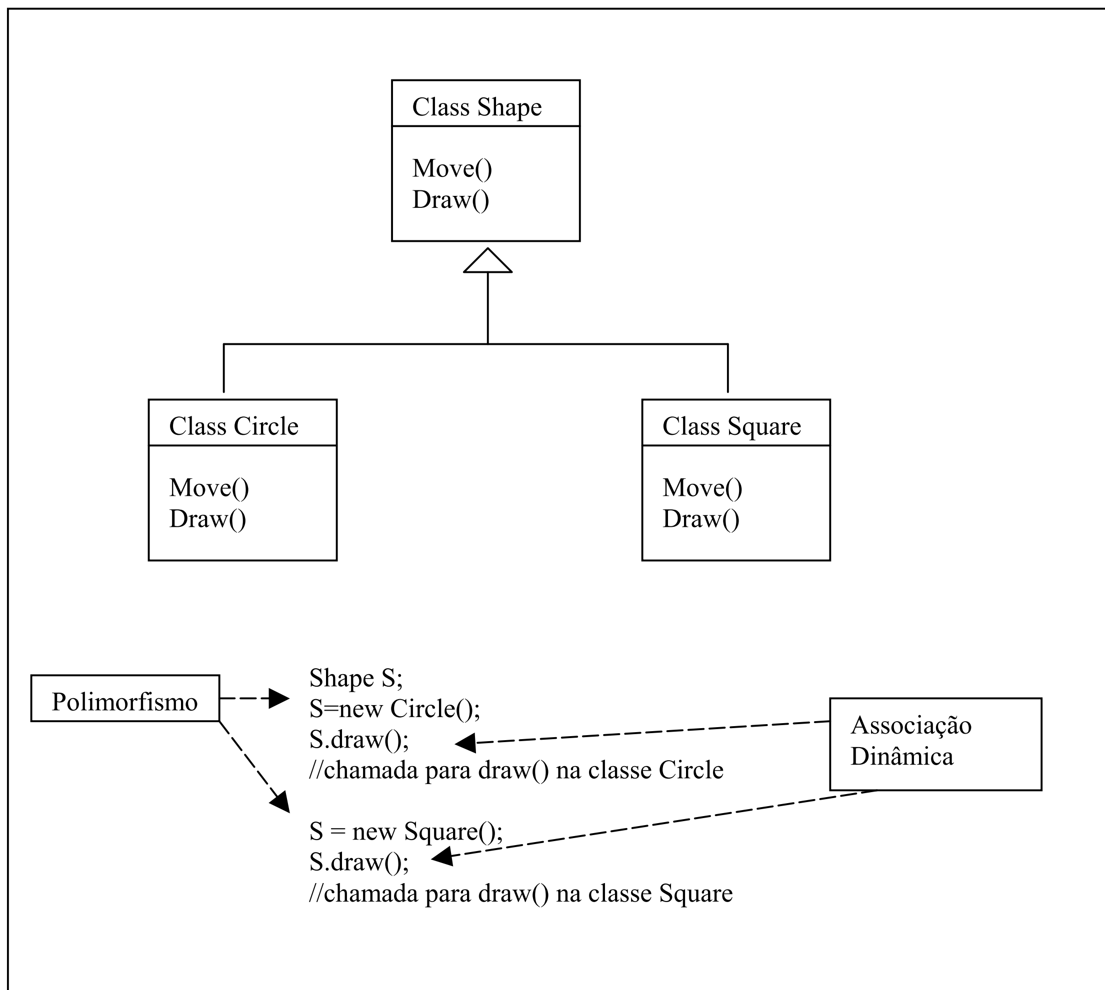

Figura 2.1 – Conceitos de associações dinâmicas e polimorfismo (Landin e Niklasson, 1995)

2.4.1.3. Classes Abstratas

Uma classe abstrata é projetada para ter subclasses, mas não é possível instanciar objetos a partir dessa classe.

Uma classe abstrata pode conter métodos abstratos, que contém somente a assinatura do método e é projetado para ser sobrescrito, não podendo ser invocado até que esteja completamente definido (Baldwin, 1999).

2.4.2. *Hot Spots* e Utilização de *Frameworks*

As partes do *framework* que são abertas à extensão e especialização são denominadas *hot spots,* que são áreas variáveis ou pontos de flexibilização (Pree, 1995).

Hot spots tornam o *framework* flexível. Eles são uma espécie de lacuna no domínio da aplicação, sendo tarefa do desenvolvedor da aplicação preenchê-las com uma solução própria.

O projeto do *framework* fornece características em dois níveis:

(1) Características do domínio, que são características relevantes do domínio úteis nas aplicações.

(2) Características estruturais, que são características que facilitam a adaptação e a evolução do *framework*.

As características estruturais incluem um arranjo de lógica (projeto) e estrutura física (classes) do *framework*.

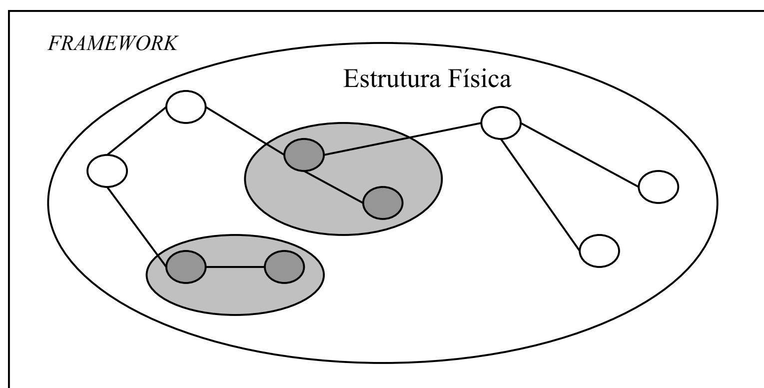

Figura 2.2 – Arquitetura de um *framework*

A Figura 2.2 exemplifica um *framework* como uma arquitetura predefinida, consistindo de blocos semi-acabados (*hot spots*) e blocos prontos para usar. As regiões sombreadas são os *hot spots* do *framework*.

Segundo Lajoie e Keller (1994) os *frameworks* não são bibliotecas de classes. Um *framework* é o projeto de um conjunto de classes que colaboram para realizar um conjunto de responsabilidades. Enquanto os componentes das bibliotecas de classes são usados individualmente, classes no *framework* são reutilizadas como um todo para resolver uma instância específica de um certo problema.

A reutilização promovida pela abordagem de *frameworks* se situa num patamar de granularidade superior à reutilização de classes, por reusar classes interligadas ao invés de classes isoladas (Silva, 2000).

Da perspectiva do desenvolvedor de aplicações, a maior diferença entre um *framework* e uma biblioteca de classes está no conhecimento necessário para utilizá-los. Os usuários da biblioteca de classes precisam somente entender a interface externa das classes e definir toda a estrutura de sua aplicação. Em contraste, usuários de *frameworks* devem entender o projeto abstrato do *framework* bem como a estrutura de suas classes, de forma a adaptá-las ou estendê-las. As Figuras 2.3, 2.4 e 2.5 ilustram essa diferença (a parte sombreada representa classes e associações que são reutilizadas).

Em geral, é mais difícil aprender a usar *frameworks* que bibliotecas de classes, contudo, seu potencial para reutilização excede bastante o potencial de reutilização da biblioteca de classes, contribuindo mais significativamente para o aumento de produtividade no desenvolvimento de *software*.

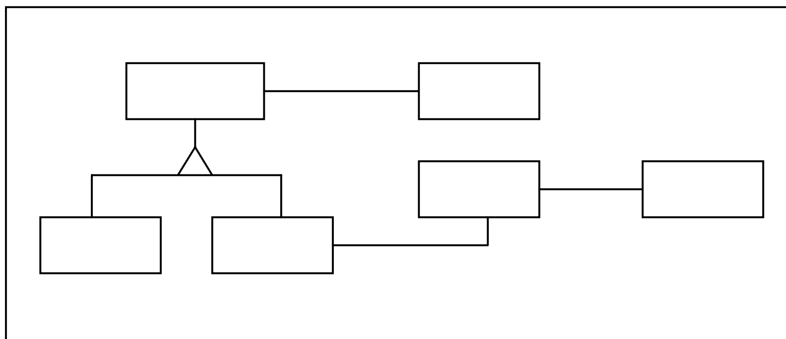

Figura 2.3 – Aplicação desenvolvida totalmente sem reutilização (Silva, 2000).

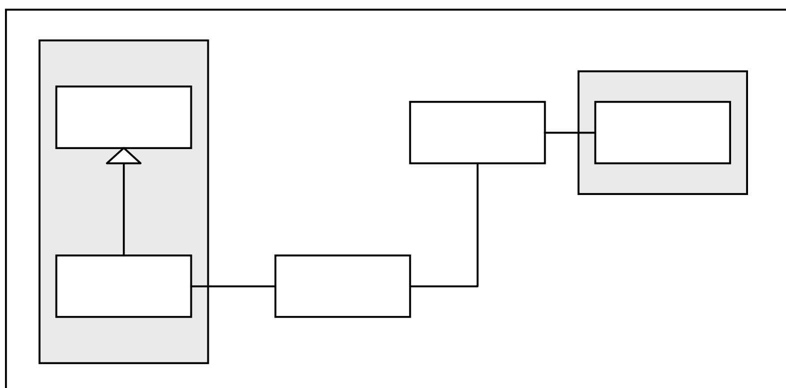

Figura 2.4 – Aplicação desenvolvida reutilizando bibliotecas de classes (Silva, 2000).

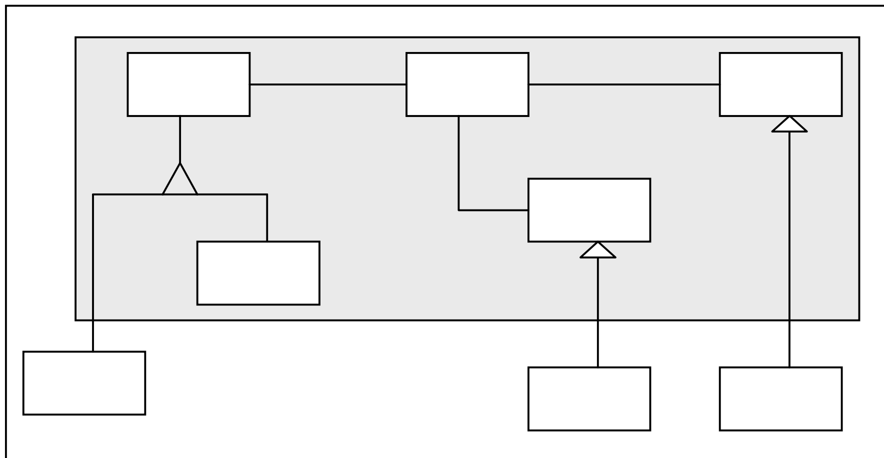

Figura 2.5 – Aplicação desenvolvida utilizando *framework* (Silva, 2000).

2.4.3. Categorização do *Framework*

Frameworks orientados a objetos são vistos como uma categoria de abstrações arquiteturais que suportam o projeto e a construção da estrutura lógica (Figura 2.6) (Fayad e outros, 1999).

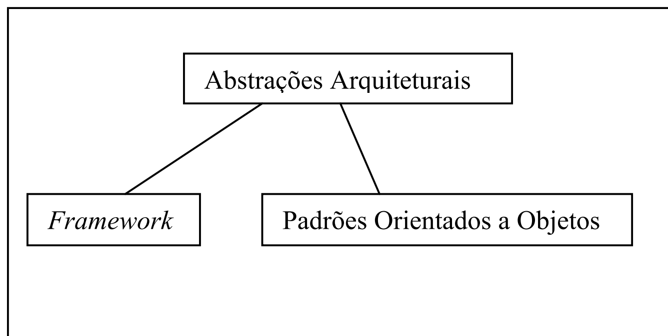

Figura 2.6 – Conjunto de abstrações arquiteturais (Fayad e outros, 1999).

O uso de abstrações no desenvolvimento de *software* é considerado uma vantagem, pois torna o desenvolvimento mais fácil já que permite ao desenvolvedor trabalhar com menos elementos – abstraindo dos detalhes. Além disso, abstração facilita a reutilização. É difícil reutilizar projeto de *software* ou implementações diretamente. Quando os potenciais projetos reutilizáveis e as implementações são abstraídas de contextos específicos e representadas por conceitos e relações generalizadas, tipicamente terão maior escopo de aplicação.

Os seguintes itens caracterizam aspectos de uma abstração arquitetural:

Estrutura: especifica a interface ou o conjunto de elementos básicos como também a estrutura estática e/ou dinâmica que os relacionam em uma composição.

Funcionalidade: conjunto de propriedades úteis que são reutilizadas quando aplicando a abstração.

Abstração: identifica e nomeia uma composição de elementos com uma certa estrutura interna e uma certa funcionalidade.

Reutilização: capacidade de ser aplicado várias vezes.

Fayad e outros (1999) explicam que *frameworks* são vistos como uma categoria de abstrações arquiteturais pois atendem aos quatro aspectos da abstração arquitetural:

- Um *framework* é uma composição de classes, cujas colaborações e responsabilidades são especificadas (Estrutura).
- De forma a prover as características requeridas do domínio, um *framework* embute as funcionalidades comuns da aplicação em seu domínio. Além disso, para ser utilizado no desenvolvimento de aplicações, um *framework* deve possuir a funcionalidade de adaptação (Funcionalidade).
- Um *framework* representa uma abstração de estruturas e funcionalidades no domínio (Abstração).
- Um *framework* expressa as características gerais de um grupo de aplicações e é reutilizável quando desenvolvendo aplicações nesse grupo (Reutilização).

2.4.4. Benefícios do Uso de *Frameworks*

Os principais benefícios da utilização de *frameworks* de aplicação orientada a objetos são a modularidade, a reusabilidade, a extensibilidade e a inversão de controle que eles fornecem aos desenvolvedores (Fayad e outros, 1999):

- Modularidade: *frameworks* promovem a modularidade por encapsular detalhes de implementação volátil em interfaces estáveis. A modularidade ajuda a aumentar a qualidade do *software* por localizar o impacto de mudanças no projeto e na implementação. Essa localização reduz o esforço requerido para entender e manter o *software* existente.
- Reusabilidade: as interfaces estáveis fornecidas pelo *framework* promovem a reusabilidade por definir um componente genérico que pode ser reaplicado para criar novas aplicações. A reusabilidade influencia no conhecimento do domínio e prioriza os esforços de desenvolvedores experientes a fim de evitar recriação e revalidação de soluções comuns a requisitos recorrentes de aplicações e desafios de projeto de *software*.

- Extensibilidade: um *framework* promove a extensibilidade por fornecer métodos adaptáveis (*hook*) explícitos (Pree, 1995). Permitindo as aplicações estenderem suas interfaces estáveis. A extensibilidade é essencial para assegurar a especialização de novos serviços da aplicação e novas características.
- Inversão de controle: a arquitetura do *framework* é caracterizada por uma inversão de controle. Isso permite ao *framework* (ao invés da aplicação) determinar que conjunto de métodos específicos da aplicação invocar em resposta a eventos externos.

Frameworks são considerados de grande importância para o desenvolvimento de aplicações orientadas a objetos. Eles são considerados como uma das principais formas através da qual sistemas orientados a objetos podem alcançar a reusabilidade.

2.4.5. Camadas do *Framework*

Os objetos do *framework* são na sua maioria descritos com classes abstratas. O projeto inclui a interface do componente e geralmente um núcleo para implementação. Por exemplo, uma classe abstrata pode definir uma estrutura para um algoritmo. Cada passo no algoritmo é definido como uma chamada para um método abstrato (na mesma classe ou em outra classe). Um *framework* pode especificar as implementações padrões para estes métodos ou deixá-los sem a implementação.

Uma aplicação derivada do *framework* pode usar diretamente as implementações padrões ou especificar novas classes concretas que herdam as classes abstratas e implementa seus métodos abstratos.

É uma boa prática separar no *framework* as interfaces e a implementação abstrata. Isto resulta em um *framework* com camadas estruturadas claras. As camadas mais altas (mais abstratas) são independentes das camadas mais baixas (mais concretas). Assim as camadas mais baixas podem ser substituídas sem requerer a mudança das mais altas.

O modelo a seguir (Figura 2.7) divide as camadas do *framework* em três: camada de interface, camada de implementação núcleo e camada padrão.

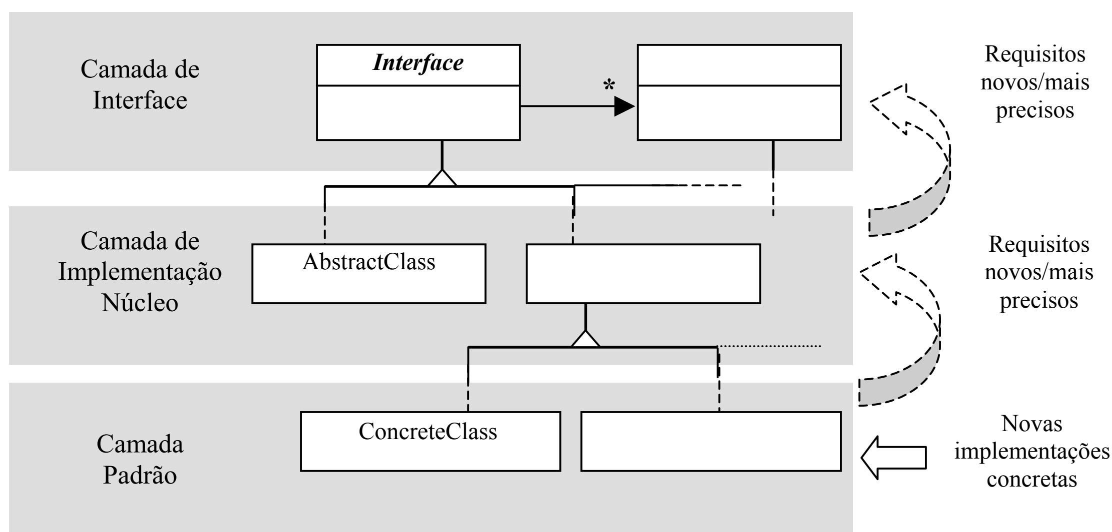

Figura 2.7 – Divisão do *framework* em camadas (Viljamaa, 2001).

A camada de interface consiste de interfaces (totalmente abstratas) que definem os componentes básicos do *framework*, seus serviços e seus relacionamentos através das assinaturas dos métodos. A camada de interface é totalmente independente da camada de baixo, por exemplo, a implementação da interface.

A camada de implementação núcleo define o comportamento padrão do *framework* por implementar parcialmente a camada de interface com as classes abstratas. A implementação abstrata tem a intenção de ser usada na maioria dos casos de desenvolvimento de aplicações. Se o desenvolvedor por alguma razão for desenvolver sua própria implementação através da implementação direta de algumas das interfaces da camada de interface, sua carga de trabalho aumenta bastante.

A camada padrão contém implementações completas (classes concretas) para circunstâncias comumente recorrentes (Viljamaa, 2001).

2.4.6. Evolução de *Frameworks*

Frameworks orientados a objetos, como outro *software*, evoluem através do tempo. Evolução controlada e previsível é importante, já que um *framework* reutilizável não só evolui como um *framework,* mas também causa a evolução da aplicação desenvolvida utilizando o *framework*.

A introdução de novos itens ou mudanças no requisito das aplicações leva a evolução da aplicação. Em um *framework* amadurecido, as camadas são bastante estáveis. Contudo, antes de um *framework* atingir a maturidade, mudanças

necessariamente ocorrem em sua estrutura. “Desenvolver *frameworks* é um processo iterativo” (Johnson e Foote, 1988). Quando o *framework* é utilizado várias vezes, o número de classes concretas na camada padrão aumenta. Depois que um número de classes concretas similares foi desenvolvida, alguns de seus comportamentos podem provavelmente ser generalizados em classes na camada padrão sem afetar o núcleo do *framework*. As linhas tracejadas na Figura 2.7 indicam as mudanças mais cruciais que resultam de novos requisitos funcionais descobertos na camada padrão, durante a adaptação do *framework*.

Como resultado, as interfaces e as classes abstratas nas camadas altas devem ser estendidas e mudadas. A tendência é que quando o *framework* se torna mais estável, as camadas do topo se tornam mais espessas e também mais estáveis (Viljamaa, 2001).

2.5. *FRAMEWORKS* E PADRÕES DE PROJETO

O uso de padrões de projeto é motivado pelo desejo de reutilização de projetos de qualidade. Gamma e outros (1995) apresentam a seguinte definição:

“Um padrão de projeto nomeia, abstrai e identifica os aspectos chaves de uma estrutura de projeto comum, tornando-se útil para criar um projeto orientado a objetos reutilizável. O padrão de projeto identifica as classes participantes e instâncias, seus papéis, suas colaborações e a distribuição de responsabilidades”. Cada padrão tem o foco em um problema particular de projeto orientado a objetos.

Em geral, um padrão tem quatro elementos essenciais:

O **nome do padrão**, que é o elemento utilizado para descrever um problema de projeto, sua solução e conseqüências em uma ou duas palavras.

O **problema** descreve quando aplicar o padrão. Ele explica o problema e seu contexto.

A **solução** descreve os elementos que constituem o projeto, seus relacionamentos, responsabilidades e colaboração.

As **conseqüências** são o resultados de aplicar o padrão. Isso inclui o impacto na flexibilidade do sistema, extensibilidade ou portabilidade.

Um padrão de projeto captura a essência de uma idéia que projetistas experientes tenham reutilizado muitas vezes para resolver um problema comum.

Os padrões de projeto são classificados de acordo com dois critérios. O primeiro critério, chamado propósito, reflete o que o padrão faz. Padrões podem ser de propósito

criacional, estrutural ou comportamental. Padrões criacionais preocupam-se com o processo de criação do objeto; padrões estruturais lidam com a composição de classes ou objetos e padrões comportamentais caracterizam a maneira na qual classes ou objetos interagem e distribuem responsabilidades.

O segundo critério, chamado escopo, especifica quando o padrão se aplica primariamente a classes ou a objetos. Padrões de classes lidam com relacionamentos entre classes e suas subclasses. Esses relacionamentos são estabelecidos através de herança, logo eles são estáticos - fixos em tempo de compilação. Padrões de objeto lidam com relacionamentos do objeto, os quais podem ser mudados em tempo de execução e são mais dinâmicos.

Um padrão de projeto é independente de domínios de aplicação e deve ser mapeado para a situação específica antes de ser implementado em classes específicas. Alguns padrões focam o mesmo problema diferindo somente no que diz respeito a sua generalidade. A documentação segue um modelo que visa os aspectos do problema, o contexto e a solução de diferentes perspectivas e diferentes níveis de abstração (Fayad e outros, 1999).

Em (Fayad e outros, 1999) padrões orientados a objetos são descritos como uma perspectiva nomeada de um assunto e para ser relevante um padrão deve expressar um tema geral recorrente que tenha provado ser útil.

Na Figura 2.8 são exibidos dois tipos de padrões, separados por linhas tracejadas e pontilhadas, representando diferentes perspectivas de um assunto.

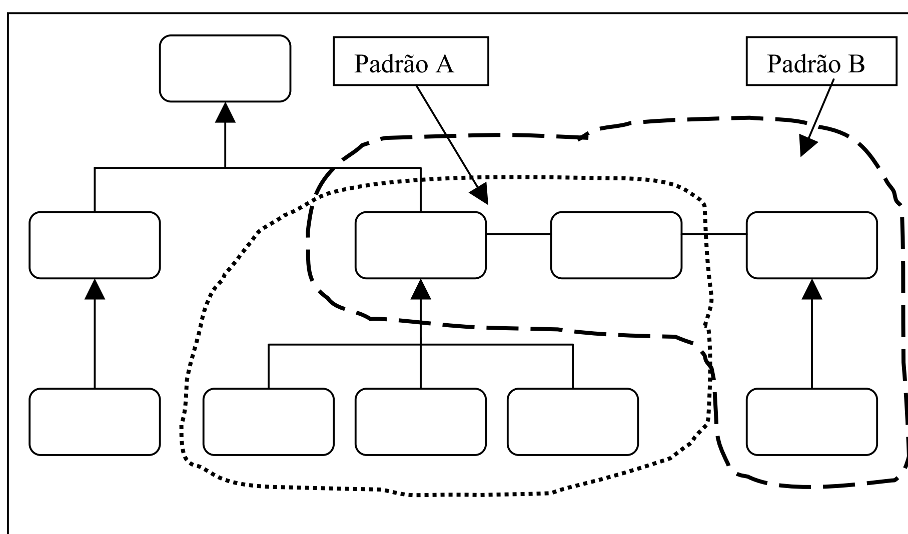

Figura 2.8 – Padrões representando diferentes perspectivas de um assunto (Fayad e outros, 1999)

2.5.1. Relacionamento entre *Frameworks* e Padrões de Projeto

Frameworks são frequentemente entendidos como sendo padrões de larga escala. Embora isto não seja verdade, o relacionamento entre padrões de projeto e *frameworks* é importante: padrões tipicamente têm o foco nos aspectos de flexibilidade de *software* e flexibilidade é exatamente o que é necessário em *frameworks* para possibilitar sua especialização nas aplicações reais.

Johnson (1992) apresenta uma discussão sobre a diferença entre padrões e *frameworks*:

- *Framework*: é um projeto reutilizável em soluções de problemas em algum domínio específico.
- Padrão: é orientado a problemas e não a soluções. Cada padrão descreve como resolver uma pequena parte de um grande problema de projeto. Padrões são perfeitamente indicados para ensinar como usar um *framework*.

Gamma e outros (1995) apresentam a seguinte diferenciação entre padrões e *frameworks*:

- Padrões de projeto são mais abstratos que *frameworks*: *frameworks* podem ser incorporados em código, enquanto que apenas exemplos de padrões podem ser incorporados em códigos.
- Padrões de projetos são elementos arquiteturais menores do que *frameworks*: um típico *framework* contém vários padrões de projeto, mas o inverso não é verdade.
- Padrões de projeto são menos especializados do que *frameworks*: *frameworks* sempre pertencem a um domínio em particular, enquanto que os padrões de projeto poderiam, em princípio, ser usados em qualquer tipo de aplicação.

Os vários papéis dos padrões de projeto e seus relacionamentos com diversos grupos são ilustrados na Figura 2.9 mostrando que padrões podem também ser utilizados como uma base para construção de *frameworks* de aplicação.

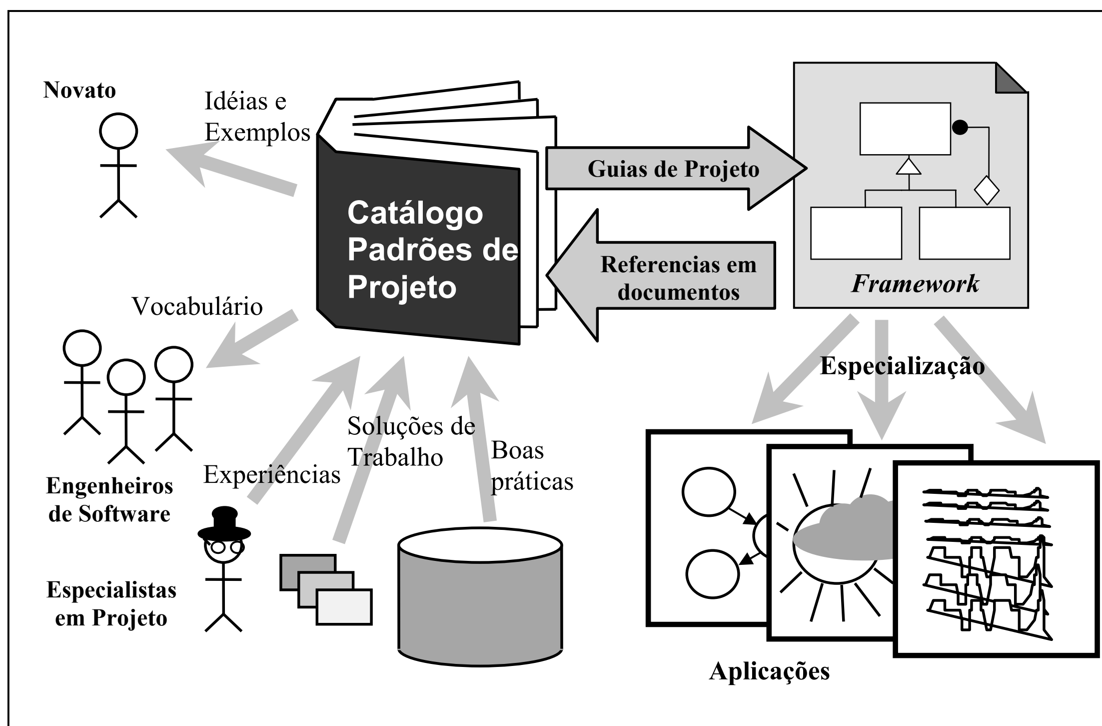

Figura 2.9 – Integração entre padrões e *frameworks* (Viljamaa, 2001).

Atualmente é amplamente aceito que padrões de projeto são bem adequados para documentar *frameworks*. Em (Johnson, 1992) é mostrado como uma documentação baseada em padrões pode ser elaborada. Segundo o autor, padrões podem descrever o objetivo do *framework*, podem permitir que programadores de aplicação utilizem um *framework* sem ter que entender em detalhes como ele funciona e podem ensinar muito sobre detalhes que estão embutidos.

Os padrões devem formar uma hierarquia onde o primeiro padrão descreve o domínio da aplicação do *framework* e fornece exemplos de como ele pode ser utilizado. Assim, eles podem ajudar a compreender o *framework* pelos exemplos dados de uma utilização típica.

Para o adaptador, os padrões oferecem a possibilidade de perceber os passos da adaptação (criando subclasses ou configurando classes do *framework*). Como resultado, o usuário vê sua adaptação numa perspectiva maior do que a de simples classes. Como mostrada na Figura 2.10, uma aplicação é decomposta em *frameworks*, que por sua vez são construídos com padrões e estes são formados por classes.

Padrões, também fazem o *framework* ser entendido, sem forçar o conhecimento pleno do seu código fonte (Viljamaa, 2001).

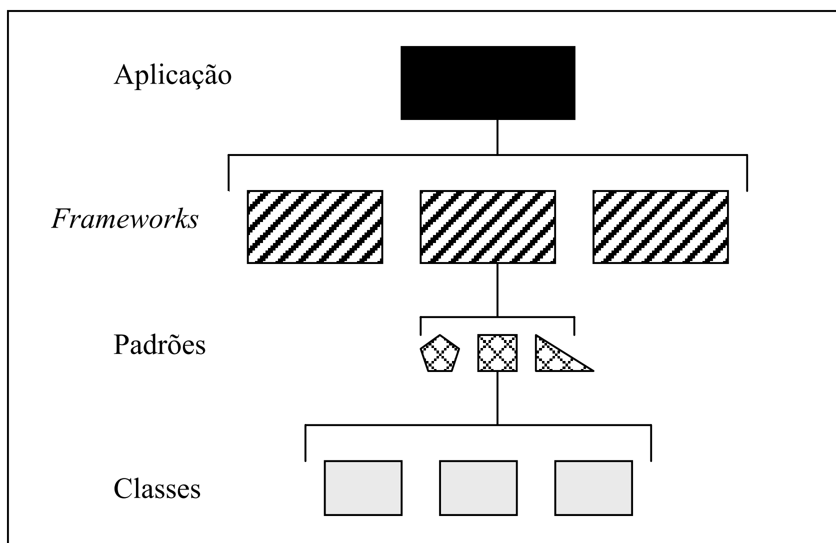

Figura 2.10 – A decomposição de uma aplicação baseada em *frameworks* e padrões (Mattsson, 1996).

2.6. DESENVOLVIMENTO DE *FRAMEWORKS*

No processo de desenvolvimento de um *framework* deve-se produzir uma estrutura de classes com a capacidade de adaptar-se a um conjunto de aplicações diferentes.

A principal característica buscada ao desenvolver um *framework* é a generalidade em relação aos conceitos e funcionalidades do domínio tratado. Além disso, é fundamental que a estrutura produzida seja flexível.

Pode-se afirmar que o desenvolvimento de um *framework* é diferente do desenvolvimento de uma aplicação padrão. A distinção mais importante é que *frameworks* tem que cobrir todos os conceitos relevantes do domínio enquanto uma aplicação se preocupa somente com os conceitos mencionados nos requisitos da aplicação (Bosh, 1999).

A complexidade em se desenvolver um *framework* deve-se aos seguintes fatores (Silva, 2000):

- Necessidade de considerar os requisitos de um conjunto significativo de aplicações de modo a torná-lo genérico.
- Necessidade de ciclos de evolução voltados a dotar a estrutura de classes do *framework* de alterabilidade (capacidade de alterar suas funcionalidades sem conseqüências imprevistas sobre o conjunto da estrutura) e extensibilidade

(capacidade de ampliar a funcionalidade presente sem conseqüências imprevistas sobre o conjunto da estrutura).

Em termos práticos, dotar um *framework* de generabilidade, alterabilidade e extensibilidade requerem uma cuidadosa identificação das partes que devem ser flexíveis e a seleção de soluções de projetos de modo a produzir uma arquitetura bem estruturada. Isto conduz a observação de princípios de projeto orientados a objetos.

Em termos ideais, um *framework* deve abranger todos os conceitos gerais de um domínio de aplicação, deixando apenas aspectos particulares para serem definidos nas aplicações específicas (Silva, 2000).

No contexto de desenvolvimento, a existência do *framework* estará sempre relacionada à existência de outros artefatos que são originadores do *framework*, originados a partir dele ou que exercem influência na definição das estruturas de classes.

A Figura 2.11 mostra as várias fontes de informação que influem na definição da estrutura de um *framework*: artefatos de *software* existentes, artefatos de *software* produzidos a partir do *framework* e o conhecimento do desenvolvedor do *framework* (ou a equipe de desenvolvimento). As setas representam o fluxo de informações que levam à produção da estrutura de classes do *framework*.

Como um *framework* geralmente não consegue ser uma construção completa de um domínio, mas sim uma descrição aproximada, é possível que a construção de aplicações sob um *framework* leve à obtenção de novos conhecimentos do domínio tratado, que talvez não estivessem disponíveis durante a construção do *framework*. Estas novas informações podem levar à necessidade de alterar o *framework*, causando a sua evolução como ilustra a Figura 2.11.

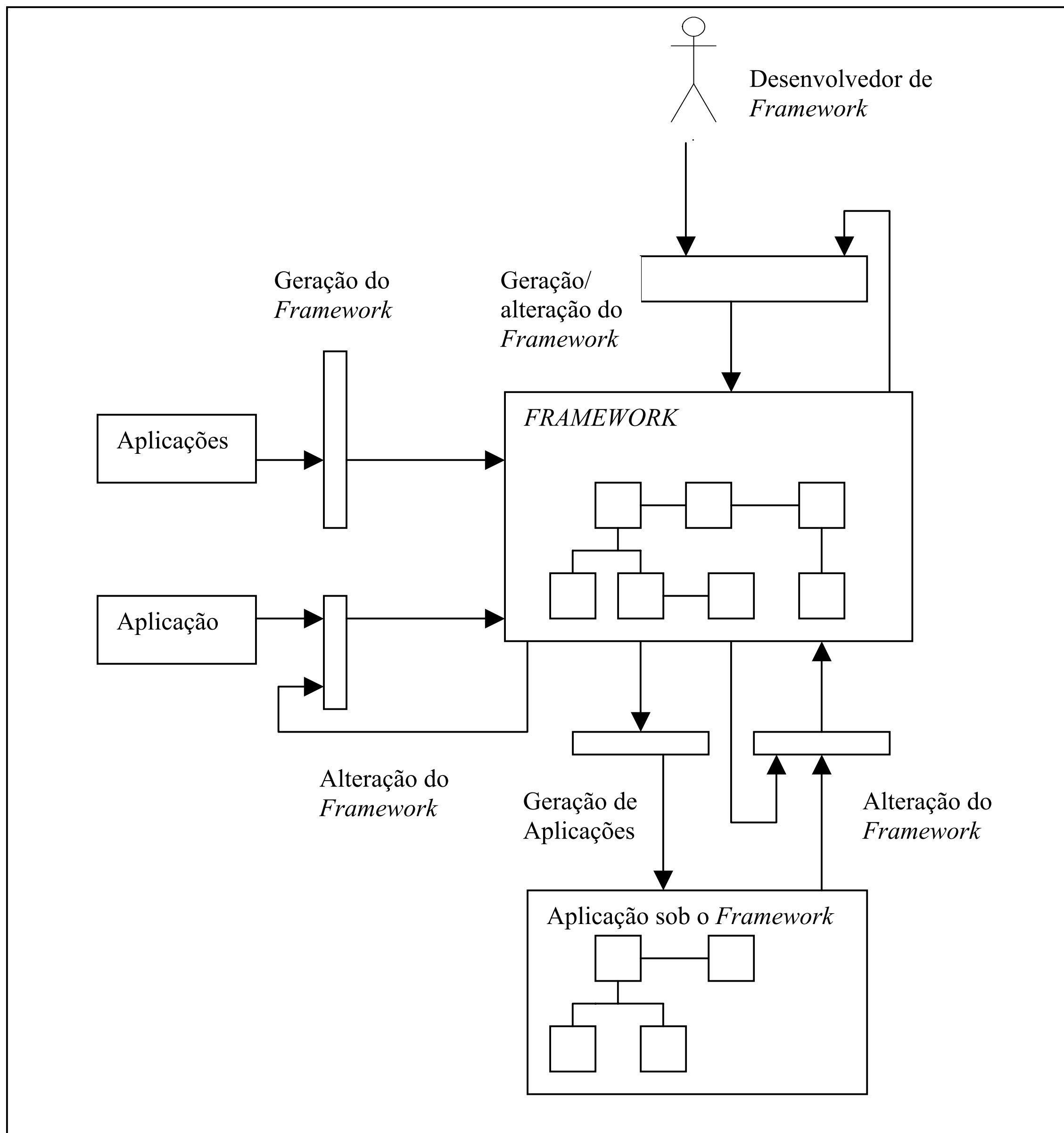

Figura 2.11 – Geração e alteração de *frameworks* e aplicações (Silva, 2000)

2.6.1. Recursos Humanos Envolvidos

Silva (2000) aborda que o desenvolvimento tradicional de aplicações envolve dois tipos de indivíduos: o desenvolvedor de aplicação e o usuário de aplicação (ambos podem corresponder a grupos de indivíduos com diferentes funções).

Desenvolvedores devem levantar os requisitos de uma aplicação, desenvolvê-la e entregá-la aos usuários, que por sua vez interagem com a aplicação apenas através de sua interface (Figura 2.12).

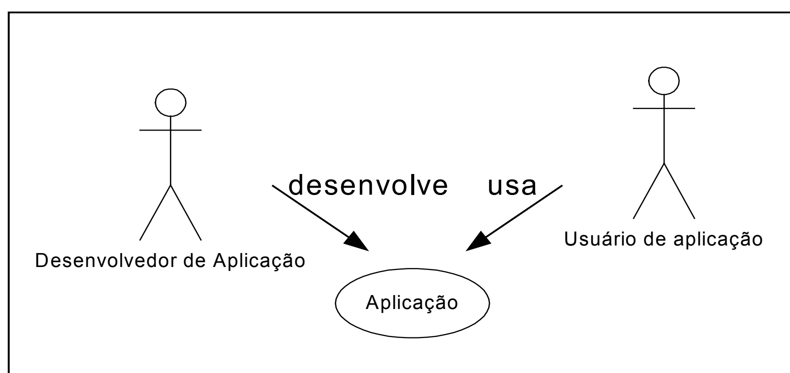

Figura 2.12 – Recurso do desenvolvimento tradicional de aplicações (Silva, 2000).

O desenvolvimento de *frameworks* introduz outro indivíduo, a saber, o desenvolvedor de *framework* (Figura 2.13). Nesse contexto, o papel do usuário de aplicação é o mesmo descrito acima. O papel do desenvolvedor de aplicações difere do caso anterior pela inserção do *framework* no processo de desenvolvimento de aplicações. Com isto, o desenvolvedor de aplicações é um usuário de um *framework* que deve estender e adaptar a estrutura deste *framework* para a produção de aplicações.

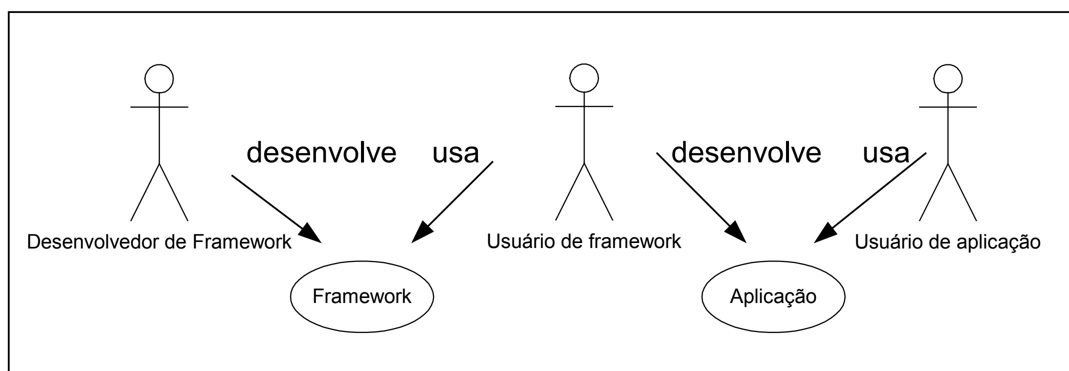

Figura 2.13 – Recursos do desenvolvimento de aplicações baseado em *frameworks* (Silva, 2000).

2.6.2. Metodologias de Desenvolvimento de *Frameworks*

Mattsson (2000) identificou que o processo de desenvolver e utilizar um *framework* orientado a objetos compreende três atividades macros mostradas na Figura 2.14. Este processo é comparado ao projeto orientado a objetos para uma aplicação (Figura 2.15).

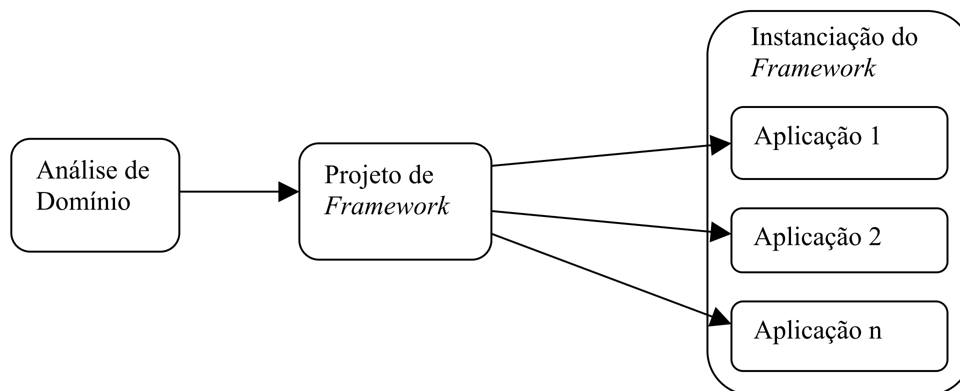

Figura 2.14 – Processo de desenvolvimento de *framework* (Markiewicz e Lucena, 2000).

Figura 2.15 – Projeto orientado a objetos tradicional (Markiewicz e Lucena, 2000).

De forma genérica as atividades macros são descritas a seguir:

- Análise de domínio - processo de identificar e organizar o conhecimento sobre alguma classe de problemas – o domínio do problema – para dar suporte à descrição e solução destes (Arango e Prieto-Díaz, 1991).
- Projeto de *framework* – o objetivo ao realizar essa atividade é obter um *framework* flexível. A atividade é árdua já que é difícil encontrar as abstrações corretas e identificar as partes estáveis e variáveis do *framework*. Para aumentar a extensibilidade e flexibilidade do *framework* padrões de projeto podem ser usados (Mattsson, 2000).
- Instanciação do *framework* – a instanciação difere dependendo do tipo de *framework*: caixa branca (o usuário constrói classes a partir das classes disponíveis), caixa preta (o usuário tem que escolher uma das classes fornecidas). Um *framework* pode ser instanciado uma ou mais vezes dentro da mesma aplicação ou instanciado em diferentes aplicações (Mattsson, 2000).

A seguir são mostrados alguns estudos e metodologias para o desenvolvimento de *frameworks*.

Johnson (1993) apresenta um processo chamado de Projeto Dirigido por Exemplos, composto de etapas de análise, projeto e teste. Ele afirma que as pessoas pensam de forma concreta e não de forma abstrata, assim a abstração de domínio é obtida a partir da generalização de casos concretos – as aplicações.

As abstrações são obtidas de uma forma *bottom up*, a partir do exame de exemplos concretos onde os aspectos semelhantes de diferentes aplicações podem dar origem a classes abstratas que agrupam as semelhanças, cabendo às classes concretas do nível hierárquico inferior fornecerem a especialização para satisfazer cada caso.

A forma tida como ideal para desenvolver um *framework* é :

- Analisar o domínio do problema - assimilar as abstrações já conhecidas, coletar exemplos de programas que podem ser construídos a partir do *framework* (mínimo de quatro ou cinco).
- Projetar abstrações que podem ser especializadas para abranger os exemplos. Recomenda-se utilizar padrões de projetos.
- Testar o *framework* usando-o para desenvolver os exemplos. Cada exemplo é uma aplicação separada e a execução do teste consiste em implementar o *software.*

Taligent (1994) propõe o desenvolvimento de *frameworks* estruturalmente menores e mais simples, que usados conjuntamente darão origem às aplicações. A justificativa para isto é que pequenos *frameworks* são mais flexíveis e podem ser reutilizados mais freqüentemente. Assim, a ênfase passa a ser o desenvolvimento de *frameworks* pequenos e direcionados a aspectos específicos do domínio.

A metodologia propõe a seqüência de quatro passos.

- Análise do domínio do problema: consiste em analisar o domínio e identificar os *frameworks* necessários para o desenvolvimento de aplicações, examinar soluções existentes e validar as informações com especialistas do domínio.
- Definição da arquitetura e projeto: consiste no refinamento da estrutura de classes obtida no passo anterior, aperfeiçoamento do projeto com o uso de padrões de projeto e validar as informações com especialistas do domínio.
- Implementação do *framework*: implementar as classes principais, testar o *framework* (gerando aplicações) e iterar para refinar o projeto.

- Disponibilização do *framework* com documentação e suporte técnico ao cliente.

A abordagem proposta por Pree (1999) é o Projeto Dirigido por *Hot Spot,* onde a essência é identificar os *hot spots* na estrutura de classes de um domínio e, a partir disso, construir o *framework* (Figura 2.16). As etapas são as seguintes:

- Na primeira etapa são feitas a identificação de classes com a ajuda de especialistas de domínio.
- Na segunda etapa são identificados os *hot spots*, que devem ser documentados com um tipo de documento criado pelo autor, chamado de cartão de *hot spots*.
- A terceira etapa, projeto do *framework*, consiste em modificar a estrutura de classes inicialmente definida, de modo a comportar a flexibilidade requerida. Essa etapa é centrada no uso de padrões de projeto para garantir a flexibilidade.
- A quarta etapa consiste num refinamento da estrutura do *framework* a partir de novas intervenções de especialistas do domínio. Se após isto o *framework* for avaliado como satisfatório, está concluída uma versão do *framework*, senão, retorna-se à etapa de identificação de *hot spots*.

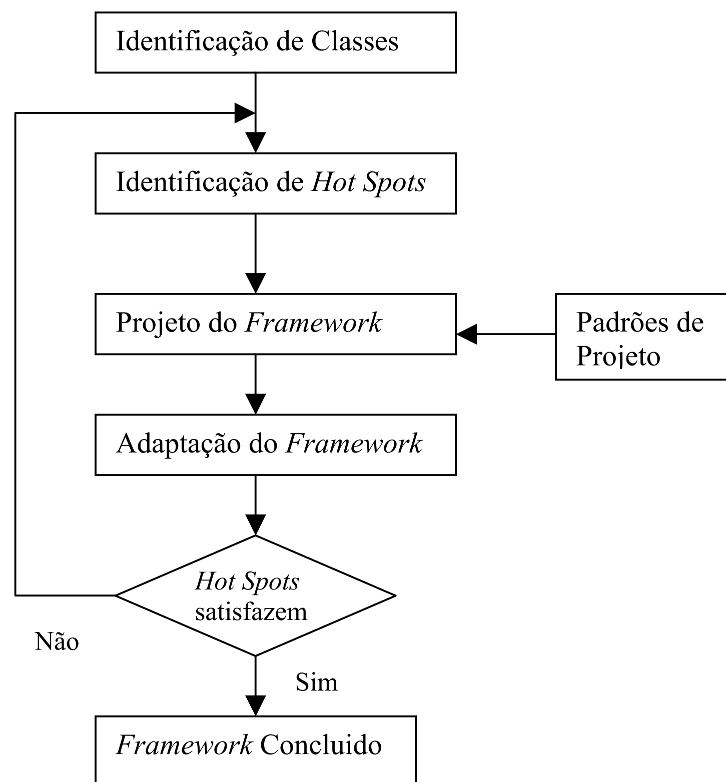

Figura 2.16 – Etapas do projeto dirigido por *hot spot* (Silva, 2000)

Bosh (1999) apresenta um modelo para desenvolvimento de *framework* composto de seis etapas:

- Na primeira etapa é feita a análise de domínio para descrever o domínio coberto pelo *framework*, capturar requisitos e conceitos principais dando como resultado o modelo de análise do domínio.
- Na segunda etapa é criado o projeto arquitetural do *framework* que tem como entrada o modelo da análise de domínio. O projetista tem que decidir por um estilo arquitetural para moldar o *framework* e assim o projeto de alto nível do *framework* é criado.
- Na terceira etapa, o projeto é refinado e classes adicionais são projetadas.
- Na quarta etapa, são implementadas as classes concretas e abstratas.
- Na quinta etapa, o *framework* é testado para determinar se o *framework* fornece a funcionalidade pretendida e também, para avaliar a usabilidade do mesmo. Para isso, desenvolvem-se aplicações testes.
- Na sexta etapa, o *framework* é documentado. Deve-se descrever como usar o *framework*, fornecer um manual do usuário e um documento do projeto que descreve como o *framework* funciona.

Schmid (1999) propõe que *frameworks* sejam construídos em ciclos por generalização sistemática baseada no modelo de classe de uma aplicação fixa.

A construção do *framework* consiste em quatro atividades (Figura 2.17):

- Atividade de modelagem da aplicação: é desenvolvido o modelo de classe de uma aplicação fixa pertencente ao domínio do *framework*. O modelo de classes contém instâncias fixas dos conceitos e entidades do domínio do *framework* e os relacionamentos fixos do mesmo.
- Análise e especificação do *hot spot*: é dividida em análise de alto nível do *hot spot* e análise e especificação detalhada do *hot spot*. A primeira identifica e coleta todos os *hot spots* do domínio e os descreve brevemente. A segunda é feita separadamente para cada *hot spot*, consistindo em analisar um *hot spot* e descrever suas características baseado no conhecimento do domínio e protótipos de aplicações.
- Projeto de alto nível do *hot spot*: o *hot spot* é implementado por um subsistema de *hot spot* que contem classes bases (abstratas), classes concretas derivadas e possivelmente classes adicionais e relacionamentos. O uso de

padrões de projeto ajuda na determinação dos detalhes da estrutura do subsistema.

- Transformação por generalização: transforma a estrutura de classes e a generaliza no que diz respeito a algum aspecto. O resultado é que aspectos que estavam estáveis (*frozen*) passam a refletir um subsistema de *hot spot*. Essa atividade tem o objetivo de introduzir a variabilidade e flexibilidade do *hot spot* na estrutura de classes.

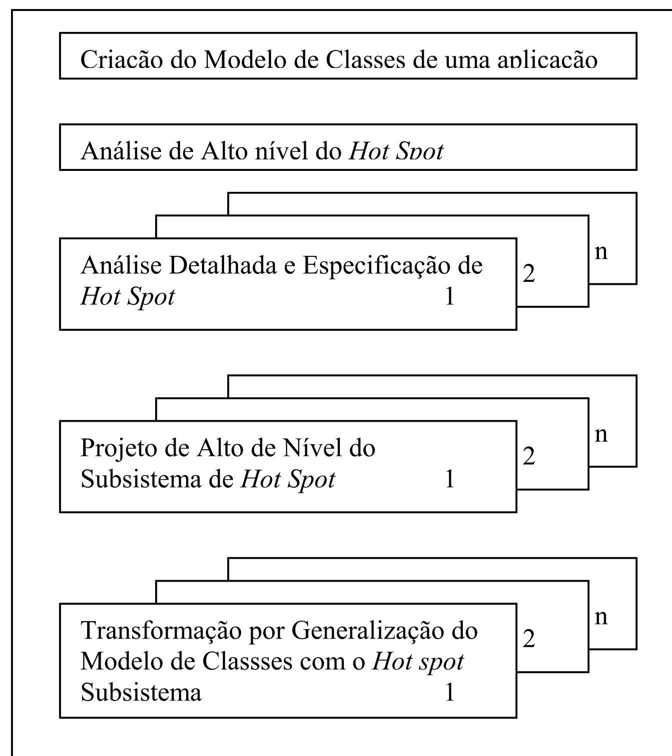

Figura 2.17 – Atividades do projeto de *framework* (Schmid, 1999)

2.6.2.1. Comparação Entre as Metodologias

As metodologias apresentadas (Sessão 2.4.2) possuem características em comuns. Os processos apresentados por Pree (1999), Schmid (1999) e Johnson (1993) iniciam com um modelo de aplicação já pronta e incluem a flexibilidade desejada mais tarde.

Em Bosh (1999) e Taligent (1994) a abordagem inicia com o modelo da análise de domínio, que torna a detecção dos *hot spots* do *framework* mais previsíveis.

Outro ponto em comum é a recomendação do uso de padrões de projeto em Pree (1999), Taligent (1994) e Schmid (1999), que ajudam na documentação e compreensão

do projeto facilitando a reutilização. Entretanto não estabelecem especificamente como utilizar os padrões.

Em todas as abordagens é ressaltada a importância da documentação para o entendimento do *framework* e sua utilização. Os aspectos que diferenciam uma metodologia da outra é a ênfase dada a algum ponto específico, que não necessariamente são conflitantes.

O que diferencia o Projeto Dirigido por Exemplos é a ênfase à análise do maior número possível de aplicações já desenvolvidas.

A característica principal das metodologias Generalização Sistemática e Projeto Dirigido por *Hot Spots* é a ênfase dada sobre os pontos de flexibilidade do *framework*.

Em Bosh (1999) a ênfase encontra-se na análise de domínio, pois será a partir da captura dos requisitos e identificação dos conceitos que o escopo do domínio será delineado e o *framework* será desenvolvido.

Já a metodologia proposta por Taligent (1994) diferencia-se das demais pelos princípios usados para o desenvolvimento tais como a visão de desenvolver um conjunto de *frameworks* menores e a preocupação de tornar o desenvolvimento das aplicações mais simples.

As metodologias propostas, entretanto, descrevem como produzir *frameworks* sem estabelecer um processo detalhado de construção de modelos que dirija o desenvolvimento, nem estabelecem um conjunto de mecanismos de descrição que contenham o projeto do *framework*. Em Silva e Price (1997) e Silva (2000) é proposto que seja utilizado um conjunto de técnicas de modelagem para *framework* baseado em técnicas usadas por metodologias de Análise e Projeto Orientados a Objetos.

2.7. DIFICULDADES E PROBLEMAS NO DESENVOLVIMENTO DE *FRAMEWORKS*

Segundo Viljamaa (2001) Os benefícios de desenvolver um *framework* são conseguidos se o *framework* puder ser utilizado várias vezes. A usabilidade de um *framework* é, contudo, difícil de prever de antemão: por exemplo, mudanças inesperadas nos requisitos do domínio podem fazer com que o *framework* fique desatualizado mais cedo que o esperado. Conseqüentemente, a decisão de começar a desenvolver um *framework* é difícil. O custo do desenvolvimento para um *framework* é muito maior que

para desenvolver uma aplicação no mesmo domínio e o retorno do investimento está em futuras economias de esforços no desenvolvimento.

Roberts e Johnson (1998) explicam que desenvolver um bom *framework* é tarefa árdua. Um *framework* deve ser simples o bastante para ser aprendido, deve prover características suficientes que possam ser usadas rapidamente e métodos adaptáveis para as características que podem ser mudadas. Ele deve incorporar uma teoria do domínio do problema, e será sempre o resultado da análise do domínio.

Um processo de desenvolvimento que oriente o desenvolvedor é necessário para o cumprimento dessa tarefa, cujos principais problemas são descritos a seguir.

2.7.1. Detecção de *Hot Spot*

Uma vez que o escopo do *framework* está decidido, deve-se determinar quais de suas propriedades devem ser flexíveis (configuráveis, extensíveis ou reimplementáveis). Assim, deve-se definir o conjunto apropriado de *hot spots* para o *framework*. É difícil definir quais partes do domínio podem ser mudadas e quais são estáveis. É importante evitar flexibilidade desnecessária, por que flexibilidade, a despeito do seu benefício, torna a compreensão e a manutenção do *framework* mais difícil.

2.7.2. Implementação e Validação

Uma das mais árduas partes na construção de *frameworks* é escolher a maneira adequada de implementar os *hot spots*. Como os padrões de projeto indicam, por exemplo, há geralmente várias possibilidades para implementar um *hot spot* e todas têm vantagens e desvantagens. Embora padrões ajudem a avaliar as diferentes alternativas, o problema é que os métodos para organizar e pesquisar padrões não são ainda suficientes.

Verificar o comportamento de um *framework* não é tarefa fácil: componentes do *framework* não podem ser validados isolados de suas instâncias específicas. A prática atual é especializar aplicações exemplo e validar o *framework* para testá-las. Dentro desse enfoque, contudo, é árduo distinguir defeitos do *framework* de defeitos da aplicação. Assim, para reduzir as incertezas um processo de desenvolvimento é desejável.

Lançar um *framework* imaturo pode ter conseqüências graves na manutenção e uso do *framework* e aplicações instanciadas. É, contudo, difícil determinar critérios

corretos de lançamento. Em linhas gerais, um *framework* deve ser reutilizável, razoavelmente estável dentro do domínio e bem documentado antes do seu lançamento. Contudo, definir e assegurar estas propriedades precisamente parecem impossíveis. O domínio do *framework* geralmente é pouco estável; tipicamente evolui constantemente. Também decidir quando o *framework* é suficientemente bem documentado é difícil, já que não há geralmente método de documentação cobrindo todos os aspectos do *framework* (Viljamaa, 2001).

2.7.3. Evolução

Embora um determinado *framework* possa ser considerado estável quando lançado, certamente haverá manutenção, pois não há como evitar mudanças nos requisitos da aplicação ao longo do tempo. Tais mudanças vão requerer um profundo entendimento do *framework*.

A evolução do *framework* pode tornar sua estrutura (interfaces, hierarquia de classes, e outros) complexa de gerenciar e entender.

À medida que ele evolui e novas instâncias são criadas, novas abstrações que devem fazer parte do *framework* podem ser criadas e a estrutura do projeto precisará ser melhorada (Viljamaa, 2001).

2.7.4. Aplicabilidade

Um dos mais complexos problemas com o uso de *frameworks* é entender o domínio do *framework* e sua aplicação ao projeto de *software* a ser construído (Fayad e outros. 1999): pode se questionar se ele cobre todos os grandes requisitos e se faz a correlação entre poder e simplicidade. A arquitetura da aplicação desejada deve caber na arquitetura do *framework*. É relativamente fácil verificar quando um *framework* contém alguma característica específica, mas é difícil determinar quando o *framework* é apropriado como um todo (Viljamaa, 2001).

2.7.5. Complexidade e Curva de Aprendizagem

Frameworks freqüentemente se tornam complexos e difíceis de entender. Isso é verdade especialmente se a localização dos *hot spot* não está claramente definida e se a sua instanciação é ambígua. Também, porque as classes no *framework* são projetadas

para trabalhar juntas e, além disso, as classes mais importantes do *framework* são abstratas, o que dificulta o aprendizado.

Segundo Johnson (1992) a utilização de exemplos pode exercer um papel importante no ensino de como usar um *framework*. Examinando adaptações de *frameworks* já existentes é fácil ter conhecimento do que o *framework* é capaz. Contudo é difícil produzir compactos e compreensíveis programas de exemplo.

3. DESENVOLVIMENTO ITERATIVO E INCREMENTAL

Neste capítulo será abordado o processo de desenvolvimento iterativo e incremental e apresentada a nossa proposta de desenvolvimento.

3.1. VISÃO GERAL

Um ciclo de vida iterativo se baseia no aumento e refinamento sucessivo de um sistema através de múltiplos ciclos de desenvolvimento da análise, projeto, implementação e de teste.

Ao contrário do clássico ciclo de vida em cascata, onde cada atividade é executada uma única vez para o conjunto inteiro de requisitos do sistema, um ciclo de vida iterativo está baseado no aperfeiçoamento sucessivo de um sistema.

É através dos vários ciclos, no desenvolvimento iterativo, que o sistema é refinado e ajustado ou que são adicionados novos requisitos. Cada ciclo trata de um conjunto relativamente pequeno de requisitos (Larman, 1999).

O objetivo do desenvolvimento incremental é adicionar funcionalidades a um sistema durante vários ciclos de liberação de produto. O produto pode ser composto de múltiplos ciclos de desenvolvimento iterativo e cada produto contém mais funcionalidades que o anteriormente gerado.

Desenvolvimento baseado em *frameworks* apresenta vantagens em relação a abordagens tradicionais de desenvolvimento de *software*, mas há obstáculos que dificultam sua aplicação. Silva (2000) identificou que os principais problemas são:

- A complexidade de desenvolvimento, que diz respeito à necessidade de produzir uma abstração de um domínio de aplicações adequada a diferentes aplicações.
- A complexidade de uso, que diz respeito ao esforço requerido para aprender a desenvolver aplicações a partir de um *framework*. Além é claro, da flexibilidade que o *framework* deve fornecer.

Verifica-se que o processo de desenvolvimento do *framework* ocorre de forma iterativa. Sua estrutura de classes é refinada ciclicamente em função da aquisição de informações ou busca de soluções de projeto mais adequadas.

Porém, as metodologias propostas atualmente estão voltadas a produzir código, fazendo com que o resultado do processo de desenvolvimento seja o código do *framework*. O entendimento do *framework* se baseia principalmente na descrição textual

dentro do código, no próprio código do *framework* e no código das aplicações geradas (Silva, 2000).

Há propostas de mecanismos auxiliares de descrição, citadas no Capítulo 2, que entretanto se voltam principalmente para a questão de como utilizar os *frameworks,* não descrevendo completamente o seu processo de desenvolvimento e conseqüentemente, dificultando o entendimento global do *framework* a partir de suas notações.

Uma solução é a utilização da abordagem de desenvolvimento iterativo e incremental aliado ao uso de técnicas de análise e projeto orientado a objetos para o desenvolvimento de *frameworks*. Com isso, obtém-se as vantagens de que a complexidade não se torna incontrolável, a realimentação é gerada mais cedo no processo e a documentação de cada fase serve para o entendimento do *framework* e sua futura utilização no desenvolvimento de aplicações.

O processo iterativo e incremental ajuda a resolver algumas das questões relativas ao desenvolvimento de *frameworks* abordadas, na medida em que limita a descrição de um domínio à sub-domínios e reduz o tempo requerido para seu desenvolvimento.

3.2. ETAPAS DO PROCESSO PROPOSTO PARA O DESENVOLVIMENTO ITERATIVO E INCREMENTAL

O processo de desenvolvimento proposto (Figura 3.1) inicia com a análise de domínio, onde passa para fase de construção, em seguida são feitos vários ciclos de desenvolvimento e a cada ciclo, os requisitos são adicionados ou refinados, é feita uma análise, os *hot spots* são avaliados e o projeto é alterado para refletir os novos requisitos ou refinamentos.

A cada ciclo é implementado um *framework* e criada uma instância de exemplo. Deve-se elaborar um roteiro ao final de cada ciclo para facilitar o entendimento e o instanciamento de aplicações a serem geradas a partir do *framework* produzido.

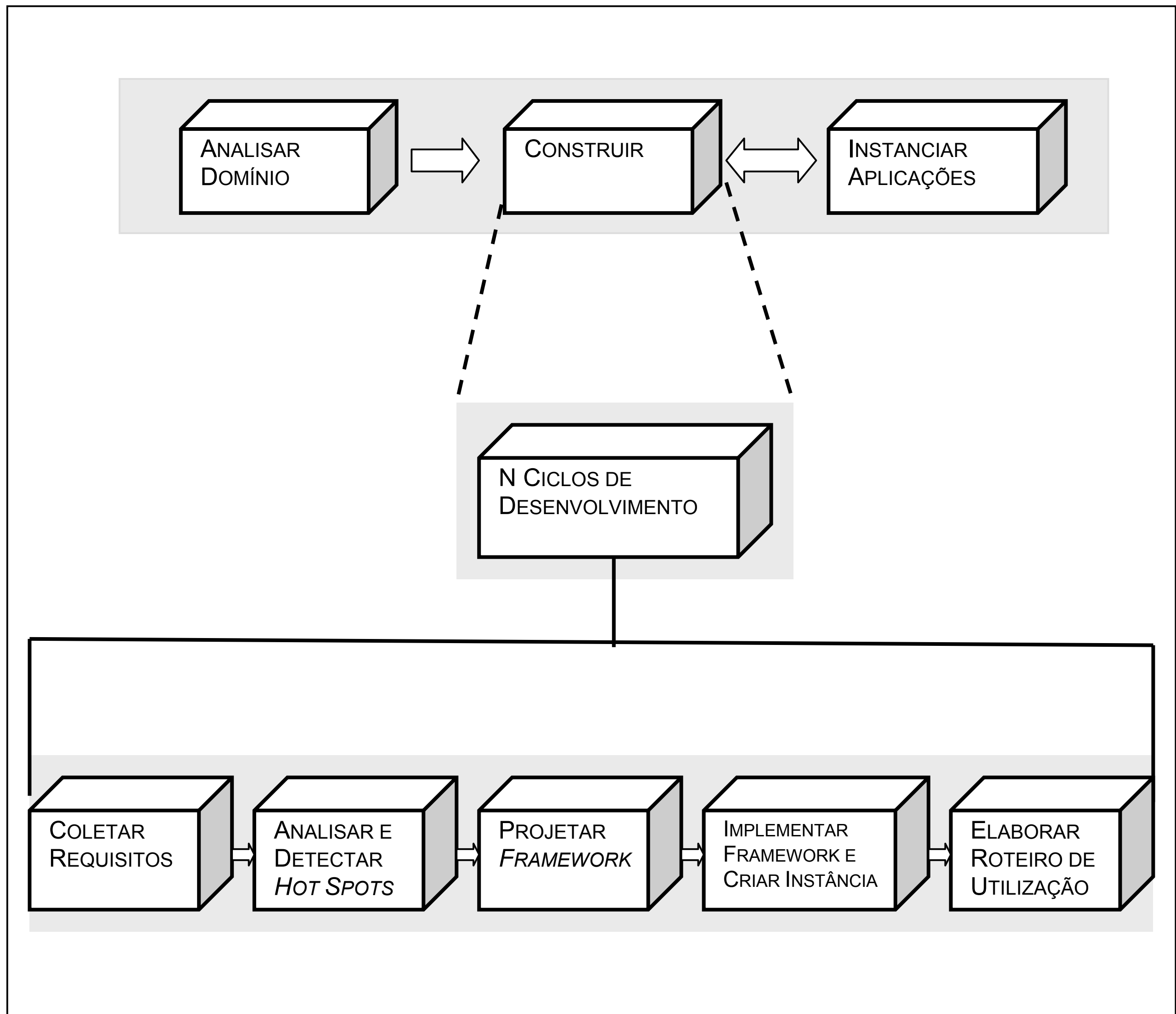

Figura 3.1 – Processo de desenvolvimento iterativo e incremental

3.2.1. Passo 1 - Analisar Domínio

A análise de domínio fornecerá a entrada para o desenvolvimento do *framework*.

Trata-se de um processo através do qual a informação a ser usada no desenvolvimento do sistema é identificada, capturada e organizada. Pode ser vista como uma análise ampla que tenta obter os requisitos do domínio do problema, incluindo requisitos futuros.

Para Neighbors (1989), a análise de domínio é uma tentativa de identificar os objetos, operações e relações que os especialistas de um domínio percebem ser importantes para a descrição deste domínio.

Arango (1991) propõe uma abordagem prática para a análise de domínio, baseada no paradigma de engenharia do conhecimento, o qual consiste nas etapas de identificar as fontes de conhecimento, adquirir o conhecimento existente e representar o conhecimento através de um modelo de domínio.

Mattsson (2000) propõe que na preparação deve se obter os requisitos iniciais através da observação, entrevistas e recursos materiais. Dependendo do domínio do problema, diferentes tipos de fontes de informação são usados para obter conhecimento sobre o domínio. Exemplos de fontes são as experiências anteriores sobre o domínio, especialistas do domínio e padrões existentes.

Há dois documentos que são importantes nessa fase e devem ser o resultado da análise de domínio: o escopo do domínio e um modelo estático contendo objetos (conceitos) importantes do domínio.

O escopo é de grande utilidade na fase de coleta de requisitos, pois ajudará a identificar se um requisito está no domínio e é válido ou se está fora e portanto é inválido (Landin e Niklasson, 1995).

É importante salientar que se deve delimitar o escopo do domínio de um *framework*. Um grande escopo torna o *framework* reutilizável para uma maior variedade de aplicações, mas ao mesmo tempo o *framework* facilmente torna-se muito grande e difícil de gerenciar. Também, a maioria das características de um *framework* robusto serão provavelmente pouco utilizadas em grande parte das aplicações derivadas. Por outro lado, um escopo muito pequeno limita a utilização do *framework* e pode não justificar o esforço gasto para o seu desenvolvimento.

O modelo estático deve conter as classes e objetos do domínio que fazem parte do mundo real e que são comuns nas aplicações deste domínio. Não deve, portanto, lidar com detalhes, nem enfatizar o processo de desenvolvimento.

O modelo do domínio servirá de documento para comunicação futura entre o desenvolvedor de aplicações e o usuário.

Resumo:

Passo 1: Análise de Domínio

Entrada: Obter informações sobre o domínio do problema através da observação, entrevistas, padrões já existentes e através de pessoas com conhecimento no domínio.

Procedimentos:

- Descrever suscintamente o objetivo do framework.
- Identificar seus limites.
- Descrever seus benefícios.
- Representar os objetos do domínio no modelo estático.

Saídas:

- Modelo do Domínio
  - Definição do escopo do domínio.
  - Modelo estático preliminar.

3.2.2. Passo 2 - Construir *Framework*

Este passo tem por finalidade desenvolver um *framework* completo. É composto de vários ciclos de desenvolvimento que são descritos nos tópicos a seguir.

3.2.2.1. Passo 2.1 – Coletar Requisitos

Requisitos são descrições das necessidades ou dos desejos para um produto. O objetivo básico da fase de requisitos é identificar e documentar o que realmente é necessário, em uma forma que comunica claramente essa informação ao cliente e aos membros da equipe de desenvolvimento (Larman, 1999).

Os requisitos definidos por apenas uma pessoa não refletirão as reais necessidades que deverão ser atendidas pelo *framework*, assim é interessante a composição de uma equipe formada por pessoas da área de desenvolvimento, área usuária e se possível pessoas com experiência no domínio. As informações poderão ser obtidas através de entrevistas e reuniões. Informações valiosas também podem ser obtidas a partir de aplicações já existentes no domínio.

Uma estratégia proposta por Landin e Niklasson (1995) é listar separadamente os requisitos de cada aplicação, identificar os requisitos gerais das aplicações e colocá-los na lista de requisitos do *framework* (Figura 3.2).

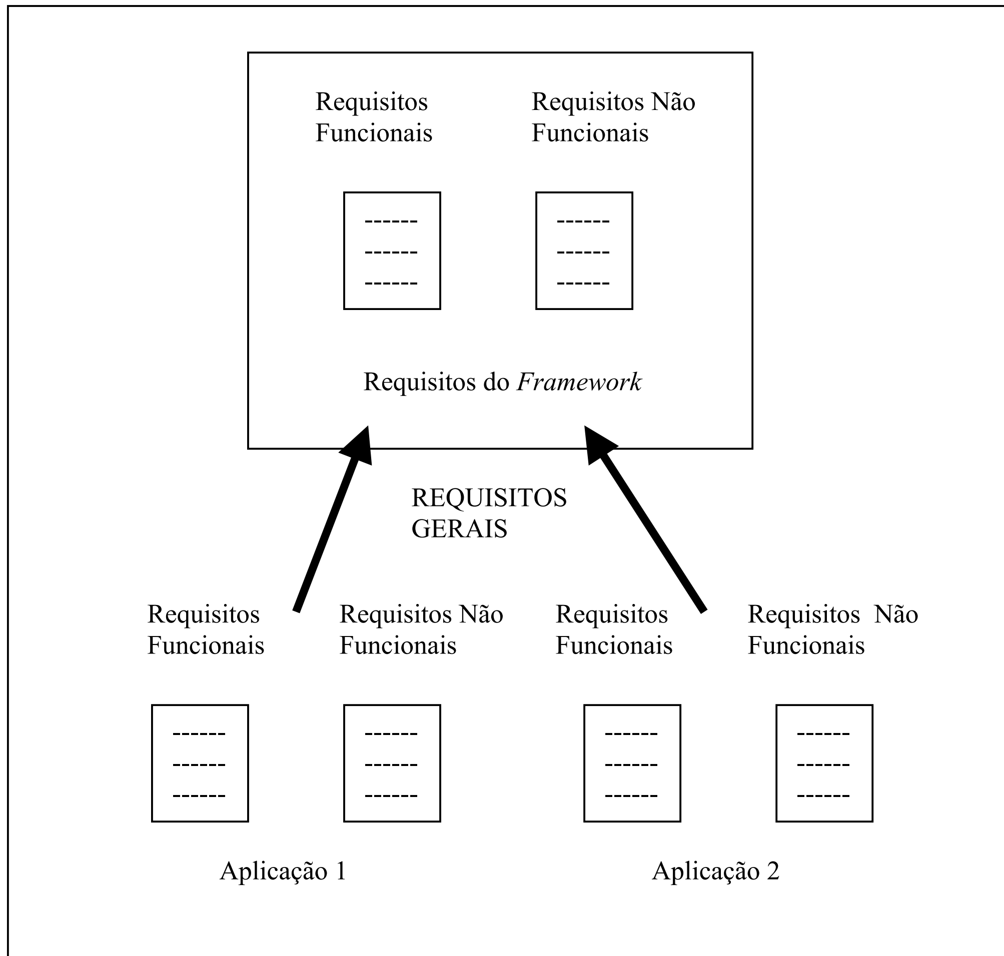

Figura 3.2 – Separação dos requisitos nas aplicações e no *framework*.

Caso não sejam identificadas aplicações relacionadas para se extrair os requisitos do *framework*, estes poderão ser abstraídos da análise do domínio em que residirá o *framework*, pois a análise fornece ao desenvolvedor um entendimento detalhado dos conceitos relevantes do domínio e da funcionalidade que o *framework* possuirá (Figura 3.3).

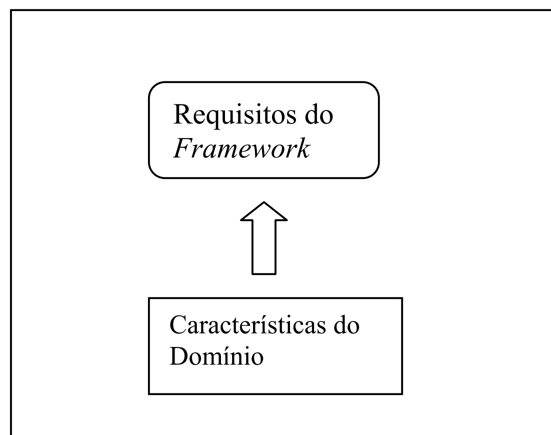

Figura 3.3 – Identificação dos requisitos do *framework* a partir do domínio

Uma técnica que melhora a compreensão dos requisitos é a criação de casos de uso (descrições narrativas de processos do domínio). Estes são identificados através da revisão dos documentos da especificação de requisitos.

Os ciclos de desenvolvimento podem ser organizados em torno dos casos de uso (Figura 3.4). A cada ciclo o caso de uso pode ser expandido para representar novos requisitos ou são criadas versões simplificadas quando o caso de uso é muito complexo para ser atacado em único ciclo (Larman, 1999).

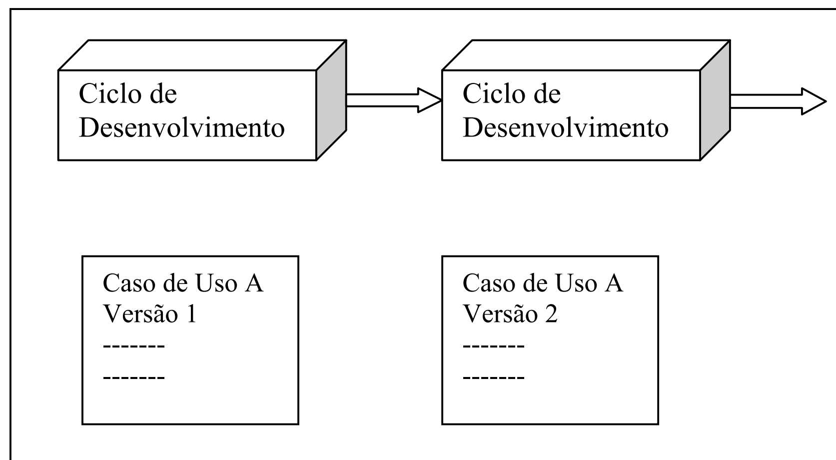

Figura 3.4 – Alocando casos de uso a ciclos de desenvolvimento (Larman, 1999)

Os casos de uso são divididos em casos uso concretos e abstratos. Os caso de uso concretos são iniciados por um ator para produzir um resultado e os abstratos contém uma seqüência de ações que são compartilhadas por outros casos de uso.

Em se tratando de *frameworks*, cujo uso não será por usuários finais (Capítulo 2), o mais adequado durante o seu desenvolvimento é a criação de casos de uso abstratos, já que estes não são utilizados por atores, mas por casos de uso concretos (da aplicação) ou outros casos de uso abstratos.

Um caso de uso concreto ou abstrato pode usar vários casos de uso abstratos para completar a seqüência de ações (Figura 3.5).

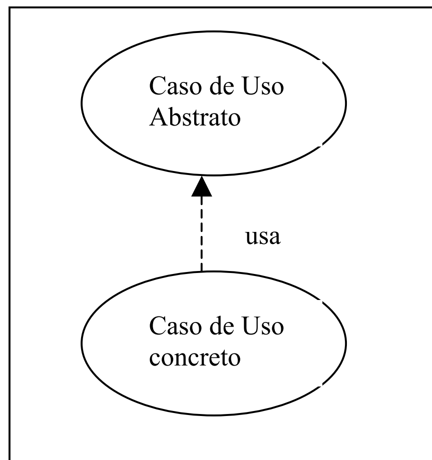

Figura 3.5 – Casos de uso abstrato e concreto

A qualidade do modelo de caso de uso terá impacto na qualidade de outros documentos subseqüentemente produzidos no processo de desenvolvimento. Assim, as seguintes recomendações devem ser observadas (Bente e outros, 2001):

- O caso de uso deve ser fácil de ler.
- As descrições não devem incluir qualquer menção ao projeto ou implementação.
- Eventos que não estão relacionados ao objetivo do caso de uso não devem ser descritos.
- A descrição deve ser completa.
- A estrutura deve ser correta e não ambígua.
- A terminologia deve ser consistente.

Também a própria especificação de requisitos para servir de base a um produto de boa qualidade deve satisfazer a uma série de características de qualidade, a seguir citamos as características propostas por Pádua (2001), mas não entraremos nos detalhes de como executar ou avaliar a qualidade dos requisitos.

- Correta: todo requisito presente realmente é um requisito do produto a ser construído.
- Precisa: todo requisito presente possui uma única interpretação.
- Completa: reflete todas as decisões de especificação que foram tomadas.
- Consistente: não há conflitos em nenhum subconjunto de requisitos presentes.

- Verificável: todos os requisitos têm um processo finito, que possa ser executado por uma pessoa ou máquina.

Como os requisitos serão utilizados para desenvolver o *framework* as inconsistências e ambigüidades deverão ser encontradas e resolvidas nessa etapa.

A documentação gerada nessa fase é a Especificação dos Requisitos onde os requisitos devem ser descritos de forma sucinta, sem entrar nos detalhes. A sua definição produz uma lista de todos os requisitos funcionais e não funcionais.

Deve-se observar que os requisitos não funcionais de um *framework* são diferentes dos requisitos não funcionais das aplicações, pois os usuários do *framework* são os desenvolvedores. Assim, por exemplo, a linguagem de programação, os padrões e outros itens podem fazer parte destes requisitos.

Resumo:

Passo 2.1 : Coleta de Requisitos

Entrada: Modelo do domínio, aplicações existentes, usuários e pessoas com conhecimento no domínio

Procedimentos:

- Obter informações sobre os requisitos a partir de diversas fontes, tais como grupos de usuários e pessoas com conhecimento no domínio através de entrevistas e reuniões.
- Verificar, nas aplicações existentes, os requisitos funcionais e não funcionais comuns e gerar a lista de requisitos do *framework*.
- Verificar características relevantes do domínio e abstraí-las para a lista de requisitos do *framework.*
- Criar casos de uso abstratos, através da revisão dos documentos da especificação de requisitos.

Saída:

- Modelo de Requisitos
  - Especificação de Requisitos.
  - Casos de Uso abstratos.

3.2.2.2. Passo 2.2 – Análise e Detecção de *Hot Spots*

O desenvolvedor obtém considerável conhecimento sobre o *framework* quando o domínio do problema é identificado e os requisitos já definem que parte do domínio do problema o *framework* vai atender. Isso torna possível que ele passe para a etapa de análise e detecção de *hot spot* com maior segurança.

Nessa etapa será delineado um modelo que preencha os requisitos levantados na etapa anterior bem como será coletado e descrito os aspectos variáveis do domínio do *framework*.

No primeiro ciclo, o modelo estático preliminar da fase de análise de domínio servirá de entrada para gerar o modelo de objetos da análise (diagrama de classes). O refinamento fará com que novas classes possam ser acrescentadas ou classes já existentes sejam removidas. A cada ciclo subseqüente, esse modelo gerado será novamente refinado, em função dos requisitos levantados na etapa de coleta de requisitos.

No modelo de objetos da análise nenhuma consideração deve ser feita a respeito do ambiente de implementação. A ênfase está nos conceitos do domínio e não nas entidades de *software*.

Uma razão para isso é que os detalhes da implementação pode colocar fora de foco o problema que o *framework* se propõe a resolver.

O modelo deve incluir abstrações do mundo real, abstrações de alto nível e o relacionamento entre essas abstrações. A idéia é capturar os conceitos de importância e descartar aqueles sem importância.

A diferença entre a atividade de análise do *framework* e de uma aplicação está no fato de que os conceitos do *framework* são generalizações dos conceitos das aplicações, ou seja, os conceitos que devem ser comuns nas aplicações são trazidos para o *framework*. É importante ressaltar que a introdução de novas abstrações deve estar dentro do domínio do *framework*.

A detecção de *hot spots* será efetuada verificando-se no modelo de domínio, modelo de requisitos ou nas aplicações já existentes, que aspectos fixos podem ser flexibilizados nas aplicações a serem desenvolvidas a partir do *framework* proposto.

Para cada flexibilidade identificada devem ser preenchidos os dados abaixo, os quais formarão o catálogo de *hot spot*:

1. Número

2. Nome do *hot spot*
3. Descrição
4. Possíveis exemplos de utilizações do aspecto variável

Há vários tipos de *hot spots* em um *framework* que podem ser adaptados para produzir aplicações específicas. Froehlich (1997) e Froehlich (1999) identificaram que um *hot spot* pode ser adaptado em cinco diferentes maneiras, dispostos na Tabela 3.1.

Tabela 3.1 – Tipos de adaptação para um *hot spot*

Tipo	Descrição
1	Habilitar uma característica
2	Desabilitar uma característica
3	Substituir uma característica
4	Aumentar uma característica
5	Adicionar uma característica

Os dois tipos mais comuns são habilitar uma característica e adicionar uma característica. Habilitar uma característica consiste em ativar características que fazem parte do *framework* mas podem não fazer parte da sua implementação padrão.

Ao contrário de habilitar uma característica onde o desenvolvedor está usando serviços existentes, provavelmente de uma forma nova, adicionar uma característica envolve adicionar algo que o *framework* não estava capacitado antes. As adições são freqüentemente feitas através da extensão de classes existentes ou adição de novas classes.

Desabilitar uma característica consiste em desativar uma característica que faz parte da implementação padrão do *framework*.

Substituir uma característica consiste em desabilitar uma característica e colocar uma nova em seu lugar

Aumentar uma característica consiste em estender uma característica sem alterar o fluxo normal de controle. Essa alteração pode interceptar o fluxo de controle existente, executar alguma ação necessária e retornar o controle de volta para o *framework*.

Em relação ao grau de apoio fornecido pelo *framework* ao desenvolvedor existem três tipos:

1) Opcional: o desenvolvedor seleciona um componente pré-definido disponível numa biblioteca.

2) Suporte Padrão: as adaptações estão associadas a instanciamentos pré definidos que dependem de informações fornecidas pelo usuário da aplicação.

3) Ilimitado: as adaptações são realizadas sem apoio do *framework*. O desenvolvedor tem liberdade para alterar o *framework* adicionando características não previstas no *framework* original.

Para completar a descrição do *hot spot* deve ser verificado em cada um deles o(s) tipo(s) de adaptação(es) necessária(s), com base na Tabela 3.1 e o grau de apoio fornecido, incluindo tais informações no catálogo do *hot spot*.

Por exemplo, se a variabilidade envolve preencher um conjunto de parâmetros e não envolve adicionar especificamente novas classes ou serviços ao *framework* então ele é definido quanto ao tipo de adaptação como 1 – habilitar uma característica e quanto ao grau de apoio como suporte padrão. Observe que as adaptações não são exclusivas e podem ser utilizadas na mesma aplicação.

No primeiro ciclo de desenvolvimento são identificadas e analisadas as possíveis variabilidades que estão fixas nos artefatos gerados. Nos ciclos subseqüentes, à medida que novas funcionalidades e/ou variabilidades são introduzidas, novas análises devem ser efetuadas e deve ser determinado quando incluir um ou vários *hot spots* em um ciclo.

Resumo:

Passo 2.2: Analisar e Detectar *Hot Spots*

Entrada:

- Modelo de domínio, requisitos e aplicações existentes (se disponíveis).

Procedimentos:

- Capturar os conceitos de importância que estão dentro do domínio;
- Identificar os *hot spots* verificando que aspectos fixos podem ser flexibilizados;
- Catalogar os *hot spots* e incluir para cada *hot spot* catalogado o tipo de adaptação que será requerida e o grau de apoio fornecido.

Saída:

- Diagrama de Classes;
- Catálogo de *Hot Spots*.

3.2.2.3. Passo 2.3 – Projetar *Framework*

Nessa fase, o modelo estático de objetos criado na análise deve ser detalhado e os *hot spots* incluídos para gerar o diagrama de classe de projeto. Tanto os pontos de flexibilidade descritos no catálogo de *hot spots* são modelados quanto as partes fixas (*frozen*).

Como mencionado no Capítulo 2, para separar o projeto do código as classes deverão ser estruturadas seguindo o modelo de camadas:

- Camada de interface formada por interfaces totalmente abstratas que definem os componentes básicos do *framework*.
- Camada núcleo define o comportamento padrão do *framework* por implementar parcialmente a camada de interface com as classes abstratas.
- Camada padrão contém as classes concretas para circunstâncias comumente recorrentes.

Um diagrama de classes de projeto ilustra as especificações para as classes de *software* e de interfaces de uma aplicação. As informações típicas incluem:

- Classes, associações e atributos.
- Interfaces, com suas operações e constantes.
- Métodos.
- Informação do tipo de atributo.
- Dependências.

Em comparação com um modelo conceitual gerado nas etapas anteriores, um diagrama de classes de projeto mostra definições para entidades de *software* e não conceitos do mundo real (Larman, 1999).

A transição do modelo de objetos criado na etapa anterior para a etapa atual deve ser o mais linear possível, pois muitos dos objetos do projeto já foram identificados e seus nomes devem se mantidos. Como a maioria dos conceitos comuns as aplicações já foram identificados nas etapas anteriores, as abstrações encontradas durante essa fase são abstrações de baixo nível.

A finalidade da fase de projeto é fornecer uma base para uma implementação genérica que servirá a várias aplicações. Ou seja, deve prover reusabilidade.

Segundo Fischer (1987), apesar do paradigma da orientação a objetos ter como fundamento a reutilização de *software* isso não significa que a orientação a objetos seja uma solução para todos os problemas. Os componentes devem ser projetados para

reutilização e há um conjunto de técnicas de projeto que tornam o *software* orientado a objetos mais reutilizável.

A seguir mostramos algumas regras que devem ser observadas durante o projeto do *framework* para torná-lo mais reutilizável:

- Preservar as abstrações identificadas nas etapas anteriores. Futuros refinamentos não devem violar as abstrações conceituais.
- Objetos transferidos diretamente da análise devem manter seus nomes.
- Manter as classes pequenas: uma classe tem o objetivo de representar uma abstração. Se uma classe tem uma quantidade grande de métodos dificultando o seu entendimento, então ela deve representar uma abstração complicada. Isso demonstra que a classe não está bem definida e provavelmente consiste de muitas abstrações diferentes.
- Tentar não estender muito a hierarquia da herança. Hierarquias com vários níveis de abstração devem ser consideradas candidatas à reestruturação.
- Fatorar diferentes definições: se algumas subclasses definem um método de uma maneira e outras o definem de outra maneira então a definição desse método é independente da superclasse, demonstrando que o método não é parte integral das subclasses e poderia ser separado em uma outra classe.
- Tornar as responsabilidades tão genéricas quanto possível: as responsabilidades devem ser colocadas onde elas logicamente pertençam, mas em alguns casos isso pode não estar claro. O projetista deve colocar a responsabilidade na classe que permita maior abstração.
- Criar classes abstratas. Procurar por responsabilidades duplicadas e fatorá-las em superlasses abstratas.

A maneira mais prática de aumentar a reusabilidade é através da utilização de padrões de projeto. O projeto do *framework* pode ser baseado em um ou mais padrões de projeto (Capítulo 2). Para isso os padrões devem ser investigados na literatura, como por exemplo, em *Design Pattern Catalog* (Gamma e outros, 1995). Se eles se aplicam ao problema, este deve ser resolvido de acordo com a proposta do padrão de projeto.

Shalloway e Trott (2002) propoem que padrões podem ser aplicados seguindo os passos abaixo:

1. Pesquise por padrões que estão dentro do seu domínio de problema. Isso leva a um conjunto de padrões a serem analisados.

2. Para o conjunto de padrões a ser analisado faça o seguinte:
    a. Selecione o padrão que fornece um contexto mais amplo para outros padrões.
    b. Aplique esse padrão no mais alto nível do projeto.
    c. Identifique padrões adicionais que podem ser úteis. Adicione-os ao conjunto de padrões a serem analisados.
    d. Repita para os conjuntos de padrões que ainda não foram analisados.
3. Adicione os detalhes necessários do projeto. Expanda as definições de métodos e classes.

Shalloway e Trott (2002) admitem que esse método funciona bem se o projetista puder entender o domínio do problema em termos de padrões, mas nem sempre isso acontece.

Uma solução alternativa é verificar os requisitos do *framework*, bem como seus pontos de flexibilidade e fazer uma descrição do problema encontrado para atender aos itens verificados. Depois, procurar no catálogo de padrões de projeto por qual ou por quais padrões resolvem o problema e então aplicá-los ao projeto.

Os padrões de projeto comumente utilizados em projetos de *frameworks* são:

A) *Strategy*

Define uma família de algoritmos, encapsula cada um e os torna intercambiáveis. *Strategy* faz com que o algoritmo varie independente dos clientes que os utilizam (Figura 3.6). Permite que se utilizem diferentes regras de negócio ou algoritmos dependendo do contexto no qual eles ocorram.

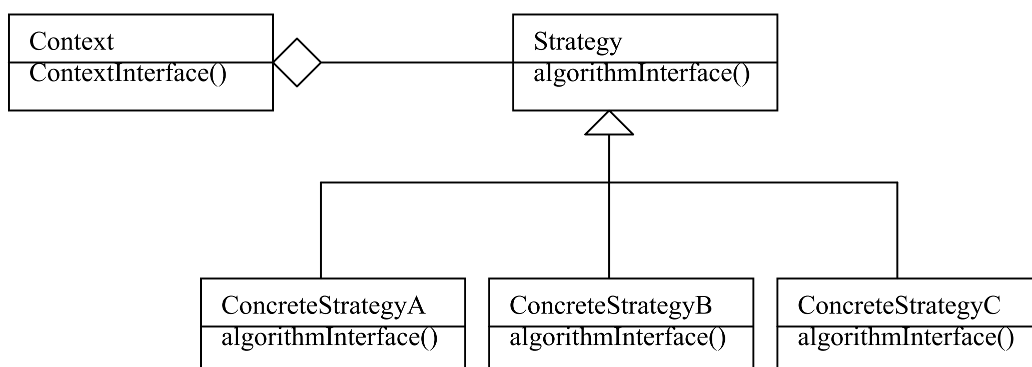

Figura 3.6 – Padrão *strategy*.

B) *Factory Method*

Define uma interface para criar um objeto, mas deixa a subclasse decidir que classe instanciar (Figura 3.7). Uma classe derivada toma a decisão de qual classe instanciar e como instanciá-la

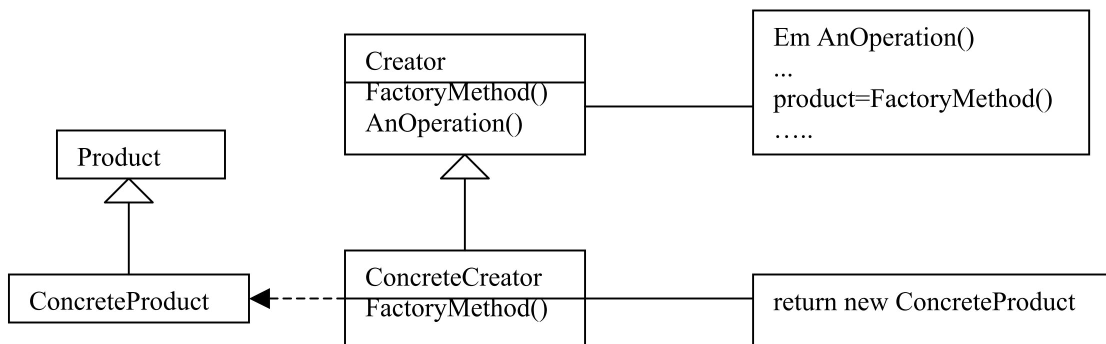

Figura 3.7 – Padrão *factory method*.

C) *Template Method*

Define um método gabarito em uma superclasse que define o esqueleto de um algoritmo, com suas partes variantes e invariantes (Figura 3.8). O *Template Method* invoca outros métodos, alguns dos quais são operações que podem ser redefinidas em uma subclasse. Assim, as subclasses podem redefinir os métodos que variam, de forma a acrescentar o seu próprio comportamento nos pontos de variação. O padrão *Template Method* ilustra o princípio de *Hollywood* (não nos chame, nós o chamaremos).

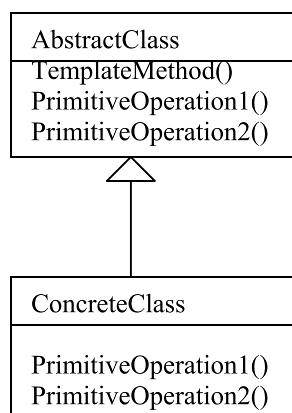

Figura 3.8 – Padrão *template method*.

Resumo

Passo 2.3: Projetar *Framework*

Entrada:

- Modelo estático de objetos da análise, catálogo de *hot spots*, catálogo de padrões de projeto.

Procedimentos:

- Detalhar o modelo estático da análise.
- Aplicar padrões de projeto, incluindo os *hot spots* e as partes fixas.
- Estruturar as classes em camadas.

Saída:

- Diagrama de Classes de Projeto.

3.2.2.4. Passo 2.4 – Implementação e Instanciação exemplo

Com a finalização do diagrama de classes de projeto para o ciclo corrente de desenvolvimento há detalhes suficientes para gerar o código do *framework*.

Um volume significativo de decisões e de trabalho foi realizado nas etapas anteriores, mas a fase de programação não é um passo trivial de geração de código. Segundo Larman (1999) em termos realistas, durante a programação serão feitas muitas mudanças e serão descobertos e resolvidos problemas com os detalhes.

Uma das forças de um processo de desenvolvimento iterativo e incremental está no fato que os resultados de um ciclo anterior realimentam o início do ciclo seguinte. Assim, resultados subseqüentes de coleta de requisitos, análise e de projeto estão sendo continuamente refinados, beneficiando-se do trabalho de implementação anterior (Figura 3.9).

A implementação em uma linguagem de programação orientada a objetos requer que se escreva código fonte para as classes e os métodos. A forma para melhor fazer a transição do projeto para a implementação é através da utilização do diagrama de classes da fase de projeto. Este já contém o nome das classes, superclasses, assinaturas de métodos e os atributos simples de uma classe.

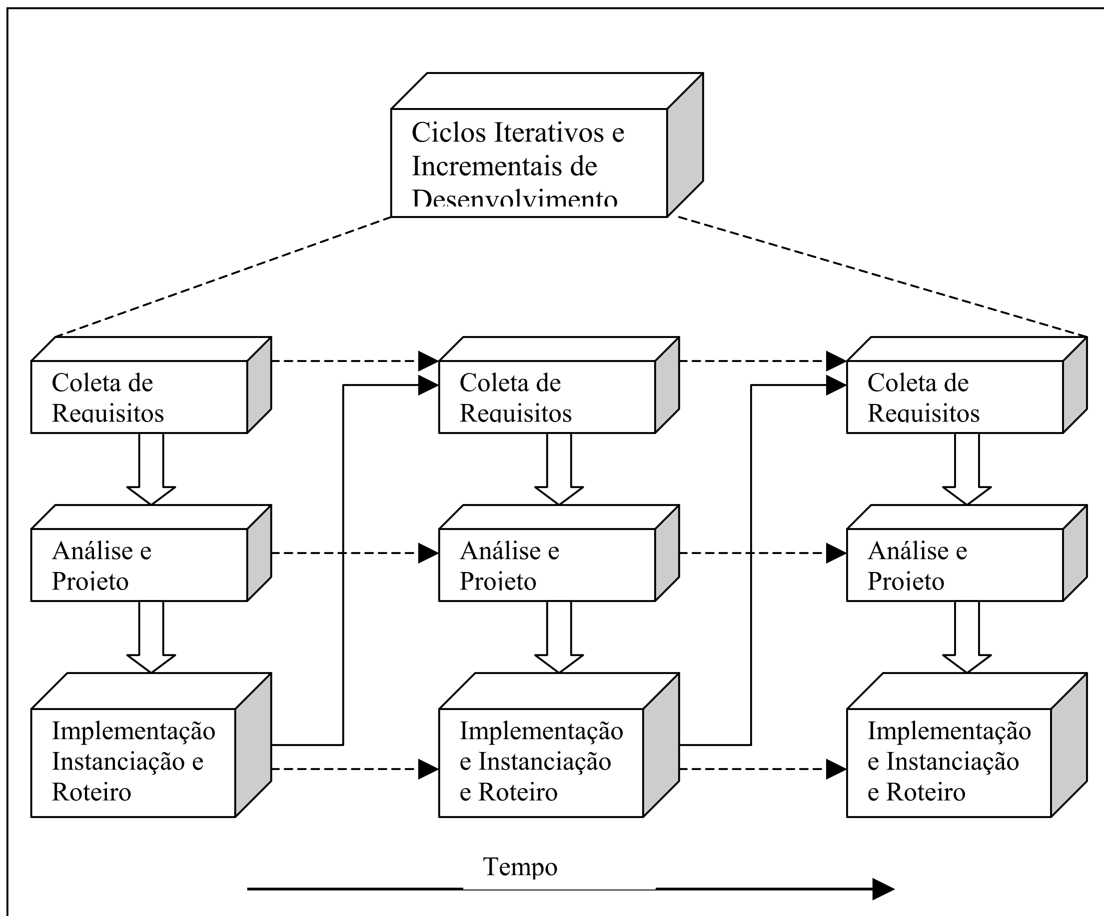

Figura 3.9 – A implementação em um ciclo de desenvolvimento influencia o ciclo posterior.

A instanciação de exemplo é feita gerando uma ou mais aplicações com o *framework* implementado. Através dessa instanciação é possível identificar se o *framework* realmente atende aos requisitos definidos e fornece a flexibilidade desejada.

Resumo:

Passo 2.4: Implementação e Instanciação exemplo

Entrada:

- Diagrama de Classes de Projeto.

Procedimentos:

- Escrever código fonte para as classes e métodos.
- Criar uma aplicação a partir do framework codificado.

Saída:

- Código fonte do framework.
- Aplicação exemplo.

3.2.2.5. Passo 2.5 – Elaborar Roteiro

A elaboração de um roteiro ajudará o desenvolvedor de aplicações a criar aplicações a partir do *framework*.

Assim, o roteiro deverá conter informações suficientes para permitir o entendimento do *framework* e sua forma de trabalho.

Um modelo proposto é o seguinte:

1) Fazer uma descrição do objetivo do *framework*.
2) Listar o que é necessário para a aplicação utilizar o *framework.*
3) Explicar cada elemento básico do *framework* que a aplicação poderá utilizar em termos de classes concretas e/ou abstratas.
4) Mostrar exemplos que demostrem a utilização dos elementos do *framework* do item 3.
5) No caso de *framework* caixa branca (o usuário constrói classes a partir das classes disponíveis) explicar como e onde o desenvolvedor poderá estender as funcionalidades do *framework* e inserir seu próprio código.
6) No caso de *framework* caixa preta (o usuário tem que escolher uma das classes fornecidas) explicar como os elementos do *framework* são acoplados à aplicação.

É fundamental que o roteiro forneça uma visão geral do *framework* e especifique quais classes concretas estão disponíveis para uso, quais podem ser estendidas, o que pode ser implementado, quais classes abstratas são usadas, entre outros aspectos.

Resumo:

Passo 2.5: Elaborar Roteiro

Entrada:

- Documentação do *framework* e aplicação exemplo.

Procedimentos:

- Fazer uma descrição do objetivo do *framework*.
- Listar o que é necessário para a aplicação utilizar o *framework*.
- Explicar cada elemento básico do *framework* que a aplicação poderá utilizar em termos de classes concretas e/ou abstratas.
- Mostrar exemplos que demostrem a utilização dos elementos do *framework*.
- No caso de *framework* "caixa branca" explicar como e onde o desenvolvedor poderá estender as funcionalidades do *framework* e inserir seu próprio código.
- No caso de *framework* "caixa preta" explicar como os elementos do *framework* são acoplados a aplicação.

Saída:

- Roteiro.

3.2.3. Passo 3 - Instanciar Aplicações

Há várias formas de usar um *framework*, algumas podem requerer um profundo conhecimento do *framework*. Todas as formas são diferentes da maneira tradicional de desenvolver *software* usando tecnologia orientada a objetos, já que todas forçam a aplicação a se encaixar no *framework*.

Desenvolver uma aplicação usando *framework* envolve a escolha do *framework* adequado para a aplicação entre a família de *frameworks*. A escolha deve ser baseada nos requisitos levantados para a aplicação a ser gerada, na documentação e no roteiro do *framework*, a fim de verificar em qual deles a aplicação se encaixa melhor.

A maneira mais fácil de usá-lo é utilizar os componentes já existentes, assim, a aplicação reutiliza as interfaces do *framework* e as regras para conectar os componentes. Programadores de aplicação somente devem saber que objetos do tipo A são conectados a objetos do tipo B; eles não têm que conhecer as especificações exatas de A e B. Os *frameworks* utilizados dessa forma são os *frameworks* do tipo caixa preta.

Nem todos os *frameworks* trabalham dessa maneira. Algumas vezes cada novo uso do *framework* vai requerer novas subclasses.

A forma de utilização que exige mais conhecimento é estendê-lo através das classes abstratas que formam o núcleo do *framework*. Embora seja a mais árdua maneira de usar o *framework*, essa é a mais poderosa.

Silva (2000) afirma que no caso geral são três questões chaves que devem ser respondidas:

Questão 1 – Quais classes. As classes concretas de uma aplicação podem ser criadas ou reutilizadas do *framework*. Então, que classes devem ser desenvolvidas pelo usuário e que classes concretas do *framework* podem ser reutilizadas.

Questão 2 – Quais métodos. Um método de uma classe abstrata pode ser classificado como:

- Abstrato: um método que tem apenas a sua assinatura definida na classe abstrata.
- *Templat*e: é definido em termos de outros métodos (métodos adaptáveis).
- Base: é completamento definido.

Ao produzir uma aplicação deve-se observar os seguintes aspectos em relação aos métodos:

- Métodos abstratos precisam ser definidos.
- Métodos *template* fornecem uma estrutura de algoritmo definida, mas permitem flexibilidade através dos métodos adaptáveis chamados. A avaliação da necessidade ou conveniência de produção de métodos deve ser estendida para os métodos adaptáveis.
- Métodos base de acordo com o projeto do *framework* não precisam ser alterados, mas é possível sobrepô-los.

Assim, ao gerar uma classe concreta para uma aplicação (considerando o caso de subclasse de classe abstrata do *framework*), que métodos devem ser definidos e que métodos herdados podem ser reutilizados.

Questão 3 – O que os métodos fazem. Os métodos, cuja estrutura deve ser definida pelo usuário do *framework*, produzem o comportamento específico da aplicação sob desenvolvimento. Assim, definidas as classes e os respectivos métodos a desenvolver, devem ser definidas quais as responsabilidades destes métodos, em que

situações de processamento eles atuam e como eles implementam a cooperação entre diferentes objetos.

A resposta para as questões anteriores deve estar contida no projeto de um *framework*, pois, a definição das classes e métodos que devem ser desenvolvidos é resultado das decisões de projeto, tomadas no desenvolvimento do *framework*.

Utilizar um *framework* sem a devida documentação dificultará o conhecimento do mesmo por parte do desenvolvedor de aplicações. Como conseqüência ele poderá perder um tempo razoável desenvolvendo o que poderia ser reutilizado e não aproveitará a característica chave do *framework*, a reutilização, que leva a economia de tempo e esforços.

Passo 3: Instanciar Aplicação

Entrada:

- Requisitos da aplicação, documentação do *framework* e roteiro.

Procedimentos:

- Escolher o *framework* adequado;
- Verificar se o *framework* é do tipo caixa branca ou caixa preta;
- Checar o roteiro para verificar:
  - Quais classes podem ser criadas ou reutilizadas;
  - Quais métodos estão disponíveis.
- Desenvolver aplicação.

Saída:

- Aplicação.

4. ESTUDO DE CASO

Este capítulo descreve um estudo de caso da nossa abordagem de desenvolvimento.

Nosso objetivo é executar um exemplo do desenvolvimento de uma família de três *frameworks* utilizando o processo de desenvolvimento proposto.

O domínio escolhido é o de editores de figuras que não é um domínio grande e cujos conceitos e funções são conhecidos para a maioria dos usuários.

Será utilizado como modelo base o *JHotDraw* versão 5.1, que é um *framework* gráfico implementado em Java para editores de aplicações baseados em objetos. *JHotDraw* foi escrito por Erich Gamma e pode ser encontrado em (Gamma, 2001).

4.1. ABORDAGEM UTILIZADA

Para este estudo de caso partimos de um *framework* completo e retiramos algumas de suas funcionalidades com o objetivo de criar uma estrutura para a simulação do desenvolvimento através de ciclos.

O *JHotDraw* foi reduzido a um conjunto mínimo inicial. Foram efetuados três ciclos de desenvolvimento seguindo os passos mostrados no Capítulo 3 (Figura 3.1) e gerados três *frameworks* (Figura 4.1). Cada *framework* da família pode ser utilizado para produzir um conjunto de aplicações que tenham os requisitos cobertos por ele.

Para comprovar a utilidade dos *frameworks* gerados, uma aplicação foi desenvolvida para cada *framework*. Estas aplicações são mostradas no Apêndice 1 da dissertação. No Apêndice 2, encontram-se alguns dos padrões de projeto que foram utilizados.

Nas seções a seguir mostramos os passos do desenvolvimento dos três *frameworks*. A notação utilizada para os diagramas e demais modelos necessários é a UML - *Unified Model Language* (Rumbaugh e outros, 1998), mostrada no Apêndice 3.

Os *frameworks* desenvolvidos são subconjuntos do *JhotDraw*. Entretanto é possível o desenvolvimento de aplicações a partir do *framework* produzido em cada ciclo.

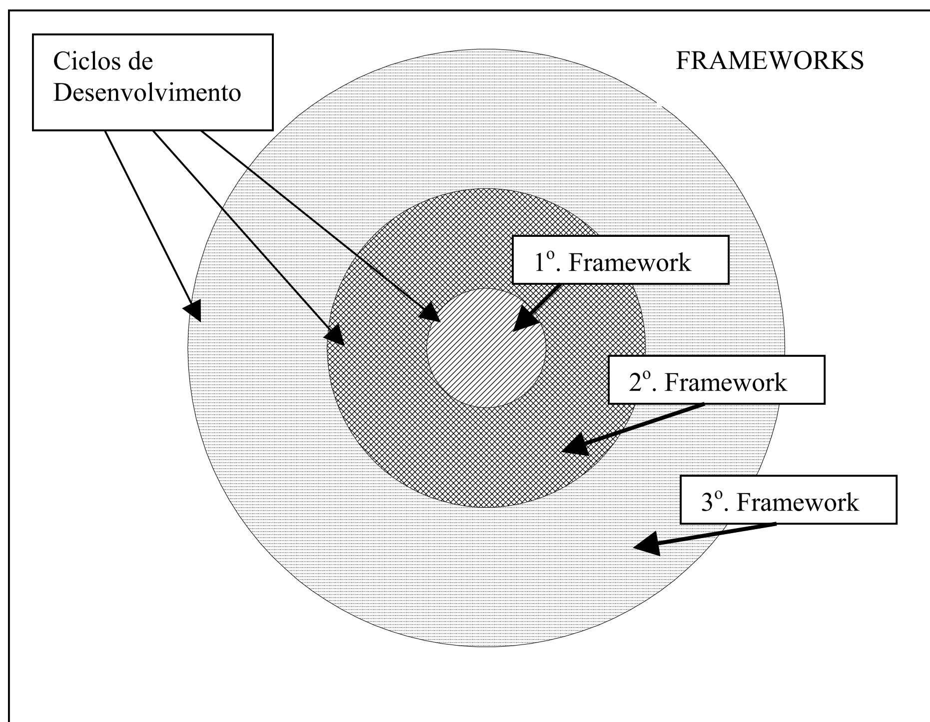

Figura 4.1 – *Framework* gerado a cada ciclo

4.2. DESENVOLVIMENTO

Seguindo os passos do processo proposto no Capítulo 3 (Figura 3.1) iniciamos o desenvolvimento do *framework*.

Passo 1 : Análise de Domínio

A) Definição do Escopo :

Objetivo:

- Prover uma estrutura reutilizável para criação de aplicações para edição de figuras.

Limites:

- Não serão consideradas figuras em três dimensões. Fica a cargo do desenvolvedor estender ou desenvolver classes para essa finalidade.

Benefícios:

- Economia de tempo para geração de aplicações de edição de figuras.
- Permite a extensão de classes para adicionar novos tipos de ferramentas e figuras.

B) Modelo estático preliminar :

O *JHotDraw* é baseado no MVC (*Model View Controller*), que separa a lógica da aplicação da interface do usuário. A visualização (*view*) é responsável por mostrar a informação na interface do usuário; o controlador (*controller*) manipula a interação do usuário e a mapeia para a funcionalidade da aplicação. O modelo (*model*) consiste da lógica e dos dados da aplicação. *Drawing* representa o modelo, o controlador é oferecido através das ferramentas (*tools*) e a *view* é notificada das mudanças nos dados.

Na Figura 4.2 é mostrado um diagrama de classes preliminar (modelo estático) onde o editor é composto pelo *DrawingEditor* que contém as ferramentas (*Tools*), a visualização (*DrawingView*) e a área de desenho, essa é composta por figuras (*Figure*) que por sua vez contém manipuladores (*Handle*).

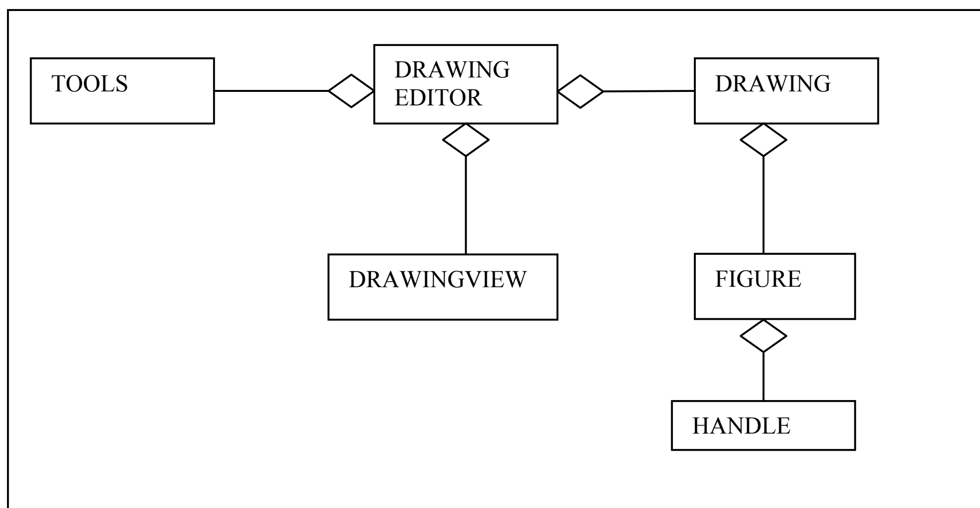

Figura 4.2 – Modelo estático

Passo 2 – Construir: Primeiro Ciclo de Desenvolvimento

Desenvolvimento do primeiro *framework.*

Passo 2.1: Coleta de Requisitos:

O editor consiste de uma área para o desenho de figura, um conjunto de figuras, ferramentas de seleção e criação de figuras básicas.

Requisitos Funcionais:

- Desenhar linhas.
- Desenhar formas básicas.
- Agrupar Figuras.
- Remover linhas e formas básicas.

- Alterar tamanho da figura.
- Mover figura.

Requisitos não funcionais:

- Ser extensível possibilitando adicionar novos tipos de ferramentas e figuras.
- Ser fácil de compreender.
- Conter documentação.

Descrição dos Casos de Uso:

01 – Caso de Uso Abstrato Desenhar Figuras

Descrição: uma figura, que pode ser linha ou forma básica, é selecionada da paleta de ferramentas e adicionada a área do editor.

02 – Caso de Uso Abstrato Remover Figura

Descrição: uma figura é selecionada da área do editor e então através da tecla *delete* ou *back space* a figura é removida.

03 – Caso de Uso Abstrato Agrupar Figura

Descrição: duas ou mais figuras são agrupadas como uma única figura e tratadas pelo editor como uma única figura.

04 – Caso de Uso Abstrato Alterar Tamanho

Descrição: uma figura da área do editor é selecionada e ao arrastar alguns dos seus pontos de manipulação tem seu tamanho aumentado ou diminuído.

05 – Caso de Uso Abstrato Mover Figura

Descrição: uma figura da área do editor é selecionada e movida para outra localização dentro da área.

Passo 2.2 – Análise e Detecção de *Hot Spots*

Análise:

No diagrama de classes da análise (Figura 4.3) são adicionados escutadores de eventos (*DrawingChangeListener* e *FigureChangeListener*) e um *grid* (*PointConstrainer*)

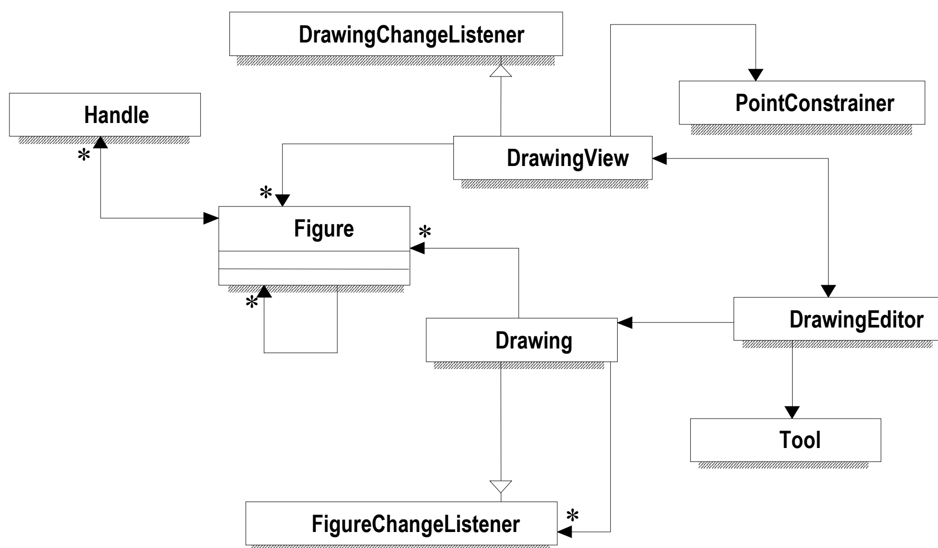

Figura 4.3 – Diagrama de classes do 1o. ciclo

Detecção de *Hot Spots*:

Número:1

Nome: variabilidade de figuras.

Descrição1: deve ser permitida a utilização de figuras existentes.

Exemplos de utilização: elipse, retângulo, polígono são diferentes alternativas que podem ser fornecidas como elementos de figuras.

Tipo de adaptação requerida: 1 – Habilitar uma característica.

Grau de apoio: suporte padrão.

Descrição 2: deve ser permitido a criação de diferentes figuras.

Exemplos de utilização: criar uma outra figura como extensão de um elipse.

Tipo de adaptação requerida: 5 – Adicionar uma característica.

Grau de apoio: ilimitado.

Número:2

Nome: variabilidade de ferramentas.

Descrição 1: deve ser permitido a utilização de ferramentas.

Exemplos de utilização: ferramenta de seleção, ferramenta de desenho de figuras.

Tipo de adaptação requerida: 1 – Habilitar uma característica.

Grau de apoio: suporte padrão.

Descrição 2: deve ser permitida a criação de diferentes ferramentas.

Exemplos de utilização: ferramenta para enviar a figura para frente ou para trás.

Tipo de adaptação requerida: 5 – adicionar uma característica.

Grau de apoio: ilimitado.

Número: 3

Nome: variabilidade de figuras agrupadas.

Descrição: deve ser permitido criar figuras agrupadas.

Exemplos de utilização: dois retângulos como uma única figura.

Tipo de adaptação requerida: 5 – Adicionar uma característica.

Grau de apoio: suporte padrão.

Passo 2.3 -Projeto

Seguindo o modelo três camadas (Sessão 2.2.5) o *framework* é assim dividido (Figura 4.4):

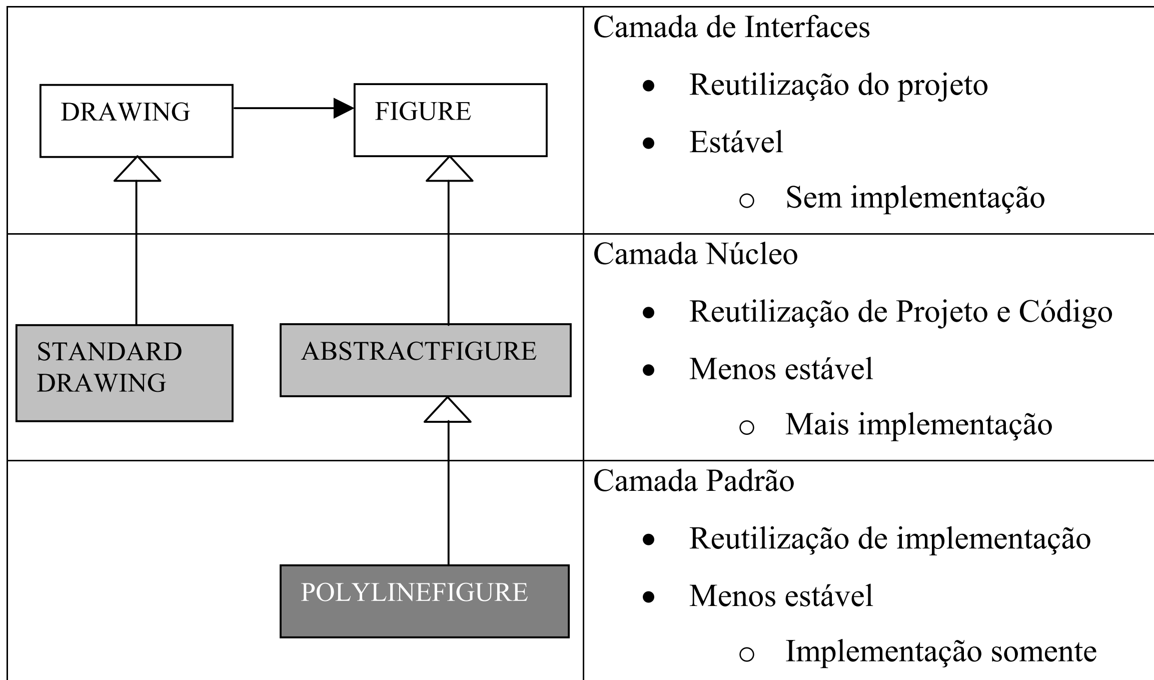

Figura 4.4 – Estruturação do *framework*

Definição de Padrões de Projeto:

Problema: um editor é composto de várias objetos (*tools*, *figures*, *drawingview*), deseja-se um coordenador para os objetos.

Padrão: *Mediator.*

Problema: permitir composição de figuras.

Padrão: *Composite.*

Problema: o *DrawingView* recebe os eventos a partir da interface do usuário, a resposta depende da ferramenta ativa no momento. Diferentes tipos de editores podem usar diferentes tipos de ferramentas.

Padrão: *State.*

Problema: permitir a criação de Ferramentas.

Padrão: *Factory Method* e *Prototype.*

Problema: quando uma ferramenta manipula uma figura deve ser atualizado o *DrawingView.*

Padrão: *Observater*, *Template Method.*

Problema: cada figura tem uma forma de ser manipulada. Deseja-se que a mesma ferramenta possa manipular figuras que tem interfaces distintas.

Padrão: *Adapter.*

Problema: um manipulador é responsável por saber sua posição e manipular figura. Manipuladores com mesma ação, mas com posição distinta, requerem classes diferentes.

Padrão: *Strategy.*

Diagrama de Classes do Projeto

A seguir é mostrado o diagrama de classes do projeto (Figura 4.5) onde as classes, subclasses e relacionamentos são detalhados.

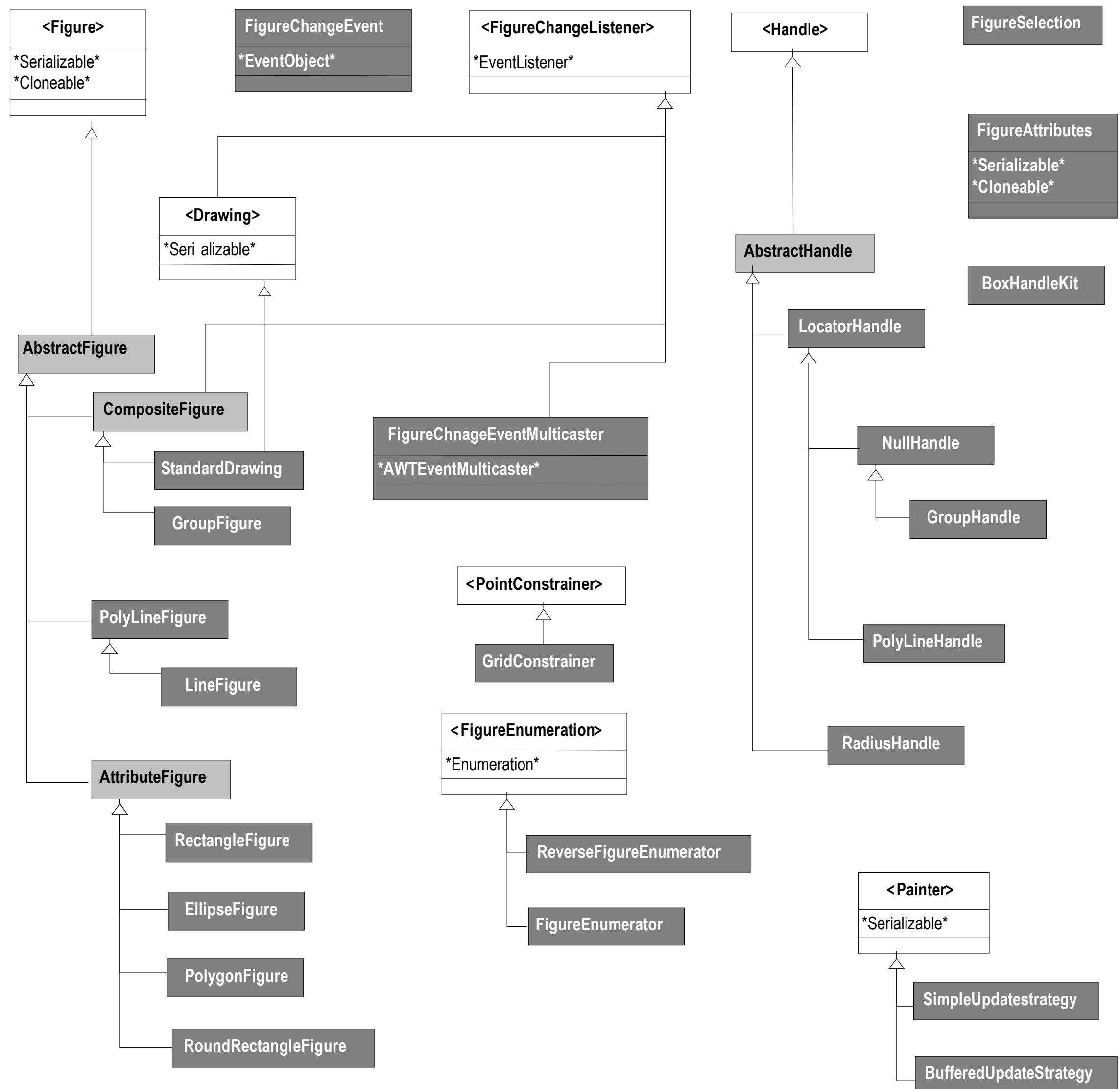

Figura 4.5 – Diagrama de classes do projeto – 1o. ciclo

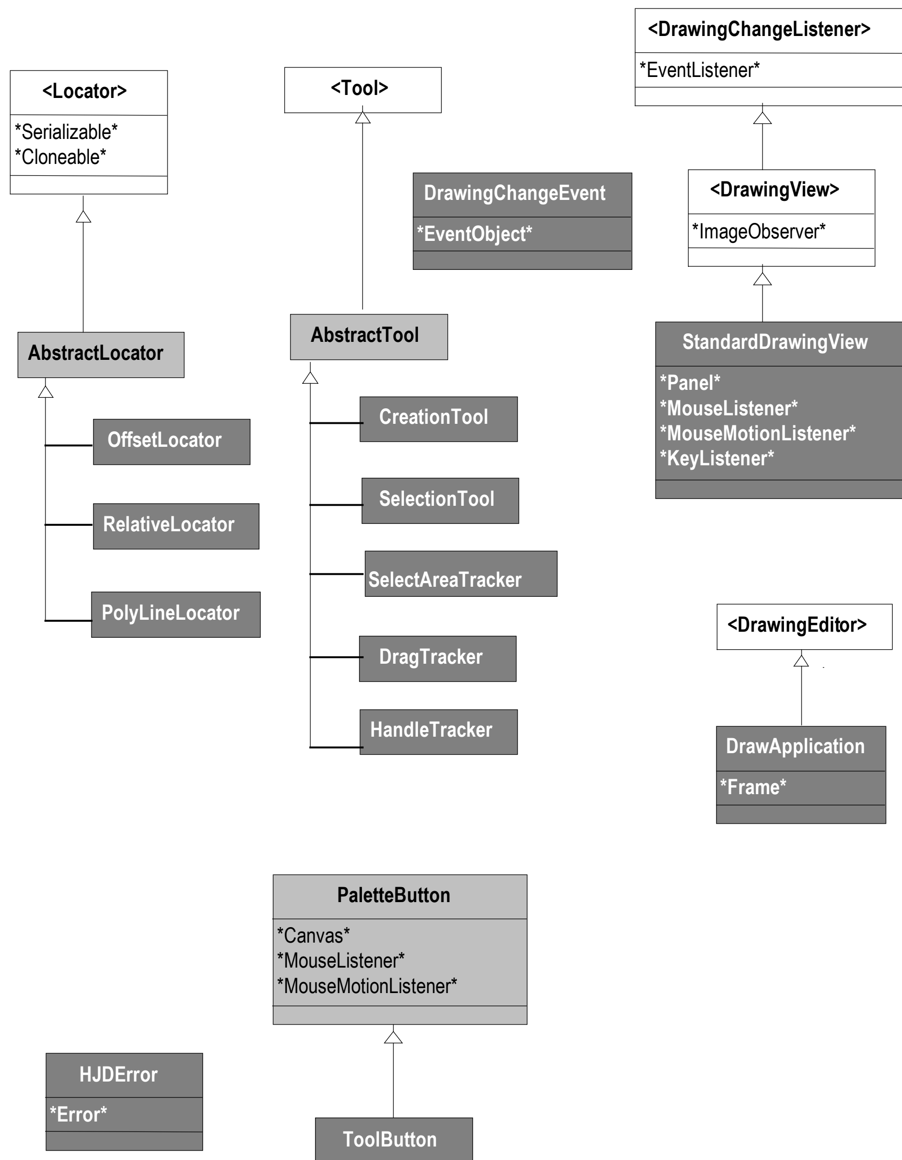

Figura 4.5 (Continuação)

Passo 2. 4 Implementação e Instanciação Exemplo

Implementação:

Para implementação foi utilizada a linguagem orientada a objetos Java.

As classes e métodos são documentados com JavaDocs (Ricarte, 2000). Como exemplo temos o seguinte trecho de código com comentários em JavaDocs:

```
/*
 * @(#)Figure.java 5.1
*/

package CH.ifa.draw.framework;

import CH.ifa.draw.util.*;
import java.awt.*;
import java.util.*;
import java.io.Serializable;

/**
 * The interface of a graphical figure. A figure knows
 * its display box and can draw itself. A figure can be
 * composed of several figures. To interact and manipulate
 * with a figure it can provide Handles.<p>
 * A figure has a set of handles to manipulate its shape or attributes.
  * Figures can have an open ended set of attributes.
 * An attribute is identified by a string.<p>
 * Default implementations for the Figure interface are provided
 * by AbstractFigure.
 *
 * @see Handle
 * @see AbstractFigure
 */

  public interface Figure
          extends Storable, Cloneable, Serializable {

   /**
    * Moves the Figure to a new location.
    * @param x the x delta
    * @param y the y delta
    */
   public void moveBy(int dx, int dy);

   /**
    * Changes the display box of a figure. This method is
    * always implemented in figure subclasses.
    * It only changes
    * the displaybox and does not announce any changes. It
    * is usually not called by the client. Clients typically call
    * displayBox to change the display box.
    * @param origin the new origin
    * @param corner the new corner
    * @see #displayBox
    */
   public void basicDisplayBox(Point origin, Point corner);
```

Instanciação Exemplo

A instanciação exemplo é gerada do seguinte modo:

- Cria-se uma extensão da classe *DrawApplication*, que define uma apresentação padrão para editores:

```
public  class App  extends DrawApplication
```

- As ferramentas são criadas e acrescentadas na paleta utilizando o método *createTools*.
- Estão implementadas as classes das figuras: *Ellipse*, *Rectangle*, *RoundRectangle*, *Line* e *Polygon* que são acrescentadas às ferramentas.

```
protected void createTools(Panel palette) {
        tool = new CreationTool(view( ), new EllipseFigure( ));
        palette.add(createToolButton(IMAGES+"ELLIPSE","Ellipse Tool", tool));
        …
        }
```

- A aplicação *stand-alone* é criada através do código:

```
public static void main(String[] args) {
        DrawApplication window = new App( );
                window.open();
}
```

O resultado é a criação de uma aplicação que é inicializada com as ferramentas e figuras disponíveis (Figura 4.6).

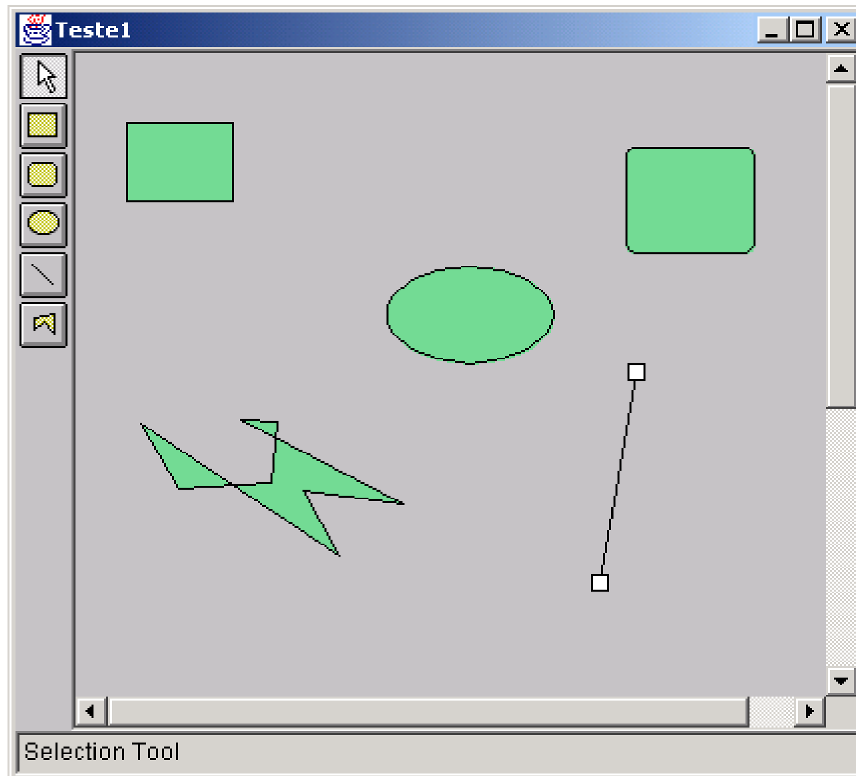

Figura 4.6 – Editor de exemplo gerado com o *framework* utilizando as figuras e ferramentas padrões.

Passo 2.5 – Elaboração de Roteiro

Objetivo: construir aplicações do tipo editor reutilizando a estrutura do *framework*.

Para construir uma aplicação utilizando o *JHotDraw* defina as figuras que serão necessárias, o comportamento de cada figura e as ferramentas para manipulá-las.

Para Criar Figuras

Há várias figuras já definidas que podem ser acrescentadas em um drawing, entretanto pode haver necessidade de criar novas figuras para cada aplicação.

Há quatro maneiras para criação de figuras:

1) Usar classes de figuras existentes: estão disponíveis na implementação padrão as classes *EllipseFigure*, *RectangleFigure*, *LineFigure* e *PolygonFigure*.

2) Criar subclasses das classes existentes: é possível estender as classes com implementações padrões e criar as novas figuras que a aplicação necessitará. Por exemplo:

```
public class myPlanetFigure extends EllipseFigure
```

3) Composição: pode ser usado em situações onde as novas funcionalidades podem ser criadas por arranjar várias figuras pré-existentes juntas. Para isso, o *JHotDraw* fornece a subclasse de *Figure*, *CompositeFigure* e sua subclasse *GroupFigure*.

   Ex: figura formada por dois elipses e um retângulo:

```
public class myFigure extends GroupFigure {
public myFigure() {
super();
RectangleFigure rect1 = new RectangleFigure(new Point(3,0), new
Point(8,10));
EllipseFigure ell1 = new EllipseFigure(new Point(0,10), new Point(10,20));
EllipseFigure ell2 = new EllipseFigure(new Point(0,20), new Point(10,30));
…
super.add(rect1);
super.add(ell1);
super.add(ell2);  }
```

4) Especializar figura: criar subclasses de *AbstractFigure*, o que vai requerer a definição de quatro métodos *basicMoveBy*, *MoveBy*, *basicdisplayBox* e *handles*. Em casos extremos uma figura especializada pode ser criada diretamente implementando a interface Figure.

Para Criar Ferramentas

As ferramentas representam o modo de interação entre a interface do usuário e a área de desenho. Selecionar uma ferramenta da paleta faz com que o usuário manipule figuras, crie novas figuras ou execute operações sobre uma figura.

O método *createTools* da classe *DrawApplication* cria por padrão a ferramenta de seleção, mas pode ser sobreposto para retornar as ferramentas requeridas pela aplicação.

```
protected void createTools(Panel palette) {
Tool tool = createSelectionTool( );
…
}
protected Tool createSelectionTool( ) {
    return new SelectionTool(view( ));
}
```

Está disponível a seguinte categoria de ferramenta:

- Ferramentas de Criação: o método *CreationTool* pode ser especializado para ser uma ferrramenta que cria figuras tais como retângulo ou elipse. Qualquer figura pré-definida pode ser criada dessa forma.

```
Tool tool = new CreationTool(view(), new RectangleFigure());
palette.add(createToolButton(IMAGES+"RECT", "Rectangle Tool", tool));
```

Para Criar Manipuladores

Manipuladores variam em comportamento. Por exemplo: podem mudar o tamanho das figuras. Podem ser anexados a qualquer parte da figura.

A interface *Handle* define os comportamentos importantes que um manipulador deve implementar. Por padrão está implementado o manipulador representado por um retângulo branco que permite que a figura seja redimensionada.

O desenvolvedor pode criar subclasse da classe *AbstractHandle* (que implementa a interface *Handle*) para fornecer o comportamento desejado na aplicação.

Passo 3 - Instanciando Aplicações

O primeiro *framework* está concluído e pode ser disponibilizado para a geração de aplicações com as características que estão presentes nele. Para comprovar a utilidade deste *framework* instanciamos a aplicação *Planet Application* (Apêndice 1).

Note que há aplicações que vão requerer características que não estão presentes no *framework* que nós desenvolvemos no primeiro ciclo.

Diante desse cenário, partiremos para o próximo ciclo de desenvolvimento a fim de incorporar novos requisitos.

Passo 2 – Construir : Segundo Ciclo de Desenvolvimento

Desenvolvimento do segundo *framework.*

Passo 2.1 - Coletar Requisitos

- Possuir Menus.
- Permitir inserir texto com formatação de fonte estilo e tamanho.
- Permitir executar comandos de edição (recortar, colar, copiar).

Casos de Uso Abstratos

01 – Caso de Uso abstrato Criar Menus

Descrição: menus podem ser criados e adicionados na aplicação conforme seja necessário.

02 – Caso de Uso Abstrato Inserir Texto

Descrição: texto pode ser inserido na área de desenho e ter formatação de fonte, estilo e tamanho. Pode ser agrupado com uma figura.

03 – Caso de Uso Abstrato Executar Comandos

Descrição: comandos são aplicados às figuras e texto, permitindo recortar, colar, duplicar, enviar para frente e para trás.

Passo 2.2 - Análise e Detecção de HotSpots

Análise:

No diagrama de classes da análise (Figura 4.4) são adicionadas, em relação ao diagrama de classes do primeiro ciclo de desenvolvimento (Figura 4.3), as classes *Text*, *Menu* e *Command*.

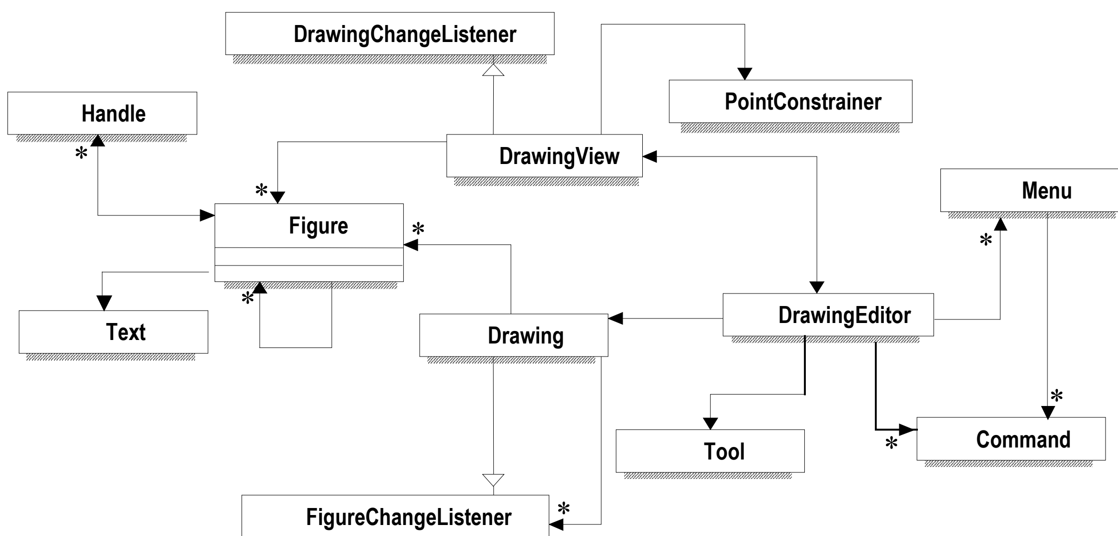

Figura 4.7 – Diagrama de classes do 2º. ciclo

Detecção de *Hot Spots*

Número:1

Nome : variabilidade de menus.

Descrição1: deve ser permitido a adição de menus.

Exemplos de utilização: menu de atributo, menu de arquivo.

Tipo de adaptação requerida: 1 – Habilitar uma característica.

Grau de apoio: opcional.

Descrição2: deve ser permitido a criação de menus.

Exemplos de utilização: menu de animação.

Tipo de adaptação requerida: 5 – Adicionar uma característica.

Grau de apoio: suporte padrão.

Número: 2

Nome: variabilidade de comandos.

Descrição1: deve ser permitido utilizar comandos.

Exemplos de utilização: apagar, copiar, colar, recortar, duplicar.

Tipo de adaptação requerida:1 – Habilitar uma característica.

Grau de apoio: opcional.

Descrição2: deve ser permitido acrescentar comandos.

Exemplos de utilização: comando para desfazer alterações (*Undo*).

Tipo de adaptação requerida: 5 – Adicionar uma característica.

Grau de apoio: ilimitado.

Número: 3

Nome: inserção de figura de texto.

Descrição: deve ser permitida a utilização de texto.

Exemplos de utilização: desenhar uma caixa de texto no editor.

Tipo de adaptação requerida: 1 – Habilitar uma característica.

Grau de apoio: opcional.

Passo 2.3 - Projeto:

Definição de Padrões de Projeto:

Problema: a aplicação pode criar ou adicionar menus pré configurados.

Padrão: *Factory method.*

Problema: a aplicação pode inserir texto na área de desenho ou dentro de uma figura.

Padrão: *Factory method.*

Problema: muitas operações (copiar, apagar, recortar, entre outros) podem ser invocadas de diferentes maneiras. Freqüentemente as operações são as mesmas nos vários editores.

Padrão: *Command.*

Diagrama de Classes do Projeto

A seguir é mostrado o diagrama de classes do projeto (Figura 4.8) onde as classes, subclasses e relacionamentos são detalhados. Em relação ao diagrama de classes do projeto do primeiro ciclo, foram adicionadas a interface *TextHolder*, as classes *TextFigure, NumberTextFigure, Command, FigureTransferCommand, BringTo FrontCommand, AlignCommand, ChangeAttributeCommand, SendToBackCommand, ToggleGridCommand, InsertImageCommand, GroupCommand, CopyCommand, CutCommand, DeleteCommand, DuplicateCommand* e *PasteCommand*.

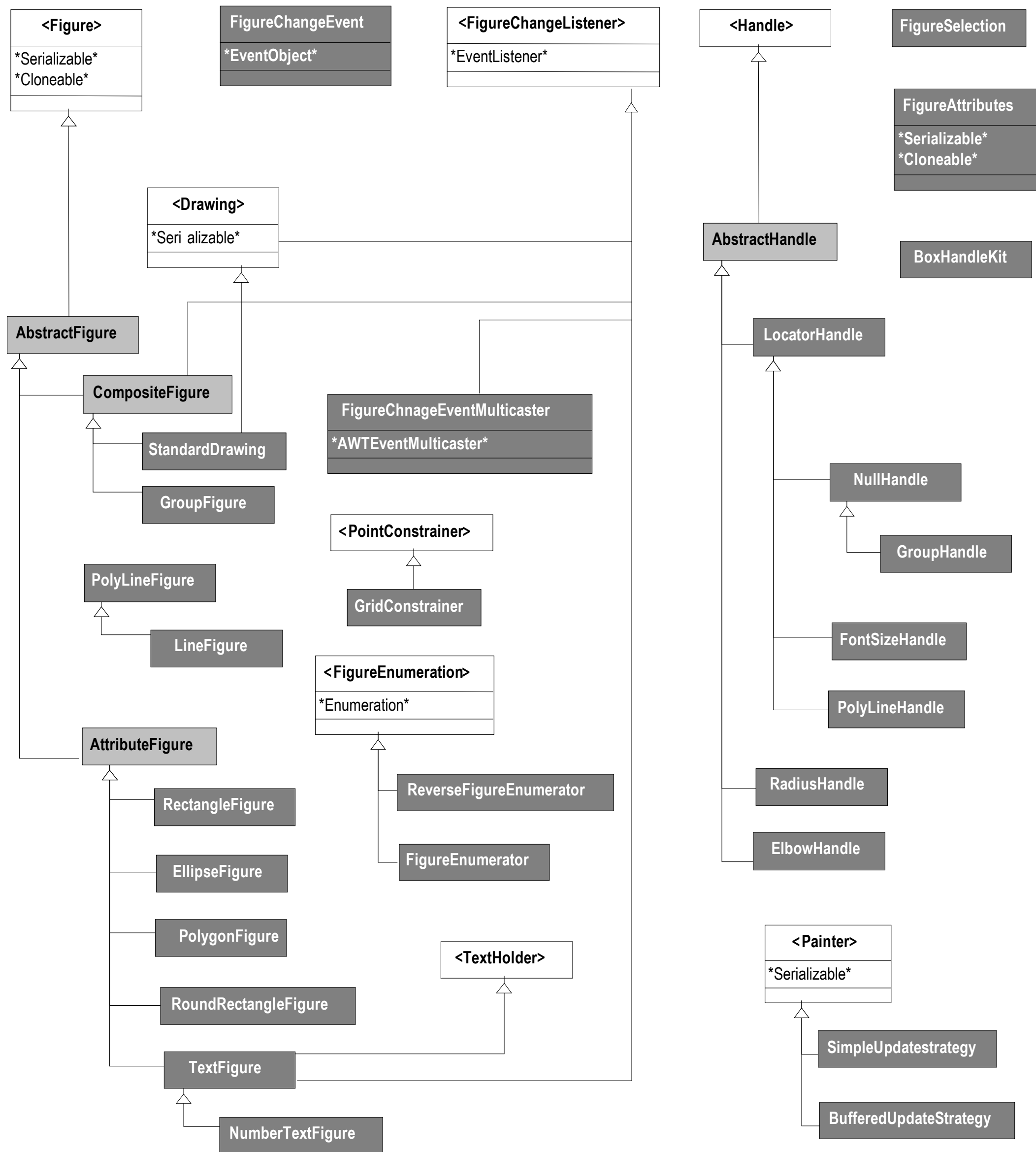

Figura 4.8 – Diagrama de classes do projeto – 2o. ciclo

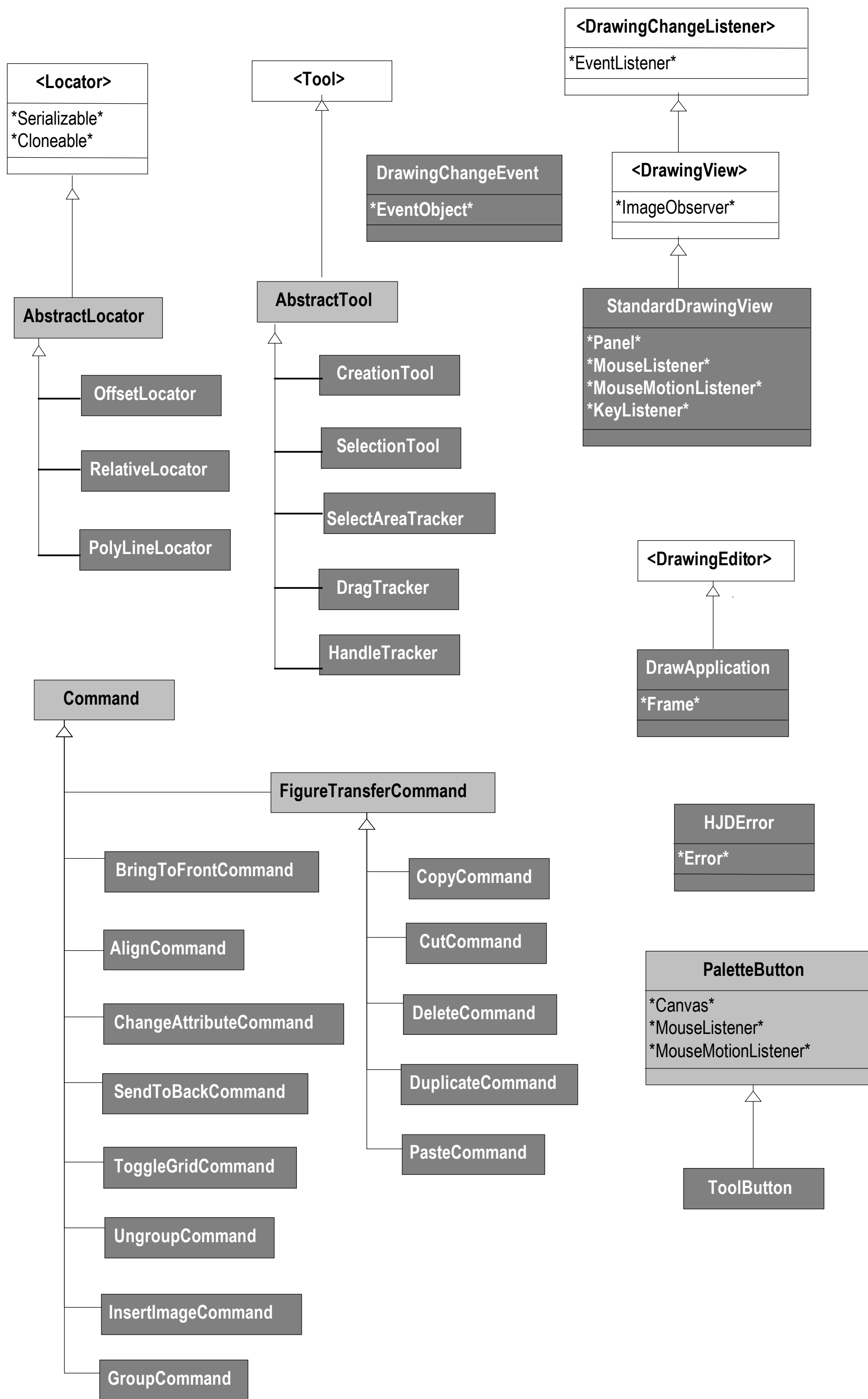

Figura 4.8 (Continuação)

Passo 2.4 - Implementação e Instanciação Exemplo

Implementação

Foram adicionadas as classes que se encontram no projeto do *framework*.

Instanciação Exemplo

A instanciação exemplo é gerada do seguinte modo:

- Cria-se uma extensão da classe *DrawApplication*, que define uma apresentação padrão para editores:

```
public  class App  extends DrawApplication
```

- As ferramentas são criadas e acrescentadas na paleta utilizando o método *createTools*.
  - Estão implementadas as classes das figuras: *Ellipse*, *Rectangle*, *RoundRectangle*, *Line* e *Polygon*.

```
protected void createTools(Panel palette) {
        tool = new CreationTool(view( ), new EllipseFigure( ));
        palette.add(createToolButton(IMAGES+"ELLIPSE","Ellipse Tool", tool));
        …
}
```

  - A ferramenta de texto é adicionada ao editor:

```
Tool tool = new TextTool(view(), new TextFigure());
        palette.add(createToolButton(IMAGES+"TEXT", "Text Tool", tool));
```

  - Os menus são criados e adicionados através dos métodos *createMenus*( ):

```
protected void createMenus(MenuBar mb) {
         mb.add(createFileMenu());
         mb.add(createEditMenu());
        …}
protected Menu createFileMenu() {
                Menu menu = new Menu("File");
        …}
```

  - Os comandos são adicionados com *CommandMenu:*

```
protected Menu createEditMenu() {
        CommandMenu menu = new CommandMenu("Edit");
        menu.add(new CutCommand("Cut", fView), new MenuShortcut('x'));
```

- A aplicação *stand-alone* é criada através do código:

```
public static void main(String[] args) {
        DrawApplication window = new App( );
                window.open();
}
```

O resultado é a criação de uma aplicação que é inicializada com as ferramentas e a configuração de menu padrão (Figura 4.9).

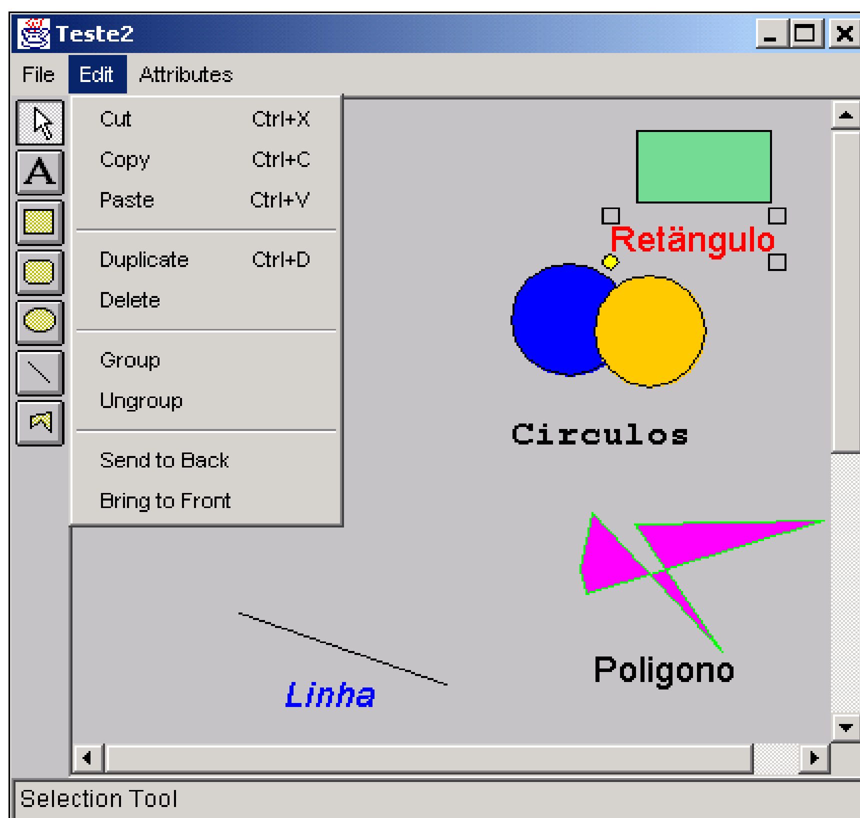

Figura 4.9 - Editor de exemplo com menu, texto e comandos de edição.

Passo 2.5 – Elaboração de Roteiro

Objetivo: construir aplicações do tipo editor reutilizando a estrutura do *framework*.

Para construir uma aplicação utilizando o *JHotDraw*, primeiro defina as figuras que serão necessárias, o comportamento de cada figura, as ferramentas para manipulá-las, os menus necessários e os comandos que serão utilizados.

Para Inserir Texto

Para criar ou editar figuras de texto há a classe *TextTool* que adiciona uma ferramenta para texto no editor. *TextTool* estende a ferramenta de criação *CreationTool*. e é adicionada ao editor da seguinte maneira:

```
Tool tool = new TextTool(view(), new TextFigure());
```

O desenvolvedor poderá criar suas próprias figuras com texto de duas formas:

1) Cria uma subclasse de *AttributeFigure* e como a figura formada tem como conteúdo algum texto editável deverá implementar a interface *TextHolder*

```
public  class myTextFigure  extends AttributeFigure
                                implements TextHolder
```

2) Cria uma subclasse da classe *TextFigure* que já implementa a interface *TextHolder*. Por exemplo:

```
public class NodeFigure extends TextFigure
```

Para Trocar atributos dos elementos

Mudar os atributos de uma figura implica que ela foi selecionada e a capacidade de executar determinada função existe.

Há três maneiras de editar os atributos de uma figura:

1) *Handles*: todas as figuras têm um método *handles* que retorna um vetor de manipuladores para cada figura. Por definição, *abstractFigure* retorna um manipulador para regular o tamanho de cada canto da figura.
2) *Tools*: embora mais frequentemente usado para criar figuras, *tools* pode ser usado para adicionar novos atributos a figuras já existentes.
3) Menus: são similares a *tools* e freqüentemente refletem a paleta de *tools*. São mais adequados para eventos orientados a seleção, tais como escolher uma cor para preenchimento ou mudar a fonte.

Para Criar novos Menus

Menus são definidos na classe *DrawApplication*. Por definição, há a seguinte estrutura de menus: *files, edit, atributes* (*color* e *fontsize*). Estes menus podem ser removidos ou adicionados conforme a necessidade da aplicação. A classe *DrawApplication* define métodos separados de criação para cada um dos menus acima.

```
protected void createMenus(MenuBar mb) {
        mb.add(createFileMenu());
        mb.add(createEditMenu());
        mb.add(createAttributesMenu());
}
```

A criação da barra de menu ocorre no método *createMenu*. O desenvolvedor pode especializá-la omitindo menus não necessários e adicionando chamadas de métodos para seus próprios menus, em sua aplicação, sobrescrevendo o método *createMenus*.

Para Criar Comandos

Comandos executam ações específicas quando selecionados. O *JHotDraw* fornece ao desenvolvedor as seguintes classes de comandos pré-definidos para utilizar em sua aplicação conforme seja necessário : *CutCommand, CopyCommand, PasteCommand, DuplicateCommand, DeleteCommand, GroupCommand, UngroupCommand, SendToBackCommand* e *BringToFrontCommand*,

O desenvolvedor também poderá criar seus próprios comandos como extensão das classes abstratas *Command* ou *FigureTransferCommand* (para classes que transferem figuras entre a área de desenho e o *clipboard*) e utilizá-los através das classes *CommandMenu, CommandChoice* ou *CommandButton* os quais permitem a subclasses definir que ação tomar quando o usuário seleciona uma opção.

Ex1. Utilização de *CommandMenu* (Figura 4.10):

```
protected Menu createEditMenu() {
CommandMenu menu = new CommandMenu("Edit");
menu.add(new CutCommand("Cut", fView), new MenuShortcut('x'));
menu.add(new CopyCommand("Copy", fView), new MenuShortcut('c'));
menu.add(new SendToBackCommand("Send to Back", fView));
menu.add(new BringToFrontCommand("Bring to Front", fView));
return menu;
```

Ex2: Utilização de *CommandChoice* (Figura 4.11):

```
protected void createAttributeChoices(Panel panel) {
        panel.add(new Label("Fill"));
        fFillColor = createColorChoice("FillColor");
        panel.add(fFillColor);
        panel.add(new Label("Pen"));
        panel.add(new Label("Arrow"));
        CommandChoice choice = new CommandChoice();
..
```

Ex3: Utilização de *CommandButton* (Figura 4.12):

```
Button button;
button = new CommandButton(new DeleteCommand("Delete", fView));
panel.add(button);
button = new CommandButton(new DuplicateCommand("Duplicate", fView));
panel.add(button);
button = new CommandButton(new GroupCommand("Group", fView));
panel.add(button);
button = new CommandButton(new UngroupCommand("Ungroup", fView));
panel.add(button);
```

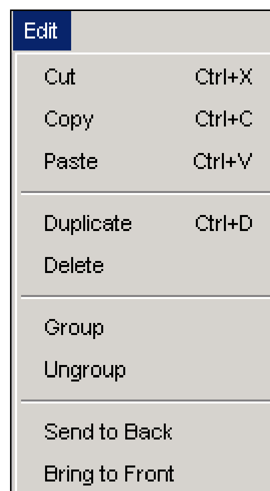

Figura 4.10 – Resultado de *CommandMenu*.

Figura 4.11 – Resultado de *CommandChoice*.

Figura 4.12 – Resultado de *CommandButton*.

Passo 3 - Instanciar Aplicações

O segundo *framework* está concluído e pode ser disponibilizado para a geração de aplicações com as características que estão presentes nele. Para comprovar a utilidade deste *framework* instanciamos a aplicação *JavaDraw* (Apêndice 1).

Assim, como aconteceu com o primeiro *framework*, novas aplicações poderão requerer características que não estão presentes no *framework* desenvolvido no segundo ciclo. Se for verificado que as novas características poderão ser utilizadas para uma larga faixa de aplicações, vale a pena partir para um próximo ciclo de desenvolvimento.

Passo 2 – Construir : Terceiro Ciclo de Desenvolvimento

Desenvolvimento do terceiro *framework.*

Passo 2.1 – Coleta de Requisitos

- Possuir conectores.
- Permitir a decoração das figuras.
- Ser executado em ambiente *Web.*

Casos de Uso Abstrato

01 – Caso de Uso Abstrato Conectores

Descrição: conectores são utilizados para interligar figuras.

02 – Caso de Uso Abstrato Decoração de Figuras

Descrição: as figuras podem ser decoradas, por exemplo, com bordas.

03 – Caso de Uso Abstrato ambiente *Web*

Descrição: o editor poderá ser gerado em ambiente *Web.*

Passo 2.2 - Analise e Detecção de *HotSpots*

Análise:

No diagrama de classes da análise (Figura 4.13) são adicionadas, em relação ao diagrama de classes do segundo ciclo de desenvolvimento (Figura 4.4), as classes *Applet, Decorator* e *Connector.*

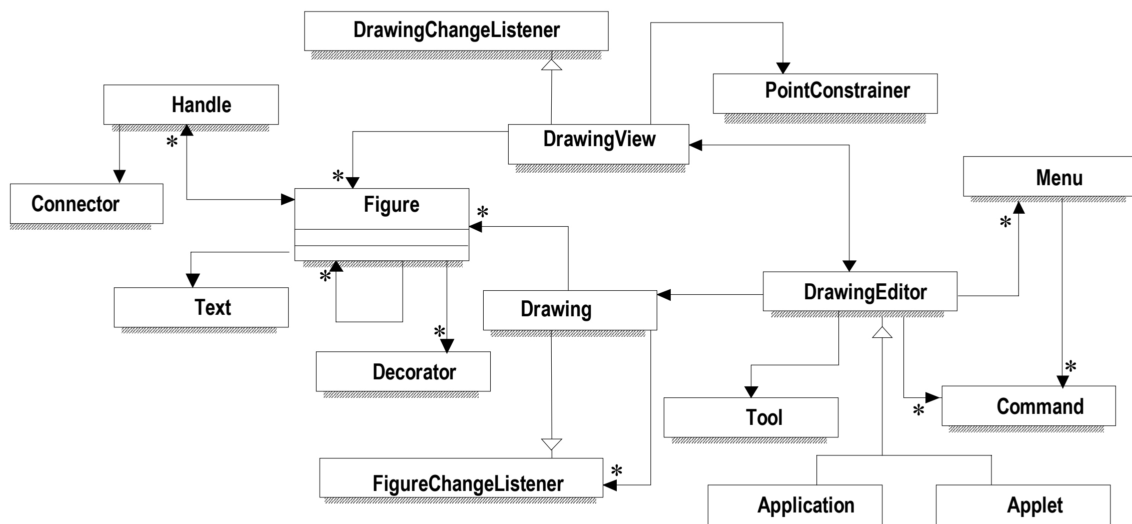

Figura 4.13 – Diagrama de classes

Detecção de *Hot Spots*:

Número: 1

Nome: conectores.

Descrição: deve ser permitido utilizar conectores entre as figuras.

Exemplos de utilização: conectar duas figuras.

Tipo de adaptação requerida:1 – Habilitar uma característica.

Grau de apoio: opcional.

Número: 2

Nome: decoração de figuras.

Descrição: deve ser permitido decorar as figuras.

Exemplos de utilização: bordas, sombras.

Tipo de adaptação requerida:1 – Habilitar uma característica.

Grau de apoio: opcional.

Número:3

Nome: variabilidade de ambiente.

Descrição: deve ser executado em ambiente Web.

Exemplos de utilização: applet.

Tipo de adaptação requerida: 1 – Habilitar uma característica.

Grau de apoio: opcional.

Passo 2.3 - Projeto:

Definição de Padrões de Projeto:

Problema: decorar as figuras com bordas, sombras e outros.

Padrão: *Decorator.*

Problema: ao criar uma conexão entre duas figuras pode haver diferentes algoritmos para calcular onde posicionar as coordenadas dos pontos extremos da conexão.

Padrão: *Strategy.*

Diagrama de Classes do Projeto

A seguir é mostrado o diagrama de classes do projeto (Figura 4.14) onde as classes, subclasses e relacionamentos são detalhados. Em relação ao diagrama de classes do projeto do segundo ciclo, foram adicionadas as interfaces *ConnectionFigure* e *LineDecoration*, as classes *LineConnection, elbowConnection, ConnectionHandle ChangeConnectionHandle, ChangeConnectionStartHandle, ChangeConnectionEnd Handle, ArrowTip* e *DrawApplet*.

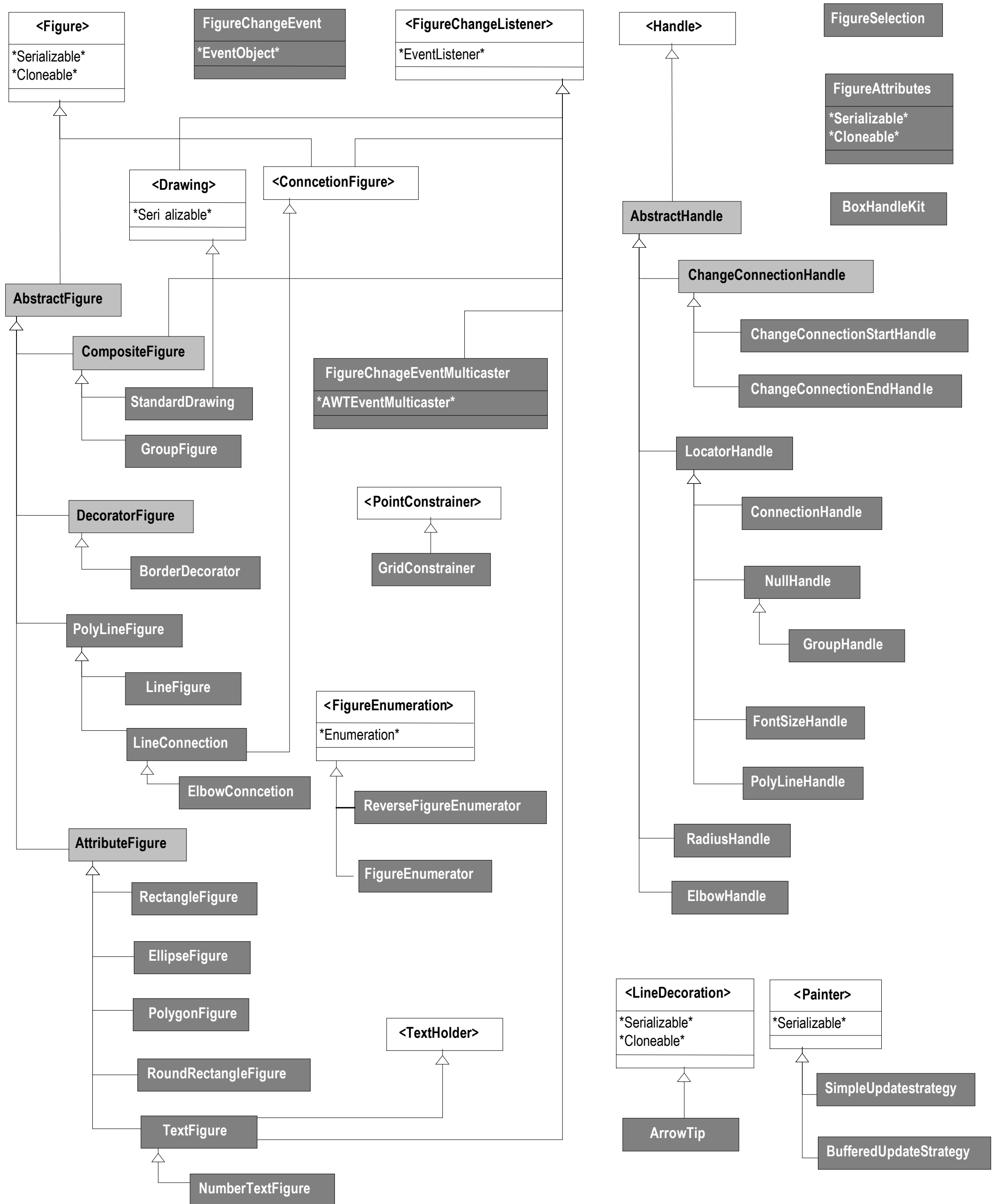

Figura 4.14 - Diagrama de classes do projeto – 3 º ciclo

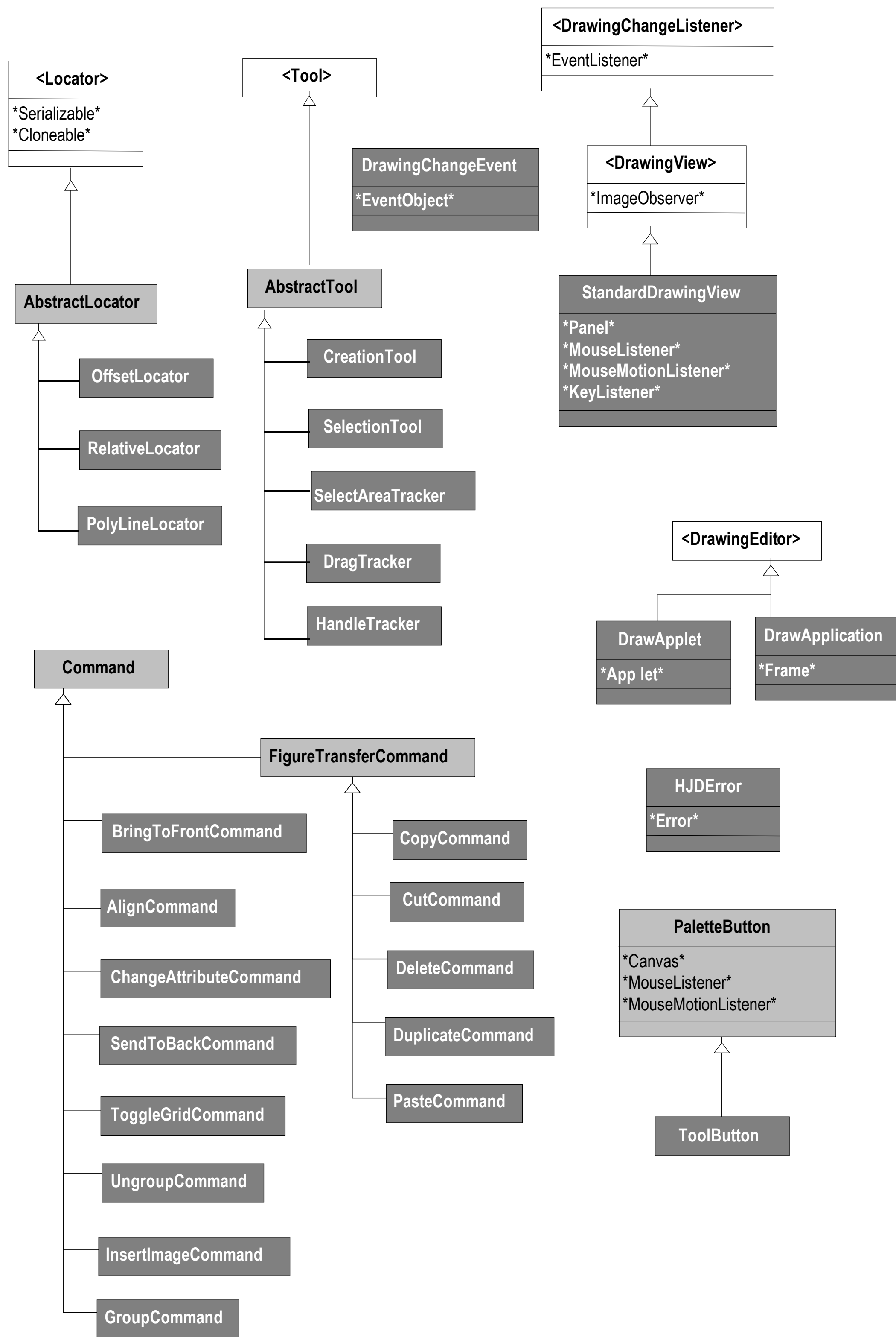

Figura 4.14 (Continuação)

Passo 2.4 – Implementação e Instanciação Exemplo

Implementação :

Foram adicionadas as classes que se encontram no projeto do *framework*

Instanciação Exemplo:

Para a instanciação exemplo foi escolhida a criação de um applet. conforme descrição a seguir.

- Cria-se uma extensão da classe *DrawApplet*, que define uma apresentação padrão para editores que executam como *applet:*

```
public class NothingApplet extends DrawApplet
```

- As ferramentas são criadas e acrescentadas na paleta utilizando o método *createTools*( ):
  - Estão implementadas as classes das figuras: *Ellipse, Rectangle, RoundRectangle, Line* e *Polygon*.

```
protected void createTools(Panel palette) {
        tool = new CreationTool(view( ), new EllipseFigure( ));
        palette.add(createToolButton(IMAGES+"ELLIPSE","Ellipse Tool", tool));
        …
}
```

- A ferramenta de texto é adicionada ao editor:

```
Tool tool = new TextTool(view(), new TextFigure());
        palette.add(createToolButton(IMAGES+"TEXT", "Text Tool", tool));
```

- A ferramenta para decoração de borda é adicionada:

```
tool = new BorderTool(view());
   palette.add(createToolButton(IMAGES+"BORDDEC", "Border Tool", tool));
```

- As ferramentas para conectores são adicionadas ao editor:

```
tool = new ConnectionTool(view(), new LineConnection());
      palette.add(createToolButton(IMAGES+"CONN", "Connection Tool", tool));
tool = new ConnectionTool(view(), new ElbowConnection());
       palette.add(createToolButton(IMAGES+"OCONN","Elbow Connection Tool", tool));
   }
```

- Os comandos são adicionados através das instâncias de *CommandChoice* e *CommandButton:*

```
protected Choice createColorChoice(String attribute) {
      CommandChoice choice = new CommandChoice()
…}
protected void createButtons(Panel panel) {
…
button = new CommandButton(new DuplicateCommand("Duplicate", fView));
      panel.add(button);
…}
```

- Cria-se uma página HTML com as *Tags Applet* para executar a aplicação como um *applet*:

```
<HTML>
<HEAD>
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=windows-1252">
<TITLE>
HTML Test Page
</TITLE>
</HEAD>
<BODY>
Teste3: NothingApplet will appear below in a Java enabled browser.<BR>
<APPLET
  CODEBASE = "."
  CODE     = "CH.ifa.draw.samples.nothing.NothingApplet.class"
  NAME     = "TestApplet"
  WIDTH    = 400
  HEIGHT   = 300
  HSPACE   = 0
  VSPACE   = 0
  ALIGN    = top
>
</APPLET>
</BODY>
</HTML>
```

O resultado é a criação de uma aplicação que é inicializada como um *applet,* com as ferramentas e a configuração padrão (Figura 4.15).

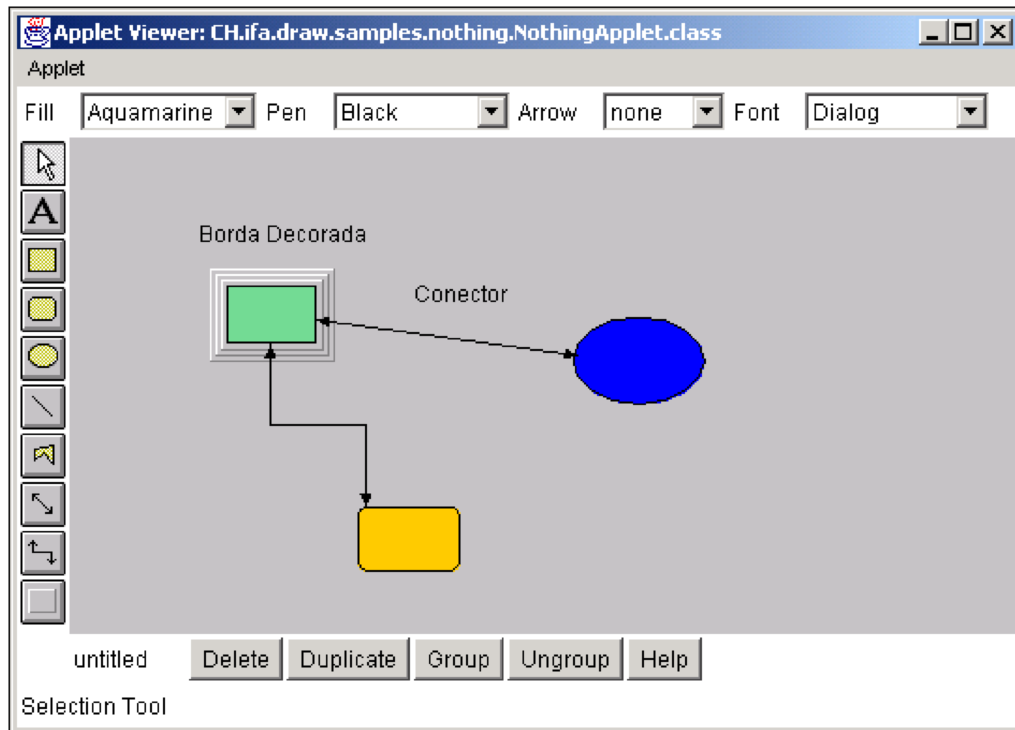

Figura 4.15 - Editor de exemplo executado como *applet*, utilizando borda, conectores e texto.

Passo 2.5 – Elaboração de Roteiro

Objetivo: Construir aplicações do tipo editor reutilizando a estrutura do *framework*.

Para construir uma aplicação utilizando o *JHotDraw* defina se será um *applet* ou uma aplicação, as figuras que serão necessárias, o comportamento de cada figura, as ferramentas para manipulá-las e os atributos que podem ser alterados em cada figura.

Para Executar em vários ambientes

O desenvolvedor tem duas maneiras de inicializar o editor:

1) Como uma aplicação: cria uma classe como subclasse da classe *Drawapplication.*
2) Como um *applet*: cria uma classe como subclasse da classe *DrawApplet.*

Para Criar Ferramentas

As ferramentas representam o modo de interação entre a interface do usuário e a área de desenho. Selecionar uma ferramenta da paleta faz com que o usuário manipule figuras, crie novas figuras ou execute operações sobre uma figura.

O método *createTools* das classes *DrawApplication* e *DrawApplet* criam por definição a ferramenta de seleção, mas pode ser sobreposto para retornar as ferramentas requeridas pela aplicação.

```
protected void createTools(Panel palette) {
        Tool tool = createSelectionTool( );
        …
}
protected Tool createSelectionTool( ) {
     return new SelectionTool(view( ));
  }
```

Há três categorias de ferramentas:

1) Ferramentas de criação: é a forma mais comum de ferramenta. O método *CreationTool* pode ser especializado para ser uma ferramenta que cria figuras tais como retângulo ou elipse. Qualquer figura pré-definida pode ser criada dessa forma.

```
Tool tool = new CreationTool(view(), new RectangleFigure());
palette.add(createToolButton(IMAGES+"RECT", "Rectangle Tool", tool));
```

2) Ferramenta de conexão: definida de maneira idêntica a *CreationTool*, exceto que é específica para criar conexões entre as figuras.

```
tool = new ConnectionTool(view(), new LineConnection());
palette.add(createToolButton(IMAGES+"CONN", "Connection Tool", tool));
```

3) Ferramentas de ação: são ferramentas utilizadas para executar ações específicas nas aplicações. A subclasse *ActionTool* fornece o ponto inicial para as ferramentas que trabalham dessa maneira.

Ex: *BorderTool* fornece a ferramenta para decorar a figura clicada com uma borda.

```
public  class BorderTool extends ActionTool
```

Action Tool fornece um método que é invocado sobre um evento do mouse. As subclasses podem sobrepor os métodos de ação para fornecer comportamento especializado da ferramenta.

```
public void mouseDown(MouseEvent e, int x, int y) {
      Figure target = drawing().findFigure (x, y);
…
      }

   public void mouseUp(MouseEvent e, int x, int y) {
      editor().toolDone();
   }
```

Para Decorar Figuras

A classe *DecoratorFigure* é utilizada para decorar figuras. O *JHotDraw* fornece por padrão a subclasse *BorderDecorator*, a qual desenha uma borda na figura

O desenvolvedor pode criar subclasses de *DecoratorFigure* e sobrescrever seus métodos para estender e filtrar comportamentos.

Para Criar Manipuladores

Manipuladores variam em comportamento. Podem mudar o tamanho da figura, mudar sua cor e outros. Manipuladores podem ser anexados a qualquer parte da figura. Há vários tipos de manipuladores, com diferentes aparências que denotam comportamentos diferentes (Figura 4.16).

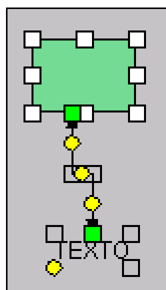

Figura 4.16 – Manipuladores de figuras.

Um retângulo vazio é um manipulador nulo serve para mostrar seleção. O retângulo branco fornece a funcionalidade de redimensionamento, o verde permite um conector ser desconectado e o ponto amarelo fornece um comportamento específico para a figura, como por exemplo, reposicionamento de segmento.

Alguns tipos de manipuladores já estão pré-definidos: *ChangeConnectionHandle, ElbowHandle, LocatorHandle* e *PolygonHandle*.

A interface *Handle* define os comportamentos importantes que um manipulador deve implementar.

O desenvolvedor pode criar subclasse da classe *AbstractHandle* (que implementa a interface *Handle*) para fornecer o comportamento desejado na aplicação.

A posição do *handle* na figura pode ser alterada através das subclasses que sobrescrevem o método *locate*. Este método retorna o ponto onde o *handle* será centrado.

```
public Point locate(Figure owner) {
        PolygonFigure plf = (PolygonFigure)owner;
            if (pointIndex < plf.pointCount())
              return ((PolygonFigure)owner).pointAt(pointIndex);
            return new Point(-1, -1);
            }
```

O comportamento do *Handle* está definido em três métodos: *invokeStart, invokeStep* e *invokeEnd*. Cada um destes métodos é chamado por eventos iniciados pelo usuário. O desenvolvedor pode decidir se o comportamento ocorrerá tão logo que o mouse é pressionado ou quando ele é liberado.

Passo 3 - Instanciando Aplicações

O terceiro *framework* está concluído. Para comprovar a utilidade deste *framework* instanciamos a aplicação *PertApplet* (Apêndice 1).

Note que qualquer aplicação ou *applet* derivada do *framework* é um *DrawingEditor*. Por definição, o *framework* fornece um frame, um menubar com alguns menus padrões, uma paleta para diferentes ferramentas (*tools*) e um *canvas* (*DrawingView*) onde os itens gráficos são colocados.

Figuras são os elementos básicos dos desenhos. Elas podem ser ligadas umas as outras através de conectores. Um conector permanece ligado a figura mesmo quando esta é movida para outro lugar. Um manipulador (*Handle*) serve como ponto de acesso visual para manipular os desenhos.

5. DISCUSSÃO DO TRABALHO REALIZADO

O desenvolvimento de *frameworks* orientados a objetos é uma tarefa complexa. Há vários problemas em se projetar e utilizar um *framework,* em especial:

- Definição de quais propriedades podem ser flexíveis (configuráveis, extensíveis ou reimplementáveis).
- Implementação dos *hot spots* e validação de suas instâncias.
- Controle da evolução do *framework*.
- Entendimento do domínio do *framework* e sua aplicabilidade.
- Controle da complexidade e dificuldade de aprendizado para sua utilização.

Para minimizar as dificuldades e orientar o desenvolvedor foi proposto um processo de desenvolvimento de *framework* utilizando a abordagem iterativa e incremental.

Um estudo de caso foi realizado seguindo a proposta do desenvolvimento iterativo e incremental. No estudo utilizamos o *framework JHotDraw* como ponto de partida e desenvolvemos três *frameworks* incrementalmente mais complexos a cada ciclo de desenvolvimento.

O estudo mostra que é possível seguindo o processo proposto, produzir *frameworks* de maneira cíclica e incremental, criando uma família de *frameworks* com crescente grau de sofisticação. Também mostra que nossa abordagem tem as seguintes vantagens:

- Controle da complexidade, dado que evita o uso de um *framework* para gerar uma aplicação que não utilize grande parte de suas funcionalidades.
- Melhor definição de quais propriedades podem ser flexíveis (configuráveis, extensíveis ou reimplementáveis) através do uso de padrões de projeto.
- Implementação dos *hot spots* com padrões de projeto e validação de suas instâncias através da geração de instâncias exemplo a cada ciclo.
- Controle da evolução do *framework* através do acompanhamento das etapas dos ciclos de desenvolvimento.
- Entendimento progressivo do domínio do *framewor*k e sua aplicabilidade através da documentação gerada a cada ciclo.

- Diminuição da dificuldade de aprendizado para sua utilização através dos roteiros progressivamente mais sofisticados que são gerados a cada ciclo.

5.1. LIMITAÇÕES

A seguir serão apresentadas algumas limitações no estudo realizado:

- O exemplo utilizado no estudo de caso é um exemplo produzido *in vitro*, não retratando integralmente o desenvolvimento no mundo real.
- Partimos de um *framework* já existente e estável, o *JHotDraw*, assim, não foram verificadas as dificuldades referentes a alocação de funcionalidades.
- São necessárias várias instanciações dos *frameworks* para melhor avaliar a reusabilidade dos mesmos.

O principal objetivo foi mostrar como se pode construir *frameworks* de modo iterativo e incremental, inclusive produzindo uma família de *frameworks,* usando a nossa abordagem.

5.2. ASPECTOS NÃO ABORDADOS

O passo de construção do *framework* abrange vários ciclos de desenvolvimento, onde a cada ciclo são incluídas novas funcionalidades ou requisitos. Surge daí várias questões que não estudamos nessa dissertação:

- Como escolher qual funcionalidade fará parte de qual ciclo.
- Como o desenvolvedor pode estimar a relação de custo/benefício na evolução de um *framework*.
- Como determinar quando se deve fazer uma próxima iteração.

.

6. CONSIDERAÇÕES FINAIS

Nos últimos anos a demanda por *software*s mais complexos, com maior qualidade e com redução no tempo e esforço de desenvolvimento tem aumentado. Para satisfazer a esses requisitos a reutilização é a forma encontrada, pois em contraposição ao desenvolvimento de todas as partes do sistema, reutilizar recursos previamente produzidos, torna possível produzir *software*s em menos tempo, com menos esforço e com mais qualidade.

A presente dissertação foi motivada pelo reconhecimento das vantagens do desenvolvimento de *software* com reutilização. Nesse sentido, produziu-se uma pesquisa direcionada para a área de desenvolvimento de *framework* de aplicações orientadas a objetos entre as abordagens de reutilização existentes.

O desenvolvimento baseado em *frameworks* apresenta vantagens em relação a abordagens tradicionais por promover a reutilização da arquitetura e do código, porém, a carência de processos de desenvolvimento voltados para a construção de *frameworks* dificulta a sua adoção.

Diante desse cenário, foram apresentados nos capítulos anteriores os conceitos de *framework* de aplicações orientadas a objetos, discutidas as principais dificuldades e problemas no seu desenvolvimento, mostrados processos de desenvolvimento existentes e proposto um processo iterativo e incremental.

6.1. CONTRIBUIÇÕES

As principais contribuições introduzidas por esta dissertação são:

- Proposta um processo para o desenvolvimento de *frameworks* de aplicações orientado a objetos com base no processo iterativo e incremental.
- O processo proposto conduz a criação de um *framework* a cada ciclo do processo de desenvolvimento, permitindo que aplicações sejam desenvolvidas de acordo com o *framework* que melhor atenda as suas necessidades.
- A abordagem incorpora o uso de padrões de projeto durante os ciclos de desenvolvimento para introduzir flexibilidade ao projeto.
- O modelo produz uma família de *frameworks,* facilitando o aprendizado por parte do desenvolvedor e levando a (re)utilização no desenvolvimento de aplicações.

6.2. TRABALHOS FUTUROS

Para trabalhos futuros, em adição ao estudo realizado, são sugeridos:

- Extensão do processo de desenvolvimento para incluir etapas de teste e manutenção de *framework.*
- Estudo de técnicas de documentação específicas para esse tipo de família de *framework.*
- Estudo de como escolher a melhor forma para se evoluir um *framework.*
- Avaliação de custo/benefício para evolução do *framework.*

A continuidade da pesquisa reportada por esta dissertação ajudará a tornar o processo de desenvolvimento de *framework* uma prática menos complexa.

REFERÊNCIAS BIBLIOGRÁFICAS

ARANGO, G.; PRIETO-DÍAZ, R. **Domain Analysis Concepts and Research Directions**. In: R. Prieto-Díaz e G.Arango (eds), Domain Analysis and Software Systems Modeling. IEEE Computer Society Press Tutorial, 1991.

BALDWIN, RICHARD G. **Object-Oriented Programming**, Java Programming, Lecture Notes 4, 1999. Disponível em: <http://www.austin.cc.tx.us/baldwin/>. Acesso em: 21 de out. 2002.

BENTE, ANDA; DAG, SJEBERG; MAGNE, JORGENSEN. **Quality and Understandability of Use Case Models**, Department of Informatics - University of Oslo Norway, Simula Research Laboratory, Norway, 2001.

BOOCH, GRADY. Designing an Application Framework. **Dr. Dobb´s Journal 19**, No. 2, 1994. Disponível em: <www.ipd.hk-r.se/michaelm/fwpages/files/ BuildingFrameworks.ps >. Acesso em: 15 de out. 2002.

BOSH, J.; MOLIN, P.; MATTSSON, M.; BENGTSSON, P.; FAYAD, M. **Framework Problems and Experience**. In Fayad, M.E.; Johnson, R.E; Schmidt, D.C. Building Application Frameworks: Object-Oriented Foundations of *Framework* Design, John Wiley & Sons.p.55-82, 1999.

DAHL, O. J.; MYHRHAUG, B.; NYGAARD, K. SIMULA-67 **Common Base Language**. Norwegian Computing Center, Oslo, Norway, 1970.

DOUGLAS, KIRK. **Pratical 4 Solutions**, Department of Computer Science, University of Strathclyde, 2002. Disponível em : <https://www.cis.strath.ac.uk/teaching/ ug/classes/52.440/prac4Sol-2002.html>. Acesso em: 07 de dez. 2002.

FAYAD, MOHAMED; DOUGLAS , C. SCHMIDT. **Building Application Frameworks: Object Oriented Foundations of Framework Design**, 1999.

FROEHLICH, G.; HOOVER, H; LIU, L; SORENSON, P. **Hooking into Object-Oriented Application Frameworks**; Proceedings of International Conference on Software Engineering; 1997.

FROEHLICH, G.; HOOVER, H; LIU, L; SORENSON, P. **Reusing Hooks**; In Fayad, M.E.,Schmidt, Douglas C; Building Application Frameworks*:* Object Oriented Foundations of *Framework* Design, 1999; pp 219-236.

GAMMA, E. R. HELM; JOHNSON, R.; VLISSIDES J. **Design Patterns: Elements of Reusable Object-Oriented Software**. Reading, MA: Addison Wesley, 1995.

GAMMA, E. **JHotDraw Framework**, 2001 Disponível em: <http://members.pingnet.ch/gamma/JHD-5.1.zip>. Acesso em: 05 de abr. 2002.

GERHARD, FISCHER. **Cognitive View of Reuse and Redesign**. IEEE *software*, 4, 60-72, 1987.

HOLMEVIK, JAN RUNE. **The History of Simula**. Institute for Studies in Research and Higher Education, Center for Technology and Society University of Trondheim, 1995.

JOHNSON, RALPH E. **Documenting Frameworks Using Patterns**; In: Proceedings of the Conference on Object-oriented programming system, languages, and applications – OOPSLA'92, pp. 63-76, Vancouver, British Columbia, Canadá, 1992. Disponível em: <ftp://st.cs.uiuc.edu/pub/patterns/papers/documenting-*frameworks*.ps>. Acesso em: 10 de maio 2001.

JOHNSON, RALPH E. **How to Design Framework**. Notes for Proceedings of the Conference on Object-oriented programming system, languages, and applications - OOPSLA93, University of Ilinois, 1993. Disponível em: <ftp://st.cs.uiuc.edu/ pub/papers/frameworks/OOPSLA93-frmwk-tut.ps>. Acesso em: 10 de maio 2001.

JOHNSON, RALPH. E.; FOOTE, B. Designing Reusable Classes. **Journal of Object-Oriented Programming**, 1988. Disponível em:<ftp://st.cs.uiuc.edu/pub/papers/frameworks/designing-reusable-classes.ps>. Acesso em: 10 de maio 2001.

LAJOIE, RICHARD; KELLER, RUDOLF K. **Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts and Motifs in Concert**; In: Proceedings of the 62nd Congress of the Association Canadienne Francaise pour l'Avancement des Sciences, Montreal, Canada, 1994 Disponível em: <ftp://st.cs.uiuc.edu/pub/patterns/papers/acfas.ps>. Acesso em: 15 de mai. 2001.

LANDIN, NIKLAS; NIKLASSON, AXEL. **Development of Object Oriented Frameworks**. 1995, 154 f. Master Thesis. Department of Communication Systems, Lund University, Sweden. Disponível em: <http://www.ipd.hk-r.se/michaelm/thesis/index.html>. Acesso em: 10 de Jan. 2002.

LARMAN, CRAIG. **Utilizando UML e Padrões Uma Introdução a Análise e ao Projeto Orientado a Objetos**. Editora BookMan, 1999.

MARKIEWICZ, MARCUS; LUCENA, CARLOS. **Understanding Object-Oriented *Framework* Engineering**, PUC-Rio Inf.MCC38/00, 2000.

MATTSSON, MICHAEL. **Evolution and Composition of Object-Oriented Frameworks**, 2000, 150 f. Thesis (Doctor of Philosophy in Engineering). Faculty of Technology, University of Karlskrona, Ronneby.

MATTSSON, MICHAEL. **Object-oriented Frameworks: A Survey of Methodological Issues**, 1996, 130 f. Licentiate Thesis. Department of Computer Science, Lund University. Sweden. Disponível em: <http://www.ipd.hk-r.se/michaelm/thesis/index.html>. Acesso em: 10 de Jan. 2002.

NYGAARD, K.; DAHL, O. J. **The Development of the Simula Languages**. In: History of Programming Languages. Academic Press, New York, 1981, 439-493.

PÁDUA, WILSON. **Engenharia de Software, Fundamentos, Métodos e Padrões**. Editora LTC, 2001.

PREE, WOLFGANG. **Design Patterns for Object-Oriented Software Development**. Reading, MA:Addison-Wesley, 1995.

PREE, W. **Hot spot Driven Development**. In Fayad, M.E.; Johnson,R.E; Schmidt, D.C. Building Application *Frameworks*: Object-Oriented Foundations of *Framework* Design, John Wiley & Sons. p.379-393, 1999.

RICARTE, IVAN LUIZ. **Documentação de Código com JavaDoc**. DCA/FEEC/UNICAMP, 2000. Disponível em: <http://www.dca.fee.unicamp.br/ courses/PooJava/javaenv/javadoc.html>. Acesso em: 07 de jan. 2003.

ROBERTS, D.; JOHNSON, R. E. **Evolving Frameworks - A Pattern Language for Developing Object-Oriented Frameworks**. University of Ilinois, 1998.

RUMBAUGH, J.; JACOBSON, I.; BOOCH, G. **The Unified Modeling Language** Reference Manual. Addison-Wesley, 1998.

SCHMID, H. A. ***Framework* Design by Systematic Generalization** . In Fayad, M.E.; Johnson,R.E; Schmidt, D.C. Building Application Frameworks: Object-Oriented Foundations of *Framework* Design, John Wiley & Sons.p.353-378, 1999.

SHALLOWAY, ALAN; TROTT, JAMES. **Design Patterns Explained. A New Perspective on Object-Oriented Design**. Ed. Addison-Wesley, 2002.

SILVA, RICARDO PEREIRA; PRICE, ROBERTO. **O Uso de Técnicas de Modelagem no Projeto de Frameworks Orientados a Objetos**. 1997. Disponível em: <http://www.inf.ufsc.br/~ricardo/publications/Asoo97.pdf>. Acesso em: 20 de out. 2002.

SILVA, RICARDO PEREIRA. **Suporte ao Desenvolvimento e Uso de Frameworks e Componentes**. 2000. 262 f. Tese (Doutorado em Ciência da Computação) -

Universidade Federal do Rio Grande do Sul, Instituto de Informática, Rio Grande do Sul.

TALIGENT WHITE PAPERS. **Building Object Oriented Framework**, inc, 1994. Disponível em <http://www.ipd.hk-r.se/michaelm/fwpages/fwbibl.html>. Acesso em: 20 de out. 2002.

VILJAMAA, ANTTI. **Pattern-Based Framework Annotation and Adaptation – A Systematic Approach**. 2001. 118 f. Thesis (Licenciate Thesis in Computer Science), University of Helsinki, Finland.

APÊNDICE 1 - Aplicações Instanciadas do *JHotDraw*

Aplicação desenvolvida a partir do primeiro *framework* : *Planet Application*

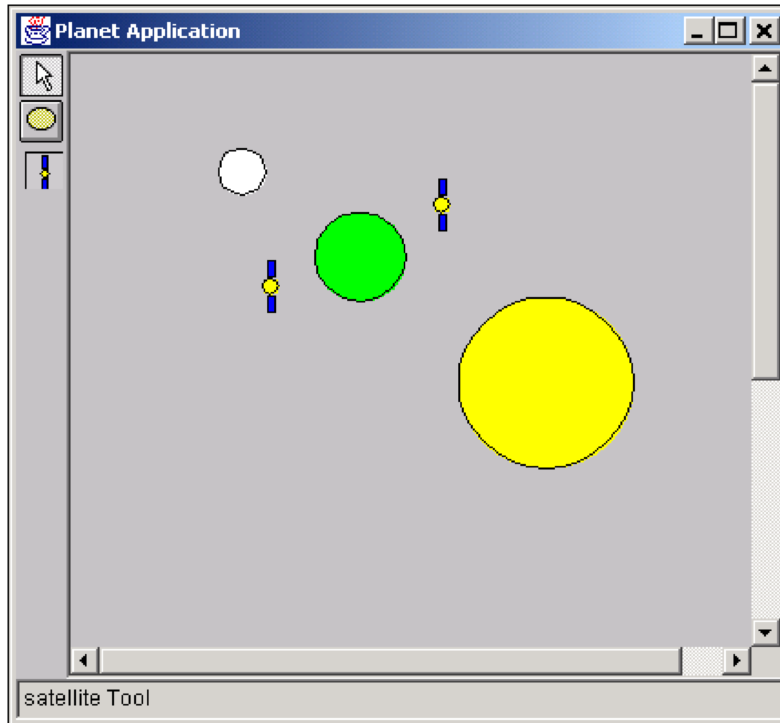

Figura A1.1 – *Interface* de *planet application*.

A aplicação *Planet* fornece ferramentas para representação de planetas e satélites - Figura A1.1 (Douglas, 2002).

A implementação consiste da subclasse *PlanetFigure*. Um planeta pequeno (menor que 40) é sempre exibido com a cor branca, o médio (entre 40 e 80) com a cor verde e um grande (maior que 80) com a cor amarela.

```
public class PlanetFigure extends EllipseFigure{
public PlanetFigure(){
super();
}…
public Color getFillColor(){
Rectangle rect = displayBox();
int size = Math.max(rect.width, rect.height);
if(size <= 40) {
return Color.white;
  }
  else if(size > 40 && size <= 80){
  return Color.green;
  }
  else
  return Color.yellow;
  }
  }
```

A Subclasse *SatelliteFigure* é composta de dois retângulos e um elipse:

```
public class SatelliteFigure extends GroupFigure {

  public SatelliteFigure() {
     super();

     RectangleFigure r1 = new RectangleFigure(new Point(3,0), new Point(8,10));
     EllipseFigure e1 = new EllipseFigure(new Point(0,10), new Point(10,20));
     RectangleFigure r2 = new RectangleFigure(new Point(3,20), new
Point(8,30));

     r1.setAttribute("FillColor", Color.blue);
     e1.setAttribute("FillColor", Color.yellow);
     r2.setAttribute("FillColor", Color.blue);

     super.add(r1);
     super.add(e1);
     super.add(r2);
```

A aplicação é criada como subclasse de *DrawApplication*. As ferramentas para criação das figuras *Planet* e *Satellite* são adicionadas ao painel:

```
public class OrreryApplication extends DrawApplication{

    public OrreryApplication(String title){
        super(title);
    }

 protected void createTools(Panel panel){
        super.createTools(panel);
        CreationTool planetTool = new CreationTool(view(), new PlanetFigure());
        ToolButton planetButton = new ToolButton(this, IMAGES+"ELLIPSE",
"Planet Tool", planetTool);
        panel.add(planetButton);
        CreationTool satelliteTool = new CreationTool(view(), new SatelliteFigure());
        ToolButton satelliteButton = new ToolButton(this, IMAGES+"SATELLITE",
"satellite Tool", satelliteTool);
        panel.add(satelliteButton);

    }
}
```

Aplicação desenvolvida a partir do segundo *framework*: *JavaDraw*

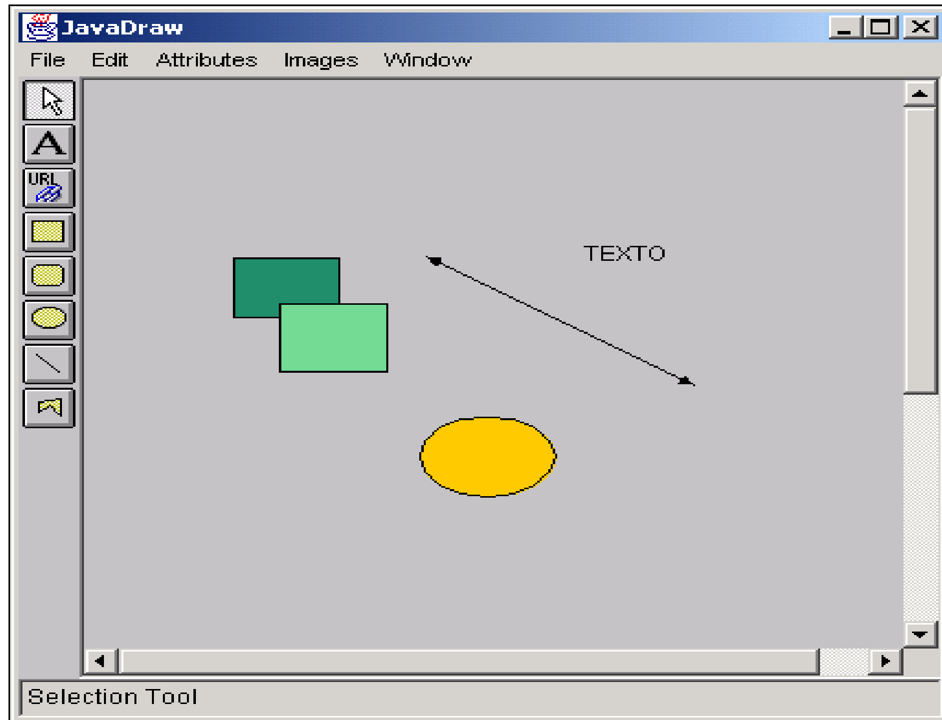

Figura A1.2 – *Interface* de *JavaDraw*.

JavaDraw é um editor que fornece menu, ferramentas, manipuladores e comandos para tratamento das figuras – Figura A1.2 (Gamma, 2001).

Em relação ao *framework* do primeiro ciclo de desenvolvimento, a estrutura deste segundo *framework* permite a criação de menus, texto e comandos de edição.

A implementação da aplicação consiste da criação da subclasse *JavaDraw*, extensão da classe *DrawApplication*. O menu é criado sobrescrevendo o método *createMenus*( ).

```
public  class JavaDrawApp extends DrawApplication{
…
protected void createMenus(MenuBar mb) {
                super.createMenus(mb);
                mb.add(createImagesMenu());
                mb.add(createWindowMenu());
   }
```

É criada a subclasse *mySelectionTool* que interpreta duplo clique do mouse e inspeciona a figura.

```
public  class MySelectionTool extends SelectionTool {

   public MySelectionTool(DrawingView view) {
      super(view);
   }

   /**
    * Handles mouse down events and starts the corresponding tracker.
    */
   public void mouseDown(MouseEvent e, int x, int y) {
      if (e.getClickCount() == 2) {
         Figure figure = drawing().findFigure(e.getX(), e.getY());
         if (figure != null) {
            inspectFigure(figure);
…
```

O método *CreateSelectionTool* é sobrescrito para chamar *MyselectionTool*:

```
protected Tool createSelectionTool() {
      return new MySelectionTool(view());
   }
```

As ferramentas são criadas e adicionadas ao painel:

```
protected void createTools(Panel palette) {
        super.createTools(palette);

        Tool tool = new TextTool(view(), new TextFigure());
        palette.add(createToolButton(IMAGES+"TEXT", "Text Tool",
tool));
tool = new URLTool(view());
        palette.add(createToolButton(IMAGES+"URL", "URL Tool",
tool));
```

Aplicação desenvolvida a partir do terceiro *framework*: *Pert Applet*

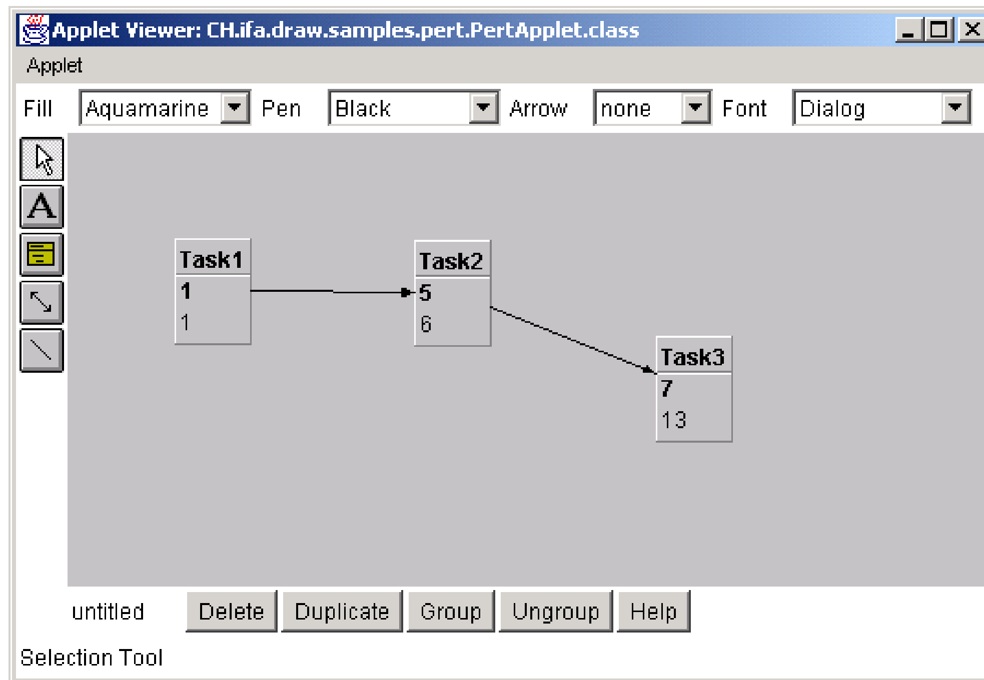

Figura A1.3 – *Interface* de *PertApplet*.

O *applet PERT* (Gamma, 2001) fornece ferramentas, manipuladores e comandos para construir um diagrama *PERT* (Figura A1.3). Uma tarefa é mostrada como uma figura composta de uma caixa com uma divisão para o nome da tarefa, uma divisão para o tempo inicial e outra para o prazo final. Somente o nome da tarefa e sua duração é editável. O prazo final é calculado automaticamente pelo editor *PERT*. Dependências circulares são exibidas com a cor vermelha.

Em relação ao *framework* do segundo ciclo de desenvolvimento, a estrutura deste terceiro *framework* permite a criação de um applet, conectores e figuras decoradas.

A implementação do editor consiste da criação da subclasse *PertFigure.*

```
public class PertFigure extends CompositeFigure {
…
   public PertFigure(){
      TextFigure name = new TextFigure();
       name.setFont(fb);
       name.setText("Task");
       //name.setAttribute("TextColor",Color.white);
       add(name);

       NumberTextFigure duration = new NumberTextFigure();
       duration.setValue(0);
       duration.setFont(fb);
       add(duration);
```

É criada a subclasse *PertFigureCreationTool* para criar novas figuras Pert em substituição a classe *CreationTool* do *framework*:

```
public  class PertFigureCreationTool extends CreationTool {

   public PertFigureCreationTool(DrawingView view) {
      super(view);
   }

   /**
    * Creates a new PertFigure.
    */
   protected Figure createFigure() {
      return new PertFigure();
   }
}
```

É criada a subclasse *PertDependency* para controle de dependências entre as tarefas:

```
public class PertDependency extends LineConnection {
…
public void handleConnect(Figure start, Figure end) {
      PertFigure source = (PertFigure)start;
      PertFigure target = (PertFigure)end;
      if (source.hasCycle(target)) {
         setAttribute("FrameColor", Color.red);
      } else {
         target.addPreTask(source);
         source.addPostTask(target);
         source.notifyPostTasks();
      }
   }
```

A aplicação é criada como subclasse de *DrawApplet*. As ferramentas de texto, figura *Pert*, dependência e linha são adicionadas ao painel:

```
public  class PertApplet extends DrawApplet {
…
protected void createTools(Panel palette) {
      super.createTools(palette);

      Tool tool;
      tool = new TextTool(view(), new TextFigure());
      palette.add(createToolButton(IMAGES+"TEXT", "Text Tool", tool));

      tool = new PertFigureCreationTool(view());
      palette.add(createToolButton(PERTIMAGES+"PERT", "Task Tool", tool));

      tool = new ConnectionTool(view(), new PertDependency());
      palette.add(createToolButton(IMAGES+"CONN", "Dependency Tool", tool));

      tool = new CreationTool(view(), new LineFigure());
      palette.add(createToolButton(IMAGES+"Line", "Line Tool", tool));
   }
```

APÊNDICE 2 - Padrões de Projeto

Padrões *Composite* e *Template Method*

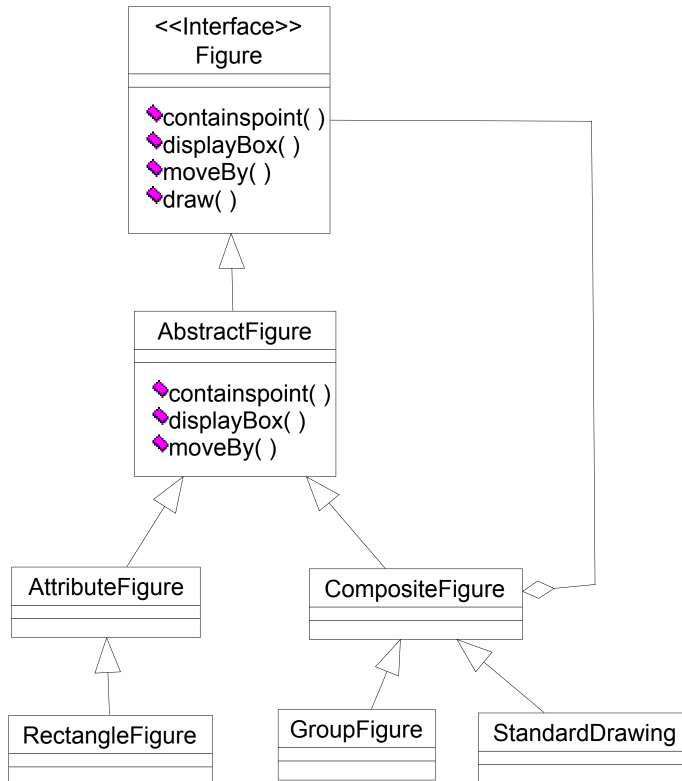

Figura A2.1 –Padrões de projeto *composite* e *template method* no *JhotDraw*.

O padrão *composite* é um padrão estrutural que descreve como construir uma hierarquia de classes composta por dois tipos de objetos: primitivos e compostos. Permite que sejam tratados de maneira uniforme objetos individuais e composição de objetos (Gamma e outros, 1995).

No *JHotDraw* a classe *CompositeFigure* permite tratar um conjunto de figuras como uma única figura (Figura A2.1).

O padrão *TemplateMethod* define a estrutura de um algoritmo deixando alguns passos para a subclasse. O método *Template* é um método da classe abstrata que chama um ou mais métodos para preencher parte de suas tarefas.

A classe Abstrata *AttributeFigure* utiliza o padrão *Template Method*. O método *draw*() dessa classe é o método *template* que chama dois outros métodos. *AttributeFigure* representa a classe abstrata desse padrão e suas subclasses representam as classes concretas (Figura A2.1).

Padrão *Decorator*

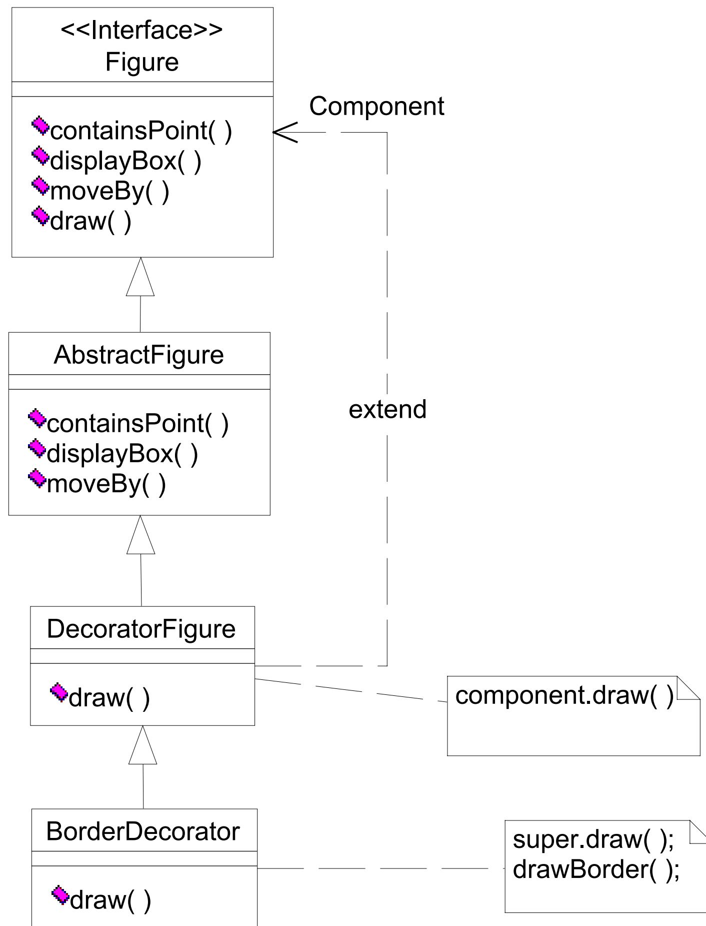

Figura A2.2 – Padrão de projeto *decorater* no *JhotDraw*.

O padrão *decorator* adiciona responsabilidades aos objetos dinamicamente. Decoradores fornecem uma alternativa flexível para que subclasses estendam funcionalidades (Gamma e outros, 1995).

No *JHotDraw* a classe *DecoratorFigure* é utilizado para decorar as figuras com bordas, sombras e outros. (Figura A2.2).

Padrões *Observer, State e Strategy*

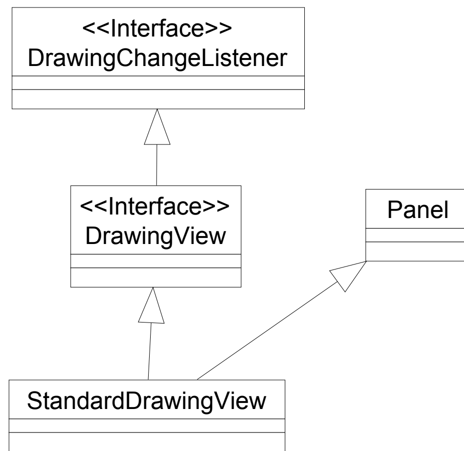

Figura A2.3 – Padrões de projeto *observer* e *strategy* no *JhotDraw*.

O *DrawinView* implementa os seguintes padrões de projeto (Figura A2.3 e Figura A2.4):

Observer: O objetivo do padrão *observer* é a notificação e atualização automática de objetos dependentes no caso de mudança de estado de um objeto (Gamma e outros, 1995). O *DrawingView* observa os desenhos que se alteram através da interface *DrawingChangeListener*

Strategy: Define uma família de algoritmos, encapsula cada um e os torna intercambiáveis. *strategy* faz com que o algoritmo varie independente dos clientes que o utiliza (Gamma e outros, 1995). O *DrawingView* é o cliente no padrão *strategy*

State: O padrão *state* permite um objeto alterar seu comportamento quando seu estado interna muda (Gamma e outros, 1995). Encapsula estados em classes concretas. O *DrawingView* representa o papel de contexto do padrão *state* e a ferramenta (*tool*) é o estado.

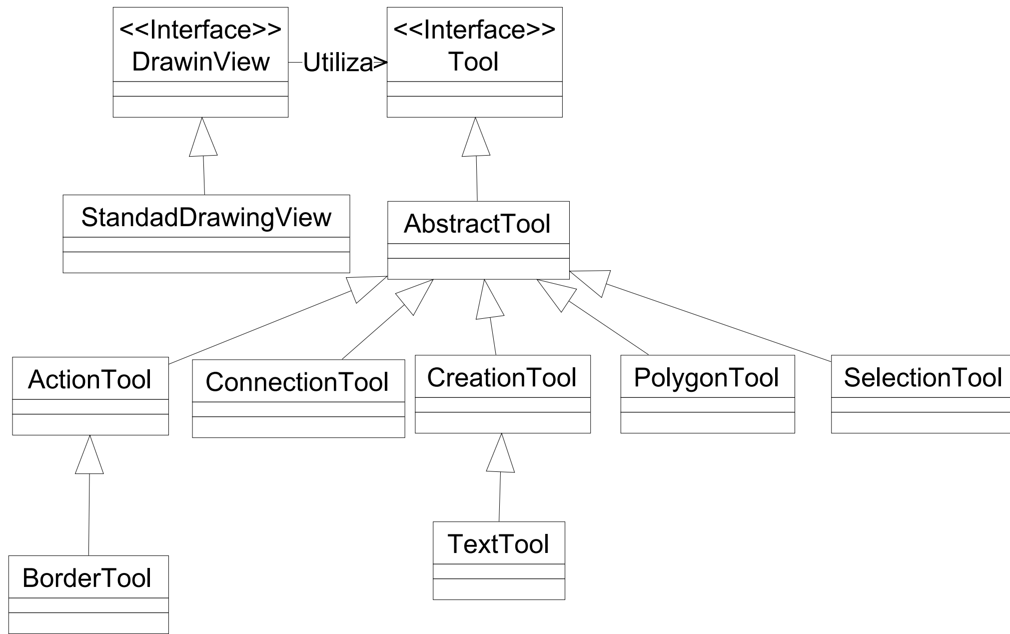

Figura A2.4 – Padrão de projeto *state* no *JhotDraw.*

Padrão *Factory Method*

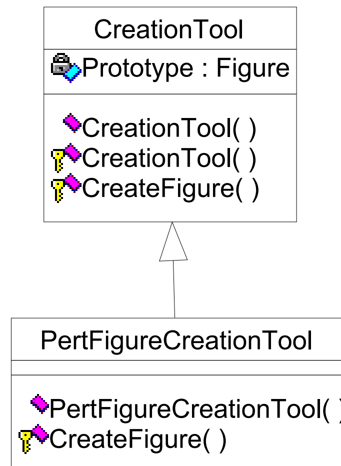

Figura A2.5 –Padrão de projeto *factory method* no JhotDraw.

O padrão *factory method* define uma interface para criar objetos, mas deixa a subclasse decidir que classe instanciar (Gamma e outros, 1995).

No *JHotDraw* o método *CreateFigure( )* é o *factory method* que retorna objetos do tipo figura *Pert* (Figura A2.5).

Padrão *Template Method*

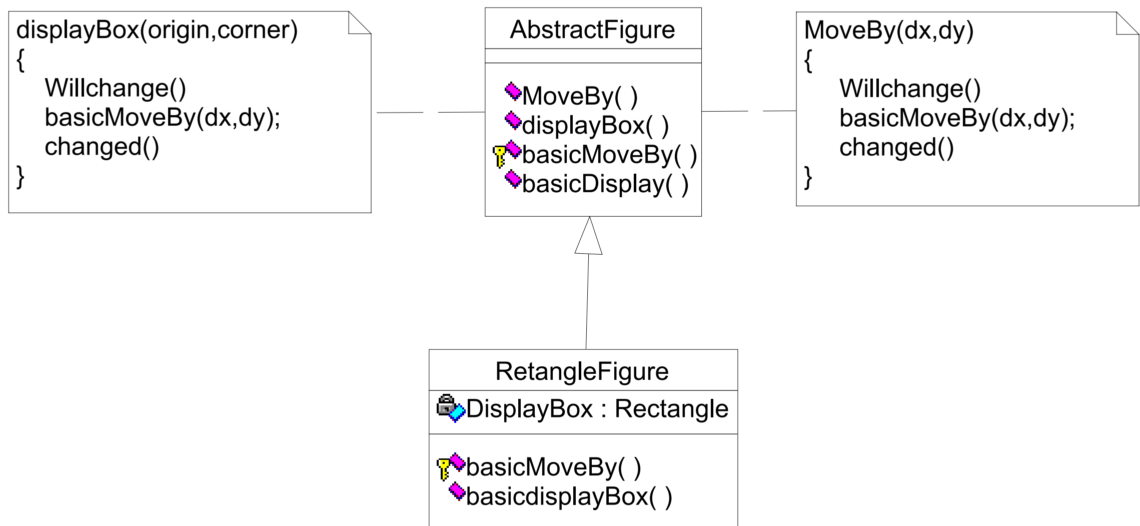

Figura A2.6 –Padrão de projeto *Template Method* no *JhotDraw*.

O padrão *TemplateMethod* define a estrutura de um algoritmo deixando alguns passos para a subclasse (Gamma e outros, 1995).

O método *Template* é um método da classe abstrata que chama um ou mais métodos para preencher parte de suas tarefas.

No *JHotDraw* a classe *AbstractFigure* define um comportamento padrão para as subclasses concretas (Figura A2.6).

Padrão *Mediator*

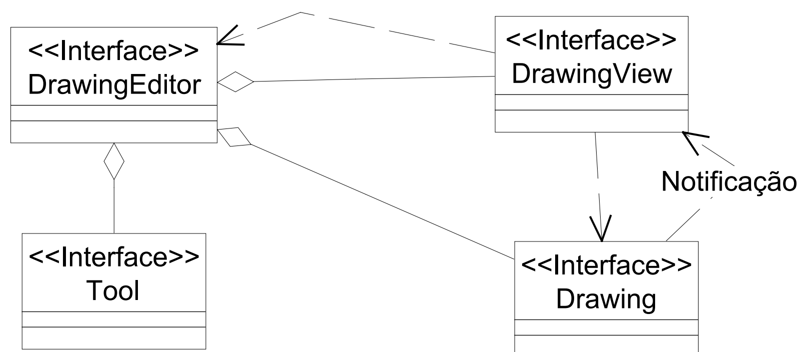

Figura A2.7 –Padrão de projeto *mediator* no *JhotDraw*.

O padrão *mediator* define um objeto que encapsula como um conjunto de objetos interage (Gamma e outros, 1995).

No *JHotDraw* o *DrawingEditor* define a interface para coordenar os diferentes objetos que fazem parte do editor de desenho (Figura A2.7).

Padrão *Command*

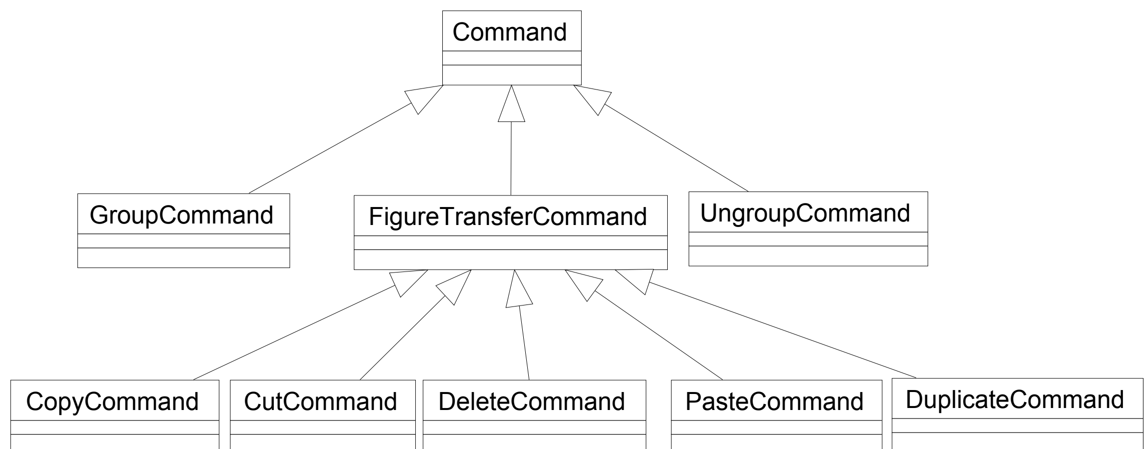

Figura A2.8 –Padrão de projeto *command* no *JhotDraw*.

O padrão *command* encapsula uma requisição como um objeto, deixando que clientes sejam parametrizados com diferentes requisições (Gamma e outros, 1995).

No *JHotDraw* os comandos encapsulam uma determinada ação a ser executada (Figura A2.8). O *JHotDraw* inclui componentes que executam os comandos : *CommandMenu*, *CommandChoice* e *CommandButton.*

APÊNDICE 3 - Amostra de UML e Notações Relacionadas

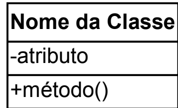

Figura A3.1 - Classe (Larman, 1999).

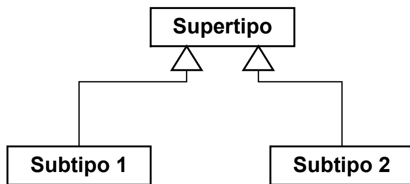

Figura A3.2 - Generalização (Larman, 1999).

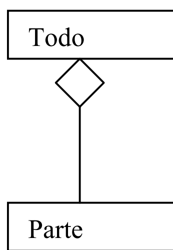

Figura A3.3 - Agregação

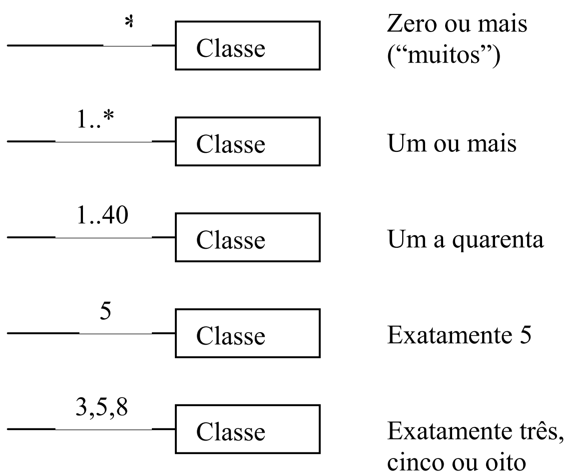

Figura A3.4 - Multiplicidade (Larman, 1999).

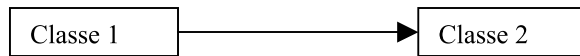

Figura A3.5 - Associações (Larman, 1999).

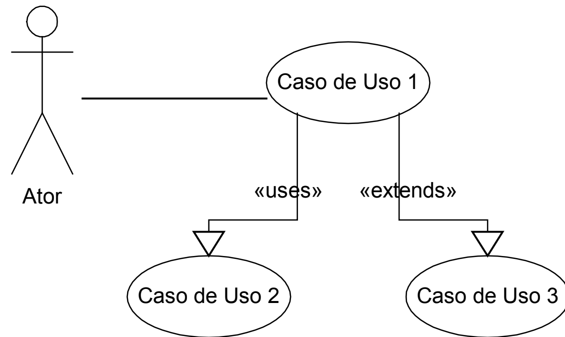

Figura A3.6 - Diagrama de Caso de Uso (Larman, 1999).