6
Aplicação

Para obter as funcionalidades propostas, uma etapa fundamental precisa ser cumprida. Esta etapa é a determinação dos pesos das arestas do grafo orientado de classes. As arestas são do tipo: abstração, generalização, dependência, realização, associação, composição e agregação.

Atribui-se um peso para cada um dos tipos de relacionamento. Esta atribuição foi concebida com foco na experimentação. Uma base amostral de modelos de diagrama de classe UML foi projetada para realizar o confronto dos componentes e dos *core concepts* extraídos pela ferramenta *AnalisadorUML* com os respectivos componentes e *core concepts* reais dos projetos.

Modificaram-se os pesos das arestas até que existisse uma sintonia fina do resultado extraído com o esperado.

A base amostral com os seus respectivos resultados e a avaliação dos pesos atribuídos são tópicos deste capítulo.

6.1.
Base Amostral

Para definição dos pesos dos relacionamentos e o desenvolvimento de testes do estudo proposto, uma base amostral foi elaborada.

Esta base consiste em quinze modelos de diagramas de classe:

- Cinco modelos que combinam *design patterns*.
- Cinco modelos de projetos de complexidade elevada.
- Cinco modelos de menor complexidade, com maior nível de abstração.

Todos os diagramas foram criados na ferramenta JUDE *Astah Professional 6.5* devido à facilidade de exportação do XMI e suporte a todos os tipos de relacionamentos previamente discutidos.

6.1.1.
Modelos de Design Patterns

Os *design patterns* têm complexidade baixa e resultados bem definidos, o que é interessante para uma primeira etapa de teste da ferramenta e sintonia grossa dos pesos dos relacionamentos.

Todos os modelos de *design patterns* propostos possuem ordenações de importância de classe e componentes facilmente detectáveis. No que tange às métricas, como são aplicações de fórmulas, não há necessidade de uma avaliação criteriosa.

No intuito de elaborar testes mais completos na ferramenta, os modelos foram compostos de dois ou mais *design patterns*. As composições desenvolvidas estão descritas nas seções a seguir. Estas seções descrevem os resultados esperados na análise dos modelos de *design patterns*.

Antes de analisar caso a caso, é esperado um padrão para a obtenção dos *core concepts* de todas as composições. As classes centrais de cada *design pattern* e as classes que contêm relacionamentos com classes de *design patterns* diferentes devem conter maior grau de importância.

O padrão para a componentização é a atribuição de um grupo de classes para o seu respectivo *design pattern*. Para isto, os projetos foram criados com o número de grupos igual ao número de *design patterns* compostos.

6.1.1.1.
Abstract Factory e Strategy

Segundo Gamma et al. (1995), o *Abstract Factory* fornece uma forma de encapsular um grupo de fábricas individuais que têm um tema em comum, sem especificar suas classes concretas. Uma classe *Cliente* cria uma implementação concreta da fábrica abstrata e usa as interfaces genéricas para criar os objetos concretos. A fábrica é uma classe responsável por construir variados objetos. A intenção é isolar a criação de objetos de seu uso e criar famílias de objetos relacionados sem ter que depender de suas classes concretas.

Ainda de acordo com Gamma et al. (1995), o *Strategy* representa operações a serem realizadas sobre os elementos de uma estrutura de objetos. Este padrão permite definir novas operações em interfaces sem alterar as classes dos elementos implementadores.

A composição *Abstract Factory* e *Strategy* é traduzida no seguinte modelo:

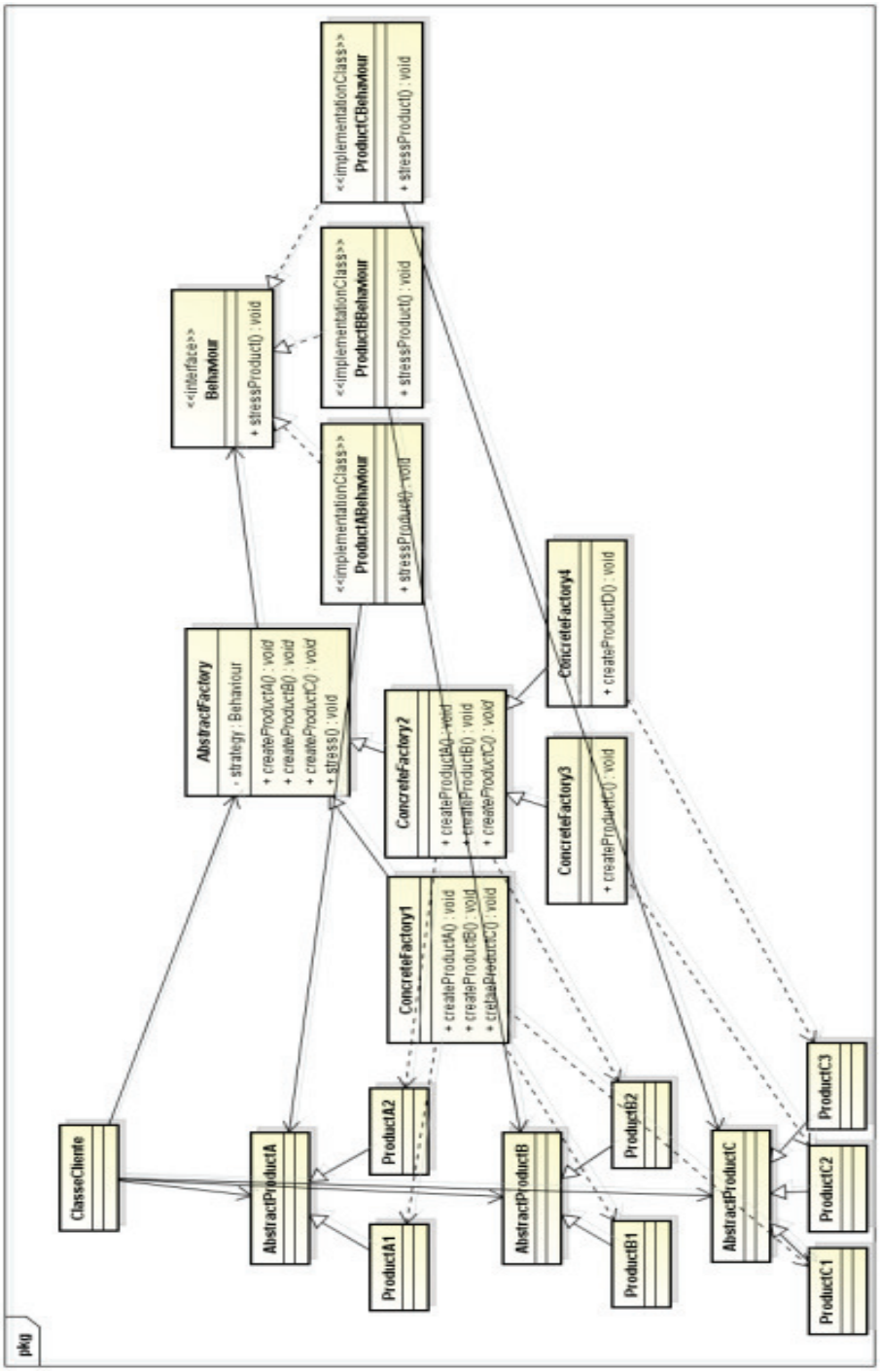

Figura 14 - Modelo da composição *Abstract Factory* e *Strategy*.

Pela análise da composição *Abstract Factory* e *Strategy*, descrita na Figura 14, as classes mais prováveis candidatas a *core concepts* seriam: *Abstract Factory* (representa a fábrica do padrão *AbstractFactory*), *ConcreteFactory1* (representa a implementação da fábrica abstrata e se conecta às classes que serão instanciadas), *ConcreteFactory2* (análogo ao *ConcreteFactory1*, com a diferença do fato desta

classe ser pai de *ConcreteFactory3* e *ConcreteFactory4*) e *Behavior* (representa a distribuição dos comportamentos do padrão *Strategy*).

Já os componentes esperados seriam: um componente com as classes do *Strategy* (*Behaviour*, *ProductABehaviour*, *ProductBBehaviour* e *ProductCBehaviour*), outro componente com parte do *AbstractFactory* (*AbstractFactory*, *ConcreteFactory2*, *ConcreteFactory3*, *ConcreteFactory4* e suas respectivas instâncias de produto) e o último componente com a segunta parte do *AbstractFactory* (*ConcreteFactory1* e suas instâncias de produto).

6.1.1.2. Bridge, Mediator e Chain of Responsability

Segundo Gamma et al. (1995), o *Bridge* é um padrão que se destina a desacoplar uma abstração da sua implementação, de modo que os dois possam variar de forma independente. O *Bridge* usa encapsulamento, agregação e herança para separar as responsabilidades em diferentes classes. Já o *Mediator* define um objeto que encapsula o comportamento de interação entre um conjunto de objetos através do objeto mediador. Os objetos não se comunicam mais diretamente. Ao invés disto, utilizam o objeto mediador para realizar tal tarefa. Isto reduz a dependência entre objetos e também o acoplamento.

O terceiro design pattern da composição é o *Chain of Responsability*. Pela definição de Gamma et al. (1995), ele é um padrão que consiste em uma fonte de objetos de comando e uma série de objetos de processamento. Cada objeto de processamento contém a lógica que define quais tipos de objetos de comando ele pode manipular e o resto do processamento é repassado para o objeto de processamento seguinte na cadeia até o término desta cadeia.

A composição *Bridge*, *Mediator* e *Chain of Responsability* é traduzida no seguinte modelo:

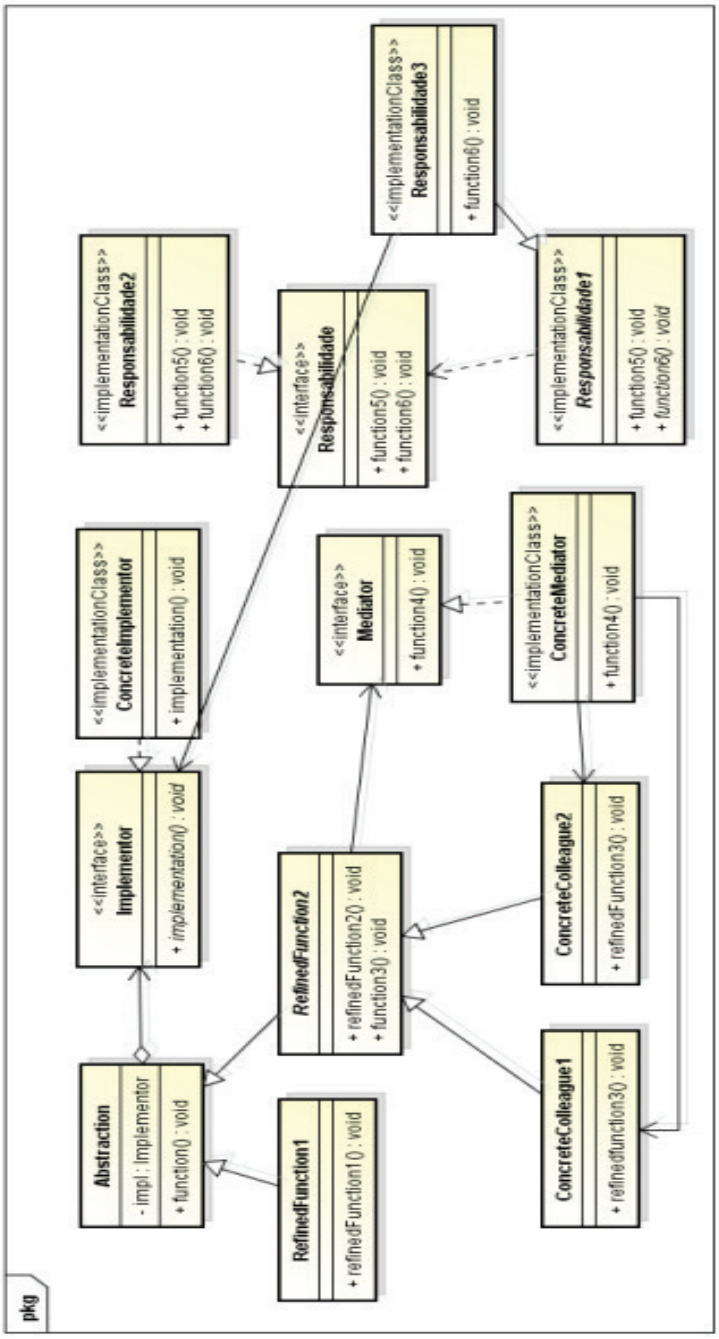

Figura 15 - Modelo da composiçao *Bridge*, *Mediator* e *Chain of Responsability*.

A análise da composição *Bridge*, *Mediator* e *Chain Of Responsability*, descrita na Figura 15, espera que as classes mais importantes do diagrama sejam: *Abstraction* (classe abstrata central do padrão *Bridge*), *Implementor* (classe interface que responde à *Abstraction*), *RefinedFunction2* (classe que herda e refina *Abstraction* e que neste modelo faz a associação com o padrão *Mediator* no papel de *Colleague*), *Responsabilidade* (classe que gera o padrão *Chain of Responsability*) e *Responsabilidade3* (implementa parte do padrão *Chain Of Responsability* e faz a associação com o padrão *Bridge*).

Os componentes esperados são: um componente contendo as classes do *Chain Of Responsability* (*Responsabilidade*, *Responsabilidade1*, *Responsabilidade2* e *Responsabilidade3*), outro componente com as classes do *Bridge* (*Abstraction*, *Implementor*, *ConcreteImplementor*, *RefinedFunction1* e *RefinedFunction2*) e o último componente com as classes do *Mediator* (*Mediator* e *ConcreteMediator*). As classes *ConcreteColleague1*, *ConcreteColleague2* pertencerão ao componente onde se encontra a classe *RefinedFunction2*, pois esta classe faz o papel do *Colleague* do padrão *Mediator*, ou pertencerão ao componente com as classes do padrão *Mediator*, pois as *ConcreteColleagues* pertencem a este padrão.

6.1.1.3. Command, Adapter e Proxy

Segundo Gamma et al. (1995), o *Command* é um padrão comportamental em que um objeto é usado para representar e sintetizar as informações para chamar um método. Estas informações incluem o nome do método, o objeto que possui o método e os valores para os parâmetros do método. Objetos de comando tornam mais fáceis a construção de componentes gerais que precisam delegar ou executar chamadas de método em um momento sem a necessidade de conhecer a classe do método ou os seus parâmetros. Os quatro termos principais do padrão são o comando, o receptor, o invocador e o cliente.

O *Adapter* é definido por Gamma et al. (1995) como um padrão que traduz a interface para uma classe em uma interface compatível para o cliente. Um adaptador permite que classes que não possuem interfaces e chamadas compatíveis possam trabalhar juntas através do fornecimento da adaptação da interface original para interface compatível ao cliente. Já o *Proxy* é uma classe que funciona como uma interface que delega a implementação para um objeto.

A composição *Command*, *Adapter* e *Proxy* é traduzida no seguinte modelo:

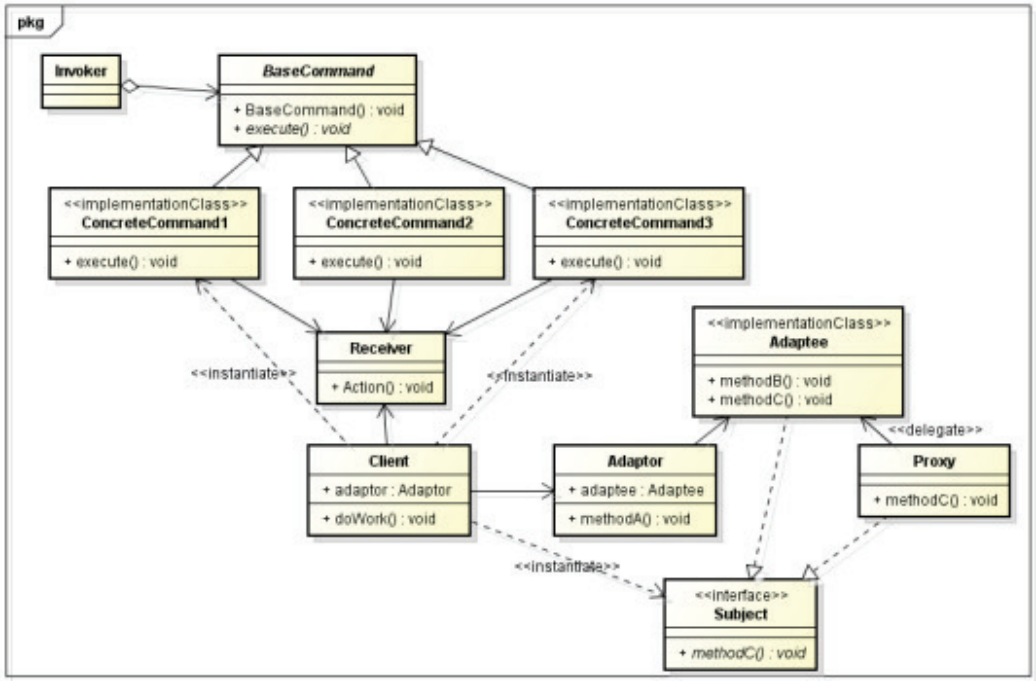

Figura 16 - Modelo da composição *Command*, *Adapter* e *Proxy*.

Na análise da composição *Command*, *Adapter* e *Proxy*, decrita na Figura 16, as seguintes classes seriam os conceitos centrais: *Client* (classe cliente que liga os padrões *Adapter*, o *Command* e o *Proxy*), *BaseCommand* (classe pai do padrão *Command*), *Subject* (classe de ligação dos padrões *Adapter*, *Proxy* e da classe cliente) e *ConcreteCommand1* ou *ConcreteCommand3* (classes instanciadas pelo cliente e que implementam uma função do padrão *Command*). Com relação à escolha das classes *ConcreteCommand1* ou *ConcreteCommand2* para conceito central do diagrama, como elas exercem a mesma função dentro do diagrama, o algoritmo de FloydWarshall escolherá uma delas para a obtenção dos menores caminhos, de preferência a primeira a ser visitada pelo algoritmo. Por isso apenas uma delas será considerada conceito central.

Do ponto de vista da componentização, o primeiro componente seria composto por classes relacionadas ao *Command* (*Invoker*, *BaseCommand*, *ConcreteCommand1*, *ConcreteCommand2* e *ConcreteCommand3*), o segundo componente relacionado ao *Proxy* e ao *Adapter* (*Proxy*, *Subject*, *Adaptee* e *Adapter*). A classe *Receiver* ficará avulsa em outro componente e a classe *Client* pertencerá ao primeiro ou ao segundo componente, pois ela se liga aos dois grupos da mesma maneira.

6.1.1.4.
Factory Method, Facade e Visitor

Segundo Gamma et al. (1995), o *Factory Method* é um padrão de criação para implementar o conceito de fábricas e lida com o problema da criação de objetos sem especificar a classe exata do objeto a ser criado. O *Facade* é utilizado como um *wrapper* para transformar subsistemas complexos em uma interface simples através do fornecimento da transformação de um conjunto de interfaces dentro do subsistema em apenas uma interface. Desta forma, o *Facade* aumenta o desacoplamento entre os diferentes subsistemas dentro do sistema.

O *Visitor* é definido por Gamma et al. (1995) como uma forma de separar um algoritmo da estrutura de objetos em que ele opera. Desta forma, é possível adicionar novas operações sem modificar a estrutura de objetos existente.

A composição *Factory Method*, *Facade* e *Visitor* é traduzida no seguinte modelo:

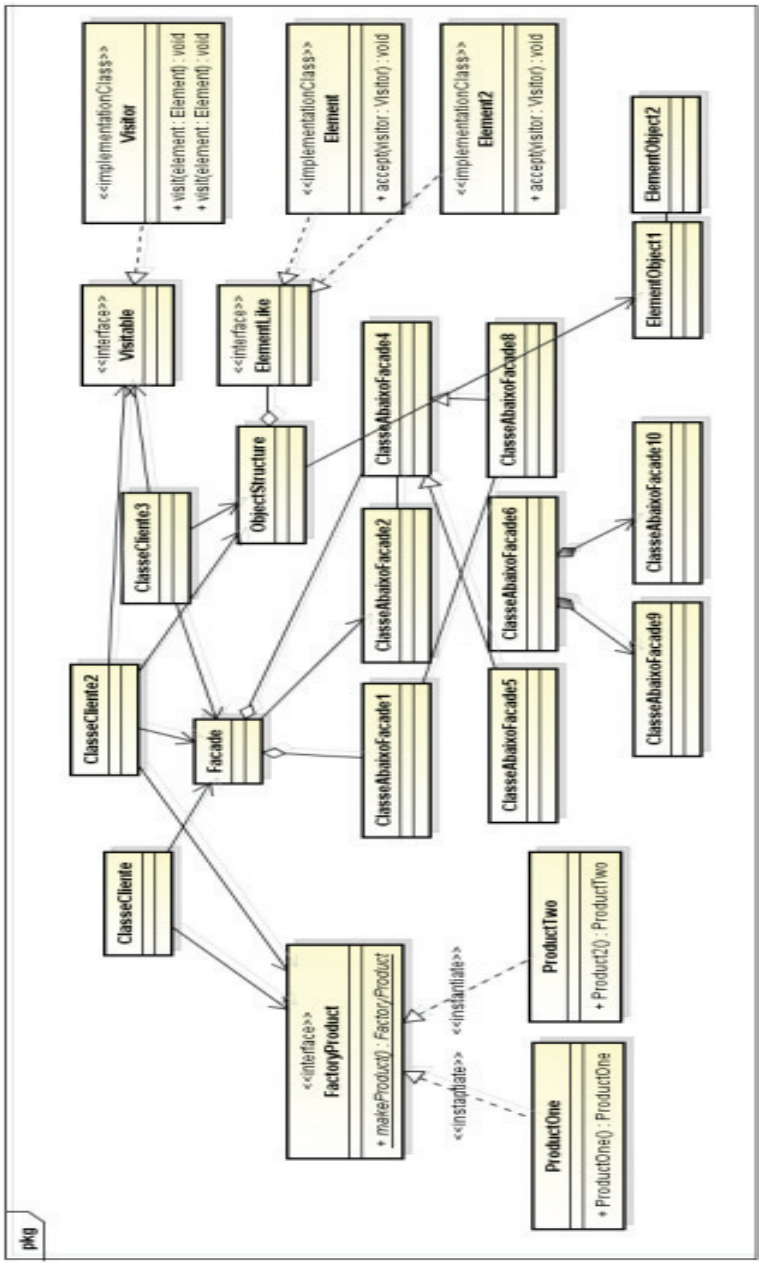

Figura 17 - Modelo da composição Factory Method, Facade e Visitor.

Para a análise da composição *Factory Method*, *Facade* e *Visitor*, esperam-se os seguintes *core concepts*: *ClasseCliente2* (representa a conexão dos três padrões), *Facade* (classe central do padrão *Facade*), *FactoryProduct* (classe central do padrão *Factory Method*), *Visitable* e *ElementLike* (classes centrais do padrão *Visitor*), *ClasseCliente* (representa a conexão do padrão *Factory Method* com o padrão *Facade*), *ObjectStructure* (representa a conexão da *ClasseCliente2* com o padrão *Visitor*).

Esta composição conteria um componente com classes do *Factory Method* (*FactoryProduct*, *ProductOne* e *ProductTwo*), outro componente com todas as classes do *Facade* (*Facade* e todas as *ClassesAbaixoFacade*) e um terceiro

componente com classes do *Visitor* (*Element*, *ElementoObject1*, *ElementObject2*, *Element2*, *ObjectStructure* e *ElementLike*). As classes clientes variam nos componentes.

6.1.1.5. Observer, Flyweight e Adapter

Segundo Gamma et al. (1995), o *Observer* é um padrão em que um objeto *Subject* mantém uma lista de interfaces dependendentes, chamados *Observers*, para notificá-los, automaticamente, perante as eventuais mudanças de estado. Os observadores, por sua vez, obrigam a implementação das notificações em objetos observadores concretos (*ConcreteObservers*).

Ainda de acordo com Gamma et al. (1995), o *Flyweight* é um objeto que minimiza o uso de memória pelo compartilhamento de dados com objetos semelhantes. Desta maneira, as definições dos estados internos em comum entre objetos são alocados no objeto *Flyweight*, instanciado via fábrica. O padrão *Adapter* foi definido anteriormente na seção 6.1.1.3.

A composição *Observer*, *Flyweight* e *Adapter* é traduzida no seguinte modelo:

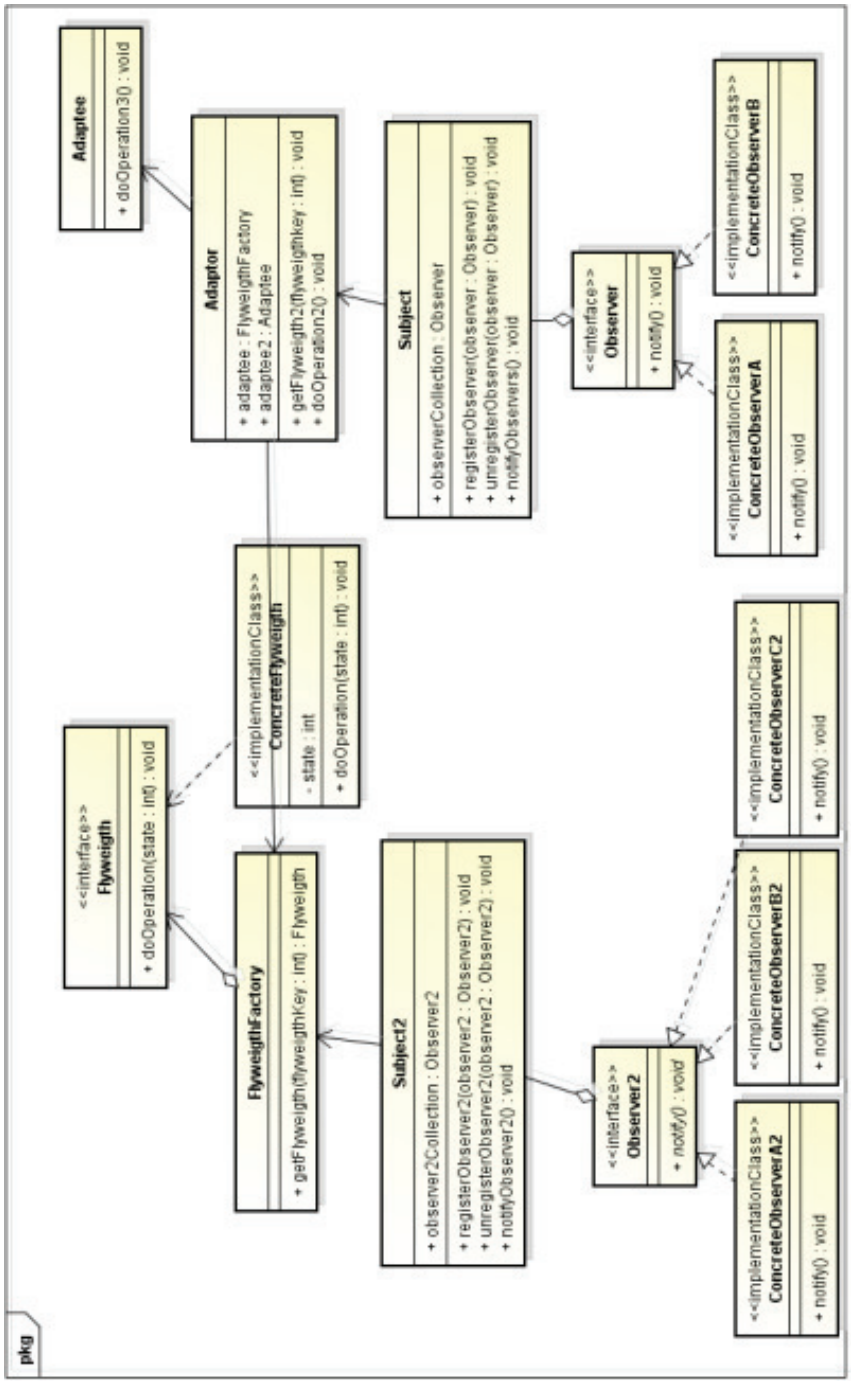

Figura 18 - Modelo da composição *Observer*, *Flyweight* e *Adapter*.

Por análise da composição *Observer*, *Flyweight* e *Adapter*, as classes centrais seriam: *Subject* e *Subject2* (representam o registro das notificações do padrão *Observer*), *Observer* e *Observer2* (representam a propagação da notificação para as classes implementadoras do padrão *Observer*) e *FlyweightFactory* (classe instanciadora do padrão *Flyweight*).

O primeiro componente esperado é composto pelas classes do padrão *Flyweight* (*FlyweightFactory*, *ConcreteFlyweight* e *Flyweight*), o segundo

componente é composto pelas classes do padrão *Adapter* (*Adaptor* e *Adaptee*) e o último componente descreve o padrão *Observer* (*Observer*, *Observer2*, *Subject2*, *Subject* e as classes *ConcreteObservers*).

6.1.1.6. Resultados Obtidos pela Ferramenta

Esta seção descreve os resultados obtidos na análise dos modelos de *design patterns*.

As atribuições dos pesos de cada tipo de relacionamento foram modificados de forma a constar que o resultado obtido pela ferramenta seja aproxidamente igual ao resultado esperado pelo arquiteto do modelo, referente aos graus de importância e à componentização apresentados anteriormente.

A Tabela 6 demonstra os graus de importância obtidos de todas as classes dentro do seu respectivo projeto de análise:

Nome do Projeto	Classes e Graus de Importância
Abstract Factory e *Strategy*	*ConcreteFactory2* (137), *ConcreteFactory1* (101), *AbstractFactory* (81), *AbstractProductC* (73), *AbstractProductB* (49), *ProductC1* (49), *AbstractProductA* (49), *Behaviour* (49), *ProductA2* (45), *ConcreteFactory3* (43), *ProductB2* (43), *ConcreteFactory4* (37), *ProductB1* (37), *ProductA1* (35), *ProductC2* (33), *ProductCBehaviour* (29), *ProductC3* (27), *ProductBBehaviour* (25), *ProductABehaviour* (25), *ClasseCliente* (19).
Bridge, *Mediator* e *Chain Of Responsability*	*Abstraction* (94), *Implementor* (90), *RefinedFunction2* (82), *Responsabilidade3* (66), *Responsabilidade1* (52), *Responsabilidade* (34), *ConcreteColleague1* (30), *Responsabilidade2* (12), *ConcreteMediator* (12), *ConcreteColleague2* (12), *ConcreteImplementor* (12), *RefinedFunction1* (12), *Mediator* (12).
Command, *Adaptor* e *Proxy*	*Client* (64), *ConcreteCommand1* (44), *Subject* (42), *BaseCommand* (42), *ConcreteCommand2* (10), *ConceteCommand3* (10), *Adaptee* (10), *Receiver* (10), *Adaptor* (10), *Proxy* (10) e *Invoker* (10).
Factory Method, *Facade* e *Visitor*	*ClasseCliente2* (203), *Facade* (195), *ObjectStructure* (163), *ElementLike* (115), *ClasseAbaixoFacade4* (89), *FactoryProduct* (89), *ClasseCliente* (61), *Visitable* (55), *Element* (53), *ClasseAbaixoFacade8* (19), *Visitor* (19), *ProductOne* (19), *ProductTwo* (19), *ElementoObject1* (19), *ElementoObject2* (19), *Element2* (19), *ClasseCliente3* (19), *ClasseAbaixoFacade1* (19), *ClasseAbaixoFacade2* (19), *ClasseAbaixoFacade5* (19), *ClasseAbaixoFacade6* (4), *ClasseAbaixoFacade9* (2), *ClasseAbaixoFacade10* (2).

Observer, *Flyweight* e *Adapter*	*Observer2* (115), *Subject* (93), *Subject2* (61), *Observer* (59), *FlyweightFactory* (57), *Adaptor* (37), *Flyweight* (37), *ConcreteObserverB2* (13), *ConcreteObserverC2* (13), *ConcreteFlyweight* (13), *ConcreteObserverA* (13), *Adaptee* (13), *ConcreteObserverB* (13), *ConcreteObserverA2* (13).

Tabela 6 - Tabela de Grau de Importância de modelos baseados em *design patterns*.

A Tabela 7, por sua vez, demonstra os componentes gerados dentro do seu respectivo projeto de análise:

Nome do Projeto	Componentes
Abstract Factory e *Strategy*	**Componente-1:** *AbstractProductC*, *ConcreteFactory1*, *ProductA1*, *ProductB1*, *ProductC1*, *ClasseCliente*. **Componente-2:** *AbstractProductB*, *ConcreteFactory4*, *ProductC3*, *AbstractFactory*, *AbstractProductA*, *ConcreteFactory3*, *ProductC2*, *ConcreteFactory2*, *ProductA2*, *ProductB2*. **Componente-3:** *Behaviour*, *ProductABehaviour*, *ProductBBehaviour*, *ProductCBehaviour*.
Bridge, *Mediator* e *Chain Of Responsability*	**Componente-1:** *Abstraction*, *ConcreteImplementor*, *RefinedFunction1*, *RefinedFunction2*, *ConcreteColleague1*, *ConcreteColleague2*, *Implementor*. **Componente-2:** *ConcreteMediator*, *Mediator*. **Componente-3:** *Responsabilidade1*, *Responsabilidade2*, *Responsabilidade3*, *Responsabilidade*.
Command, *Adaptor* e *Proxy*	**Componente-1:** *Adaptor*. **Componente-2:** *Adaptee*, *Subject*, *Proxy*, *Client*. **Component-3:** *Invoker*, *BaseCommand*, *ConcreteCommand1*, *ConcreteCommand2*, *ConcreteCommand3*.
Factory Method, *Facade* e *Visitor*	**Componente-1:** *ProductOne*, *ProductTwo*, *ClasseCliente*, *FactoryProduct*. **Componente-2:** *Element*, *ElementObject1*, *ElementObject2*, *Element2*, *ObjectStructure*, *ElementLike*. **Componente-3:** *Facade*, *ClasseCliente2*, *ClasseCliente3*, *ClasseAbaixoFacade1*, *ClasseAbaixoFacade2*, *ClasseAbaixoFacade4*, *ClasseAbaixoFacade5*, *ClasseAbaixoFacade8*, *Visitor*, *Visitable*. **Componente-4:** *ClasseAbaixoFacade6*, *ClasseAbaixoFacade9*, *ClasseAbaixoFacade10*.
Observer, *Flyweight* e *Adapter*	**Componente-1:** *FlyweightFactory*, *ConcreteFlyweight*, *Flyweight*. **Componente-2:** *Adaptor*, *Adaptee*. **Componente-3:** *Observer*, *Observer2*, *Subject2*, *ConcreteObserverA2*, *ConcreteObserverB2*, *ConcreteObserverC2*, *Subject*, *ConcreteObserverA*, *ConcreteObserverB*.

Tabela 7 - Tabela de componentes de modelos baseados em *design patterns*.

Os modelos de *design patterns* e seus resultados geraram a primeira versão da distribuição dos pesos dos tipos de relacionamento UML. Esta versão representa uma sintonia que será refinada nos próximos passos da avaliação. A Tabela 8 exibe as atribuições dos pesos:

Tipo de Relacionamento	Peso
Abstração	40
Generalização	60
Dependência	20
Realização	80
Associação	200
Composição	10
Agregação	10

Tabela 8 - Tipos de relacionamento e pesos gerados pelos modelos de *design patterns*.

6.1.2. Modelos Complexos

Os modelos complexos são utilizados para sintonizar os pesos dos tipos de relacionamento UML de maneira mais apurada.

Elaboraram-se cinco diagramas de classe para projetos mais complexos:

- QoS[14]
- QaC[15]
- AvalProc[16]
- MunicípiosEficientes
- AnalisadorUML

O projeto *QoS* define a qualidade de serviços de celular para a empresa *Vivo*. Serviços como *MMS*, *SMS*, *SMSP2P*, *WAP*, *Internet3G*, *Download* e *Upload de arquivos* são testados em variados dispositivos móveis para avaliar métricas de erro, tempo de resposta, tempos de *download* e *upload* e tempos de envio e recebimento.

O projeto *QaC* trabalha com o *QoS* via *web service* e define agendamentos de testes a serem executados nos dispositivos. O *QaC* é um projeto feito para a empresa *Vivo* e funciona como o *front-end* de análise dos serviços através também da exibição dos resultados.

Já o projeto *AvalProc*, também implementado para a empresa *Vivo*, realiza a avaliação de processos concorrentes. Os processos são estipulados pela própria

[14] Quality Of Service
[15] Quality Assurance Center
[16] Avaliação de Processos

Vivo para empresas fornecedoras, candidatas a resolução dos processos, responderem a questionários pré-formulados. As respostas do questionário deverão conter preços e comentários da forma de resolução. Assim, especialistas atribuem notas às respostas de cada fornecedor candidato e um laudo é gerado com todas as respostas, preços e notas dos processos. Desta maneira, se determina qual a empresa fornecedora que melhor se adequou ao processo.

O *Municípios Eficientes* foi um projeto resultante da parceria *ICA/SEPLAG/PRODERJ*. A proposta é desenvolver um controle das receitas tributárias municipais. O foco são os impostos de *ISSQN*[17], *IPTU*[18] e *Dívida Ativa*. Uma ferramenta de apoio à decisão e gestão foi desenvolvida com a elaboração de gráficos pré-concebidos, relatórios e cubos de informações, bem como a previsão de receitas mensais e anuais por impostos e a detecção de irregularidade de empresas na emissão do *ISSQN*.

Por fim, o projeto *AnalisadorUML* é o projeto de desenvolvimento fruto deste trabalho e já foi descrito anteriormente.

A própria estimativa de complexidade extraída pelo *AnalisadorUML* aliada com as métricas de tamanho servem de parâmetro para determinação da complexidade dos projetos citados. A Tabela 9 ilustra as estimativas de complexidade e algumas métricas de tamanho dos projetos:

Projeto	Número de Classes	Número de Relacionamentos	Estimativa de Complexidade	Comparação da estimativa de complexidade com o *FactoryMethod-Facade-Visitor*
FactoryMethod-Facade-Visitor	23	44	1.240	$\frac{1.240}{1.240} = 1$
QoS	129	486	39.256	$\frac{39.256}{1.240} = 31{,}66$
QaC	668	3071	1.382.208	$\frac{1.382.208}{1.240} = 1.114{,}68$
AvalProc	329	1874	360.880	$\frac{360.880}{1.240} = 291{,}03$

[17] Imposto Sobre Serviço de Qualquer Natureza
[18] Imposto sobre a Propriedade Predial e Territorial Urbana

MunicípiosEficientes	408	2221	253.646	$\frac{253.646}{1.240} = 204,55$
AnalisadorUML	125	619	51.220	$\frac{51.220}{1.240} = 41,31$

Tabela 9 - Métricas de tamanho e estimativas de complexidade dos projetos e a comparação com os *design patterns*.

Como se verifica na Tabela 9, os diagramas de classe gerados para os projetos complexos são modelos de grande porte (explicitado pela métrica de tamanho *Número de Classes*) e também um tanto quanto ilegíveis (explicitado pela métrica de tamanho *Número de Relacionamentos*). Por isso, os modelos de diagramas de classe dos projetos complexos não foram demonstrados no texto deste estudo, visto que as suas visualizações não iriam agregar nenhuma informação ao leitor.

Os modelos de diagramas de classe dos projetos *QaC* e *QoS* foram gerados automaticamente por engenharia reversa pela ferramenta *AltovaUMLModel* e exportada para o *AstahProfessional 6.5*. Isto porque foi proposto pela *Vivo*, um projeto de refatoração do código para desassociar o uso de um framework antigo, sobre o qual as duas ferramentas funcionam.

Este framework antigo funciona como suporte à comunicação cliente-*controller*-servidor e à comunicação servidor-banco de dados. Porém, atualmente, no ambiente Java, existe um framework altamente difundido que suporta a mesma funcionalidade, que é o *spring*. Desta maneira, para desenvolver a desassociação do framework do *QoS* e do *QaC*, basta remover as classes referentes a este framework e inserir as classes de configuração do s*pring*.

Assim, para aplicar esta modificação no modelo, todas as classes que representam o framework da *Vivo* foram retiradas dos diagramas de classes e dois novos projetos *QaCSemFramework* e *QoSSemUtils* foram criados. Os projetos *QaC* e *QoS* foram mantidos no seu formato original, com o uso do framework antigo, para confronto de estimativas de complexidade e coesão com *QaCSemFramework* e *QoSSemUtils*, respectivamente.

O *QacSemFramework* tem estimativa de complexidade 1.016.498 contra 1.382.208 do *QaC*. O *QosSemUtils* tem estimativa de complexidade 41.508 contra 39.256 do *QoS*. Ou seja, a remoção do framework é benéfica para a diminuição da complexidade do *QaC*, mas não é benéfica para o *QoS*.

Aliadas às complexidades, as coesões foram estudadas com e sem o framework, através da utilização da componentização com o mesmo número de grupos. Vale lembrar que quanto maior a estimativa de coesão, menor a coesão dentro dos componentes de um projeto. O projeto *QaCSemFramework* obteve estimativa de coesão 36.645 e o *QaC* obteve 39.640. Já o *QoSSemUtils* acusou estimativa de coesão 13.800 e o projeto *QoS* 14.100. Isto foi importante para argumentar que a remoção do framework implicará um aumento na coesão entre as classes, visto que as estimativas diminuiram. Ou seja, a remoção do framework aumentará o poder de manutenção e reusabilidade das ferramentas.

Os diagramas de classes dos projetos *AvalProc* e *AnalisadorUML* foram escolhidos para realizar a sintonia fina dos pesos. Visto que as complexidades destes dois projetos são menores, será mais fácil confrontar os *core concepts* e os componentes de classe esperados e obtidos.

Em ambos os diagramas de classe foram modeladas apenas as classes do servidor e do *controller*. As classes do cliente ficaram de fora para não haver dificuldade na avaliação da componentização. Quando as classes do cliente são desenvolvidas no modelo, a componentização tende a criar grupos de classe do cliente e grupos de classe do servidor, quando, na verdade, o interessante da componentização é separar os assuntos do diagrama de classes. A ferramenta não agregaria informações se separasse apenas o servidor do cliente, pois esta informação é transparente ao arquiteto.

Na avaliação do projeto *AvalProc* são esperados como conceitos centrais as classes *Documento* (guarda informações dos diferentes tipos de documento que representam os questionários feitos pela *Vivo* e as respostas dos fornecedores), *Projeto* (define todas as informações iniciais de quais empresas são candidatas à resolução do processo, qual é o processo e quais os recursos envolvidos) e *Análise* (classe que guarda todas as informações da etapa de análise com atribuições de notas para as respostas dos fornecedores).

Já na componentização, o *AvalProc* contém quatro componentes bem definidos. O primeiro componente é composto por todas as classes envolvidas na criação do projeto que descreve o processo (ponto inicial da aplicação), com a consequente criação do questionário e das empresas envolvidas na disputa. O segundo componente se refere às classes do processo de avaliação dos projetos, onde são dadas notas às respostas dos fornecedores e é realizado uma pré-seleção

das propostas. O terceiro componente contém todas as classes responsáveis por montar o laudo com todos os detalhes do projeto, juntamente com a aplicação de fórmulas para a escolha final da empresa fornecedora. O último componente é composto pelas configurações do sistema.

Com relação à avaliação dos *core concepts* do projeto *AnalisadorUML*, três classes deveriam ter maior importância: *ProjetoServiceImpl* (classe concreta da interface *ProjetoService*, que implementa ou repassa a implementação de todas as operações do módulo *Projeto*), *LerXMI* (classe responsável por ler o arquivo XMI e tranformá-lo em um grafo orientado) e *ProjetoJDBCDAO* (classe concreta da interface *ProjetoDAO*, que realiza toda a comunicação com o banco de dados do módulo *Projeto*).

No que tange à componentização, a análise feita para o projeto *AnalisadorUML* espera cinco componentes. O primeiro componente é composto pelas classes responsáveis pela leitura do arquivo XMI. O segundo componente contém as classes do módulo *Configuração*. O terceiro componente possui as classes utilizadas no *Data Access Object* e no *Data Transfer Object*, pois todas estas classes herdam da mesma classe, responsável pela comunicação com o banco de dados. O quarto componente contém as classes responsáveis pelo agendamento, componentização e obtenção dos conceitos centrais, visto que a componentização e os conceitos centrais compartilham o uso do algoritmo *Floyd Warshall* em suas detecções. Por fim, o último componente é composto pelas classes restantes do módulo *Projeto*.

Estabeleceu-se um limite mínimo de noventa por cento de acerto no confronto dos *core concepts* e componentes esperados com os extraídos pela ferramenta *AnalisadorUML*. Para isto, os pesos dos diferentes tipos de relacionamento UML foram alterados para configurar uma sintonia final nas atribuições dos pesos no grafo orientado de classes.

O resultado da sintonia fina é encontrado na Tabela 10:

Tipo de Relacionamento	Peso
Abstração	35
Generalização	40
Dependência	25
Realização	50

Associação	105
Composição	10
Agregação	15

Tabela 10 - Pesos atribuídos finais para cada tipo de relacionamento.

Para estudar os benefícios reais do ponto de vista de processo de software e de modelos que contêm classes do cliente, do servidor e do *controller*, foi desenvolvido o projeto *MunicípiosEficientes*. Este projeto tem porte médio-grande e a arquitetura do software foi definida antes da codificação. Assim, inicialmente já havia informações de requisitos e o esboço de diagrama de classes bem elaborado.

Era necessário precificar o projeto, estimar o esforço de trabalho e prever a capacidade de manutenção.

Para as duas primeiras informações foram utilizadas a estimativa de complexidade e as métricas de tamanho. Estes resultados calculam o tamanho da ferramenta e a dificuldade de codificar o diagrama. Através da comparação da estimativa de complexidade e das métricas do *MunicípiosEficientes* com as mesmas informações do *AvalProc* e do *QoS* (projetos fechados de complexidade similar e custo bem-definido), o preço do projeto ficou transparente, bem como o estimativa de esforço.

Na previsão da capacidade de manutenção, estudaram-se a estimativa de coesão e as métricas de acoplamento. A estimativa de coesão, com o mesmo número de grupos, para o *MunicípiosEficientes* foi 50.220 e para o *QaC* foi 39.640. Como quanto maior a estimativa de coesão obtida na ferramenta, menor a coesão, é fácil conferir que o projeto *MunicípioEficientes* é bem menos coeso do que o *QaC*, uma vez que a estimativa de coesão do *MunicípiosEficientes* é maior e a estimativa de complexidade é bem menor (253.646 contra 1.382.208).

A coesão menor no *MunicípiosEficientes* era algo previsto, porque a ferramenta abrange um número grande de funcionalidades de assuntos diferentes (os diferentes impostos contêm regras diferentes entre si) e o *QaC* abrange somente os diferentes tipos de serviço de dispositivos móveis.

Para o processo de software, as informações de complexidade e de grupos de classes, juntamente com as métricas, foram importantes na divisão do trabalho. Dois desenvolvedores mais experientes foram alocados para implementar as classes e os componentes mais complexos e que contêm uma coesão baixa. Em

contrapartida, oito desenvolvedores menos experientes implementaram o resto do diagrama menos complexo.

Esta divisão foi fundamental do ponto de vista do processo. Como os desenvolvedores mais experientes desenvolveram componentes menos coesos, eles eram responsáveis por manter a integridade desta coesão com outros componentes. Para isto, um número maior de validações e testes foram implementados para o conjunto de classes em questão. E, sempre que um desenvolvedor menos experiente relacionava estas classes testadas com classes menos complexas no seu componente, uma comunicação entre estes desenvolvedores se tornou obrigatória para manutenção da integridade.

Outro ponto importante é a distribuição das classes não-complexas entre desenvolvedores inexperientes. Visto que as classes não-complexas são maioria em um projeto complexo, é possível alocar um número maior de desenvolvedores inexperientes no projeto, sem que eles estejam alterando as mesmas classes simultaneamente. Este fator interfere, visto que se dois desenvolvedores alteram a mesma classe, surge uma etapa de *merge* que realizará a integração da modificação dos dois desenvolvedores. A etapa de *merge* tem que ser minimizada, pois pode trazer eventuais erros de consistência, padronização de codificação e tempo desperdiçado de implementação.

No que diz respeito aos diagramas que modelam as classes do cliente junto com as classes do servidor e do *controller*, a solução foi aumentar o número de componentes para existir uma subdivisão dos componentes do cliente e do servidor.

A diferença observada na componentização é que o servidor é separado por assunto ou por uso de um determinado framework que force a coesão. Já o cliente é separado por componentes visuais de tela. Por exemplo, um dos componentes gerados para a parte cliente no projeto *MunicípioEficientes* era composto por classes que geram gráficos em tela e outro componente do cliente composto por classes que geram *grids*.

6.1.3.
Modelos de Maior Nível de Abstração

O último passo para a avaliação da ferramenta *AnalisadorUML* e do estudo proposto é a verificação do mapeamento de pesos dos tipos de relacionamento UML.

Para verificar o mapeamento, foram desenvolvidos cinco modelos de diagramas de classe em um nível de abstração maior. Dois destes modelos serão discutidos nesta seção.

6.1.3.1.
Projeto Amplo

O *ProjetoAmplo* foi desenvolvido para conter todos os tipos de relacionamento dos quais o estudo suporta.

Para o caso de associação n-ária, o peso da aresta atribuído é igual à metade do peso de uma associação, ou seja, $\frac{105}{2} = 52{,}5$.

A Figura 19 ilustra o modelo implementado para o *ProjetoAmplo*:

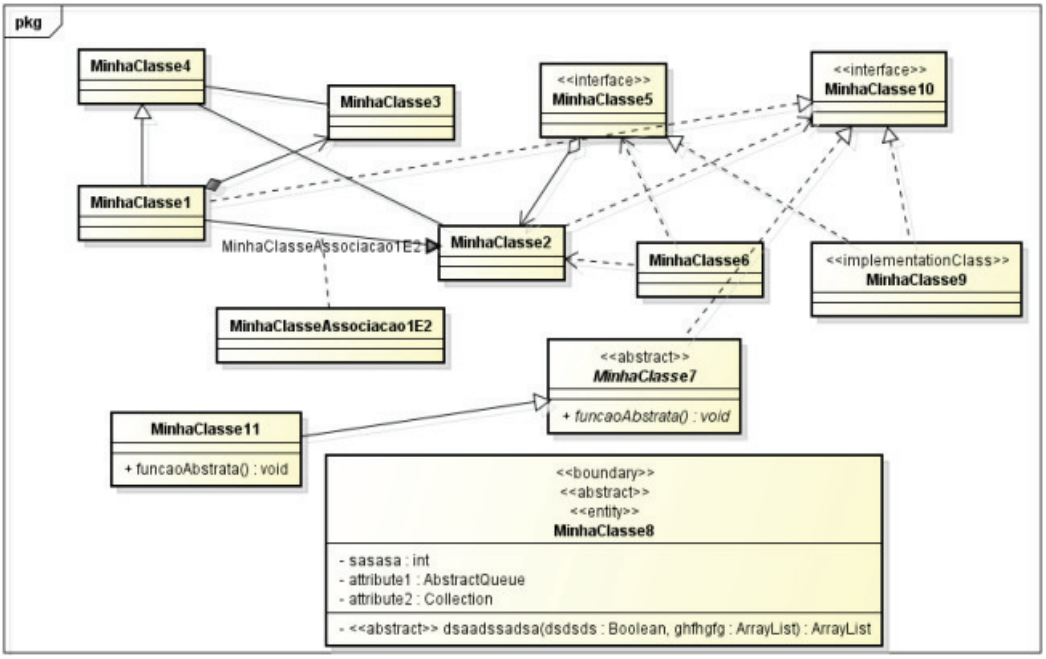

Figura 19 - Diagrama de classes do projeto amplo.

Pela análise do modelo do *ProjetoAmplo*, as classes esperadas como classes centrais seriam: *MinhaClasse10* (interface das classes implementadoras *MinhaClasse1*, *MinhaClasse2*, *MinhaClasse7*, *MinhaClasse9*), *MinhaClasse1* (herda por generalização de *MinhaClasse4*, tem uma relação de composição com

MinhaClasse3 e se associa com *MinhaClasse2* por um relacionamento qualificado pela classe de associação *MinhaClasseAssociacao1E2*) e *MinhaClasse2* (além dos relacionamentos descritos, também se relaciona com *MinhaClasse4* por associação e com *MinhaClasse6* por dependência).

Para os componentes, é possível notar no diagrama um grupo de coesão mais forte entre as classes *MinhaClasse1*, *MinhaClasse2*, *MinhaClasse3* e *MinhaClasseAssociacao1E2* e outro grupo de coesão entre as classes *MinhaClasse2*, *MinhaClasse5*, *MinhaClasse6*, *MinhaClasse7*, *MinhaClasse9*, *MinhaClasse10*, *MinhaClasse11*. Desta forma, a classe *MinhaClasse2* estará dividida entre os dois componentes. Porém, espera-se que esta classe seja alocada no segundo grupo, visto que os relacionamentos de dependência com *MinhaClasse6* e de agregação com *MinhaClasse5* são mais fortes que os relacionamentos de associação com *MinhaClasse4* e de associação n-ária com *MinhaClasseAssociacao1E2*. Por fim, a classe *MinhaClasse8*, como não tem nenhum relacionamento, não será alocada a nenhum componente.

6.1.3.2. Talisman

O Projeto *Talisman* foi retirado da monografia descrita em Staa (2011). O diagrama foi conferido pelo autor da monografia para confrontar os valores esperados dos componentes e das classes centrais esperadas por ele, com as obtidas pela ferramenta.

A Figura 20 ilustra o modelo dos tipos de dados da ferramenta Talisman:

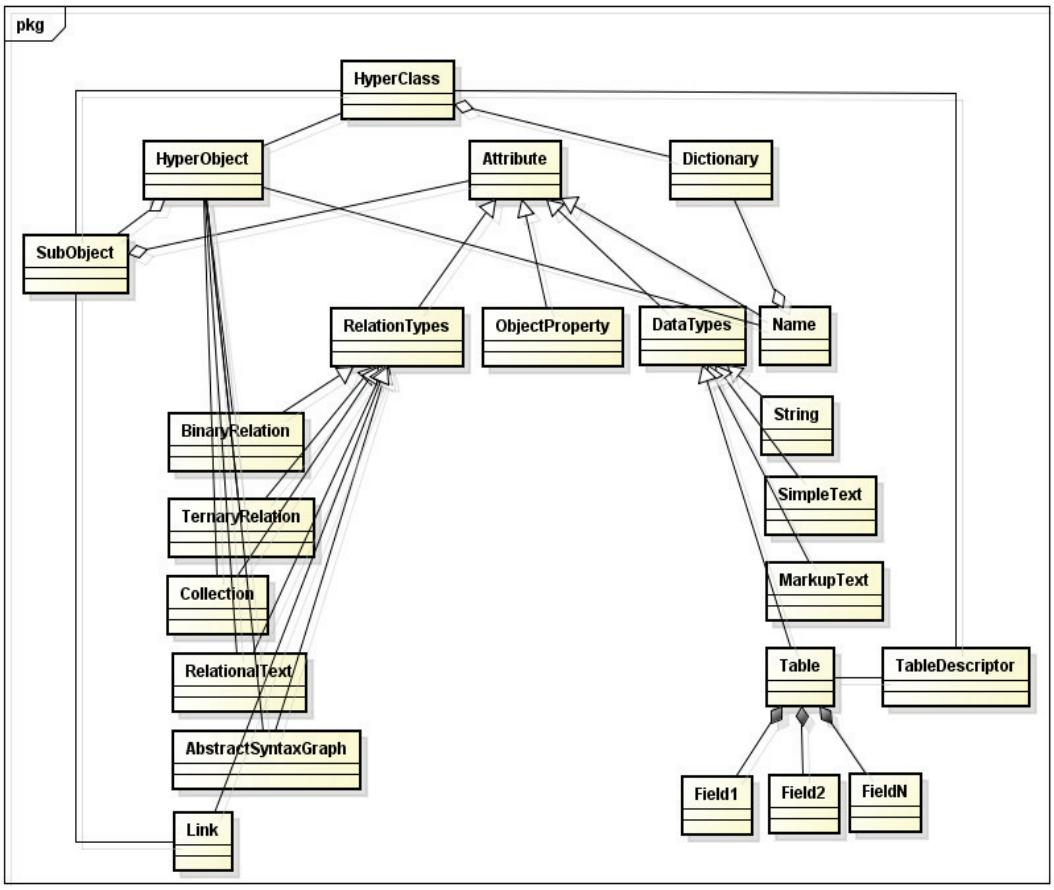

Figura 20 - Diagrama de classes de tipos de dados da ferramenta Talisman.

Através da análise do modelo da ferramenta *Talisman*, os *core concepts* do diagrama de classes seriam *Attribute* (pai e avô de grande parte do diagrama), *DataType* (pai de quatro classes) e *RelationType* (pai de seis classes).

No que se refere a componentização, o primeiro grupo de classes conteria a classe *RelationTypes* e seus filhos, o segundo seria composto da classe *DataType* e seus filhos e o terceiro grupo com o restante das classes.

6.1.3.3. Resultados Obtidos pela Ferramenta

Esta seção descreve os resultados obtidos na análise dos modelos de projetos com maior nível de abstração.

Os pesos de cada tipo de relacionamento foram atribuídos pelos modelos de projetos complexos e não sofreram alteração nesta base amostral. Os cinco modelos mais abstratos serviram de teste da atribuição de pesos na ferramenta.

A Tabela 11 demonstra os graus de importância das classes:

Nome do Projeto	Classes e Graus de Importância

ProjetoAmplo	*MinhaClasse10* (60), *MinhaClasse1* (44), *MinhaClasse2* (40), *MinhaClasse7* (28), *MinhaClasse5* (16), *MinhaClasse6* (10) , *MinhaClasse4* (10) , *MinhaClasse9* (10) , *MinhaClasse11* (10) , *MinhaClasse3* (10) , *MinhaClasseAssociacao1E2* (10), *MinhaClasse8* (0).
Talisman	*Attribute* (346), *DataTypes* (254), *RelationTypes* (234), *Table* (150), *Name* (120), *Dictionary* (86), *SubObject* (54), *HyperClass* (48), *String* (22), *SimpleText* (22), *MarkupText* (22), *Field1* (22), *Field2* (22), *FieldN* (22), *TableDescriptor* (22), *AbstractSyntaxGraph* (22), *HyperObject* (22), *BinaryRelation* (22), *TernaryRelation* (22), *Collection* (22), *RelationalText* (22), *ObjectProperty* (22) e *Link* (22).

Tabela 11 - Graus de importância das classes de projetos abstratos.

A Tabela 12, por sua vez, demonstra os componentes gerados:

Nome do Projeto	Componentes
ProjetoAmplo	**Componente-1:** *MinhaClasse1*, *MinhaClasse3*, *MinhaClasse4*, *MinhaClasseAssociacao1E2*. **Componente-2:** *MinhaClasse2*, *MinhaClasse6*, *MinhaClasse7*, *MinhaClasse9*, *MinhaClasse11*, *MinhaClasse5*, *MinhaClasse10*. **Componente-3:** *MinhaClasse8*.
Talisman	**Componente-1:** *DataTypes*, *String*, *SimpleText*, *MarkupText*, *Table*, *Field1*, *Field2*, *FieldN*, *TableDescriptor*. **Componente-2:** *RelationTypes*, *BinaryRelation*, *TernaryRelation*, *Collection*, *RelationalText*, *AbstractSyntaxGraph*, *Link*. **Componente-3:** *HyperClass*, *Dictionary*, *Name*, *ObjectProperty*, *SubObject*, *HyperObject*, *Attribute*.

Tabela 12 - Componentes gerados dos projetos abstratos.

Pode-se observar que os resultados obtidos estão de acordo com os resultados esperados. Em cima disto, conclui-se, que a determinação dos pesos de cada tipo de relacionamento UML entre classes se mostrou eficaz.

6.2. Reflexão sobre os Pesos Atribuídos

A Tabela 10, descrita na seção 6.1.2, representa a distribuição de pesos para cada tipo de relacionamento entre classes UML. É possível discutir o resultado obtido pela ferramenta através da análise da coesão embutida nos relacionamentos.

Na visualização de um grafo orientado, quanto menor for o peso da aresta ou do caminho entre dois nós, mais coesos eles são. Assim, quanto mais forte e restritivo for um relacionamento UML, menor a distância entre as classes.

Na avaliação das definições dos relacionamentos da seção 4.1.1, a composição e a agregação são as relações mais fortes. O objeto “Todo” é composto por objetos “Parte”, com a diferença que, na composição, se o “Todo”

morre, os objetos “Parte” também morrem. Desta maneira, a composição é o relacionamento mais forte por conter restrições de agregação e de dependência.

Depois da composição e agregação, o relacionamento mais forte é a dependência, porque para um objeto existir, ele depende da existência do outro. Logo atrás, estão a abstração e a generalização. Os dois relacionamentos identificam características comuns entre várias classes que serão encontradas em uma superclasse (classe pai), com a diferença que, na abstração, também existe um relacionamento de realização, a classe pai força a implementação de determinadas funções abstratas nas classes filhas. Desta maneira, a abstração é um relacionamento mais forte que a generalização.

Sobram os relacionamentos de realização e associação. A realização acontece quando uma classe implementa funções ou comportamentos especificados por outra classe e a associação denota uma conexão simples entre classes. É fácil observar que a realização impõe mais restrições no relacionamento que a associação, ou seja, existe uma intimidade maior entre as classes.

Através da análise semântica acima, já era esperado que a ordenação dos relacionamentos do mais forte para o mais fraco seria: composição, agregação, dependência, abstração, generalização, realização e associação.