Universidade Federal do Rio de Janeiro

Escola Politécnica

Departamento de Eletrônica e de Computação

# Framework de Aplicações utilizando Plug-ins

Autor:

Alexandre de Almeida Rego Campinho

Autor:

Leonardo Campos de Melo

Orientador:

Flávio Luis de Mello, DSc

Examinador:

Prof Roberto Lins de Carvalho, PhD

Examinador:

Prof. Jorge Lopes de Souza Leão, DIng

DEL

Junho de 2009

## DEDICATÓRIA

*Dedico esse trabalho aos meus pais que me apoiaram ao longo desse caminho, aos meus irmãos que estiveram sempre dispostos a ajudar, mesmo que a distancia, às vezes, tenha dificultado.*
*Faço também uma dedicação especial a minha avó, que me recebeu de braços abertos, com muito carinho e atenção, em sua casa. E aos meus tios e primos que me deram suporte, e foram companhias muito agradáveis, ao longo dessa jornada.*
*Aos amigos de faculdade que foram companheiros para todos os momentos e fazem parte desta conquista.*
*Aos meus amigos de Jacareí, que apesar da distancia, fazem parte da minha vida e sempre farão.*
*Com certeza não seria capaz sem ajuda dessas pessoas.*
*Muito Obrigado.*

*Alexandre de Almeida Rego Campinho*

*Este trabalho é dedicado a todos meus familiares e companheiros de faculdade, pela colaboração com meu crescimento profissional e pessoal, sem eles eu não conseguiria chegar onde cheguei.*
*Dedico especialmente aos meus pais por todo apoio e carinho necessário para completar esta longa caminhada.*
*Muito obrigado.*

*Leonardo Campos de Melo*

## AGRADECIMENTO

Aos nossos pais que nos deram a vida e nos ensinaram a vivê-la com dignidade e honestidade e que se doaram por inteiro e renunciaram alguns de seus sonhos para que muitas vezes pudéssemos realizar os nossos.

Aos amigos e familiares que compartilharam e alimentaram nossos ideais, compreendendo nossa falta de tempo, finais de semana de estudo e noites em claro, tendo sempre uma palavra e um sorriso a nos incentivar para que chegássemos ao final de mais uma etapa.

Não podemos deixar de agradecer nosso amigo, Thiago Arakaki, que nos ajudou diversas vezes a sanar dúvidas durante o desenvolvimento.

Somos gratos a todos os professores do DEL pelas lições e conhecimentos passados, em especial ao professor e orientador Flávio Luis de Mello, pelo apoio e incentivo na construção desse trabalho.

Vocês fazem parte dessa vitória e da nossa história.

## RESUMO

O *Eclipse RCP* é um conjunto mínimo de plug-ins necessários para a criação de aplicações de propósitos gerais e baseadas na arquitetura do Eclipse. Ele permite que os desenvolvedores utilizem da arquitetura do Eclipse para projetar aplicações *standalone* flexíveis e extensíveis reutilizando muitas das funcionalidades já existentes e padrões de código inerentes ao próprio Eclipse. Muitas pessoas que já construíram, ou estão concluindo, aplicações RCP declaram que maior valor retornado por usar RCP é que permite sejam criadas rapidamente aplicações profissionais, com um look-*and*-feel nativo, multiplataforma, permitindo que o desenvolvedor foque no valor adicional da sua aplicação. Hoje em dia algumas grandes empresas desenvolvem, ou já desenvolveram aplicações RCP voltadas para o meio comercial ou open-source, temos como exemplo a NASA, IBM, Adobe, Novell, entre outras. O feedback em relação a essa tecnologia foi muito animador para comunidade. O presente projeto pretende mostrar como nos utilizamos de todos os benefícios da tecnologia RCP, para fazer o *refactoring* de uma aplicação já existente, buscando construir um projeto que atendesse todos os requisitos como: desacoplamento, escalabilidade, código reutilizável e de fácil manutenção.



Palavras-Chave: Eclipse RCP, plug-ins, stand-alone, multiplataforma.

## ABSTRACT

Eclipse RCP is a minimal plug-in set required for building applications based on Eclipse Plug-in architecture. It allows developers to use the Eclipse architecture in order to design flexible and extensible stand-alone applications by reusing a lot of existing functionality and coding patterns avaiable from Eclipse. Many RCP application developers state that the main benefit from using RCP is that it allows them to quickly build professional-looking application, with native look-and-feel, for multiple platforms. Nowadays, some companies and organizations, such as, NASA, IBM, Adobe, Novell, have already had some sort of RCP experience, on bulding commercial and open-source application. Their feedback for the community about the technology was very encouraging. This project presents how to use all RCP benefits, in order to refactor an existing application, seeking for a project with such features as uncoupling, scalable, reusable and easy maintainance code.

SIGLAS

UFRJ – Universidade Federal do Rio de Janeiro

POO – Programação Orientada à Objeto

RCP – Rich Client Platform

POE – Programação Orientada a Eventos

GEF - Graphical Editing Framework

J2EE – Java 2 Enterprise Edition

J2ME – Java 2 Mobile Edition

PDA – Personal Digital Assistants

RCP – Rich Client Plataform

IDE – Integrated Development Environment

POE – Programação Orientada à Eventos

MVC – Model-View-Controller

UI – User Interface

SWT – Standard Widget Toolkit

JRE – Java Runtime Environment

PDE – Plug-in Development Environment

JAR – Java ARchive

CLR - Common Language Runtime

EPL - Eclipse Public License

POC – Proof of Concept

# Sumário

# Lista de Figuras

# Lista de Tabelas

# Introdução

## 1.1. Tema

Criação de plug-ins, utilizando a linguagem de desenvolvimento Java. O projeto utiliza o framework Eclipse RCP, conceitos de "Padrões de Projeto" e arquitetura de software.

## 1.2. Delimitação

Neste projeto se dará ênfase à comunicação entre plug-ins de forma desacoplada. Entre os plug-ins apresentados, haverá um de edição gráfica, um browser, e um editor de textos.

## 1.3. Justificativa

O Java, conhecido por ser a linguagem de programação mais portável do mundo, não se restringe a nenhum modelo de arquitetura e a nenhuma empresa, mantendo ainda assim, sua velocidade e estabilidade. [...] Nesta linguagem, de acordo com o tipo de aplicação, é possível construir sistemas críticos e precisos, como por exemplo, a sonda Spirit enviada pela Nasa para Marte [10].

De 1998 até hoje a tecnologia evoluiu muito possuindo um dos maiores repositórios de projetos livres do mundo, o java.net. Em 1999 surgiu a plataforma para desenvolvimento e distribuição corporativa batizado de J2EE e a plataforma J2ME para dispositivos móveis, celulares, PDAs e outros aparelhos limitados.

Atualmente Java é uma das linguagens mais usadas e serve para qualquer tipo de aplicação, entre elas: *web*, *desktop*, servidores, *mainframes*, jogos, aplicações móveis, chips de identificação, etc. Dentre os *frameworks* de desenvolvimento em Java, o Eclipse RCP se destaca pela fácil construção de plug-ins.

Antes do surgimento do RCP, companhias e indústrias freqüentemente utilizavam *software* proprietário. Atualmente, as mesmas estão procurando por soluções mais

amplas, que podem se adaptar mais facilmente as diversas situações de projeto. Como um resultado disso, um software para satisfazer tais pré-requisitos precisa não somente ser escalável e robusto, mas também modular por natureza.

No próprio *framework* Eclipse RCP, tudo são plug-ins. A própria (IDE) do mesmo é constituída de um conjunto de plug-ins inter-relacionados. Desta forma, as aplicações construídas sob esta plataforma também são plug-ins. Com este tipo de desenvolvimento, é fácil adicionar ou retirar qualquer parte de um programa, reduzindo o esforço em um projeto. Além disso, devido ao constante e enorme número de projetos open source buscando desenvolvedores, a quantidade de plug-ins aumenta monotonicamente.

Muitas soluções podem ser desenvolvidas, integradas ou até mesmo reformuladas para uma operação de forma mais efetiva e estável utilizando o Eclipse RCP. A tecnologia e as ferramentas existem, pode-se notar que a utilização deste framework agrega um ganho considerável em diversos aspectos como custo, portabilidade e modularidade.

## 1.4. Objetivos

O presente projeto tem como objetivo apresentar um *refactoring* de parte de um *software* já existente. Esta nova versão melhorada usará uma linguagem mais atual, e interface mais amigável ao usuário. Além disso, será feito um estudo sobre as principais características dos plug-ins, enfatizando o Eclipse RCP como *framework* de desenvolvimento, e a utilização de Padrões de Projeto.

## 1.5. Metodologia

Para alcançar os objetivos propostos para este trabalho foram realizadas as seguintes etapas: O primeiro passo foi estudar os conceitos relacionados à plug-ins e compreender aspectos do desenvolvimento utilizando o Eclipse RCP através de levantamentos bibliográficos. Em seguida, foi criado um ambiente de desenvolvimento Java. Após isto, foram decididos que Padrões de Projeto seriam utilizados para facilitar o desenvolvimento proposto, e elaborados documentos parciais com idéias relevantes, e decisões importantes para serem apresentadas no documento final, seguido da

implementação dos plug-ins. Na etapa seguinte foram realizados testes de comunicação entre os plug-ins, e testes de carga sobre diferentes soluções para comunicação inter plug-ins. Por fim foi efetuada a redação conclusiva do projeto final.

## 1.6. Descrição

Neste capítulo é apresentada uma visão geral do trabalho e o contexto no qual está inserido.

No capítulo 2, são abordadas as tecnologias utilizadas durante o desenvolvimento do projeto. Nesta seção mostraremos os Padrões de Projeto, framework de desenvolvimento e a biblioteca gráfica utilizada, além de uma visão geral sobre plug-ins, e uma explicação sobre a POE.

O capítulo 3 apresenta a metodologia utilizada para a implementação de cada plug-in, suas configurações, dependências e diagrama de componentes.

O capítulo 4 mostra como funciona a comunicação entre os plug-ins de forma independente, e um teste de carga de duas possíveis formas de implementação desta troca de mensagens.

O capítulo 5 apresenta a conclusão do trabalho.

# Fundamentação Teórica

## 1.7. Padrões de Projeto

Um Padrão de Projeto (*Design Patterns*) pode ser definido como "a solução para um problema em um contexto". Isto significa que os padrões de projeto nos oferecem soluções prontas para utilizarmos em determinados problemas que podemos enfrentar quando desenvolvemos um *software* orientado a objetos.

Os padrões de projeto são observados através da experiência de um desenvolvedor com projetos de *software*, o que significa que os padrões são descobertos, e não inventados. Como eles fornecem soluções prontas para a solução de problemas, existem diversas vantagens no seu uso, como a reutilização de códigos, modelos e arquiteturas, a padronização de designs e boas práticas, a criação de uma linguagem comum entre desenvolvedores, entre outras.

Podemos classificar os padrões de projeto de várias maneiras, porém o mais comum é classificá-los de acordo com os tipos de problemas que eles resolvem. De acordo com esse critério, os padrões podem ser: de criação, que lidam com o problema da criação de objetos; estruturais, que lidam com os relacionamentos entre objetos; comportamentais, que lidam com a atribuição de responsabilidades aos objetos; e outros.

### 1.7.1. Singleton

O padrão *Singleton* é um padrão de projeto voltado à criação de objetos. O objetivo é garantir uma instância única e acessível de forma global e uniforme para toda classe que implemente este padrão. Neste contexto, este padrão foi usado para tornar a interface de cada módulo da arquitetura única e acessível de forma global.

Outra forma de garantir o acesso global é o uso de uma variável global. Porém a variável global não garante a unicidade. A solução proposta pelo padrão *Singleton* é tornar a própria classe responsável pela sua única instância. A classe que implementa o

padrão *Singleton* garante o acesso à sua instância e ainda intercepta as requisições para criação de novos objetos, garantindo que nenhuma outra instância seja criada.

O padrão *Singleton* garante que um único objeto de uma classe particular irá existir, mas também pode ser útil para objetos grandes e caros que não devem ser instanciados múltiplas vezes.

A vantagem de se criar uma única instância da mesma classe é a fácil manipulação da memória e, além disso, é uma solução que restringe o número de instâncias, fornecendo apenas o necessário ou o suficiente por razões tecnológicas e de negócios. Por exemplo, nós queremos apenas uma única instância de um objeto FILA para ter acesso a um Banco de Dados.

Em Java, uma maneira simples de criar esse padrão seria utilizando os membros *static* e os métodos para executarem as funções do *Singleton*. Para esse elementos, não precisa haver instâncias. A classe pode ser final com um construtor *private*, prevenindo que nenhuma instância da classe seja criada.

Infelizmente, essa abordagem possui algumas desvantagens:

- Às vezes se julga necessário obter informação em tempo de execução para preparar o objeto *singleton* para ser utilizado. Por exemplo, uma aplicação deve garantir que o objeto *singleton* está propriamente inicializado antes de ser utilizado.

- Métodos que são static não podem ser utilizados para implementar interfaces, perdendo assim essas vantagens da linguagem Java. Todas as referências ao objeto *singleton* devem ser atribuídas com o nome da classe.

Numa solução proposta pelos autores do livro *Gang-of-Four´s Design Patterns* [1], livro tradicional de padrões de projeto, existe a proposta que minimiza os dois problemas indicados acima criando uma proteção estática ao redor de uma referência ao construtor da classe. Veja o exemplo abaixo:

```
/**
 * Classe utilitária na qual pode possuir apenas uma instância na máquina virtual.
 *
 * Utilize Singleton instance() para acessar a instancia.
 */
public class Singleton {
```

```
        /**
         * O construtor pode ser privado
         * para prevenir que a classe seja
         * instanciada, mas isso também
         * torna impossível instanciar subclasses
         * de Singleton.
         */
        protected Singleton(){
                //...
        }

        /**
         *  Manipulador para a única instância do Singleton
         */
        static private Singleton _instance = null;

        /**
         *
         * @return A única instância dessa classe
         */
        static private Singleton instance(){
                if (null == _instance){
                        _instance = new Singleton();
                }
                return _instance;
        }

        //.. métodos adicionais omitidos

}
```

A listagem deste trecho de código nos mostra que para uma classe ser um *singleton*, deve-se garantir que haverá apenas uma instância na aplicação e que se deve fornecer um ponto de acesso à mesma. Neste sentido, é de conhecimento comum que, para criar uma instância de uma classe, devemos chamar o seu construtor. Assim, para resolvermos o problema, devemos restringir o acesso ao construtor, tornando-o um método privado. Em seguida, deve-se utilizar um método público que faça o controle da instanciação, de modo que ela só possa ser feita uma vez. A figura 2.1 mostra o diagrama de classes de um *singleton*:

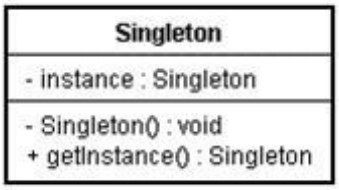


Figura 2.1: Diagrama de Classes de um *Singleton*.

O diagrama de classes da Figura 2.1 mostra apenas o necessário para a instanciação do *singleton*. Obviamente, a classe deve conter outros atributos e métodos. A Figura 2 mostra um diagrama de seqüências que representa o processo de criação de um *singleton*. Quando um objeto chama o *singleton* pela primeira vez, é realizada a instanciação da classe. Os demais acessos irão utilizar a instância já criada.

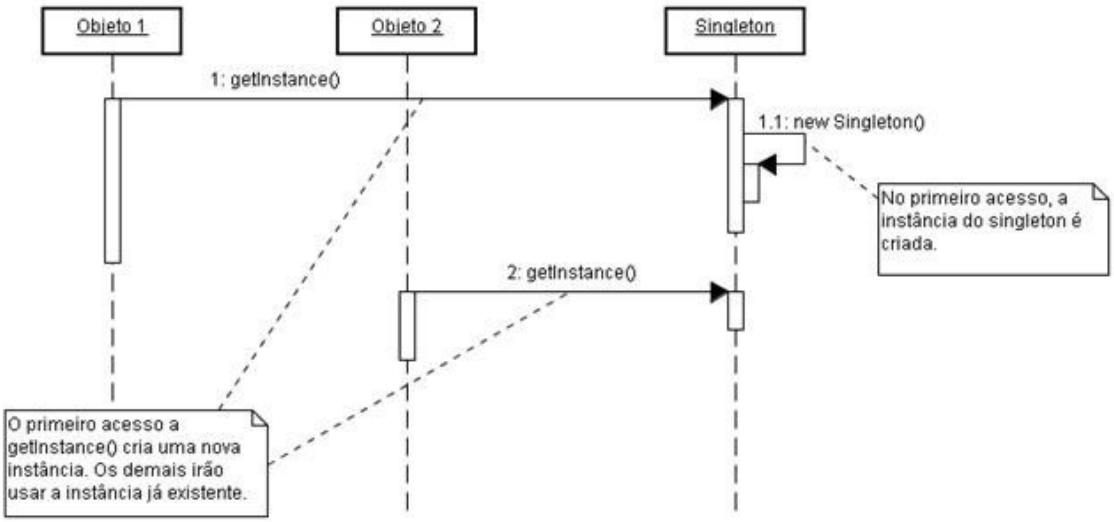


Figura 2.2: Diagrama de Seqüências de um *Singleton*.

Apesar de o *singleton* ser um dos padrões mais simples, uma situação em que realmente é necessário que exista apenas uma instância de uma classe é difícil e o seu mal uso pode levar a muitos problemas. O uso abusivo de *singletons* levam a soluções nas quais as dependência entre objetos é muito forte e a testabilidade é fraca. Adicionalmente, os *singletons* são difíceis de escalar, pois é difícil garantir uma única instância de um *singleton* em um cluster, onde existem várias JVMs. Outro problema é que os *singletons* dificultam o *hot redeploy* por permanecerem em *cache*, bloqueando mudanças de configuração. Por esses e outros problemas, deve-se ter cuidado com o uso abusivo de *singletons*.

### 1.7.2. Mediador

Usualmente, um programa é feito de um grande número de classes, de modo que a lógica de negócios é distribuída por essas classes. Entretanto, uma vez que o número de classes cresce significativamente, especialmente durante a manutenção e o *refactoring*, o problema de comunicação entre as classes pode ficar cada vez mais

complexo. Isso dificulta a leitura do código, e o debug, tornando o programa difícil de ser mantido. Além disso, fica muito difícil fazer alguma alteração no programa, por que uma simples mudança nos leva a alterar várias classes, uma vez que o programa possui alto nível de acoplamento.

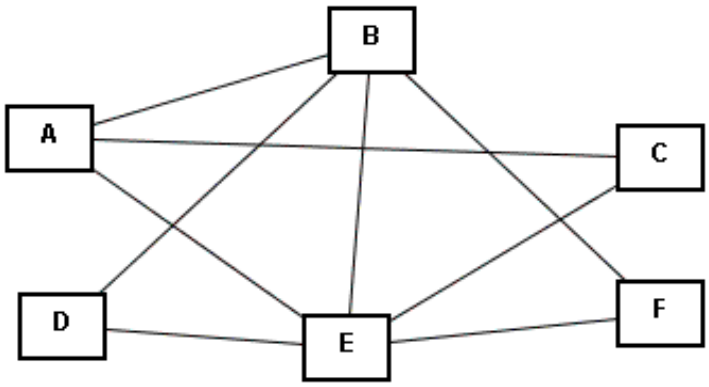


Figura 2.3: Interação entre objetos sem a utilização do mediador (extraída de [8])

Com a utilização do padrão de projeto *Mediador*, a comunicação entre objetos e feita através de um objeto mediador. Desta forma, os objetos não se comunicarão diretamente, e ao invés disso, se comunicarão com o mediador. [13]

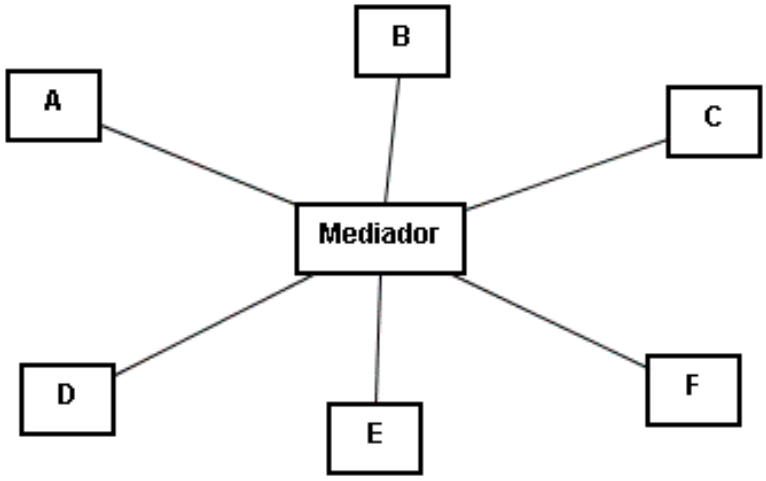


Figura 2.4: Interação entre objetos utilizando o Mediador (extraída de [8])

Este procedimento reduz drasticamente a dependência entre as classes, e o acoplamento entre os objetos. O mediador é considerado um padrão de projeto comportamental, por alterar o comportamento do programa em questão. Podemos fazer uma analogia a um sistema não computacional, conforme a figura 2.5.

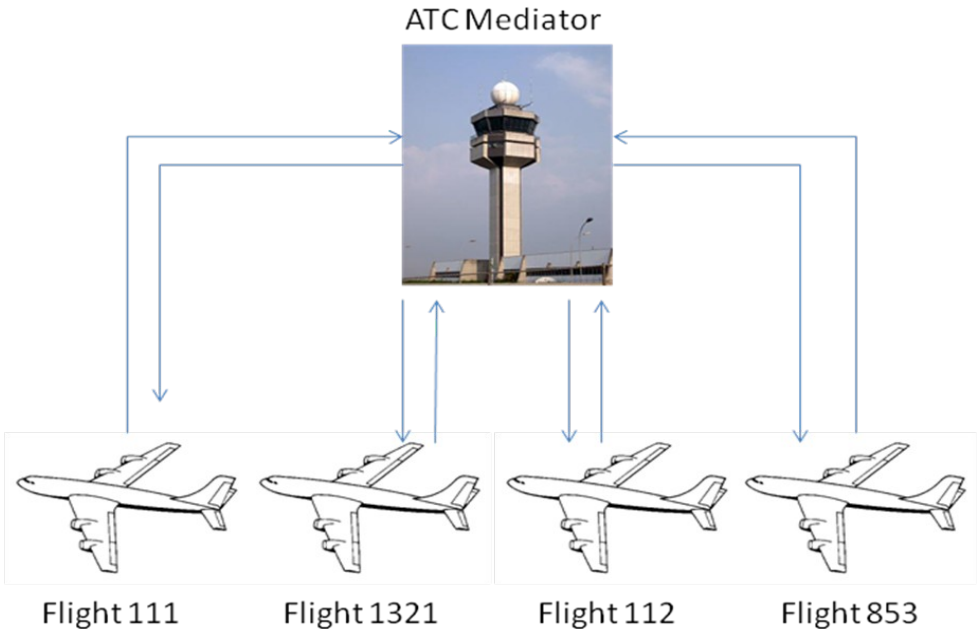


Figura 2.5: Analogia não computacional ao Padrão de Projeto Mediador

Um exemplo de aplicação do padrão Mediator é uma interface gráfica de usuário, onde diferentes *widgets* (botões, caixas de seleção, etc.) estão relacionados entre si. Quando um botão é clicado, uma lista de seleção deve ser ativada, por exemplo. Ao invés de colocar o comportamento nos *widgets*, criamos um mediador, que recebe "eventos", e manda outros *widgets* executarem suas ações conforme os eventos ocorrem. Desta forma, podemos usar um mesmo botão em janelas diferentes, pois o comportamento relativo a uma ação específica ("clicado", "selecionado", "movimentado") está implementada fora dele. [13]

Este padrão deve ser utilizado quando:

- Um conjunto de objetos se comunica de uma forma bem determinada, porém complexa;
- A reutilização de uma classe é difícil, pois ela tem associação com muitas outras;

- Um comportamento que é distribuído entre várias classes deve ser extensível em ter que criar muitas subclasses;
- É necessário desacoplamento total de duas classes, como por exemplo, no desenvolvimento de plug-ins.

Vantagens:

- Limita extensão por herança, de forma que para estender ou alterar o comportamento, basta criar uma subclasse do mediador;
- Desacopla objetos, de forma a promover naturalmente a reutilização de código pronto;
- Simplifica o protocolo, de forma que a relação das classes com um mediador são muito mais simples de manter do que muitas espalhadas, ficando mais clara a forma como os objetos interagem.

Desvantagem:

- A utilização exagerada deste padrão de projeto, pode levar a um sistema monolítico.

### 1.7.3. Model-View-Controller

O objetivo de muitos sistemas de computadores é a de recuperar dados a partir de um banco de dados e exibi-los para o usuário. Após o usuário alterar os dados, o sistema armazena as atualizações em o banco de dados. Como o fluxo de informação é fundamental entre o banco de dados e a interface com usuário, é natural pensar em vincular essas duas partes, para reduzir a quantidade de codificação e melhorar o desempenho da aplicação. No entanto, esta abordagem tem vários problemas significativos. Um deles é que a interface com o usuário tende a mudar com muito mais freqüência do que a base de dados. Outro problema em acoplar os dados com a interface do usuário, é que as aplicações tendem a incorporar lógica de negócios que vão muito além da transmissão de dados.

Desta forma, percebe-se a necessidade de modularizar o desenvolvimento, de forma a poder facilmente fazer modificações em cada parte do sistema individualmente. A proposta de solução deste problema é materializada através do padrão Model-View-Controller (MVC [12]).

Na arquitetura MVC o modelo representa os dados da aplicação e as regras do negócio que governam o acesso e a modificação dos dados. O modelo mantém o estado persistente do negócio e fornece ao controlador a capacidade de acessar as funcionalidades da aplicação encapsuladas pelo próprio modelo.

Um componente de visualização renderiza o conteúdo de uma parte particular do modelo e encaminha para o controlador as ações do usuário; acessa também os dados do modelo via controlador e define como esses dados devem ser apresentados.

Um controlador define o comportamento da aplicação, é ele que interpreta as ações do usuário e as mapeia para chamadas do modelo. Em um cliente de aplicações Web essas ações do usuário poderiam ser cliques de botões ou seleções de menus. As ações realizadas pelo modelo incluem ativar processos de negócio ou alterar o estado do modelo. Com base na ação do usuário e no resultado do processamento do modelo, o controlador seleciona uma visualização a ser exibida como parte da resposta a solicitação do usuário. Há normalmente um controlador para cada conjunto de funcionalidades relacionadas.

Temos a figura 2.6 que mostra o relacionamento entre as 3 camadas do modelo MVC.

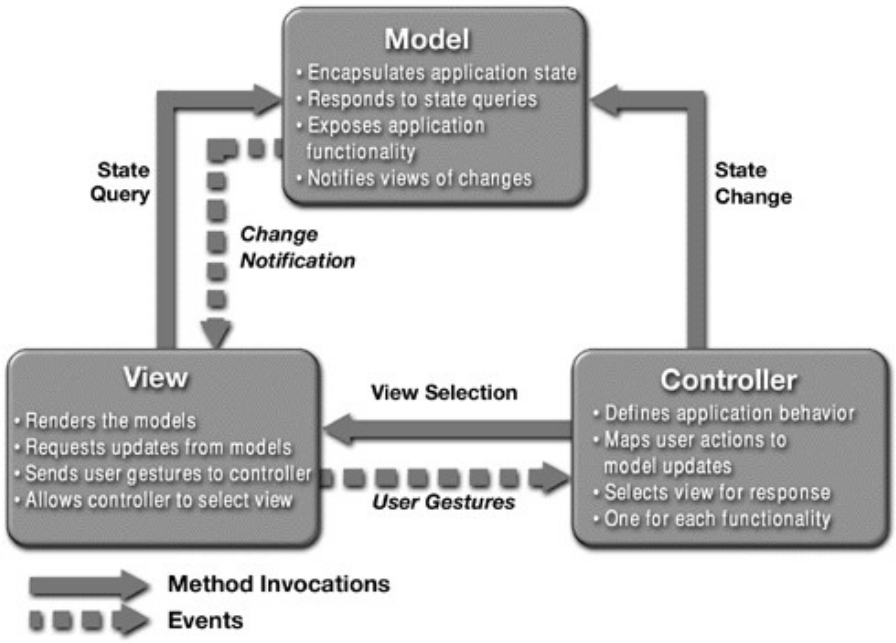


Figura 2.6: Arquitetura MVC. (extraída de [11])

**Model**

O *model* é um objeto de persistência de dados que pode representar um arquivo, uma atividade ou até mesmo uma tabela de um banco de dados. Existem, no entanto, algumas ressalvas que devem ser feitas acerca do *model*:

- Deve conter todas as informações importantes e que podem ser editadas pelo usuário. Se alguma coisa for ser editada via editor gráfico, esta edição precisa ser avisada ao *model*.
- Não deve conhecer nada sobre a *view* ou o *controller*. O *model* tem que ser totalmente desacoplado dos demais componentes. O framework se encarregará de fazer as ligações necessárias entre esses componentes.
- Deve implementar um mecanismo de notificação quando ocorrem mudanças em suas propriedades

**View**

A *view* é a representação visual do *model* na aplicação, isto é, efetua a apresentação dos dados através de sua interface com o usuário. Deve possuir as seguintes características:

- Assim como o *model*, esse componente não deve conhecer nada sobre os demais componentes.
- Não deve conter nenhuma informação relevante que já esteja representada no *model*.

**Controller**

O *controller* é o componente principal do framework. Ele é responsável por monitorar as mudanças do *model* e modificar a *view* correlata. Sob esta ótica, exige-se do *controller* que ele conheça tanto a *view* como o *model*.

É importante ressaltar, que tanto a *view* quanto o *controller*, dependem do *model*. Entretanto, o *model* não depende nem da *view*, nem do *controller*. Este é um dos principais benefícios da separação em camadas: permitir que o modelo seja construído e testado, independentemente da forma em que ocorrerá a interação com o usuário.

Vantagens:

- Suporta múltiplas interfaces com o usuário. Como a *view* é separada do *model*, e não há dependência direta do *model* em relação à *view*, a interface com o usuário pode ter várias *views*, mostrando os mesmos dados, ao mesmo tempo.
- Acomoda mudanças. A interface com o usuário costuma se alterar mais rapidamente que a lógica de negócios, ou o modelo de dados. Desta forma, a cor, fonte, ou tamanho da exibição, não fazem diferença para as outras camadas do sistema, podendo até mesmo, haver uma personalização da interface, de acordo com o tipo do usuário.
- Desenvolvimento em paralelo. Devido às camadas serem isoladas, é possível o desenvolvimento simultâneo de cada uma delas.

Desvantagens:

- Complexidade. Este padrão de projeto introduz novas camadas e níveis de abstração, aumentando ligeiramente a complexidade da solução. Devido a isso, requer profissionais mais experientes para que o desenvolvimento continue rápido. Além disso, aumenta a dificuldade do debug do software.
- Aplicações pequenas. Como é necessário mais tempo para modelar aplicações usando este padrão, o custo pode não compensar para aplicações pequenas.

### 1.7.4. Factory Method

O principal objetivo desse padrão é definir uma interface responsável pela criação de objetos, mas que entretanto permite delegar às suas subclasses a decisão sobre qual classe instanciar [1]. A idéia central é que ao invés do cliente chamar o operador *new* quando necessitar de uma instanciação, ele chama um método abstrato (*Factory Method*) especificado em alguma classe. A subclasse concreta torna-se responsável decidir que tipo exato de objeto vai criar e retornar.

Esse tipo de abordagem permite que mudemos a subclasse concreta que cria o objeto sem que o cliente saiba, e também permite estender essa funcionalidade através

dessa construção de outras subclasses sem afetar os clientes. O mais difícil deste padrão de projeto é identificar quando usá-lo, por isso vamos listar algumas situações conhecidas nas quais se devem usar esse padrão quando:

- Uma classe (o criador) não pode antecipar a classe dos objetos que deve criar;
- Uma classe quer que suas subclasses especifiquem os objetos criados;
- Classes delegam responsabilidade para uma subclasse de apoio e necessita-se localizar num ponto único o conhecimento de qual subclasse está sendo usada.

Os participantes desse padrão são:

- *Product*
  - Define a interface dos objetos criados pelo F*actory Method*
- *ConcreteProduct*
  - Implementa a interface *Product*
- *Creator*
  - Declara o *Factory Method* que retorna um objeto do tipo *Product*
  - Às vezes pode definir uma implementação default de um *Factory Method* que retorna um objeto default *ConcreteProduct*
  - Pode chamar o *Factory Method* para criar um objeto produto.
- *ConcreteCreator*
  - faz *override* do *Factory Method* para retornar uma instancia de um *ConcreteProduct*

A figura 2.7 ilustra a interligação entre os participantes:

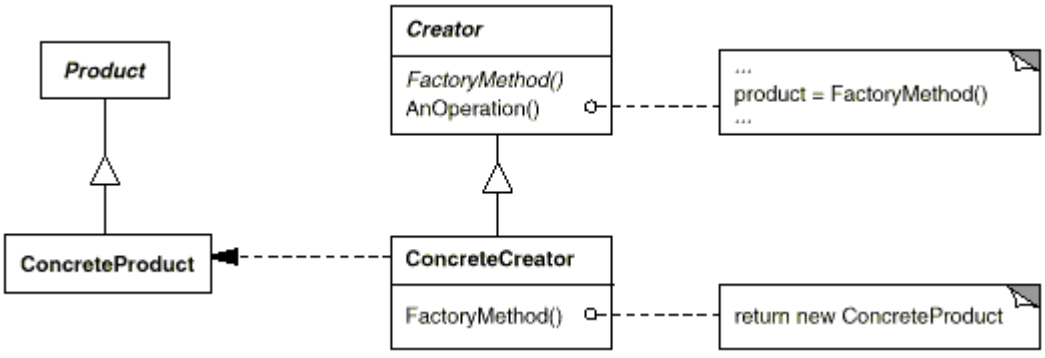


Figura 2.7: Estrutura *Factory Method*( extraída de [1])

### 1.7.5. Abstract Factory

Às vezes nos deparamos com um sistema que deve ser independente de como seus produtos são criados, compostos, e representados, ou ainda, deve ser configurado com uma entre varias famílias de produtos. Possivelmente estamos enfrentando um caso, no qual o padrão *Abstract Factory* [1] deve ser implementado, pois esse permite a criação de famílias de objetos relacionados ou dependentes, através de uma única *interface* e sem que a classe concreta seja especificada. Outros casos em que podemos pensar em utilizar este padrão de projeto é quando o sistema possui uma família de objetos projetada para uso conjunto, ou quando queremos fornecer uma biblioteca de classes de produto e revelar apenas sua interface e não sua implementação. O *Abstract Factory*, assim como o *Singleton*, é um padrão de criação.

A implementação desse padrão traz benefícios para a aplicação, como isolamento das classes concretas e dos clientes das classes de implementação, facilita a troca da família de produtos, promove consistência entre estes mesmos produtos. Por outro lado, a escolha deste padrão, torna mais complexa a adição de novos tipos de aplicações [1].

Os participantes desse padrão são:

- *AbstractFactory*
    - Declara uma interface para operações que criam objetos como produtos abstratos
- *ConcreteFactory*
    - Implementa as operações para criar objetos para produtos concretos
- *AbstractProduct*
    - Declara uma interface para objetos de um tipo
- *ConcreteProduct*
    - Define um objeto a ser criado pela *ConcreteFactory*
    - Implementa a interface de *AbstractProduct*
- *Client*

- Usa apenas interfaces declaradas pelas classes *AbstractFactory* e *AbstractProduct*

Para esclarecer as ainda mais segue o diagrama de classes do Padrão de Projeto.

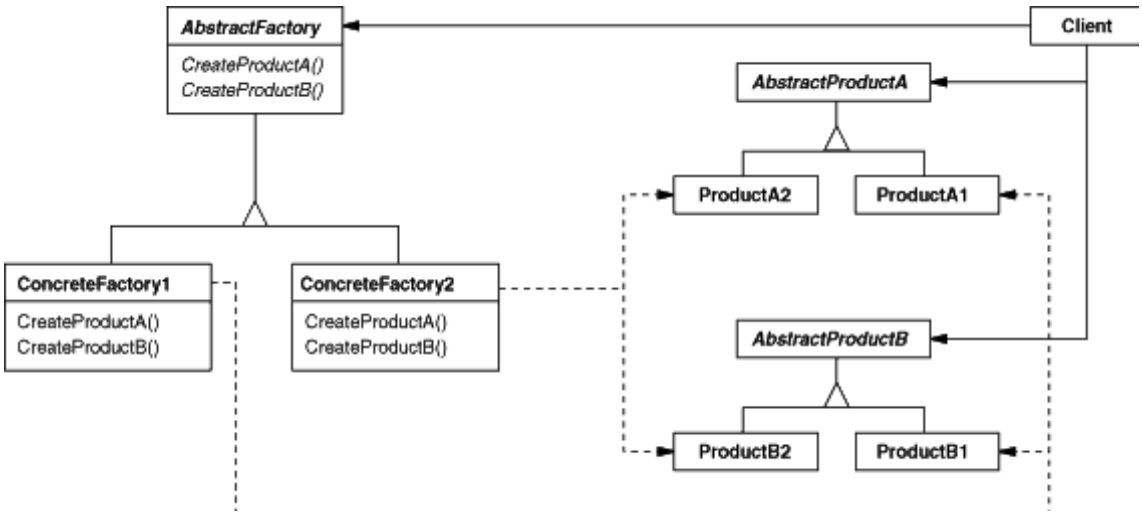


Figura 2.8: Diagrama de Classes (extraída de [1])

## 1.8. Framework de Desenvolvimento

*Framework* são geradores de aplicações ou soluções que estão diretamente ligados a um domínio especifico, isto é, a uma família de problemas relacionados. E estão presentes em diferentes contextos, como, desenho artístico, composição musical, compiladores para diferentes linguagens de programação, sistemas de suporte a decisão e etc. Restringindo o escopo para a área de desenvolvimento de *software*, um framework consiste em um software utilizado por desenvolvedores para implementar a estrutura padrão de uma aplicação para um determinado sistema operacional.

Há uma grande confusão entre os termos *framework* e biblioteca, esclarecendo a diferença, temos que "[...] uma biblioteca consiste de código, seja sob a forma autônoma de funções, principalmente no código nativo da execução, de classes ou de ambas. Já um *framework* é um conjunto de bibliotecas e contém como por exemplo no .NET, o 'root' Mscorlib.dll, que define a CLR (Common Language Runtime) e, por padrão, é incluída em todos os projetos relacionados, e adicionalmente em bibliotecas de conjuntos, como o Windows Forms, Web Forms, Ling etc. *Frameworks* são instalados, o que não obriga a utilização de todas as bibliotecas incluídas. Podemos criar um *framework* por escrito, separando os códigos em bibliotecas apropriadas. Em linhas

gerais, definimos um *framework* como uma coleção de bibliotecas que representa o contexto de que foram criadas para serem utilizadas na aplicação. ”[3]

Um *framework* tem o objetivo de facilitar o desenvolvimento permitindo que os designers e programadores passem mais tempo preocupados com os requisitos do software, por exemplo, ao invés de ficar lidando com detalhes mais “baixo-nivel” do funcionamento do sistema [3] . Por isso, o uso de framework torna-se tão atraente aos desenvolvedores, uma vez, que permite que alguns problemas, tratamentos e funcionamentos sejam abstraídos para o usuário.

Sendo assim, os *frameworks* facilitam o trabalho com tecnologias complexas, agrupa vários objetos discretos e componentes em algo muito mais utilizável, e as vezes até nos força a implementar códigos mais consistentes; com menos *bugs*. Além disto, as aplicações tendem não só a ser mais flexíveis, mas também qualquer pessoa pode testar e “debuggar” facilmente o código.

## 1.9. Eclipse

O Eclipse é um projeto open source, iniciado pela IBM e mais tarde transferido à Eclipse Foundation, que visa criar uma plataforma de desenvolvimento composta de diversas frameworks, ferramentas com o objetivo de construir, implantar e manter softwares durante todo o seu ciclo de vida. O projeto Eclipse é totalmente escrito em Java.

O Eclipse é muito conhecido no meio acadêmico e profissional por sua vocação como IDE para a linguagem Java, propósito para o qual foi inicialmente projetado. Neste mérito, ele constitui uma ferramenta extremamente madura, com recursos avançados de *debug*, *deploy*, refatoração , controle de versões e outras características que auxiliam em muito o trabalho do desenvolvedor Java.

Além disto, é possível estender a funcionalidade provida pelo Eclipse através da criação de plug-ins utilizando o próprio Eclipse. A arquitetura do Eclipse permite inclusive que se construam aplicações com propósitos completamente distintos do propósito inicial de IDE. O processo de criação de plug-ins e aplicações a partir do Eclipse serão melhor detalhados mais adiante.

### 1.9.1. Arquitetura

Para entender o funcionamento do Eclipse e o processo de criação de plug-ins e aplicações a partir do Eclipse, faz-se necessário primeiro estudar a sua arquitetura. A arquitetura do Eclipse é totalmente composta de plug-ins. O seu funcionamento básico está baseado na composição das diversas funcionalidades contribuídas por eles e pela possibilidade de cada um abrir à contribuição do outro. Tudo no Eclipse é um plug-in ou está baseado num. Isso é extremamente importante no sentido de que existe uma uniformidade no tratamento dos componentes do Eclipse, não existem componentes que não sejam plug-ins e sejam tratados de maneira diferente.

No Eclipse, existe um conjunto base de plug-ins, denominado *Platform*, que fornecem os serviços básicos da plataforma aos outros. Esse é o conjunto mínimo necessário para a criação de aplicações baseadas no Eclipse. O *Platform* é composto de dois subconjuntos de extensões: *Core* e UI. O *Core* é a parte responsável por serviços não relacionados à interface com o usuário e o UI pelos relacionados à interface com o usuário.

O Core é composto ainda de dois subconjuntos: Runtime e Workspace. O Runtime provê as funcionalidades básicas de *runtime*, ou seja, de chamada e instanciação de plug-ins, suas classes e recursos. É tarefa do Runtime também descobrir quais plug-ins estão disponíveis e verificar se todas as dependências foram resolvidas quando do início da aplicação. O Workspace é o responsável pelas funções relacionadas ao uso de projetos e arquivos.

O UI, por sua vez, é composto dos seguintes subconjuntos: SWT, JFace e Workbench. O SWT (Standard Widget Toolkit) é um Widget Toolkit criado pela IBM para criação de aplicações Desktop em Java, provendo as abstrações básicas necessárias, como janelas, botões, tabelas, menus, *lists*, *checkboxs* e outros componentes gráficos encontrados em abundância nas interfaces gráficas das aplicações dos dias de hoje. O SWT também faz todo o tratamento de eventos relacionados à interação do usuário com a aplicação através de um sistema de eventos. Dessa forma, o SWT lida diretamente com interrupções geradas pelo hardware e com o sistema operacional hospedeiro e suas *system calls*. Por esse motivo, o SWT é dependente de plataforma, mas existe uma implementação para cada plataforma popular atualmente.

O SWT foi criado durante o projeto inicial do Eclipse pela IBM, com a decisão de não adotar o Swing, biblioteca padrão do Java para criação de interfaces gráficas. O Swing provê uma camada de abstração para as interfaces gráficas criadas que as tornam

totalmente independentes da plataforma em que estão rodando, bastando que exista para ela uma JRE (*Java Runtime Environment*). Essa independência da plataforma vem ao custo de um desempenho lento e uma interface com componentes muito distintos dos componentes nativos do sistema operacional, de modo que a aplicação ficava com o visual estranho quando colocada próxima às demais aplicações do sistema. Por esse motivo a IBM decidiu não utilizá-lo e criar, do zero, uma nova biblioteca para criação e interfaces gráficas. O SWT foi projetado para criar interfaces gráficas com componentes nativos do sistema operacional hospedeiro e com desempenho mais aceitável.

O JFace é uma biblioteca que utiliza o SWT para criação de abstrações gráficas de nível mais alto, como wizards, viewers, diálogos, toolbars, entre outros. Assim como SWT, o JFace, é um conjunto de plug-ins, mas pode ser utilizado fora da plataforma Eclipse, para construção de aplicações Java com interface gráfica.

O Workbench é um conjunto de plug-ins que se apóia no JFace para prover abstrações de nível ainda mais alto, como *views*, editores e perspectivas, abstrações diretamente relacionadas com o Eclipse e sua interface gráfica.

A figura 2.9 abaixo retrata a arquitetura do Eclipse:

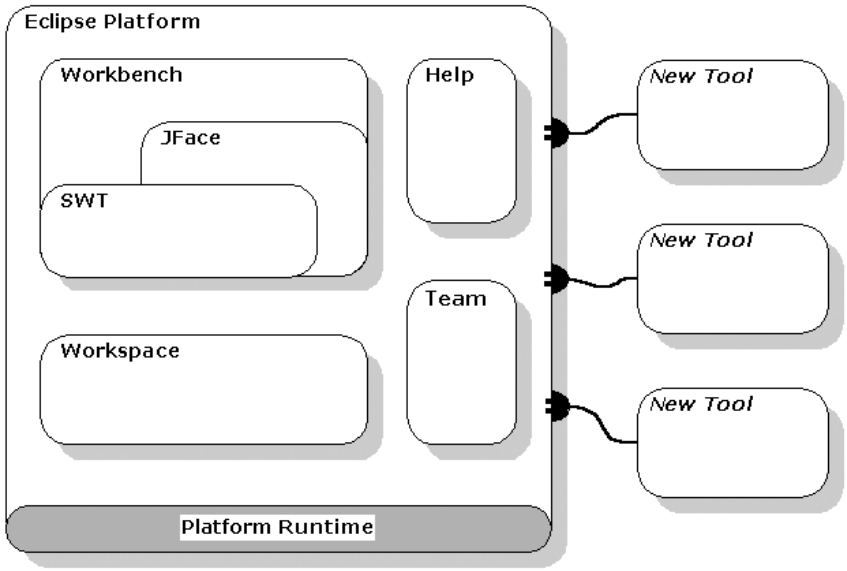


Figura 2.9: Arquitetura Eclipse

Os demais plug-ins podem se apoiar nos plug-ins básicos do Platform ou em outros plug-ins. Na figura 2.10, vemos como o conjunto de plug-ins que constituem as

ferramentas para desenvolvimento Java e de plug-ins se apóiam na estrutura básica provida pelo Platform:


Figura 2.10: Plug-In do Eclipse do Ambiente de Desenvolvimento Java

### 1.9.2. Eclipse RCP

O Eclipse RCP é um conjunto mínimo de plug-ins necessários para a criação de aplicações de propósitos gerais e baseadas na arquitetura do Eclipse. Historicamente, o Eclipse RCP surgiu quando, ainda na versão 2.1, os desenvolvedores do projeto original do Eclipse identificaram que muitas pessoas estavam construindo aplicações *Rich Client* utilizando o Eclipse como base e o mecanismo de plug-ins para estender suas funcionalidades. Muitas dessas aplicações não possuíam relação alguma com o desenvolvimento de softwares, sendo aplicações de uso mais gerais. Foi proposta então uma mudança radical na arquitetura da plataforma, de modo a facilitar o trabalho dessas pessoas que queriam desenvolver aplicações baseadas no Eclipse.

Essa mudança deveria extrair um conjunto básico de plug-ins que proveriam todos os serviços da plataforma. A intenção era que os desenvolvedores de aplicações *Rich Client* não precisassem carregar desnecessariamente plug-ins ligados apenas a IDE. Na época do Eclipse 2.1, os plug-ins estavam todos muito acoplados e não existia esse conjunto básico. Foi então necessário um colossal esforço para separar os interesses do sistema em subsistemas e refinar a estrutura de plug-ins. Como resultado desse esforço, o Eclipse 3.0 apresentou uma melhora significativa na arquitetura, facilitando inclusive o desenvolvimento de plug-ins para a IDE pela comunidade.

O conjunto mínimo de plug-ins foi denominado de Eclipse RCP. O desenvolvimento de uma aplicação RCP tornou-se apenas uma questão de implementar plug-ins Eclipse que se apóiam sobre os providos pelo Eclipse RCP e provêem as funcionalidades requeridas pela aplicação em questão. Os plug-ins desenvolvidos para a

IDE também podem ser integrados a qualquer aplicação RCP através do mesmo mecanismo básico utilizado para integrar plug-ins comuns a IDE. Isso abre um conjunto infinito de possibilidades de utilizar na sua aplicação frameworks e ferramentas criadas pela comunidade para suportar o projeto principal do Eclipse.

Pela nova arquitetura, a própria IDE Eclipse se tornou um exemplo de aplicação RCP. Conceitualmente, não existe mais qualquer diferença entre uma aplicação RCP de propósito geral e a IDE Eclipse. Ambas são aplicações RCP, compostas, em última instância, de plug-ins, em que todos possuem uma estrutura conhecida.

Por ser baseada na arquitetura do Eclipse, uma aplicação RCP usufrui naturalmente das seguintes vantagens:

- Estrutura modular baseada em componentes – as aplicações RCP são construídas pela composição de componentes conhecidos como plug-ins. Os plug-ins podem ser versionados e compartilhados por várias aplicações. Múltiplas versões de um mesmo plug-in podem ser instaladas lado a lado e aplicações podem ser configuradas para usar a exata versão de que precisam. Isso provê uma estrutura modular e facilita a manutenção e evolução das aplicações RCP.
- Infra-estrutura pré-existente – toda a infra-estrutura necessária para a construção de *Rich Clients* é provida pela plataforma RCP. O desenvolvedor não precisa se preocupar em implementar mecanismos de UI, ajuda, carregamento automático de componentes quando eles precisarem ser utilizados, atualização pela rede, tratamento de erros, entre outros, podendo-se ater as particularidades do software que está sendo desenvolvido, simplificando o processo e economizando tempo e reduzindo custos.

- Interface de usuário nativa – ao contrário de muitas aplicações *desktop* Java, a plataforma RCP, através do uso do SWT, possibilita a criação de aplicações que possuam um visual nativo do sistema operacional onde o sistema está instalado, melhorando a experiência do usuário final da aplicação.

- Portabilidade – como toda aplicação Java, as aplicações criadas com a plataforma RCP são portáveis. O mesmo código da aplicação pode ser utilizado para construir versões específicas para cada sistema operacional, para cada

ambiente onde se quer instalar o sistema. Isso permite o desenvolvimento de aplicações portáveis com muito menos esforço.

- Operação desconectada – as aplicações RCP podem rodar *standalone*, sem necessidade de uma conexão com rede ou internet.

- Ferramenta de desenvolvimento – o Eclipse provê o PDE (Plug-in Development Environment), um conjunto de plug-ins que provêem uma poderosa e eficiente ferramenta para desenvolvimento de plug-ins e aplicações RCP.

- Sistema inteligente de instalação e atualizações – a estrutura componentizada de aplicação RCP permite que se instale ou atualize facilmente a aplicação pelo simples *deploy* e troca de componentes. Os componentes podem ser instalados através da *update* sites, Java Start, simples cópia de arquivos ou sistemas complexos de gerenciamento de componentes.
- Grande quantidade de componentes prontos – qualquer componente do projeto Eclipse ou de plug-ins para o Eclipse podem ser utilizados dentro de uma aplicação RCP. Com isso, tem-se uma variedade de componentes disponíveis, como editores de texto, *consoles*, *frameworks* para edição gráfica, de modelagem, ferramentas para geração de relatórios, manipulação de dados, dentre outras.

- Robustez – caso algum componente esteja causando problemas no sistema, a plataforma RCP garante a possibilidade de desabilitá-lo ou substituí-lo em tempo de execução, sem precisar parar a aplicação. Além disso, o Eclipse possui a filosofia de *lazy loading*, ou seja, um componente só será carregado quando ele realmente for ser utilizado, economizando recursos e evitando erros crônicos causados por um componente que nem estava sendo utilizado.

- Extensibilidade – os plug-ins de uma aplicação RCP podem expor um ponto de extensão, ou seja, uma funcionalidade que pode ser contribuída por plug-ins de terceiros, sem ser necessário que se tenha conhecimento da exata implementação

dos plug-ins contribuintes. Com isso se possibilita a criação de uma comunidade em torno da aplicação.

- Suporte a internacionalização – o PDE possui um mecanismo prático para extrair strings e permitir a criação de plug-ins internacionalizados. Além disso, a aplicação pode detectar automaticamente o idioma do sistema operacional da máquina hospedeira e carregar a configuração adequada.

### 1.9.3. Subprojetos

Como foi dito, o projeto Eclipse não cuida apenas do desenvolvimento de uma IDE Java poderosa e flexível. Existem infinitos outros subprojetos que estendem a funcionalidade básica de IDE para criar uma completa ferramenta para se construir, implantar e manter softwares durante todo o seu ciclo de vida.

Dentre os subprojetos Eclipse, cabe destacar os seguintes:

- BIRT (Business Intelligence and Reporting Tools), conjunto de plug-ins relacionados à área de Business Intelligence e à criação de relatórios.
- CDT (C/C++ Development Tools), conjunto de plug-ins relacionados ao desenvolvimento de sistemas na linguagem de programação C/C++.
- DTP (Data Tools Platform), conjunto de plug-ins relacionados a bancos de dados.
- EMF (Eclipse Model Framework), conjunto de plug-ins para criação de modelos.
- GEF (Graphical Editing Framework), conjunto de plug-ins relacionados à criação de editores gráficos.
- JDT (Java Development Tools), conjunto de plug-ins relacionados ao desenvolvimento de sistemas na linguagem de programação Java (IDE Java).
- TPTP (Test & Performance Tools Platform), conjunto de plug-ins relacionados à criação de ferramentas de teste e desempenho.

## 1.10. Plug-ins

### 1.10.1. Estrutura

Existem alguns conceitos básicos sobre o Eclipse, seus plug-ins e a plataforma RCP que devem ser dominados:

- Workbench – abstração ligada à interface gráfica propriamente dita da plataforma. É composta de abstrações como *views*, *editors* e *perspectives*. É importante ressaltar que a *workbench* não é a janela do Eclipse (ou aplicação RCP). De fato, existe apenas uma workbench para cada Eclipse aberto e ela pode estar ligada a uma ou mais *workbench windows*, que são essas sim as janelas do Eclipse.

- Workspace – abstração ligada aos projetos do usuário, os arquivos e recursos contidos nesse projeto e a área onde esses projetos são armazenados fisicamente. Cada *workbench* opera sobre um determinado Eclipse e apenas uma *workbench* pode operar sobre um dado *workspace* por vez. Todos os arquivos e projetos criados são armazenados dentro da pasta correspondente ao *workspace* no sistema de arquivos local. O Eclipse suporta também arquivos e projetos referenciados por links simbólicos, permitindo assim uma abstração completa da localização dos recursos referenciados no *workspace*.

- View – tipo de janela exibida dentro da *workbench window*. Uma *view* tem a finalidade primária de visualização de dados. Uma *view* pode ser livremente redimensionada e reposicionada dentro da *workbench window*, empilhada com outras *views* e até desconectada da janela principal, originando outra janela. Em termos de API, É representada pela interface org.eclipse.ui.IViewPart e pela sua implementação default, a classe org.eclipse.ui.ViewPart.

- Editor – tipo de janela exibida dentro da *workbench window*. Um editor tem a finalidade primária de permitir a edição de um determinado recurso através de algum tipo de manipulação por parte do usuário. Existem editores de texto, imagem, diagramas, entre outros. Um editor está necessariamente ligado a um org.eclipse.ui.IEditorInput, interface que representa a entrada (*input*) de um editor. Um editor pode ser reposicionado dentro de uma área conhecida como *Editor Area*, mas não pode ser desconectado. Também só pode ser empilhado com outros *editors*. Em termos de API, é representado pela interface org.eclipse.ui.IEditorPart e pela sua implementação default, a classe org.eclipse.ui.EditorPart. Está associado ao paradigma "abrir-modificar-salvar".

- Perspective – conjunto de janelas relacionadas a um determinado objetivo ou finalidade. Apenas uma perspective é visualizada por vez no Eclipse. Apesar de se poder abrir e fechar livremente *editors* e *views* dentro de uma *perspective*, que define o conjunto inicial desses elementos que será mostrado na tela.

- Plug-in – unidade básica da arquitetura do Eclipse. Um componente é assim chamado na computação quando se "pluga" ao conjunto principal para prover funcionalidades que antes o conjunto principal não possuía. No Eclipse, um plug-in é um projeto Java que contribui com funcionalidade para a plataforma Eclipse e para outros plug-ins, através do mecanismo de extensions e extension points.

- Feature – grupo de plug-ins que estão funcionalmente relacionados, constituindo um subsistema e devendo ser instalados e atualizados simultaneamente. É através da definição de *features* que se pode utilizar o mecanismo nativo de *install* e *update* do Eclipse em aplicações RCP. Ao se definir uma *feature*, podem-se impor restrições de instalação ou

atualização, como necessidade de se aceitar um termo de compromisso, necessidade de instalar antes outras *features*, entre outras.

- Fragment – fragmento de plug-in, que pode ser usado para estender a funcionalidade provida pelo plug-in. Ele funciona basicamente como um *patch* que é carregado em tempo de execução, alterando uma determinada versão do plug-in original e possivelmente provendo novas funcionalidades. *Fragments* são usados principalmente para criar versões especiais de um determinado plug-in para um determinado ambiente de execução. Também podem ser usados para fornecer novos recursos, como arquivos de imagens, ou para fornecer versões internacionalizadas de um plug-in.

- Extension Point – ponto em que um plug-in se abre para a contribuição de outros. Um *extension point* é a definição de uma funcionalidade que o plug-in permite que seja implementada por qualquer outro plug-in sem necessidade que ele o conheça previamente. A implementação do *extension point* é a *extension*.

- Extension – implementação do *extension point* de um plug-in. Geralmente consiste de uma classe que implementa uma interface determinada no *extension point*. Em tempo de execução, o *runtime* da plataforma detecta todas as *extensions* para um determinado *extension point* e permite que a classe onde esteja definido o *extension point* instancie os elementos descritos na *extension* sem precisar conhecê-los previamente, bastando saber que todos eles implementam a interface requerida e utilizando-a para chamar os métodos necessários.

- Action – opção que aparece em menu bars, *coolbars* e *context menus*, representa uma ação que pode ser tomada por um usuário. Podem-se restringir os elementos (recursos) sobre os quais uma *action* atua ou mesmo especificar circunstâncias sobre as quais ela ficará desabilitada. A

*action* é representada pelas interfaces org.eclipse.ui.IAction e org.eclipse.ui.IActionDelegate e pode ser adicionada a um menu ou coolbar programaticamente ou através de *extensions* (esse recurso facilita a tarefa de desacoplar os plug-ins, visto que o mecanismo de verificação de *extensions* ocorre dinamicamente e em tempo de execução).

- Coolbar – barra de ferramentas onde podem ser adicionadas *actions*. A *coolbar* pode ser livremente reposicionada dentro da *workbench window*.
- Target – conjunto de plug-ins a partir dos quais a aplicação RCP será construída. Todos os plug-ins que forem necessários para a construção da aplicação RCP devem estar no diretório especificado para o *target*. Podem-se definir configurações mais refinadas para um *target* criando-se uma *target definition*.

- Product – definição do produto final a ser construído. Um *product*, materializado num arquivo **.product*, contém todas as configurações de aplicação RCP, como quais plug-ins serão incluídas na aplicação, qual o nome do executável a ser gerado para a aplicação, ícone desse executável, *splash screen*, *about*, dentre outras. Podem-se criar várias versões (perfis) para uma mesma aplicação RCP pela simples criação de vários *products*. O editor padrão do Product possui um wizard de exportação da aplicação, facilitando o seu deploy, podendo-se ainda exportar-la para diversas plataformas.

### 1.10.2. Desenvolvimento

O processo de criação de plug-ins no Eclipse é bem simples e será descrito adiante. Para facilitar todo o processo, o projeto Eclipse inovou criando uma ferramenta própria para o desenvolvimento de plug-ins usando o próprio Eclipse. Essa ferramenta é o PDE (Plug-in Development Environment), conjunto de plug-ins que fornece uma perspective, com algumas *views* e *editors* relacionados ao processo. Recomenda-se fortemente usar o PDE para desenvolver plug-ins.

Um plug-in é um projeto Java como outro qualquer. Ele possui dois arquivos que o descrevem: plug-in.xml e MANIFEST.MF. Tais arquivos podem ser diretamente editados utilizando um editor visual, chamado "Plug-in Manifest Editor". Esse é o editor padrão para editar esses dois arquivos e facilita sobremaneira a tarefa de configurar os mesmos. Conjuntamente esses arquivos descrevem uma série de parâmetros do plug-in. A seguir falaremos sobre os principais. Cada um deles corresponde a uma guia no Plug-in Manifest Editor.

- dependencies – são os plug-ins dos quais esse plug-in depende. Um plug-in só pode fazer uso de classes, interfaces e *extension points* de outro plug-in se ele o tiver como dependência. Em tempo de execução, o runtime do Eclipse verifica se todas as dependências de um plug-in estão disponíveis (se os plug-ins necessários estão fisicamente colocados na pasta plug-ins dentro do diretório de instalação do Eclipse). Caso eles não estejam disponíveis, o runtime acusará um erro ao se tentar acessar aquele plug-in.

- runtime – são as configurações relativas ao *classpath* (JARs externos que devam ser utilizados) do plug-in e quais classes ele disponibiliza (exporta) para outros plug-ins. Outro plug-in só pode fazer uso das classes desse plug-in se as classes a serem utilizadas estiverem dentro dos pacotes exportados pelo plug-in fonte das classes.

- extensions – são pacotes que permitem ao plug-in contribuir com outros plug-ins. Contribuições visuais como *views*, *editors* e *perspectives* também são realizadas através de extensions.

- extension points – são artifícios que permitem ao plug-in disponibilizar funcionalidades para outros plug-ins. É possível com esse mecanismo, criar menus, editores e outros conteúdos dinâmicos, resultantes das diversas contribuições de outros plug-ins e que são determinados apenas em tempo de execução.

Um aspecto interessante dos plug-ins é que eles só são carregados quando se utiliza uma classe deles. Quando isso acontece, a classe que representa o plug-in propriamente dito (uma subclasse de org.eclipse.core.runtime.Plug-in ou org.eclipse.ui.plug-in.AbstractUIPlug-in, dependendo se o plug-in faz alguma contribuição para a interface gráfica do Eclipse, ou não) é instanciada, dando início ao ciclo de vida do plug-in. Então a classe requerida é instanciada e retornada. Esse mecanismo permite economizar muita memória na medida em que muitos recursos não são utilizados em uma sessão de uso do Eclipse e, portanto, não precisam ser carregados em memória.

Para se criar um plug-in, portanto, deve-se configurar o seu plug-in.xml e MANIFEST.MF de acordo com as características dele. Depois se devem codificar as classes necessárias ao funcionamento dele, tomando cuidado especial na codificação das classes que implementam as *extensions* desse plug-in.

Uma vez finalizado seu desenvolvimento, um plug-in pode ser exportado sobre a forma de um JAR ou de uma pasta contendo os arquivos *.class e recursos necessários. A partir da versão 3.0 do Eclipse, a forma de JAR passou a ser a forma recomendada de exportar um plug-in. O plug-in exportado deve então ser colocado dentro da pasta "plug-ins" contida na pasta raiz de instalação do Eclipse.

Como os arquivos JAR usam um padrão de compressão baseado no formato de compressão ZIP de modo que qualquer aplicativo que suporte esse formato possa abrir e manipular um *.jar, caímos em uma questão muito importante que é a proteção do código fonte.

A linguagem Java por possuir uma maneira diferente de compilação, ela possui uma maior facilidade de sofrer engenharia reversa, uma vez que não é compilada para código de máquina e sim para um código intermediário.

Desenvolvedores podem "ofuscar" o código fonte [9], ou seja, torná-lo mais difícil de entender. Essa prática tem o propósito de dificultar a engenharia reversa, ou apenas funcionar como desafio recriacional. Hoje existem diversos programas conhecidos como *obfuscators* que transformam o código legível em um código "embaralhado" usando diferentes técnicas. Dessa forma, a utilização dos arquivos JAR e mesmo dessas técnicas, de modo algum garante a segurança de seu código, por isso deve-se ficar atento aos requisitos de sua aplicação, finalidades e o modo que será colocada a disposição dos usuários.

## 1.11. Programação Orientada a Eventos (POE)

Para implementar a funcionalidade de uma interface gráfica, pode-se fazer uso de uma repetição sem fim. Um *thread* verifica se o mouse foi movido, se algum botão ou alguma tecla foi pressionado, e em caso positivo, é realizada alguma ação se necessário, e este ciclo é repetido indefinidamente. Esse tipo de programação é considerado extremamente ineficiente, pois é necessário explicitamente "ler" os periféricos para tratar as ações do usuário, e dessa forma, o programa perde muito tempo e processamento fazendo essas leituras. Em um sistema multitarefa, esse tipo de implementação seria inviável, pois os demais processos ficariam muito tempo parados.

Um evento é um acontecimento, algo que acontece ou se faz acontecer sobre um objeto da aplicação. Um acontecimento que o objeto pode reconhecer. Na programação dita orientada a eventos, os eventos são desencadeados através de ações de usuários interagindo com o programa, por outro programa requisitando uma troca de dados, ou pelo próprio *Windows*. Mexer e clicar botões do mouse, pressionar teclas no teclado, são ações que desencadeiam eventos. Como exemplo de eventos, podemos citar: *Click*, *MouseMove*, *Keypress*. Se uma aplicação está interessada em um evento específico (por exemplo, clique em um botão), deve solicitar ao sistema para "escutar" o evento. Se a aplicação não está interessada, seu processamento continua de forma normal. É importante observar que a aplicação não espera pela ocorrência de eventos: isso é controlado pelo sistema.

Para que um componente ou container possa "escutar" eventos, é preciso associar um *listener* ao objeto que gera o evento. *Listeners* são classes criadas especificamente para o tratamento de eventos. Quando um evento é gerado, o sistema de execução notifica o *listener* correspondente, invocando o método que trata o evento. Caso não exista nenhum *listener* associado ao objeto que gera o evento, este não terá qualquer efeito.

Para cada tipo de evento temos um *listener* correspondente. Um objeto pode ser registrado como um *listener* se for uma instância de uma classe que implementa uma determinada interface. Um único *listener* pode ser associado a múltiplos objetos geradores de eventos, assim como vários *listeners* podem ser associados a um único evento.

Os *listeners* são implementados através de interfaces. Uma interface define um conjunto de métodos que uma classe deve implementar mas não define como esses métodos devem ser implementados, representando um modelo de como os métodos devem ser escritos. Diferentemente de classes abstratas, interfaces não servem para criar novas classes, mas sim para implementar funcionalidades diversas. Uma classe implementa uma ou mais interfaces, de certa forma, substituindo a herança múltipla em Java. Podemos citar como exemplo um *MouseListener*, que é uma interface para eventos relacionados ao mouse.

A POE é bastante flexível e permite um sistema assíncrono. Programas com interface com o usuário geralmente utilizam tal paradigma, sistemas operacionais também são outro exemplo de programas que utilizam programação orientada a eventos, este em dois níveis. No nível mais baixo encontram-se o tratamento de interrupções como tratadores de eventos de hardware, com a CPU realizando o papel de disparador. No nível mais alto encontram-se os processos sendo disparados novamente pelo sistema operacional.

## 1.12. GEF

### 1.12.1. Introdução

O GEF (*Graphical Editing Framework*) é um *framework* que permite o desenvolvedor criar um rico editor gráfico a partir de um modelo de aplicação existente. Com esse editor é possível fazer simples modificações em seu modelo, como trocar propriedades de um elemento ou operações complexas como mudar a estrutura do seu modelo de diversas maneiras ao mesmo tempo. Todas essas modificações em seu modelo podem ser tratadas no editor gráfico usando funções comuns como *drag and drop*, *copy and paste*, e ações invocadas por *menus* ou *toolbars*. Isso graças a sua arquitetura MVC.

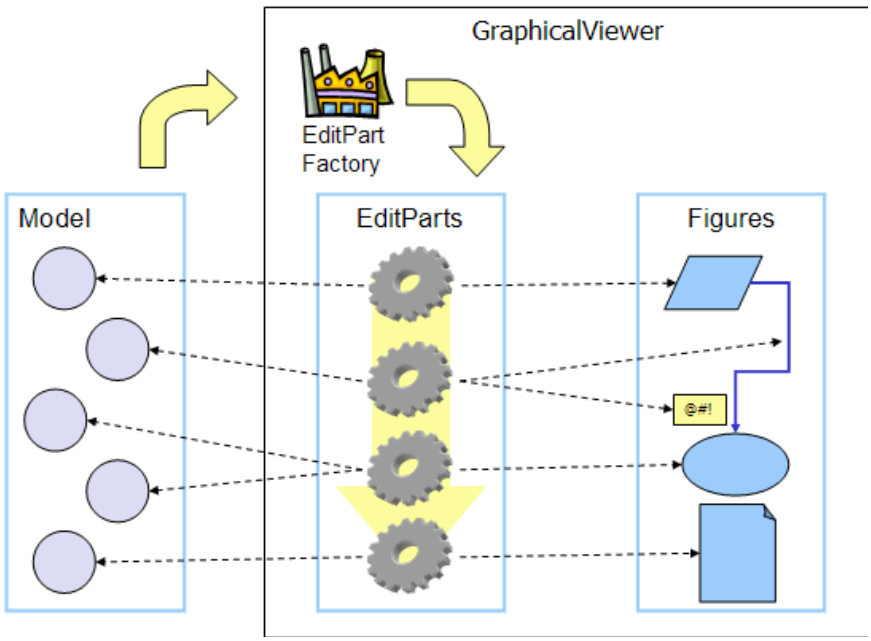


Figura 2.11: Arquitetura MVC da GEF

A Figura 2.11 apresenta o *Model* que possui a responsabilidade de armazenar seus dados. Estes dados são apresentados em suas *views* correspondentes através das *Figures*. As *EditParts* preenchem as *Figures* com os dados do *Model*, e por este motivo podem ser consideradas como *Controllers*. As *EditParts*, por sua vez são criadas e inicializadas por uma *EditPartFactory*, um objeto de criação alinhado com o padrão *Abstract Factory*. Por Fim, a *GraphicalViewer* tem a função de exibir as *Figures* e chamar todos os métodos apropriados de todos os outros participantes.

O GEF, como a maioria dos outros plug-ins do projeto Eclipse, está licenciado sob a licença EPL (Eclipse Public License), o que garante que seu código é aberto e permite que ele seja utilizado em aplicações comerciais, de código fechado.

Como todos os plug-ins visuais do Eclipse, ele faz uso do SWT para criação de componentes visuais e para capturar eventos gerados pela interação do usuário com a interface gráfica. Toda visualização gráfica é feita via o framework Draw2D, que é um framework padrão de desenho 2D baseado na SWT da eclipse.org.

### 1.12.2. Draw2D

O termo "sistema *lightweight*" é usado para sistemas gráficos que são hospedados dentro de um controle *heavyweight* (este ligado com seu recurso visual

nativo, como a AWT, e são somente compostos por componentes retangulares). Os objetos em um sistema *lightweight*, conhecidos como *Figures* na Draw2D, são tratados como se fossem janelas normais, e por serem um simples objeto Java não possui recurso correspondente no sistema operacional. Eles podem ter foco e seleção, capturar efeitos do *mouse,* ter seu próprio sistema de coordenadas, e ter um *cursor*. A vantagem é que eles são muito mais flexíveis do que o de sistema de janelas, no qual geralmente são compostos por componentes retangulares. Ele permite criarmos e manipularmos objetos gráficos de formatos arbitrários. Uma vez que, eles simulam um sistema *heavyweight* com uma única janela *heavyweight*, eles nos permite criar um complexo display gráfico sem consumir muitos recursos do sistema.

Sob esta ótica, a Draw2D prove um sistema gráfico *lightweight* que o GEF depende para mostrar os elementos gráficos. Ela é empacotada no Eclipse como um plug-in separado, org.eclipse.draw2d. As *figures* são análogas a *windows* em um sistema gráfico *heavyweight*, podendo ter formas arbitrárias não retangulares, e ainda, podem ser aninhadas para compor uma cena complexa ou controles customizados. A arquitetura pode ser vista na figura 2.12.

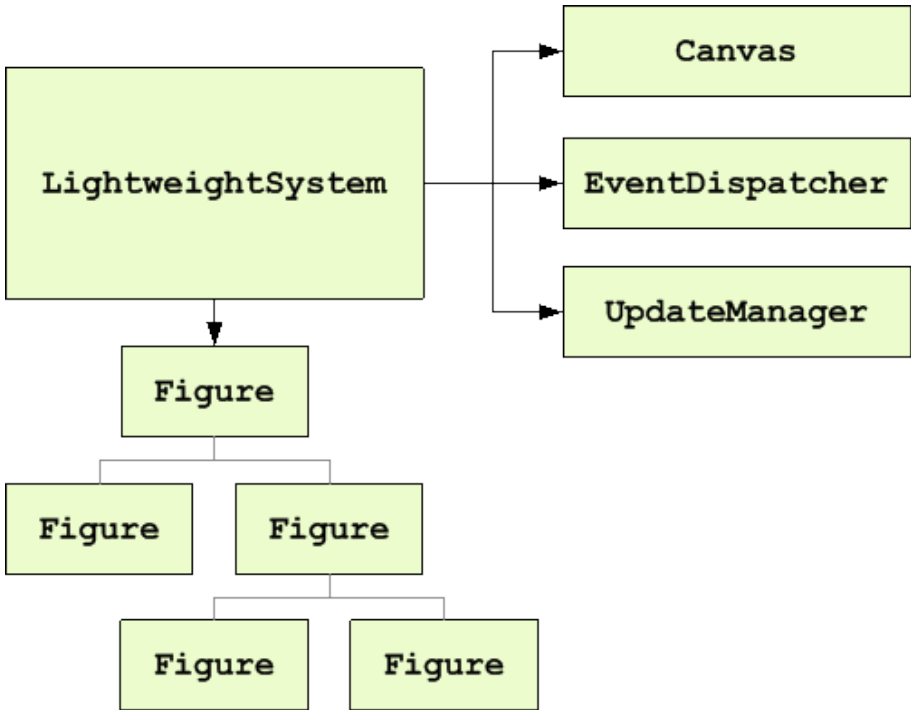


Figura 2.12: Arquitetura Draw2D

O *LightweightSystem* associa a *figure* com um SWT canvas. O mesmo também possui *listeners* para maioria dos eventos da SWT, e encaminha-os para um *EventDispatcher*, que traduz os eventos para as figuras. No entanto, o evento de pintura é encaminhado para *UpdateManager*, que coordena a pintura e o layout.

As *Figures* podem ser transparentes ou opacas, e podem ser ordenadas em camadas. Deste modo permite que partes de um diagrama possam ser escondidas ou excluídas por certas operações. A Draw2D é uma biblioteca gráfica *standalone* que pode ser usada para criar *views* no Eclipse. A figura 2.13 apresenta a interação da GEF com a biblioteca Draw2D.

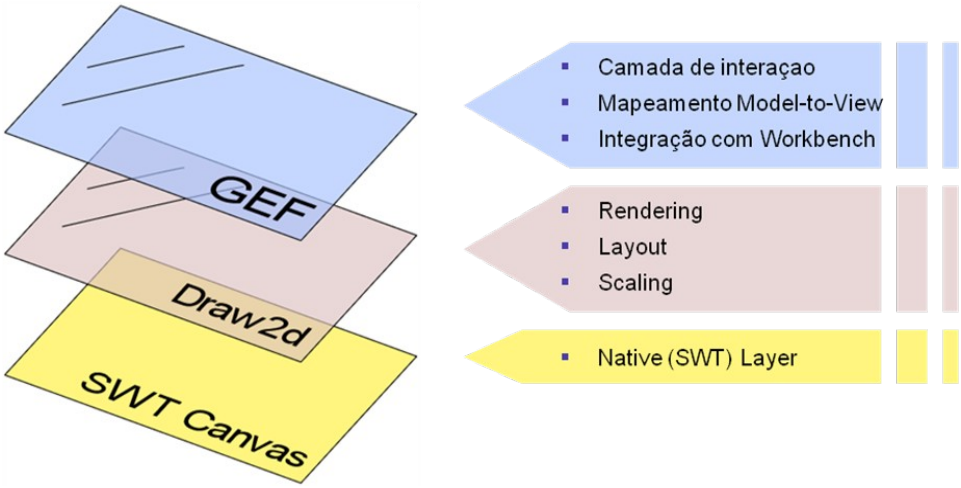


Figura 2.13: Integração da GEF com a Draw2D (estraída de [14])

Seguem ainda algumas outras funcionalidades da biblioteca GEF:

- *Rendering* - renderização de imagens, que podem ser atualizadas e marcadas como inválidas, de modo a serem posteriormente redesenhadas na tela.
- *Layout* - posicionamento automático dos elementos na tela de acordo com alguma regra pré-definida, independente do tamanho do editor.
- *Coordinate systems* - sistema de coordenadas absoluto e relativo para posicionamento dos elementos no diagrama.
- *Scaling* - capacidade de representar os elementos de maneira consistente e proporcional quando se faz o uso de "zoom out" ou "zoom in".
- *Hit testing* - verificar se em um dado ponto existe alguma figura.
- *Layers* - camadas de figuras, para facilitar a representação delas, permitindo operar somente em uma dada camada, ao invés de operar em todas.
- *Connections* - ligações entre os elementos gráficos, conhecidos como *nodes*
- *Routing* - roteamento, ou seja, definição do trajeto automático das connections, de acordo com alguma regra pré-definida.

A vantagem de se usar o plug-in Draw2d para prover figuras ao GEF é que ele possui uma integração maior, provendo facilidades ao desenvolvedor que, de outro modo, ele teria que implementa-las sozinho.

# Metodologia

O cenário que nos foi proposto era de criar um framework que possuísse uma arquitetura, na qual seria possível adicionar novos plug-ins de forma prática e fácil. O mesmo também deveria agir apenas como uma "casca" receptora de outras extensões, e ignorar completamente a relação entre elas. A solução deveria ser escalável de forma que adição de novos componentes não deveria prejudicar o desempenho do framework como um todo e muito menos onerar o sistema operacional.

Dessa forma temos a seguir o diagrama de componentes da aplicação. No qual a única dependência (ver figura 3.1) que os plug-ins têm com o framework é o *Common* que é obrigatório para os plugins que desejam se comunicar com os outros que também estão ligados ao *Common*. Caso a extensão não necessite de comunicação, ela não precisa ter essa dependência.

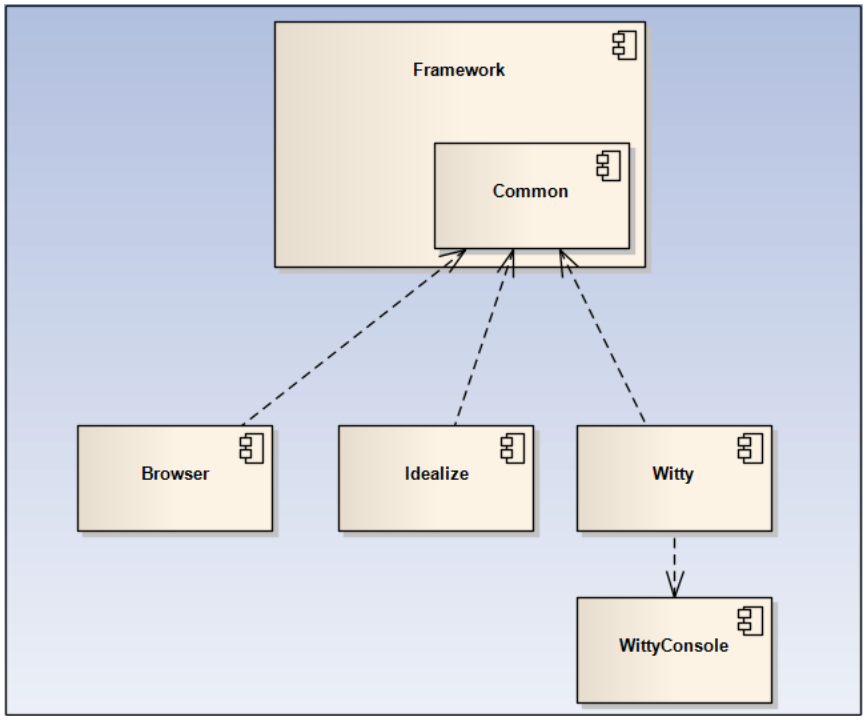


Figura 3.1: Diagrama de Componentes

## 1.13. Plug-in Container (Principal)

O plug-in container é o framework principal da aplicação RCP em questão. Nele ficarão todas as configurações do workbench, e de menus padrão importados do Eclipse que serão utilizados pelos outros plug-ins. O conceito empregado é receber outros plug-ins, que serão utilizados na forma de perspectivas. O presente programa utiliza a IDE do Eclipse, de forma a incluir todos os componentes que estiverem na pasta interna "plug-ins" no momento da inicialização do sistema. É importante relembrar que as dependências dos componentes utilizados são mantidas, de forma que estas devem ser utilizadas na formação do produto principal, para que as perspectivas independentes funcionem normalmente.

Em uma aplicação RCP criada no Eclipse, as ações (actions) e itens de contribuição registradas na classe ApplicationActionBarAdvisor da aplicação principal, podem ser utilizadas em qualquer perspectiva, e devido a isso, adotamos esta estratégia para padronizar alguns menus e barras utilizados em qualquer aplicação. Para fazer o registro de tais ações, primeiro é necessário criar uma variável privada para cada um deles conforme exemplo abaixo:

```
        private
;
        IWorkbenchAction exitAction;
        private IContributionItem perspectiveItem;
        private IWorkbenchAction aboutAction;
        private IWorkbenchAction closeAction;
        private IWorkbenchAction closeAllAction;
        private IWorkbenchAction saveAction;
        private IWorkbenchAction saveAsAction;
        private IWorkbenchAction saveAllAction;
        private IWorkbenchAction revertAction;
        private IWorkbenchAction undoAction;
        private IWorkbenchAction redoAction;
        private IWorkbenchAction cutAction;
        private IWorkbenchAction copyAction;
        private IWorkbenchAction pasteAction;
        private IWorkbenchAction selectAllAction;
                private IWorkbenchAction findAction
```

Após a declaração de tais ações e itens de contribuição, é necessário atribuir um valor a eles, recuperado de *Factories* padrão do Eclipse, e no caso das ações, registrá-las no ambiente. Isto é feito sobrescrevendo o método makeActions conforme código abaixo:

```
        @Override
        protected void makeActions(IWorkbenchWindow window)
        {
```

```
        this.perspectiveItem = ContributionItemFactory.PERSPECTIVES_SHORTL-
IST.create(window);

        this.exitAction = ActionFactory.QUIT.create(window);
        this.register(this.exitAction);

        this.closeAction = ActionFactory.CLOSE.create(window);
        this.register(this.closeAction);

        this.closeAllAction = ActionFactory.CLOSE_ALL.create(window);
        this.register(this.closeAllAction);

        this.saveAction = ActionFactory.SAVE.create(window);
        this.register(this.saveAction);

        this.saveAsAction = ActionFactory.SAVE_AS.create(window);
        this.register(this.saveAsAction);

        this.revertAction = ActionFactory.REVERT.create(window);
        this.register(this.revertAction);

        this.saveAllAction = ActionFactory.SAVE_ALL.create(window);
        this.register(this.saveAllAction);

        this.undoAction = ActionFactory.UNDO.create(window);
        this.register(this.undoAction);

        this.redoAction = ActionFactory.REDO.create(window);
        this.register(this.redoAction);

        this.cutAction = ActionFactory.CUT.create(window);
        this.register(this.cutAction);

        this.copyAction = ActionFactory.COPY.create(window);
        this.register(this.copyAction);

        this.pasteAction = ActionFactory.PASTE.create(window);
        this.register(this.pasteAction);

        this.selectAllAction = ActionFactory.SELECT_ALL.create(window);
        this.register(this.selectAllAction);

        this.findAction = ActionFactory.FIND.create(window);
        this.register(this.findAction);

        this.aboutAction = ActionFactory.ABOUT.create(window);
        this.register(this.aboutAction);
    }
```

Após este passo, as ações estão prontas para serem acessadas em qualquer parte do sistema. Esta configuração de acesso pode ser feita em menus e/ou barras. Apesar de herdar componentes da IDE do Eclipse ajudar na grande maioria das situações, existem casos em que as contribuições não são pertinentes ao software em questão. Devido a isso, além de adicionar, em alguns casos é necessário remover menus, e ambas as ações citadas, são sobrescrevendo o método fillMenuBar, conforme situação abaixo:

```
@Override
	protected void fillMenuBar(IMenuManager menuBar) {
		MenuManager perspectivesMenu = new MenuManager("Change Perspective", "per-
spectiva");

		// Creating new Menus
		MenuManager fileMenu = new MenuManager("&File", IWorkbenchActionConstant-
s.M_FILE);
		MenuManager editMenu = new MenuManager("&Edit", IWorkbenchActionConstant-
s.M_EDIT);
		MenuManager helpMenu = new MenuManager("&Help", IWorkbenchActionConstant-
s.M_HELP);

		menuBar.add(fileMenu);
		menuBar.add(editMenu);
		menuBar.add(new GroupMarker(IWorkbenchActionConstants.MB_ADDITIONS));
		menuBar.add(helpMenu);

		// File
		fileMenu.add(perspectivesMenu);
		perspectivesMenu.add(this.perspectiveItem);
		fileMenu.add(new GroupMarker(IWorkbenchActionConstants.MB_ADDITIONS));
		fileMenu.add(new Separator());
		fileMenu.add(this.closeAction);
		fileMenu.add(this.closeAllAction);
		fileMenu.add(new Separator());
		fileMenu.add(this.saveAction);
		fileMenu.add(this.saveAsAction);
		fileMenu.add(this.saveAllAction);
		fileMenu.add(this.revertAction);
		fileMenu.add(new Separator());
		fileMenu.add(this.exitAction);

		// Edit
		editMenu.add(new GroupMarker(IWorkbenchActionConstants.MB_ADDITIONS));
		editMenu.add(this.undoAction);
		editMenu.add(this.redoAction);
		editMenu.add(new Separator());
		editMenu.add(this.cutAction);
		editMenu.add(this.copyAction);
		editMenu.add(this.pasteAction);
		editMenu.add(new Separator());
		editMenu.add(this.selectAllAction);
		editMenu.add(new Separator());
		editMenu.add(this.findAction);
		editMenu.add(new Separator());

		// Help
		helpMenu.add(new GroupMarker(IWorkbenchActionConstants.MB_ADDITIONS));
		helpMenu.add(new Separator());
		helpMenu.add(this.aboutAction);

		// Removing unnecessary menus
		ActionSetRegistry reg = WorkbenchPlug-in.getDefault().getActionSetRegistry();
		IActionSetDescriptor[] actionSets = reg.getActionSets();

		String actionSetId = "org.eclipse.ui.actionSet.openFiles";
		this.removeActionSet(reg, actionSets, actionSetId);
```

```
		actionSetId = "org.eclipse.ui.edit.text.actionSet.convertLineDelimitersTo
	this.removeActionSet(reg, actionSets, actionSetId);

		actionSetId = "org.eclipse.ui.edit.text.actionSet.navigation";
		this.removeActionSet(reg, actionSets, actionSetId);

		actionSetId = "org.eclipse.ui.edit.text.actionSet.annotationNavigation";
		this.removeActionSet(reg, actionSets, actionSetId);
		}
```

Segue abaixo também, o código do método que remove um actionSet do framework:

```
private void removeActionSet(ActionSetRegistry reg, IActionSetDescriptor[] actionSets, String action-
SetId) {
		for (int i = 0; i < actionSets.length; i++) {
			if (!actionSets[i].getId().equals(actionSetId)) {
				continue;
			}
			IExtension ext = actionSets[i].getConfigurationElement().getDeclaringExten-
sion();
			reg.removeExtension(ext, new Object[] { actionSets[i] });
		}
		}
```

É importante perceber que com as adições de itens ao argumento deste método (menuBar), a organização dos menus é criada. Ao criar um menuManager, como por exemplo o “File” , foi utilizado um “&” antes do nome, para associar o atalho “alt + f” ao mesmo. Analogamente ao feito acima, pode-se associar contribuições ao coolbar sobrescrevendo o método fillCoolbar, e adicionando as ações a argumento coolBar.

Com os menus padrão já adicionados, pode-se prosseguir com a configuração de inicialização do aplicativo. Tal configuração é feita sobrescrevendo o método preWindowOpen da classe ApplicationWorkbenchWindowAdvisor, onde pode-se, por exemplo, ajustar o tamanho em que a aplicação abre, se a coolbar e perspectivebar aparecem, dentre outras coisas. Segue código de exemplo abaixo:

```
@Override
	public void preWindowOpen() {
		IWorkbenchWindowConfigurer configurer = this.getWindowConfigurer();
		configurer.setInitialSize(new Point(1200, 900));
		configurer.setShowCoolBar(true);
		configurer.setShowPerspectiveBar(true);

		}
```

Normalmente as aplicações RCP possuem uma perspectiva padrão, na qual a aplicação se inicia. Tal configuração é feita na classe ApplicationWorkbenchAdvisor, passando para o método getInitialWindowPerspectiveId, o ID da mesma. Entretanto, o plug-in principal é completamente desacoplado dos componentes anexos, só conhecendo as perspectivas no momento em que é inicializado. Quando uma perspectiva é setada como inicial, ela automaticamente torna-se a default. O que a aplicação faz, é verificar quais são as perspectivas registradas, e determina a primeira como inicial, isto é, default. Na situação proposta, o *browser* é a primeira perspectiva registrada, e por este motivo é a perspectiva padrão. Se o *browser* for retirado, uma das outras duas perspectivas se apresentará como default. Devido a isso, durante sua inicialização é necessário fazer uma varredura nas perspectivas registradas, e apresentar uma delas, para que o usuário não inicie o sistema de uma forma não funcional. Segue abaixo, o código que realiza este procedimento:

```
@Override
	public String getInitialWindowPerspectiveId() {

		IPerspectiveRegistry reg = WorkbenchPlug-in.getDefault().getPerspectiveRegistry();
		if ((reg != null) && (reg.getPerspectives().length > 0)) {
			return reg.getPerspectives()[0].getId();
		}
		return null;
		}
```

O último passo de configuração do container é puramente visual, e consiste na alteração das preferências da aplicação. Tal procedimento deve ser feito na classe ApplicationWindowAdvisor, sobrescrevendo o método initialize, de modo que as alterações sejam aplicadas para todos os plug-ins anexados ao framework principal. Desta forma pode-se, por exemplo, alterar as tabs de views e editores para o modelo doa versão atual do Eclipse. Segue o código abaixo que mostra a utilização deste artifício:

```
@Override
	public void initialize(IWorkbenchConfigurer configurer) {
		super.initialize(configurer);
		PlatformUI.getPreferenceStore().setValue(IWorkbenchPreferenceConstant-
s.SHOW_TRADITIONAL_STYLE_TABS, false);

	}
```

## 1.14. Plug-in Mediador de Eventos

O plug-in mediador de eventos utiliza o padrão de projeto *Mediator*, e é o componente que faz a conexão entre os demais plug-ins existentes no ambiente. Apesar de não ter nenhum elemento visual, ele é o elemento central do sistema, responsável pelo fato de ser possível realizar a comunicação independente citada anteriormente.

Por ser o elemento responsável pela troca de mensagens, este componente tem que necessariamente ser adicionado à lista de dependências de qualquer plug-in do sistema onde seja necessário utilizar a comunicação.

O funcionamento do mesmo consiste em uma lista de *listeners*, onde os componentes quem desejam ouvir os eventos devem se registrar. Quando um evento ocorre, a classe responsável verifica quais *listeners* registrados podem interagir com a situação.

Segue a baixo um esquema explicativo:

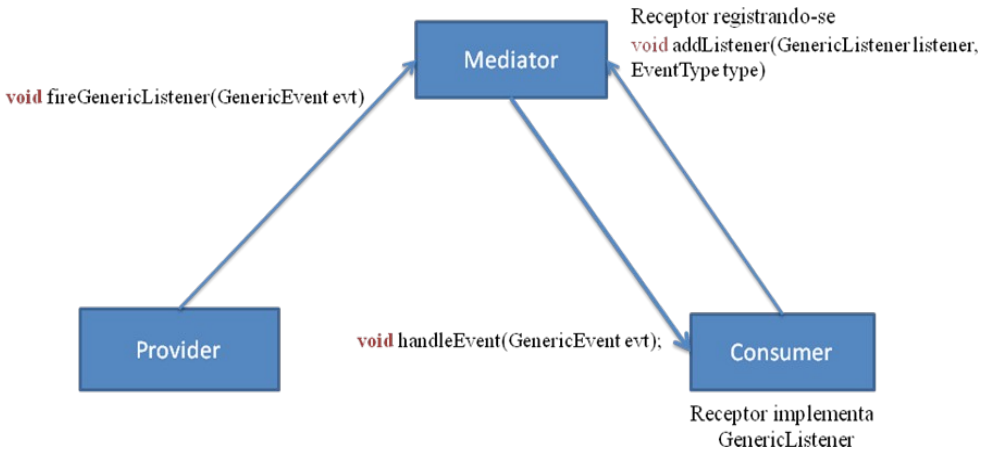


Figura 3.2: Esquema mediador

O funcionamento do mediador utilizado no sistema é constituído de três classes, mais detalhadamente descritas a seguir:

**-GenericEvent:** é uma classe que estende *EventObject*, e é onde o evento propriamente dito é instanciado. A classe *GenericEvent* é o argumento para o método disparador, e nela estão os tipos de eventos disparados e objetos necessários para a execução da ação, que podem ser visto no código abaixo:

```
public class GenericEvent extends EventObject {
```

```
	private static final long serialVersionUID = 1L;

	public enum EventType {

		FIRE_BROWSER, ADD_CHILD;

		private EventType() {

		}

		public String value() {
			return this.name();
		}

		public static EventType fromValue(String v) {
			return valueOf(v);
		}
	}

	private Object eventObject;
	private EventType eventType;

	public GenericEvent(Object source, EventType eventType,Object e) {
		super(source);
		this.eventObject = e;
		this.eventType = eventType;
	}

	public Object getObject() {
		return this.eventObject;
	}

	public EventType getEventType() {
		return this.eventType;
	}
}
```

A classe mostra os tipos de evento FIRE_BROWSER e ADD_CHILD, e os métodos de recuperação do objeto enviado junto ao evento, para realizar a ação disparada.

**-GenericListener:** é uma interface que estende *EventListener*, e comporta o método executado após o disparo do evento, e devido a isso, as classes que recebem mensagens, devem implementar a classe *GenericListener*. Seu código é bastante simples, e é mostrado a seguir:

```
public interface GenericListener extends EventListener {

	public void handleEvent(GenericEvent evt);
	}
```

**-ListenersList:** é uma classe que utiliza o padrão de projeto singleton, ou seja, é garantido que existe uma única instância desta classe, acessível de forma global. Para que a classe utilize este padrão, é necessário que seu construtor seja privado, de forma a tornar obrigatório o uso do método público *getInstance*() para instanciar o objeto.O trecho de código a seguir detalha o método citado:

```
private static ListenersList instance;
public static ListenersList getInstance() {
        if (instance == null) {
                instance = new ListenersList();
        }
        return instance;
        }
```

Esta classe possui como um de seus atributos, a lista de *listeners* onde os plug-ins ouvidores se registrarão. Por ser uma classe singleton, a lista de *listeners* será sempre a mesma para qualquer componente que a instancie. Além disso, é a responsável pelo método que dispara os eventos que serão enviados. O método que dispara os eventos redirecionados pelo mediador se encontra a seguir:

```
public void fireGenericListener(GenericEvent evt) {
        for (GenericListener listener : this.listeners) {
                listener.handleEvent(evt);
        }
        }
```

Finalmente, a *ListenersList* possui uma lista onde serão armazenados todos os *listeners* registrados como ouvintes de eventos. Durante a inicialização de uma classe que pretende tratar um evento, ela deve chamar o método *addListener* usando a si mesma como argumento. Após este registro, a classe está preparada para receber um evento. Quando um evento for disparado pelo método *fireGenericListener*, os *listeners* registrados (que necessariamente devem implementar *GenericListener*) receberão o evento, e verificarão se são capazes de tratá-lo dentro do método sobrescrito *handleEvent*. Caso a resposta seja afirmativa, o método *getObject*() do evento deverá ser acionado para adquirir o objeto com os atributos necessários para o tratamento do evento. A implementação mais simples para o modelo de listas, adição e remoção de ouvintes, é citado segue a seguir:

```
private Set<GenericListener> listeners;

private ListenersList() {
        this.listeners = new HashSet <GenericListener>();
}

public void addListener(GenericListener listener) {
        this.listeners.add(listener);
}

public void removeListener(GenericListener listener) {
        this.listeners.remove(listener);
        }
```

No modelo acima, como é necessário varrer todos os ouvintes para saber qual está preparado para tratar o evento (o próprio listener faz esta verificação), foi usado um Set (que é um tipo de ArrayList onde a ordenação não importa) no lugar de um ArrayList para armazenar os listeners, devido ao menor consumo de processamento que uma lista normal teria. Entretanto, esta implementação acarreta uma perda desnecessária de tempo enviando mensagens para listeners que não podem tratar o evento. Para solucionar este problema, utilizou-se de uma implementação mais inteligente, fazendo com que cada tipo de evento, crie uma lista de listeners. Desta forma, somente os ouvintes preparados para tratar determinado evento, receberão a mensagem.

Utilizando este novo formato, o meio de concretizar uma troca de mensagens é ligeiramente modificado. Como agora existe um conjunto de listeners, a classe ao se registrar deve indicar qual tipo de evento ela é capaz de tratar, sendo desta forma, adicionada à lista correta. Para a realização desse procedimento, o os métodos *addListener* e *removeListener* sofreram modificações para receber além da classe, um tipo de evento como parâmetro. Além disso, o método *fireGenericListener* foi modificado para buscar a lista correta de ouvintes, e somente para seus elementos encaminhar a mensagem. Neste novo modelo, não é necessário que no método *handleEvent* a classe registrada verifique se o tipo do evento está correto ao recebê-lo, pois qualquer evento recebido pela mesma, ela avisou ao se registrar, que é capaz de tratar. Abaixo se encontra a nova implementação da classe:

```
public class ListenersList {

        private static ListenersList instance;
        private Map<EventType, Set<GenericListener>> listeners;
```

```
	private ListenersList() {
		this.listeners = new HashMap<EventType, Set<GenericListener>>();
	}

	public static ListenersList getInstance() {
		if (instance == null) {
			instance = new ListenersList();
		}
		return instance;
	}

	public void addListener(GenericListener listener, EventType type) {
		if (!this.listeners.containsKey(type)) {
			Set<GenericListener> newListenerList = new HashSet<GenericListener>();
			this.listeners.put(type, newListenerList);
		}
		this.listeners.get(type).add(listener);

	}

	public void removeListener(GenericListener listener, EventType type) {
		Set<GenericListener> tempListenersList = this.listeners.get(type);
		tempListenersList.remove(listener);
		if (tempListenersList.isEmpty()) {
			this.listeners.remove(type);
		}
	}

	public void fireGenericListener(GenericEvent evt) {
		Set<GenericListener> tempListenersList = this.listeners.get(evt
				.getEventType());
		for (GenericListener listener : tempListenersList) {
			listener.handleEvent(evt);
		}

	}

}
```

## 1.15. Plug-in de Browser

A extensão do browser foi criada a partir do exemplo que é disponibilizado no próprio site do Eclipse. O mesmo utiliza o navegador padrão do sistema operacional, ou seja, a janela de navegação nada mais é do que uma instância do seu Browser padrão. Na figura 3.3 temos a foto do exemplo, sem alterações, sendo executado.

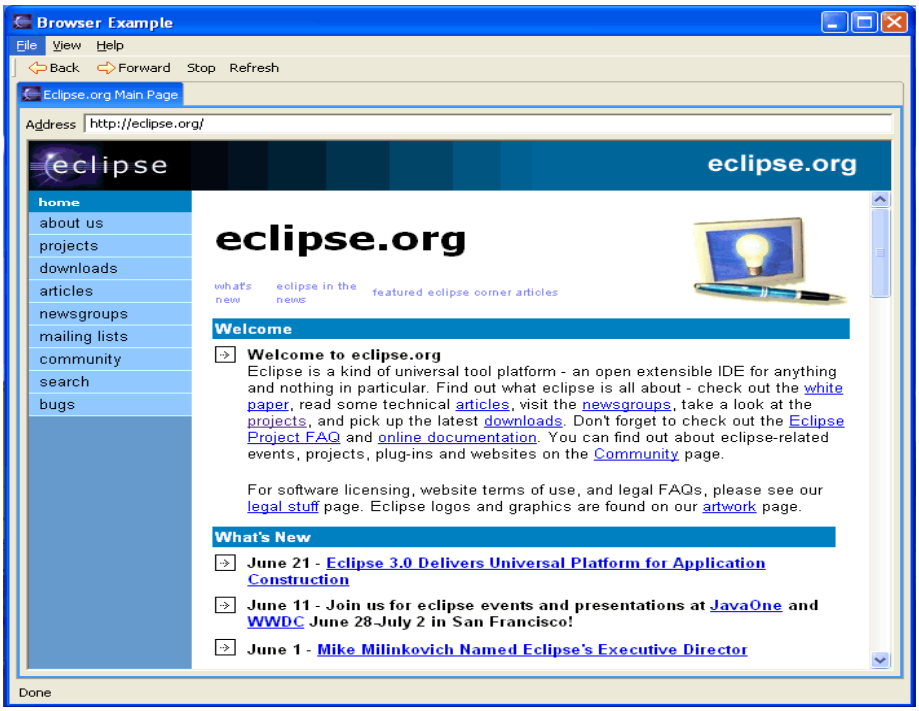

Figura 3.3: Exemplo do Browser RCP

Para que ele atendesse a nossas necessidades de requisitos foram necessárias algumas alterações no código. Primeiramente retiramos uma *view*, que apenas tinha a função de simular o histórico de um navegador.

Como o *browser* seria agregado ao plug-in principal, tivemos que tomar as devidas precauções para que todas suas funcionalidades continuassem disponíveis quando executado através do Container.

O primeiro passo foi adicionar a perspectiva como uma *extension*, permitindo que ela fosse visualizada no plug-in que a incorporou, o id escolhido para o elemento foi "br.ufrj.del.browser.browserPerspective", como mostra a figura 3.4 a seguir.

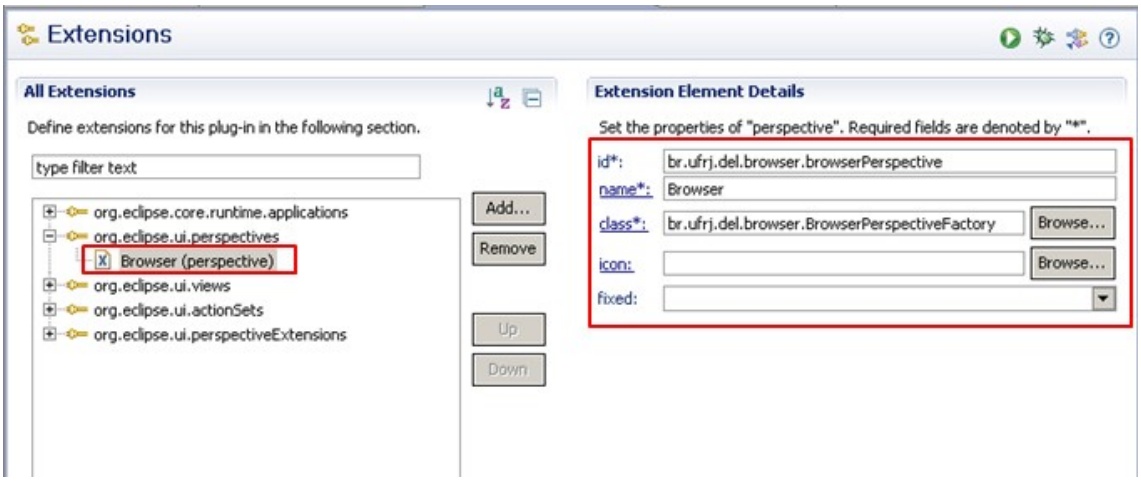


Figura 3.4: Extensão da Perspectiva

Outro passo semelhante foi realizado para o caso das *actions* que incorporam o *browser*, pois existe a necessidade de que elas sejam adicionadas a um *actionSet* pelo mesmo motivo do passo anterior, ilustrado na figura 3.5 a seguir.

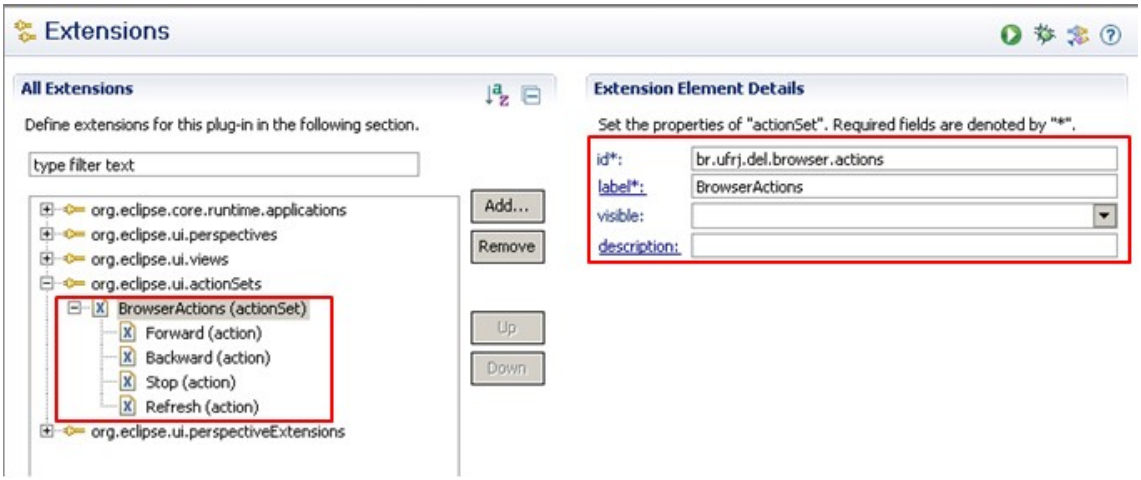


Figura 3.5: ActionSet

Agora temos que relacionar as ações criadas com a perspectiva para que eles sejam ativados e desativados conjuntamente. Essa ação previne que um toolbar, por exemplo, fique disponível fora do contexto adequado. A figura 3.6 ilustra como relacionamos o *actionSet* e a respectiva perspectiva.

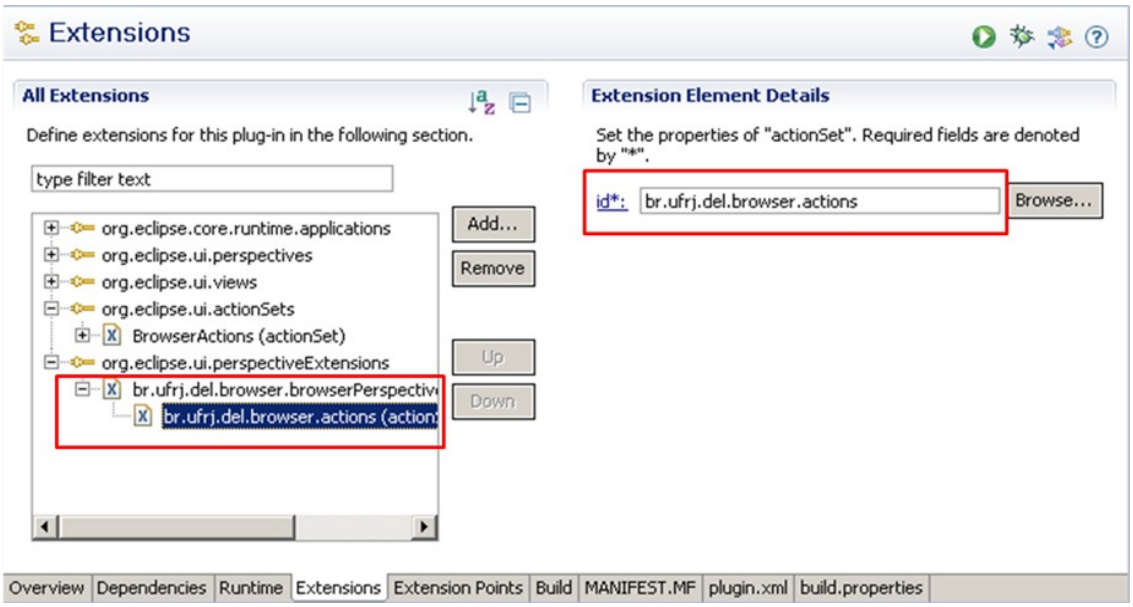


Figura 3.6: Perspectiva *Extention*

Depois de realizadas as devidas alterações nosso browser ficou pronto para ser incorporado ao plug-in receptor. O último passo é adicionar o navegador como nova funcionalidade do framework principal, como mostra figura 3.7 a seguir.

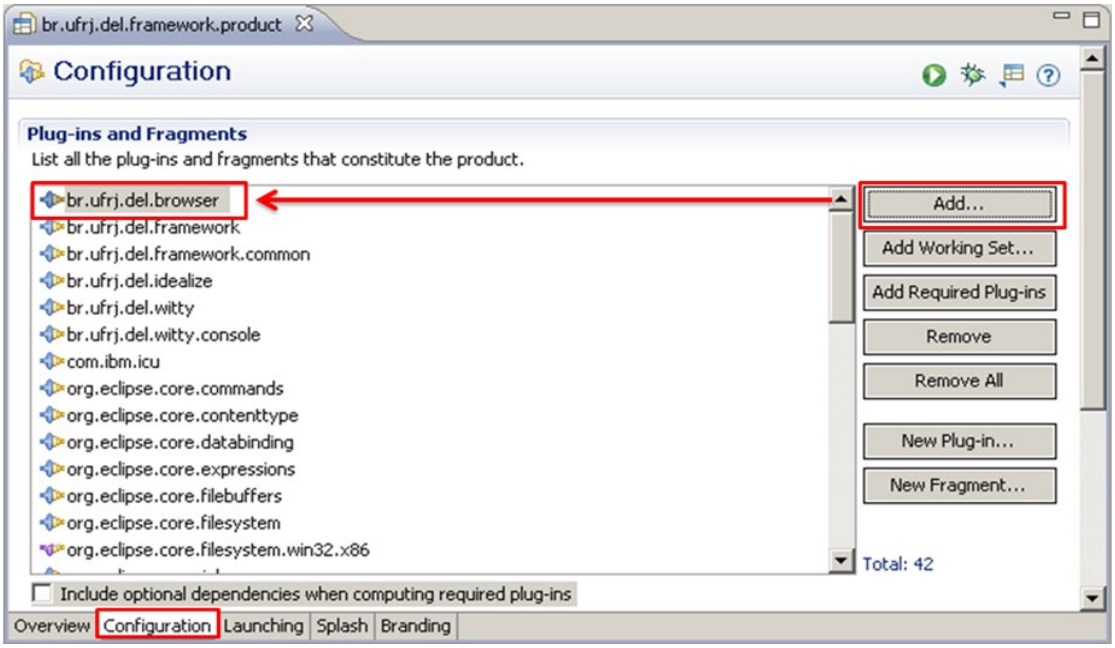


Figura 3.7: Plug-ins do framework principal

## 1.16. Plug-in Idealize

O idealize permite criar e editar diagramas. Ele manipula dois tipos de objetos, representados por retângulos e elipses. É possível conectar qualquer dos objetos usando um dos dois tipos de conexões, representados por uma linha solida ou tracejada. Cada conexão é orientada, no sentido que ela começa no objeto fonte e termina no objeto alvo. E sua direção é indicada por uma seta, e ela pode ser colocada arrastando-a do objeto fonte ou alvo a um novo objeto. Objetos no editor podem ser selecionados por um *click* ou sendo envolvido pela ferramenta de seleção, e nessa condição podem ser deletados. Toda manipulação do modelo, como adicionar ou remover objetos, movê-los, redimensionar, etc., podem ser desfeitas ou refeitas. Finalmente, o editor integra-se com as *views* padrão *Properties* e *Outline* do Eclipse.

Como vimos no item sobre a GEF, ela possui a arquitetura MVC, por isso nada mais natural que nossa aplicação siga essa mesmo estrutura. Desta maneira, passaremos a seguir em cada um dos elementos constituintes da arquitetura MVC e explicaremos como ela foi aplicada em nosso plug-in.

### 1.16.1. Model

No nosso caso o modelo consiste em três tipos de objetos: o *DiagramShapes*, que possui as formas, dois tipos de formas; *ElipticalShape* e *RectangularShape*, e a a conexão *Connection*.

O modelo tem algumas funções e obrigações dentro da aplicação:

- carrega toda informação que pode ser editada ou vista pelo usuário
- fornece maneiras de persistir o modelo
- deve desconhecer a implementação da *view* e do *controller*

fornece uma maneira para que suas mudanças possam ser propagadas

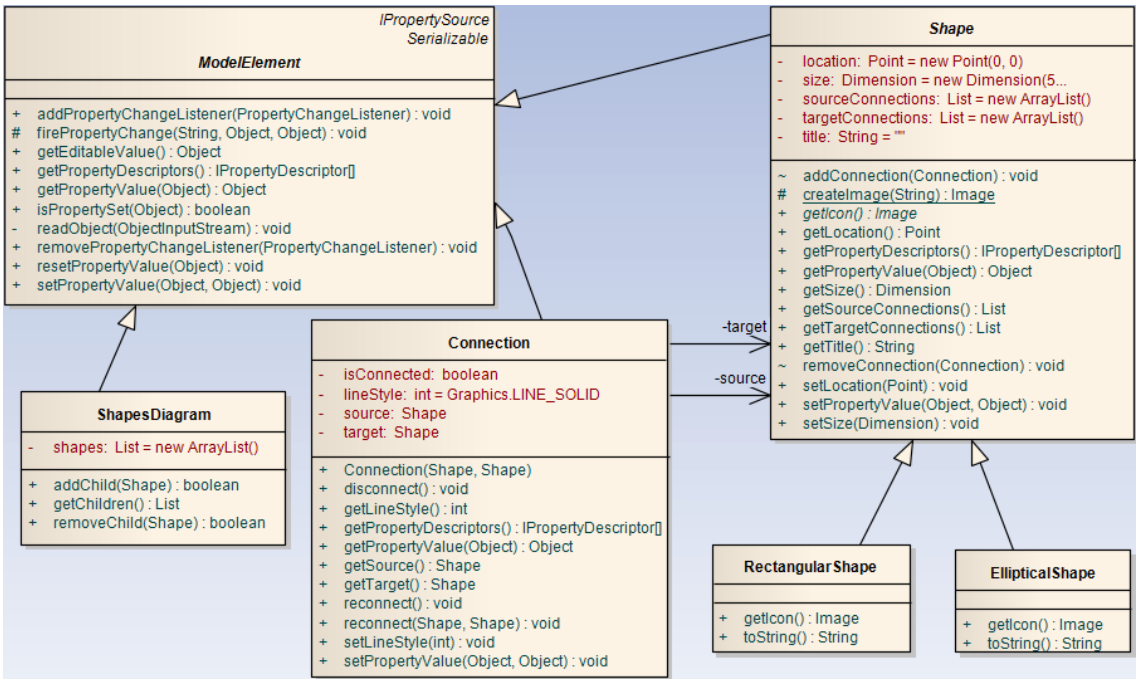


Figura 3.8: Diagrama de classes do Idealize (Model)

A persistência do modelo, conforme apresentado na figura 3.8, é garantida quando a classe *ModelElement* implementa a interface java.io.Serializable juntamente com o método *readObject*(). Essa solução permite que o editor seja salvo no formato binário. No entanto, vale a ressalva que isso pode funcionar para certas aplicações, mas não fornece portabilidade. Em casos mais complexos, deve se implementar o modelo salvando em XML ou formato similar.

As classes *EllipticalShape* e *RectangularShape* compartilham funcionalidades em comum por isso podem ser especializações de uma classe em comum. Em particular, ambas representam objetos que ocupam certo lugar, possuem um tamanho diferente de zero, podem ter inícios ou terminações de conexões e um *label* qualquer. E como podemos ver no diagrama de classe, a classe *Shape* possui um atributo para cada uma dessas propriedades respectivamente. E qualquer mudança nessas propriedades devem ser comunicadas a todos os *listeners*.

A classe *Connection* possui os atributos *source* e *target* que são do tipo Shapes e representam as formas em que essa conexão pode estar ligada. E assim como as outras classes do modelo estende a classe *ModelElement* herdando assim seus métodos .

A classe *ShapeDiagram* possui a funcionalidade de container mantendo uma coleção de formas e notificando os *listeners* sobre mudanças na coleção. E também é a

classe *root* da aplicação e por isso seu *controller* será o principal da aplicação, como será visto a seguir.

### 1.16.2. View

A *view* é na verdade a parte mais simples da aplicação isso porque a GEF fornece implementações padrão muito poderosas dos componentes visuais, utilizando a Draw2D temos que tudo que implementa a interface IFigure pode ser tratado como *view*.

Devido à simplicidade do modelo do editor, não tivemos que criar figuras que representariam nosso modelo, apenas usamos figuras pré-definidas. No nosso caso os objetos são representados pelas classes *RectangeFigure* e pela *Ellipse.* Apesar de sua *view* poder não ter qualquer referência ao *model* ou ao *controller,* ela deve ter um atributo visual para cada aspecto importante do modelo que o usuário possa querer inspecionar ou mudar.

A classe responsável por criar e inicializar o editor gráfico é a *ShapesEditor* . Ela estende a classe *GraphicalEditorWithFlyoutPalette*, que é a forma especializada de um editor gráfico, um tipo de *editor part*, equipado com uma paleta que hospeda as entradas das ferramentas. As classes que a estendem devem implementar dois métodos, *getPaletteRoot* e *getPalettePreferences*. O primeiro deve retornar a paleta raíz preenchida com as *tool entries,* que são tipos especializados de entradas da paleta capazes de instalar ferramentas no domínio do editor. As preferências da paleta, retornadas pelos segundo método, especifica se a paleta é visível ou *collapsed*, a localização e sua largura. A figura a seguir mostra a paleta de nosso editor gráfico.

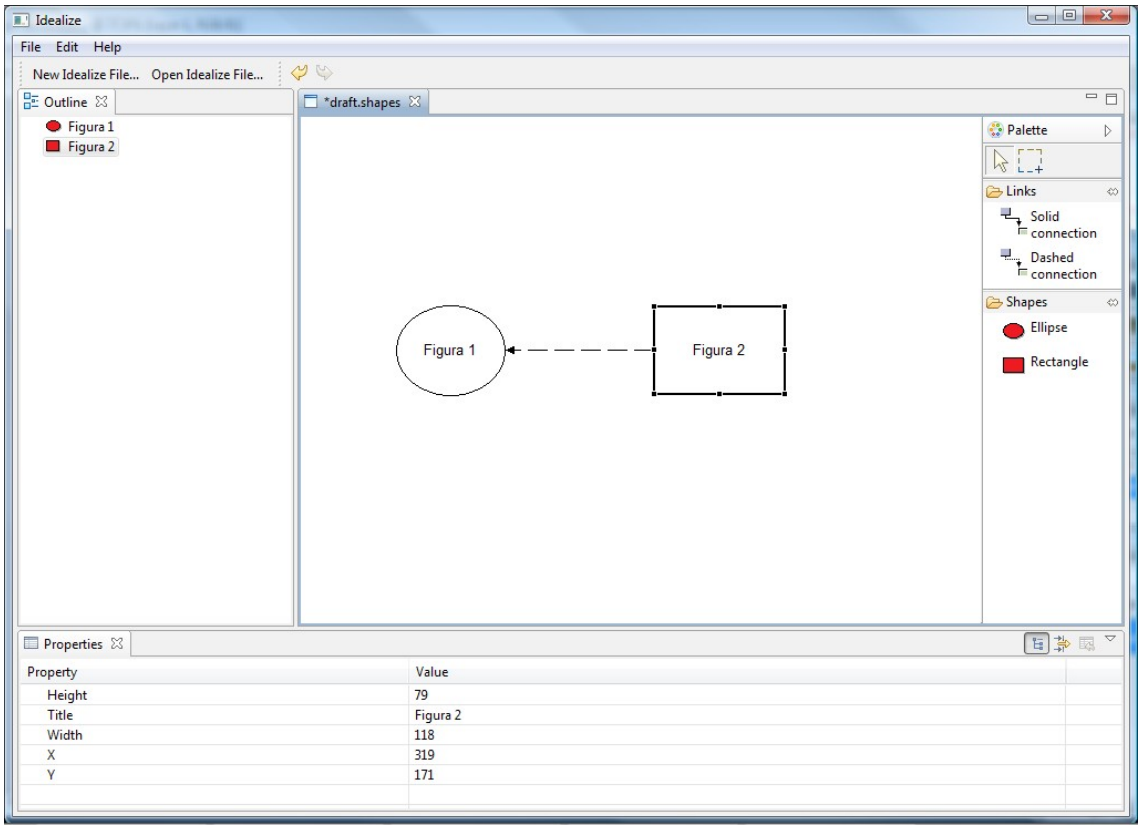


Figura 3.9: Tela de trabalho do plugin Idealize

### 1.16.3. Controller

O *controller*, ou EditParts como é conhecido no contexto da GEF, será a parte alvo de nosso estudo nesse item. Normalmente, em aplicações desse tipo, existiram pelo menos dois tipos de EditParts. A primeira é o *contents* ou *root* que é o elemento raiz do seu modelo, em nosso caso como foi dito é o *ShapeDiagram*. As outras são os verdadeiros *controllers*, no qual são responsáveis por controlar uma porção especifica do modelo, que não seja a *root*.

A diferença de responsabilidades entre as EditParts não são apenas refletidas em como elas manipulam o modelo, mas também no controle do comportamento gráfico. A *EditPart* raiz é responsável por criar o *Figure container* com seu *layout* inicial. Já as com função de controladores são responsáveis por prover os gráficos para suas respectivas associações dos elementos do modelo. Visto que, tanto o modelo quando a view não tem conhecimento uma da outra, é função do controlador “escutar” as mudanças no modelo e atualizar sua representação visual.

A *EditPartFactory* talvez seja a classe mais simples de todas. Basicamente, ela retorna a instância apropriada de uma *EditPart*, utilizando o padrão *Abstract Factory*.

A figura 3.10 mostra as classes do controle que foram implementadas na aplicação.

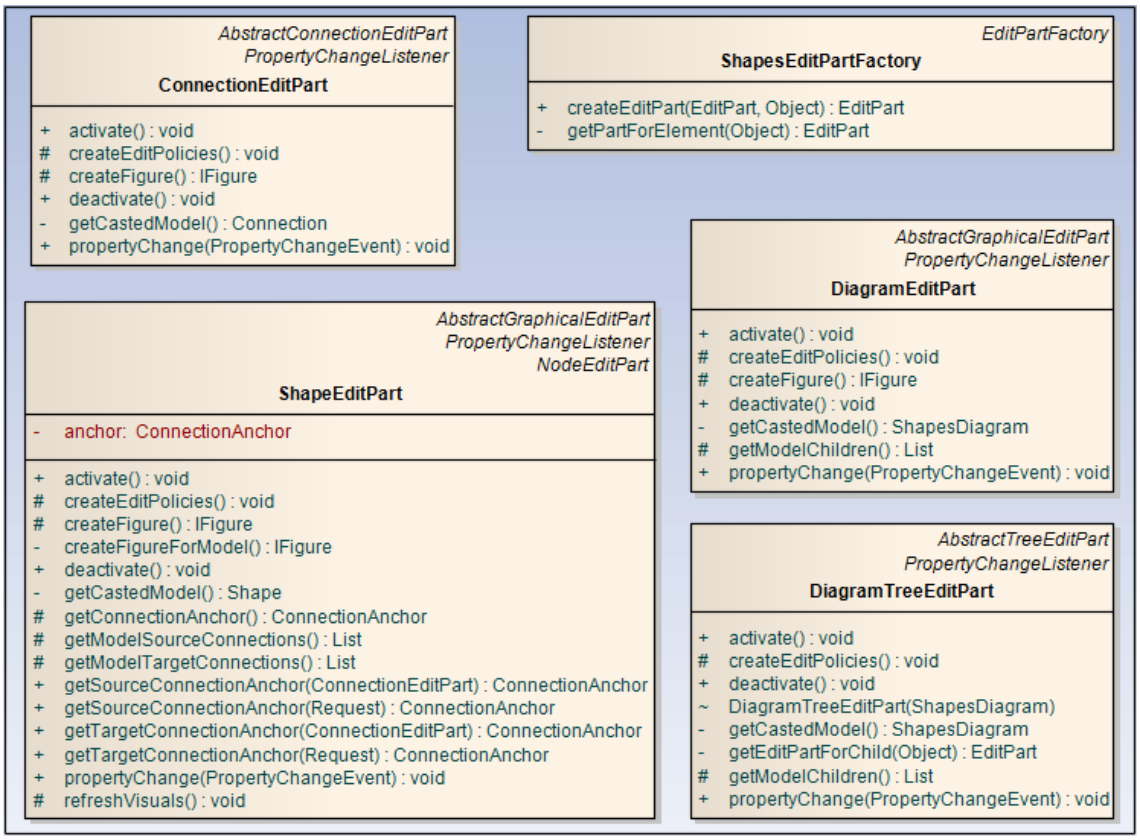


Figura 3.10: Diagrama de Classes Controller

As *EditParts* trabalham da seguinte forma, na maioria das vezes, as ferramentas mandam pedido para a *EditPart* na qual a figura esta sob o mouse. Por exemplo, se clicarmos em um retângulo, o controlador associado a ele, recebe um pedido de seleção ou de uma edição direta. Sem se preocupar se um ou mais controladores foram escolhidos como alvos do pedido, pois eles não manipularão os pedidos. Ao invés, delegam a tarefa para as *EditPolices* registradas. Cada política pede um comando para o pedido recebido, e não querendo tratar tal pedido pode retorna nulo. O mecanismo de ter as políticas tratando os pedidos ao invés das *EditParts*, permite que mantenhamos eles pequenos e altamente especializados. Isso, em retorno, significa facilidade de depuração e um código mais sustentável. O último pedaço do quebra-cabeça são os comandos. Ao invés de modificar o modelo diretamente, a GEF precisa que façamos isso com a ajuda

dos comandos. Cada comando deve implementar aplicar e desfazer mudanças ao modelo ou parte dele, pois assim os editores poderão automaticamente suportar as ações de fazer e desfazer alterações.

### 1.16.4. Actions

A questão de como são tratadas as *actions* na aplicação é importante para o entendimento da aplicação. O diagrama de seqüência a seguir ilustra o caminho que acontece quando o usuário seleciona opção deletar no editor.

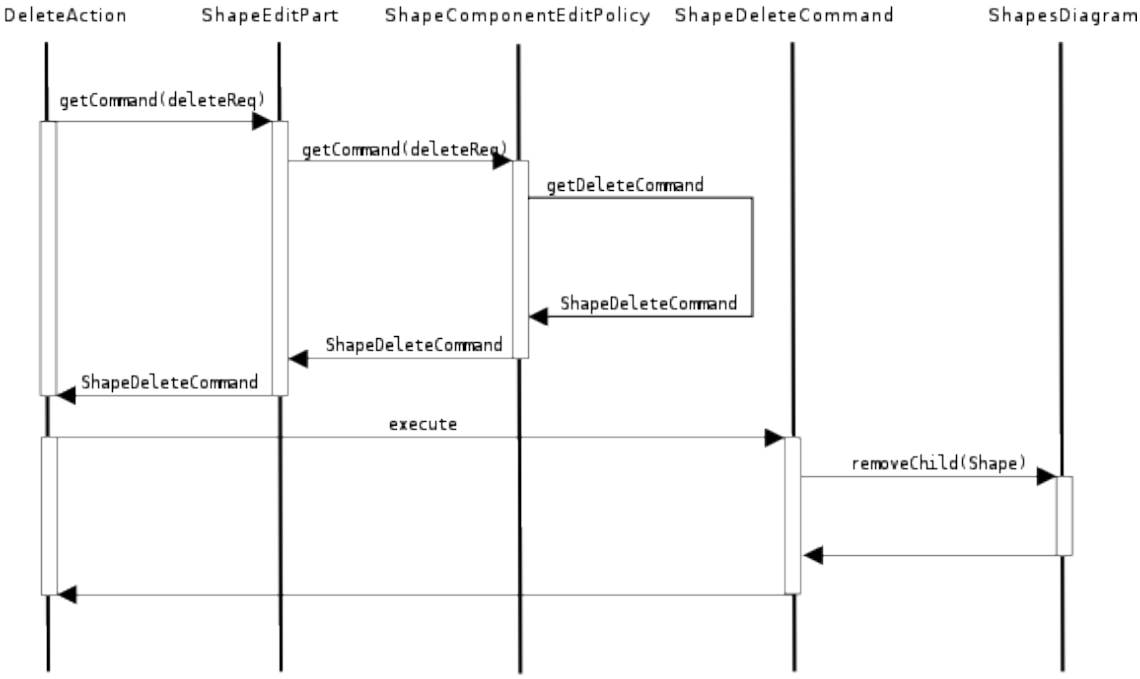


Figura 3.11: Seqüência da chamada para ação delete

Quando a ação de deletar é chamada, ela verifica se qualquer objeto selecionado no momento presente é uma instância da classe *EditPart.* Para cada objeto ela pedi um comando para a *EditPart,* e elas checam se alguma EditPolicies criada nelas entende e estão dispostas o pedido de *delete.* Para as formas, a classe *ShapeComponetEditPolicy* avisa que pode tratar o pedido, e quando o comando é solicitado retorna uma instancia do *ShapeDeleteCommand.* A action assim executa o comando, que remove a forma do diagrama. O diagrama dispara um *propertyChangeEvent* que é tratado pela *DiagramEditPart* que por ultimo conduz ate a figura representando a forma deletada e a remove do editor.

### 1.16.5. Outiline View

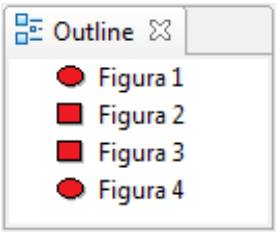


Figura 3.12: Outline

A view do Outline é usada para prover uma visão alternativa e muitas vezes mais sucinta dos dados editados. Nos editores do Java, ele é usado para mostrar os imports, variáveis, métodos das classes editadas sem entrar em detalhes do código. Sendo assim os editores gráficos também podem se beneficiar dessa utilidade. E em nosso caso utilizamos para mostrar os conteúdos editados em forma de *tree,* e por isso sua *EditPart* é implementada por uma *tree view.*

O Outline utiliza do mesmo mecanismo que foi empregado na *DiagramEditPart,* setando uma *EditPartFactory*. Em adição, a seleção de algum objeto na janela do editor é sincronizada com a do *Outiline*, utilizando uma classe utilitária da GEF que concilia o estado de seleção entre duas *EditParts*. A classe ShapeTreeEditPartFactory retorna uma instancia ou de uma *ShapeTreeEditPart* ou de uma *DiagramTreeEditPart* dependendo do tipo do modelo. Ambas implementam a interface PropertyChangeListener e reagem a mudanças ajustando sua representação visual do modelo.

Para adicionar o *Outline* à perspectiva, tivemos alterar a classe *Perspective*, que ficou da seguinte forma:

```
public class Perspective implements IPerspectiveFactory {

	public void createInitialLayout(IPageLayout layout) {
		String editorArea = layout.getEditorArea();
		//...
		layout.addView(IPageLayout.ID_OUTLINE, IPageLayout.LEFT, .20f, editorArea);
		//...
	}
}
```

Para imprimir o atributo *title* no Outline a classe *ShapeTreeEditPart* que é responsável por controlar as alterações no mesmo chama o seguinte método:

```
protected String getText() {
        return getCastedModel().toString();
}
```

### 1.16.6. Properties

Da mesma forma que o *Outline*, a view *Properties* é padrão do Eclipse, e foi utilizada para melhor visualização dos dados do objeto. Essa view, como todas que são padrão, possui um *listener* para seleção de objetos, para cada seleção verifica se aquele objeto implementa a interface *IPropertySource*. E caso esteja, usa os métodos da interface para interrogar o objeto ou objetos selecionados sobre suas propriedades e e exibi-las na forma de tabela na *Properties view*.

Graças à estrutura descrita acima, exibir as propriedades dos objetos editados é apenas questão de implementar os métodos da interface *IPropertySource*, em nosso contexto a classe que a implementa é a ModelElement, o que pertmite que todas as classes do modelo tenham esse comportamento. Os métodos utilizados nessa operação são *setPropertyValue* e *getPropertyValue,* que são usados para *settar* as propriedades e para recuperá-las respectivamente. A figura a seguir mostra as propriedades de um objeto selecionado dentro do editor.

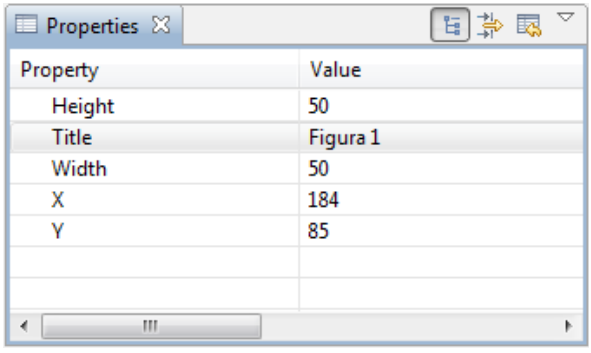


Figura 3.13: Properties View

### 1.16.7. Persistência

Como foi dito anteriormente utilizamos a serialização do objeto para persistir os dados existentes no editor em um arquivo binário. A seguir podemos ver o método *save* da classe *ShapesEditor*.

```
private void save(IPath path) {
		try {
			ObjectOutput out = new ObjectOutputStream(new FileOutputStream(path.to-
String()));
			out.writeObject(diagram);
			out.close();
			getCommandStack().markSaveLocation();
		} catch (IOException e) {
			e.printStackTrace();
		}
}
```

O método *writeObject* é responsável por escrever o estado do objeto de uma classe particular, para que o método readObject correspondente possa restaurá-lo. No caso do editor o objeto que precisamos salvar o estado é pertecente a classes *ShapesDiagram* que contem todos os dados que persistir.

Uma vez salvo, ao abrirmos o arquivo temos que recuperar as informações do arquivo binário e isso é feito no seguinte trecho de código da classe ShapesEditorInput:

```
ObjectInputStream in = new ObjectInputStream (new FileInputStream(file));
diagram = (ShapesDiagram) in.readObject();
```

O método readObject retorna um instancia de ShapesDiagram que é colocada no editor, permitindo que o usuário visualize o arquivo que foi aberto.

## 1.17. Plug-in Witty

O plug-in Witty apresentado neste projeto consiste de um *refactoring* parcial do aplicativo de mesmo nome. O componente Witty é formado por quatro segmentos que fazem parte da perspectiva que define o plug-in, são eles:

-*AddressView*: Barra de endereços, utilizada para disparar eventos;

-*TreeFolder*: *Folder* com quatro *views* fixas, sendo que cada uma delas implementa uma *Controltree;*

-*ConsoleView*: Utilizada para exibir informações ao usuário;

-Editor de textos: Um editor de textos funcional, com suporte a abrir, salvar e editar.

A versão apresentada é focada na camada visual da aplicação original, não possuindo ainda a lógica de negócios do aplicativo original implementada. Segue uma figura que mostra a aparência do plug-in.

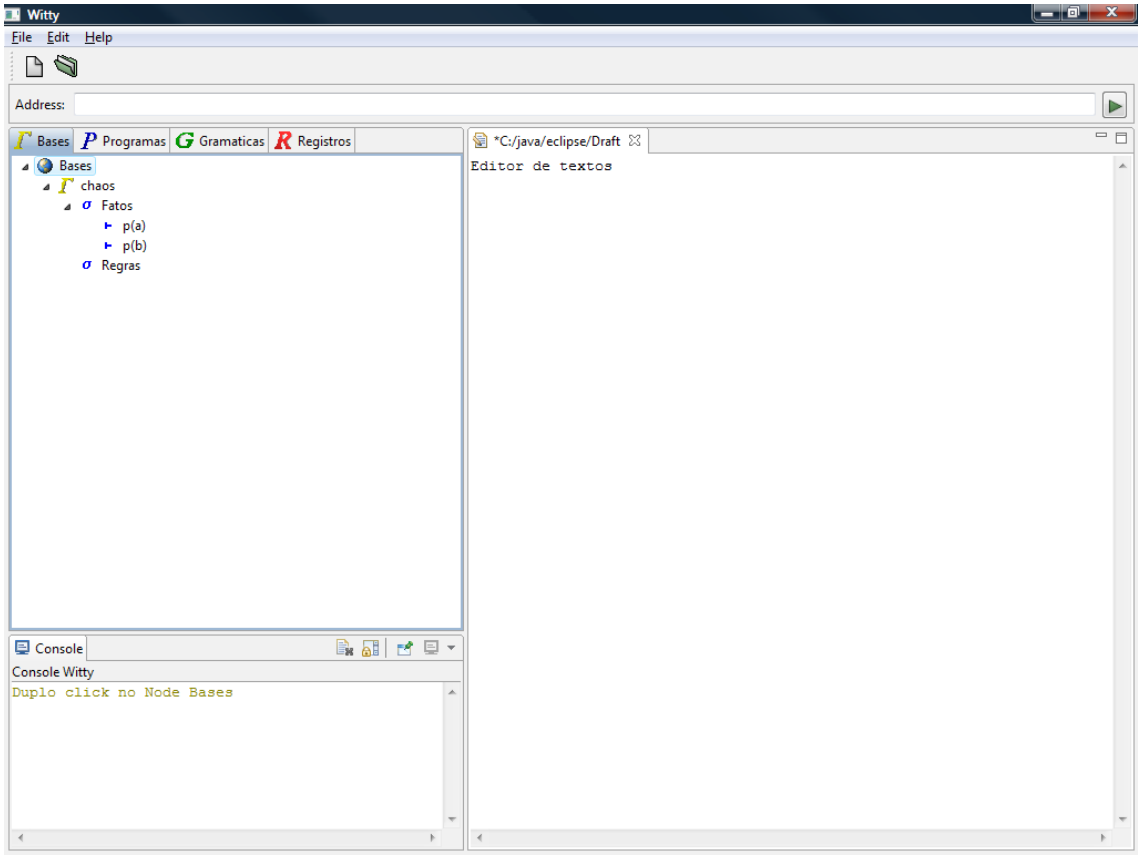


Figura 3.14: Visão geral do plug-in Witty

### 1.17.1. Perspectiva

Conforma citado na seção 3.1, cada plug-in anexado ao sistema principal deve ser uma perspectiva diferente. Desta forma, o componente Witty entrará no sistema como uma perspectiva de mesmo nome. Para que isto ocorra, deve ser criada uma *extension* para a perspectiva, conforme mostrado na seção 3.3 para o plug-in *Browser*.

A perspectiva deve ser configurada para atender aos requisitos visuais do plug-in, visto que a posição de cada componente é definida durante sua inicialização, sobrescrevendo o método *createInitialLayout*. Antes de definir como tal método foi utilizado, dois métodos auxiliares que foram utilizados em sua construção serão definidos.

O método *setCloseableAndMoveableFalse* faz com que uma *view* não possa ser fechada nem movida, enquanto o método *addFixedViewToFolder* adiciona uma *view* a um *folder*, que também não pode ser fechada ou movida. A implementação dos métodos vem a seguir:

```
	private void addFixedViewToFolder(IFolderLayout folder, String viewID, IPageLayout layout)
{
		folder.addView(viewID);
		IViewLayout viewLayout = layout.getViewLayout(viewID);
		this.setCloseableAndMoveableFalse(viewLayout);

	}

	private void setCloseableAndMoveableFalse(IViewLayout viewLayout) {
		viewLayout.setCloseable(false);
		viewLayout.setMoveable(false);
		}
}
```

Após a definição destes, o método *createInitialLayout* pode ser definido com mais clareza. Após configurar a *EditorArea* para ficar visível (devido ao editor de textos), são adicionados na ordem: *AddressView*, *Folder*(que comporta as *TreeViews*) e a *ConsoleView*, conforme explicitado em código abaixo:

```
public void createInitialLayout(IPageLayout layout) {

		String editorArea = layout.getEditorArea();
```

```
                layout.setEditorAreaVisible(true);
                layout.setFixed(true);

                // Add Address View
                layout.addStandaloneView(AddressView.ID, false, IPageLayout.TOP, 0.06f, editor-
Area);
                IViewLayout addressViewLayout = layout.getViewLayout(AddressView.ID);
                this.setCloseableAndMoveableFalse(addressViewLayout);

                // Add Folder
                IFolderLayout treeFolder = layout.createFolder(TreeFolder_ID, IPageLayout.LEFT,
0.35f, editorArea);
                // Add TreeViews in Folder
                this.addFixedViewToFolder(treeFolder, BasesView.ID, layout);
                this.addFixedViewToFolder(treeFolder, ProgramasView.ID, layout);
                this.addFixedViewToFolder(treeFolder, GramaticasView.ID, layout);
                this.addFixedViewToFolder(treeFolder, RegistrosView.ID, layout);

                // Add Console View          layout.addView("br.ufrj.del.witty.console.WittyCon-
soleView", IPageLayout.BOTTOM, 0.7f, TreeFolder_ID);
                IViewLayout viewConsoleLayout = layout.getViewLayout("br.ufrj.del.witty.con-
sole.WittyConsoleView");
                this.setCloseableAndMoveableFalse(viewConsoleLayout);

        }
```

É importante ressaltar, que a *AddressView* foi adicionada com um método diferente (*addAtandaloneView*) para que a visibilidade de seu título pudesse ser descartada.

### 1.17.2. Address View

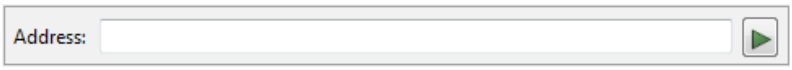


Figura 3.15: Visão da AddressView

É a *view* mais simples do componente Witty. Conforme figura acima, ela consiste apenas de um *label,* uma caixa de textos e um botão. Foi construída com SWT e seus componentes visuais são estáticos, sendo necessário para sua implementação somente uma classe.

Apesar da simplicidade, esta *view* foi de extrema importância para o sistema, pois funcionou como elemento disparador para todos os eventos de comunicação citados neste documento. Ao pressionar o botão *Enter* com o foco na caixa de textos ou acionar o botão, o método *fireGenericListener* é chamado, disparando os eventos.

### 1.17.3. TreeFolder

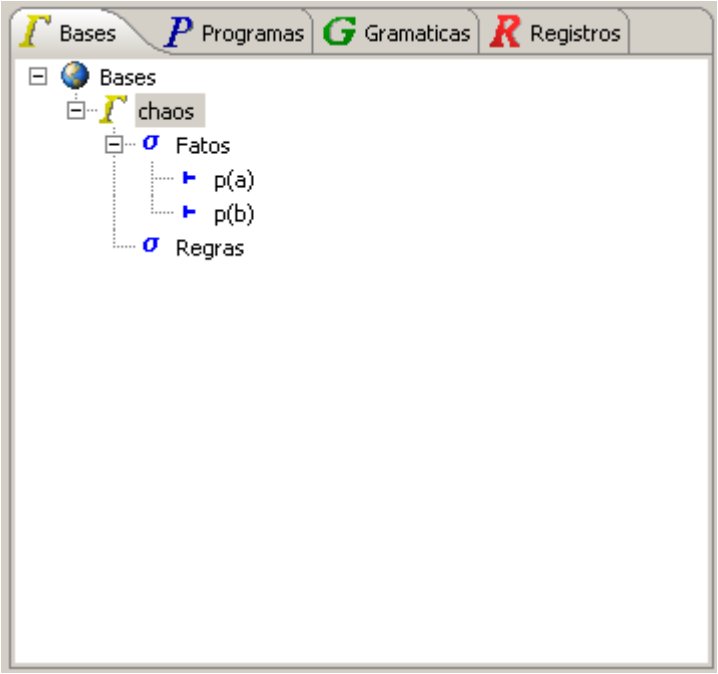


Figura 3.16: Visão do TreeFolder

O componente *TreeFolder* consiste de um espaço pré-definido pela aplicação, onde são inseridas quatro *views* fixas a saber: Bases, Programas, Gramáticas e Registros. Cada uma das *views* inseridas no *folder* implementa uma *control tree* com suas funcionalidades básicas, de contração, expansão, adição de filhos dentre outras.

A *tree* view é um dos elementos mais comuns de utilizados para apresentar dados ao usuário. Na SWT, este elemento é implementado utilizando o *Tree widget*. Seguindo o padrão de projeto MVC na *TreeViewer*, JFace simplifica a utilização do *Tree widget* delegando a tarefa de controle de conteúdo para o *ContentProvider* e a produção de *labels* para o *LabelProvider*. Desta forma, para uma simples apresentação da *view*, é necessário construir uma *TreeViewer*, e configurá-la com seu *LabelProvider* e *ContentProvider*.

Além dos componentes citados, é necessário a criação do modelo de dados, pelo qual os *providers* se orientarão para construir a *view*. Conforme a figura 3.16, podemos perceber a presença de quatro tipos diferentes de objeto e, portanto, cada um deles tem um modelo diferente. Inicialmente foi criado um modelo base para o nó, do qual os

outros estenderiam as funcionalidades, de acordo com o comportamento específico de cada um. Segue abaixo a implementação da classe base:

```
public class Node implements IAdaptable {

	private String name;

	private NodeParent parent;

	public Node(String name) {
		this.name = name;
	}

	public String getName() {
		return this.name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public void setParent(NodeParent nodeParent) {
		this.parent = nodeParent;
	}

	public NodeParent getParent() {
		return this.parent;
	}

	@Override
	public String toString() {
		return this.getName();
	}

	public Object getAdapter(Class key) {
		return null;
	}

	}
}
```

Pode-se perceber que é um componente muito simples, possuindo apenas um nome, e um nó pai. É importante lembrar que esta classe não possui filhos. A classe nó pai estende esta, adicionando a paternidade a suas funcionalidades. A codificação do componente *NodeParent* vem a seguir:

```
public class NodeParent extends Node {

	private List<Node> children;

	public NodeParent(String name) {
		super(name);
```

```
		this.children = new ArrayList<Node>();
	}

	public void addChild(Node child) {
		this.children.add(child);
		child.setParent(this);
	}

	public void removeChild(Node child) {
		this.children.remove(child);
		child.setParent(null);
	}

	public Node[] getChildren() {
		return this.children.toArray(new Node[this.children.size()]);
	}

	public boolean hasChildren() {
		return this.children.size() > 0;
	}
}
```

Após a implementação do nó pai, pode-se construir as duas classes restantes. Percebeu-se que ambas tem as mesmas funcionalidades de *NodeParent*, e por este motivo ambas estendem *NodeParent* sem alterações. Com a definição destas quatro classes, o modelo de dados está pronto.

Pode-se agora falar sobre a classe da *view*, responsável pela apresentação dos dados no formato treeview. Esta estende *ViewPart*, e dentro do método sobrescrito *createPartControl* é que se definem os *providers* da mesma. Segue abaixo o código padrão para criação de uma *control tree*:

```
public class View extends ViewPart {

	private TreeViewer viewer;
	public String visibleRootName;

	@Override
	public void createPartControl(Composite parent) {

		this.viewer = new TreeViewer(parent, SWT.MULTI | SWT.H_SCROLL |
SWT.V_SCROLL);
		this.viewer.setContentProvider(new TreeContentProvider(this.viewer, this.visibleRoot-
Name));
		this.viewer.setLabelProvider(new TreeLabelProvider());
		this.viewer.setInput(this.getViewSite());
		this.viewer.expandAll();
		this.getSite().setSelectionProvider(this.viewer);
	}

	@Override
	public void setFocus() {
		this.viewer.getControl().setFocus();
```

```
	}
	}
```

Conforme código acima, pode-se perceber os métodos *setContentProvider* e *setLabelProvider*, responsáveis respectivamente por configurar o *content* e o *label providers* para a *view* em questão. Segue uma breve descrição do funcionamento de cada um deles:

-*LabelProvider*: é responsável pela parte visual do componente. Ele verifica qual o tipo de objeto recebeu, e retorna a imagem correspondente, justificando a necessidade de classes diferentes estendendo *NodeParent*, conforme falado anteriormente.

-*ContentProvider*: responsável por verificar mudanças no modelo, tratar eventos e atualizar a *view* conforme necessário.

Devido à extrema semelhança entre os componentes do *folder*, foi criada uma classe base, fazendo com que cada *view* que estendesse tal componente, precisasse codificar apenas a única diferença, que é o nome da raiz da árvore. Para exemplificar tal feito, segue abaixo o código de um dos componentes, a *BasesView*:

```
public class BasesView extends View {
	public static final String ID = "br.ufrj.del.witty.BasesView";

	public BasesView() {
		this.visibleRootName = "Bases";
	}
}
```

O provedor de conteúdo é o lugar onde se adiciona a situação inicial da *control* tree. Por este motivo, é nesta classe que foi adicionado um *mock* de dados (para apresentar a *control tree* em funcionamento) e o registro da própria como ouvinte na *ListenersList* citada na seção 3.2. Segue a codificação deste *mock*, mostrando ambos procedimentos citados:

```
private void initialize() {

		ListenersList.getInstance().addListener(this, EventType.ADD_CHILD);
```

```
		this.visibleRoot = new NodeRoot(this.visibleRootName);
		/*
		 * root.addChild(p1); root.addChild(p2);
		 */

		this.invisibleRoot = new NodeParent("");
		this.invisibleRoot.addChild(this.visibleRoot);
		if (this.visibleRootName == "Bases") {
			Node pa = new Node("p(a)");
			Node pb = new Node("p(b)");
			NodeSpecialParent fatos = new NodeSpecialParent("Fatos");
			NodeSpecialParent regras = new NodeSpecialParent("Regras");
			NodeParent chaos = new NodeParent("chaos");

			this.visibleRoot.addChild(chaos);
			chaos.addChild(fatos);
			chaos.addChild(regras);
			fatos.addChild(pa);
			fatos.addChild(pb);

		}

	}
```

Como esta classe foi registrada para receber mensagens, a mesma deve implementar *GenericListener*, e consequentemente o evento *HandleEvent*, conforme código abaixo:

```
@Override
	public void handleEvent(GenericEvent evt) {
		if (evt.getEventType() == EventType.ADD_CHILD) {
			if (this.visibleRootName == "Bases") {
				String childName = (String) evt.getObject();
				this.visibleRoot.addChild(new NodeParent(childName));
				Perspective.writeConsoleMessage("Adicionando em Bases um filho
de nome \"" + childName + "\"",
						MessageKind.MSG_INFORMATION);
				this.viewer.refresh();
			}
		}
	}
```

O evento apresentado adiciona um filho ao nó raiz da *view* Bases, e escreve na console o nome do filho adicionado. Além deste evento, foi criado outro que imprime na console o nome de um nó qualquer onde seja dado um duplo click. A codificação de tal evento vem a seguir:

```
viewer.addDoubleClickListener(new IDoubleClickListener() {

			@Override
			public void doubleClick(DoubleClickEvent event) {
```

```
                                ISelection selection = View.this.viewer.getSelection();
                                Node node = (Node) ((IStructuredSelection) selection).getFirstEle-
ment();
                                String nodeName = node.getName();
                                Perspective.writeConsoleMessage("Duplo click no Node " + node-
Name, MessageKind.MSG_INFORMATION);
                        }

});
```

### 1.17.4. Console View

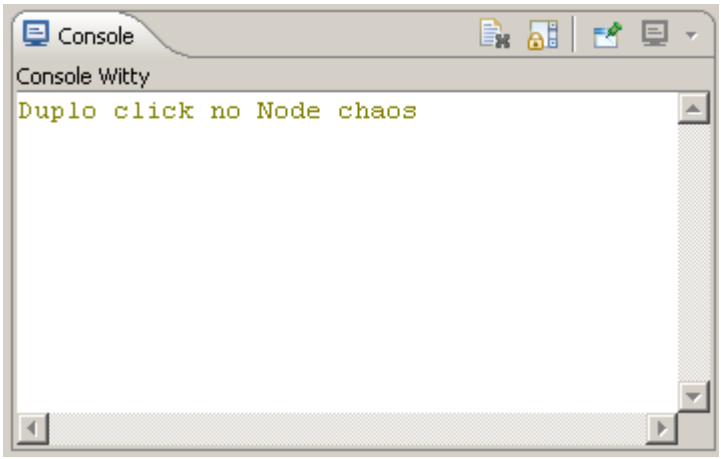


Figura 3.17: Visão da ConsoleView

A *ConsoleView* nada mais é do que uma adaptação feita da console nativa do eclipse. Foi decidido que não a utilizaríamos integralmente devido a algumas diferenças entre seu funcionamento e o esperado pela aplicação proposta, que seguem a seguir:

- pode ser fechada e trocada de lugar, atrapalhando o design fixo proposto

- podem ser adicionadas várias em um único folder, sendo que somente uma é necessária. Devido a isso, ela traria um menu de adição de novas consoles, induzindo o usuário ao erro.

Para corrigir estes problemas sem alterar a IDE do Eclipse, ao invés de simplesmente mostrar a console padrão (com o método *showView*), a *view* console do eclipse foi totalmente importada, e modificada, para atender ao Witty (adicionada normalmente, com o método *addView*, conforme seção 3.5.1).

Após ter adicionado a *view* ao ambiente, é necessário interagir com a mesma. Para esta finalidade, utilizou-se a classe *ConsoleDisplayMgr*, comumente utilizada para este fim, sendo customizada de acordo com a preferência do usuáriou ou desenvolvedor

envolvido. Seu funcionamento consiste em verificar qual a console ativa, e direcionar para ela as informações a serem exibidas. Como a utilização da console no Witty pode ser disparada por qualquer componente, foi decidido que a instância desta classe seria gerada na própria perspectiva. Segue abaixo o código responsável por instanciar a console do Witty:

```
	private final ConsoleDisplayMgr consoleDisplayMgr = new ConsoleDisplayMgr("Console
Witty");;
```

Após sua instanciação, foi declarado um método público também na perspectiva, de forma a ser acessada por qualquer componente do plug-in que necessitar utilizar a console. Segue abaixo a sua implementação:

```
public static void writeConsoleMessage(String message, MessageKind messageType) {
		consoleDisplayMgr.getDefault().println(message, messageType);
		}
```

É um método de utilização muito simples, tendo como primeiro argumento a mensagem a ser escrita na console, e como segundo argumento o tipo de mensagem. Atualmente, a classe possui três tipos de mensagem: MSG_INFORMATION, MSG_ERROR e MSG_WARNING, sendo cada uma delas escrita com uma cor diferente. Outros tipos de mensagem podem ser facilmente adicionados, assim como é permitida a troca das cores dos tipos de mensagem já existentes. A enumeração dos tipos de mensagem segue abaixo, seguida do método responsável pelas cores referentes a cada um deles.

```
public enum MessageKind {

		MSG_INFORMATION, MSG_ERROR, MSG_WARNING;

		private MessageKind() {

		}

		public String value() {
			return this.name();
		}

		public static MessageKind fromValue(String v) {
			return valueOf(v);
		}
		return valueOf(v);
		}
}
//-----------------------------------------------------------------
```

```
private MessageConsoleStream getNewMessageConsoleStream(
                        MessageKind messageType) {
                int swtColorId = SWT.COLOR_DARK_GREEN;

                switch (messageType) {
                case MSG_INFORMATION:
                        swtColorId = SWT.COLOR_DARK_YELLOW;
                        break;
                case MSG_ERROR:
                        swtColorId = SWT.COLOR_RED;
                        break;
                case MSG_WARNING:
                        swtColorId = SWT.COLOR_BLUE;
                        break;
                default:
                }

                MessageConsoleStream msgConsoleStream = this.getMessageConsole().newMes-
sageStream();
                msgConsoleStream.setColor(Display.getCurrent().
getSystemColor(swtColorId));
                return msgConsoleStream;
}
```

Para adicionar um novo tipo de mensagem, deve-se adicionar um elemento ao enum *MessageKind*, e no método *MessageConsoleStream* definir a cor com a qual será apresentada na console.

### 1.17.5. Editor de Textos

O primeiro passo para criar um editor é criar uma classe que estenda a classe *EditorPart* e implemente a interface *IEditorPart*. Em nosso caso para prover ao nosso editor muitos dos comportamentos padrão de um editor de texto, fizemos que nossa classe SimpleEditor estendesse a classe *AbstractTextEditor* que já satisfaz as condições descritas a cima.

Outro requisito é implementar o método createPartControl que descreve o visual do editor, justamente igual como é feito para as *views*. E também como as views, a EditorPart é projetada baseada em um modelo, que normalmente são baseados no conteúdo de um determinado *input*.

Os métodos a seguir devem ser implementados como parte da implementação do editor.

- createPartControl(Composite)

- doSave(IProgressMonitor)
- doSaveAs()
- init(IEditorSite, IEditorInput)
- isDirty()
- isSaveAsAllowed()
- setFocus()

Nem todos esses métodos foram implementados dentro da classe SimpleEditor, alguns desses são da classe *AbstracTextEditor* e são herdados pelo nosso editor.

A classe *PathEditorInput* é a nossa *input* , que será chamada no método init, se não houver erros no procedimento será aberto um novo editor. Ela também será responsável pela *flag* de alterações no editor, se sim, o método isDirty retorna verdadeiro e o método doSave será chamado quando o usuário selecionar alguma das opções de salvar. Caso queira que o usuário possa salvar em outro arquivo diferente do de entrada, seu editor apenas precisa sobrescrever o método isSaveAsAllowed. Uma vez que o método doSave foi chamado, ele executa o método doSaveAs diretamente com o caminho definido pelo usuário.

A seguir temos uma figura do editor de textos.

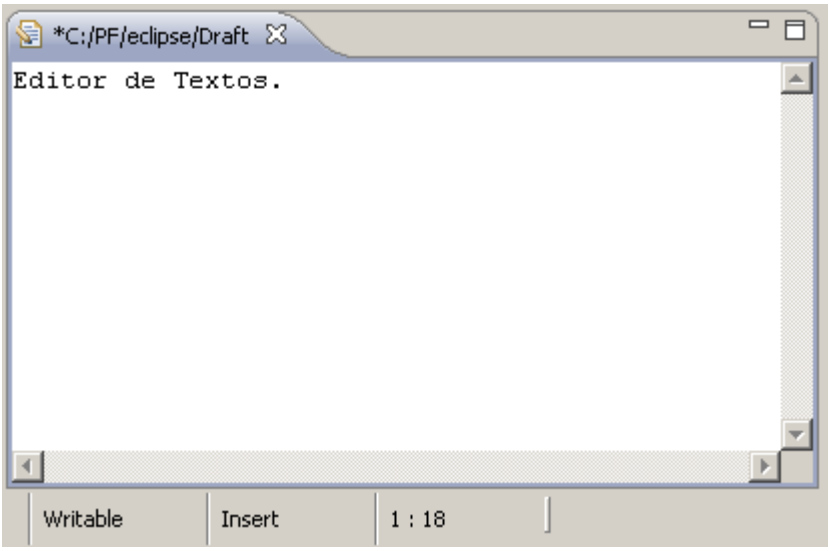


Figura 3.18: Visão do editor de textos e sua barra de status

A versão do editor aqui apresentada é a mais simples possível, e não recebe nenhum evento de outra parte da aplicação. Faz parte do escopo do projeto utilizar um

editor que suporte *syntax highlighting*, e devido a isso, foi criado um package dentro do projeto com um exemplo de editor (que no caso é um editor XML) que o estende, utilizando esta funcionalidade.

# Resultados

Um dos pontos mais relevantes do projeto apresentado é a troca de mensagens entre seus plug-ins independentes. O plug-in mediador utilizado para fazer esta conexão, foi explicitado na seção 3.2 , porém a forma em que esta integração acontece apontando duas formas possíveis de implementação, onde uma delas era visivelmente superior à outra em desempenho. O presente capítulo mostrará uma PoC (*Proof of Concept*) com este funcionamento, assim como apresentará o resultado de testes de carga realizados com as duas versões de implementação do plug-in.

## 1.18. Interação entre os plug-ins

Dentre os vários eventos disparados pelos componentes da aplicação, dois deles foram codificados de forma a implementar a característica de desconexão do padrão de projeto mediador. Ambos os eventos são disparados no mesmo momento pela mesma classe, e encaminhados para *views* diferentes. Apesar de um dos ouvintes fazer parte da mesma perspectiva que a classe que gerou o evento, e do outro estar em uma perspectiva diferente, as duas situações geradas possuem acoplamento nulo entre o gerador e o consumidor do evento.

O elemento disparador dos eventos citados, é a classe *AddressView* (do plug-in Witty), que comporta uma caixa de textos como objeto da ação. Em uma das situações, o texto é utilizado para criar um filho no root da *BasesView* do Witty, e na outra, para acionar o navegador do componente browser. Seguem abaixo os comandos utilizados para o disparo respectivamente da adição de um nó à árvore do Witty, e ao navegador do browser:

```
ListenersList.getInstance().fireGenericListener(new GenericEvent(this, GenericEvent.EventType.ADD_CHILD, AddressView.this.address.getText()));
ListenersList.getInstance().fireGenericListener(new GenericEvent(this, GenericEvent.EventType.FIRE_BROWSER, AddressView.this.address.getText()));
```

Apesar de os eventos já estarem sendo disparados, os componentes acima citados ainda não estão registrados para receber os eventos, e conseqüentemente não tomam conhecimento que eles estão ocorrendo. No caso do Witty, a classe responsável pela provisão de conteúdo das *views*, é a *TreeContentProvider*, e no caso do browser, a

própria *BrowserView*. Devido a isso, ambas as classes implementam a interface *GenericListener*, e durante sua inicialização, são registradas na *ListenersList* como ouvintes, conforme código a seguir:

```
ListenersList.getInstance().addListener(this, EventType.ADD_CHILD);
ListenersList.getInstance().addListener(this, EventType.FIRE_BROWSER);
```

Após esta estruturação dos eventos, fez-se um teste com o produto gerado pelo framework, apontando os plug-ins citados no capítulo 3 como dependência. Mesmo depois de retirar os componentes *Browser* e Idealize da pasta plug-ins da aplicação, o framework inicializa normalmente não apresentando as perspectivas retiradas. Além disso, o evento gerado na *AddressView* permanece alterando a *BasesView* sem qualquer tipo de complicação, de forma que esta situação serve como PoC para a utilização do padrão de projeto mediador com plug-ins do Eclipse RCP.

## 1.19. Teste de carga no Mediador de Eventos

Nesta seção, serão apresentados testes com variação de carga, utilizando os dois modelos de implementação da classe *ListenersList* citados no capítulo 3.2. O Método 1, dito “normal”, corresponde a situação na qual existe apenas uma lista para todos os ouvintes do sistema, e cada um deles recebe toda mensagem enviada, e verifica se é um receptor para a mesma ou não. O método 2, chamado simplesmente de otimizado, corresponde a situação na qual existe uma lista de ouvintes para cada tipo de evento disparado, sendo desta forma, os evcntos direcionados somente para os *listeners* que precisam receber o mesmo. Para realização dos mesmos, houve uma modificação nos eventos apresentados na seção anterior, de forma a ficarem condizentes com o caso de testes apresentado a seguir.

A idéia dos testes é simular o impacto no desempenho que ocorre no processo, caso existam vários listeners registrados na *ListenersList* que tratam tipos de evento diferentes do disparado. Para isto, foram necessárias as seguintes medidas:

-desativação do disparo do evento que acionava o *Browser*, direcionando um link para o navegador;

-desativação da funcionalidade do evento ADD_CHILD, mas mantendo o disparo do mesmo;

-registro de um número variado de ouvintes do tipo FIRE_BROWSER na classe singleton *ListenersList;*

-registro de um único *listener* do tipo ADD_CHILD ;

-simular a situação proposta com os dois modelos de implementação da classe *ListenersList.*

Com os procedimentos descritos acima, o cenário terá um evento do tipo ADD_CHILD disparado, um *listener* do tipo ADD_CHILD, e um número variado de ouvintes do tipo FIRE_BROWSER. Desta forma, deseja-se acompanhar a variação do tempo de resposta do sistema, ou seja, a diferença de tempo entre a chamada do método, e o fim do processamento do mesmo.

Com o modelo definido, deve-se escolher a escala em que os dados serão tomados, visto que o tempo de resposta de um único ciclo é muito pequeno. Sabendo deste detalhe, foi estipulado que 1.000 ciclos seriam estatisticamente suficientes para cada quantidade de ouvintes registrados. O Próximo passo é estipular a faixa em que o número de *listeners* irá variar, para obter uma quantidade razoável de dados. Foi definido que uma variação de 500 a 100.000 *listeners* do tipo FIRE_BROWSER seria suficiente, com um intervalo de 500 elementos por aferição.

Inicialmente, para realizar os procedimentos citados pensou-se em utilizar a própria *ViewConsole* do plug-in Witty, imprimindo o tempo de resposta de cada ciclo na mesma. Constatou-se durante seu uso, que a mesma não suporta um número excessivo de informação tal como as que são produzidas na presente simulação, impossibilitando a tomada de dados por este meio. Para resolver o problema, foi utilizada outra alternativa de implementação, gravando os dados em um arquivo texto, retirando então, a limitação imposta anteriormente pela console.

Para não existir a necessidade de repetir os testes várias vezes, foram feitos *loops* intercalados, para:

- repetir o ciclo básico de disparo 1000 vezes, para a aferição estatística do tempo de resposta;

- adicionar à classe *ListenersList* 500 ouvintes (para respeitar a variação de elementos a cada 500);

-repetir os dois ciclos anteriores 200 vezes, para que os 500 *listeners* iniciais se tornem 100.000 no fim do processo.

Esta seqüência foi repetida duas vezes, uma para cada modelo de implementação proposto. O método utilizado para medir os intervalos de tempo, foi o System.nanoTime(), que registra um tempo em nanosegundos, em relação a um mesmo referencial aleatório. Segue abaixo a codificação responsável pelos loops citados, e a gravação do arquivo com os dados:

```
Button b = new Button(parent, SWT.PUSH);
b.addSelectionListener(new SelectionAdapter() {
@Override
public void widgetSelected(SelectionEvent e)
{
PrintWriter pw;
try
{
        pw = new PrintWriter(new FileOutputStream("C:" + "/" + "Testes" + ".txt", false), false);

        int indexListeners = 0;
        while (indexListeners < 200)
        {
                int index = 0;
                pw.print(500 + indexListeners * 500 + "- ");
                long startTime = System.nanoTime();
                while (index < 1000)
                {
                        ListenersList.getInstance().fireGenericListener(new GenericEvent(this, Gen-
                        ericEvent.EventType.ADD_CHILD, AddressView.this.address.getText()));
                        index++;
                }
                long endTime = System.nanoTime();
                pw.println(String.valueOf(endTime - startTime));
                int indexNumber = 0;
                while (indexNumber < 500 && indexListeners < 200)
                {
                        ListenersList.getInstance().addListener(new
                        BrowserView(), EventType.FIRE_BROWSER);
                        indexNumber++;
                }
                indexListeners++;
}
pw.close();
} catch (FileNotFoundException e1) {
e1.printStackTrace();
}
}
});
```

Após a tomada dos dados, foram construídos dois gráficos, um para cada modelo de implementação, consolidando os resultados. Seguem abaixo respectivamente a implementação comum e a otimizada:

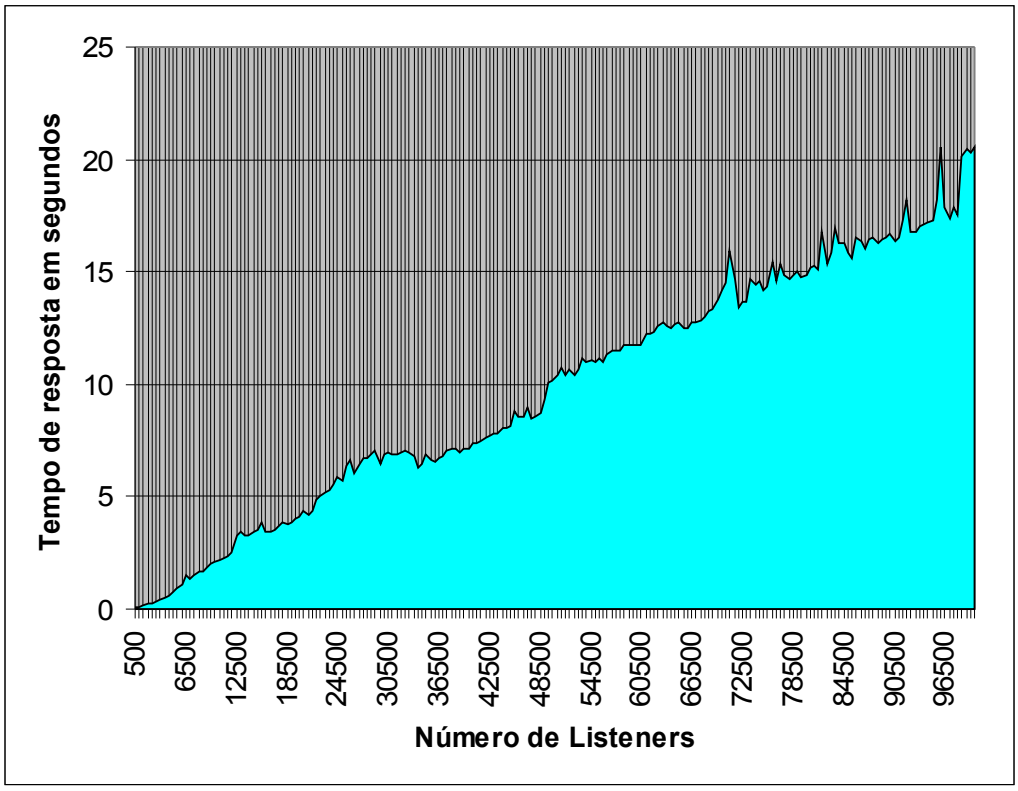


Figura 4.1: Modelo de implementação “normal” (método 1)

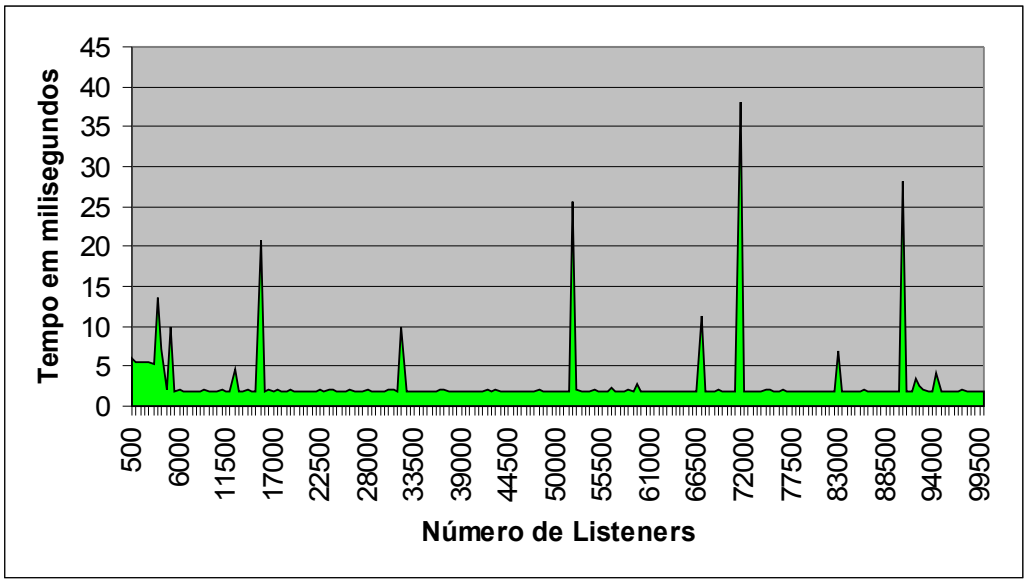


Figura 4.2: Modelo de implementação otimizado (método 2)

Conforme pudemos ver, ambos os gráficos se comportaram conforme o esperado:

-No modelo de implementação normal, existe um aumento aproximadamente linear no tempo de resposta, de acordo com a quantidade de *listeners* no sistema, independente do tipo de evento disparado;

-No modelo de implementação otimizado, o tempo independe do número de ouvintes total, permanecendo praticamente constante para qualquer quantidade de *listeners* não tratados registrados.

# Conclusão

Atualmente não se concebe um processo de desenvolvimento de software bem projetado sem a utilização da orientação a objetos, pois esta permite agregar um maior nível de qualidade aos sistemas desenvolvidos sob seus paradigmas. Entretanto utilizar somente a orientação a objeto não maximiza a qualidade do software. O uso de padrões de projeto propicia a construção de aplicações e ou estruturas de código de forma flexível e a documentação de soluções reaproveitáveis.

Neste projeto, assim como a independência entre seus componentes, a escalabilidade, reusabilidade e qualidade do software foram prioridades. Para atingir tais objetivos, além de utilizar a IDE do Eclipse RCP, foi necessária a implementação de vários padrões de projeto.

Através dos padrões de projeto é possível identificar os pontos comuns entre duas soluções diferentes para um mesmo problema. Conhecer esses pontos comuns nos permite desenvolver soluções cada vez melhores e mais eficientes que podem ser reutilizadas, permitindo, assim, o avanço do conhecimento humano.

O presente projeto teve como objetivo apresentar um *refactoring* de parte de um *software* já existente. Foi realizado um estudo sobre as principais características dos plug-ins, enfatizando o Eclipse RCP como *framework* de desenvolvimento, e a utilização dos Padrões de Projeto. Após a implementação dos plug-ins, foram realizados testes de comunicação entre os mesmos, e testes de carga sobre diferentes soluções para comunicação inter plug-ins. A codificação otimizada da classe moderadora dos eventos apresentou um ganho de desempenho considerável, principalmente em sistemas com um grande número de componentes que trocam mensagens, quando utilizado o método otimizado.

Entende-se que a próxima etapa de trabalho seria o refactoring completo e funcional das aplicações Witty e Idealize. A estrutura de *framework* desenvolvida neste trabalho oferecerá as condições necessárias a este desenvolvimento.

# Bibliografia


[1] GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. *Design Patterns: Elements of Reusable Object-Oriented Sofware*. Bookman, 2004.

[2] ALEXANDER, Christopher. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1987.

[3] DARLA, D., Revista TI Digital, Editora Arteccom, nº 3 , pp. 36-43, maio, 2009.

[4] FREEMAN, Eric; FREEMAN, Elisabeth. *Head First Design Patterns* . O'Reilly, 2004.

[5] COHEN, Marcelo. Notas de Aula de Laboratório de Programação II, PUC Rio Grande do Sul, 10/05/2009, <http://www.inf.pucrs.br/~flash/lapro2/lapro2_eventos.pdf>.

[6] WIKIPEDIA FOUNDATION, Inc. Software Framework, 11/05/2009, <http://en.wikipedia.org/wiki/Software_framework>.

[7] MICROSOFT CORPORATION. Model-View-Controller, Microsoft Developer Network, 10/05/2009, <http://msdn.microsoft.com/en-us/library/ms978748.aspx>.

[8] SOUZA, Vitor E., Curso de – Padrões de Projeto Módulo 4: Padrões de Comportamento, 23/05/2009, <http://www.disi.unitn.it/~vitorsouza/sites/default/files/DesignPatterns04.pdf>.

[9] DOEDERLEIN, Osvaldo Pinali. Bytecode: Escondendo e Revelando. Java Magazine, 66, DevMedia Group, p.52-61, 2009

[10] OPEN SOURCE. University Meetup, 23/05/2009, <http://osum.sun.com/group/javabrasil>.

[11] SUN MICROSYSTEMS, Inc. 10/05/2009, <http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/app-arch/app-arch2.html>..

[12] MOLSKI, Fernando Ricardo. Customização de Software com ênfase na Arquitetura MVC E STRUTS, 23/05/2009, <http://www.devmedia.com.br/articles/viewcomp.asp?comp=8830>.

[13] BESEN, Renato. Padrão de Projeto Mediador, 23/05/2009, <http://www.inf.ufsc.br/~renatob/20071/engenharia_de_software/Design%20Patterns%20-%20Mediator.pdf>.

[14] ECLIPSECON 2008, 20/04/2009 <http://www.eclipsecon.org/2008>..