

WILLIAN RIGHETTO PIMENTEL

**IMPACTO DOS DESIGN PATTERNS EM UM PROJETO DE
SOFTWARE**

Monografia apresentada ao PECE – Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para conclusão do curso de MBA em Tecnologia de Software.

São Paulo
2015

WILLIAN RIGHETTO PIMENTEL

**IMPACTO DOS DESIGN PATTERNS EM UM PROJETO DE
SOFTWARE**

Monografia apresentada ao PECE – Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para a conclusão do curso de MBA em Tecnologia de Software.

Área de Concentração: Tecnologia de Software

Orientador: Prof. Dra. Gabriela Cabel Barbarán

São Paulo
2015

DEDICATÓRIA

Dedico este trabalho aos meus pais, familiares e amigos pelo apoio, compreensão e incentivo durante o curso.

AGRADECIMENTOS

À Universidade de São Paulo – USP pela oportunidade de fazer o curso.

À todos os professores do curso de Tecnologia de Software que ministraram suas aulas com excelência e profundidade.

À Prof. Dra. Gabriela Cabel Barbarán pela orientação deste trabalho, dedicando seu tempo e conhecimento neste projeto.

RESUMO

Neste trabalho foi estudado o impacto de sete *design patterns* (*Factory Method*, *Abstract Factory*, *Flyweight*, *Visitor*, *Decorator*, *Template Method*) em seis atributos de qualidade, em um projeto de *software* cuja equipe de desenvolvimento é formada por desenvolvedores iniciantes. No intuito de identificar os impactos dos *design patterns* foi observado o comportamento dos componentes do projeto, sem o uso dos *patterns* e posteriormente o comportamento com o uso dos *patterns*. No final do estudo foram listadas recomendações para o uso adequado dos *design patterns* pelos desenvolvedores iniciantes. Foi mostrado que alguns *design patterns*, dependendo do cenário de uso, podem impactar negativamente muitos atributos de qualidade, por exemplo, deixam o código mais difícil de ser lido, mais difícil de ser apreendido e compreendido. Por outro lado, também foi mostrado que alguns *design patterns*, podem ser bastante úteis ao projeto de *software*, deixando o código mais simples de ser lido, melhor modularizado, entre outros benefícios.

Palavras-chave: *Design patterns*, Atributos de Qualidade, *Composite*, *Factory Method*, *Abstract Factory*, *Flyweight*, *Visitor*, *Decorator*, *Template Method*

ABSTRACT

This work studied the impact of seven design patterns (Composite, Factory Method, Abstract Factory, Flyweight, Visitor, Decorator, Template Method) in six quality attributes in a software project whose development team consists of junior developers. In order to identify the impacts of design patterns, it was observed the behavior of the components of a project without using design patterns and then the behavior with the use of design patterns. At the end of the study were listed recommendations for the appropriate use of design patterns for junior developers. It was shown that some design patterns, depending on the usage scenario, can negatively impact many quality attributes, for example: make the code harder to be read, more difficult to be learned and understood. Moreover, it was also shown that some design patterns can be quite useful for the project, making the code simpler to be read, better modularized, among other benefits.

Keywords: Design Patterns, Quality Attributes, Composite, Factory Method, Abstract Factory, Flyweight, Visitor, Decorator, Template Method

LISTA DE ILUSTRAÇÕES

	Pág.
Figura 1: Estrutura do design pattern Composite	19
Figura 2: Estrutura do design pattern Factory Method	20
Figura 3: Estrutura do design pattern Abstract Factory	21
Figura 4: Estrutura do design pattern Flyweight	23
Figura 5: Estrutura do design pattern Template Method	24
Figura 6: Estrutura do design pattern Visitor	25
Figura 7: Estrutura do design pattern Decorator.....	27
Figura 8: Características e sub-características de qualidade.....	28
Figura 9: Design patterns analisados nos 17 estudos escolhidos	32
Figura 10: Diagrama de classe original do caso de uso Identificar Regiões Críticas	37
Figura 11: Diagrama de Classe do caso de uso Identificar Regiões Críticas, com aplicação do design pattern Composite	38
Figura 12: Diagrama de classe original para criação dos meios de transporte	40
Figura 13: Diagrama de classes da funcionalidade de criação de objetos de cada veículo, com aplicação do design pattern Factory Method.....	41
Figura 14: Diagrama de classe original para criação de transações	44
Figura 15: Diagrama de classes da funcionalidade de criação de veículos e de transações, com aplicação do design pattern Abstract Factory	45
Figura 16: Diagrama de classes da funcionalidade de coletar e armazenar dados de posição GPS dos veículos.....	50
Figura 17: Diagrama refatorado utilizando design pattern Flyweight.....	51
Figura 18: Diagrama de classes da funcionalidade de geração de relatórios administrativos	53
Figura 19: Diagrama refatorado utilizando design pattern Template Method	54
Figura 20: Diagrama de classes das funcionalidades de Comprar Créditos, Reembolsar Créditos e Usar Créditos.....	56
Figura 21: Diagrama refatorado utilizando design pattern Visitor.....	57
Figura 22: Diagrama de classes da funcionalidade de obter identificação do veículo	60
Figura 23: Diagrama refatorado utilizando design pattern Decorator	61

LISTA DE TABELAS

	Pág.
Tabela 1: Classificação dos Design patterns.....	17
Tabela 2: Estimativa do impacto dos design patterns em três atributos de qualidade	30
Tabela 3: Características e Atributos de Qualidade do Sistema	36
Tabela 4: Classificação da avaliação dos Design patterns	65
Tabela 5: Resultados do questionário aplicado.....	66

SUMÁRIO

	Pág.
1. INTRODUÇÃO	11
1.1 Motivações.....	11
1.2 Objetivo.....	11
1.3 Justificativas.....	12
1.4 Estrutura do Trabalho	13
2. REVISÃO DA LITERATURA	14
2.1 Processo de <i>Software</i>	14
2.1.1 Atividades Genéricas de um Processo	14
2.2 <i>Design pattern</i>	15
2.2.1 Classificação dos <i>Design patterns</i>	16
2.2.2 <i>Design pattern: Composite</i>	18
2.2.3 <i>Design pattern: Factory Method</i>	19
2.2.4 <i>Design pattern: Abstract Factory</i>	20
2.2.5 <i>Design pattern: Flyweight</i>	22
2.2.6 <i>Design pattern: Template Method</i>	23
2.2.7 <i>Design pattern: Visitor</i>	24
2.2.8 <i>Design pattern: Decorator</i>	26
2.3 Atributos de Qualidade de Produtos de <i>Software</i>	27
2.4 Estudos dos impactos dos <i>design patterns</i> em projetos de <i>software</i>	28
2.4.1 Estudo de Khomh & Gueheneuc.....	29
2.4.2 Estudo de Ali & Elish.....	31
2.5 Considerações do Capítulo.....	33
3. IDENTIFICAÇÃO DOS IMPACTOS DO USO DE DESIGN PATTERNS.....	34
3.1 Projeto de <i>Software</i>	34
3.2 Processos de Desenvolvimento.....	35
3.3 Atributos de qualidade	35
3.4 <i>Design patterns</i> analisados.....	37
3.4.1 <i>Composite</i>	37
3.4.2 <i>Factory Method</i>	40
3.4.3 <i>Abstract Factory</i>	44
3.4.4 <i>Flyweight</i>	50
3.4.5 <i>Template Method</i>	53
3.4.6 <i>Visitor</i>	56
3.4.7 <i>Decorator</i>	59
3.5 Considerações do capítulo.....	63
4. ANÁLISE DOS RESULTADOS	64
4.1 Avaliação dos <i>Design patterns</i>	64
4.2 Análise dos <i>Design patterns</i>	66
4.2.1 Análise <i>Design pattern Composite</i>	68
4.2.2 Análise <i>Design pattern Factory Method</i>	68
4.2.3 Análise <i>Design pattern Abstract Factory</i>	69
4.2.4 Análise <i>Design pattern Flyweight</i>	69
4.2.5 Análise <i>Design pattern Visitor</i>	69
4.2.6 Análise <i>Design pattern Decorator</i>	70
4.2.7 Análise <i>Design pattern Template Method</i>	70

4.3	Lista de Recomendações.....	71
4.4	Considerações do capítulo.....	72
5.	CONSIDERAÇÕES FINAIS	73
5.1	Contribuições do Trabalho	73
5.2	Trabalhos Futuros.....	74
	REFERÊNCIAS.....	75
	APÊNDICE – QUESTIONÁRIO.....	76

1. INTRODUÇÃO

Este capítulo apresenta as motivações, o objetivo, as justificativas e a estrutura do trabalho.

1.1 Motivações

A área acadêmica trata o tema *design patterns* como um assunto fundamental e básico na engenharia de *software*. Porém ao ingressar no mercado de trabalho, seja por falta de conhecimento ou por falta de experiência, os profissionais da área de desenvolvimento muitas vezes utilizam os *patterns* de forma indiscriminada no projeto, ou ainda não os utilizam por falta de conhecimento técnico. Tanto o uso excessivo de *design patterns* quanto a falta deles podem fazer com que os projetos de *software* se tornem mais acoplados e complexos.

1.2 Objetivo

O objetivo do trabalho é analisar os *design patterns* do GOF (*Gang of Four*), quando a aplicação de *design pattern* pode ser útil e quando pode ser prejudicial ao projeto da análise dos *design patterns* do GOF, e propor uma lista de recomendações para o uso adequado dos *patterns* em projetos cuja equipe é formada por desenvolvedores iniciantes (desenvolvedores com pouca experiência profissional na área de desenvolvimento de *software* e que nunca trabalharam com os *design patterns* GOF), evitando assim impactos negativos nos projetos de *software*. No intuito de verificar a coerência da lista proposta, um projeto real foi analisado, identificando as situações em que os *design patterns* são mais prejudiciais do que benéficos.

1.3 Justificativas

Alguns estudos sugerem que o uso de *design patterns* podem não resultar em bom *design*. Os *patterns* podem também atrapalhar futuras extensões por tentar simplificar trechos de código (Khomh & Gueheneuc, 2008). Por exemplo, um dos problemas conhecidos dos *design patterns* é do espalhamento de trechos de código em diversas classes, onde o código necessário para resolver um problema é espalhado sobre diversas classes que tem como objetivo resolver problemas pontuais específicos. Esse espalhamento de código faz com que o *software* tenha sua capacidade de reusabilidade e manutenibilidade reduzidas (Maghawry & Dawood, 2010).

A literatura disponível sobre os padrões de projeto é focada na identificação e documentação dos padrões, ao invés de relatórios e análises da experiência de uso deles (Zhang & Budgen, 2012). Dessa forma o desenvolvedor inexperiente pode não conseguir aplicar os *design patterns* de modo correto no sistema em que está trabalhando, fazendo com que os *design patterns*, como já mencionado, atrapalhem a futura manutenção e reusabilidade do *software* (Maghawry & Dawood, 2010).

As principais vantagens que são ditas para se utilizar *design patterns* são: (1) Melhorar a qualidade do *software* e a produtividade do programador; (2) Aumentar as experiências de *design* de novatos, estudando a noção de padrões de projeto; (3) Promover a adoção de boas práticas mesmo para profissionais experientes; (4) Melhorar a comunicação entre os profissionais de desenvolvimento. Porém, não existem evidências empíricas que de fato justifiquem essas vantagens (Ali & Elish, 2013).

1.4 Estrutura do Trabalho

O Capítulo 2 REVISÃO BIBLIOGRÁFICA apresenta os tópicos relacionados com a construção e análise deste trabalho, como a definição de *design patterns* e definição de atributos de qualidade.

O Capítulo 3 IDENTIFICAÇÃO DOS IMPACTOS DO USO DE *DESIGN PATTERNS* apresenta a aplicação de sete *design patterns* em no projeto de *software* SITUR.

O Capítulo 4 ANÁLISE DOS RESULTADOS apresenta a análise da aplicação dos sete *design patterns* sobre seis atributos de qualidade no projeto SITUR.

O Capítulo 5 CONSIDERAÇÕES FINAIS descreve as contribuições do trabalho e idéias para trabalhos futuros.

O APENDICE apresenta o questionário utilizado para entrevista a respeito do impacto dos *design patterns* sobre os atributos de qualidade em um projeto de *software*.

2. REVISÃO DA LITERATURA

Este capítulo tem como objetivo expor os conceitos necessários para entendimento dos demais capítulos deste trabalho: Conceito de processo de software e suas principais atividades, conceito de design patterns e detalhamento dos design patterns utilizados neste trabalho, conceito de atributos de qualidade e a demonstração de dois estudos sobre o impacto dos design patterns em projetos de software.

2.1 Processo de *Software*

De acordo com Pressman & Maxim (2001), quando se elabora um produto ou sistema é importante percorrer uma série de passos previsíveis que ajudem a criar a tempo um resultado de alta qualidade. O roteiro que é seguido é chamado de Processo de *Software*.

Os engenheiros de *software* e seus gerentes adaptam o processo a suas necessidades e depois o seguem. Um processo é importante porque fornece estabilidade, controle e organização para uma atividade que pode, se deixada sem controle, tornar-se bastante caótica. No entanto, uma abordagem moderna de engenharia de *software* precisa ser “ágil”: precisa exigir apenas aquelas atividades, controles e documentação adequados à equipe de projeto e ao produto que deve ser produzido.

Em nível de detalhe, o processo adotado depende do *software* que está sendo construído. Um processo poderia ser apropriado à criação de *softwares* para o sistema de uma aeronave, enquanto um processo inteiramente diferente seria indicado para a criação de uma página de Internet, por exemplo.

2.1.1 Atividades Genéricas de um Processo

Ainda de acordo com Pressman & Maxim (2001) um processo genérico é formado por cinco atividades:

- A. Comunicação – Envolve alta comunicação e colaboração com o cliente (e outros *stakeholders*) e abrange o levantamento de requisitos e outras atividades relacionadas.
- B. Planejamento – Estabelece um plano para o trabalho de engenharia de *software*. Descreve as tarefas técnicas a serem conduzidas, os riscos prováveis, os recursos que serão necessários, os produtos de trabalho a serem produzidos e um cronograma de trabalho.
- C. Modelagem – Inclui a criação de modelos que permitam ao desenvolvedor e ao cliente, entender melhor os requisitos do *software* e o projeto que vai satisfazer a esses requisitos.
- D. Construção – Combina geração de código (manual ou automática) e os testes necessários para revelar erros no código;
- E. Implantação – O *software* é entregue ao cliente, que avalia o produto e fornece comentários.

Os *design patterns* citados e/ou utilizados neste trabalho, por serem focados na orientação a objetos, serão utilizados nas atividades de Modelagem (análise e projeto) e construção (codificação e testes).

2.2 Design pattern

Um *pattern* descreve determinado problema, que ocorre repetidamente em um ambiente específico, e descreve também uma solução para este problema, de forma que é possível aplicar a solução toda vez que o problema acontecer (Gamma, et al., 1994).

Segundo Gamma, et al. (1994), um *pattern* possui quatro elementos essenciais:

- Nome: Forma de identificar o *pattern*
- Problema: Descreve quando aplicar o *pattern*. Explica o problema e seu contexto.

- Solução: Descreve os elementos que formam o *design*, seus relacionamentos, responsabilidades e colaborações. A solução não descreve um *design* concreto ou implementação, pois um *pattern* é como um *template* que pode ser aplicado em diferentes situações.
- Consequências: São os resultados e trade-offs de aplicação do *pattern*. As consequências são fundamentais para avaliar alternativas de *design* para o projeto e para entender os custos e benefícios da aplicação do *pattern*.

Para este trabalho foram escolhidos os design patterns do GOF (Gang of Four) por ser um grupo de patterns abrangente e bastante utilizados pelos engenheiros de software.

2.2.1 Classificação dos *Design patterns*

De acordo com Gamma, et al. (1994), os *patterns* são classificados em dois critérios (ver tabela 1):

- O primeiro reflete o propósito do *pattern*, podendo ser um *pattern* de criação, estrutural ou comportamental. *Patterns* criacionais estão relacionados à criação de objetos, estruturais a composição das classes ou objetos e os comportamentais que caracterizam as formas em que as classes ou objetos interagem e distribuem responsabilidades (Gamma, et al., 1994).
- O segundo critério, chamado de escopo, especifica se o *pattern* lida com o relacionamento entre classes ou com o relacionamento entre objetos.

Tabela 1: Classificação dos Design patterns

		Propósito		
		Criacional	Estrutural	Comportamental
Escopo	Classe	<i>Factory Method</i>	<i>Adapter</i>	<i>Interpreter</i> <i>Template Method</i>
	Objeto	<i>Abstract Factory</i> <i>Builder</i> <i>Prototype</i> <i>Singleton</i>	<i>Adapter</i> <i>Bridge</i> <i>Composite</i> <i>Decorator</i> <i>Facade</i> <i>Flyweight</i> <i>Proxy</i>	<i>Chain of Responsibility</i> <i>Command</i> <i>Iterator</i> <i>Mediator</i> <i>Memento</i> <i>Observer</i> <i>State</i> <i>Strategy</i> <i>Visitor</i>

Fonte: (Gamma, et al., 1994)

Para este trabalho foram escolhidos alguns *design patterns* que costumam ser mais utilizados em projetos de *software*: *Abstract Factory*, *Composite*, *Flyweight* (Khomh & Gueheneuc, 2008), *Visitor*, *Decorator*, *Template Method*, *Factory Method* (Zhang & Budgen, 2012)

2.2.2 *Design pattern: Composite*

De acordo com Gamma, et al. (1994):

Intenção:

Compor objetos em estruturas de árvore para representarem hierarquias parte-todo. *Composite* permite aos seus utilizadores tratarem objetos individuais e composições de objetos de maneira uniforme.

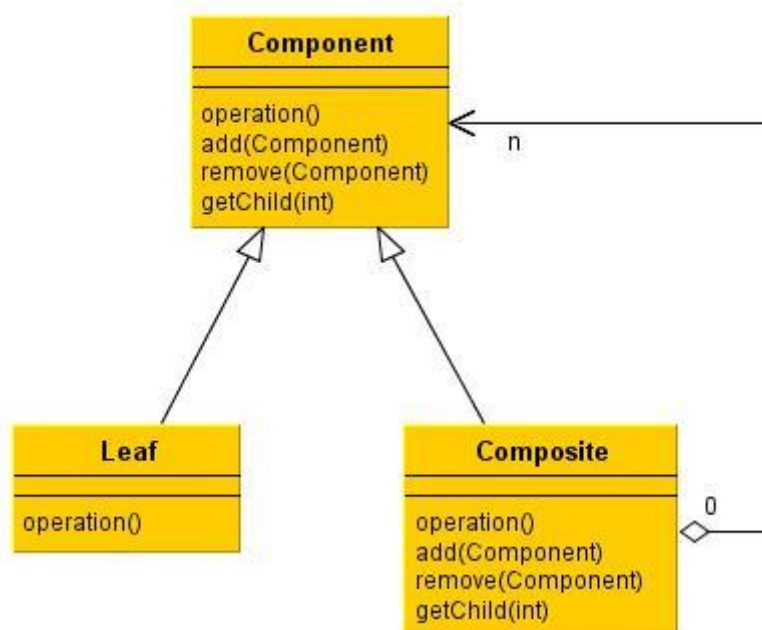
Motivação:

Alguns sistemas permitem construir componentes complexos a partir de componentes simples. O usuário pode agrupar componentes menores para formar componentes maiores. Estes componentes maiores também podem ser agrupados para formar componentes ainda maiores. Um exemplo são aplicações gráficas, como editores de desenho: Uma implementação simples pode definir classes para primitivas gráficas, como texto, linhas e outras classes para atuarem como containers. O padrão *Composite* descreve como usar a composição recursiva de maneira que os usuários não tenham que fazer a distinção entre as classes primitivas e as de container. A chave para o padrão *Composite* é uma classe abstrata que representa tanto as primitivas como os containers.

Aplicabilidade:

O uso do *Composite* deve ocorrer para:

- Representar hierarquias parte-todo de objetos.
- Que os utilizadores sejam capazes de ignorar a diferença entre composições de objetos e objetos individuais, ou seja, ambos serão tratados de maneira uniforme.

Estrutura:Figura 1: Estrutura do *design pattern Composite*

Fonte: (Gamma, et al., 1994)

2.2.3 Design pattern: Factory Method

De acordo com Gamma, et al. (1994):

Intenção:

Definir uma interface para criar objetos, mas permitir que suas subclasses decidam qual classe instanciar.

Motivação:

Esse *design pattern* é utilizado quando existe a necessidade de separar a interface de uma classe de sua implementação (da classe concreta). Esse cenário é encontrado frequentemente em *frameworks*, que utilizam classes abstratas para definir e manter relacionamentos entre objetos. Utilizando o *Factory Method*, o cliente implementa as funcionalidades esperadas pelo *framework* sem que ele tenha o conhecimento de como e qual lógica foi implementada.

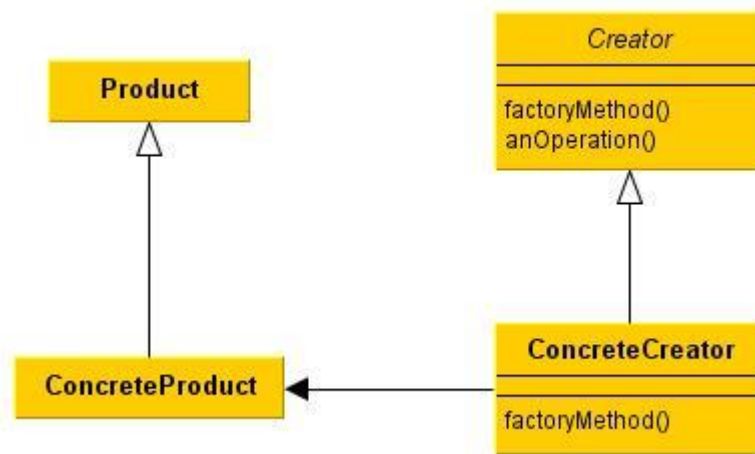
Aplicabilidade:

O uso do *Factory Method* ocorre quando:

- Uma classe não pode antecipar a classe de objetos que deve criar
- Classes delegam a responsabilidade de instanciação para uma entre muitas subclasses.

Estrutura:

Figura 2: Estrutura do *design pattern Factory Method*



Fonte: (Gamma, et al., 1994)

2.2.4 Design pattern: Abstract Factory

De acordo com Gamma, et al. (1994):

Intenção:

Disponibilizar uma interface para criar uma família de objetos relacionados ou dependentes sem especificar suas classes concretas.

Motivação:

Quando se deseja isolar a aplicação da implementação da classe concreta, que poderia ser um componente ou framework específico no qual a aplicação conheceria apenas uma interface e a implementação concreta seria conhecida apenas em tempo de execução ou compilação.

Exemplo, se uma aplicação precisasse ser implementada de forma a oferecer suporte a plataformas diferentes: desktop e mobile. Seria definido uma família de componentes para cada plataforma e uma fábrica que os instancia de acordo com a plataforma alvo na qual a aplicação estará sendo executada.

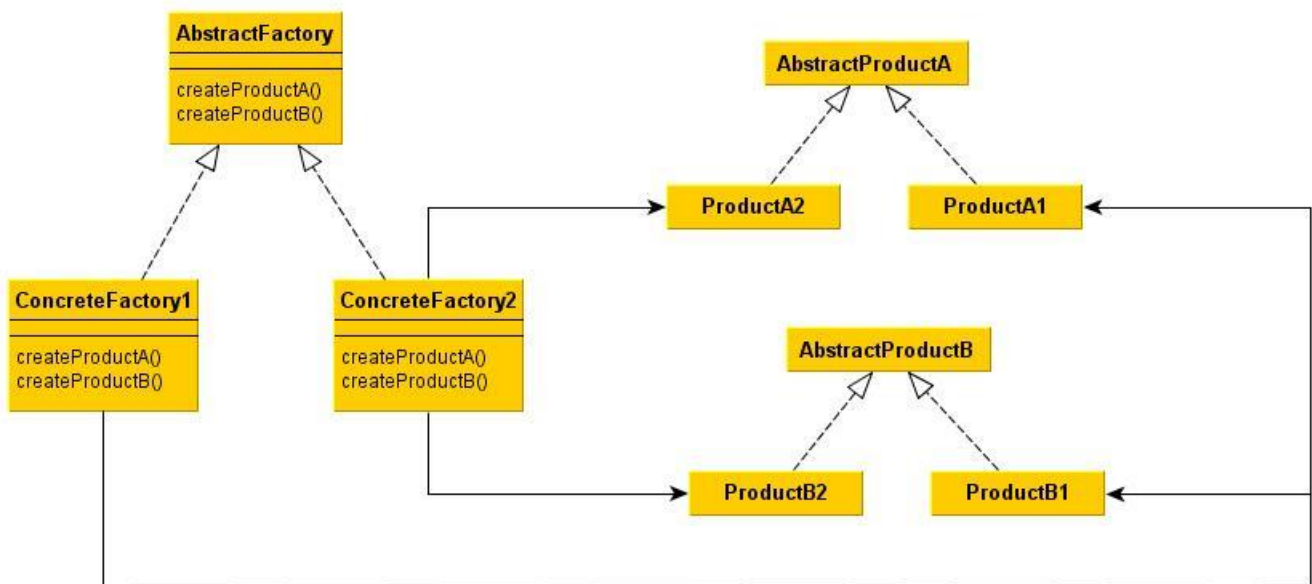
Aplicabilidade:

O uso do *Abstract Factory* ocorre quando:

- Um sistema deve ser independente de como seus produtos são criados, compostos e representados.
- Um sistema deve ser gerado utilizando as configurações de um produto (dentre diversos produtos disponíveis)
- Se deseja prover uma classe de biblioteca de produtos e disponibilizar apenas suas interfaces, não suas implementações.

Estrutura:

Figura 3: Estrutura do *design pattern Abstract Factory*



Fonte: (Gamma, et al., 1994)

2.2.5 Design pattern: Flyweight

De acordo com Gamma, et al. (1994):

Intenção:

Usar compartilhamento para suportar de forma eficiente grandes quantidades de objetos.

Motivação:

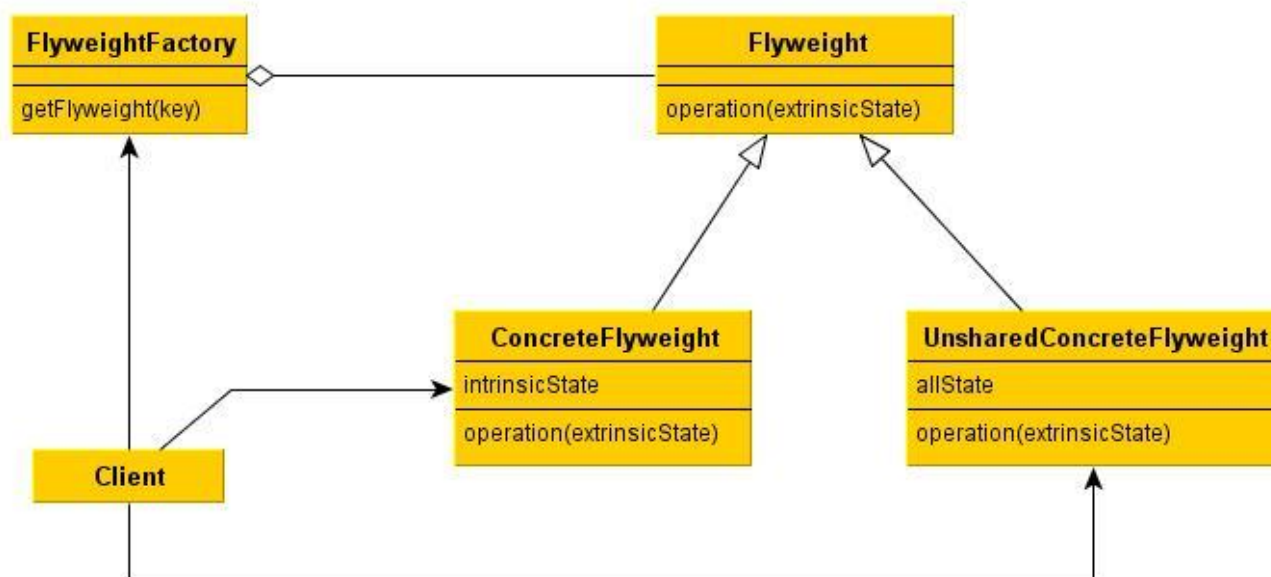
Reduzir o consumo de memória e reduzir custo de tempo de execução. Por exemplo, um editor de texto onde cada caractere é representado por um objeto - pode não haver recursos (memória) suficientes para textos muito grandes.

Aplicabilidade:

- A aplicação utiliza um grande número de objetos.
- Os custos de armazenagem são altos devido ao grande número de objetos.
- Muitos grupos de objetos podem ser substituídos por poucos objetos compartilhados.
- A aplicação não depende da identidade dos objetos, uma vez que os objetos podem ser compartilhados.

Estrutura:

Figura 4: Estrutura do *design pattern* Flyweight



Fonte: (Gamma, et al., 1994)

2.2.6 Design pattern: Template Method

De acordo com Gamma, et al. (1994):

Intenção:

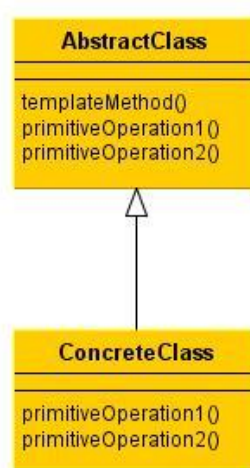
Definir o esqueleto de um algoritmo em uma operação, postergando alguns passos para subclasses, dessa forma as subclasses podem redefinir certos passos do algoritmo sem mudar a estrutura do mesmo.

Motivação:

Permitir que subclasses realizem a implementação de parte de um algoritmo, dessa forma cada implementação pode variar a implementação do trecho específico.

Aplicabilidade:

- Para implementar as partes de um algoritmo que são fixas (não variam) e deixar para as subclasses a implementação do comportamento não fixa (específica, que irá variar).
- Para unificar códigos iguais de mais de uma subclasse em uma única superclasse, evitando duplicação de código.

Estrutura:Figura 5: Estrutura do *design pattern Template Method*

Fonte: (Gamma, et al., 1994)

2.2.7 Design pattern: Visitor

De acordo com Gamma, et al. (1994):

Intenção:

Representar uma operação a ser executada nos elementos de uma estrutura de objetos. *Visitor* permite definir uma nova operação sem mudar as classes dos elementos sobre os quais opera.

Motivação:

Considere um compilador que representa programas como árvores sintáticas abstratas. Ele precisará realizar diversas operações nesta árvore, como verificação

se todas variáveis foram definidas, otimização de código, análise de fluxo entre outras. Distribuir estas operações em várias classes torna o sistema mais difícil de ser compreendido e mantido.

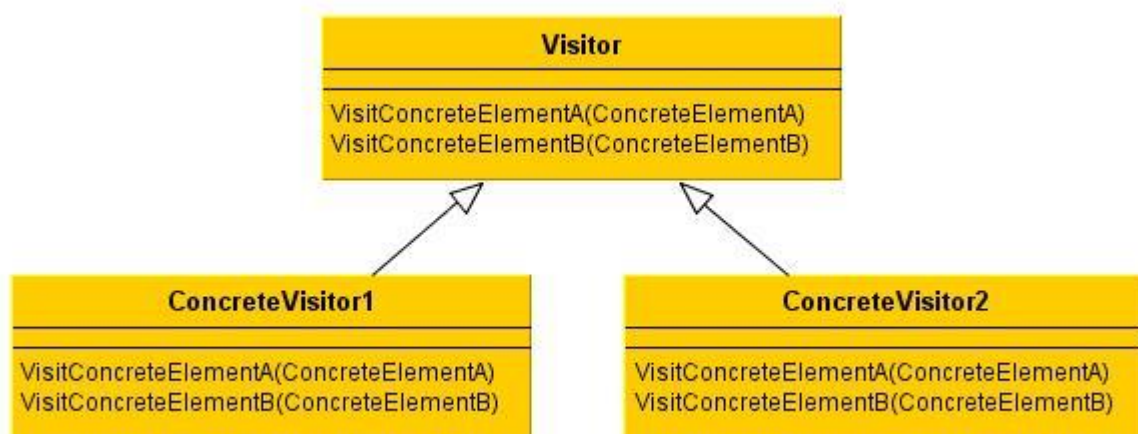
Com a solução do padrão *Visitor*, encapsula-se em uma classe os comportamentos de uma mesma operação, quando atuando em nós diferentes. Desta forma introduz-se uma nova hierarquia.

Aplicabilidade:

- Uma estrutura de objetos contém várias classes de objetos com diferentes interfaces, e se deseja fazer operações nesses objetos que dependam de suas classes concretas.
- Queremos separar as operações dos objetos alvo para não “poluir” o código. *Visitor* nos permite manter as operações definindo-as em uma única classe.
- O conjunto de objetos-alvo raramente muda (uma vez que cada novo objeto requer novos métodos em todos os *Visitors*), mas frequentemente deseja-se definir novas operações.

Estrutura:

Figura 6: Estrutura do *design pattern Visitor*



Fonte: (Gamma, et al., 1994)

2.2.8 Design pattern: Decorator

De acordo com Gamma, et al. (1994):

Intenção:

Acrescentar responsabilidades a um objeto dinamicamente. Prover alternativa flexível ao uso de subclasses para se estender a funcionalidade de uma classe

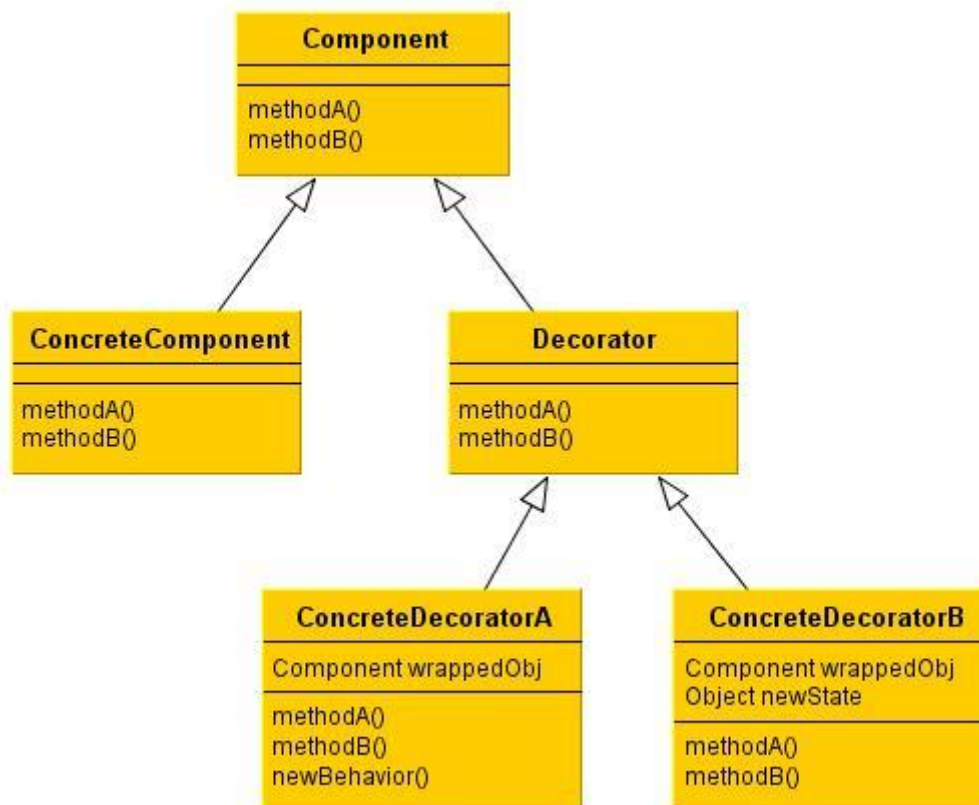
Motivação:

Algumas vezes se quer adicionar responsabilidades para objetos individuais, e não para toda a classe.

O objeto usado possui as funcionalidades básicas, mas é necessário adicionar funcionalidades adicionais a ele que podem ocorrer antes ou depois da funcionalidade básica.

Aplicabilidade:

- Para adicionar responsabilidades a objetos individuais de forma dinâmica e transparente, sem afetar outros objetos.
- Para responsabilidades que podem ser removidas
- Quando a extensão através de herança é impraticável. Algumas vezes uma grande quantidade de extensões independentes são possíveis; e seria necessário um imenso número de subclasses para suportar cada combinação possível entre elas.
- Quando uma definição de classe pode estar escondida ou não disponível para herdar.

Estrutura:Figura 7: Estrutura do *design pattern Decorator*

Fonte: (Gamma, et al., 1994)

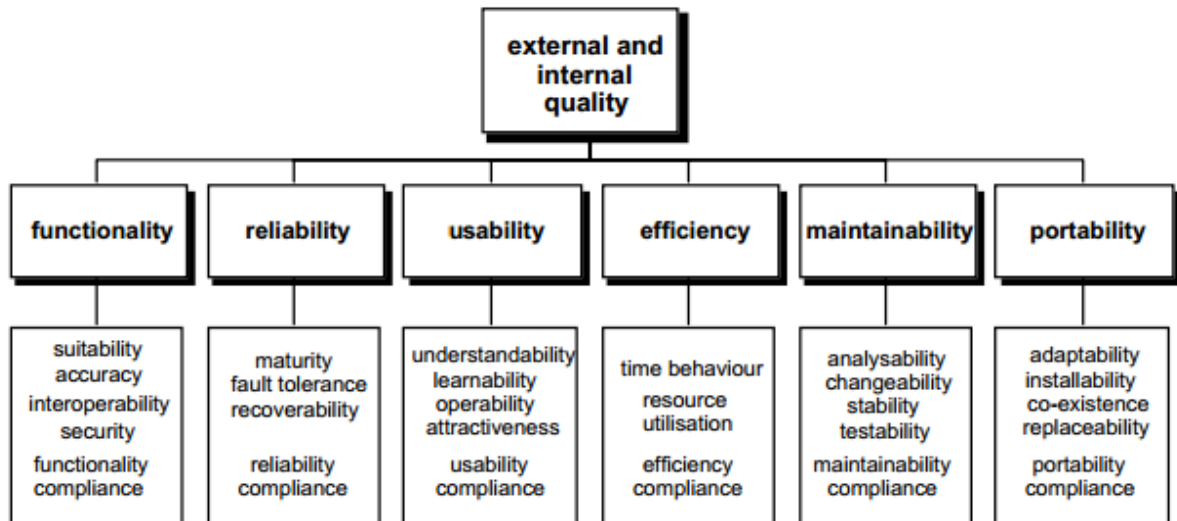
2.3 Atributos de Qualidade de Produtos de Software

Desenvolver ou selecionar produtos de *software* de alta qualidade é de primordial importância. Especificação e avaliação da qualidade do produto de *software* são fatores chave para garantir qualidade adequada. Isto pode ser alcançado pela definição apropriada das características de qualidade, levando em consideração o uso pretendido do produto de *software*. É importante que cada característica relevante de qualidade do produto de *software* seja especificada e avaliada utilizando, quando possível, métricas validadas ou amplamente aceitas. (ISO/IEC, 2000)

A norma ISO 9126 propõe atributos de qualidade, distribuídos em seis características principais: Funcionalidade, Confiabilidade, Usabilidade, Eficiência,

Manutenibilidade e Portabilidade. Cada característica é dividida em sub-características, que podem ser medidas por meio de métricas externas e internas.

Figura 8: Características e sub-características de qualidade



Fonte: ISO/IEC FDIS 9126-1:2000

Atributos de qualidade são os fatores gerais que afetam o comportamento em tempo de execução, o projeto do sistema e a experiência do usuário. Eles representam áreas de preocupação que têm o potencial de grande impacto na aplicação entre camadas. Alguns desses atributos são relacionados com a concepção geral do sistema, enquanto outros são específicos em tempo de execução, *design* de projeto, ou problemas centrados no usuário. A medida em que a aplicação possui uma combinação desejada de atributos de qualidade como usabilidade, desempenho, confiabilidade e segurança, indica o sucesso do projeto e da qualidade geral do aplicativo de *software*. (msdn, 2014)

2.4 Estudos dos impactos dos *design patterns* em projetos de *software*

Alguns estudos foram realizados para identificar o real impacto dos *design patterns* em projetos de *software*. O objetivo desses estudos era quantificar e qualificar o impacto na qualidade geral do sistema para confirmar ou refutar a hipótese que os

design patterns impactam positivamente no projeto. Neste trabalho foram escolhidos dois estudos:

- 1- Khomh & Gueheneuc: realizaram pesquisa com vinte engenheiros de software com experiência comprovada em *design patterns*.
- 2- Ali & Elish: compilaram dezenove estudos a respeito do impacto dos *design patterns* GoF sobre atributos de qualidade de *software*

Estes dois estudos foram escolhidos para serem detalhados pois possuem diferentes perfis (o primeiro é baseado na experiência empírica dos desenvolvedores e o segundo é baseado em diversos outros estudos).

2.4.1 Estudo de Khomh & Gueheneuc

No estudo de Khomh & Gueheneuc (2008) foram realizadas pesquisas, utilizando questionário, para que vinte engenheiros de *software* com experiência comprovada no uso de *design patterns* do GOF em desenvolvimento e manutenção de *software* avaliassem o impacto dos *design patterns* na qualidade dos sistemas, com base nas suas experiências profissionais.

Khomh & Gueheneuc (2008) escolheram alguns conjuntos de atributos de qualidade, entre eles:

Atributos relacionados ao *design*:

- Expansibilidade: O grau em que um *design* pode ser expandido.
- Reusabilidade: O grau em que parte do *design* pode ser reusado em outro local

Atributo relacionado à implementação:

- Compreensibilidade: O grau em que o código fonte de um sistema pode ser facilmente entendido

O resultado da pesquisa (Khomh & Gueheneuc, 2008) é apresentado na Tabela 2.

Tabela 2: Estimativa do impacto dos design patterns em três atributos de qualidade

<i>Design patterns</i>	Expansibilidade	Compreensibilidade	Reusabilidade
<i>A.Factory</i>	+	-	+
<i>Builder</i>	+	+	-
<i>F. Method</i>	+	-	+
<i>Prototype</i>	+	+	+
<i>Singleton</i>	-	+	-
<i>Adapter</i>	+	-	+
<i>Bridge</i>	+	+	-
<i>Composite</i>	+	+	+
<i>Decorator</i>	+	-	-
<i>Facade</i>	+	+	-
<i>Flyweight</i>	-	-	-
<i>Proxy</i>	-	-	+
<i>Ch. Of Resp</i>	+	-	+
<i>Command</i>	+	-	-
<i>Interpreter</i>	+	+	+
<i>Iterator</i>	+	+	+
<i>Mediator</i>	+	+	-
<i>Memento</i>	-	-	-
<i>Observer</i>	+	-	+
<i>State</i>	+	+	-
<i>Strategy</i>	+	+	-
<i>T.Method</i>	+	-	+
<i>Visitor</i>	+	-	-

Fonte: (Khomh & Gueheneuc, 2008)

Legenda:

O sinal + representa que o impacto do uso do *pattern* em determinado atributo de qualidade é positivo, enquanto que o sinal - indica que o impacto do *pattern* é negativo ou neutro no atributo de qualidade.

Os três atributos de qualidade apresentados neste estudo são utilizados no estudo de *design patterns* no capítulo 3. O resultado desse estudo também é levado em consideração para criação da lista de recomendações para o uso adequado dos *patterns* (capítulo 4).

2.4.2 Estudo de Ali & Elish

Ali & Elish (2013) compilaram dezenove estudos a respeito do impacto dos *design patterns* GoF sobre atributos de qualidade de *software*, e destes estudos 17 foram escolhidos para serem analisados: Em 9 estudos foram realizados experimentos para analisar os efeitos dos design patterns no atributo de qualidade Manutenção (Maintenance). Em outros 3 estudos foi analisada a evolução dos design patterns e como eles afetam as classes em que são utilizados. Outros 2 estudos fazem a análise do impacto do uso de design patterns na performance dos sistemas em que são utilizados. Os últimos 3 estudos analisam o uso dos design patterns e as falhas que são encontradas nos softwares.

Figura 9: Design patterns analisados nos 17 estudos escolhidos

Pattern type	Quality attribute	Maintainability										Evolution and change-proneness			Performance		Faults		
		[1]	[2]	[3]	[4]	[5]	[6]	[7]	[11]	[12]	[14]	[18]	[19]	[13]	[15]	[8]	[9]	[17]	
Creational patterns	Pattern																		
	Abs. Factory	√	√	√	√		√					√				√	√		
	Builder												√					√	
	Factory Method												√					√	
	Singleton															√	√	√	
Structural patterns	Adapter											√	√	√		√	√	√	
	Bridge																		
	Composite	√	√	√	√	√	√	√	√							√			
	Decorator	√	√	√	√	√	√	√								√	√		
	Facade														√				
	Flyweight																		
	Proxy												√	√		√		√	
Behavioral Patterns	Chain of Resp.								√										
	Command											√	√	√		√		√	
	Interpreter																		
	Iterator												√	√				√	
	Mediator																		
	Memento																		
	Observer	√	√			√						√				√	√		
	State								√				√	√	√	√	√	√	
	Strategy												√	√		√		√	
	Template															√	√		
	Visitor	√	√				√			√		√	√	√				√	

Fonte: (Ali & Elish, 2013)

A figura 9 apresenta os design patterns e os atributos de qualidade analisados em cada estudo.

O resultado de cada estudo foi:

Estudos 1 e 12: De forma geral, os design patterns avaliados impactaram positivamente no atributo de qualidade Manutenabilidade, ou seja, a manutenção do sistema se tornou mais fácil com o uso dos design patterns.

Estudo 2: No atributo de qualidade Manutenabilidade, os design patterns Observer e Decorator impactaram positivamente, o Visitor impactou negativamente, o Composite também impactou negativamente mas não tanto quanto o Visitor e o Abstract Factory teve comportamento neutro.

Estudo 3: No atributo de qualidade Manutenabilidade, os design patterns Abstract Factory e Composite impactaram negativamente, o Decorator teve comportamento neutro.

Estudos 4, 6, 7, 11: De forma geral, os design patterns avaliados impactaram negativamente no atributo de qualidade Manutenabilidade

Estudo 5: De forma geral, os design patterns avaliados tiveram comportamento neutro no atributo de qualidade Manutenabilidade

Estudos 14, 18, 19: De forma geral, os design patterns avaliados impactaram negativamente no atributo de qualidade *Evolution* e *Change-proneness*, ou seja, mais mudanças ocorreram no sistema com o uso dos design patterns.

Estudo 13: No atributo de qualidade Performance, o design pattern State impactou positivamente para sistemas complexo e negativamente para sistemas simples.

Estudo 15: No atributo de qualidade Performance, o design pattern Facade teve uma performance melhor que no uso do design pattern Command.

Estudo 8: No atributo de qualidade Propensão a falhas, de forma geral, os design patterns impactaram de forma neutra, exceto o Adapter que impactou negativamente e o Observer que impactou positivamente.

Estudo 9: No atributo de qualidade Propensão a falhas os design patterns Observer e Singleton impactaram negativamente, o Factory Method impactou positivamente. O Template e o Decorator não foram possíveis de serem avaliados.

Estudo 17: No atributo de qualidade Propensão a falhas, de forma geral, os design patterns impactaram de forma negativa.

Os *design patterns Visitor, Abstract Factory e Decorator*, estudados por Ali & Elish (2013), são utilizados no estudo de *design patterns* no capítulo 3. O resultado desse estudo também é levado em consideração para criação da lista de recomendações para o uso adequado dos *patterns* (capítulo 4).

2.5 Considerações do Capítulo

Neste capítulo foi apresentado o conceito de Design pattern e foram detalhados os design patterns utilizados nos demais capítulos deste trabalho. Neste capítulo também foram apresentados dois trabalhos: de Khomh & Gueheneuc e de Ali & Elish, onde foram estudados alguns design patterns sobre alguns atributos de qualidade específicos. Os design patterns e os atributos de qualidade apresentados nesses trabalhos são utilizados tanto no capítulo 3 quanto no capítulo 4 deste trabalho.

3. IDENTIFICAÇÃO DOS IMPACTOS DO USO DE DESIGN PATTERNS

O projeto de *software* utilizado para a identificação dos impactos do uso de *design patterns* é o Sistema de Transporte Inteligente (SITUR), o mesmo que foi utilizado durante o curso MBA do PECE (Programa de Educação Continuada Poli USP) de Tecnologia de *Software*. Os *design patterns*, mencionados no capítulo 2, são aplicados na parte do projeto em que melhor se adequa ao *design pattern* em questão.

3.1 Projeto de *Software*

O Sistema de Transporte Inteligente - SITUR é um sistema que auxilie a locomoção de cidadãos através de transporte público. Tem como objetivos:

- Agilizar a compra de créditos por parte do usuário para utilização em transportes públicos na cidade;
- Autenticar rapidamente a passagem do usuário no meio de transporte;
- Disponibilizar ao usuário os melhores meios de transporte para chegar ao seu destino.

O sistema possui grande número de funções e é complexo por possuir transferência de dados utilizando rede de celular, servidores, aplicativo móvel, banco de dados e alto número de usuários simultâneos. Desta forma o sistema deve ser altamente modularizado e facilmente expandido (para novos módulos / funcionalidades sejam facilmente implementados), deve ser possível reutilizar componentes / módulos no projeto com pouca, ou nenhuma, modificação.

A equipe que irá desenvolvê-lo é uma equipe novata da empresa contratada para desenvolver o *software*, sem experiência com projetos desse porte e também sem experiência com o uso prático de *design patterns*, desta forma o *design* deve ser simples e o código fonte de fácil compreensão.

3.2 Processos de Desenvolvimento

As fases do processo de desenvolvimento são:

- Comunicação: Início de projeto, identificação das necessidades da prefeitura da cidade e levantamento dos requisitos para o projeto SITUR.
- Planejamento: Estudo de identificação dos principais riscos do projeto, recursos necessários que serão utilizados no projeto SITUR, os produtos de trabalho que serão produzidos, definição das estimativas e criação do cronograma do projeto
- Modelagem: Criação dos modelos de análise e projeto necessários para melhor entendimento do cliente do projeto SITUR e para auxiliar os desenvolvedores na criação do projeto.
- Construção: Codificação e testes
- Implantação: A fase de implantação será realizada por outra empresa, a ser contratada pelo cliente (prefeitura da cidade)

3.3 Atributos de qualidade

Atributos de qualidade a serem analisados são observados nas fases de modelagem e construção do projeto.

Fase de Modelagem (*Design*):

- Reusabilidade (*Reusability*): O grau em que um *design* pode ser reusado em outro componente / módulo no projeto com pouca, ou nenhuma, modificação.
- Expansibilidade (*Expandability*): O grau em que um *design* pode ser expandido.

- Simplicidade (*Simplicity*): O grau em que um *design* de um sistema pode ser facilmente entendido.

Fase de desenvolvimento de Construção (Implementação):

- Compreensibilidade (*Understandability*): O grau em que o código fonte de um sistema pode ser facilmente entendido.
- Apreensibilidade (*Learnability*): O grau em que o código fonte de um sistema pode ser facilmente aprendido.
- Modularidade (*Modularity*): O grau em que o *software* é composto por componentes, permutáveis separados, cada um dos quais realiza uma função e contém todos os elementos necessários para conseguir isso.

Estes foram os atributos de qualidade escolhidos para serem estudados pois, como apresentado no item 3.1, são os atributos que combinam com as características do sistema e da equipe desenvolvedora do sistema, como pode ser visto na tabela 3.

Tabela 3: Características e Atributos de Qualidade do Sistema

Característica	Atributo de Qualidade
Altamente modularizado	Modularidade (<i>Modularity</i>)
Facilmente expandido	Expansibilidade (<i>Expandability</i>)
Altamente reutilizável	Reusabilidade (<i>Reusability</i>)
Design simples	Simplicidade (<i>Simplicity</i>)
Código fonte de fácil compreensão	Compreensibilidade (<i>Understandability</i>)
Código fonte facilmente aprendido	Apreensibilidade (<i>Learnability</i>)

Os atributos de qualidade descritos na tabela 3 são subcaracterísticas da norma ISO 9126: Os atributos Modularidade, Expansibilidade e Reusabilidade são subcaracterísticas de Funcionalidade (*Functionality*), já o atributo Simplicidade faz parte da característica Manutenibilidade (*Maintainability*) e a Compreensibilidade e Apreensibilidade são subcaracterísticas da Usabilidade (*Usability*).

3.4 Design patterns analisados

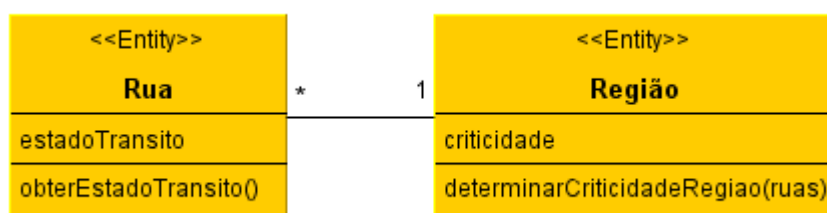
Os *Design Patterns* analisados neste capítulo são: *Composite*, *Factory Method*, *Abstract Factory*, *Flyweight*, *Template Method*, *Visitor* e *Decorator*. Estes foram os *design patterns* escolhidos para serem analisados pois costumam ser os mais utilizados em projetos de software. Além disso, estes *design patterns* foram também utilizados nos estudos de Khomh & Gueheneuc (2008) e Zhang & Budgen (2012).

3.4.1 Composite

A aplicação deste *pattern* foi analisada na modelagem da funcionalidade de identificação de regiões críticas com base no estado do trânsito, onde o sistema busca por ruas com estado de trânsito congestionado e identifica a criticidade da região. Caso de uso analisado: Identificar Regiões Críticas.

Na versão sem implementação do *design pattern Composite*, como pode ser visto na figura 10, o diagrama de projeto possui apenas a classe Rua associada à classe Região: Diversas ruas formam uma região, e a região é capaz de determinar sua criticidade (se está congestionada ou não) com base no estado do trânsito das ruas que a formam. As classes de *Boundary* (Interface) e *Controller* (Controlador) foram removidas para exibir o diagrama de forma simplificada.

Figura 10: Diagrama de classe original do caso de uso Identificar Regiões Críticas

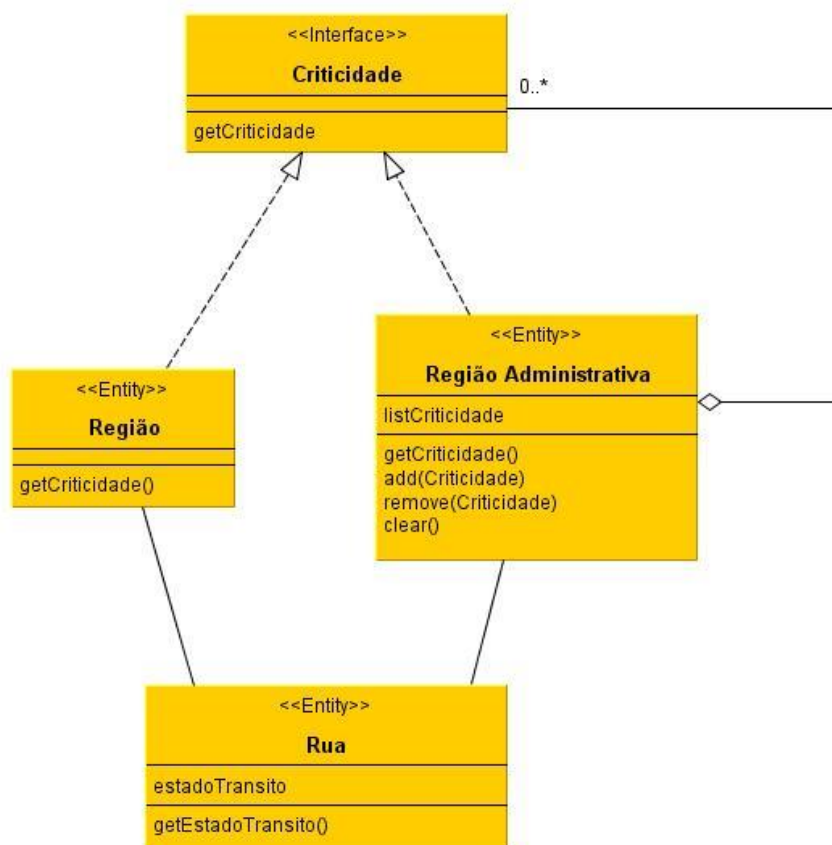


Fonte: Elaborado pelo autor.

Para utilizar o *design pattern Composite*, a versão original do diagrama foi modificada para incluir a classe Região Administrativa. Essa classe representa uma área maior que uma região convencional, ela possui uma lista de “Críticidade”, ou seja, pode ser formada por diversas Regiões ou por diversas Regiões Administrativas (ou uma mistura de ambas).

O Diagrama refatorado utilizando *design pattern Composite* pode ser observado na figura 11.

Figura 11: Diagrama de Classe do caso de uso Identificar Regiões Críticas, com aplicação do *design pattern Composite*



Fonte: Elaborado pelo autor.

Abaixo segue um exemplo da implementação do *design pattern* (na linguagem Java):

```
public interface Criticidade {

    public Criticidade getCriticidade();

}

public class Regiao implements Criticidade {

    @Override
    public Criticidade getCriticidade(){
        //Returns criticidade
    }

}

public class RegiaoAdministrativa implements Criticidade {

    List<Criticidade> listCriticidade = new
    ArrayList<Criticidade>();

    public void add(Criticidade criticidade) {
        listCriticidade.add(criticidade);
    }

    public void remove(Criticidade criticidade) {
        listCriticidade.remove(criticidade);
    }

    @Override
    public Criticidade getCriticidade(){
        Criticidade criticidade = new Criticidade();

        foreach(criticidade c : listCriticidade) {
            //realiza o cálculo da criticidade
        }

        return criticidade
    }

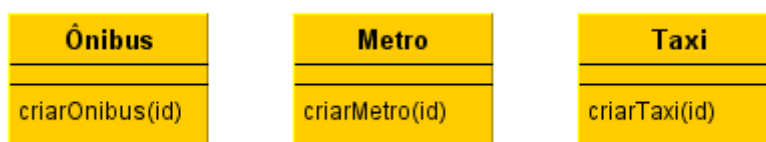
}
```

3.4.2 *Factory Method*

A aplicação deste *pattern* é analisada na modelagem da funcionalidade de criação de objetos do tipo Taxi, Metrô e Ônibus. Um objeto para cada veículo deve existir para que operações possam ser realizadas como, por exemplo, obtenção da posição do veículo.

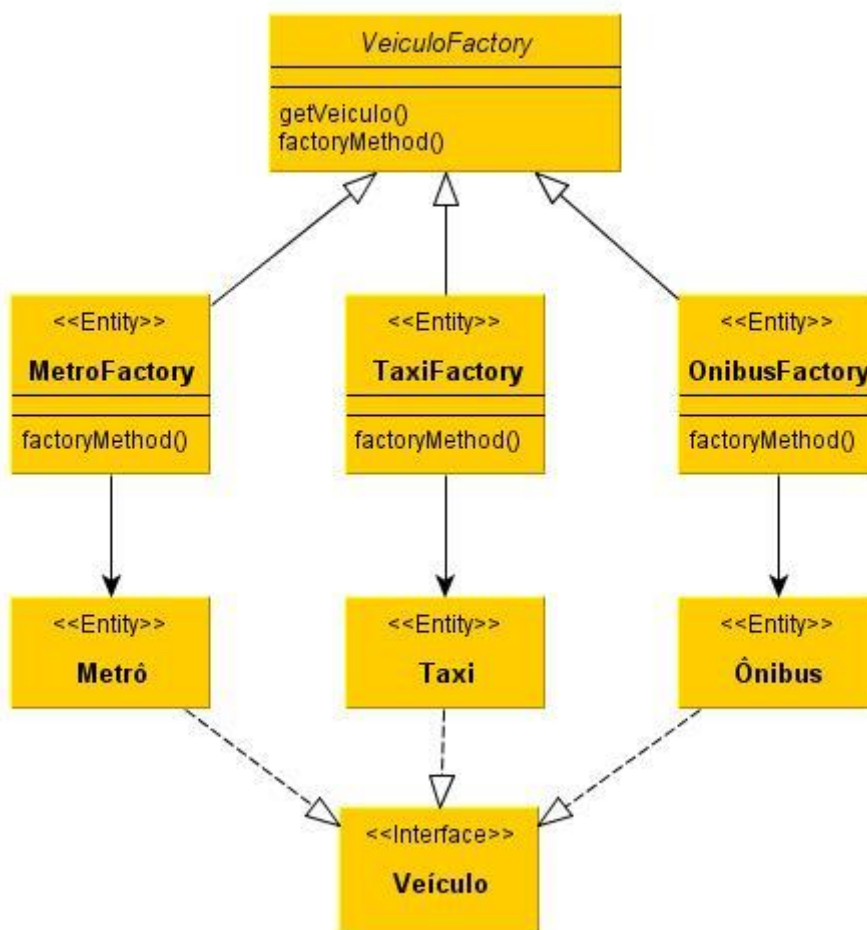
Na versão sem implementação do *design pattern Factory Method*, como pode ser visto na figura 12, o diagrama de projeto possui apenas as classes Ônibus, Metro e Taxi.

Figura 12: Diagrama de classe original para criação dos meios de transporte



Fonte: Elaborado pelo autor.

Figura 13: Diagrama de classes da funcionalidade de criação de objetos de cada veículo, com aplicação do *design pattern Factory Method*



Fonte: Elaborado pelo autor.

Para utilizar o *design pattern Factory Method*, como pode ser visto na Figura 13, foram criadas as classes:

Classe abstrata VeiculoFactory: Classe Abstrata, com o método abstrato `factoryMethod`. Esse método será implementado pelas classes factory dos meios de transporte. Esta classe também possui o método `getVeiculo()`, responsável por chamar o método `factoryMethod` e retornar o veículo criado.

Interface Veículo: É apenas uma interface genérica para representar Veículos.

Entidades Metrô, Taxi e Ônibus: São as classes concretas que implementam a interface Veículo, representam os modais (meios de transporte).

Entidades MetroFactory, TaxiFactory e ÔnibusFactory: São as classes concretas que implementam o método factoryMethod (herdado da classe abstrata VeiculoFactory). A classe MetroFactory é responsável por criar corretamente as instâncias da classe Metrô, bem como a classe TaxiFactory é responsável por criar corretamente as instâncias da classe Taxi e a classe ÔnibusFactory é responsável por criar corretamente as instâncias da classe Ônibus.

A seguir é mostrado um exemplo da implementação do *design pattern* (na linguagem Java):

```
public interface Veiculo {
}

public abstract class VeiculoFactory {

    public Veiculo getVeiculo() {
        Veiculo veiculo = factoryMethod();
        //algoritmo do metodo
        return veiculo;
    }

    public abstract Veiculo factoryMethod();
}

public class Metro implements Veiculo {
    //Implementacao da classe Metro
}

public class Taxi implements Veiculo {
    //Implementacao da classe Taxi
}

public class Onibus implements VeiculoFactory {
    //Implementacao da classe Onibus
}

public class MetroFactory extends VeiculoFactory {
    protected Veiculo factoryMethod() {
        //algoritmo do metodo
        return new Metro();
    }
}
```

```
public class TaxiFactory extends VeiculoFactory {
    protected Veiculo factoryMethod() {
        //algoritmo do metodo
        return new Taxi();
    }
}

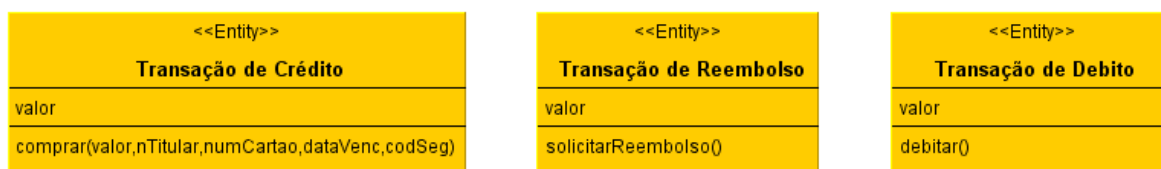
public class OnibusFactory extends VeiculoFactory {
    protected Veiculo factoryMethod() {
        //algoritmo do metodo
        return new Onibus();
    }
}
```

3.4.3 Abstract Factory

A aplicação deste *pattern* foi analisada na modelagem da funcionalidade de criação de objetos do tipo Transação: Reembolso, Débito e Crédito.

Na versão sem implementação do *design pattern Abstract Factory*, como pode ser visto na figura 14, o diagrama de projeto possui apenas as classes Transação de Crédito, Transação de Reembolso e Transação de Débito.

Figura 14: Diagrama de classe original para criação de transações



Fonte: Elaborado pelo autor.

Para a correta implementação do *design pattern Abstract Factory*, foram considerados dois perfis diferentes para criação de transação: As transações feitas para pessoas acima de 60 anos e para pessoas abaixo de 60 anos. Existem esses dois perfis, pois é considerado que cada perfil tem um algoritmo diferente para criação dos objetos, desta forma são utilizadas diferentes classes “factory” (uma para cada perfil)

Como o AbstractFactory é uma fábrica de outras fábricas, o melhor exemplo é demonstrar o *design pattern* utilizando, no mínimo, duas fábricas diferentes.

Figura 15: Diagrama de classes da funcionalidade de criação de veículos e de transações, com aplicação do *design pattern Abstract Factory*



Fonte: Elaborado pelo autor.

Para utilizar o *design pattern Abstract Factory*, como pode ser visto na figura 15, foram criadas as classes:

Interface Reembolso: É uma interface para indicar uma transação do tipo Reembolso – possui o método `reembolsar()`

Interface Débito: É uma interface para indicar uma transação do tipo Débito – possui o método `debitar()`

Interface Crédito: É uma interface para indicar uma transação do tipo Credito – possui o método creditar()

Entidade ReembolsoMenor60Anos: Classe concreta que implementa a interface Reembolso, essa classe possui o algoritmo reembolsar para o perfil do utilizador possui menos de 60 anos.

Entidade ReembolsoMaior60Anos: Classe concreta que implementa a interface Reembolso, essa classe possui o algoritmo reembolsar para o perfil do utilizador possui mais de 60 anos.

Entidade DebitoMenor60Anos: Classe concreta que implementa a interface Debito, essa classe possui o algoritmo debitar para o perfil do utilizador possui menos de 60 anos.

Entidade DebitoMaior60Anos: Classe concreta que implementa a interface Debito, essa classe possui o algoritmo debitar para o perfil do utilizador possui mais de 60 anos.

Entidade CreditoMenor60Anos: Classe concreta que implementa a interface Credito, essa classe possui o algoritmo creditar para o perfil do utilizador possui menos de 60 anos.

Entidade CreditoMaior60Anos: Classe concreta que implementa a interface Credito, essa classe possui o algoritmo creditar para o perfil do utilizador possui mais de 60 anos.

Interface TransacaoFactory: Interface para transação genérica, possui os métodos getReembolso(), getCredito() e getDebito()

Entidade TransacaoMaior60AnosFactory: Classe concreta implementadora da interface TransacaoFactory para perfis com mais de 60 anos

Entidade `TransacaoMenor60AnosFactory`: Classe concreta implementadora da interface `TransacaoFactory` para perfis com menos de 60 anos

Classe `RealizaTransacao`: Classe de entrada para o *Abstract Factory*, quando esta classe é criada, recebe um objeto do tipo `TransacaoMaior60AnosFactory` ou `TransacaoMenor60AnosFactory`.

Abaixo segue um exemplo da implementação do *design pattern* (na linguagem Java):

```
public interface Reembolso {
    public void reembolsar();
}

public interface Debito {
    public void debitar();
}

public interface Credito {
    public void creditar();
}

public class ReembolsoMaior60Anos implements Reembolso {
    public void reembolsar() {
        //Implementacao de reembolso para maiores de 60 anos
    }
}

public class ReembolsoMenor60Anos implements Reembolso {
    public void reembolsar() {
        //Implementacao de reembolso para menores de 60 anos
    }
}

public class DebitoMaior60Anos implements Debito {
    public void debitar() {
        //Implementacao de debito para maiores de 60 anos
    }
}

public class DebitoMenor60Anos implements Debito {
    public void debitar() {
        //Implementacao de debito para menores de 60 anos
    }
}

public class CreditoMaior60Anos implements Credito {
    public void creditar() {
        //Implementacao de credito para maiores de 60 anos
    }
}
```

```
}

public class CreditoMenor60Anos implements Credito {
    public void creditar() {
        //Implementacao de credito para menores de 60 anos
    }
}

public interface TransacaoFactory {
    public Reembolso getReembolso(String valor);
    public Debito getDebito(String valor);
    public Credito getCredito(String valor);
}

public class TransacaoMaior60AnosFactory implements TransacaoFactory
{
    public Reembolso getReembolso(String valor) {
        return new ReembolsoMaior60Anos(valor);
    }

    public Debito getDebito(String valor) {
        return new DebitoMaior60Anos(valor);
    }

    public Credito getCredito(String valor){
        return new CreditoMaior60Anos(valor);
    }
}

public class TransacaoMenor60AnosFactory implements TransacaoFactory
{
    public Reembolso getReembolso(String valor) {
        return new ReembolsoMenor60Anos(valor);
    }

    public Debito getDebito(String valor) {
        return new DebitoMenor60Anos(valor);
    }

    public Credito getCredito(String valor){
        return new CreditoMenor0Anos(valor);
    }
}
```

```
public class RealizaTransacao() {
    private TransacaoFactory transFactory;

    public Trans(TransacaoFactory tipoTransacao) {
        this.tipoTransacao = tipoTransacao;
    }

    public void executarTransacao(String transacao, String valor) {
        if(transacao.equals("Reembolso") {
            Reembolso reembolso =
transFactory.getReembolso(valor);
        }

        if(transacao.equals("Debito") {
            Debito debito = transFactory.getDebito(valor);
        }

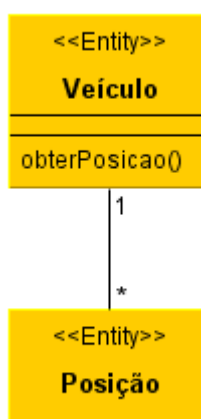
        if(transacao.equals("Credito") {
            Credito credito = transFactory.getCredito(valor);
        }
    }
}
```

3.4.4 Flyweight

A aplicação deste *pattern* foi analisada na modelagem da funcionalidade de coletar e armazenar dados de posição GPS dos veículos para, posteriormente, ser gerada a criticidade das regiões.

Na versão sem implementação do *design pattern Flyweight*, como pode ser visto na Figura 16, o diagrama de classes de projeto possui apenas a classe Veículo associada a classe Posição: Um veículo possui muitas posições e uma posição possui apenas um veículo em um momento específico. As classes de *Boundary* (Interface) e *Controller* (Controlador) foram removidas para exibir o diagrama de forma simplificada.

Figura 16: Diagrama de classes da funcionalidade de coletar e armazenar dados de posição GPS dos veículos

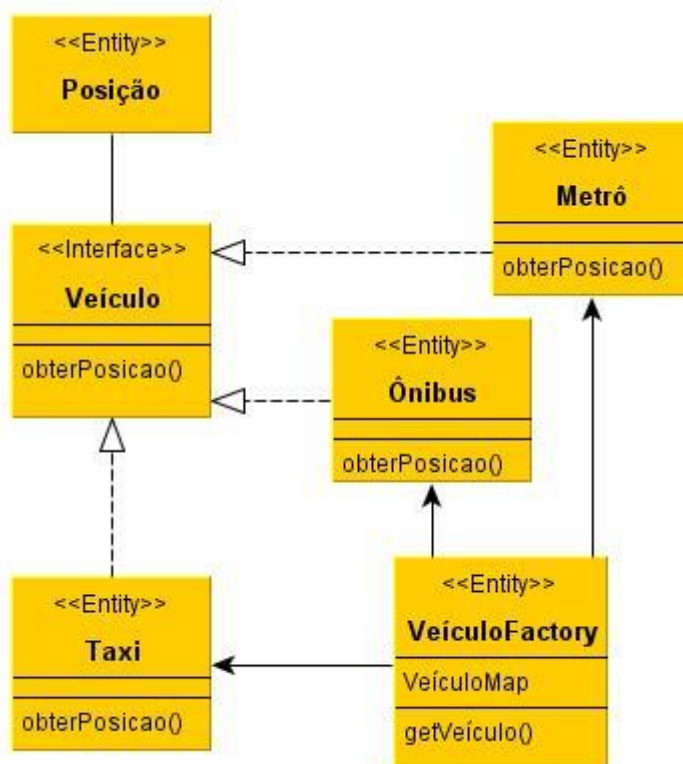


Fonte: Elaborado pelo autor.

Para utilizar o *design pattern Flyweight*, a versão original do diagrama foi modificada (ver Figura 17) e foram incluídas as classes Taxi, Ônibus e Metrô. Essas classes representam os modais cobertos pelo sistema do SITUR. Também foi incluída a classe VeículoFactory, essa classe é responsável por gerenciar as instâncias dos objetos que representam os modais. Toda vez que o sistema precisa de uma instância de algum modal, o método getVeículo() é chamado e o veículo é buscado na coleção veículoMap. Se o veículo desejado não existir na coleção (ainda não ter

seja criada uma instância desse veículo), a classe VeículoFactory irá criar a instância do veículo e irá armazená-la na coleção veículoMap. Na próxima necessidade de uso a instância desse veículo, a instância existente irá ser buscada na coleção, dessa forma não será necessário criar uma instância diferente a cada necessidade de uso do veículo.

Figura 17: Diagrama refatorado utilizando *design pattern Flyweight*



Fonte: Elaborado pelo autor.

Abaixo segue um exemplo da implementação do *design pattern* (na linguagem Java):

```
public interface Veículo {
    public Posicao obterPosicao()
}

public class Taxi implements Veículo {
    @Override
    public Posicao obterPosicao() {
        //Returns posicao
    }
}

public class Ônibus implements Veículo {
    @Override
    public Posicao obterPosicao() {
        //Returns posicao
    }
}

public class Metrô implements Veículo {
    @Override
    public Posicao obterPosicao() {
        //Returns posicao
    }
}

public class VeículoFactory {

    private static HashMap<String, Veículo> veiculoMap = new
HashMap<String, Veículo>();

    public static Veículo getVeículo(String placa) {
        Veículo veiculo = veiculoMap.get(placa);

        if(veiculo == null) {
            veiculo = new Veiculo(placa);
            veiculoMap.put(placa, veiculo);
        }

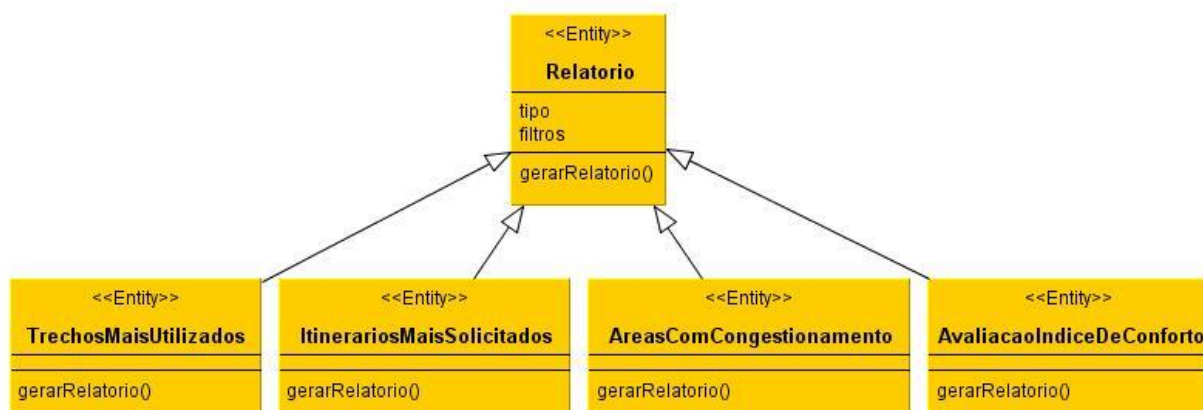
        return veiculo;
    }
}
```

3.4.5 *Template Method*

A aplicação deste *pattern* será analisada na modelagem da funcionalidade de geração de relatórios administrativos. Relatórios de trechos mais utilizados, itinerários mais solicitados, área com congestionamento e avaliação do índice de conforto.

Na versão sem implementação do *design pattern Template Method*, como pode ser visto na Figura 18, o diagrama de projeto possui a classe Relatório, que é estendida pelas classes TrechosMaisUtilizados, ItinerariosMaisSolicitados, AreasComCongestionamento e AvaliacaoIndiceDeConforto. Cada subclasse de relatório possui sua própria implementação do método gerarRelatorio. As classes de *Boundary* (Interface) e *Controller* (Controlador) foram removidas para exibir o diagrama de forma simplificada.

Figura 18: Diagrama de classes da funcionalidade de geração de relatórios administrativos

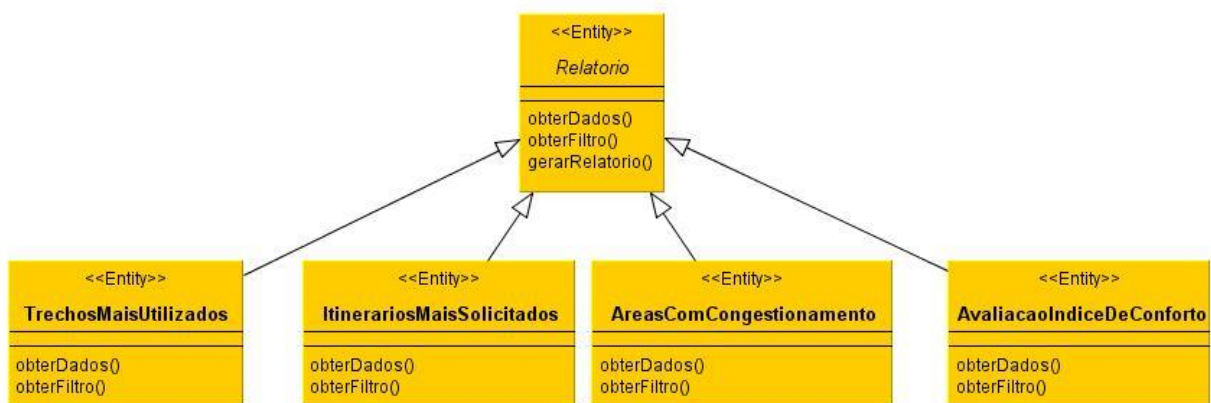


Fonte: Elaborado pelo autor.

Para utilizar o *design pattern Template Method*, na classe abstrata Relatório foram incluídos os métodos abstratos obterDados e obterFiltro (ver Figura 19). O método gerarRelatório possui a chamada para geração de relatório, não sendo necessário cada subclasse reimplementar esse método. Cada subclasse possui sua implementação própria dos métodos obterDados e obterFiltro, assim elas são

responsáveis apenas por manter os métodos de acordo com seu objetivo. A lógica da geração do relatório em si, que é igual para todos tipos de relatório, fica apenas no método gerarRelatorio da classe Relatorio.

Figura 19: Diagrama refactorado utilizando *design pattern Template Method*



Fonte: Elaborado pelo autor.

Abaixo segue um exemplo da implementação do *design pattern* (na linguagem Java):

```

public abstract class Relatorio {
    abstract Dados obterDados();
    abstract Filtro obterFiltro();

    //Template Method
    public final RelatorioFinal gerarRelatorio() {
        Dados dados = obterDados();
        Filtro filtro = obterFiltro();
        return GeradorDeRelatorio.gerarRelatorio(dados, filtro);
    }
}

public class TrechosMaisUtilizados extends Relatorio {
    Dados obterDados() {
        //lógica para obtenção dos dados corretos
        return dados;
    }

    Filtro obterFiltro() {
        //lógica para obtenção do filtro desejado
        return filtro;
    }
}
  
```

```
public class ItinerariosMaisSolicitados extends Relatorio {
    Dados obterDados() {
        //lógica para obtenção dos dados corretos
        return dados;
    }

    Filtro obterFiltro(){
        //lógica para obtenção do filtro desejado
        return filtro;
    }
}

public class AreasComCongestionamento extends Relatorio {
    Dados obterDados() {
        //lógica para obtenção dos dados corretos
        return dados;
    }

    Filtro obterFiltro(){
        //lógica para obtenção do filtro desejado
        return filtro;
    }
}

public class AvaliacaoIndiceDeConforto extends Relatorio {
    Dados obterDados() {
        //lógica para obtenção dos dados corretos
        return dados;
    }

    Filtro obterFiltro(){
        //lógica para obtenção do filtro desejado
        return filtro;
    }
}
```

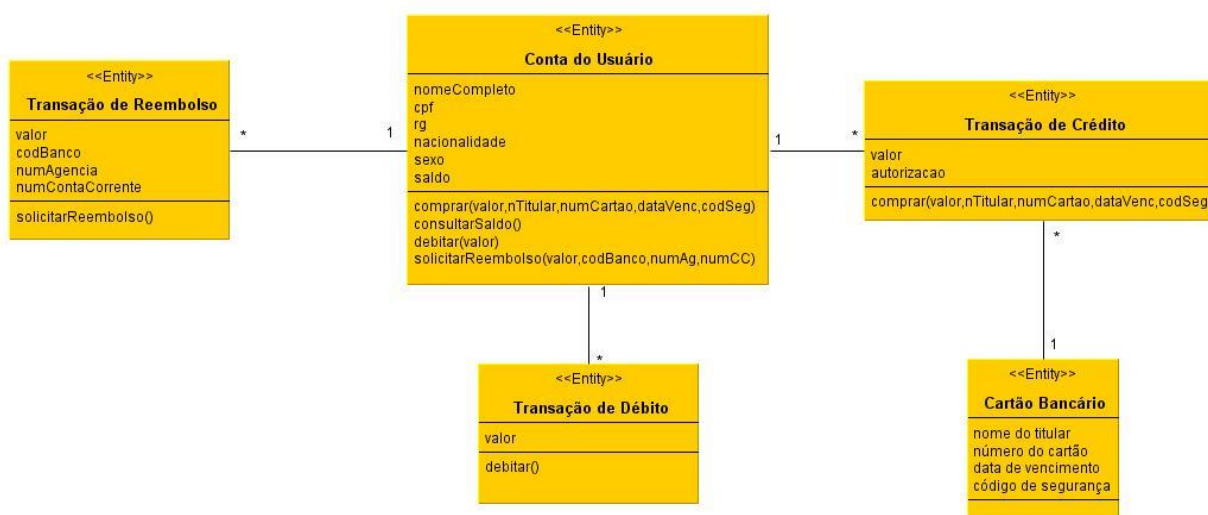
3.4.6 Visitor

A aplicação deste *pattern* foi analisada na modelagem das funcionalidades Comprar Créditos: compra de crédito por parte do passageiro para utilização no transporte público da cidade, Reembolsar Créditos: reembolso de crédito não utilizado pelo passageiro e Usar Créditos: uso dos créditos por parte do usuário.

Para demonstração do *design pattern Visitor*, é considerado que, para comemorar o aniversário da cidade, a prefeitura decide não cobrar pelas passagens de transporte dos usuários (seja qual for a modalidade de transporte) durante um dia.

Na versão sem implementação do *design pattern Visitor* (ver Figura 20), o diagrama possui as classes Transação de crédito, débito e reembolso, todas ligadas com a classe conta do usuário (ao chegar uma requisição na conta do usuário, esta é responsável por criar uma transação de débito, crédito ou reembolso). Uma limitação deste diagrama é que não é possível interferir facilmente nas transações para mudar o comportamento das mesmas sem alterar seus códigos.

Figura 20: Diagrama de classes das funcionalidades de Comprar Créditos, Reembolsar Créditos e Usar Créditos

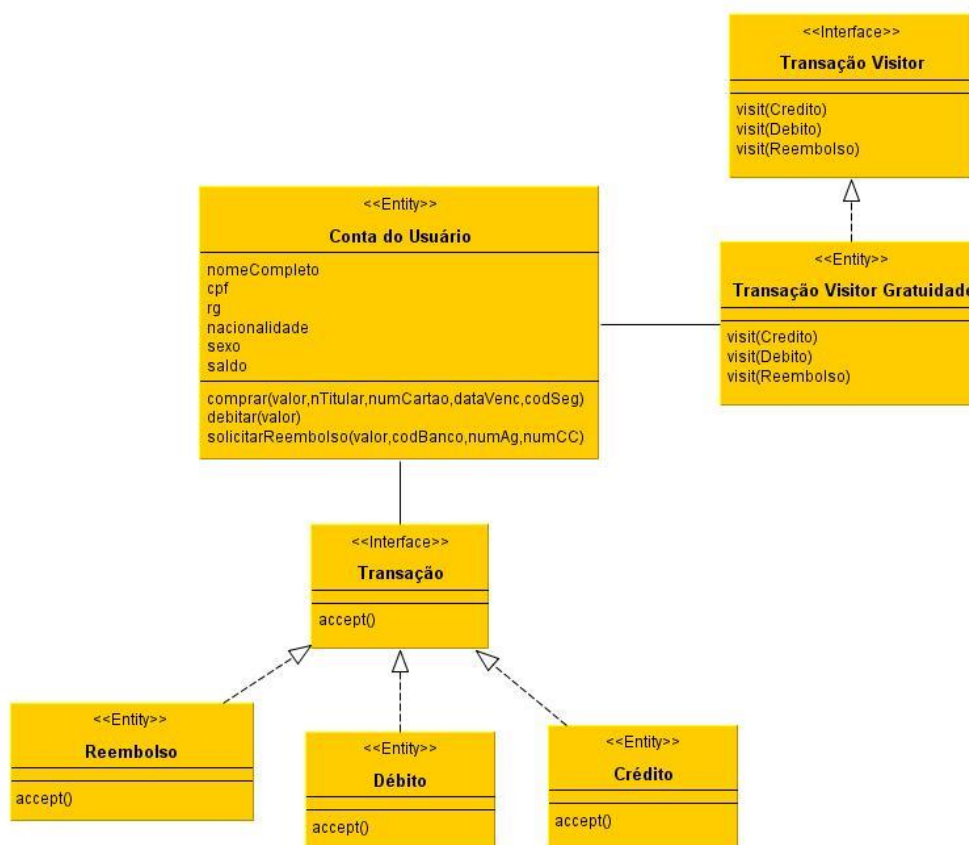


Fonte: Elaborado pelo autor.

Na versão com o *design pattern Visitor* (ver Figura 21), o diagrama foi alterado para ter uma interface transação e os três tipos de transação (reembolso, débito e crédito) implementam essa interface. A interface possui um método `accept()`, esse método é responsável por receber, via parâmetro, o *Visitor* desejado. Também foram adicionadas outras duas classes: A interface *Transação Visitor* e sua implementadora *Transação Visitor Gratuidade*. Caso seja necessário mais algum tipo de transação, além de gratuidade, será necessário apenas a nova transação implementar *Transação Visitor*, dessa forma o método `accept()` das classes de *Transação* poderá receber o novo tipo de *Visitor* criado.

No dia do aniversário da cidade, todos os tipos de transações saberão, de acordo com a implementação do *Visitor* *Transação Visitor Gratuidade*, como não realizar a cobrança de passagens e, quando passar a data comemorativa, é necessário apenas não utilizar o *Transação Visitor Gratuidade*, para o sistema voltar a cobrar normalmente.

Figura 21: Diagrama refatorado utilizando *design pattern Visitor*



Fonte: Elaborado pelo autor.

Abaixo segue um exemplo da implementação do *design pattern* (na linguagem Java):

```
public interface Transacao {
    public void accept(TransacaoVisitor Visitor);
}

public class Reembolso implements Transacao {

    @override
    public void accept(TransacaoVisitor Visitor){
        Visitor.visit(this)
    }
}

public class Debito implements Transacao {

    @override
    public void accept(TransacaoVisitor Visitor){
        Visitor.visit(this)
    }
}

public class Credito implements Transacao {

    @override
    public void accept(TransacaoVisitor Visitor){
        Visitor.visit(this)
    }
}

public interface TransacaoVisitor {
    public void visit(Credito credito);
    public void visit(Debito debito);
    public void visit(Reembolso reembolso);
}

public class TransacaoVisitorGratuidade implements TransacaoVisitor
{
    @override
    public void visit(Credito credito) {
        //algoritmo para quando existir gratuidade no sistema
    }

    @override
    public void visit(Debito debito) {
        //algoritmo para quando existir gratuidade no sistema
    }

    @override
    public void visit(Reembolso reembolso) {
        //algoritmo para quando existir gratuidade no sistema
    }
}
```

```
public class ContaDoUsuario {
    public void comprar() {
        Transacao credito = new Credito();
        credito.accept(new TransacaoVisitorGratuidade);
    }

    public void debitar() {
        Transacao debito = new Debito();
        debito.accept(new TransacaoVisitorGratuidade);
    }

    public void solicitarReembolso() {
        Transacao reembolso = new Reembolso();
        reembolso.accept(new TransacaoVisitorGratuidade);
    }
}
```

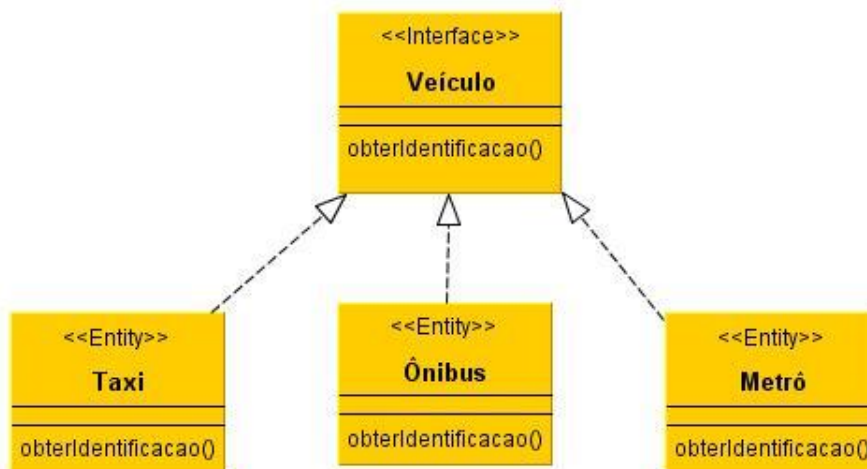
3.4.7 Decorator

Para demonstração do *design pattern Decorator*, foi considerado o cenário:

A interface Veículo possui o método obterIdentificacao(), responsável por retornar o código de identificação do veículo específico. Porém uma nova regulamentação da prefeitura fez uma mudança na identificação de todos os veículos, da cidade. A nova identificação de um veículo é extraída de um algoritmo criado para gerar a identificação.

O atual diagrama de classes da funcionalidade de obter identificação do veículo (Figura 22), possui as classes Taxi, Ônibus e Metrô, todas implementando a interface Veículo.

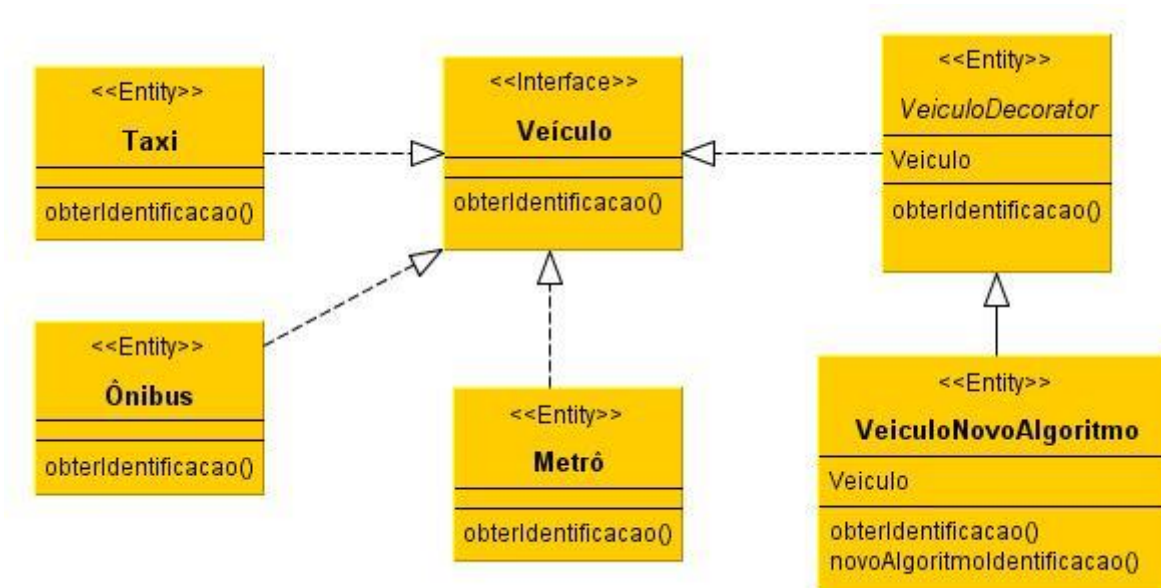
Figura 22: Diagrama de classes da funcionalidade de obter identificação do veículo



Fonte: Elaborado pelo autor.

Com a utilização do *pattern Decorator* (ver Figura 23), foram incluídas duas novas classes ao diagrama: A classe abstrata *VeiculoDecorator* e a classe *VeiculoNovoAlgoritmo*. A classe *VeiculoDecorator* armazena o veículo cuja identificação será extraída, enquanto que a classe *VeiculoNovoAlgoritmo* é a responsável por conhecer o novo algoritmo para geração da identificação do veículo. Dessa forma não é necessário alterar o método `obterIdentificacao()` das classes *Metrô*, *Ônibus* e *Trem*, é necessário apenas criar um objeto *VeiculoNovoAlgoritmo* passando o veículo desejado como parâmetro no construtor.

Figura 23: Diagrama refatorado utilizando *design pattern Decorator*



Fonte: Elaborado pelo autor.

Abaixo segue um exemplo da implementação do *design pattern* (na linguagem Java):

```

public interface Veiculo {
    public void obterIdentificacao();
}

public class Ônibus implements Veiculo {
    @override
    public void obterIdentificacao() {
        //Algoritmo para obter identificação do onibus
    }
}

public class Taxi implements Veiculo {
    @override
    public void obterIdentificacao() {
        //Algoritmo para obter identificação do taxi
    }
}

public class Metrô implements Veiculo {
    @override
    public void obterIdentificacao() {
        //Algoritmo para obter identificação do metrô
    }
}
  
```

```
public abstract class VeiculoDecorator implements Veiculo {  
    protected Veiculo decoratedVeiculo;  
  
    public VeiculoDecorator(Veiculo decoratedVeiculo) {  
        this.decoratedVeiculo = decoratedVeiculo  
    }  
  
    public void obterIdentificacao() {  
        decoratedVeiculo.obterIdentificacao();  
    }  
}  
  
public class VeiculoNovoAlgoritmo extends VeiculoDecorator() {  
  
    public VeiculoNovoAlgoritmo(Veiculo veiculo) {  
        super(veiculo);  
    }  
  
    @override  
    public void obterIdentificacao() {  
        antigaIdentificacao =  
decoratedVeiculo.obterIdentificacao();  
        novaIdentificacao =  
        novoAlgoritmoIdentificacao(antigaIdentificacao);  
    }  
  
    private String novoAlgoritmoIdentificacao(String old) {  
        //novo algoritmo para identificação veiculo  
    }  
}
```

3.5 Considerações do capítulo

Neste capítulo foi apresentado o projeto de software utilizado para auxiliar a análise e aplicação dos *design patterns*: *Composite*, *Factory Method*, *Abstract Factory*, *Flyweight*, *Template Method*, *Visitor* e *Decorator*. Foi realizada a análise antes e após o uso dos *design patterns*, além de um código java exemplo de como implementar o *design pattern*.

4. ANÁLISE DOS RESULTADOS

Após realizar a análise do projeto de software antes e depois da aplicação dos design patterns, é analisado o resultado da aplicação de cada design pattern com base nos atributos de qualidade: *Abstract Factory*, *Composite*, *Flyweight*, *Visitor*, *Decorator*, *Template Method* e *Factory Method*.

4.1 Avaliação dos *Design patterns*

A fim de investigar o impacto dos *design patterns* sobre a qualidade geral do sistema, foi elaborado um questionário. Este questionário teve como objetivo auxiliar na análise dos *design patterns* do GOF para, no final da análise, ser proposta uma lista de recomendações para o uso adequado dos *patterns* em projetos cuja equipe é formada por desenvolvedores com pouca experiência profissional na área de desenvolvimento de *software* e que nunca trabalharam com os *design patterns* GOF.

Participou desta avaliação um desenvolvedor com experiência de 5 anos como desenvolvedor, sendo que 2 anos foram participando de projetos que utilizam extensivamente *design patterns*.

A lista de recomendações para o uso adequado dos *patterns* leva em consideração as respostas do questionário e os resultados da pesquisa de Khomh & Gueheneuc (tabela 2).

Os *design patterns* analisados no questionário foram: *Abstract Factory*, *Composite*, *Flyweight*, *Visitor*, *Decorator*, *Template Method* e *Factory Method*

Os atributos de qualidade analisados foram:

- Reusabilidade: O grau em que um pedaço de um *design* pode ser reusado em outro componente / módulo no projeto com pouca, ou nenhuma, modificação.
- Expansibilidade: O grau em que um *design* pode ser expandido.

- Simplicidade: O grau em que um *design* de um sistema pode ser facilmente entendido
- Compreensibilidade: O grau em que o código fonte de um sistema pode ser facilmente entendido
- Apreensibilidade: O grau em que o código fonte de um sistema pode ser facilmente aprendido
- Modularidade: O grau em que o *software* é composto por componentes, permutáveis separados, cada um dos quais realiza uma função e contém todos os elementos necessários para conseguir isso.

Para a avaliação do impacto de cada *design pattern* no projeto foi usada a seguinte escala:

Tabela 4: Classificação da avaliação dos Design patterns

Letra	Impacto
A	Positivo
B	Neutro
C	Negativo

Positivo: Significa que o *design pattern* impacta positivamente no atributo de qualidade.

Neutro: Significa que o *design pattern* não possui impacto relevante no o atributo de qualidade.

Negativo: Significa que o *design pattern* impacta negativamente no atributo de qualidade.

4.2 Análise dos *Design patterns*

Na tabela 5 é apresentado o resultado do questionário aplicado.

Tabela 5: Resultados do questionário aplicado

Atributo de Qualidade	<i>Design pattern</i>						
	DP1	DP2	DP3	DP4	DP5	DP6	DP7
Reusabilidade	A	A	A	B	C	C	A
Expansibilidade	A	A	B	B	A	A	A
Simplicidade	A	B	B	C	B	C	B
Compreensibilidade	A	B	C	C	C	C	B
Apreensibilidade	A	B	C	C	B	B	B
Modularidade	A	A	A	B	B	C	A

Legenda:

DP1 = *Design pattern Composite*

DP2 = *Design pattern Factory Method*

DP3 = *Design pattern Abstract Factory*

DP4 = *Design pattern Flyweight*

DP5 = *Design pattern Visitor*

DP6 = *Design pattern Decorator*

DP7 = *Design pattern Template Method*

De acordo com os resultados apresentados na tabela 3 verificou-se que:

Considerando os sete *design patterns* estudados em 6 atributos de qualidade, 39% dos atributos de qualidade foram impactados positivamente pelos *design patterns*, 35% dos atributos de qualidade não tiveram impactos relevantes e 26% dos atributos de qualidade foram impactados negativamente pelos *design patterns*.

O resultado do questionário aplicado se aproximou da conclusão do estudo realizado por Khomh & Gueheneuc:

- Composite: Tanto neste trabalho quanto no estudo de Khomh & Gueheneuc, o design pattern Composite demonstrou impactar positivamente nos atributos de qualidade estudados.
- Factory Method: No estudo de Khomh & Gueheneuc o design pattern Factory Method impactou negativamente no atributo compreensibilidade, neste trabalho o impacto resultante foi neutro. Os atributos de qualidade Expansibilidade e Reusabilidade tiveram o mesmo resultado.
- Abstract Factory: No estudo de Khomh & Gueheneuc o design pattern Abstract Factory impactou positivamente no atributo Expansibilidade, neste trabalho o impacto resultante foi neutro. Os atributos de qualidade Reusabilidade e Compreensibilidade tiveram o mesmo resultado.
- Flyweight: No estudo de Khomh & Gueheneuc o design pattern Flyweight impactou negativamente nos atributos Reusabilidade e Expansibilidade, neste trabalho o impacto resultante foi neutro. O atributo de qualidade Compreensibilidade teve o mesmo resultado.
- Visitor e Decorator: Os atributos de qualidade Reusabilidade, Expansibilidade e Compreensibilidade tiveram o mesmo resultado neste trabalho e no estudo de Khomh & Gueheneuc.
- Template Method: No estudo de Khomh & Gueheneuc o design pattern Template Method impactou negativamente no atributo Compreensibilidade, neste trabalho o impacto resultante foi neutro. Os atributos de qualidade Reusabilidade e Expansibilidade tiveram o mesmo resultado.

Segue estudo detalhado de cada *design pattern*, realizado após análise do resultado do questionário respondido pelo desenvolvedor experiente:

4.2.1 Análise *Design pattern Composite*

O padrão *Composite* permite que seus usuários tratem da mesma forma objetos individuais e suas composições: No exemplo da figura 11, as classes *Regiao* e *RegiaoAdministrativa* são do tipo “*Criticidade*” e dessa forma possuem o método *getCriticidade()*, ambas as classes são tratadas de forma uniforme por quem as usa (sem mesmo precisar distinguir se é uma *Regiao* ou uma *RegiaoAdministrativa*). Isso tudo faz com que o atributo de qualidade “*Simplicidade*” e “*Compreensibilidade*” seja positivo. A reusabilidade desse *pattern* é positiva pois, facilmente outros módulos ou componentes podem utilizar/chamar esse *design*, sem precisar efetuar modificações de código. A expansibilidade também é positiva pois facilmente é possível incrementar novas funcionalidades no *design*, ou apenas melhorar os algoritmos já implementados.

4.2.2 Análise *Design pattern Factory Method*

A reusabilidade desse padrão é positiva, uma vez que ele pode ser utilizado em qualquer parte do projeto de modo que não precise de modificações (a criação de novos objetos é a mesma, independente do módulo / componente do projeto). A expansibilidade também é positiva, no exemplo da Figura 13, caso surja a necessidade de criação de um novo tipo de veículo, basta criar o *Factory* do novo veículo. Porém neste ponto a simplicidade, compreensibilidade e apreensibilidade não são positivas, pois a necessidade de criar um *Factory* para cada tipo de veículo aumenta a complexidade do código, fazendo com que a curva de aprendizado seja maior e a manutenção mais complexa (Quanto mais classes são necessárias em determinado *design*, maiores as chances de problemas no futuro). A modularidade é positiva pois, o código de criação de cada tipo de veículo fica restrito a sua classe específica.

4.2.3 Análise *Design pattern Abstract Factory*

O trecho de *design* que utiliza o padrão *Abstract Factory* é facilmente reutilizável em outras partes do projeto e é bem modularizado – cada algoritmo é implementado dentro da sua respectiva classe, por isso a Reusabilidade e Modularidade são positivas. Porém por ser um *design* mais complexo, a Compreensibilidade e Apreensibilidade são pontos negativos para o projeto, ainda mais para equipes novatas que não possuem experiências com o uso de *design patterns*. A expansibilidade é neutra neste *pattern*, ela será fácil caso haja adição ou remoção de funcionalidades que façam parte do modelo de negócios utilizado no *pattern*. No exemplo da figura 15, a expansibilidade será fácil se for incluso mais um perfil de utilizador, por exemplo, de crianças.

4.2.4 Análise *Design pattern Flyweight*

A aplicação do *design pattern Flyweight* é útil para economizar memória (não criar muitos objetos do mesmo tipo, sendo que estes poderiam ser armazenados e todas as vezes que precisassem ser utilizados, basta extraí-los e utilizá-los). Como este *design pattern* possui um uso muito específico, as desvantagens podem ser grandes com seu uso, a menos que a equipe esteja bem consciente que realmente é necessário utilizar este *pattern*. A apreensibilidade, compreensibilidade e simplicidade são negativas pois o *pattern* aumenta a complexidade do sistema.

Este padrão deve ser utilizado com cuidado e a situação em que será utilizado deve ser bem estudado, pois muitas vezes este *pattern* é utilizado junto de outros *patterns*, aumentando bastante sua complexidade.

4.2.5 Análise *Design pattern Visitor*

A reusabilidade desse padrão em outros módulos/componentes é negativa, pois é um padrão muito específico para determinada funcionalidade, podendo ser difícil sua reutilização. A compreensibilidade também é negativa, uma vez que a aplicação deste *pattern* dificulta a leitura e a compreensão do código / diagrama. A

expansibilidade é positiva pois, é muito fácil criar diferentes *Visitor* e reutilizá-los quando for necessário. Em contra partida, essa facilidade de criar muitos *Visitors* acaba fazendo com que a modularidade, apreensibilidade e simplicidade sejam apenas neutras.

4.2.6 Análise *Design pattern Decorator*

O *design pattern Decorator* é bastante expansível, permitindo que novos *Decorators* sejam criados quando necessário para que sejam utilizados no sistema. Porém isso faz com que a sua modularidade seja negativa, pois se muitos *Decorators* forem criados, ao invés de o código antigo do sistema seja atualizado, o sistema terá muito código legado que não é mais utilizável, deixando sua modularidade bastante negativa e atrapalhando a manutenção do sistema. Isso também faz com que a compreensibilidade e simplicidade sejam negativas. A reusabilidade é negativa pois, assim como o *Visitor*, o *Decorator* é um padrão muito específico para determinada funcionalidade, podendo ser difícil sua reutilização em outros componentes/módulos.

4.2.7 Análise *Design pattern Template Method*

O *design pattern Template Method* possui modularidade positiva - uma vez que as classes de domínio são responsáveis pelos seus algoritmos específicos, e a classe abstrata genérica, possui a implementação genérica, utilizando os algoritmos implementados nas classes de domínio. No exemplo da figura 19, as classes que representam os relatórios específicos, implementam os métodos obterDados e obterFiltros, e a classe abstrata Relatório, utiliza esses métodos para gerar o relatório desejado. Isso faz com que, além da modularidade, a expansibilidade seja positiva, é bastante simples aumentar o número de funcionalidades utilizando este padrão, no exemplo da figura 19, é simples criar novos tipos de relatório. A reusabilidade também é positiva, pois esse padrão permite que em qualquer módulo / componente seja possível utilizar o trecho de *design*.

4.3 Lista de Recomendações

Abaixo é listado as recomendações para o uso dos *design patterns Composite, Factory Method, Abstract Factory, Flyweight, Template Method, Visitor e Decorator*.

- 1- *Design Pattern Composite*: pode ser considerado simples e fácil o suficiente para ser utilizado por equipes iniciantes, sem impactos negativos no projeto, ou que prejudique sua manutenção no futuro.
- 2- *Design Pattern Factory Method*: pode ser considerado simples e fácil para ser utilizado por equipes iniciantes, sem impactos negativos no projeto.
- 3- *Design Pattern Abstract Factory*: este pattern é mais simples e fácil de ser utilizado por equipes iniciantes do que o *Design Pattern Abstract Factory* – porém deve ser levado em consideração que cada padrão tem sua aplicabilidade específica.
- 4- *Design Pattern Flyweight*: deve ser utilizado com cuidado e a situação em que será utilizado deve ser bem estudado, pois muitas vezes este *pattern* é utilizado junto de outros *patterns*, aumentando bastante sua complexidade.
- 5- *Design Pattern Visitor*: Em equipes iniciantes, é recomendável tomar cuidado com esse padrão, principalmente para ele não ser utilizado quando não é necessário (podendo causar grandes problemas a utilização de um *Visitor* no momento errado, de acordo com a estratégia da equipe).
- 6- *Design Pattern Decorator*: Em equipes iniciantes é recomendável utilizar com cuidado este padrão, principalmente para não deixar códigos legados antigos, que não são mais utilizados, pelo sistema, fazendo com que sua manutenção seja muito complexa.
- 7- *Design Pattern Template Method*: A utilização deste padrão pode ser feita por equipes novatas, uma vez que não existem pontos que impactam negativamente no projeto, além de ser um padrão de uso simples e que não aumenta a complexidade do sistema (ao contrário, traz melhorias, como boa modularidade, que faz com que o sistema se torne mais simples e de fácil manutenção).

4.4 Considerações do capítulo

Após a análise realizada com base no questionário respondido, foram listadas 7 recomendações para o uso saudável de design patterns no projeto de software para tentar minimizar ao máximo possíveis problemas que possam surgir com o uso desses patterns.

5. CONSIDERAÇÕES FINAIS

Neste trabalho foram apresentados alguns conceitos referentes à projeto de software, atributos de qualidade e design patterns. Além disso, foram apresentados artigos, que também estudaram o impacto de design patterns em projetos de software.

5.1 Contribuições do Trabalho

Com este estudo foi mostrado que a maioria dos *design patterns* impactam positivamente, ou de forma neutra, nos atributos de qualidade. No total, 39% dos atributos de qualidade tiveram impacto positivo com o uso dos *design patterns* e 35% impacto neutro. O restante, 26% dos atributos de qualidade, foram impactados negativamente pelos *design patterns*, principalmente os *patterns Flyweight* e *Visitor*, dessa forma uma equipe de desenvolvedores inciantes deve tomar muito cuidado ao utilizar esses *design patterns*, para não prejudicar a futura manutenção do *software*, por exemplo, dificultando a compreensibilidade do código que utiliza esses *design patterns*.

De forma geral, o atributo de qualidade “Expansibilidade” foi o que teve maior impacto positivo (cinco *design patterns* impactaram positivamente este atributo de qualidade), desta forma é possível incluir facilmente novas funcionalidades em determinado componente sem que haja impactos negativos ao componente/projeto (por exemplo, não existe a necessidade de ter de refazer outras funcionalidades já implementadas). Os atributos de qualidade “Modularidade” e “Reusabilidade” também são impactados positivamente pelos *design patterns*, resultando em aumento da modularidade dos componentes e a maior reusabilidade de um design de um componente em outro componente.

O atributo de qualidade que teve maior impacto negativo foi “Compreensibilidade” (quatro *design patterns* impactaram negativamente este atributo de qualidade), desta forma o código fonte do componente que utiliza esses *design patterns* pode se tornar mais difícil de ser entendido pelos desenvolvedores.

Os atributos de qualidade “Simplicidade” e “Apreensibilidade” foram impactados de forma neutra por quatro *design patterns*, desta forma esses *design patterns* não possuem impacto relevante sobre estes dois atributos de qualidade.

5.2 Trabalhos Futuros

Como trabalho futuro é desejável que o estudo destes *design patterns* sobre os atributos de qualidade seja realizado em diferentes tipos de projetos reais, concluindo assim quais impactos os *design patterns* causam em cada tipo de projeto. Também é desejável realizar entrevistas com mais desenvolvedores, incluindo desenvolvedores iniciantes com pouca experiência com os *design patterns*. Também é desejável, em trabalho futuro, aumentar o número de *design patterns* e dos atributos de qualidade estudados, para aumentar a lista de recomendações com mais design patterns.

REFERÊNCIAS

- ALI, M. & ELISH, M. O., 2013. A Comparative Literature Survey of Design Patterns Impact on Software Quality. *Information Science and Applications (ICISA), 2013 International Conference on*, June, pp. 1 - 7.
- FREEMAN, E. & FREEMAN, E., 2004. *Head First Design Patterns*. s.l.:s.n.
- GAMMA, E., HELM, R., JOHNSON, R. & VLISSIDES, J., 1994. *Design patterns Elements of Reusable Object-Oriented Software*. s.l.:s.n.
- ISO/IEC, 2000. *9126-1: Software Product Quality – Part 1: Quality model*. s.l.:s.n.
- KHOMH, F. & GUEHENEUC, Y.-G., 2008. Do Design Patterns Impact Software Quality Positively?. *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, 1-4 April , pp. 274 - 278.
- MAGHAWRY, N. E. & DAWOOD, A. R., 2010. Aspect Oriented GoF Design. *Informatics and Systems (INFOS), 2010 The 7th International Conference on*, 28-30 March , pp. 1-7.
- msdn, 2014. *Chapter 16: Quality Attributes*. [Online]
Available at: <http://msdn.microsoft.com/en-us/library/ee658094.aspx>
[Acesso em 18 10 2014].
- PRESSMAN, R. S. & MAXIM, B. R., 2001. *Software Engineering A Practitioner's Approach*. Eighth ed. s.l.:s.n.
- ZHANG, C. & BUDGEN, D., 2012. What Do We Know about the Effectiveness of Software Design Patterns?. *Software Engineering, IEEE Transactions on (Volume:38 , Issue: 5)*, September, pp. 1213 - 1231.

APÊNDICE – QUESTIONÁRIO

A fim de investigar o impacto dos *design patterns* sobre a qualidade geral do sistema, este questionário tem como objetivo auxiliar na análise dos *design patterns* do GOF. Considerando que os padrões são usados de forma adequada em um projeto de *software* para resolver os problemas em que são propostos, por favor, circule a letra correspondente à sua resposta às questões 1 a 7.

Os atributos de qualidade são:

Reusabilidade: O grau em que um pedaço de um *design* pode ser reusado em outro local

Expansibilidade: O grau em que um *design* pode ser estendido

Simplicidade: O grau em que um *design* de um sistema pode ser facilmente entendido

Compreensibilidade: O grau em que o código fonte de um sistema pode ser facilmente entendido

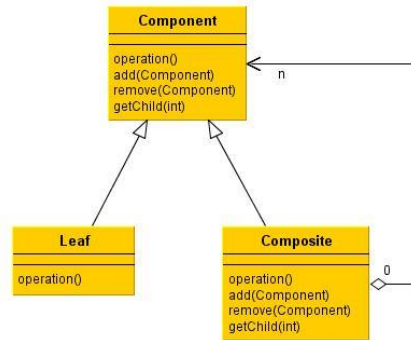
Apreensibilidade: O grau em que o código fonte de um sistema pode ser facilmente aprendido

Modularidade: O grau em que o *software* é composto por componentes, permutáveis separados, cada um dos quais realiza uma função e contém todos os elementos necessários para conseguir isso.

Para a avaliação do impacto de cada *design pattern* no projeto, deve ser utilizada a seguinte escala:

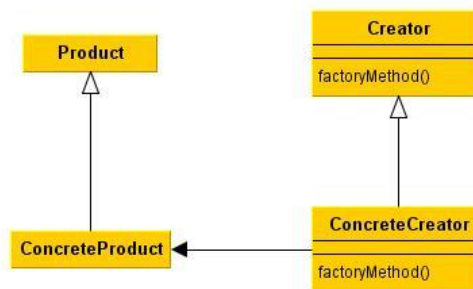
Letra	Impacto
A	Positivo
B	Neutro
C	Negativo

1- Em sua opinião, qual impacto do design pattern *Composite* em um projeto de software:



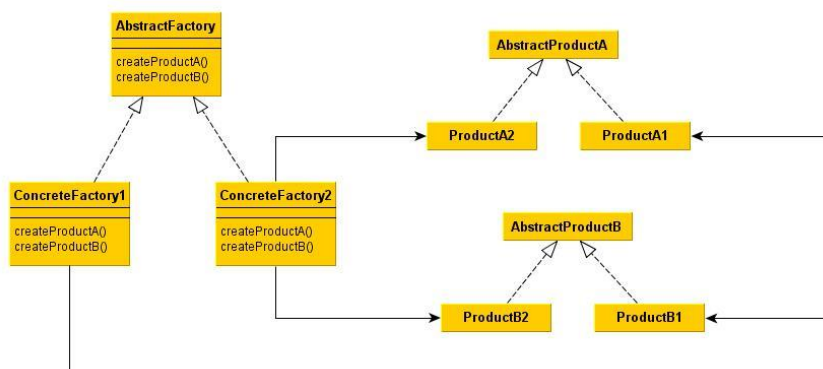
<i>Composite</i>			
Reusabilidade	A	B	C
Expansibilidade	A	B	C
Simplicidade	A	B	C
Compreensibilidade	A	B	C
Apreensibilidade	A	B	C
Modularidade	A	B	C

2- Em sua opinião, qual impacto do design pattern *Factory Method* em um projeto de software:



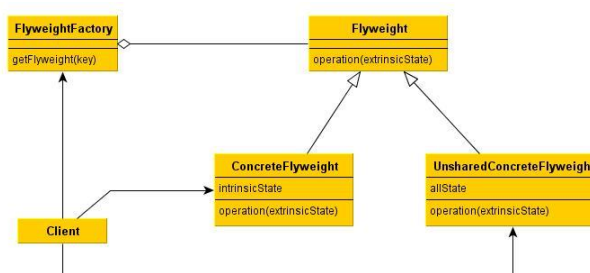
<i>Factory Method</i>			
Reusabilidade	A	B	C
Expansibilidade	A	B	C
Simplicidade	A	B	C
Compreensibilidade	A	B	C
Apreensibilidade	A	B	C
Modularidade	A	B	C

3- Em sua opinião, qual impacto do design pattern Abstract Factory em um projeto de software:



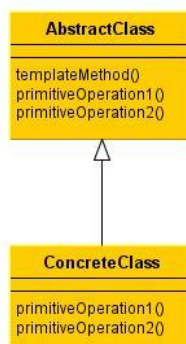
<i>Abstract Factory</i>			
Reusabilidade	A	B	C
Expansibilidade	A	B	C
Simplicidade	A	B	C
Compreensibilidade	A	B	C
Apreensibilidade	A	B	C
Modularidade	A	B	C

4- Em sua opinião, qual impacto do design pattern Flyweight em um projeto de software:



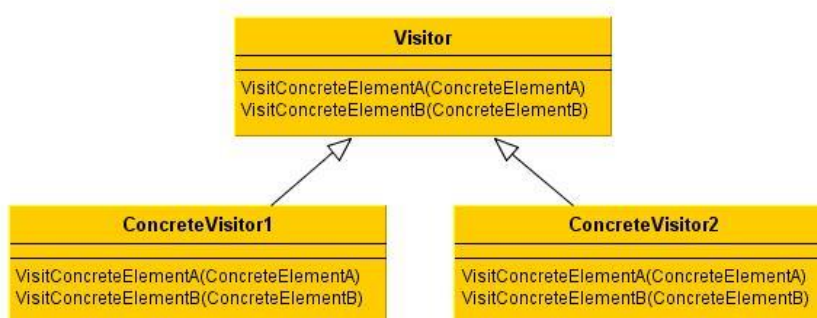
<i>Flyweight</i>			
Reusabilidade	A	B	C
Expansibilidade	A	B	C
Simplicidade	A	B	C
Compreensibilidade	A	B	C
Apreensibilidade	A	B	C
Modularidade	A	B	C

- 5- *Em sua opinião, qual impacto do design pattern Template Method em um projeto de software:*



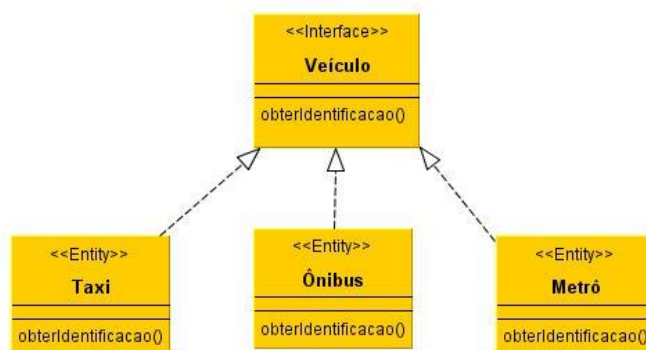
<i>Template Method</i>			
Reusabilidade	A	B	C
Expansibilidade	A	B	C
Simplicidade	A	B	C
Compreensibilidade	A	B	C
Apreensibilidade	A	B	C
Modularidade	A	B	C

- 6- *Em sua opinião, qual impacto do design pattern Visitor em um projeto de software:*



<i>Visitor</i>			
Reusabilidade	A	B	C
Expansibilidade	A	B	C
Simplicidade	A	B	C
Compreensibilidade	A	B	C
Apreensibilidade	A	B	C
Modularidade	A	B	C

7- Em sua opinião, qual impacto do design pattern Decorator em um projeto de software:



<i>Decorator</i>			
Reusabilidade	A	B	C
Expansibilidade	A	B	C
Simplicidade	A	B	C
Compreensibilidade	A	B	C
Apreensibilidade	A	B	C
Modularidade	A	B	C